Implementing a Servant Evictor in Java

On this page:

The `EvictorBase` Class in Java

The evictor we show here is designed as an abstract base class: in order to use it, you derive a class from the `Evictor.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:

Java

```
package Evictor;

public abstract class EvictorBase implements Ice.ServantLocator
{
    public
    EvictorBase()
    {
        _size = 1000;
    }

    public
    EvictorBase(int size)
    {
        _size = size < 0 ? 1000 : size;
    }

    public abstract Ice.Object
    add(Ice.Current c, Ice.LocalObjectHolder cookie);

    public abstract void
    evict(Ice.Object servant, java.lang.Object cookie);

    synchronized public final Ice.Object
    locate(Ice.Current c, Ice.LocalObjectHolder cookie)
    {
        // ...
    }

    synchronized public final void
    finished(Ice.Current c, Ice.Object o, java.lang.Object cookie)
    {
        // ...
    }

    synchronized public final void
    deactivate(String category)
    {
        // ...
    }

    // ...

    private int _size;
}
```

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

The `locate`, `finished`, and `deactivate` methods are inherited from the `ServantLocator` base class; these methods implement the logic to maintain the queue in LRU order and to add and evict servants as needed. The methods are synchronized, so the evictor's internal data structures are protected from concurrent access.

The `add` and `evict` methods 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 abstract, 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.
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 last two points deserve some extra explanation.

- The evictor queue must be maintained in least-recently used order, that is, every time an invocation arrives and we find an entry for the identity in the evictor map, we also must locate the servant's identity on the evictor queue and move it to the front of the queue. However, scanning for that entry is inefficient because it requires *O*(*n*) time. To get around this, we store an iterator in the evictor map that marks the corresponding entry's position in the evictor queue. This allows us to dequeue the entry from its current position and enqueue it at the head of the queue in *O*(1) time.

  Unfortunately, the various lists provided by `java.util` do not allow us to keep an iterator to a list position without invalidating that iterator as the list is updated. To deal with this, we use a special-purpose linked list implementation, `Evictor.LinkedList`, that does not have this limitation. `LinkedList` has an interface similar to `java.util.LinkedList` but does not invalidate iterators other than iterators that point at an element that is removed. For brevity, we do not show the implementation of this list here — you can find the implementation in the code examples for this manual in the Ice distribution.

- We maintain a use count as part of the map in order to avoid incorrect eviction of servants. 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.

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:

Java

```
package Evictor;

public abstract class EvictorBase implements Ice.ServantLocator
{
    // ...

    private class EvictorEntry
    {
        Ice.Object servant;
        java.lang.Object userCookie;
        java.util.Iterator<Ice.Identity> queuePos;
        int useCount;
    }

    private void evictServants()
    {
        // ...
    }

    private java.util.Map<Ice.Identity, EvictorEntry> _map =
        new java.util.HashMap<Ice.Identity, EvictorEntry>();
    private Evictor.LinkedList<Ice.Identity> _queue =
        new Evictor.LinkedList<Ice.Identity>();
    private int _size;
}
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. The map key is the identity of the Ice object, and the lookup value is of type `EvictorEntry`. The queue simply stores identities, of type `Ice::Identity`.

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.

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

Java

```java
synchronized public final Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    //
    // Check if we have a servant in the map already.
    //
    EvictorEntry entry = _map.get(c.id);
    if (entry != null) {
        //
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        entry.queuePos.remove();
    } else {
        //
        // We do not have entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        entry = new EvictorEntry();
        Ice.LocalObjectHolder cookieHolder = new Ice.LocalObjectHolder();
        entry.servant = add(c, cookieHolder); // Down-call
        if (entry.servant == null) {
            return null;
        }
        entry.userCookie = cookieHolder.value;
        entry.useCount = 0;
        _map.put(c.id, entry);
    }

    //
    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(entry.useCount);
    _queue.addFirst(c.id);
    entry.queuePos = _queue.iterator();
    entry.queuePos.next(); // Position iterator on the element.

    cookie.value = entry;
    return entry.servant;
}
```

The code uses an `EvictorEntry` as the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`.

We first look for an existing entry in the evictor map, using the object identity as the key. If we have an entry in the map already, we dequeue the corresponding identity from the evictor queue. (The `queuePos` member of `EvictorEntry` is an iterator that marks that entry's position in the evictor queue.)

Otherwise, we do not have an entry in the map, so we create a new one and call the `add` method. 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 null 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 final few lines of code increment the entry's use count, add the entry at the head of the evictor queue, store the entry's position in the queue, and assign the entry to the cookie that is returned from `locate`, before returning 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:

Java

```java
synchronized public final void
finished(Ice.Current c, Ice.Object o, java.lang.Object cookie)
{
    EvictorEntry entry = (EvictorEntry)cookie;

    // Decrement use count and check if
    // there is something to evict.
    //
    --(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:

Java

```java
private void 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.
    //
    java.util.Iterator<Ice.Identity> p = _queue.riterator();
    int excessEntries = _map.size() - _size;
    for (int i = 0; i < excessEntries; ++i) {
        Ice.Identity id = p.next();
        EvictorEntry e = _map.get(id);
        if (e.useCount == 0) {
            evict(e.servant, e.userCookie); // Down-call
            e.queuePos.remove();
            _map.remove(id);
        }
    }
}
```

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.

Java

```java
synchronized public final void
deactivate(String category)
{
    _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:

Java

```
private void 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.
    //
    java.util.Iterator<Ice.Identity> p = _queue.riterator();
    int numEntries = _map.size();
    for (int i = 0; i < excessEntries && _map.size() > _size;
         ++i) {
        Ice.Identity id = p.next();
        EvictorEntry e = _map.get(id);
        if (e.useCount == 0) {
            evict(e.servant, e.userCookie); // Down-call
            e.queuePos.remove();
            _map.remove(id);
        }
    }
}
```

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 be become idle again very quickly and get evicted as soon as the next request arrives.

Using Servant Evictors in Java

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