org.jax.mgi.shr.dbutils.bcp Document Design

Author: Mike Walker

Created: March 20, 2003

Last Modified: September 16, 2004 23:51

1 Purpose of Document

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

2 Introduction

The purpose of this package is to encapsulate the functionality of the Sybase bcp utility for bulk copying data into the database, but it has been extended over time to be compatible with other databases such as MySQL and Oracle. In these cases it is not the actual bcp command which is being run but some other similar mechanism.

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

This class was developed and tested with Java1.4.

3 Configuration

Configuration is accomplished by use of the org.jax.mgi.shr.config package. The configuration classes pertaining to the bcp package are **BCPManagerCfg** and **BCPWriterCfg**. They are located directly in the config package as a means to get access to the protected methods from the configuration classes. The **BCPManagerCfg** is used to configure the **BCPManager** class. Details of how this is done can be found in the config package documentation, but in brief, the configuration parameters are read in from configuration files and java system properties. The configuration file names are indicated on the java command line by use of the CONFIG system property (see example 1). The configurator class, **BCPManagerCfg** looks up the values for these parameters and provide accessor methods for them (see Example 2). If you have more than one object that needs different configurations, for example, two BCPManagerCfg instances need to be configured individually, then you can prefix the base parameter names in the configuration file with an additional string of choice. An example configuration file is shown in Example 3 where the prefixes ‘RADAR’ and ‘MGD’ are used. That prefix string can then be passed to a **BCPManagerCfg** object in its constructor, forcing it to lookup those parameters prefixed by that string first and, if not found, search on the un-prefixed string secondly (see Example 4).

EXAMPLE 1. setting the configuration

```
java -DCONFIG=configfile -DPARM1=OVERRIDE -DPARM2=OVERRIDE2 <app>
```

EXAMPLE 2. using configuration objects

```
// the constructor will cause all configuration files and java
// properties to be read for the known parameters

BCPManagerCfg bcpConfig = new BCPManagerCfg();

// values are now accessable

Boolean okToDropIndexes = bcpConfig.getOkToDropIndexes();
```

EXAMPLE 3. configuration using prefixes

```
# radar values
RADAR_BCP_OK_TO_DROP_INDEXES=true
RADAR_BCP_OK_TO_TRUNCATE_LOG=true

# mgd values
MGD_BCP_OK_TO_DROP_INDEXES=false
MGD_BCP_OK_TO_TRUNCATE_LOG=true
```

EXAMPLE 4. using configuration objects with prefixing

```
// the constructor will cause all configuration files and java
// properties to be read for the known parameters prefixed by
// the given string

BCPManagerCfg radarConfig = new BCPManagerCfg('RADAR');
BCPManagerCfg mgdConfig = new BCPManagerCfg('MGD');
```

The following list shows configuration parameters for setting aspects of the bcp process. All these parameters are initially read from the configuration by a **BCPManagerCfg** object, but can be changed by calling set methods on the **BCPManager** object.

BCP_PATH

The directory the bcp files will be created in. The default is the current runtime directory.

BCP_DELIMITER

The field delimiter for the bcp file. The default is tab. The words 'space' and 'tab' are allowed values to represent these delimiters.

BCP_OK_TO_OVERWRITE

Indicator of whether or not to overwrite existing bcp files. If this option is set to false then a new filename with a numbered prefix will be created. The default is false.

BCP_USE_TEMPFILE

Indicator of whether or not to use temporary files when creating bcp files. These files will be deleted automatically after executing the bcp command. The deafult is false.

BCP_PREVENT_EXECUTE

Indicator of whether or not to prevent executing the bcp command on call to the execute method. The default is false.

BCP_REMOVE_AFTER_EXECUTE

Indicator of whether or not to remove the bcp file after executing it. The default is false.

BCP_RECORD_STAMPING

Indicator of whether or not to automatically time stamp the record when writing to the bcp file. The default is false. When using DAO classes to create bcp files, then leave this option set to false since the DAO class performs record stamping separately.

BCP_TRUNCATE_LOG

Indicator of whether or not to truncate the database log before executing the bcp command.

BCP_AUTO_FLUSH

Indicator of whether or not to flush the bcp file after each record is written. The default is false.

BCP_TRUNCATE_TABLE

Indicator of whether or not to truncate the database table before executing the bcp command. The default is false.

BCP_DROP_INDEXES

Indicator of whether or not to drop indexes on the database table before executing the bcp command. When this option is set to true, the indexes will be recreated after executing the bcp command. The default is false.

BCP_DROP_TRIGGERS

Indicator of whether or not to drop triggers on the database table before executing the bcp command. When this option is set to true, the triggers will be recreated after executing the bcp command. The default is false.

BCP_PRESQL

Designates the sql commands (as a pipe separated list) that should get executed prior to executing the bcp command. For example “sql one|sql two|sql three”.

BCP_POSTSQL

Designates the sql commands (as a pipe separated list) that should get executed after executing the bcp command. For example “sql one|sql two|sql three”.

All boolean parameters in the configuration file can be represented by the following strings without regard for case: true, false, 0, 1, yes or no.

The other configuration parameters required are for configuring the database connection. These are **DBSERVER**, **DBNAME**, **DBUSER**, **DBPASSWORDFILE**, **DBURL**, **DBSCHEMADIR**. These parameters are discussed in org.jax.mgi.shr.dbutils package documentation.

The **BCPManager** will instantiate a **BCPManagerCfg** and read configuration parameters from it. You can also pass in a reference to a **BCPManagerCfg** through the constructor if you wanted to use prefixing as in Example 4. See examples of this in Example 5.

EXAMPLE 5. how the BCPManager integrates with configuration

```
// the default constructor will cause
// all configuration files and system properties to be read

BCPManager bcp = new BCPManager();

// you could also use prefixing of configuration variables

BCPManagerCfg primaryConfig = new BCPManagerCfg('PRIMARY');
BCPManagerCfg secondaryConfig = new BCPManagerCfg('SECONDARY');
BCPManager bcpPrimary = new BCPManager(primaryConfig);
BCPManager bcpSecondary = new BCPManager(secondaryConfig);
```

A **BCPWriter** inherits configuration from the parent **BCPManager** class. You can override these inherited values separately for each table by using a **BCPWriterCfg** constructed in a way to use prefixing and then set only the parameters in the configuration file that you want to override and prefix those parameters.

4 Overview of Classes

The **BCPManager** class is used to manage multiple bcp files for a given database connection and a set of bcp configuration parameters. It is used to obtain a **BCPWriter** object for a each bcp file. And it is used to execute the bcp command for all bcp files.

The **BCPWriter** class is used for writing and validating records for a bcp file. There is one instance for each bcp file and each instance is obtained from a **BCPManager**. It has a write() method which excepts a Vector. This Vector contains items which represent data to be written to a bcp file. All the items in the Vector will be converted to Strings and validated. The BCPWriter has an instance of a **Table** class which is used for validating the data.

The **Table** class represents a database table. It has column metadata and is used for validating column data prior to writing to the bcp file. It is internal to the **BCPWriter** class. You can get a reference to it by calling the getTable() method. The **Table** class requires access to a database which is obtained through a reference to a **SQLDataManager** (see Section 5)**.**

The **BCPManagerCfg** class is used for configuring a **BCPManager**. It reads a set of configuration files and java system properties, and provides lookup methods for parameters which pertain to a **BCPManager**.

The **BCPWriterCfg** class is used for configuring a **BCPWriter**. It reads a set of configuration files and java system properties, and provides lookup methods for parameters which pertain to a **BCPWriter**.

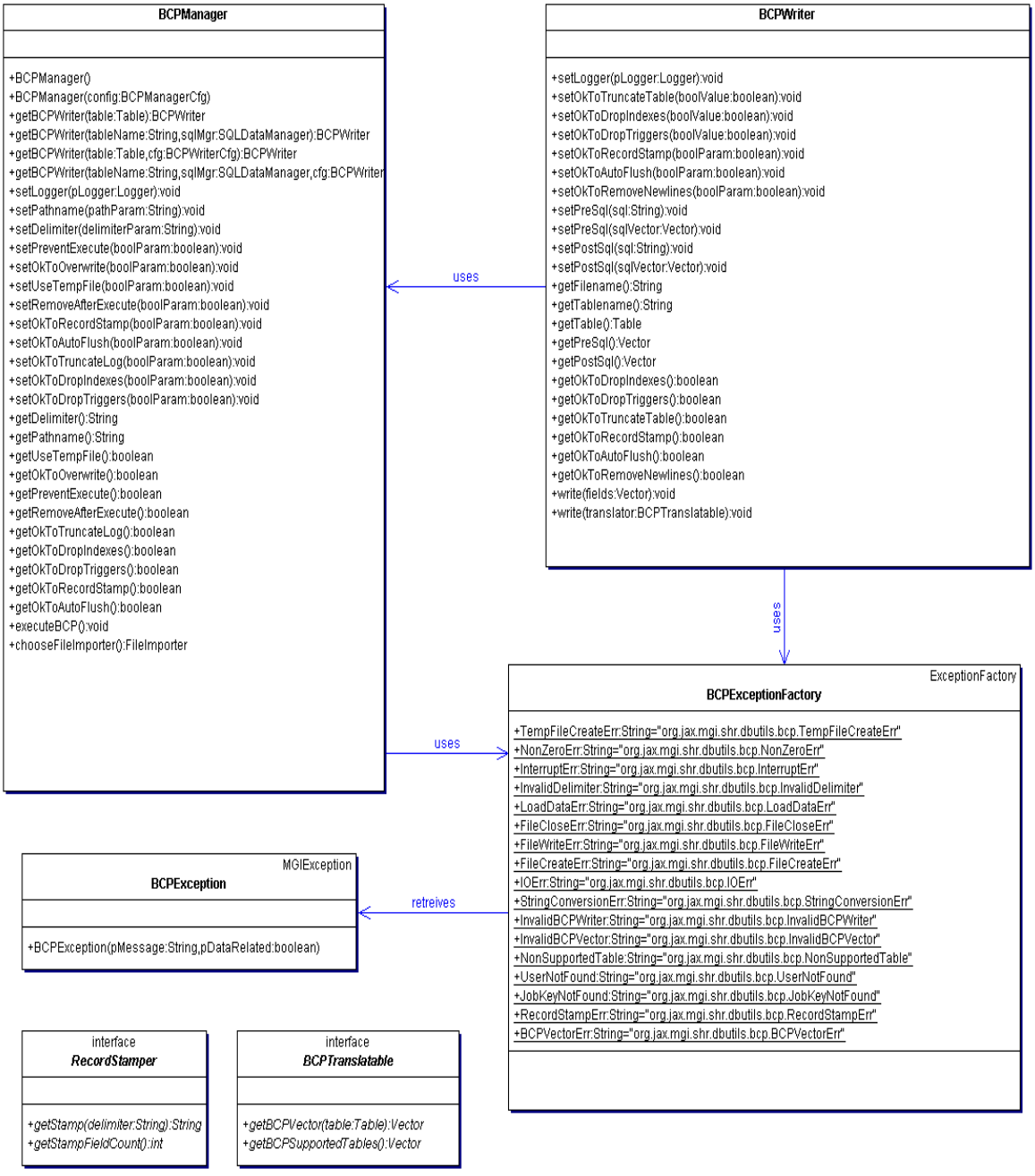
The **BCPTranslatable** interface is used by a **BCPWriter**. An object which implements this interface can be passed in as a parameter to the write() method of the **BCPWriter**. It has one method defined call getBCPVector(). The intent of that method is to return a Vector containing items which represent a record of data to be written to a bcp file. The order of the Vector items should be the order of fields expected by the bcp command. This is an alternative to calling the write() method and passing in a Vector.

The **BCPException** is a subclass of **MGIException** from the lib_java_exception product. It is used for exceptions occurring during bcp processing and is created by a **BCPExceptionFactory**.

The **BCPExceptionFactory** is a subclass of **ExceptionFactory** from the lib_java_exception product. It stores named instances of **BCPExceptions** and provides get methods for obtaining copies of them for exception handling.

The **RecordStamper** is the interface for a family of classes used for time stamping records in advance of writing them to a bcp file. There is an implementation for each timestamp format defined in the mgi databases. The **Table** class knows which type of record stamp is being used by the table and provides a getRecordStamp() method for obtaining the appropriate instance of a **RecordStamper**.

5 Class Diagram



6 Using a BCPManager

In order to write to a bcp file you first obtain a **BCPWriter** from the **BCPManager.** This is done by calling the getBCPWriter() method, passing in the target table name as a parameter. If a **BCP-**

Writer class requires different configuration settings from other **BCPWriters**, then you can use configuration prefixing as discussed in Section 3. A Table object is created internal to the **BCP-Writer**. It is used for validating records written to the bcp file before executing the file.

To write a bcp record, you would call the write() method of the **BCPWriter** and pass in a Vector of field values. Each item in the Vector should represent a column in the corresponding table and in the expected order. The vector can also contain other vectors. If this is the case, then the write() method behaves as if it was given multiple bcp lines to write for one method call.

To run the bcp command call the execute() method on the **BCPManager** object. If there are more than one bcp files, this will execute all of them in the order they were created (see example 6).

EXAMPLE 6. pseudo code for using a BCPManager and BCPWriter

```
BCPManager bcpManager = new BCPManager();
BCPWriter writer = bcp.getBCPWriter('tablename');
for each record of data
{
  create a vector v and add values

  call writer.write(v);
}
bcpManager.execute();
```

An alternative to passing in a Vector to the **BCPWriter** write() method is to pass in an object which implements the **BCPTranslatable** interface. The **BCPWriter** will be able to obtain the Vector itself by calling the getBCPVector() method on the **BCPTranslatable** interface. This interface is defined as follows:

```
public interface BCPTranslatable
{
  public Vector getBCPVector(Table table);
}
```

To implement this, you would write a class and provide the getBCPVector() method. In this method you are given the **Table** class for which the bcp vector is targeted for. You would create a Vector object and add the data items specific for that table. The Vector could contain Vectors instead of scalars, which would cause multiple bcp lines to be written. Once this method was implemented, this class would then be declared as implementing the BCPTranslateble interface using the implements clause (see example 7). You would then be able to pass this class to the BCPWriter as in Example 8.

EXAMPLE 7. pseudo code for implementing a BCPTranslatable interface

```
public class className implements BCPTranslatable
{
   // class variables declared here
   public Vector getBCPVector(Table table)
   {
```

```
      instantiate a new vector
      if ('target1' == table.getName())
         add items to vector for the target table1
      else if ('target2' == table.getName())
         add items to vector for the target table2
      continue else-if logic for each table this method supports

      return the new vector
   }

   more methods may follow

}
```

EXAMPLE 8. using a BCPWriter with a BCPTranslatable object

```
BCPManager bcpManager = new BCPManager();
BCPWriter writer = bcp.getBCPWriter('tablename');
for each bcp translatable object
{
  writer.write(object);
}
bcpManager.execute();
```

Data validation occurs each time you write to the bcp file. The **BCPWriter** makes a call to the validate() method of the associated **Table** object. This will validate each field of the Vector against each column within the database. If there is a validation error then a **DataException** is thrown. A validation error can occur from a number of problems. The number of items in the Vector may not match the number of expected columns; a string could be too big for the column size; a null value could be given for a non nullable field; a data type conversion may fail or a data type may not be supported. Only String, Integer, Float, Boolean, Timestamp are currently allowed data types (this is based on the supported types from the lib_java_types product). If you want to use primitive values, the **DataVector** class can be used instead of Vector (see lib_java_types). See table 1 for a summary of conversion rules.

Table 1: Data Type Conversion Rules

Database type	Allowed Java types	Allowed String values
Char	String	Any one character string
Varchar	All	Any string of right size
Integer	String, Integer	Any string which represents an Integer
Timestamp	String, Timestamp	yyyy-MM-dd HH:mm:ss.SSSSSSSSS yyyy/MM/dd HH:mm:ss.SSSSSSSSS
LongVarchar	All	Any string of right size

Table 1: Data Type Conversion Rules

Database type	Allowed Java types	Allowed String values
Float	String, Integer, Float	Any string which represents a Float
Bit	String, Boolean	0 or 1

The **BCPWriter** objects will execute SQL statements both previous to executing the bcp and post. To accomplish this, run the setPreSql() and setPostSql() methods or set these attributes in the configuration file.

If pre and post SQL statements have been set and the **BCPWriter** object has also been configured to drop indexes, truncate the transaction log and truncate the table, the order of events will be as follows: execute pre SQL; drop all indexes; truncate the table; execute the bcp command; truncate the transaction log; add the indexes; execute the post SQL statements. If the user needs to have the transaction log truncated before dropping the indexes, then this would be included as part of the pre SQL statements.