The C++ Cache Template

This class allows you to efficiently maintain a cache that is backed by secondary storage, such as a Berkeley DB database, without holding a lock on the entire cache while values are being loaded from the database. If you want to create evictors for servants that store their state in a database, the `Cache` class can simplify your evictor implementation considerably.

You may also want to examine the implementation of the Freeze background save evictor in the source distribution; it uses `IceUtil::Cache` for its implementation.

The `Cache` class has the following interface:

C++

```
template<typename Key, typename Value>
class Cache {
public:
    typedef typename std::map</* ... */, /* ... */>::iterator Position;

    bool pin(const Key& k, const Handle<Value>& v);
    Handle<Value> pin(const Key& k);
    void unpin(Position p);

    Handle<Value> putIfAbsent(const Key& k, const Handle<Value>& v);

    Handle<Value> getIfPinned(const Key&, bool = false) const;

    void clear();
    size_t size() const;


protected:
    virtual Handle<Value> load(const Key& k) = 0;
    virtual void pinned(const Handle<Value>& v, Position p);

    virtual ~Cache();
};
```

Note that `Cache` is an abstract base class — you must derive a concrete implementation from `Cache` and provide an implementation of the `load` and, optionally, of the `pinned` member function.

Internally, a `Cache` maintains a map of name-value pairs. The key and value type of the map are supplied by the `Key` and `Value` template arguments, respectively. The implementation of `Cache` takes care of maintaining the map; in particular, it ensures that concurrent lookups by callers are possible without blocking even if some of the callers are currently loading values from the backing store. In turn, this is useful for evictor implementations, such as the Freeze background save evictor. The `Cache` class does not limit the number of entries in the cache — it is the job of the evictor implementation to limit the map size by calling `unpin` on elements of the map that it wants to evict.

Your concrete implementation class must implement the `load` function, whose job it is to load the value for the key `k` from the backing store and to return a handle to that value. Note that `load` returns a value of type `IceUtil::Handle`, that is, the value must be heap-allocated and support the usual reference-counting functions for smart pointers. (The easiest way to achieve this is to derive the value from `IceUtil::Shared`.)

If `load` cannot locate a record for the given key because no such record exists, it must return a null handle. If `load` fails for some other reason, it can throw an exception, which is propagated back to the application code.

Your concrete implementation class typically will also override the `pinned` function (unless you want to have a cache that does not limit the number of entries; the provided default implementation of `pinned` is a no-op). The `Cache` implementation calls `pinned` whenever it has added a value to the map as a result of a call to `pin`; the `pinned` function is therefore a callback that allows the derived class to find out when a value has been added to the cache and informs the derived class of the value and its position in the cache.

The `Position` parameter is a `std::iterator` into the cache's internal map that records the position of the corresponding map entry. (Note that the element type of map is opaque, so you should not rely on knowledge of the cache's internal key and value types.) Your implementation of `pinned` must remember the position of the entry because that position is necessary to remove the corresponding entry from the cache again.

The public member functions of `Cache` behave as follows:

`bool pin(const Key& k, const Handle<Value>& v);`

To add a key-value pair to the cache, your evictor can call `pin`. The return value is true if the key and value were added; a false return value indicates that the map already contained an entry with the given key and the original value for that key is unchanged.

`pin` calls `pinned` if it adds an entry.

This version of `pin` does *not* call `load` to retrieve the entry from backing store if it is not yet in the cache. This is useful when you add a newly-created object to the cache.

Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

```
Handle<Value> pin(const Key& k);
```

A second version of `pin` looks for the entry with the given key in the cache. If the entry is already in the cache, `pin` returns the entry's value. If no entry with the given key is in the cache, `pin` calls `load` to retrieve the corresponding entry. If `load` returns an entry, `pin` adds it to the cache and returns the entry's value. If the entry cannot be retrieved from the backing store, `pin` returns null.

`pin` calls `pinned` if it adds an entry.

The function is thread-safe, that is, it calls `load` only once all other threads have unpinned the entry.

Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

```
Handle<Value> putIfAbsent(const Key& k, const Handle<Value>& v);
```

This function adds a key-value pair to the cache and returns a smart pointer to the value. If the map already contains an entry with the given key, that entry's value remains unchanged and `putIfAbsent` returns its value. If no entry with the given key is in the cache, `putIfAbsent` calls `load` to retrieve the corresponding entry. If `load` returns an entry, `putIfAbsent` adds it to the cache and returns the entry's value. If the entry cannot be retrieved from the backing store, `putIfAbsent` returns null.

`putIfAbsent` calls `pinned` if it adds an entry.

The function is thread-safe, that is, it calls `load` only once all other threads have unpinned the entry.

Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

```
Handle<Value> getIfPinned(const Key& k, bool wait = false) const;
```

This function returns the value stored for the key `k`.

- If an entry for the given key is in the map, the function returns the value immediately, regardless of the value of `wait`.
- If no entry for the given key is in the map and the `wait` parameter is false, the function returns a null handle.
- If no entry for the given key is in the map and the `wait` parameter is true, the function blocks the calling thread if another thread is currently attempting to load the same entry; once the other thread completes, `getIfPinned` completes and returns the value added by the other thread.

```
void unpin(Position p);
```

This function removes an entry from the map. The iterator p determines which entry to remove. (It must be an iterator that previously was passed to `pinned`.) The iterator p is invalidated by this operation, so you must not use it again once `unpin` returns. (Note that the `Cache` implementation ensures that updates to the map never invalidate iterators to existing entries in the map; `unpin` invalidates only the iterator for the removed entry.)

```
void clear();
```

This function removes all entries in the map.

```
size_t size() const;
```

This function returns the number of entries in the map.

See Also

- Servant Evictors
- The C++ Handle Template
- The C++ Shared and SimpleShared Classes
- Background Save Evictor