Namespace Migration

This page outlines the process to employ the namespace migration of Sqoop from *com.cloudera.sqoop* to *org.apache.sqoop*. The specific JIRA issues tracking this work are filed under SQOOP-369.

Note: The information provided here is general information. Any detail specific to a particular aspect of this migration should be noted in the relevant JIRA issue.

Overview

Migration of Sqoop source code from *com.cloudera.sqoop* to *org.apache.sqoop* is a prerequisite for releasing Sqoop under Apache Incubator. Considering that there is a lot of third party code that is developed on top of/to work with Sqoop, this migration is particularly risky for backward compatibility and thus requires careful handling. This document outlines the steps that seem reasonable for such migration.

If you are helping out with this migration and feel that the steps outlined here are not valid in some scenarios, please initiate the discussion on the developer mailing list and help update this document.

Migration

General Approach

When migrating a class from its previous namespace to the new, the key requirement to address is that any code written to the old class should still work at binary compatibility level. This also implies that such code should be able to recompile with the migrated code base **without any modifications**. In order to enable this, the general approach is as follows:

- Any old public/protected/default access level API should be preserved as is, but marked deprecated.
- Any logic that exists in this API should be migrated to the new namespace.

Note that API is not just method signatures but includes all aspects of implementation such as class hierarchies, type compatibility, static and non-static state etc. The following sections outline these steps as they apply to various API scenarios.

Simple Public Class

Consider the class:

```
package com.cloudera.sqoop.io;

public class NamedFifo {

   public NamedFifo(String pathname) {
     ...
   }

   public void create() {
     ...
   }
}
```

To migrate such a class, do the following:

- Create a new class in the new namespace and move all the APIs there.
- Modify the old class to mark it deprecated and remove all APIs from there.
- Modify the old class to make sure that the constructors remain the same and delegate to the corresponding super constructors.
- When referencing the new class from the old class, please use the fully qualified class name.

Here is the outcome:

```
// Old class
package com.cloudera.sqoop.io;

/**
 * @deprecated use org.apache.sqoop.io.NamedFifo instead.
 * @see org.apache.sqoop.io.NamedFifo
 */
public class NamedFifo extends org.apache.sqoop.io.NamedFifo {

   public NamedFifo(String pathname) {
     super(pathname);
   }
}

// New class
package org.apache.sqoop.io;

public class NamedFifo {
 public NamedFifo(String pathname) {
     ...
   }

   public void create() {
     ...
   }
}
```

Simple Public Class with Static Constants

Consider the following class:

```
package com.cloudera.sqoop.lib;

import java.io.Closeable;

public class LargeObjectLoader implements Closeable {
  public static final long DEFAULT_MAX_LOB_LENGTH = 16 * 1024 * 1024;

  @Override
  public void close() throws IOException {
    ...
  }
}
```

To migrate this, do the following:

- Migrate as per the directions of migrating Simple Public Class (above).
- Create the same constants in the new class.
- Reference the new class constants from the old class to preserve the static API.

```
// Old class
package com.cloudera.sqoop.lib;

/**
 * @deprecated use org.apache.sqoop.lib.LargeObjectLoader instead.
 * @see org.apache.sqoop.lib.LargeObjectLoader
 */
public class LargeObjectLoader extends org.apache.sqoop.lib.LargeObjectLoader {
  public static final long DEFAULT_MAX_LOB_LENGTH =
          org.apache.sqoop.lib.LargeObjectLoader.DEFAULT_MAX_LOB_LENGTH;

  ...
}

// New class
package org.apache.sqoop.lib;

import java.io.Closeable;

public class LargeObjectLoader implements Closeable {
  public static final long DEFAULT_MAX_LOB_LENGTH = 16 * 1024 * 1024;

  @Override
  public void close() throws IOException {
    ...
  }
}
```

Simple Utility Class

A utility class is a simple class that has a private constructor and all methods as statics. Here is an example:

```
package com.cloudera.sqoop.lib;

import ...;

public final class BigDecimalSerializer {

  private BigDecimalSerializer() { }

  static final BigInteger LONG_MAX_AS_BIGINT = 16 * 1024 * 1024;

  ...

  public static void write(BigDecimal d, DataOutput out) throws IOException {
    ...
  }
```

To migrate this class, do the following:

- Create a static class with the same API and implementation in the new namespace.
- Update the old class so that it delegates all invocations to the new class.

Here is how it will look like:

```
// New class
package org.apache.sqoop.lib;

import ...;

public final class BigDecimalSerializer {

  private BigDecimalSerializer() { }

  public static final BigInteger LONG_MAX_AS_BIGINT = 16 * 1024 * 1024;

  ...

  public static void write(BigDecimal d, DataOutput out) throws IOException {
    ...
  }
}


// Old class
/**
 * @deprecated use org.apache.sqoop.lib.BigDecimalSerializer instead.
 * @see org.apache.sqoop.lib.BigDecimalSerializer
 */
public final class BigDecimalSerializer {

  private BigDecimalSerializer() { }

  static final BigInteger LONG_MAX_AS_BIGINT =
      org.apache.sqoop.lib.BigDecimalSerializer.LONG_MAX_AS_BIGINT;

  public static void write(BigDecimal d, DataOutput out) throws IOException {
    org.apache.sqoop.lib.BigDecimalSerializer.write(d, out);
  }
  ...
}
```

Concrete Singletons

Concrete singletons are classes with static state that is bound to concrete implementation. Consider the class below:

```
package com.cloudera.sqoop.io;

import ...

public final class LobReaderCache {

  private Map<Path, LobFile.Reader> readerMap;

  private LobReaderCache() {
    this.readerMap = new TreeMap<Path, LobFile.Reader>();
  }

  private static final LobReaderCache CACHE;
  static {
    CACHE = new LobReaderCache();
  }

  public static LobReaderCache getCache() {
    return CACHE;
  }

  public static Path qualify(Path path, Configuration conf)
      throws IOException {
    ...
  }

  public LobFile.Reader get(Path path, Configuration conf)
      throws IOException {
    ...
  }
}
```

In such cases you can only partially migrate it the class to the new namespace. Specifically, the non-static part can be migrated using inheritance but the static/singleton aspect will remain in the old class. The steps involved are the following:

- Migrate the non-static logic to a new base class in the new namespace.
- Retain the singleton logic in the old class.

The outcome will be something like the following:

```
// Old class
package com.cloudera.sqoop.io;

import ...

/**
 * @deprecated use org.apache.sqoop.io.LobReaderCache instead.
 * @see org.apache.sqoop.io.LobReaderCache
 */
public final class LobReaderCache extends org.apache.sqoop.io.LobReaderCache {

  public static final Log LOG = org.apache.sqoop.io.LobReaderCache.LOG;

  private static final LobReaderCache CACHE;
  static {
    CACHE = new LobReaderCache();
  }

  public static LobReaderCache getCache() {
    return CACHE;
  }

  public static Path qualify(Path path, Configuration conf)
      throws IOException {
    return org.apache.sqoop.io.LobReaderCache.qualify(path, conf);
  }

  // non-static API all migrated to base class
}

// New class
package org.apache.sqoop.io;

import ...

public class LobReaderCache {

  private Map<Path, LobFile.Reader> readerMap;

  protected LobReaderCache() {
    this.readerMap = new TreeMap<Path, LobFile.Reader>();
  }

  public LobFile.Reader get(Path path, Configuration conf)
      throws IOException {
    ...
  }

  public static Path qualify(Path path, Configuration conf)
      throws IOException {
    ...
  }
}
```

Migrating Class Hierarchies

When migrating class hierarchies, you must create a parallel class-hierarchy in the new namespace using every migrated class. However, the important aspect of this migration is that in order to preserve the type-compatibility, you will have to have the new namespace classes inherit from their old super classes. Consider the following two class hierarchy:

```
// Class 1
package com.cloudera.sqoop.lib;

import ...

public abstract class LobRef<DATATYPE, CONTAINERTYPE, ACCESSORTYPE>
    implements Closeable, Writable {
  ...
}

// Class 2
package com.cloudera.sqoop.lib;

import ...

public class ClobRef extends LobRef<String, String, Reader> {
  ...
}
```

To migrate this over to new namespace do the following:

- Start with the lowest level class and create its equivalent in the new namespace.
- The class thus created in the new namespace should inherit from the old class's super class.
- Repeat this process for every ancestor class in the hierarchy.

The outcome for the above example will be as follows:

```
// Old class 1
package com.cloudera.sqoop.lib;

import ...
/**
 * @deprecated use org.apache.sqoop.lib.LobRef instead.
 * @see org.apache.sqoop.lib.LobRef
 */
public abstract class LobRef<DATATYPE, CONTAINERTYPE, ACCESSORTYPE>
    extends org.apache.sqoop.lib.LobRef<DATATYPE, CONTAINERTYPE, ACCESSORTYPE> {
  ...
}

// Old class 2
package com.cloudera.sqoop.lib;

/**
 * @deprecated use org.apache.sqoop.lib.ClobRef instead.
 * @see org.apache.sqoop.lib.ClobRef
 */
public class ClobRef extends org.apache.sqoop.lib.ClobRef {
  ...
}


// New class 1
package org.apache.sqoop.lib;

import ...
public abstract class LobRef<DATATYPE, CONTAINERTYPE, ACCESSORTYPE>
    implements Closeable, Writable {
  ...
}

// New class 2
package org.apache.sqoop.lib;

import ...

public class ClobRef
  extends com.cloudera.sqoop.lib.LobRef<String, String, Reader> {
  ...
}
```

Doing this will ensure that the type compatibility is preserved for all instances of the old classes with respect to their type hierarchies.