Java 8 - Optional类深度解析

调用一个方法得到了返回值却不能直接将返回值作为参数去调用别的方法。我们首先要判断这个返回值是否为null，只有在非空的前提下才能将其作为其他方法的参数。这正是一些类似Guava的外部API试图解决的问题。一些JVM编程语言比如Scala、Ceylon等已经将对在核心API中解决了这个问题。

新版本的Java，比如Java 8引入了一个新的Optional类。Optional类的Javadoc描述如下:

> 这是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。

Optional类包含的方法

of

> 为非null的值创建一个Optional。

of方法通过工厂方法创建Optional类。需要注意的是，创建对象时传入的参数不能为null。如果传入参数为null，则抛出NullPointerException 。

```
//调用工厂方法创建Optional实例
Optional<String> name = Optional.of("Sanaulla");
//传入参数为null，抛出NullPointerException.
Optional<String> someNull = Optional.of(null);
```

ofNullable

> 为指定的值创建一个Optional，如果指定的值为null，则返回一个空的Optional。

ofNullable与of方法相似，唯一的区别是可以接受参数为null的情况。示例如下:

```
//下面创建了一个不包含任何值的Optional实例
//例如，值为'null'
Optional empty = Optional.ofNullable(null);
```

isPresent

非常容易理解

> 如果值存在返回true，否则返回false。

类似下面的代码:

```
//isPresent方法用来检查Optional实例中是否包含值
if (name.isPresent()) {
  //在Optional实例内调用get()返回已存在的值
  System.out.println(name.get());//输出Sanaulla
}
```

get

> 如果Optional有值则将其返回，否则抛出NoSuchElementException。

上面的示例中，get方法用来得到Optional实例中的值。下面我们看一个抛出NoSuchElementException的例子:

```
//执行下面的代码会输出: No value present
try {
  //在空的Optional实例上调用get()，抛出NoSuchElementException
  System.out.println(empty.get());
} catch (NoSuchElementException ex) {
  System.out.println(ex.getMessage());
}
```

ifPresent

> 如果Optional实例有值则为其调用consumer，否则不做处理

要理解ifPresent方法，首先需要了解Consumer类。简答地说，Consumer类包含一个抽象方法。该抽象方法对传入的值进行处理，但没有返回值。Java8支持不用接口直接通过lambda表达式传入参数。

如果Optional实例有值，调用ifPresent()可以接受接口段或lambda表达式。类似下面的代码:

```
//ifPresent方法接受lambda表达式作为参数。
//lambda表达式对Optional的值调用consumer进行处理。
name.ifPresent((value) -> {
  System.out.println("The length of the value is: " + value.length());
});
```

orElse

> 如果有值则将其返回，否则返回指定的其它值。

如果Optional实例有值则将其返回，否则返回orElse方法传入的参数。示例如下:

```
//如果值不为null，orElse方法返回Optional实例的值。
//如果为null，返回传入的消息。
//输出: There is no value present!
System.out.println(empty.orElse("There is no value present!"));
//输出: Sanaulla
System.out.println(name.orElse("There is some value!"));
```

orElseGet

> orElseGet与orElse方法类似，区别在于得到的默认值。orElse方法将传入的字符串作为默认值，orElseGet方法可以接受Supplier接口的实现用来生成默认值。示例如下:

```
//orElseGet与orElse方法类似，区别在于orElse传入的是默认值，
//orElseGet可以接受一个lambda表达式生成默认值。
//输出: Default Value
System.out.println(empty.orElseGet(() -> "Default Value"));
//输出: Sanaulla
System.out.println(name.orElseGet(() -> "Default Value"));
```

orElseThrow

> 如果有值则将其返回，否则抛出supplier接口创建的异常。

在orElseGet方法中，我们传入一个Supplier接口。然而，在orElseThrow中我们可以传入一个lambda表达式或方法，如果值不存在来抛出异常。示例如下:

```
try {
  //orElseThrow与orElse方法类似。与返回默认值不同，
  //orElseThrow会抛出lambda表达式或方法生成的异常

  empty.orElseThrow(ValueAbsentException::new);
} catch (Throwable ex) {
  //输出: No value present in the Optional instance
  System.out.println(ex.getMessage());
}
```

ValueAbsentException定义如下:

```
class ValueAbsentException extends Throwable {

  public ValueAbsentException() {
    super();
  }

  public ValueAbsentException(String msg) {
```

```
    super(msg);
  }

  @Override
  public String getMessage() {
    return "No value present in the Optional instance";
  }
}
```

map

map方法文档说明如下:

> 如果有值，则对其执行调用mapping函数得到返回值。如果返回值不为null，则创建包含mapping返回值的Optional作为map方法返回值，否则返回空Optional。

map方法用来对Optional实例的值执行一系列操作。通过一组实现了Function接口的lambda表达式传入操作。如果你不熟悉Function接口，可以参考我的这篇博客。map方法示例如下:

```
//map方法执行传入的lambda表达式参数对Optional实例的值进行修改。
//为lambda表达式的返回值创建新的Optional实例作为map方法的返回值。
Optional<String> upperName = name.map((value) -> value.toUpperCase());
System.out.println(upperName.orElse("No value found"));
```

flatMap

> 如果有值，为其执行mapping函数返回Optional类型返回值，否则返回空Optional。flatMap与map(Funtion)方法类似，区别在于flatMap中的mapper返回值必须是Optional。调用结束时，flatMap不会对结果用Optional封装。

flatMap方法与map方法类似，区别在于mapping函数的返回值不同。map方法的mapping函数返回值可以是任何类型T，而flatMap方法的mapping函数必须是Optional。

参照map函数，使用flatMap重写的示例如下:

```
//flatMap与map(Function)非常类似，区别在于传入方法的lambda表达式的返回类型。
//map方法中的lambda表达式返回值可以是任意类型，在map函数返回之前会包装为Optional。
//但flatMap方法中的lambda表达式返回值必须是Optionl实例。
upperName = name.flatMap((value) -> Optional.of(value.toUpperCase()));
System.out.println(upperName.orElse("No value found"));//输出SANAULLA
```

filter

filter个方法通过传入限定条件对Optional实例的值进行过滤。文档描述如下:

> 如果有值并且满足断言条件返回包含该值的Optional，否则返回空Optional。

读到这里，可能你已经知道如何为filter方法传入一段代码。是的，这里可以传入一个lambda表达式。对于filter函数我们应该传入实现了Predicate接口的lambda表达式。如果你不熟悉Predicate接口，可以参考这篇文章。

现在我来看看filter的各种用法，下面的示例介绍了满足限定条件和不满足两种情况:

```
//filter方法检查给定的Option值是否满足某些条件。
//如果满足则返回同一个Option实例，否则返回空Optional。
Optional<String> longName = name.filter((value) -> value.length() > 6);
System.out.println(longName.orElse("The name is less than 6 characters"));//输出Sanaulla

//另一个例子是Optional值不满足filter指定的条件。
Optional<String> anotherName = Optional.of("Sana");
Optional<String> shortName = anotherName.filter((value) -> value.length() > 6);
//输出: name长度不足6字符
System.out.println(shortName.orElse("The name is less than 6 characters"));
```

一些例子

- 一个综合例子

```
public class OptionalDemo {

  public static void main(String[] args) {
    //创建Optional实例，也可以通过方法返回值得到。
    Optional<String> name = Optional.of("Sanaulla");

    //创建没有值的Optional实例，例如值为'null'
    Optional empty = Optional.ofNullable(null);

    //isPresent方法用来检查Optional实例是否有值。
    if (name.isPresent()) {
      //调用get()返回Optional值。
      System.out.println(name.get());
    }

    try {
      //在Optional实例上调用get()抛出NoSuchElementException。
      System.out.println(empty.get());
    } catch (NoSuchElementException ex) {
      System.out.println(ex.getMessage());
    }

    //ifPresent方法接受lambda表达式参数。
    //如果Optional值不为空，lambda表达式会处理并在其上执行操作。
    name.ifPresent((value) -> {
      System.out.println("The length of the value is: " + value.length());
    });

    //如果有值orElse方法会返回Optional实例，否则返回传入的错误信息。
    System.out.println(empty.orElse("There is no value present!"));
    System.out.println(name.orElse("There is some value!"));

    //orElseGet与orElse类似，区别在于传入的默认值。
    //orElseGet接受lambda表达式生成默认值。
    System.out.println(empty.orElseGet(() -> "Default Value"));
    System.out.println(name.orElseGet(() -> "Default Value"));

    try {
      //orElseThrow与orElse方法类似，区别在于返回值。
      //orElseThrow抛出由传入的lambda表达式/方法生成异常。
```

```
            empty.orElseThrow(ValueAbsentException::new);
        } catch (Throwable ex) {
            System.out.println(ex.getMessage());
        }

        //map方法通过传入的lambda表达式修改Optonal实例默认值。
        //lambda表达式返回值会包装为Optional实例。
        Optional<String> upperName = name.map((value) -> value.toUpperCase());
        System.out.println(upperName.orElse("No value found"));

        //flatMap与map(Funtion)非常相似，区别在于lambda表达式的返回值。
        //map方法的lambda表达式返回值可以是任何类型，但是返回值会包装成Optional实例。
        //但是flatMap方法的lambda返回值总是Optional类型。
        upperName = name.flatMap((value) -> Optional.of(value.toUpperCase()));
        System.out.println(upperName.orElse("No value found"));

        //filter方法检查Optiona值是否满足给定条件。
        //如果满足返回Optional实例值，否则返回空Optional。
        Optional<String> longName = name.filter((value) -> value.length() > 6);
        System.out.println(longName.orElse("The name is less than 6 characters"));

        //另一个示例，Optional值不满足给定条件。
        Optional<String> anotherName = Optional.of("Sana");
        Optional<String> shortName = anotherName.filter((value) -> value.length() > 6);
        System.out.println(shortName.orElse("The name is less than 6 characters"));

    }
}
```

上述代码输出如下:

```
Sanaulla
No value present
The length of the value is: 8
There is no value present!
Sanaulla
Default Value
Sanaulla
No value present in the Optional instance
SANAULLA
SANAULLA
Sanaulla
The name is less than 6 characters
```

- 在 Java 8 中提高 Null 的安全性

假设我们有一个像这样的类层次结构:

```
class Outer {
    Nested nested;
    Nested getNested() {
        return nested;
    }
}
class Nested {
    Inner inner;
    Inner getInner() {
        return inner;
    }
```

```
}
class Inner {
    String foo;
    String getFoo() {
        return foo;
    }
}
```

解决这种结构的深层嵌套路径是有点麻烦的。我们必须编写一堆 null 检查来确保不会导致一个 NullPointerException:

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner != null) {
    System.out.println(outer.nested.inner.foo);
}
```

我们可以通过利用 Java 8 的 Optional 类型来摆脱所有这些 null 检查。map 方法接收一个 Function 类型的 lambda 表达式，并自动将每个 function 的结果包装成一个 Optional 对象。这使我们能够在一行中进行多个 map 操作。Null 检查是在底层自动处理的。

```
Optional.of(new Outer())
    .map(Outer::getNested)
    .map(Nested::getInner)
    .map(Inner::getFoo)
    .ifPresent(System.out::println);
```

还有一种实现相同作用的方式就是通过利用一个 supplier 函数来解决嵌套路径的问题:

```
Outer obj = new Outer();
resolve(() -> obj.getNested().getInner().getFoo());
    .ifPresent(System.out::println);
```

调用 obj.getNested().getInner().getFoo()) 可能会抛出一个 NullPointerException 异常。在这种情况下，该异常将会被捕获，而该方法会返回 Optional.empty()。

```
public static <T> Optional<T> resolve(Supplier<T> resolver) {
    try {
        T result = resolver.get();
        return Optional.ofNullable(result);
    }
    catch (NullPointerException e) {
        return Optional.empty();
    }
}
```

请记住，这两个解决方案可能没有传统 null 检查那么高的性能。不过在大多数情况下不会有太大问题。