**POCO C++库学习和分析 -- Cache**

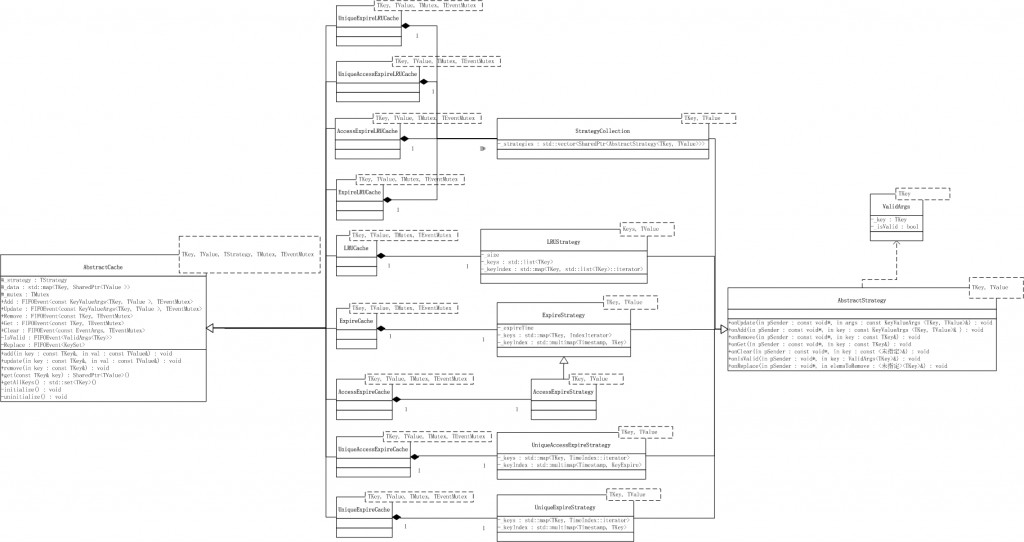
**1. Cache概述**

        在STL::map或者STL::set中，容器的尺寸是没有上限的，数目可以不断的扩充。并且在STL的容器中，元素是不会自动过期的，除非显式的被删除。Poco的Cache可以被看成是STL中容器的一个扩充，容器中的元素会自动过期(即失效)。在Poco实现的Cache框架中，基础的过期策略有两种。一种是LRU(Last Recent Used)，另外一种是基于时间的过期(Time based expiration)。在上述两种过期策略之上，还提供了两者之间的混合。

        下面是相关的类：  
        1. LRUCache: 最近使用Cache。在内部维护一个Cache的最大容量M，始终只保存M个元素于Cache内部，当第M+1元素插入Cache中时，最先被放入Cache中的元素将失效。  
        2. ExpireCache: 时间过期Cache。在内部统一管理失效时间T，当元素插入Cache后，超过时间T，则删除。  
        3. AccessExpireCache: 时间过期Cache。同ExpireCache不同的是，当元素被访问后，重新开始计算该元素的超时时间，而不是只从元素插入时开始计时。  
        4. UniqueExpireCache: 时间过期Cache。同ExpireCache不同的是，每一个元素都有自己单独的失效时间。  
        5. UniqueAccessExpireCache：时间过期Cache。同AccessExpireCache不同的是，每一个元素都有自己单独的失效时间。  
        6. ExpireLRUCache：时间过期和LRU策略的混合体。当时间过期和LRU任一过期条件被触发时，容器中的元素失效。  
        7. AccessExpireLRUCache：时间过期和LRU策略的混合体。同ExpireLRUCache相比，当元素被访问后，重新开始计算该元素的超时时间，而不是只从元素插入时开始计时。  
        8. UniqueExpireLRUCache：时间过期和LRU策略的混合体。同ExpireLRUCache相比，每一个元素都有自己单独的失效时间。  
        9. UniqueAccessExpireLRUCache：时间过期和LRU策略的混合体。同UniqueExpireLRUCache相比，当元素被访问后，重新开始计算该元素的超时时间，而不是只从元素插入时开始计时。

**2. Cache的内部结构**

**2.1 Cache类**

        下面是Poco中Cache的类图：  
  
  
        从类图中我们可以看到所有的Cache都有一个对应的strategy类。事实上strategy类负责快速搜索Cache中的过期元素。Cache和strategy采用了Poco中的同步事件机制([POCO C++库学习和分析 -- 通知和事件 （四）](http://blog.csdn.net/arau_sh/article/details/8673557) )。  
  
  
        让我们来看AbstractCache的定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698434)

1. **template** <**class** TKey, **class** TValue, **class** TStrategy, **class** TMutex = FastMutex, **class** TEventMutex = FastMutex>
2. **class** AbstractCache
3. /// An AbstractCache is the interface of all caches.
4. {
5. **public**:
6. FIFOEvent<**const** KeyValueArgs<TKey, TValue >, TEventMutex > Add;
7. FIFOEvent<**const** KeyValueArgs<TKey, TValue >, TEventMutex > Update;
8. FIFOEvent<**const** TKey, TEventMutex>                         Remove;
9. FIFOEvent<**const** TKey, TEventMutex>                         Get;
10. FIFOEvent<**const** EventArgs, TEventMutex>                    Clear;
12. **typedef** std::map<TKey, SharedPtr<TValue > > DataHolder;
13. **typedef** **typename** DataHolder::iterator       Iterator;
14. **typedef** **typename** DataHolder::const\_iterator ConstIterator;
15. **typedef** std::set<TKey>                      KeySet;
17. AbstractCache()
18. {
19. initialize();
20. }
22. AbstractCache(**const** TStrategy& strat): \_strategy(strat)
23. {
24. initialize();
25. }
27. **virtual** ~AbstractCache()
28. {
29. uninitialize();
30. }
32. // ...........
34. **protected**:
35. **mutable** FIFOEvent<ValidArgs<TKey> > IsValid;
36. **mutable** FIFOEvent<KeySet>           Replace;
38. **void** initialize()
39. /// Sets up event registration.
40. {
41. Add     += Delegate<TStrategy, **const** KeyValueArgs<TKey, TValue> >(&\_strategy, &TStrategy::onAdd);
42. Update  += Delegate<TStrategy, **const** KeyValueArgs<TKey, TValue> >(&\_strategy, &TStrategy::onUpdate);
43. Remove  += Delegate<TStrategy, **const** TKey>(&\_strategy, &TStrategy::onRemove);
44. Get     += Delegate<TStrategy, **const** TKey>(&\_strategy, &TStrategy::onGet);
45. Clear   += Delegate<TStrategy, **const** EventArgs>(&\_strategy, &TStrategy::onClear);
46. IsValid += Delegate<TStrategy, ValidArgs<TKey> >(&\_strategy, &TStrategy::onIsValid);
47. Replace += Delegate<TStrategy, KeySet>(&\_strategy, &TStrategy::onReplace);
48. }
50. **void** uninitialize()
51. /// Reverts event registration.
52. {
53. Add     -= Delegate<TStrategy, **const** KeyValueArgs<TKey, TValue> >(&\_strategy, &TStrategy::onAdd );
54. Update  -= Delegate<TStrategy, **const** KeyValueArgs<TKey, TValue> >(&\_strategy, &TStrategy::onUpdate);
55. Remove  -= Delegate<TStrategy, **const** TKey>(&\_strategy, &TStrategy::onRemove);
56. Get     -= Delegate<TStrategy, **const** TKey>(&\_strategy, &TStrategy::onGet);
57. Clear   -= Delegate<TStrategy, **const** EventArgs>(&\_strategy, &TStrategy::onClear);
58. IsValid -= Delegate<TStrategy, ValidArgs<TKey> >(&\_strategy, &TStrategy::onIsValid);
59. Replace -= Delegate<TStrategy, KeySet>(&\_strategy, &TStrategy::onReplace);
60. }
62. **void** doAdd(**const** TKey& key, **const** TValue& val)
63. /// Adds the key value pair to the cache.
64. /// If for the key already an entry exists, it will be overwritten.
65. {
66. Iterator it = \_data.find(key);
67. doRemove(it);

70. KeyValueArgs<TKey, TValue> args(key, val);
71. Add.notify(**this**, args);
72. \_data.insert(std::make\_pair(key, SharedPtr<TValue>(**new** TValue(val))));
74. doReplace();
75. }
77. **void** doAdd(**const** TKey& key, SharedPtr<TValue>& val)
78. /// Adds the key value pair to the cache.
79. /// If for the key already an entry exists, it will be overwritten.
80. {
81. Iterator it = \_data.find(key);
82. doRemove(it);

85. KeyValueArgs<TKey, TValue> args(key, \*val);
86. Add.notify(**this**, args);
87. \_data.insert(std::make\_pair(key, val));
89. doReplace();
90. }
92. **void** doUpdate(**const** TKey& key, **const** TValue& val)
93. /// Adds the key value pair to the cache.
94. /// If for the key already an entry exists, it will be overwritten.
95. {
96. KeyValueArgs<TKey, TValue> args(key, val);
97. Iterator it = \_data.find(key);
98. **if** (it == \_data.end())
99. {
100. Add.notify(**this**, args);
101. \_data.insert(std::make\_pair(key, SharedPtr<TValue>(**new** TValue(val))));
102. }
103. **else**
104. {
105. Update.notify(**this**, args);
106. it->second = SharedPtr<TValue>(**new** TValue(val));
107. }
109. doReplace();
110. }
112. **void** doUpdate(**const** TKey& key, SharedPtr<TValue>& val)
113. /// Adds the key value pair to the cache.
114. /// If for the key already an entry exists, it will be overwritten.
115. {
116. KeyValueArgs<TKey, TValue> args(key, \*val);
117. Iterator it = \_data.find(key);
118. **if** (it == \_data.end())
119. {
120. Add.notify(**this**, args);
121. \_data.insert(std::make\_pair(key, val));
122. }
123. **else**
124. {
125. Update.notify(**this**, args);
126. it->second = val;
127. }
129. doReplace();
130. }
132. **void** doRemove(Iterator it)
133. /// Removes an entry from the cache. If the entry is not found
134. /// the remove is ignored.
135. {
136. **if** (it != \_data.end())
137. {
138. Remove.notify(**this**, it->first);
139. \_data.erase(it);
140. }
141. }
143. **bool** doHas(**const** TKey& key) **const**
144. /// Returns true if the cache contains a value for the key
145. {
146. // ask the strategy if the key is valid
147. ConstIterator it = \_data.find(key);
148. **bool** result = **false**;

151. **if** (it != \_data.end())
152. {
153. ValidArgs<TKey> args(key);
154. IsValid.notify(**this**, args);
155. result = args.isValid();
156. }
158. **return** result;
159. }
161. SharedPtr<TValue> doGet(**const** TKey& key)
162. /// Returns a SharedPtr of the cache entry, returns 0 if for
163. /// the key no value was found
164. {
165. Iterator it = \_data.find(key);
166. SharedPtr<TValue> result;
168. **if** (it != \_data.end())
169. {
170. // inform all strategies that a read-access to an element happens
171. Get.notify(**this**, key);
172. // ask all strategies if the key is valid
173. ValidArgs<TKey> args(key);
174. IsValid.notify(**this**, args);
176. **if** (!args.isValid())
177. {
178. doRemove(it);
179. }
180. **else**
181. {
182. result = it->second;
183. }
184. }
186. **return** result;
187. }
189. **void** doClear()
190. {
191. **static** EventArgs \_emptyArgs;
192. Clear.notify(**this**, \_emptyArgs);
193. \_data.clear();
194. }
196. **void** doReplace()
197. {
198. std::set<TKey> delMe;
199. Replace.notify(**this**, delMe);
200. // delMe contains the to be removed elements
201. **typename** std::set<TKey>::const\_iterator it    = delMe.begin();
202. **typename** std::set<TKey>::const\_iterator endIt = delMe.end();
204. **for** (; it != endIt; ++it)
205. {
206. Iterator itH = \_data.find(\*it);
207. doRemove(itH);
208. }
209. }
211. TStrategy          \_strategy;
212. **mutable** DataHolder \_data;
213. **mutable** TMutex  \_mutex;
215. **private**:
216. // ....
217. };

        从上面的定义中，可以看到AbstractCache是一个value的容器，采用map保存数据，

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698434)

1. **mutable** std::map<TKey, SharedPtr<TValue > > \_data;

        另外AbstractCache中还定义了一个TStrategy对象，

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698434)

1. TStrategy          \_strategy;

        并且在AbstractCache的initialize()函数中，把Cache的一些函数操作委托给TStrategy对象。其函数操作接口为：  
        1. Add : 向容器中添加元素  
        2. Update : 更新容器中元素  
        3. Remove : 删除容器中元素  
        4. Get : 获取容器中元素  
        5. Clear : 清除容器中所有元素  
        6. IsValid: 容器中是否某元素  
        7. Replace: 按照策略从strategy中获取过期元素，并从Cache和Strategy中同时删除。将触发一系列的Remove函数。  
  
        这几个操作中最复杂的是Add操作，其中包括了Remove、Insert和Replace操作。

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698434)

1. **void** doAdd(**const** TKey& key, SharedPtr<TValue>& val)
2. /// Adds the key value pair to the cache.
3. /// If for the key already an entry exists, it will be overwritten.
4. {
5. Iterator it = \_data.find(key);
6. doRemove(it);

9. KeyValueArgs<TKey, TValue> args(key, \*val);
10. Add.notify(**this**, args);
11. \_data.insert(std::make\_pair(key, val));
13. doReplace();
14. }

        而Replace操作可被Add、Update、Get操作触发。这是因为Cache并不是一个主动对象[(POCO C++库学习和分析 -- 线程 （四）](http://blog.csdn.net/arau_sh/article/details/8618571))，不会自动的把元素标志为失效，需要外界也就是调用方触发进行。

        在Cache类中另外一个值得注意的地方是，保存的是TValue的SharedPtr。之所以这么设计，是为了线程安全，由于replace操作可能被多个线程调用，所以解决的方法，要么是返回TValue的SharedPtr，要么是返回TValue的拷贝。同拷贝方法相比，SharedPtr的方法要更加廉价。

**2.2 Strategy类**

        Strategy类完成了对\_data中保存的<key-value>pair中key的排序工作。每个Strategy中都存在一个key的容器，其中LRUStrategy中是std::list<TKey>，ExpireStrategy、UniqueAccessExpireStrategy、UniqueExpireStrategy中是std::multimap<Timestamp, TKey>。

       对于LRU策略，这么设计我是可以理解的。每次访问都会使key被重置于list的最前端。为了实现对list快速访问，增加一个std::map<TKey, Iterator>容器，每次对list容器进行插入操作时，把插入位的itorator保存入map中，这样对于list的访问效率可以从O(n)变成O(log(n)),因为不需要遍历了。下面是相关的代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698434)

1. **void** onReplace(**const** **void**\*, std::set<TKey>& elemsToRemove)
2. {
3. // Note: replace only informs the cache which elements
4. // it would like to remove!
5. // it does not remove them on its own!
6. std::**size\_t** curSize = \_keyIndex.size();
8. **if** (curSize < \_size)
9. {
10. **return**;
11. }
13. std::**size\_t** diff = curSize - \_size;
14. Iterator it = --\_keys.end(); //--keys can never be invoked on an empty list due to the minSize==1 requirement of LRU
15. std::**size\_t** i = 0;
17. **while** (i++ < diff)
18. {
19. elemsToRemove.insert(\*it);
20. **if** (it != \_keys.begin())
21. {
22. --it;
23. }
24. }
25. }

        LRUStrategy的replace操作是，只在curSize超过设定的访问上限\_size时触发，把list容器中排在末尾的(curSize-\_size)个元素标志为失效。  
  
        而对于Time base expired策略，还如此设计，我觉得不太合适。在时间策略的strategy类中，存在着两个容器，一个是std::map<TKey, IndexIterator>，另外一个是std::multimap<Timestamp, TKey>。进行插入操作时，代码为：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698434)

1. **void** onAdd(**const** **void**\*, **const** KeyValueArgs <TKey, TValue>& args)
2. {
3. Timestamp now;
4. IndexIterator it = \_keyIndex.insert(**typename** TimeIndex::value\_type(now, args.key()));
5. std::pair<Iterator, **bool**> stat = \_keys.insert(**typename** Keys::value\_type(args.key(), it));
6. **if** (!stat.second)
7. {
8. \_keyIndex.erase(stat.first->second);
9. stat.first->second = it;
10. }
11. }

        可以看到map容器中保存的是multimap中pair对的itorator。其replace操作如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698434)

1. **void** onReplace(**const** **void**\*, std::set<TKey>& elemsToRemove)
2. {
3. // Note: replace only informs the cache which elements
4. // it would like to remove!
5. // it does not remove them on its own!
6. IndexIterator it = \_keyIndex.begin();
7. **while** (it != \_keyIndex.end() && it->first.isElapsed(\_expireTime))
8. {
9. elemsToRemove.insert(it->second);
10. ++it;
11. }
12. }

        可以看到这是对multimap的遍历，效率为O(n)。  
  
        如果这样的话，我觉得完全可以把std::map<TKey, IndexIterator>和std::multimap<Timestamp, TKey>合二为一，定义成为std::map<TKey, Timestamp>，replace的操作仍然采用遍历，效率为O(n).  
        对于基于时间的策略，O(n)的效率可能不能接受。我觉得可能的解决方法有两种。第一，把Cache变成主动对象，内部定期的收集失效元素，而不由外部触发。这样虽然并没有提高replace操作效率，但把replace操作和外部接口的add等操作分开了。外部调用接口的效率提高了。第二，在内部实现多个map容器，分组管理不同过期时间的对象。

**3. 开销**

        Poco中的Cache类比std::map要慢，其中开销最大的操作为add操作。采用Time Expire策略的Cache要比采用LRU策略的Cache更慢。并且由于Cache类引入了SharePtr和Strategy，其空间花费也要大于std::map。所以在没有必要使用Cache的情况下，还是使用map较好。

**4. 例子**

        下面是Cache的一个示例：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8698434)

1. #include "Poco/LRUCache.h"
2. **int** main()
3. {
4. Poco::LRUCache<**int**, std::string> myCache(3);
5. myCache.add(1, "Lousy"); // |-1-| -> first elem is the most popular one
6. Poco::SharedPtr<std::string> ptrElem = myCache.get(1); // |-1-|
7. myCache.add(2, "Morning"); // |-2-1-|
8. myCache.add(3, "USA"); // |-3-2-1-|
9. // now get rid of the most unpopular entry: "Lousy"
10. myCache.add(4, "Good"); // |-4-3-2-|
11. poco\_assert (\*ptrElem == "Lousy"); // content of ptrElem is still valid
12. ptrElem = myCache.get(2); // |-2-4-3-|
13. // replace the morning entry with evening
14. myCache.add(2, "Evening"); // 2 Events: Remove followed by Add
15. }

（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8698434>）