Implementing a Servant Evictor in C++

On this page:

The `EvictorBase` Class in C++

The evictor we show here is designed as an abstract base class: in order to use it, you derive a class from the `EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definition as follows:

C++

```
class EvictorBase : public Ice::ServantLocator {
public:
    EvictorBase(int size = 1000);

    virtual Ice::ObjectPtr locate(const Ice::Current& c, Ice::LocalObjectPtr& cookie);

    virtual void finished(const Ice::Current& c, const Ice::ObjectPtr&,
                          const Ice::LocalObjectPtr& cookie);

    virtual void deactivate(const std::string&);

protected:
    virtual Ice::ObjectPtr add(const Ice::Current&, Ice::LocalObjectPtr&) = 0;

    virtual void evict(const Ice::ObjectPtr&, const Ice::LocalObjectPtr&) = 0;

private:
    // ...
};

typedef IceUtil::Handle<EvictorBase> EvictorBasePtr;
```

Note that the evictor has a constructor that sets the size of the queue, with a default argument to set the size to 1000.

The `locate`, `finished`, and `deactivate` functions are inherited from the `ServantLocator` base class; these functions implement the logic to maintain the queue in LRU order and to add and evict servants as needed.

The `add` and `evict` functions are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are pure virtual, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant, allowing the subclass to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map not only stores servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue. Storing the queue position is not strictly necessary — we store the position for efficiency reasons because it allows us to locate a servant's position in the queue in constant time instead of having to search through the queue in order to maintain its LRU property.
3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The need for the use count deserves some extra explanation: suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

The evictor queue does not store the identity of the servant. Instead, the entries on the queue are iterators into the evictor map. This is useful when the time comes to evict a servant: instead of having to search the map for the identity of the servant to be evicted, we can simply delete the map entry that is pointed at by the iterator at the tail of the queue. We can get away with storing an iterator into the evictor queue as part of the map, and storing an iterator into the evictor map as part of the queue because both `std::list` and `std::map` do not invalidate forward iterators when we add or delete entries (except for invalidating iterators that point at a deleted entry, of course).

Reverse iterators *can* be invalidated by modification of list entries: if a reverse iterator points at `rend` and the element at the head of the list is erased, the iterator pointing at `rend` is invalidated.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

C++

```
class EvictorBase : public Ice::ServantLocator {
    // ...

private:

    struct EvictorEntry;
    typedef IceUtil::Handle<EvictorEntry> EvictorEntryPtr;

    typedef std::map<Ice::Identity, EvictorEntryPtr> EvictorMap;
    typedef std::list<EvictorMap::iterator> EvictorQueue;

    struct EvictorEntry : public Ice::LocalObject
    {
        Ice::ObjectPtr servant;
        Ice::LocalObjectPtr userCookie;
        EvictorQueue::iterator queuePos;
        int useCount;
    };

    EvictorMap _map;
    EvictorQueue _queue;
    Ice::Int _size;

    IceUtil::Mutex _mutex;

    void evictServants();
};
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. In addition, we use a private `_mutex` data member so we can correctly serialize access to the evictor's data structures.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit — we will discuss this function in more detail shortly.

The `EvictorEntry` structure serves as the cookie that we pass from `locate` to `finished`; it stores the servant, the servant's position in the evictor queue, the servant's use count, and the cookie that we pass from `add` to `evict`.

The implementation of the constructor is trivial. The only point of note is that we ignore negative sizes:

C++

```
EvictorBase::EvictorBase(Ice::Int size) : _size(size)
{
    if (_size < 0)
        _size = 1000;
}
```

We could have stored the size as a `size_t` instead. However, for consistency with the Java implementation, which cannot use unsigned integers, we use `Ice::Int` to store the size.

Almost all the action of the evictor takes place in the implementation of `locate`:

C++

```
Ice::ObjectPtr
EvictorBase::locate(const Ice::Current& c, Ice::LocalObjectPtr& cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    //
    // Check if we have a servant in the map already.
    //
    EvictorEntryPtr entry;
    EvictorMap::iterator i = _map.find(c.id);
    if (i != _map.end()) {
        //
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        entry = i->second;
        _queue.erase(entry->queuePos);
    } else {
        //
        // We do not have an entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        entry = new EvictorEntry;
        entry->servant = add(c, entry->userCookie); // Down-call
        if (!entry->servant) {
            return 0;
        }
        entry->useCount = 0;
        i = _map.insert(std::make_pair(c.id, entry)).first;
    }

    //
    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(entry->useCount);
    entry->queuePos = _queue.insert(_queue.begin(), i);

    cookie = entry;

    return entry->servant;
}
```

The first step in `locate` is to lock the `_mutex` data member. This protects the evictor's data structures from concurrent access. The next step is to instantiate a smart pointer to an `EvictorEntry`. That smart pointer acts as the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`. That same smart pointer is also the value type of our map entries, so we do not store two copies of the same information redundantly — instead, smart pointers ensure that a single copy of each `EvictorEntry` structure is shared by both the cookie and the map.

The next step is to look in the evictor map to see whether we already have an entry for this object identity. If so, we remove the entry from its current queue position.

Otherwise, we do not have an entry for this object identity yet, so we have to create one. The code creates a new evictor entry, and then calls `add` to get a new servant. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we return zero to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The last few lines of `locate` add the entry for the current request to the head of the evictor queue to maintain its LRU property, increment the use count of the entry, set the cookie that is returned from locate to point at the `EvictorEntry`, and finally return the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

C++

```
void
EvictorBase::finished(const Ice::Current&,
                      const Ice::ObjectPtr&,
                      const Ice::LocalObjectPtr& cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    EvictorCookiePtr ec = EvictorCookiePtr::dynamicCast(cookie);

    // Decrement use count and check if
    // there is something to evict.
    //
    --(ec->entry->useCount);
    evictServants();
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

C++

```
void
EvictorBase::evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    EvictorQueue::reverse_iterator p = _queue.rbegin();
    int excessEntries = static_cast<int>(_map.size() - _size);

    for (int i = 0; i < excessEntries; ++i) {
        EvictorMap::iterator mapPos = *p;
        if (mapPos->second->useCount == 0) {
            evict(mapPos->second->servant, mapPos->second->userCookie);
            p = EvictorQueue::reverse_iterator(_queue.erase(mapPos->second->queuePos));
            _map.erase(mapPos);
        }
        else
            ++p;
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

C++

```
void
EvictorBase::deactivate(const std::string& category)
{
    IceUtil::Mutex::Lock lock(_mutex);

    _size = 0;
    evictServants();
}
```

Note that, with this implementation of `evictServants`, we only scan the tail section of the evictor queue for servants to evict. If we have long-running operations, this allows the number of servants in the queue to remain above the evictor size if the servants in the tail section have a non-zero use count. This means that, even immediately after calling `evictServants`, the queue length can still exceed the evictor size.

We can adopt a more aggressive strategy for eviction: instead of scanning only the excess entries in the queue, if, after looking in the tail section of the queue, we still have more servants in the queue than the queue size, we keep scanning for servants with a zero use count until the queue size drops below the limit. This alternative version of `evictServants` looks as follows:

C++

```
void
EvictorBase::evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // try to evict servants until the length drops
    // below the limit.
    //
    EvictorQueue::reverse_iterator p = _queue.rbegin();
    int numEntries = static_cast<int>_map.size();

    for (int i = 0; i < numEntries && _map.size() > _size; ++i) {
        EvictorMap::iterator mapPos = *p;
        if (mapPos->second->useCount == 0) {
            evict(mapPos->second->servant, mapPos->second->userCookie);
            p = EvictorQueue::reverse_iterator(_queue.erase(mapPos->second->queuePos));
            _map.erase(mapPos);
        }
        else
            ++p;
    }
}
```

The only difference in this version is that the terminating condition for the `for`-loop has changed: instead of scanning only the excess entries for servants with a use count, this version keeps scanning until the evictor size drops below the limit.

Which version is more appropriate depends on your application: if locating and evicting servants is expensive, and memory is not at a premium, the first version (which only scans the tail section) is more appropriate; if you want to keep memory consumption to a minimum, the second version is more appropriate. Also keep in mind that the difference between the two versions is significant only if you have long-running operations and many concurrent invocations from clients; otherwise, there is no point in more aggressively scanning for servants to remove because they are going to become idle again very quickly and get evicted as soon as the next request arrives.

Using Servant Evictors in C++

Using a servant evictor is simply a matter of deriving a class from `EvictorBase` and implementing the `add` and `evict` methods. You can turn a servant locator into an evictor by simply taking the code that you wrote for `locate` and placing it into `add` — `EvictorBase` then takes care of maintaining the cache in least-recently used order and evicting servants as necessary. Unless you have clean-up requirements for your servants (such as closing network connections or database handles), the implementation of `evict` can be left empty.

One of the nice aspects of evictors is that you do not need to change anything in your servant implementation: the servants are ignorant of the fact that an evictor is in use. This makes it very easy to add an evictor to an already existing code base with little disturbance of the source code.

Evictors can provide substantial performance improvements over default servants: especially if initialization of servants is expensive (for example, because servant state must be initialized by reading from a network), an evictor performs much better than a default servant, while keeping memory requirements low.

See Also

- Servant Evictors
- Object Identity
- Default Servants