Ant2 File/FileSet/Culler Proposal

By: David Rees (dave@ubiqsoft.com)

1 Overview

The following gives a proposal for how Files and FileSets could be handled in Ant. Generally, it is intended to support “Files” and sets of these files from all walks of life including directory, ftp, http and VisualAge packages/classes (which make a good test case). It also explain the use of cullers for filtering sets of files.

Not described yet is Mapper functionality which should be in the final design.

2 Ant2 Design Assumptions

This proposal depends on a few features being available in Ant2. I also feel they are good ideas in general.

2.1 Element Sub-Types

Ant2 needs to provide support for the transparent use of a sub-type where a type is expected. The methods of a element should not have to be extended just to support new sub-types.

This is needed for this design to support addition of new AntFiles and other items without having to change the many elements that use them.

For purposes of the below design I will assume that if a element accepts A (addA() or seta()) it will accept subclasses of A.

2.2 Element Creators

Ant2 needs to extend the framework of IntrospectionHelper to support creators for types. In general, a creator is a static method or a Factory class that is used to create the instance of a type. This allows the actually class created to be a sub-type of the class that the IntrospectionHelper expects. Note that constructors could be still be used as a fallback if a type does not support the creator API.

This is needed for this proposal to support the automatic creation of an AntFile based on the string passed.

I will assume that if a element class A has the static method create(String) that IntrospectionHelper will call it with the attribute’s string.

2.3 Type Registry

Ant2 needs to support a registry of types and elements. This registry needs to support inheritance.

This is needed for this proposal to support the automatic creation of a AntFile based on the string passed.

I will assume that a type is somehow aware of his sub-types.

2.4 Finalization/Handle Registry

Ant2 needs to support a registry for handles that need to be finalized at the end of a task execution.

This needed for this proposal to close FTP, File, HTTP and other handles that are opened for AntFiles.

3 Ant2 Design Possibilities

4

There are a few Ant2 design possibilities that I think could go either way. Depending on their inclusion this design could be altered as described.

4.1 Auto-Inclusion of Sub-Elements

Some Ant1 types include their sub-element and support the sub-elements attributes and sub-elements in their own API (FileSet in Copy). Personally I would rather force people to do extra typing, but if we did want to support this then I would suggest we support auto-inclusion where a element can indicate it auto-includes a sub-elements type and then Helper would do all the work. As a possible example includeType() instead of add/createType.

If we did support this then backward compatibility would be easier for some of the design below. Specifically PatternCuller could automatically be included into FileSet so that you could still specify includes/excludes at that level.

4.2 Named Sub-Elements and Attribute/Sub-Element Equivalence

Generally it seems that in many cases an attribute and a sub-element are used for the same thing. Also, it is possible for the same type to be included for different “types”. The problem is that there is no way (that I know) to indicate the “name” of included element. For instance, in the following contrived example here is no way to indicate

```
<CopyAndDelete>
  <FileSet ... /> “files to copy”
  <FileSet ... /> “files to delete”
</CopyAndDelete>
```

I think that we need to support this somehow, at least by convention. Perhaps by using an “parentAttribute” attribute.

If there are named sub-elements, then in certain places in this design an attribute could be interchanged with a sub-element. This would allow plugging in of new types (since attributes don’t have the element name to indicate what type they are). Also, for the

BasicFileSet the source file could be listed as sub-element (where right now it has to be an attribute to not confuse it with other sub-elements).

5 AntFile

5.1 Overview

An AntFile represents a “file”. This file could be located on a file system, ftp site, web site or perhaps in a zip file. Specific sub-types handle the differences between file sources. New types can be added by developers to support new types of sources.

Like a java.io.File AntFile can also act as a “directory” which contains additional AntFiles. This can be interesting in cases where a given file can be both (Zip/Archives).

5.2 XML Usage

AntFiles are defined from strings. They are defined using an URL syntax with a few added extensions for additional “schemes” (the part before the colon). In addition, no schemes is assumed to be a local file. As examples:

```
“c:\temp\”
“ftp://ftp.ubiqsoft.com/file.txt”
“ftp://username@password:ftp.ubiqsoft.com/secretFile”
“http://httpd.apache.org/related_projects.html”
```

Zip and various archive formats are supported as well. For these a source archive needs be defined in place of the server portion of the URL. As examples:

```
“zip:<c:\temp\file.zip>”
“zip:<ftp://ftp.ubiqsoft.com/zipfile>/META-INF”
```

In the case of archives they can even been nested as shown below. Notice that if you just ignore the angled brackets the path reads “normally”.

```
“jar:<zip:<c:\temp\file.zip>/ejb.jar>/Main.java”
```

For specific uses AntFiles can also be defined as elements as follows:

```
<AntFile name=”c:\files\file.txt” />
```

Specific schemes may support additional attributes for their type (e.g. retries or PASV for FTP). I am not sure how to support this when an AntFile is defined as an attribute.

Note that different schemes may only support a subset of the AntFile functionality (this will be described in a chart somewhere once coding begins). In the case where unsupported functionality is used a BuildException will be raised.

5.3 Java Usage

The AntFile API is very similar to java.io.File. One difference is in file creation which is done using static creator methods rather than constructors so the actual returned object may be a sub-type of AntFile.

The methods support are listed below with basic descriptions of their functionality. Creation is described after that.

As described above, if functionality is not supported a BuildExcpetion will be raised (specifically a AntFileFunctionNotSupportedException).

5.3.1 Instance Methods supported

Arguments are only listed if they different or important for the description.

`canRead(), canWrite(), exists(),length(), lastModified(),isHidden()`

These return the attribute as it applies for their implementation.

`getName()`

Will return the basic (leaf) name of the AntFile.

`getPath(), toString(), getParent()`

These will return the appropriate URL string.

`getParentFile()`

Returns the parent AntFile or null if this is the top of the scheme.

`getInputStream(), getOutputStream(boolean append), getJavaFile()`

These methods are used to access or modify the actual File. The append flag for getOutputStream() can be set to true if you want to append to that file. getJavaFile() can be used to extract an actual java.io.File for those implementations where it is supported. Generally this is how copying and moving will be done (and is done now). Filters will be applied as normal on these streams.

`renameTo(), createNewFile(), delete(), mkdir(), mkdirs()`

These methods essentially do what they describe for their implementation (if it supports it).

`compareTo(AntFile), equals(), hashCode()`

These are all done based on the getPath() result.

`setLastModified(), setReadOnly()`

These will make the required change (if possible) for the indicated implementation.

`isDirectory(), isFile()`

This gets a little tricky. Generally I lean towards zip files being creatable as directories, but when accessed when navigating/scanning they are treated as files. This allows them to be used in a AntFile path, but they will be treated as directories when descending.

The other option is to make it an option, which I like but am not where sure to define it.

`list(), listFiles(), getChild(String)`

getChild() is used to create a child AntFile of the appropriate type for a given name. This method is used by the creator methods (see below).

`setPath(),setPath(AntFile, String), setPath(String, String)`

setPath will be used to allow reusing an AntFile instance when doing things like scanning. Note if this is supported then setters would need to copy their AntFiles when setting.

`iterator(), iterator(ScanFilter)`

This returns an Iterator walks the tree of files this file represents. In the case of a non-directory it will simply iterate once (itself). For most schemes this will simply be a directory based search. The Iterator constructor will also support accepting a "ScanFilter" that it will uses to prune the search as follows (for efficiency).

```
boolean shouldScanDirectory(AntFile)
```

5.3.2 Class Methods Supported (Creation)

Creation of AntFiles is done through a set of static creation methods on AntFile. These methods have similar signatures to the java.io.File constructors , but are methods so they can return the correct subclass instance as required. It is quite likely these will end up on a Factory class rather than being AntFile static methods.

`create(String)`

This method creates the correct instance of AntFile based on the scheme indicated in the String (as describe above).

`create(AntFile,String), create(String,String)`

Creates an instance of AntFile relative to the passed AntFile. The scheme will be the same as the passed AntFile. The second method acts as if an AntFile is created based on the first String and then passed to the first method.

`createFromArchive(AntFile archiveFile, String scheme, String relativePath)`

Only is valid for archive schemes. Creates an AntFile of the indicated scheme on the passed archiveFile.

5.3.3 File Methods not supported

`getAbsoluteFile(), getCanonicalFile(),getAbsolutePath(), getCanonicalPath(), isAbsolute(), toURL()`

I wonder if these can be supported because AntFile supports a richer set of “directories”. They possibly could be supported if there is real need.

5.3.4 Implementation specific methods

Specific implementations can support additional methods which can be accessed by casting down to that type. Obviously a cast exception will be raised if the AntFile is not that sub-type. Maybe getJavaFile() should go here?

5.3.5 File interface

As a convenience, the above API will actually be defined on the interface File in the org.apache.ant package which is sub-typed by AntFile. This allows easy transition of a class to using an AntFile by simply changing the import statement. However, since that can cause confusion it is also possible to simply use AntFile and rename as appropriate.

5.3.6 Contexts

There also needs to be some way to setup additional characteristics for a given scheme. Things like retries and such. My thinking is it could be either additional stuff in the Context or attributes when an AntFile is defined as a element.

5.4 Implementation

This is just a basic overview of the implementation design. Much of it can be derived from the APIs describe above. AntFiles are implemented as classes that implement the AntFile interface. Specific implementations are responsible for supporting the methods they can and raising an BuildException for those they can’t. Notes on the specific implementations are below.

5.4.1 Creation

When the String AntFile creators are called they will first strip off the scheme. They will then look through the Ant2 type registry for all AntFile sub-types and ask each one if it supports the indicated scheme. The first sub-type found that supports the scheme has its creator called with String and the result is turned (if the Ant2 registry supports ordering then this can be used to indicate which type is used).

All other creator methods are passed to the type of the passed AntFile.

5.4.2 File Scheme
The basic file scheme will simply wrap File instances. The AntFile object will have a File instance and will pass all method calls to it. Converting results back to AntFiles as needed. The creation methods will do the same.

5.4.3 FTP Scheme
My current thoughts are to simply use the ORO Net Components classes that are already used for the FTP component. AntFile objects will wrap FTPFile objects.

In addition, when a file's attributes are first accessed it will register a FTPClient in the Ant2 handle registry. This FTPClient will be reused across all FTPFile objects for this server.
Streams will be supported as expected.

5.4.4 HTTP Scheme
This method will wrap the Java URLConnection and URL classes. Hopefully there could be some caching to reduce connections.
Streams will be supported as expected.

5.4.5 Archive Schemes
Archive schemes represent directory structure inside a file (e.g. Zip). Generally these are internally created by first getting access to the archive file and then opening a read/write stream on it.

Initially the modification functionality will be pretty limited (no deleting or replacing files). Its possible to create a new archive with the changes and then moving it over the original one, but I haven't decided how much I like this.

For Archives is likely that a directory based scan will not work, so they will implement a iterator that just iterates through the archive. It would probably ignore the ScanFilter.

6 FileSets
A file set represents a group of files. This design extends this concept to include new types of FileSets. The original Ant1 FileSet functionality is described as a CullingFileSet in this design.

6.1 XML API
FileSet itself does not have an XML API other than it supports references. Instead, its subtypes have APIs based on their type.

6.1.1 CullingFileSet
CullingFileSet is what was called a FileSet in Ant1. The main changes are that it now uses Cullers for selecting it files rather than PatternSets. PatternSets are now used within PatternCuller (described below).

A CullingFileSet takes a dir attribute which is an AntFile string. This attribute can also be defined using an "named element" as described above.
In addition, CullingFileSet takes a set of Cullers that will be applied to the files in that directory to determine if they should be included. See Cullers below for more information on Cullers.
CullingFileSets are *not* ordered. The order in which they return their results is scheme dependent.

See Cullers below for examples.

6.1.2 BasicFileSet
A BasicFileSet simply represents an ordered set of files. Files can be defined in the attribute "includes" as a comma separated list of AntFiles. They can also be added as sub-elements. In addition, the attribute "includesfile" can be used indicate a file that lists files to be included.

BasicFileSet also supports a creator that takes a String. This allows it to be used as an attribute in a element if that element indicates it accepts BasicFileSets. The creator will treat the String as if were the "includes" element.

My expectation is that this FileSet can/will replace Path.

Examples:
```
<BasicFileSet includes="c:\temp\file.txt,
ftp://ftp.ubiqsoft.com/file.txt" />
<BasicFileSet>
  <AntFile name="c:\temp\file.txt>
  <AntFile name"ftp://ftp.ubiqsoft.com/file.txt" />
</BasicFileSet>
```

6.1.3 FileSetSet
A FileSetSet simply groups a set of FileSets together. It supports FileSet sub-elements. Ordering is preserved to the degree that its sub-elements support it.

6.2 Java API
FileSet will support a very simple API. Note that while it returns an array, the ordering is only deterministic for "ordered" FileSets as defined above.

`included(), filesIncluded(), dirsIncluded()`

Simply returns the indicated items as an array. The last two methods use the isDirectory() to select which ones are used.

`iterator()`

Returns an java.util.Iterator that can be used to get the files selected one at a time. Potentially each request may result in processing to determine the next file.

6.3 Implementation

The FileSet interface/class will support the API as described above.

6.3.1 CullingFileSet

A CullingFileSet simply gets the iterator of its dir and iterates over it. It applies its cullers as it goes and returns only those that are selected. It can also act as a ScanFilter for pruning the search.

The included() methods simply iterate through all the results and build an array before returning it.

6.3.2 BasicFileSet

Simply iterates through its list of files.

6.3.3 FileSetSet

Iterates through each of its contained FileSets in turn.

7 Cullers

7.1 Usage

Cullers determine the files returned for a CullingFileSet. Specific culler types support selecting files based on their attributes or perhaps contents.

An example says it best, the following selects all files that end in ".txt" and are read only.

```
<copy toDir="\tmp\toDir">
  <CullingFileSet dir="\tmp\fromDir">
    <include name="*.txt" />
    <FileAttributeCuller canWrite="no" />
  </CullingFileSet>
</copy>
```

For some schemes it is possible that certain culler attributes may not be valid. In these cases a BuildException is raised as described under AntFile.

7.2 Culler Types

Note that it easy to add additional cullers. See "developing cullers" for more information.

7.2.1 PatternCuller

A PatternCuller contains a set of PatternSet elements. It selects files based on if they support its patterns.

In addition, it supports pruning the search based on if a given directory could ever contain files that would match its patterns.

7.2.2 FileAttributeCuller

Supports selecting files based on their attributes. The default for all attributes is to ignore that attribute so a blank FileAttributeCuller will select all files.

7.2.2.1.1 Basic Attributes

Each attribute corresponds to the similar method on the Java File class.

`canRead`

true/yes, false/no, *ignore

`canWrite`

true/yes, false/no, *ignore

`exists`

true/yes, false/no, *ignore

`fileDir`

file, dir, *ignore

7.2.2.1.2 Date Attributes

For date attributes any String that can be read by Java Date or "ignore" is supported. This includes minutes, seconds and milliseconds.

`dateEquals`

Selects file if its date is the same to the millisecond as the indicated date

`dateNotEquals`

Selects file if its date is the different (to the millisecond) from the indicated date

`dateAfter`

Selects file if its date is after the indicated date

`dateBefore`

Selects file if its date is before the indicated date

7.2.2.1.3 Length Attributes

For length attributes, any long that can be read by Java Long or "ignore" is supported. The length is in bytes.

`lengthEqual`

Selects file if its length equals the indicated length.

`lengthNotEqual`

Selects file if its length does not equal the indicated length.

`lengthLessThan`

Selects file if its length is less than the indicated length.

`lengthGreaterThan`

Selects file if its length is greater than the indicated length.

7.2.2.1.4 Examples

Select only files:

```
<fileset dir="${from.dir}" >
  <fileattributeculler fileDir="file" />
</fileset>
```

Select files created since the start of the millennium:

```
<fileset dir="\tmp\fromDir">
  <include name="*.txt" />
  <FileAttributeCuller dateAfter="1/1/2001" />
</fileset>
```

Select non-empty files:

```
<fileset dir="\tmp\fromDir">
  <include name="*.txt" />
  <FileAttributeCuller lengthNotEqual="0" />
</fileset>
```

7.2.2.2 FileCompareCuller

Supports selecting files relative to another directory. The same set of attributes as FileAttributeCuller are supported except that they are applied to the file in the same relative position in compare directory. In addition, the value "culee" can be used for date/length attributes to compare the file to the file being selected.

7.2.2.2.1 Examples

Copies only those files that do not exist in the toDir:

```
<copy toDir="${to.dir}">
  <fileset dir="${from.dir}">
    <filecompareculler compareDir="${to.dir}" exists="no" />
  </fileset>
</copy>
```

Makes a clone of the fromDir in the toDir. It first deletes those files that do no exist in the fromDir. Then copies only those files that don't exist or have changed.

```
<mkdir dir="${to.dir}" />
<delete>
  <fileset dir="${to.dir}">
    <filecompareculler compareDir="${from.dir}" exists="no" />
  </fileset>
</delete>
<copy toDir="${to.dir}">
  <fileset dir="${from.dir}">
    <cullerset logic="or">
      <filecompareculler compareDir="${to.dir}" exists="no" />
      <filecompareculler
        compareDir="${to.dir}"
        dateNotEqual="cullee" />
    </cullerset>
  </fileset>
</copy>
```

7.2.2.3 CullerSet

A collection of Cullers that is also a Culler itself (they can be nested). CullerSet supports simple logic for how to combine the results of its contained cullers. Cullers are tested in the order they are listed. Short circuiting is used so some cullers may not be called for a given file (e.g. with "and" logic the first false nested culler will result in the CullerSet returning false immediately).

7.2.2.3.1 Attributes

`logic`

or*, and, xor
Used to indicate what logic is used to combine my nested cullers.

`not`

true/yes, false/no*
If "true/yes" the CullerSet's value is reversed.

7.2.2.3.2 * Examples

Selects all directories and only writable files:

```
<cullerset logic="or" >
  <fileattributeculler fileDir="dir" />
  <fileattributeculler fileDir="file" canWrite="yes" />
</cullerset>
```

7.2.2.4 Possible Cullers

ContainsFileCuller which will select files based on their contents using a RegExp.
a ByteCompareCuller that compares actual contents.

7.3 Developing Cullers

Additional FileCullers can be added easily by creating a class that implements the Culler API of shouldCull(AntFile).