Java Magazine

Coding, Java 17

Modern file input/output with Java: Let's get practical

Ben Evans | December 2, 2021

Learn about file system–specific extensions, such as file attributes and symbolic links; traverse directories, temporary files, zip files, and more.

In the first article in this series, "Modern file input/output with Java Path API and Files helper methods," you met Java's `Path` API as well as an older API.

Here, in the second part, you'll learn how the `Path` API (also known as NIO.2) handles file system–specific extensions, including how to access features such as file attributes and symbolic links. I'll also show you how to traverse a directory structure easily, as well as how to watch for changes. To wrap up, this article will look at temporary files (such as those used to indicate file locks) and how to handle .zip files.

File attributes

Each file system has its own set of attributes—and its own interpretation of what those file attributes mean. Fortunately, all modern versions of Java support file attributes across the many file systems via implementations of the `FileAttributeView` and `BasicFileAttributes` interfaces.

By the way, you need to take extra care when writing file system–specific code. Always ensure that your logic and exception handling cover all the cases where your code might run on a different file system.

Listing 1 manipulates the file permissions of a text file (`sample.txt`) that is located in a user's home directory. The user (`ben`) wants to grant others in the same group the permission to read the file. The system-specific POSIX `PosixFileAttributes` class lets you add the read-only permission to the group (and provides an example of the API).

Listing 1. File attribute support in NIO.2

```
import java.nio.file.attribute.*;

import static java.nio.file.attribute.PosixFilePermission.GROUP_READ;

try {
    var teamList = Path.of("/Users/ben/sample.txt");
    PosixFileAttributes attrs =
        Files.readAttributes(teamList, PosixFileAttributes.class);

    Set<PosixFilePermission> posixFilePermissions = attrs.permissions();
```

```
    var owner = attrs.owner().getName();
    var perms = PosixFilePermissions.toString(posixFilePermissions);
    System.out.format("%s %s%n", owner, perms);

    posixFilePermissions.add(GROUP_READ);
    Files.setPosixFilePermissions(teamList, posixFilePermissions);
} catch(IOException e) {
    e.printStackTrace();
}
```

Copy code snippet

The code begins by importing some attribute classes and a `PosixFilePermission` constant, and then it gets a handle for the `sample.txt` file. Using a helper method from `Files`, it gains access to the `PosixFileAttributes` and subsequently to the `PosixFilePermission` attributes. The code echoes the current permissions and then adds the new group read permission to the file.

(I left in some unnecessary type information to aid readability in the example. In working code, some of this boilerplate could be removed.)

Symbolic links

Apart from these types of basic attributes, Java also has an extensible system for supporting special operating system features. I'll showcase this support by walking through an example of symbolic link support. Symbolic links (symlinks) are available in many common operating systems including (but not limited to) Linux, macOS, and Windows.

A symlink can be thought of as a pointer to another file or directory, and in most cases these links are treated transparently. For example, using a symlink to change to a directory will put you in the directory that the symlink is pointing to. However, some software, such as backup utilities, needs to discern and manipulate symbolic links as a special case. NIO.2 allows this.

Java support for symlinks basically follows the semantics of the UNIX implementation. The following example examines and follows a symlink for `/opt/maven` that points to the specific directory for `apache-maven-3.6.1`. The file system looks like the following:

```
ben$ ls -l /opt/
total 0
drwxr-xr-x 9 root wheel 288 Apr 21 2019 apache-maven-3.6.1
lrwxr-xr-x 1 root wheel 18 Apr 21 2019 maven -> apache-maven-3.6.1
```

Listing 2 explores the symbolic link.

Listing 2. Exploring a symbolic link

```
Path path = Path.of("/opt/maven");
try {
    if (Files.isSymbolicLink(path)) {
        Path contents = Files.readSymbolicLink(path);
        Path target = path.toRealPath();
        System.out.println("Link contents: "+ contents);
        System.out.println("Link target: "+ target);
        var attrs = Files.readAttributes(target, BasicFileAttributes.class);
        System.out.println(attrs);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Copy code snippet

The code starts by checking whether the path is a symlink and, if so, it reads the contents of the symlink. This is expressed as a `Path` object—but it does not point to the target of the link. In this case, `contents` contains the single path component `apache-maven-3.6.1`, which does not point to the actual physical path, `/opt/apache-maven-3.6.1`.

Instead, to access the file, use `toRealPath()`, which returns a `Path` that is the fully resolved, physical path.

Note: This discussion covers only *symbolic* links. In the Java APIs, multiple hard-linked files (assuming they are supported by the OS) are *not* distinguished. For example, all hard links are treated as equivalent.

Dealing with directories in NIO.2

Java's ability to navigate directories was given a major overhaul in NIO.2. The addition of the new `java.nio.file.DirectoryStream` interface and its implementing classes allow you to perform the following types of operations:

- Iterate over entries in a directory
- Deal with large directory structures
- Filter entries using regular expressions and MIME-based content detection
- Perform recursive move, copy, and delete operations via the `walkFileTree()` method

Listing 3 shows a simple example of how to use a regular expression pattern match to list all the `.java` files in a project directory.

Listing 3. Finding the matching files

```
try {
    var dir = Path.of("/Users/ben/projects/openjdk/jdk");
    DirectoryStream<Path> stream = Files.newDirectoryStream(dir, "*.java");
    for (Path entry: stream) {
        System.out.println(entry.getFileName());
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Copy code snippet

The code begins from a base `Path` and constructs a `DirectoryStream`. It's confusing because despite the name, this class does *not* implement the `Stream` interface (in fact, `DirectoryStream` predates Java 8). Instead, the class is intended to be used as a filtering tool. The code iterates over the entries that have been found and prints each one out.

Listing 3 shows how easy this API is to use when dealing with a single directory. But there's a problem that you will see if you run the code: It returns no results. This is because the code will look only at a single directory, which is not very useful.

What can you do if you need to recursively filter across multiple directories?

For example, Java's source files typically sit in a package structure. And the OpenJDK source code is not stored in the main directory but further down in the file system in `src/java.base/share/classes/java/lang/`. Thus, the source files exist in multiple directories, at multiple levels of depth in the file system. You need a way to handle this layout, and recursion that walks the directory tree is a natural way to achieve this.

Walking a directory tree is a relatively uncommon feature of Java and involves a number of interfaces and implementation details. The key method to use for walking the directory is

```
Files.walkFileTree(Path startingDir, FileVisitor<? super Path> visitor);
```

Providing the `startingDir` is easy enough, but what about an implementation of the `FileVisitor` interface? Unfortunately, it is not as simple as providing a lambda; in fact, you need to implement at least five methods.

- `FileVisitResult preVisitDirectory(T dir)`
- `FileVisitResult preVisitDirectoryFailed(T dir, IOException exc)`
- `FileVisitResult visitFile(T file, BasicFileAttributes attrs)`
- `FileVisitResult visitFileFailed(T file, IOException exc)`
- `FileVisitResult postVisitDirectory(T dir, IOException exc)`

That looks like a fair amount of work, but fortunately the API supplies a default implementation, the `SimpleFileVisitor`. Carrying on from

the example in **Listing 3** of finding `.java` source files in a directory, you can now list `.java` source files from all the directories that sit underneath `/Users/ben/projects/openjdk/jdk`. **Listing 4** demonstrates this use of the `walkFileTree()` method.

Listing 4. Finding Java source code in subdirectories

```
var startingDir = Path.of("/Users/ben/projects/openjdk/jdk");
Files.walkFileTree(startingDir, new FindJavaVisitor());

public class FindJavaVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
        if (file != null && attrs != null) {
            if (file.getFileName().toString().endsWith(".java")) {
                System.out.println(file.getFileName());
            }
        }
        return FileVisitResult.CONTINUE;
    }
}
```

Copy code snippet

You start by creating a `Path` object and then call the `Files.walkFileTree()` method. Pass in a `FindJavaVisitor`, which is an implementation that extends the `SimpleFileVisitor`. In other words, let the `SimpleFileVisitor` do most of the work of traversing the directories and so on. The only code you have to write is when you override the `visitFile()` method. In this method, you write some simple code to see if a file ends with `.java` and echo its name to `stdout` if it does.

Other use cases could be to recursively move, copy, delete, or otherwise modify files. In most cases, you'll only need to implement an extension to the `SimpleFileVisitor` class, but the flexibility is there in the API if you want to implement your own complete `FileVisitor`.

Watch services and file change notifications

Files and directories are entities that can change in various ways, given their attributes as well as their contents. How do you know they changed so you can react to that event?

The Java APIs provide the ability to monitor a file or directory for changes. This is done through the `WatchService` class in `java.nio.file`. This class uses VM-managed client threads to keep an eye on registered files or directories for changes, and it will return an event when a change is detected. You do *not* need to manage these threads.

This sort of event notification can be useful for security monitoring, refreshing data from a properties file, looking for log creation or changes, and many other use cases. In **Listing 5**, `WatchService` is used to detect any changes to the home directory of the user `ben` and to print a modification event to the console.

Listing 5. Using WatchService

```
try {
    var watcher = FileSystems.getDefault().newWatchService();
    var dir = Path.of("/Users/ben");

    var registered = dir.register(watcher, ENTRY_MODIFY);

    while (!shutdown) {
        WatchKey key = null;
        try {
            key = watcher.take();
            for (WatchEvent<?> event : key.pollEvents()) {
                if (event == null) {
                    continue;
                }
                if (event.kind() == ENTRY_MODIFY) {
```

```
                    System.out.println("Home dir changed!");
                }
            }
            key.reset();
        } catch (InterruptedException e) {
            // Log interruption
            shutdown = true;
        }
    }
} catch (IOException | InterruptedException e) {
    e.printStackTrace();
}
```

Copy code snippet

The initial setup involves getting the default `WatchService` and a handle on the `ben` home directory. Next is the registration of a modification watch on that directory. Then, using the *safe shutdown* pattern (usually seen with a volatile boolean variable `shutdown`), the `take()` method on `WatchService` blocks until a `WatchKey` becomes available. As soon as the key is made available, the code polls that `WatchKey` for a list of `WatchEvent` objects.

If a `WatchEvent` is found of type `ENTRY_MODIFY`, the code communicates that fact by echoing the event. Finally, reset the key so it's ready for the next event.

This technique is not widely used, but it can be a useful approach and a good thing to keep in your toolbox of I/O recipes.

By the way, some file system implementations may occasionally return a null event, so you should check for that condition.

Temporary files

Helper methods in the `Files` class include capabilities to create and work with temporary files and directories. (Note that UNIX-like operating systems clean up temporary working areas such as `/tmp` automatically, often at system reboot. You should *never* treat these areas as permanent storage.)

The simplest way to create a temporary file is

```
Path p;
try {
    p = Files.createTempFile("temp", null);
} catch (IOException e) {
    e.printStackTrace();
}
// code to work with p
```

After you've finished with the file, by default, you need to delete it. This quickly becomes tiresome, but fortunately the API has a simple solution. The `Files` class provides convenience methods for tasks such as an output stream from a `Path`. These helpers come with the possibility of specifying options for how the file is to be opened and closed, such as the following:

```
Path p;
try {
    p = Files.createTempFile("temp", null);
    System.out.println(p.getFileName());
    try (var output = Files.newOutputStream(p, DELETE_ON_CLOSE)) {
        // do work with the OutputStream
        // Temp file will be automatically cleaned up
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

As you can see, the temporary file will be automatically deleted when the file is closed, which will occur when the inner `try` block is complete. This is much cleaner and safer than requiring manual cleanup and demonstrates a nice interaction with the try-with-resources language feature.

Listing 6 unpacks a `.jar` file into a temporary directory. This uses not only the temporary directory handling but also a `ZipInputStream`. This is a very useful class for working with `.zip` files (remember that JAR files are really just `.zip` files with a metadata directory stored in them

along with the code).

Listing 6. Working with .zip files

```
public static Path unpackJar(String zipFilePath) throws IOException {
    var tmpDir = Files.createTempDirectory(Path.of("/tmp"), "jar-extract");

    try (var zipIn = new ZipInputStream(new FileInputStream(zipFilePath))) {
        var entry = zipIn.getNextEntry();

        while (entry != null) {
            var newFile = tmpDir.resolve(entry.getName());
            if (entry.isDirectory()) {
                Files.createDirectory(newFile);
            } else {
                Files.copy(zipIn, newFile);
            }
            zipIn.closeEntry();
            entry = zipIn.getNextEntry();
        }
    }

    return tmpDir;
}
```

Copy code snippet

Conclusion

In this article, you've delved deep into Java's `NIO.2` API and seen some techniques that allow you to use platform-specific features and to traverse structures and watch for changes. These are broadly high-level operations.

What I haven't done so far is to discuss the low-level and high-performance features of modern Java I/O. These APIs will be the subject of the next article in this series.

Dig deeper

- Modern file input/output with Java Path API and Files helper methods
- Working and unit testing with temporary files in Java
- The Java Tutorials: Watching a directory for changes
- The joy of writing command-line utilities, Part 1: Finding duplicate files
- The joy of writing command-line utilities, Part 2: The souped-up way to find duplicate files

Ben Evans

Ben Evans (@kittylyst) is a Java Champion and Senior Principal Software Engineer at Red Hat. He has written five books on programming, including *Optimizing Java* (O'Reilly) and *The Well-Grounded Java Developer* (Manning). Previously he was Lead Architect for Instrumentation at New Relic, a founder of jClarity (acquired by Microsoft) and a member of the Java SE/EE Executive Committee.

Previous Post

Fight ambiguity and improve your code with Java 17's sealed classes

Mala Gupta | 14 min read

Next Post

Binary bit manipulation and CAN bus hardware interfaces in Java

Eric J. Bruno | 17 min read