org.jax.mgi.shr.cache Design Document

Author: Mike Walker, Jon Beal

Created: March 19, 2003

Last Modified: September 16, 2004 23:36

1 Purpose of Document

This document describes the classes belonging to the org.jax.mgi.shr.cache package and provides source code examples for common usage patterns.

2 Introduction

The overall intent of the org.jax.mgi.shr.cache package is to provide a frameworks for caching query result sets within loader applications and for caching web pages within WI applications. This document will take a two part approach to discussing this package. The first part will discuss query caching and the second part will discuss web page caching.

This product integrates with the org.jax.mgi.shr.exception package, the org.jax.mgi.shr.log package and the org.jax.mgi.shr.dbutils package.

This product was coded and tested on Java 1.4 JDK.

3 ResultSet Caching

3.1 Introduction

Query caching involves the caching in memory of query results such that performance gains can be achieved by searching in-memory caches instead of making database calls across a network. The frameworks classes are responsible for creating, maintaining and querying the cache. Two caching strategies are provided, lazy and full. The primary use of this frameworks is in the development of lookup classes which provide lookup methods for obtaining objects from the database. These lookup classes extend the frameworks classes. The lookups provide the sql with which to access the database and the frameworks base classes provide the caching functionality.

3.2 Overview of Classes

3.2.1 KeyValue

The **KeyValue** class is used as a data value object for storing the keys and values as required for caching operations. All internal caches are implementations of the Map interface and therefore all put and get will require keys and values. When extending the frameworks to create lookup classes, one of the responsibilities faced is to provide a **RowDataInterpreter** class which parses a row (or multiple ordered rows) of query results and creates a single **KeyValue** object. For more

information on the **RowDataInterpreter** class, see the org.jax.mgi.shr.dbutils system documentation. This is the way the caching frameworks obtains data from the query results for maintaining the cache.

3.2.2 RowDataCacheHandler

The **RowDataCacheHandler** is an abstract class which works together with the **RowDataCacheStrategy** class to handle the caching and retrieval of database query results. It has a reference to the cache and provides sql strings for initializing the cache and for adding new entries to the cache. It calls upon the **RowDataCacheStrategy** for performing lookups in the cache.

3.2.3 RowDataCacheStrategy

The **RowDataCacheStrategy** is an abstract class which works together with the **RowDataCacheHandler** class to perform caching and retrieval of database query results. It provides two abstract methods, lookup(Object, Map) and init(Map) which together defines the behavior of any caching strategy.

3.2.4 FullCacheStrategy

The **FullCacheStrategy** class extends the **RowDataCacheStrategy** class and provides a basic full caching strategy. The cache is fully initialized. A lookup then will look for an item within the cache and if it is not found it will return null without looking for it in the database.

3.2.5 LazyCacheStrategy

The **LazyCacheStrategy** class extends the **RowDataCacheStrategy** class and provides a basic lazy caching strategy. The data is first looked up in the cache and if it is not found it is added to the cache after a database lookup. These caches are not fully initialized in advance as in the FullCacheStrategy. They can however be partially initialized to obtain a subset of data that may be considered more relevant or more likely to be searched on. Any data not found within the partially initialized cache will be added to the cache by database query.

3.2.6 CachedLookup

The **CachedLookup** class extends the **RowDataCacheHandler** class and additionally provides two basic lookup methods, one which throws an exception if the data was not found and another which returns null if the data was not found. The subclass can choose which lookup method to use. This is an abstract class. It is intended to be extended by concrete lookup classes which desire pluggable caching strategies at runtime (that is, they can do either lazy or full caching).

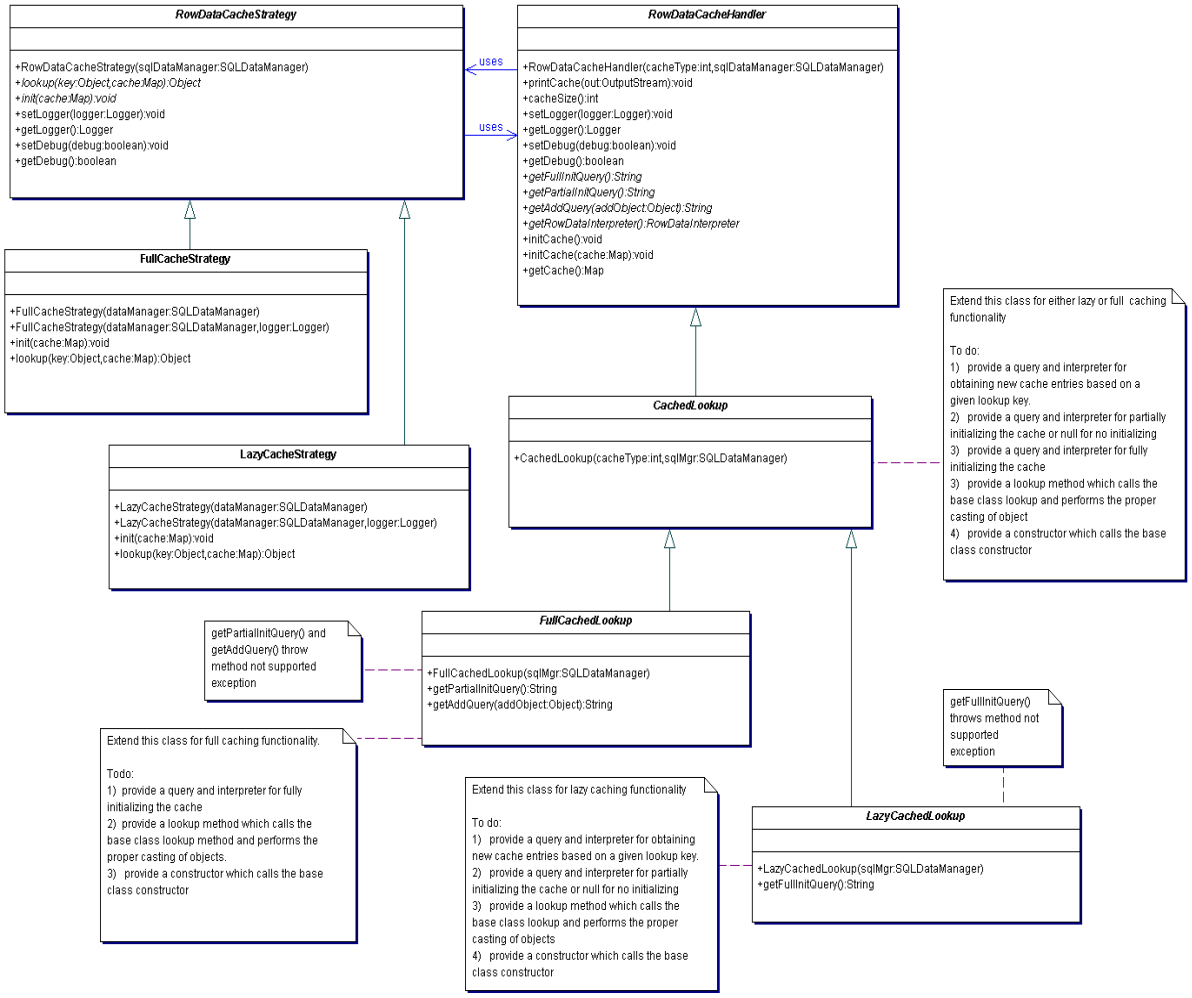
3.2.7 LazyCachedLookup

The **LazyCachedLookup** is an extension of the **CachedLookup** class which performs a lazy caching strategy only. This class is an abstract class. It is intended to be extended by concrete lookup classes which only use lazy caching.

3.2.8 FullCachedLookup

The **FullCachedLookup** is an extension of the **CachedLookup** class which performs a full caching strategy only. This class is an abstract class. It is intended to be extended by concrete lookup classes which only use full caching.

3.3 Class Diagram



3.4 About Query Caching Strategies

A caching strategy can be either lazy or full. On full caching, an initialization query is performed and all the results are put into a cache. When a lookup call is made, the full cache strategy will look up the key within the cache and if it is not found it will return a null. With lazy caching, either no initialization or a partial initialization is performed. When a lookup call is made, the lazy cache strategy will first look up the data in the cache. If it is not found, then it will subsequently

look up the data in the database. If it is found in the database, then it is added to the cache. If it is not found in the database then a null is returned.

Two classes have been created to implement these strategies. They are the **LazyCacheStrategy** and the **FullCacheStrategy**. Instances of these classes are created by the **RowDataCacheHandler** and are called upon by the handler when performing a lookup. Each of the strategy classes have a lookup method which take the key object to look up (which can be any java object) and the cache to look in. The **RowDataCacheHandler** constructor accepts which type of strategy to use. This is designated by constants defined within the **CacheConstants** class.

3.5 About Query Caching Handlers

The **RowDataCacheHandler** class contains the cache and provides methods for initializing, printing and sizing the cache. It is an abstract class and it is intended to be extended by a concrete lookup class. The abstract methods are designed for providing the sql queries required for caching initialization and for looking up data by key within the cache, and for providing a **RowDataInterpreter** class for interpreting query results (see the org.jax.mgi.shr.dbutils package for more information on the **RowDataInterpreter**). These methods are called by the **RowDataCacheStrategy** class. To summarize the interaction between the cache handler and the cache strategy, the **RowDataCacheHandler** calls the **RowDataCacheStrategy** to perform lookups, which calls the **RowDataCacheHandler** to get the sql queries it needs to do database queries and to get the **RowDataInterpreter** for parsing the query results.

The following is a list of abstract method definitions from the **RowDataCacheHandler** class:

- public abstract String getFullInitQuery()
- public abstract String getPartialInitQuery()
- public abstract String getAddQuery(Object)
- public abstract RowDataInterpreter getRowDataInterpreter()

The getFullInitQuery() method is called by the **FullCacheStrategy** class to get the full initialization query for the cache. The getPartialInitQuery() method is called by the **LazyCacheStrategy** to get a query with which to partially initialize the cache. The getAddQuery() method is called by the **LazyCacheStrategy** class when a lookup in the cache fails to return an item and a database lookup is required. If the concrete lookup class is going to support both lazy and full caching strategies, then all of the above methods will need to be implemented. If the lookup class is only going to support full caching then only the getFullInitQuery() and getRowDataInterpreter() need to be implemented. The remaining methods can be written to throw a runtime exception. If the lookup class is only going to support lazy caching, then only the methods getPartialInitQuery(), getAddQuery() and getRowDataInterpreter() need to be implemented.

Three convenient classes have been designed that follow the above approach and make the development of lookup classes easier. These classes are **CachedLookup**, **LazyCachedLookup** and **FullCachedLookup**. The **LazyCachedLookup** class throws a runtime exception on those methods reserved for full caching and leaves the remaining methods defined as abstract. The **FullCachedLookup** class throws a runtime exception on those methods reserved for lazy caching and

leaves the remaining methods defined as abstract. The **CachedLookup** can go either way with the caching strategy and may determine which way to go during instantiation.

One important aspect for the strategy classes is to be able to parse query results in a way to determine the keys and values that are required for caching. This is done with the use of the **RowDataInterpreter**. This class is discussed in details in the org.jax.mgi.shr.dbutils package documentation, but briefly, it is the pluggable class which is used to parse a row of data. It gets plugged into the **ResultsNavigator** class which calls upon it when it needs to create a data object based on a row of data. To facilitate the putting and getting of data in the cache, the **RowDataCacheStrategy** class expects to be able to interpret query results as a **KeyValue** object. The **KeyValue** object is simply a class which provides accessor methods for a key object and a value object. There are no constraints on these objects. They can be any instance of the java Object class. The lookup class will have to provide a **RowDataInterpreter** which creates a **KeyValue** object based on the current row during the call to the interpret(RowReference) method. The RowDataCacheStrategy class will check the return type from the **RowDataInterpreter** and will throw an exception if it is not a **KeyValue** object. See example 1 which illustrates using an inner class to return a **RowDataInterpreter** in the getRowDataInterpreter() method.

EXAMPLE 1. getRowDataInterpreter() method

```
public RowDataInterpreter getRowDataInterpreter() {
    class Interpreter implements RowDataInterpreter
    {
        public Object interpret (RowReference row)
            throws DBException
        {
            return new KeyValue(row.getString(1),
                                row.getInt(2));
        }
    }
    return new Interpreter();
}
```

To have a look at examples which use this framework, look at the lib_java_dbsmgd product and, more specifically, at the org.jax.mgi.dbs.mgd.lookup package. There you will find good examples in **OrganismKeyLookup** which extends **FullCachedLookup** and **VocabKeyLookup** which extends **CachedLookup**.

4 Lookup Example

As an example we will consider a practical application of the frameworks using the MGD database. This example will provide a full cache for logical databases from the ACC_LogicalDB table.

To get full caching behavior from the frameworks, the implementation would extend the FullCachedLookup class. This entails providing a constructor which calls the super constructor, providing a lookup method and implementing the required abstract methods. See the example below.

EXAMPLE 2. implementation of a FullCachedLookup

```
public class LogicalDBLookup extends FullCachedLookup
{
    public LogicalDBLookup ()
        throws CacheException,
               ConfigException, DBException
    {
        super(
           SQLDataManagerFactory.getShared(
             SchemaConstants.MGD));
    }

    public Integer lookup (String logicalDB)
        throws KeyNotFoundException,
               DBException, CacheException
    {
        return (Integer)super.lookup(logicalDB);
    }

    public String getFullInitQuery ()
    {
        return new String(
          "SELECT name, _LogicalDB_key " +
          "FROM ACC_LogicalDB");
    }

    public RowDataInterpreter getRowDataInterpreter()
    {
        class Interpreter implements RowDataInterpreter
        {
            public Object interpret (RowReference row)
                throws DBException
            {
                return new KeyValue(row.getString(1),
                                    row.getInt(2));
            }
        }
        return new Interpreter();
    }
}
```

5 WI Caching

5.1 Overview of Classes

Rather than present a detailed API for each class, we only present an overview here. The details are covered in the javadocs for each class, so we do not replicate them here.

5.1.1 TextCacheExceptionFactory

The `TextCacheExceptionFactory` is a subclass of `org.jax.mgi.shr.exception.ExceptionFactory` and is used to raise any necessary exceptions for the `*TextCache` classes. All exceptions raised are `org.jax.mgi.shr.exception.MGIException` objects, though they will have different attributes to reflect the cause of the exception.

5.1.2 TextCache interface

`TextCache` is the fundamental interface on which all the other `String`-caching classes are built. To cache a `String`, it must be identified as a certain "`textType`" and must be associated with an identifier that is unique within its `textType`. The notion of a `textType` allows us to store multiple types of `String`s within the same cache.

Operations included in this interface are:

- clearing an entry, all entries for a `textType`, or all entries from the entire cache
- asking for the age of an entry
- getting the hit rate (hits divided by requests, as a percentage) for a particular `textType` or for the entire cache
- clearing the hit rate for a particular `textType` or for the entire cache
- retrieving an entry as a `String`, a `StringBuffer`, or a `Collection` of `String`s (splitting on newline characters)
- get a count of hits for a `textType` or for the whole cache
- get a count of misses for a `textType` or for the whole cache
- adding an entry to the cache

5.1.3 AbstractTextCache

`AbstractTextCache` is an abstract class which implements many of the methods in `TextCache`, making it easier for subclasses to implement the interface.

It handles completely the tracking and computing of hits, misses, and hit rates. `AbstractTextCache` also implements all of the public methods for retrieving and adding cache entries; they are implemented in terms of protected abstract methods `primitiveGet()` and `primitivePut()` which will need to be defined in subclasses.

5.1.4 DiskTextCache

`DiskTextCache` is a subclass of `AbstractTextCache` which stores all cache entries on disk in a specified directory in the file system.

Example 1: Checking for an item in the cache

```
try
{
    TextCache cache = new DiskTextCache ("/tmp/cache");
    String foo = cache.get ("marker", "10603");
    if (foo != null)
    {
        // found cache entry
    }
}
catch (MGIException exc)
{
    System.out.println (exc.toString());
}
```

5.1.5 FastTextCache

`FastTextCache` is a subclass of `AbstractTextCache` which implements a two-level cache for improved performance. A `FastTextCache` uses a `DiskTextCache` internally for primary storage of all cache entries. Once we have a successful cache hit when retrieving an entry, we copy that entry into an in-memory storage area so that it may be retrieved even faster for future requests. This scheme works well for situations where requests are not evenly distributed -- where some entries are requested more frequently than others.

Example 2: Adding an item to the cache

```
try
{
    TextCache cache = new FastTextCache ("/tmp/cache");
    String foo = "HTML code for marker detail for Kit";
    cache.put ("marker", "10603", foo);
}
catch (MGIException exc)
{
    System.out.println (exc.toString());
}
```

5.1.6 ExpiringObjectCache

The `ExpiringObjectCache` provides a mechanism for temporarily caching generated objects and associating them with a `String` name. For example, there could be an `Object` that takes a considerable amount of time or resources to construct, but which changes only rarely. This `Object` could be generated, then stored in an `ExpiringObjectCache` with a certain timeout value indicating how long that `Object` should be considered valid. This is especially useful for long-running systems like web applications.

Separate `ExpiringObjectCaches` may be constructed, or the class also provides a single instance which may be shared easily by using the `getSharedCache()` method to retrieve it.

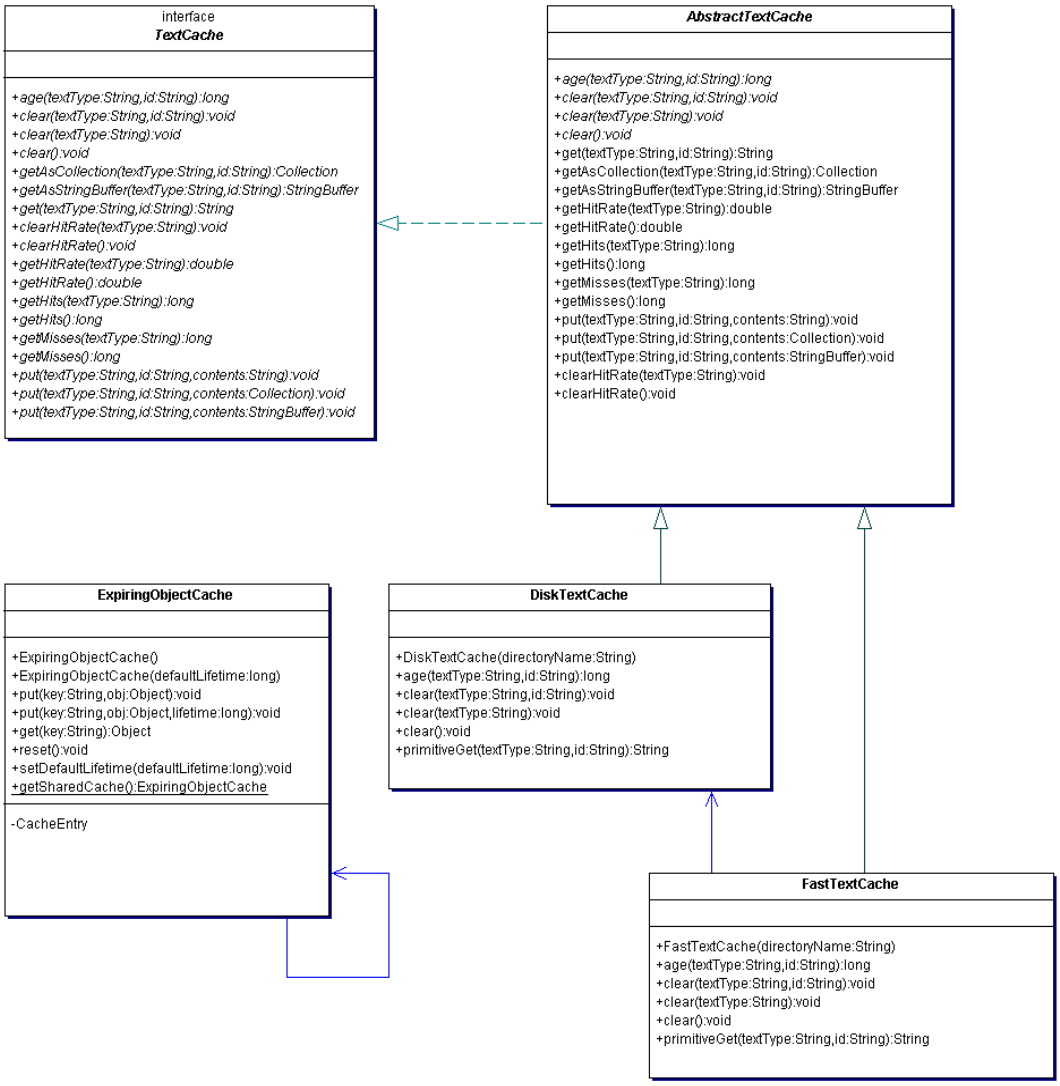
Each `ExpiringObjectCache` has a default timeout value for any `Object` added to it. There is a standard ten minute timeout if a particular one is not specified when instantiating the new `ExpiringObjectCache`. An `ExpiringObjectCache`'s default timeout may also be set during runtime using the `setDefaultLifetime()` method. When adding an `Object` to the cache, a timeout value particular to that `Object` may be set as well, if it needs to differ from the default.

When retrieving the `Object` associated with a particular `String`, the `get()` method will return null if the `Object`'s timeout has expired or if that association has not been added to the cache yet. In addition to the methods to add `Objects` to the cache and to retrieve them, there is also a method to clear all `Objects` from it, effectively forcing their timeouts to expire.

Example 3: use of an ExpiringObjectCache

```
// get a new cache with a default timeout of one hour
// (60 minutes * 60 seconds/minute = 3600 seconds)
ExpiringObjectCache cache = new ExpiringObjectCache (3600);

// construct objects obj1 and obj2 somehow (by hitting the
// database or by retrieving something over the web)
...

// add obj1 to the cache, relying on the default timeout
cache.put ("myObject1", obj1);

// add obj2 to the cache, but ensure that it times out in
// only five minutes
cache.put ("myObject2", obj2, 5 * 60);

// time passes...

// how to retrieve an object from the cache...
myObj = cache.get ("myObject1");
if (myObj == null)
{
    // object has timed out, so regenerate it and add it
    // to the cache again
}

// how to reset the entire cache, timing out all objects
cache.reset();
```

5.2 Class Diagram



5.3 Configuration

The `DiskTextCache` and `FastTextCache` classes both require that a directory be specified in their constructors. These WI related classes in `org.jax.mgi.shr.cache` require no additional configuration.