**POCO C++库学习和分析 -- 内存管理 (三)**

        看完Poco库中的智能指针，基本上Poco中的内存管理已经快结束了。其他的部分都是些边边角角的东西，非常的简单。下面一一介绍。

**4. AutoReleasePool**

        AutoReleasePool类的出现也同样是为了解决用户动态分配对象的释放问题，但同智能指针AutoPtr和SharedPtr通过把堆上的对象包装成栈对象，再通过引用计数在类的析构函数中实现自动删除对象的解决方案不同是，其策略为构造一个容器，用来存储动态对象的指针，在AutoReleasePool析构函数中统一释放。

        这个过程和java语言中的垃圾收集机制是类似的，只不过AutoReleasePool实现的非常简单，在AutoReleasePool销毁时释放资源，而在java语言中会连续不断的定时检查并释放闲置资源。当然为了实现这个过程，AutoReleasePool对所释放的类是有要求的，释放的类必须实现release()接口。下面通过一个例子来说明问题：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. #include "Poco/AutoReleasePool.h"
2. **using** Poco::AutoReleasePool;
3. **class** C
4. {
5. **public**:
6. C()
7. {}
8. **void** release()
9. {
10. **delete** **this**;
11. }
12. };
14. **int** main(**int** argc, **char**\*\* argv)
15. {
16. AutoReleasePool<C> pool;
17. C\* pC = **new** C;
18. pool.add(pC);
19. pC = **new** C;
20. pool.add(pC);
21. **return** 0;
22. }
23. // all C's deleted

        其类图如下：



        在图中可以看出，AutoReleasePool实际上就是原生指针的一个容器，在其内部定义为：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. std::list<C\*> ObjectList \_list

       需要注意的是，如果同时使用AutoReleasePool和AutoPtr对指针进行管理时，应该如此实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. AutoReleasePool<C> arp;
2. AutoPtr<C> ptr = **new** C;
3. ...
4. arp.add(ptr.duplicate());

       很明显此刻AutoReleasePool和AutoPtr对对象应该共享所有权。

**5. 动态工厂模板（DynamicFactory Class Template）**

       Poco中实现了一个动态工厂的模板，支持通过类名来创建类。其实现技术和前面的文章"[Foundation库SharedLibrary模块分析](http://blog.csdn.net/arau_sh/article/details/8569638)"中介绍的类似。

       动态工厂类DynamicFactory是抽象工厂类AbstractFactory的容器。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. **template** <**class** Base>
2. **class** DynamicFactory
3. /// A factory that creates objects by class name.
4. {
5. ....
6. std::map<std::string, AbstractFactory\*> FactoryMap \_map;
7. }

       AbstractFactory其实是模板类AbstractInstantiator<Base>的代称，是典型的工厂模式，用来实现具体类的创建，每个类工厂AbstractFactory对应一个类的创建。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. **template** <**class** Base>
2. **class** DynamicFactory
3. /// A factory that creates objects by class name.
4. {
5. **public**:
6. **typedef** AbstractInstantiator<Base> AbstractFactory;
7. ....
8. }

       而一个动态工厂DynamicFactory则可负责从一个基类继承的一组继承类的创建工作。如果有多个基类，用户必须创建多个动态工厂DynamicFactory去实现类的创建。

       同样为了完成通过类名来创建类的工作，DynamicFactory必须对类工厂进行注册，以便使编译器在编译时，建立类名和类的对应关系。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. **void** registerClass(**const** std::string& className)
2. /// Registers the instantiator for the given class with the DynamicFactory.
3. /// The DynamicFactory takes ownership of the instantiator and deletes
4. /// it when it's no longer used.
5. /// If the class has already been registered, an ExistsException is thrown
6. /// and the instantiator is deleted.
7. {
8. registerClass(className, **new** Instantiator<C, Base>);
9. }
11. **void** registerClass(**const** std::string& className, AbstractFactory\* pAbstractFactory)
12. /// Registers the instantiator for the given class with the DynamicFactory.
13. /// The DynamicFactory takes ownership of the instantiator and deletes
14. /// it when it's no longer used.
15. /// If the class has already been registered, an ExistsException is thrown
16. /// and the instantiator is deleted.
17. {
18. poco\_check\_ptr (pAbstractFactory);
20. FastMutex::ScopedLock lock(\_mutex);
22. std::auto\_ptr<AbstractFactory> ptr(pAbstractFactory);
23. **typename** FactoryMap::iterator it = \_map.find(className);
24. **if** (it == \_map.end())
25. \_map[className] = ptr.release();
26. **else**
27. **throw** ExistsException(className);
28. }

       接下来再来看DynamicFactory类中的创建工作：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. **template** <**class** Base>
2. **class** DynamicFactory
3. /// A factory that creates objects by class name.
4. {
5. Base\* createInstance(**const** std::string& className) **const**
6. /// Creates a new instance of the class with the given name.
7. /// The class must have been registered with registerClass.
8. /// If the class name is unknown, a NotFoundException is thrown.
9. {
10. FastMutex::ScopedLock lock(\_mutex);
12. **typename** std::map<std::string, AbstractFactory\*>::const\_iterator it = \_map.find(className);
13. **if** (it != \_map.end())
14. **return** it->second->createInstance();
15. **else**
16. **throw** NotFoundException(className);
17. }
18. }

         DynamicFactory类在找个合适的类工厂后，就把任务交给了类AbstractFactory去完成。

         下面是使用动态类工厂的一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

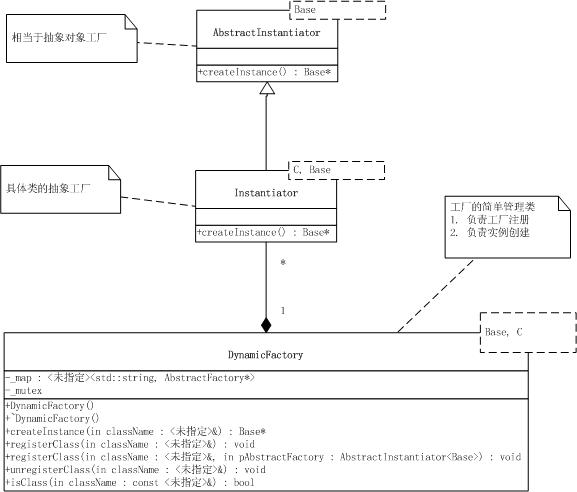
1. #include "Poco/DynamicFactory.h"
2. #include "Poco/SharedPtr.h"
3. **using** Poco::DynamicFactory;
4. **using** Poco::SharedPtr;
5. **class** Base
6. {
7. };
8. **class** A: **public** Base
9. {
10. };
11. **class** B: **public** Base
12. {
13. };
15. **int** main(**int** argc, **char**\*\* argv)
16. {
17. DynamicFactory<Base> factory;
18. factory.registerClass<A>("A"); // creates Instantiator<A, Base>
19. factory.registerClass<B>("B"); // creates Instantiator<B, Base>
20. SharedPtr<Base> pA = factory.createInstance("A");
21. SharedPtr<Base> pB = factory.createInstance("B");
22. // you can unregister classes
23. factory.unregisterClass("B");
24. // you can also check for the existence of a class
25. **bool** haveA = factory.isClass("A"); // true
26. **bool** haveB = factory.isClass("B"); // false (unregistered)
27. **bool** haveC = factory.isClass("C"); // false (never registered)
28. **return** 0;
29. }

         由于在Poco中用Poco::Instantiator类创建对象时使用的是类对象的默认构造函数，所以对于类创建时期望不使用默认构造函数或者对构造函数有一些特殊初始化过程要求的情况，用户必须自己实现抽象构造工厂。下面是其一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. #include "Poco/DynamicFactory.h"
2. **using** Poco::DynamicFactory;
3. **using** Poco::AbstractInstantiator;
4. **class** Base
5. {
6. };
7. **class** A: **public** Base
8. {
9. };
10. **class** C: **public** Base
11. {
12. **public**:
13. C(**int** i): \_i(i)
14. {}
15. **private**:
16. **int** \_i;
17. };
19. **class** CInstantiator: **public** AbstractInstantiator<Base>
20. {
21. **public**:
22. CInstantiator(**int** i): \_i(i)
23. {}
24. Base\* createInstance() **const**
25. {
26. **return** **new** C(\_i);
27. }
28. **private**:
29. **int** \_i;
30. };
32. **int** main(**int** argc, **char**\*\* argv)
33. {
34. DynamicFactory<Base> factory;
35. factory.registerClass<A>("A");
36. factory.registerClass("C", **new** CInstantiator(42));
37. **return** 0;
38. }

         最后给出 AbstractFactory模块的类图：



**6. 内存池（Memory Pools）**

        同以往看过的内存池比较，Poco中内存池相当简单。既不支持对象的分配，也不对内存块大小进行分级，并且释放后的内存的合并策略也很简单。但这绝不是说它简陋，对于大多数情况，我觉得其完全够用了。同AutoReleasePool比较，两者的不同之处在于，AutoReleasePool中内存的分配是交由用户进行的，AutoReleasePool只负责释放，而MemoryPool的思想是，内存的分配和释放都由其管理。

       首先来回顾一下内存池的作用：

       1. 解决应用程序频繁申请和释放内存带来的执行效率问题

       2. 解决内存碎片问题

       下面是Poco中内存池函数调用的一些特性：

       1. Poco::MemoryPool使用std::vector维护了一组固定大小的内存块指针，每个内存块大小都相等

       2. 可以通过MemoryPool::get()获得一个内存块的指针，如果池中内存块不够时，一个新的内存块会被分配。但当池中内存块数目到达池定义的上限时，一个OutOfMemoryException异常会被抛出。

       3. 调用MemoryPool::release(void\* ptr)将把内存块释放入池中

       其头文件中的定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. **class** Foundation\_API MemoryPool
2. /// A simple pool for fixed-size memory blocks.
3. ///
4. /// The main purpose of this class is to speed-up
5. /// memory allocations, as well as to reduce memory
6. /// fragmentation in situations where the same blocks
7. /// are allocated all over again, such as in server
8. /// applications.
9. ///
10. /// All allocated blocks are retained for future use.
11. /// A limit on the number of blocks can be specified.
12. /// Blocks can be preallocated.
13. {
14. **public**:
15. MemoryPool(std::**size\_t** blockSize, **int** preAlloc = 0, **int** maxAlloc = 0);
16. /// Creates a MemoryPool for blocks with the given blockSize.
17. /// The number of blocks given in preAlloc are preallocated.
19. ~MemoryPool();
21. **void**\* get();
22. /// Returns a memory block. If there are no more blocks
23. /// in the pool, a new block will be allocated.
24. ///
25. /// If maxAlloc blocks are already allocated, an
26. /// OutOfMemoryException is thrown.
28. **void** release(**void**\* ptr);
29. /// Releases a memory block and returns it to the pool.
31. std::**size\_t** blockSize() **const**;
32. /// Returns the block size.
34. **int** allocated() **const**;
35. /// Returns the number of allocated blocks.
37. **int** available() **const**;
38. /// Returns the number of available blocks in the pool.
40. **private**:
41. MemoryPool();
42. MemoryPool(**const** MemoryPool&);
43. MemoryPool& operator = (**const** MemoryPool&);
45. **enum**
46. {
47. BLOCK\_RESERVE = 128
48. };
50. **typedef** std::vector<**char**\*> BlockVec;
52. std::**size\_t** \_blockSize;
53. **int**         \_maxAlloc;
54. **int**         \_allocated;
55. BlockVec    \_blocks;
56. FastMutex   \_mutex;
57. };

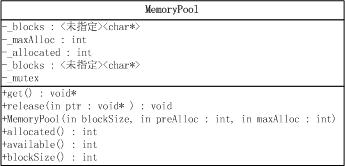
       其中\_maxAlloc是内存池可分配的最大内存块数，\_blockSize是每个内存块的大小。

       下面是内存池的一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. #include "Poco/MemoryPool.h"
2. #include <string>
3. #include <iostream>
4. **using** Poco::MemoryPool;
5. **int** main(**int** argc, **char**\*\* argv)
6. {
7. MemoryPool pool(1024); // unlimited number of 1024 byte blocks
8. // MemoryPool pool(1024, 4, 16); // at most 16 blocks; 4 preallocated
9. **char**\* buffer = **reinterpret\_cast**<**char**\*>(pool.get());
10. std::cin.read(buffer, pool.blockSize());
11. std::streamsize n = std::cin.gcount();
12. std::string s(buffer, n);
13. pool.release(buffer);
14. std::cout << s << std::endl;
15. **return** 0;
16. }

       最后给出MemoryPool的类图：



**7. 单件（Singletons）**

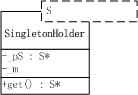
       Poco中的单件可以由类模板SingletonHolder完成。其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8639863)

1. **template** <**class** S>
2. **class** SingletonHolder
3. /// This is a helper template class for managing
4. /// singleton objects allocated on the heap.
5. /// The class ensures proper deletion (including
6. /// calling of the destructor) of singleton objects
7. /// when the application that created them terminates.
8. {
9. **public**:
10. SingletonHolder():
11. \_pS(0)
12. /// Creates the SingletonHolder.
13. {
14. }
16. ~SingletonHolder()
17. /// Destroys the SingletonHolder and the singleton
18. /// object that it holds.
19. {
20. **delete** \_pS;
21. }
23. S\* get()
24. /// Returns a pointer to the singleton object
25. /// hold by the SingletonHolder. The first call
26. /// to get will create the singleton.
27. {
28. FastMutex::ScopedLock lock(\_m);
29. **if** (!\_pS) \_pS = **new** S;
30. **return** \_pS;
31. }
33. **private**:
34. S\* \_pS;
35. FastMutex \_m;
36. };

       一眼可以望穿的代码，实在没有什么可以说的。噢，补充一下。在Poco中的Singleton模型里并没有使用DCLP（Double Checked Locking）模式。什么是DCLP。可以参考文章[Double Checked Locking 模式](http://blog.csdn.net/arau_sh/article/details/5579874)。

      其类图如下:



（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8639863>）