The Cache Framework

Keeping things around.

applied informatics

Overview

- Motivation
- Features
- Interface
- Examples
- Performance Costs
- Memory Costs

Why Caching?

- STL containers make a good job in allowing you to store data
  - but you can't limit the size of the container, a map/set can potentially grow to infinite size
  - you can't let elements expire
- POCO's caching framework supports exactly that
  - only std::map like behavior is supported so far
  - don't use caching if you don't need these properties (more on that in the "Costs" section)

Features

- POCO's caching framework supports the two most commonly used cache algorithms
  - Least Recently Used (LRU)
    - Size limited
    - Keeps the last accessed elements in cache
  - Time based expiration
    - Not size limited
    - Either with a default time value applied to all entries or a unique time value
  - Combinations: LRU combined with expiration

Classes

- Poco::LRUCache
  #include "Poco/LRUCache.h"
- Poco::ExpireCache
  #include "Poco/ExpireCache.h"
- Poco::UniqueExpireCache
  #include "Poco/UniqueExpireCache.h"
- Poco::ExpireLRUCache
  #include "Poco/ExpireLRUCache.h"
- Poco::UniqueExpireLRUCache
  #include "Poco/UniqueExpireLRUCache.h"

Classes (cont)

- Poco::AccessExpireCache
  #include "Poco/AccessExpireCache.h"
- Poco::AccessExpireLRUCache
  #include "Poco/AccessExpireLRUCache.h"
- Poco::UniqueAccessExpireCache
  #include "Poco/UniqueAccessExpireCache.h"
- Poco::UniqueAccessExpireLRUCache
  #include "Poco/UniqueAccessExpireLRUCache.h"

Interface: AbstractCache

- Realized as template<TKey, TVal, TStrat>
- void add(const TKey&, const TVal&)
  replaces existing entries
- void remove(const TKey&)
- bool has(const TKey&)
- Poco::SharedPtr<TVal> get(const TKey&)
- void clear()
- std::size_t size()
- void forceReplace()
- TStrat is the strategy used for cache replacement

Getting Elements

- Why return a SharedPtr?
  - accessing a cache triggers cache replacement (add(), but also get(), size())
  - a thread-safe cache must guarantee that no outside data gets invalidated by replacement, thus
    - we can't return a reference/pointer
    - we have no iterators for cache
  - we had a choice: return a copy or return a SharedPtr
  - SharedPtr is cheaper than copying.

Interface – Events

- FIFOEvent<const KeyValueArgs<const TKey, TVal>> Add
  FIFOEvent<const TKey> Remove
  FIFOEvent<const TKey> Get
  FIFOEvent<const TKey> Clear

- Events are thrown before the operation is performed
- Useful for:
  - profiling a cache
  - GUI Code to visualize cache content

```
#include "Poco/LRUCache.h"
[...]

Poco::LRUCache<int, std::string> myCache(3);
myCache.add(1, "Lousy"); // |-1-| -> first elem is the most popular one
Poco::SharedPtr<std::string> ptrElem = myCache.get(1); // |-1-|
myCache.add(2, "Morning"); // |-2-1-|
myCache.add(3, "USA");  // |-3-2-1-|

// now get rid of the most unpopular entry: "Lousy"
myCache.add(4, "Good"); // |-4-3-2-|
poco_assert (*ptrElem == "Lousy"); // content of ptrElem is still valid

ptrElem = myCache.get(2); // |-2-4-3-|

// replace the morning entry with evening
myCache.add(2, "Evening"); // 2 Events: Remove followed by Add
```

Time Based Expiration

- ExpireCache has a default timeout for all values (600000 ms = 10 minutes)
  - Poco::ExpireCache<int, std::string> e; // 10min
  - Poco::ExpireCache<int, std::string> e(1000); // 1sec
- UniqueExpireCache has unique expire value per element
  - each value type must implement const Poco::Timestamp& getExpiration()
  - for builtin C++ data types use Poco::ExpirationDecorator<TVal>

```
#include "Poco/UniqueExpireCache.h"
#include "Poco/ExpirationDecorator.h"
[...]

typedef Poco::ExpirationDecorator<std::string> ExpString;

Poco::UniqueExpireCache<int, ExpString> myCache;
myCache.add(1, ExpString("test", 500));  // expires after 500ms
myCache.add(2, ExpString("test", 1500)); // expires after 1500ms

poco_assert (myCache.size() == 2);

// 1 second passes...
poco_assert (myCache.size() == 1);
Poco::SharedPtr<ExpString> ptrVal = myCache.get(1);
poco_assert (ptrVal.isNull());
```

Expiration based on last access time

- AccessExpireCache:
  elements in the cache expire if they are not accessed within a given cache specific time interval
- UniqueAccessExpireCache:
  the time interval can be defined for each individual element of the cache

Cache Internals

- AbstractCache uses a CacheStrategy:
  - LRUStrategy for LRUCache
  - ExpireStrategy for ExpireCache
- ExpireLRUCache uses StrategyCollection containing:
  - LRUStrategy
  - ExpireStrategy
- UniqueExpireStrategy uses StrategyCollection containing:
  - LRUStrategy
  - UniqueExpireStrategy

Cache Performance

- A cache is slower than a std::map:
  - cache replacement costs time
- most expensive method: add()
  - first search and remove an old entry
  - insert into the AbstractCache
  - then insert into the Strategy
  - do cache replacement

Cache Performance (cont'd)

- Costs
  - all operations are still O(log(n))
  - exact: O((x + numofstrategies) * (log(n))
    i.e. ExpireLRUCache is more expensive than LRUCache
    - x:={1, 2} depending on method called

Cache Performance (Memory)

- AbstractCache without strategy requires approx. the same memory as a std::map; overhead is one SharedPtr object per entry.
- Additional overhead per strategy:
  each must store the keys at least
- Strategies are optimized for speed not memory.
  Typically, you need two different views on the keys so you can guarantee O(log(n)), otherwise you would end up with O(n).

Cache Performance (Memory per Entry)

- Base costs: sizeof(TKey)+sizeof(TVal)
- LRUCache:
  sizeof(Poco::SharedPtr) + sizeof(TKey)*2
  + sizeof(std::list::iterator)
- ExpireCache, AccessExpireCache:
  sizeof(Poco::SharedPtr) + sizeof(TKey)*2
  + sizeof(std::multimap::iterator) + sizeof(Poco::Timestamp)
- UniqueExpireCache, UniqueAccessExpireCache:
  sizeof(Poco::SharedPtr) + sizeof(Key)*2
  + sizeof(std::multimap::iterator)

Cache Performance (Memory per Entry)

- ExpireLRUCache, AccessExpireLRUCache:
  sizeof(Poco::SharedPtr) + sizeof(TKey)*4
  + sizeof(std::multimap::iterator) + sizeof(std::list::iterator)
  + sizeof(Poco::Timestamp)
- UniqueExpireLRUCache, UniqueAccessExpireLRUCache:
  sizeof(Poco::SharedPtr) + sizeof(TKey)*4
  + sizeof(std::multimap::iterator) + sizeof(std::list::iterator)

Tips

- Caches offer expiration and LRU support – if you don't need these features, don't use cache!
- Time based expiration works without a background thread
  - if nobody accesses the cache, nothing will be replaced
  - you can call forceReplace()
- Caching Framework is extensible:
  extend Poco::AbstractStrategy to implement new strategies, or combine them with existing strategies

Tips (cont'd)

- don't use has() followed by get(), it's faster (and safer with expiration) to go directly to get and check if the returned SharedPtr contains null
- Cache resembles a std::map, for set functionality, use Poco::Void as value:

```
Poco::LRUCache<std::string, Poco::Void>
```

applied informatics

www.appinf.com | info@appinf.com
T +43 4253 32596 | F +43 4253 32096