1.0.8 Vavr - 函数式编程库 (javaslang)

背景

Java在吸收了很多现代编程语言的优点后，从8往后的版本均引入了很多其他编程语言中经过验证比较好用的特性，但受限于 Java 标准库的通用性要求和二进制文件大小，它对我们工作中使用频率最高的集合类支持较差、函数式编程的 API 支持也比较有限。

既然要谈vavr，那么先要谈为什么要使用vavr，vavr是为了增强Java的函数式编程体验的。Java 8引入了函数式编程范式，思路是：将函数作为其他函数的参数传递，其实在Java 8之前，Java也支持类似的功能，但是需要使用接口实现多态，或者使用匿名类实现。不管是接口还是匿名类，都有很多模板代码，因此Java 8引入了Lambda表达式，正式支持函数式编程。

vavr是在尝试让Java拥有跟Scala类似的语法。使用起来很简单，只需要添加对 io.vavr:vavr 库的 Maven 依赖即可。vavr提供了更好用、不可变的集合框架；更好的函数式编程特性；Monad；元组；模式匹配，等。

(Hystrix是Netflix开源的限流、熔断降级组件，Hystrix已经不再更新了，而在github主页上引导到了另一个替代项目——resilience4j，这个以轻量级著称的项目是基于Java 8开发的，并且只使用了唯一的一个外部依赖库，也就是vavr.)

示例

集合

"A persistent collection when modified produces a new version of the collection while preserving the current version.

Maintaining multiple versions of the same collection might lead to inefficient CPU and memory usage. However, the Vavr collection library overcomes this by sharing data structure across different versions of a collection.

This is fundamentally different from Java's unmodifiableCollection() from the Collections utility class, which merely provides a wrapper around an underlying collection.

Trying to modify such a collection results in UnsupportedOperationException instead of a new version being created. Moreover, the underlying collection is still mutable through its direct reference."

Java的可变集合带来的麻烦远比它带来的方便多，多线程环境下的不可控，使jdk的设计者不得不使用类似ConcurrentHashMap这样的类来解决之前造成的麻烦。

而过多的集合类无法一一去给它们扩展方便的方法，导致想要方便的处理一个集合，即使Java的版本到了14，仍然不得不先转换成Stream再做处理。

List

vavr的List是不可变的链表，在该链表对象上的操作都会生成一个新的链表对象。
使用Java 8的代码,使用vavr实现相同的功能，则更加直接：

List

```
Arrays.asList(1, 2, 3).stream().reduce((i, j) -> i + j);
IntStream.of(1, 2, 3).sum();

//io.vavr.collection.List
List.of(1, 2, 3).sum();
```

再比如我们常写的，把dto list转换为另一个list，比起一个for循环，使用MapReduce是比较合适的：

Map

```
io.vavr.collection.List<XxxDto> dtoList = ...;
io.vavr.collection.List<Xxx> resultList = dtoList.map(XxxDto::from)
```

Map

举例我们想要过滤一个map，通常我们会怎么做？java集合缺少filterKeys, filterValues这样基本的方法。

Map

```
// java8 with guava
// Map -> Stream -> Filter -> MAP
Map<String, Integer> salary = ImmutableMap.<String, Integer> builder()
                .put("John", 1000)
                .put("Jane", 1500)
                .put("Adam", 2000)
                .put("Tom", 2000)
                .build();
Map<Integer, String> result = salary.entrySet().stream()
        .filter(map -> map.getValues() >= 2000)
        .collect(Collectors.toMap(p -> p.getKey(), p -> p.getValue()));

// vavr
Map<String, String> map1 = HashMap.of("key1", "val1", "key2", "val2", "key3", "val3");
Map<String, String> result = map1.filterValues(v -> v.contains("3"));
```

Vavr和Java互转

最重要的，是vavr的集合类和java的集合类，可以非常方便的互相转化，这对我们已经建设很长时间的项目来说，是很必要的，一个新的、所谓更酷的东西，如果不能跟已有的生态兼容，那么它对我们是没什么吸引力的。

Java Vavr Conversion

```
// Java to Vavr Conversion
java.util.List<Integer> javaList = java.util.Arrays.asList(1, 2, 3, 4);
List<Integer> vavrList = List.ofAll(javaList);

java.util.stream.Stream<Integer> javaStream = javaList.stream();
Set<Integer> vavrSet = HashSet.ofAll(javaStream);


// Vavr to Java Conversion
Integer[] array = List.of(1, 2, 3).toJavaArray(Integer.class);
java.util.Map<String, Integer> map = List.of("1", "2", "3").toJavaMap(i -> Tuple.of(i, Integer.valueOf(i)));
```

Either

Either 表示可能有两种不同类型的值，分别称为左值或右值。只能是其中的一种情况。Either 通常用来表示成功或失败两种情况。惯例是把成功的值作为右值，而失败的值作为左值。可以在 Either 上添加应用于左值或右值的计算。应用于右值的计算只有在 Either 包含右值时才生效，对左值也是同理。

下例根据随机的布尔值来创建包含左值或右值的 Either 对象。Either 的 map 和 mapLeft 方法分别对右值和左值进行计算。

Either

```
import io.vavr.control.Either;
import java.util.concurrent.ThreadLocalRandom;

public class Eithers {

  private static ThreadLocalRandom random = ThreadLocalRandom.current();

  public static void main(String[] args) {
    Either<String, String> either = compute()
        .map(str -> str + " World")
        .mapLeft(Throwable::getMessage);
    System.out.println(either);
  }

  private static Either<Throwable, String> compute() {
    return random.nextBoolean()
        ? Either.left(new RuntimeException("Boom!"))
        : Either.right("Hello");
  }
}
```

Try

Monad的应用。Try 用来表示一个可能产生异常的计算。Try 接口有两个实现类，Try.Success 和 Try.Failure，分别表示成功和失败的情况。Try.Success 封装了计算成功时的返回值，而 Try.Failure 则封装了计算失败时的 Throwable 对象。Try 的实例可以从接口 CheckedFunction0、Callable、Runnable 或 Supplier 中创建。Try 也提供了 map 和 filter 等方法。值得一提的是 Try 的 recover 方法，可以在出现错误时根据异常进行恢复。

下例中，第一个 Try 表示的是 1/0 的结果，显然是异常结果。使用 recover 来返回 1。第二个 Try 表示的是读取文件的结果。由于文件不存在，Try 表示的也是异常。

Try

```
Try<Integer> result = Try.of(() -> 1 / 0).recover(e -> 1);
System.out.println(result);

Try<String> lines = Try.of(() -> Files.readAllLines(Paths.get("1.txt")))
    .map(list -> String.join(",", list))
    .andThen((Consumer<String>) System.out::println);
System.out.println(lines);
```

模式匹配

Match在快说车中的应用

另外夹带一点根本文无关的 我觉得有不少人被Spring带着跑偏了 - 刘欣

参考链接

Introduction to Vavr - baeldung

使用 Vavr 进行函数式编程

vavr：让你像写Scala一样写Java

VAVR：颠覆你的 Java 体验

More about vavr