RxIo

Asynchronous non-blocking File Reader and Writer library for Java

About

The `AsyncFiles` class allows JVM applications to easily read/write files asynchronously with non-blocking IO. `AsyncFiles` take advantage of Java AsynchronousFileChannel to perform asynchronous I/O operations. `AsyncFiles` provides equivalent operations to the standard JDK Files class but using non-blocking IO and an asynchronous API with different asynchronous idioms, namely: `CompletableFuture`, jayield `AsyncQuery`, reactive-streams `Publisher`, Kotlin coroutines and Kotlin Asynchronous Flow.
In section Usage we present some examples using the `AsyncFiles` class side by side with the corresponding blocking version of Files.

Installation

First, in order to include it to your project, simply add this dependency:

Maven

```
<dependency>

<groupId>com.github.javasync</groupId>
    <artifactId>RxIo</artifactId>
    <version>1.2.5</version>
</dependency>
```

Gradle

```
implementation
'com.github.javasync:RxIo:1.2.5'
```

Usage

Kotlin examples:

```
suspend fun copyNio(from: String, to: String) {
  val data = Path(from).readText() // suspension point
  Path(to).writeText(data)         // suspension point
}

Path("input.txt")
  .lines()   // Flow<String>
  .onEach(::println)
  .collect() // block to wait for completion
```

```
fun copy(from: String, to: String) {
    val data = File(from).readText()
    File(to).writeText(data)
}

Path("input.txt")
  .readLines() // List<String>
  .forEach(::println)
```

Java examples:

```
AsyncFiles
  .readAllBytes("input.txt")
  .thenCompose(bytes -> AsyncFiles.writeBytes("output.txt",
bytes))
  .join(); // block if you want to wait for completion
```

```
Path in = Paths.get("input.txt");
Path out = Paths.get("output.txt");
byte[] bytes =
Files.readAllBytes(in);
Files.write(out, bytes);
```

```
AsyncFiles
  .asyncQuery("input.txt")
  .onNext((line, err) -> out.println(line))
.blockingSubscribe(); // block if you want to wait for
completion
```

```
Path path = Paths.get("input.txt");
Files
  .lines(path)
  .forEach(out::println)
```

```
List<String> data = asList("super", "brave", "isel", "gain");
AsyncFiles
  .write("output.txt", data) // writing lines to output.txt
  .join(); // block if you want to wait for completion
```

```
List<String> data = asList("super",
"brave", "isel", "gain");
Path path = Paths.get("output.txt")
Files.write(path, data);
```

The `AsyncFiles::lines()` returns a reactive `Publisher` which is compatible with Reactor or RxJava streams. Thus we can use the utility methods of Reactor `Flux` to easily operate on the result of `AsyncFiles::lines()`. In the following example we show how to print all words of a gutenberg.org file content without repetitions:

```
Flux
    .from(AsyncFiles.lines(file))
    .filter(line -> !line.isEmpty())                    // Skip empty lines
    .skip(14)                                           // Skip gutenberg header
    .takeWhile(line -> !line.contains("*** END OF "))  // Skip gutenberg footnote
    .flatMap(line -> Flux.fromArray(line.split("\\W+")))
    .distinct()
    .doOnNext(out::println)
    .doOnError(Throwable::printStackTrace)
    .blockLast(); // block if you want to wait for completion
```

Alternatively, the `AsyncFiles::asyncQuery()` returns an `AsyncQuery` that allows asynchronous subscription and chaining intermediate operations such as `filter`, `map` and others. We can rewrite the previous sample as:

```
AsyncFiles
    .asyncQuery(file)
    .filter(line -> !line.isEmpty())                    // Skip empty lines
    .skip(14)                                           // Skip gutenberg header
    .takeWhile(line -> !line.contains("*** END OF "))  // Skip gutenberg footnote
    .flatMapMerge(line -> AsyncQuery.of(line.split("\\W+")))
    .distinct()
    .subscribe((word, err) -> {
        if(err != null) err.printStackTrace();
        else out.println(word);
    })
    .join(); // block if you want to wait for completion
```