Java 8 - 其它更新: 字符串， base64,...

处理数值

Java8添加了对无符号数的额外支持。Java中的数值总是有符号的，例如，让我们来观察Integer:

int可表示最多2 ** 32个数。Java中的数值默认为有符号的，所以最后一个二进制数字表示符号(0为正数，1为负数)。所以从十进制的0开始，最大的有符号正整数为2 ** 31 - 1。

你可以通过Integer.MAX_VALUE来访问它:

```
System.out.println(Integer.MAX_VALUE);      // 2147483647
System.out.println(Integer.MAX_VALUE + 1);  // -2147483648
```

Java8添加了解析无符号整数的支持，让我们看看它如何工作:

```
long maxUnsignedInt = (1l << 32) - 1;
String string = String.valueOf(maxUnsignedInt);
int unsignedInt = Integer.parseUnsignedInt(string, 10);
String string2 = Integer.toUnsignedString(unsignedInt, 10);
```

就像你看到的那样，现在可以将最大的无符号数2 ** 32 - 1解析为整数。而且你也可以将这个数值转换回无符号数的字符串表示。

这在之前不可能使用parseInt完成，就像这个例子展示的那样:

```
try {
    Integer.parseInt(string, 10);
}
catch (NumberFormatException e) {
    System.err.println("could not parse signed int of " + maxUnsignedInt);
}
```

这个数值不可解析为有符号整数，因为它超出了最大范围2 ** 31 - 1。 算术运算

Math工具类新增了一些方法来处理数值溢出。这是什么意思呢? 我们已经看到了所有数值类型都有最大值。所以当算术运算的结果不能被它的大小装下时，会发生什么呢?

```
System.out.println(Integer.MAX_VALUE);      // 2147483647
System.out.println(Integer.MAX_VALUE + 1);  // -2147483648
```

就像你看到的那样，发生了整数溢出，这通常是我们不愿意看到的。

Java8添加了严格数学运算的支持来解决这个问题。Math扩展了一些方法，它们全部以exact结尾，例如addExact。当运算结果不能被数值类型装下时，这些方法通过抛出ArithmeticException异常来合理地处理溢出。

```
try {
    Math.addExact(Integer.MAX_VALUE, 1);
}
catch (ArithmeticException e) {
    System.err.println(e.getMessage());
    // => integer overflow
}
```

当尝试通过toIntExact将长整数转换为整数时，可能会抛出同样的异常:

```
try {
    Math.toIntExact(Long.MAX_VALUE);
}
catch (ArithmeticException e) {
    System.err.println(e.getMessage());
    // => integer overflow
}
```

处理文件

Files工具类首次在Java7中引入，作为NIO的一部分。JDK8 API添加了一些额外的方法，它们可以将文件用于函数式数据流。让我们深入探索一些代码示例。 列出文件

Files.list方法将指定目录的所有路径转换为数据流，便于我们在文件系统的内容上使用类似filter和sorted的流操作。

```
try (Stream<Path> stream = Files.list(Paths.get(""))) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> !path.startsWith("."))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("List: " + joined);
}
```

上面的例子列出了当前工作目录的所有文件，之后将每个路径都映射为它的字符串表示。之后结果被过滤、排序，最后连接为一个字符串。如果你还不熟悉函数式数据流，你应该阅读我的Java8数据流教程。

你可能已经注意到，数据流的创建包装在try-with语句中。数据流实现了AutoCloseable，并且这里我们需要显式关闭数据流，因为它基于IO操作。

> 返回的数据流是DirectoryStream的封装。如果需要及时处理文件资源，就应该使用try-with结构来确保在流式操作完成后，数据流的close方法被调用。

查找文件

下面的例子演示了如何查找在目录及其子目录下的文件:

```
Path start = Paths.get("");
int maxDepth = 5;
try (Stream<Path> stream = Files.find(start, maxDepth, (path, attr) ->
        String.valueOf(path).endsWith(".js"))) {
    String joined = stream
        .sorted()
        .map(String::valueOf)
        .collect(Collectors.joining("; "));
    System.out.println("Found: " + joined);
}
```

find方法接受三个参数: 目录路径start是起始点，maxDepth定义了最大搜索深度。第三个参数是一个匹配谓词，定义了搜索的逻辑。上面的例子中，我们搜索了所有JavaScirpt文件(以.js结尾的文件名)。

我们可以使用Files.walk方法来完成相同的行为。这个方法会遍历每个文件，而不需要传递搜索谓词。

```
Path start = Paths.get("");
int maxDepth = 5;
try (Stream<Path> stream = Files.walk(start, maxDepth)) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> path.endsWith(".js"))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("walk(): " + joined);
}
```

这个例子中，我们使用了流式操作filter来完成和上个例子相同的行为。

读写文件

将文本文件读到内存，以及向文本文件写入字符串在Java 8 中是简单的任务。不需要再去摆弄读写器了。Files.readAllLines从指定的文件把所有行读进字符串列表中。你可以简单地修改这个列表，并且将它通过Files.write写到另一个文件中:

```
List<String> lines = Files.readAllLines(Paths.get("res/nashorn1.js"));
lines.add("print('foobar');");
Files.write(Paths.get("res/nashorn1-modified.js"), lines);
```

要注意这些方法对内存并不十分高效，因为整个文件都会读进内存。文件越大，所用的堆区也就越大。

你可以使用Files.lines方法来作为内存高效的替代。这个方法读取每一行，并使用函数式数据流来对其流式处理，而不是一次性把所有行都读进内存。

```
try (Stream<String> stream = Files.lines(Paths.get("res/nashorn1.js"))) {
    stream
        .filter(line -> line.contains("print"))
        .map(String::trim)
        .forEach(System.out::println);
}
```

如果你需要更多的精细控制，你需要构造一个新的BufferedReader来代替:

```
Path path = Paths.get("res/nashorn1.js");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    System.out.println(reader.readLine());
}
```

或者，你需要写入文件时，简单地构造一个BufferedWriter来代替:

```
Path path = Paths.get("res/output.js");
try (BufferedWriter writer = Files.newBufferedWriter(path)) {
    writer.write("print('Hello World');");
}
```

BufferedReader也可以访问函数式数据流。lines方法在它所有行上面构建数据流:

```
Path path = Paths.get("res/nashorn1.js");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    long countPrints = reader
        .lines()
        .filter(line -> line.contains("print"))
        .count();
    System.out.println(countPrints);
}
```

目前为止你可以看到Java8提供了三个简单的方法来读取文本文件的每一行，使文件处理更加便捷。

不幸的是你需要显式使用try-with语句来关闭文件流，这会使示例代码有些凌乱。我期待函数式数据流可以在调用类似count和collect时可以自动关闭，因为你不能在相同数据流上调用终止操作两次。

java.util.Random

在Java8中java.util.Random类的一个非常明显的变化就是新增了返回随机数流(random Stream of numbers)的一些方法。

下面的代码是创建一个无穷尽的double类型的数字流，这些数字在0(包括0)和1(不包含1)之间。

```
Random random = new Random();
DoubleStream doubleStream = random.doubles();
```

下面的代码是创建一个无穷尽的int类型的数字流，这些数字在0(包括0)和100(不包括100)之间。

```
Random random = new Random();
IntStream intStream = random.ints(0, 100);
```

那么这些无穷尽的数字流用来做什么呢? 接下来，我通过一些案例来分析。记住，这些无穷大的数字流只能通过某种方式被截断(limited)。

示例1: 创建10个随机的整数流并打印出来:

```
intStream.limit(10).forEach(System.out::println);
```

示例2: 创建100个随机整数:

```
    List<Integer> randomBetween0And99 = intStream
                                        .limit(100)
                                        .boxed()
                                        .collect(Collectors.toList());
```

对于高斯伪随机数(gaussian pseudo-random values)来说，random.doubles()方法所创建的流不能等价于高斯伪随机数，然而，如果用java8所提供的功能是非常容易实现的。

```
Random random = new Random();
DoubleStream gaussianStream = Stream.generate(random::nextGaussian).mapToDouble(e -> e);
```

这里，我使用了Stream.generate api，并传入Supplier 类的对象作为参数，这个对象是通过调用Random类中的方法nextGaussian()创建另一个高斯伪随机数。

接下来，我们来对double类型的伪随机数流和double类型的高斯伪随机数流做一个更加有意思的事情，那就是获得两个流的随机数的分配情况。预期的结果是: double类型的伪随机数是均匀的分配的，而double类型的高斯伪随机数应该是正态分布的。

通过下面的代码，我生成了一百万个伪随机数，这是通过java8提供的api实现的:

```
Random random = new Random();
DoubleStream doubleStream = random.doubles(-1.0, 1.0);
LinkedHashMap<Range, Integer> rangeCountMap = doubleStream.limit(1000000)
    .boxed()
    .map(Ranges::of)
    .collect(Ranges::emptyRangeCountMap, (m, e) -> m.put(e, m.get(e) + 1),
Ranges::mergeRangeCountMaps);

rangeCountMap.forEach((k, v) -> System.out.println(k.from() + "\t" + v));
```

代码的运行结果如下:

```
    -1      49730
    -0.9    49931
    -0.8    50057
    -0.7    50060
    -0.6    49963
    -0.5    50159
    -0.4    49921
    -0.3    49962
    -0.2    50231
    -0.1    49658
    0       50177
    0.1     49861
    0.2     49947
    0.3     50157
    0.4     50414
    0.5     50006
    0.6     50038
    0.7     49962
    0.8     50071
    0.9     49695
```

为了类比，我们再生成一百万个高斯伪随机数:

```
Random random = new Random();
DoubleStream gaussianStream = Stream.generate(random::nextGaussian).mapToDouble(e -> e);
LinkedHashMap<Range, Integer> gaussianRangeCountMap =
    gaussianStream
            .filter(e -> (e >= -1.0 && e < 1.0))
            .limit(1000000)
            .boxed()
            .map(Ranges::of)
            .collect(Ranges::emptyRangeCountMap, (m, e) -> m.put(e, m.get(e) + 1),
Ranges::mergeRangeCountMaps);

gaussianRangeCountMap.forEach((k, v) -> System.out.println(k.from() + "\t" + v));
```

上面代码输出的结果恰恰与我们预期结果相吻合，即: double类型的伪随机数是均匀的分配的，而double类型的高斯伪随机数应该是正态分布的。

java.util.Base64

> Java8中java.util.Base64性能比较高，推荐使用。请参考:

- 性能对比: https://wizardforcel.gitbooks.io/java8-new-features/content/11.html
- 源代码: http://git.oschina.net/benhail/javase8-sample

该类提供了一套静态方法获取下面三种BASE64编解码器:

1)Basic编码: 是标准的BASE64编码，用于处理常规的需求

```
// 编码
String asB64 = Base64.getEncoder().encodeToString("some string".getBytes("utf-8"));
System.out.println(asB64); // 输出为: c29tZSBzdHJpbmc=
// 解码
byte[] asBytes = Base64.getDecoder().decode("c29tZSBzdHJpbmc=");
System.out.println(new String(asBytes, "utf-8")); // 输出为: some string
```

2)URL编码: 使用下划线替换URL里面的反斜线“/”

```
String urlEncoded = Base64.getUrlEncoder().encodeToString("subjects?abcd".getBytes("utf-8"));
System.out.println("Using URL Alphabet: " + urlEncoded);
// 输出为:
Using URL Alphabet: c3ViamVjdHM_YWJjZA==
```

3)MIME编码: 使用基本的字母数字产生BASE64输出，而且对MIME格式友好: 每一行输出不超过76个字符，而且每行以“\r\n”符结束。

```
StringBuilder sb = new StringBuilder();
for (int t = 0; t < 10; ++t) {
  sb.append(UUID.randomUUID().toString());
}
byte[] toEncode = sb.toString().getBytes("utf-8");
String mimeEncoded = Base64.getMimeEncoder().encodeToString(toEncode);
System.out.println(mimeEncoded);
```