链滴

使用 Optional 类来解决 NullPointerException

作者：jianzh5

原文链接：https://ld246.com/article/1600735867186

来源网站：链滴

Optional类是Java 8新增的一个类，用以解决程序中常见的 NullPointerException异常问题。本篇章将详细介绍Optional类，以及如何用它消除代码中的null检查。

避免使用null检查

作为Java开发人员，几乎所有人都遇到过 NullPointerException异常，大多数人遇到 NullPointerException异常时都会在异常出现的地方加上if代码块来判断值不为空，比如下面的代码：

```
public void bindUserToRole(User user) {
    if (user != null) {
        String roleId = user.getRoleId();
        if (roleId != null) {
            Role role = roleDao.findOne(roleId);
            if (role != null) {
                role.setUserId(user.getUserId());
                roleDao.save(role);
            }
        }
    }
}
```

这是比较普遍的做法，为了避免出现 NullPointerException异常，手动对可能为null值进行了处理，过代码看起来非常糟糕，业务逻辑被淹没在if逻辑判断中，也许下面的代码看起来可读性稍好一些：

```
public String bindUserToRole(User user) {
    if (user == null) {
        return;
    }

    String roleId = user.getRoleId();
    if (roleId == null) {
        return;
    }

    Role = roleDao.findOne(roleId);
    if (role != null) {
        role.setUserId(user.getUserId());
        roleDao.save(role);
    }
}
```

上面的代码避免了深层的if语句嵌套，但本质上是一样的，方法内有三个不同的返回点，出错后调试不容易，因为你不知道是那个值导致了 NullPointerException异常。

基于上面的原因，Java 8中引入了一个新的类Optional，用以避免使用null值引发的种种问题。

Optional类

java.util.Optional<T>类是一个封装了 Optional值的容器对象，Optional值可以为 null，如果值存，调用 isPresent()方法返回true，调用 get()方法可以获取值。

创建Optional对象

Optional类提供类三个方法用于实例化一个Optional对象，它们分别为 empty()、of()、ofNullable()这三个方法都是静态方法，可以直接调用。

- empty()方法用于创建一个没有值的Optional对象：

```
Optional<String> emptyOpt = Optional.empty();
```

empty()方法创建的对象没有值，如果对emptyOpt变量调用 isPresent()方法会返回false，调用get()法抛出NullPointerException异常。

- of()方法使用一个非空的值创建Optional对象：

```
String str = "Hello World";
Optional<String> notNullOpt = Optional.of(str);
```

- ofNullable()方法接收一个可以为null的值：

```
Optional<String> nullableOpt = Optional.ofNullable(str);
```

如果str的值为null，得到的nullableOpt是一个没有值的Optional对象。

提取Optional对象中的值

如果我们要获取User对象中的roleId属性值，常见的方式是直接获取：

```
String roleId = null;
if (user != null) {
   roleId = user.getRoleId();
}
```

使用Optional中提供的map()方法可以以更简单的方式实现：

```
Optional<User> userOpt = Optional.ofNullable(user);
Optional<String> roleId = userOpt.map(User::getRoleId);
```

使用orElse()方法获取值

Optional类还包含其他方法用于获取值，这些方法分别为：

- orElse()：如果有值就返回，否则返回一个给定的值作为默认值；
- orElseGet()：与 orElse()方法作用类似，区别在于生成默认值的方式不同。该方法接受一个Supplie<? extends T>函数式接口参数，用于生成默认值；
- orElseThrow()：与前面介绍的 get()方法类似，当值为null时调用这两个方法都会抛出NullPointerException异常，区别在于该方法可以指定抛出的异常类型。

下面来看看这三个方法的具体用法：

```
String str = "Hello World";
Optional<String> strOpt = Optional.of(str);
String orElseResult = strOpt.orElse("Hello Shanghai");
String orElseGet = strOpt.orElseGet(() -> "Hello Shanghai");
String orElseThrow = strOpt.orElseThrow(
```

```
        () -> new IllegalArgumentException("Argument 'str' cannot be null or blank."));
```

此外，Optional类还提供了一个ifPresent()方法，该方法接收一个Consumer<? super T>函数式接，一般用于将信息打印到控制台：

```
Optional<String> strOpt = Optional.of("Hello World");
strOpt.ifPresent(System.out::println);
```

使用filter()方法过滤

filter()方法可用于判断Optional对象是否满足给定条件，一般用于条件过滤：

```
Optional<String> optional = Optional.of("lw900925@163.com");
optional = optional.filter(str -> str.contains("164"));
```

在上面的代码中，如果 filter()方法中的Lambda表达式成立，filter()方法会返回当前Optional对象值否则，返回一个值为空的Optional对象。

如何正确使用Optional

通过上面的例子可以看出，Optional类可以优雅的避免 NullPointerException带来的各种问题，不，你是否真正掌握了Optional的用法？假设你试图使用Optional来避免可能出现的 NullPointerException异常，编写了如下代码：

```
Optional<User> userOpt = Optional.ofNullable(user);
if (userOpt.isPresent()) {
    User user = userOpt.get();
    // do something...
} else {
    // do something...
}
```

坦白说，上面的代码与我们之前的使用if语句判断空值没有任何区别，没有起到Optional的正真作用：

```
if (user != null) {
    // do something...
} else {
    // do something...
}
```

当我们从之前版本切换到Java 8的时候，不应该还按照之前的思维方式处理null值，Java 8提倡函数编程，新增的许多API都可以用函数式编程表示，Optional类也是其中之一。这里有几条关于Optiona使用的建议：

- **尽量避免在程序中直接调用Optional对象的get()和isPresent()方法；**
- **避免使用Optional类型声明实体类的属性；**

第一条建议中直接调用get()方法是很危险的做法，如果 Optional的值为空，那么毫无疑问会抛出 NullPointerException异常，而为了调用 get()方法而使用 isPresent()方法作为空值检查，这种做法与传的用if语句块做空值检查没有任何区别。

第二条建议避免使用 Optional作为实体类的属性，它在设计的时候就没有考虑过用来作为类的属性如果你查看Optional的源代码，你会发现它没有实现 java.io.Serializable接口，这在某些情况下是很

要的（比如你的项目中使用了某些序列化框架），使用了 Optional作为实体类的属性，意味着他们能被序列化。

下面我们通过一些例子讲解Optional的正确用法：

正确创建Optional对象

上面提到创建Optional对象有三个方法，empty()方法比较简单，没什么特别要说明的。主要是 of()和fNullable()方法。当你很确定一个对象不可能为null的时候，应该使用 of()方法，否则，尽可能使用 oNullable()方法，比如：

```
public static void method(Role role) {
    // 当Optional的值通过常量获得或者通过关键字new初始化，可以直接使用of()方法
    Optional<String> strOpt = Optional.of("Hello World");
    Optional<User> userOpt = Optional.of(new User());

    // 方法参数中role值不确定是否为null，使用ofNullable()方法创建
    Optional<Role> roleOpt = Optional.ofNullable(role);
}
```

orElse()方法的使用

```
return str != null ? str : "Hello World"
```

上面的代码表示判断字符串str是否为空，不为空就返回，否则，返回一个常量。使用Optional类可表示为：

```
return strOpt.orElse("Hello World")
```

简化if-else

```
User user = ...
if (user != null) {
    String userName = user.getUserName();
    if (userName != null) {
        return userName.toUpperCase();
    } else {
        return null;
    }
} else {
    return null;
}
```

上面的代码可以简化成：

```
User user = ...
Optional<User> userOpt = Optional.ofNullable(user);

return userOpt.map(User::getUserName)
            .map(String::toUpperCase)
            .orElse(null);
```

总结一下，新的Optional类让我们可以以函数式编程的方式处理null值，抛弃了Java 8之前需要嵌套

量if-else代码块，使代码可读性有了很大的提高。

原文链接：使用 Optional 类来解决 NullPointerException