The Java™ Tutorials

Trail: Essential Classes
Lesson: Basic I/O
Section: File I/O (Featuring NIO.2)

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Walking the File Tree

Do you need to create an application that will recursively visit all the files in a file tree? Perhaps you need to delete every `.class` file in a tree, or find every file that hasn't been accessed in the last year. You can do so with the `FileVisitor` interface.

This section covers the following:

The FileVisitor Interface

To walk a file tree, you first need to implement a `FileVisitor`. A `FileVisitor` specifies the required behavior at key points in the traversal process: when a file is visited, before a directory is accessed, after a directory is accessed, or when a failure occurs. The interface has four methods that correspond to these situations:

- `preVisitDirectory` – Invoked before a directory's entries are visited.
- `postVisitDirectory` – Invoked after all the entries in a directory are visited. If any errors are encountered, the specific exception is passed to the method.
- `visitFile` – Invoked on the file being visited. The file's `BasicFileAttributes` is passed to the method, or you can use the file attributes package to read a specific set of attributes. For example, you can choose to read the file's `DosFileAttributeView` to determine if the file has the "hidden" bit set.
- `visitFileFailed` – Invoked when the file cannot be accessed. The specific exception is passed to the method. You can choose whether to throw the exception, print it to the console or a log file, and so on.

If you don't need to implement all four of the `FileVisitor` methods, instead of implementing the `FileVisitor` interface, you can extend the `SimpleFileVisitor` class. This class, which implements the `FileVisitor` interface, visits all files in a tree and throws an `IOError` when an error is encountered. You can extend this class and override only the methods that you require.

Here is an example that extends `SimpleFileVisitor` to print all entries in a file tree. It prints the entry whether the entry is a regular file, a symbolic link, a directory, or some other "unspecified" type of file. It also prints the size, in bytes, of each file. Any exception that is encountered is printed to the console.

The `FileVisitor` methods are shown in bold:

```
import static java.nio.file.FileVisitResult.*;

public static class PrintFiles
    extends SimpleFileVisitor<Path> {

    // Print information about
    // each type of file.
    @Override
    public FileVisitResult visitFile(Path file,
                                     BasicFileAttributes attr) {
        if (attr.isSymbolicLink()) {
            System.out.format("Symbolic link: %s ", file);
        } else if (attr.isRegularFile()) {
            System.out.format("Regular file: %s ", file);
        } else {
            System.out.format("Other: %s ", file);
        }
```

```
            System.out.println("(" + attr.size() + "bytes)");
            return CONTINUE;
        }

        // Print each directory visited.
        @Override
        public FileVisitResult postVisitDirectory(Path dir,
                                                  IOException exc) {
            System.out.format("Directory: %s%n", dir);
            return CONTINUE;
        }

        // If there is some error accessing
        // the file, let the user know.
        // If you don't override this method
        // and an error occurs, an IOException
        // is thrown.
        @Override
        public FileVisitResult visitFileFailed(Path file,
                                               IOException exc) {
            System.err.println(exc);
            return CONTINUE;
        }
    }
```

Kickstarting the Process

Once you have implemented your `FileVisitor`, how do you initiate the file walk? There are two `walkFileTree` methods in the `Files` class.

- `walkFileTree(Path, FileVisitor)`
- `walkFileTree(Path, Set<FileVisitOption>, int, FileVisitor)`

The first method requires only a starting point and an instance of your `FileVisitor`. You can invoke the `PrintFiles` file visitor as follows:

```
Path startingDir = ...;
PrintFiles pf = new PrintFiles();
Files.walkFileTree(startingDir, pf);
```

The second `walkFileTree` method enables you to additionally specify a limit on the number of levels visited and a set of `FileVisitOption` enums. If you want to ensure that this method walks the entire file tree, you can specify `Integer.MAX_VALUE` for the maximum depth argument.

You can specify the `FileVisitOption` enum, `FOLLOW_LINKS`, which indicates that symbolic links should be followed.

This code snippet shows how the four-argument method can be invoked:

```
import static java.nio.file.FileVisitResult.*;

Path startingDir = ...;

EnumSet<FileVisitOption> opts = EnumSet.of(FOLLOW_LINKS);

Finder finder = new Finder(pattern);
Files.walkFileTree(startingDir, opts, Integer.MAX_VALUE, finder);
```

Considerations When Creating a FileVisitor

A file tree is walked depth first, but you cannot make any assumptions about the iteration order that subdirectories are visited.

If your program will be changing the file system, you need to carefully consider how you implement your `FileVisitor`.

For example, if you are writing a recursive delete, you first delete the files in a directory before deleting the directory itself. In this case, you delete the directory in `postVisitDirectory`.

If you are writing a recursive copy, you create the new directory in `preVisitDirectory` before attempting to copy the files to it (in `visitFiles`). If you want to preserve the attributes of the source directory (similar to the UNIX `cp -p` command), you need to do that *after* the files have been copied, in `postVisitDirectory`. The `Copy` example shows how to do this.

If you are writing a file search, you perform the comparison in the `visitFile` method. This method finds all the files that match your criteria, but it does not find the directories. If you want to find both files and directories, you must also perform the comparison in either the `preVisitDirectory` or `postVisitDirectory` method. The `Find` example shows how to do this.

You need to decide whether you want symbolic links to be followed. If you are deleting files, for example, following symbolic links might not be advisable. If you are copying a file tree, you might want to allow it. By default, `walkFileTree` does not follow symbolic links.

The `visitFile` method is invoked for files. If you have specified the `FOLLOW_LINKS` option and your file tree has a circular link to a parent directory, the looping directory is reported in the `visitFileFailed` method with the `FileSystemLoopException`. The following code snippet shows how to catch a circular link and is from the `Copy` example:

```
@Override
public FileVisitResult
    visitFileFailed(Path file,
        IOException exc) {
    if (exc instanceof FileSystemLoopException) {
        System.err.println("cycle detected: " + file);
    } else {
        System.err.format("Unable to copy:" + " %s: %s%n", file, exc);
    }
    return CONTINUE;
}
```

This case can occur only when the program is following symbolic links.

Controlling the Flow

Perhaps you want to walk the file tree looking for a particular directory and, when found, you want the process to terminate. Perhaps you want to skip specific directories.

The `FileVisitor` methods return a `FileVisitResult` value. You can abort the file walking process or control whether a directory is visited by the values you return in the `FileVisitor` methods:

- `CONTINUE` – Indicates that the file walking should continue. If the `preVisitDirectory` method returns `CONTINUE`, the directory is visited.
- `TERMINATE` – Immediately aborts the file walking. No further file walking methods are invoked after this value is returned.
- `SKIP_SUBTREE` – When `preVisitDirectory` returns this value, the specified directory and its subdirectories are skipped. This branch is "pruned out" of the tree.
- `SKIP_SIBLINGS` – When `preVisitDirectory` returns this value, the specified directory is not visited, `postVisitDirectory` is not invoked, and no further unvisited siblings are visited. If returned from the `postVisitDirectory` method, no further siblings are visited. Essentially, nothing further happens in the specified directory.

In this code snippet, any directory named `SCCS` is skipped:

```
import static java.nio.file.FileVisitResult.*;

public FileVisitResult
     preVisitDirectory(Path dir,
         BasicFileAttributes attrs) {
    (if (dir.getFileName().toString().equals("SCCS")) {
         return SKIP_SUBTREE;
    }
    return CONTINUE;
}
```

In this code snippet, as soon as a particular file is located, the file name is printed to standard output, and the file walking terminates:

```
import static java.nio.file.FileVisitResult.*;

// The file we are looking for.
Path lookingFor = ...;

public FileVisitResult
    visitFile(Path file,
        BasicFileAttributes attr) {
    if (file.getFileName().equals(lookingFor)) {
        System.out.println("Located file: " + file);
        return TERMINATE;
    }
    return CONTINUE;
}
```

Examples

The following examples demonstrate the file walking mechanism:

- `Find` – Recurses a file tree looking for files and directories that match a particular glob pattern. This example is discussed in Finding Files.
- `Chmod` – Recursively changes permissions on a file tree (for POSIX systems only).
- `Copy` – Recursively copies a file tree.
- `WatchDir` – Demonstrates the mechanism that watches a directory for files that have been created, deleted or modified. Calling this program with the `-r` option watches an entire tree for changes. For more information about the file notification service, see Watching a Directory for Changes.

Previous page: Links, Symbolic or Otherwise
Next page: Finding Files