17-new-io-package

Contents

- NIO
  - Reading
  - Writing
  - Memory-mapping
  - Async
  - Path
  - Files

NIO

Java `NIO` or `new-io`, is basically the new improved implementation of the standard `java.io`, this new implementation which was greatly improved in 1.7 allows one to work with IO streams in a non blocking way, which can greatly alleviate the load on particular systems, such as web servers. For example in a regular scenario where each new request to a web server creates a new thread, or uses free thread from a pool, instead of blocking the entire thread waiting for an java.io operation, using the old implementation, with the new approach one can use java.nio to open a channel to a file in a non blocking way, and when ready to read from the file or device.

Reading

The two examples show the usage of the new `API`, compared to the old one, in both cases the implementation is blocking however, the new `NIO API` is much cleaner, does most of the work internally, is wrapped in nice short methods which deliver the data easily, however it has less control over how the data stream is read.

```
public class OldIOReadExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("
           example.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class NIOReadExample {
    public static void main(String[] args) {
        try {
            List<String> lines = Files.readAllLines(Paths.get("example.txt"),
                StandardCharsets.UTF_8);
            for (String line : lines) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Writing

The two examples show the usage of the new `API`, compared to the old one, in both cases the implementation is blocking however, the new `NIO API` is much cleaner, does most of the work internally, is wrapped in nice short methods which deliver the data easily, however it has less control over how the data stream is write.

```
public class OldIOWriteExample {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("
           example.txt"))) {
            writer.write("Hello, World!");
            writer.newLine();
            writer.write("This is a sample text.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class NIOWriteExample {
    public static void main(String[] args) {
        List<String> lines = Arrays.asList("Hello, World!", "This is a sample
            text.");
        try {
            Files.write(Paths.get("example.txt"), lines, StandardCharsets.
               UTF_8);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Memory-mapping

The implementation below deals with memory mapped files, which is a feature of the `NIO` library, allows one to map a given file resource to virtual memory, the OS can serve the file contents as if from the main memory which should make reading `mmaped` files more effective. This is a well known feature used in other languages such as C or C++.

```
public class NIOMemoryMappedReadExample {
    public static void main(String[] args) {
        try (FileChannel fileChannel = FileChannel.open(Paths.get("example.
           txt"), StandardOpenOption.READ)) {
            MappedByteBuffer buffer = fileChannel.map(FileChannel.MapMode.
               READ_ONLY, 0, fileChannel.size());

            for (int i = 0; i < buffer.limit(); i++) {
                System.out.print((char) buffer.get());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Async

In the example below one can see how the channels `API` might be used to read from a file, in an `async` manner, while the future is not finished, other work can be done, one can poll on the future over a specific period of time to check if it is ready, but the reading itself happens without blocking the current thread.

The code below demonstrates how to read and write from a file using the `async` approach in `nio`, notice how both the calls tr read and write are non blocking, the thread is not stopped waiting for response from the calls, instead

```
public class AsyncFileReadExample {
    public static void main(String[] args) {
        try (AsynchronousFileChannel fileChannel = AsynchronousFileChannel.
           open(Paths.get("example.txt"), StandardOpenOption.READ)) {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            Future<Integer> result = fileChannel.read(buffer, 0);

            while (!result.isDone()) {
                System.out.println("Doing something else while file is being
                   read...");
                // Do something else in the meantime (e.g., a longer
                   computation)
            }

            int bytesRead = result.get();  // Blocks future here if not yet
               done
            System.out.println("Bytes read: " + bytesRead);

            // rewind the buffer, reset the buffer pointer back so it can be
               read
```

```
            buffer.rewind();

            // read the contents of the buffer, that can be done in many ways
               , here
            // one just copies back to a regular byte array,
            byte[] data = new byte[buffer.remaining()];
            buffer.get(data);
            System.out.println("File content: " + new String(data));
        } catch (IOException | InterruptedException | java.util.concurrent.
           ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class AsyncFileWriteExample {
    public static void main(String args[]) {
        try (AsynchronousFileChannel fileChannel = AsynchronousFileChannel.
           open(Paths.get("example.txt"), StandardOpenOption.WRITE,
           StandardOpenOption.CREATE))
        {
            // Create a buffer.
            ByteBuffer mBuf = ByteBuffer.allocate(26);
            // Write some bytes to the buffer.
            for(int i=0; i<26; i++)
            mBuf.put((byte)('A' + i));
            // Reset the buffer so that it can be written.
            mBuf.rewind();
            // Write the buffer from start to the output file.
            fChan.write(mBuf);
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
            System.exit(1);
        }
    }
}
```

Path

Another key feature added to the `NIO` library which serves as a bridge between the native `java.lang` File class and the new `NIO` packages. Below are listed some of the methods which are present in the Path class which are mostly meant to work with system paths, and are the backbone of the `nio` package library

Method	Description
compareTo(Path other)	Compares this `Path` to another `Path` lexicographically.
endsWith(Path other)	Checks if this `Path` ends with the specified `Path`.
endsWith(String other)	Checks if this `Path` ends with the given string, treating it as a sequence of name elements.

Method	Description
equals(Object other)	Compares this `Path` with another object for equality.
getFileName()	Returns the file name or the last name element of the `Path`.
getName(int index)	Returns the name element at the specified index from the `Path`.
getNameCount()	Returns the number of name elements in the `Path`.
getParent()	Returns the parent path of this `Path`, or `null` if there is no parent.
getRoot()	Returns the root of the `Path`, if available.
hashCode()	Returns a hash code for the `Path`.
isAbsolute()	Returns `true` if the `Path` is absolute.
iterator()	Returns an iterator over the name elements of the `Path`.
normalize()	Returns a normalized version of the `Path`, removing redundant elements like `.` and `...`
of(String first, String… more)	Creates a `Path` from one or more strings.
of(URI uri)	Converts a `URI` to a `Path`.
register(WatchService watcher, Kind<?>… events)	Registers the `Path` with a `WatchService` to monitor file system events.
register(WatchService watcher, Kind<?>[] events, Modifier… modifiers)	Registers the `Path` with a `WatchService` with extra settings.
relativize(Path other)	Constructs a relative path from this path to the given `Path`.
resolve(Path other)	Appends the provided `Path` to this `Path`.
resolve(Path first, Path… more)	Appends one or more paths to this `Path`.
resolve(String other)	Appends the given string (path segment) to this `Path`.
resolve(String first, String… more)	Appends one or more strings to this `Path`.
resolveSibling(Path other)	Resolves the `Path` against the parent of this path.
resolveSibling(String other)	Resolves the `Path` against the parent with a sibling represented by a string.
startsWith(Path other)	Checks if this `Path` starts with the specified `Path`.
startsWith(String other)	Checks if this `Path` starts with the given string.
subpath(int beginIndex, int endIndex)	Returns a subpath of this `Path` from `beginIndex` to `endIndex`.
toAbsolutePath()	Converts this `Path` to an absolute path.
toFile()	Converts the `Path` to a `File` object.
toRealPath(LinkOption… options)	Returns the real path after resolving symbolic links and verifying the file's existence.
toString()	Returns the string representation of the `Path`.
toUri()	Converts the `Path` to a `URI`.

`When updating legacy code bases it is possible to convert from the old File to the new Path and vice versa by calling the toPath or toFile methods which will do exactly that.`

Files

The files class mostly contains static methods, which are also meant to work with the `Path` class, already mentioned above, the `Files` class is similarly to the `Path` class one of the cornerstones of the `NIO` library in java. Some of the most notable methods from the `Files` class are listed below

Method	Description
copy(InputStream, Path, CopyOption…)	Copies data from an `InputStream` to the target `Path`.
copy(Path, OutputStream)	Copies the contents of a file (from a `Path`) to an `OutputStream`.
copy(Path, Path, CopyOption…)	Copies a file from one `Path` to another, with optional copy options.
createAndCheckIsDirectory(Path, FileAttribute<?\>…)	Creates a directory and checks if it was successfully created.
createBufferedReaderLinesStream(BufferedReader)	Creates a `Stream\<String\>` of lines from a `BufferedReader`.
createDirectories(Path, FileAttribute<?\>…)	Creates a directory along with any necessary non-existent parent directories.
createDirectory(Path, FileAttribute<?\>…)	Creates a new directory at the specified `Path`.
createFile(Path, FileAttribute<?\>…)	Creates a new file at the given `Path` with optional file attributes.
createFileChannelLinesStream(FileChannel, Charset)	Creates a `Stream\<String\>` of lines from a `FileChannel`, using the specified `Charset`.
createLink(Path, Path)	Creates a hard link between two paths (source and target).
createTempDirectory(Path, String, FileAttribute<?\>…)	Creates a temporary directory in the given `Path`, with an optional prefix.
createTempFile(Path, String, String, FileAttribute<?\>…)	Creates a temporary file in the given `Path`, with an optional prefix and suffix.
delete(Path)	Deletes the file or directory at the given `Path`.
deleteIfExists(Path)	Deletes the file or directory if it exists, otherwise does nothing.
exists(Path, LinkOption…)	Checks if the file or directory at the given `Path` exists.
find(Path, int, BiPredicate<Path, BasicFileAttributes>, FileVisitOption…)	Finds files in a directory tree based on a matching condition (`BiPredicate`).
followLinks(LinkOption…)	Configures whether symbolic links should be followed during file operations.
getAttribute(Path, String, LinkOption…)	Retrieves a specific file attribute of the file at the given `Path`.
getFileStore(Path)	Returns the `FileStore` (e.g., disk partition) where the file is located.
getLastModifiedTime(Path, LinkOption…)	Retrieves the last modified time of the file at the given `Path`.
getOwner(Path, LinkOption…)	Retrieves the owner of the file at the given `Path`.
getPosixFilePermissions(Path, LinkOption…)	Retrieves the POSIX file permissions of the file at the given `Path`.
isAccessible(Path, AccessMode…)	Checks if the file can be accessed with the given access modes (e.g., read, write).
isDirectory(Path, LinkOption…)	Checks if the given `Path` represents a directory.
isExecutable(Path)	Checks if the file at the given `Path` is executable.
isHidden(Path)	Checks if the file at the given `Path` is hidden.
isReadable(Path)	Checks if the file at the given `Path` is readable.
isRegularFile(Path, LinkOption…)	Checks if the given `Path` represents a regular file.
isSameFile(Path, Path)	Checks if two `Path` objects refer to the same file.
isWritable(Path)	Checks if the file at the given `Path` is writable.

Method	Description
lines(Path)	Returns a `Stream\<String\>` of all lines in the file at the given `Path`.
lines(Path, Charset)	Returns a `Stream\<String\>` of all lines in the file using the specified `Charset`.
list(Path)	Returns a `Stream\<Path\>` of all entries in the directory at the given `Path`.
move(Path, Path, CopyOption…)	Moves a file from one `Path` to another, with optional move options.
notExists(Path, LinkOption…)	Checks if the file or directory at the given `Path` does not exist.
read(InputStream, int)	Reads up to `n` bytes from an `InputStream`.
readAllBytes(Path)	Reads all bytes from a file at the given `Path`.
readAllLines(Path)	Reads all lines from a file at the given `Path`.
readAllLines(Path, Charset)	Reads all lines from a file using the specified `Charset`.
readAttributes(Path, Class<A>, LinkOption…) <A extends BasicFileAttributes>	Reads the file attributes as an instance of the given class `A`.
readAttributes(Path, String, LinkOption…)	Reads a set of file attributes identified by a string.
readString(Path)	Reads the contents of a file into a `String`.
readString(Path, Charset)	Reads the contents of a file into a `String` using the specified `Charset`.
setAttribute(Path, String, Object, LinkOption…)	Sets a file attribute for the file at the given `Path`.
setLastModifiedTime(Path, FileTime)	Sets the last modified time of the file at the given `Path`.
setOwner(Path, UserPrincipal)	Sets the owner of the file at the given `Path`.
setPosixFilePermissions(Path, Set<PosixFilePermission>)	Sets the POSIX file permissions for the file at the given `Path`.
size(Path)	Returns the size of the file at the given `Path`.
walk(Path, FileVisitOption…)	Returns a `Stream\<Path\>` that walks through the directory tree starting at the given `Path`.
walk(Path, int, FileVisitOption…)	Returns a `Stream\<Path\>` that walks the directory tree to a specified depth.
walkFileTree(Path, FileVisitor<? super Path>)	Walks a file tree starting from the given `Path`, visiting each file using a `FileVisitor`.
walkFileTree(Path, Set<FileVisitOption>, int, FileVisitor<? super Path>)	Walks a file tree with additional options and a depth limit, visiting each file using a `FileVisitor`.
write(Path, Iterable<? extends CharSequence>, Charset, OpenOption…)	Writes an `Iterable` of character sequences to a file using the specified `Charset`.
write(Path, Iterable<? extends CharSequence>, OpenOption…)	Writes an `Iterable` of character sequences to a file.
write(Path, byte[], OpenOption…)	Writes a byte array to a file at the given `Path`.
writeString(Path, CharSequence, Charset, OpenOption…)	Writes a `CharSequence` to a file using the specified `Charset`.
writeString(Path, CharSequence, OpenOption…)	Writes a `CharSequence` to a file.