



下载APP



## 14 | 禁止空指针，该怎么避免崩溃的空指针？

2021-12-17 范学雷

《深入剖析Java新特性》

课程介绍 &gt;



讲述：范学雷

时长 10:03 大小 9.22M



你好，我是范学雷。今天，我们讨论 Java 的空指针。

我们都知道空指针，它的发明者开玩笑似的，称它是一个价值 10 亿美元的错误；同时呢，他还称 C 语言的 get 方法是一个价值 100 亿美元的错误。空指针真的错得这么厉害吗？get 方法又有什么问题？我们能够在 Java 语言里改进或者消除空指针吗？

我们从阅读案例开始，来看一看该怎么理解这些问题，以及怎么降低这些问题的影响。

### 阅读案例




通常地，一个人的姓名包括两个部分，姓（Last Name）和名（First Name）。在有些文化里，也会使用中间名（Middle Name）。所以，我们通常可以使用姓、名、中间名这三

个要素来标识一个人的姓名。用代码的形式表示出来，就是下面的代码这样。

 复制代码

```
1 public record FullName(String firstName,
2     String middleName, String lastName) {
3     // blank
4 }
```


中间名并不是必需的，因为有的人使用中间名，有的人不使用。现在我们假设，需要判断一个人的中间名是不是黛安（Diane）。这个判断的逻辑，可能就像下面的代码这样。

 复制代码

```
1 private static boolean hasMiddleName(
2     FullName fullName, String middleName) {
3     return fullName.middleName().equals(middleName);
4 }
```


这个判断的逻辑是没有问题的。但是它的代码实现，就存在没有校验空指针的错误。如果一个人不使用中间名，那么 `FullName.middleName` 这个方法的返回值就是一个空指针。如果一个对象是空指针，那么调用它的任何方法，都会抛出空指针异常（`NullPointerException`）。

我们可以试着使用 JDK 11 的 JShell，看一看空指针异常的异常信息是什么样子的。

 复制代码

```
1 $ jshell -v
2 | Welcome to JShell -- Version 11.0.13
3 | For an introduction type: /help intro
4
5 jshell> String a = null;
6 a ==> null
7 | created variable a : String
8
9 jshell> a.equals("b");
10 | Exception java.lang.NullPointerException
11 |     at (#2:1)
```

然后，我们再试试看 JDK 17 里，空指针异常信息是什么样子的。

 复制代码

```
1 $ jshell -v
2 | Welcome to JShell -- Version 17
3 | For an introduction type: /help intro
4
5 jshell> String a = null;
6 a ==> null
7 | created variable a : String
8
9 jshell> a.equals("b");
10 | Exception java.lang.NullPointerException: Cannot invoke "String.equals(Object)"
11 | at (#2:1)
```


对比一下，我们可以看到，JDK 17 的异常信息里，包含了调用者（`REPL.JShell11.a`）和被调用者（`String.equals(Object)`）的信息；而 JDK 11 里，调用者的信息需要从调用堆栈里寻找，而且没有被调用者的信息。

这是空指针异常的一个小的改进。它简化了问题排查的流程，提高了问题排查的效率。

好的，我们再回到主题，看一看空指针异常到底有什么危害。按照我们前面讨论过的中间名的逻辑，有的人不使用中间名。那么，如果一个对象的中间名是空值，也就意味着他没有中间名。可是，在上面的实现代码里，如果中间名是空值，`hasMiddleName` 抛出了空指针异常，而不是通过返回值来表示这个对象没有中间名。

这当然是一个错误。我们需要检查返回值有没有可能是空指针，然后才能继续使用返回值。这是一个 C 语言或者 Java 语言软件工程师需要掌握的基本常识。当然，这也是一个我们编码的时候，需要遵守的纪律。

检查返回值有没有可能是空指针需要额外的代码，而且不符合我们的思维习惯。下面的代码，我添加了空指针的检查，这就让它看起来就有点臃肿。这就是精准控制的代价。

 复制代码

```
1 private static boolean hasMiddleNameImplA(
2     FullName fullName, String middleName) {
3     if (fullName.middleName() != null) {
4         return fullName.middleName().equals(middleName);
5     }
6
7     return middleName == null;
8 }
```

**空指针的问题，其实是我们人类行为方式的一个反映。**无论是纪律还是常识，如果没有配以强制性的手段，都没有办法获得 100% 的执行。如果不能 100% 地执行，一个危害就会从一个小小的局部，蔓延到一个庞大的系统。

今天的应用程序，我们几乎可以肯定地说，都是由很多小的部件组合起来的。其中，99% 以上的部件，我们都不了解，甚至都不知道它们的存在。任何一个小的部件出了问题，都会蔓延开来，酝酿出一个更大的问题。


在 C 语言和 Java 语言里，存在着大量的空指针。不管我们怎么努力，也不管我们经验多么丰富，总是会时不时地就忘了检查空指针。而忘了检查这样的小错误，很可能就蔓延成严重的事故。所以，空指针发明者称它是一个价值 10 亿美元的错误。

那有什么办法能够降低空指针的负面影响呢？

## 避免空指针


降低空指针的负面影响的最重要的办法，就是不要产生空指针。没有空指针的代码，代码更简洁，风险也更小。

比如说，我们可以使用空字符串来替代字符串的空指针。如果用这种思路，我们就可以把阅读案例里 FullName 档案类，修改成不使用空指针的版本了。

 复制代码

```
1 public record FullName(String firstName,
2     String middleName, String lastName) {
3     public FullName(String firstName,
4         String middleName, String lastName) {
5         this.firstName = firstName == null ? "" : firstName;
6         this.middleName = middleName == null ? "" : middleName;
7         this.lastName = lastName == null ? "" : lastName;
8     }
9 }
```

这样，我们就不用检查空指针了；因此，也就不担心空指针带来的问题了。所以，代码的使用也就变得简洁了起来。

 复制代码

```
1 private static boolean hasMiddleName(  
2     FullName fullName, String middleName) {  
3     return fullName.middleName().equals(middleName);  
4 }
```

在很多场景下，我们都可以使用空值来替代空指针，比如，空的字符串、空的集合。在 API 设计的时候，如果碰到了使用空指针的规范或者代码，我们要停下来想一想，有没有替代空指针的办法？如果能够避免空指针，我们的代码会更健壮，更容易维护。

## 强制性检查

不过，不是在所有的情况下我们都能够避免空指针的。如果空指针不能避免，降低空指针的负面影响的另外一个办法，就是在使用空指针的时候，执行强制性的检查。所谓强制性的检查，对于编程语言来说，指的是我们通常能够依赖的是编译器的能力，以及新的接口设计思路。

## 不尽人意的 Optional


在 JDK 8 正式发布，而后再在 JDK 9 和 11 持续改进的 Optional 工具类是 JDK 试图降低空指针风险的一个尝试。

设计 Optional 的目的，是希望开发者能够先调用它的 Optional.isPresent 方法，然后再调用 Optional.get 方法获得目标对象。按照设计者的预期，这个 Optional 类的使用应该像下面的代码这样。

 复制代码


```
1 private static boolean hasMiddleName(  
2     FullName fullName, String middleName) {  
3     if (fullName.middleName().isPresent()) {  
4         return fullName.middleName().get().equals(middleName);  
5     }  
6  
7     return middleName == null;  
8 }
```

当然，我们还需要修改 FullName 的 API，就像下面的代码这样。

 复制代码

```
1 public final class FullName {
2     // snipped
3     public Optional<String> middleName() {
4         return Optional.ofNullable(middleName);
5     }
6     // snipped
7 }
```

遗憾的是，我们也可以不按照预期的方式使用它，比如下面的代码，我们就没有调用 `Optional.isPresent` 方法，而是直接使用了 `Optional.get` 方法。这不在设计者的预期之内，但是这是合法的代码。

 复制代码

```
1 private static boolean hasMiddleName(FullName fullName, String middleName) {
2     return fullName.middleName().get().equals(middleName);
3 }
```

如果 `Optional` 指代的对象不存在，或者是个空指针，`Optional.get` 方法就会抛出 `NoSuchElementException` 异常。和空指针异常一样，这个异常也是运行时异常。虽然这个异常的名字不再叫做空指针异常，但它实质上依然是空指针异常。当然，这个异常也具有和空指针异常相同的问题。

如果你对比一下使用空指针的代码和使用 `Optional` 类的代码，就会发现这两个类型的代码，不论是正确的使用方法还是错误的使用方法，它们在形式上是相似的。`Optional` 带来了不必要的复杂性，然而它并没有简化开发者的工作，也没有解决掉空指针的问题。

被寄予厚望的 `Optional` 的设计，不能尽如人意。

## 新特性带来的新希望

那么，对于空指针的检查，我们能不能借助编译器，让它变得更强硬一点呢？下面的例子，就是我们使用新特性来解决空指针问题的一个新的探索。

我们希望返回值的检查是强制性的。如果不检查，就没有办法得到返回值指代的真实对象。实现的思路，就是使用封闭类和模式匹配。



首先呢，我们定义一个指代返回值的封闭类 `Returned`。为什么使用封闭类呢，因为封闭类的子类可查可数。可查可数，也就意味着我们可以有简单的模式匹配。

[复制代码](#)

```
1 public sealed interface Returned<T> {
2     Returned.Undefined UNDEFINED = new Undefined();
3
4     record ReturnValue<T>(T returnValue) implements Returned {
5     }
6
7     record Undefined() implements Returned {
8     }
9 }
```

然后呢，我们就可以使用 `Returned` 来表示返回值了。

[复制代码](#)

```
1 public final class FullName {
2     // snipped
3     public Returned<String> middleName() {
4         if (middleName == null) {
5             return Returned.UNDEFINED;
6         }
7
8         return new Returned.ReturnValue<>(middleName);
9     }
10    // snipped
11 }
```

最后，我们来看看 `Returned` 是怎么使用的。

[复制代码](#)

```
1 private static boolean hasMiddleName(FullName fullName, String middleName) {
2     return switch (fullName.middleName()) {
3         case Returned.Undefined undefined -> false;
4         case Returned.ReturnValue rv -> {
5             String returnedMiddleName = (String)rv.returnValue();
6             yield returnedMiddleName.equals(middleName);
7         }
8     };
9 }
```

这种使用了封闭类和模式匹配的设计，极大地压缩了开发者的自由度，强制要求开发者的代码必须执行空指针的检查，只有这样才能编写下一步的代码。这种看似放弃了灵活性的设计，恰恰把开发者从低级易犯的错误中解救了出来。不论是对写代码的开发者，还是对读代码的开发者来说，这都是一件好事。

好事情的背后，往往都意味着一些妥协。比如说吧，使用空指针的代码，我们可以轻松地使用档案类；使用 Optional 和 Returned 的代码，我们就要重新回到传统的类上面来了。

无论档案类、封闭类还是模式匹配，对于 Java 来说，都还是新鲜的技术。要想让这些技术之间熟练配合，还需要一些这样或者那样的磨练，包括不停地改进，组合效应的新研究等。

## 总结

好，到这里，我来做个小结。前面，我们讨论了空指针带来的问题，以及降低空指针负面影响的一些办法。

总体来说，在我们的代码里，尽量不要产生空指针。没有空指针，也就没有了空指针的烦恼。

如果避免不了空指针，我们就要看看能不能执行强制性的检查。比如使用封闭类和模式匹配的组合形式，让编译器和接口设计帮助我们实施这种强制性。

如果不能实施强制性的检查，我们就要遵守空指针的编码纪律。也就是说，对于可能是空指针的变量，先检查后使用。

如果面试中聊到了空指针的问题，你可以聊一聊空指针的危害，以及我们这一次学习到的解决办法。

## 思考题

今天，我们使用封闭类和模式匹配来降低空指针危害的例子，有点像我们前面提到过的替代异常处理的错误码方案。其实，一个带有返回值的方法，通常要考虑三种情况：正常情况、异常情况以及空指针。我们可以把空指针解读为正常情况，也可以解读为异常情况。



如果要在返回值这个封闭类里考虑进这三种情况，我们该怎么设计这个封闭类以及它的许可类呢？这是我们这一次的思考题。

为了方便你阅读，我把我们这次讨论用到的 Returned 的实现代码拷贝到了下面。你可以在此基础上修改。

[复制代码](#)

```
1 public sealed interface Returned<T> {
2     Returned.Undefined UNDEFINED = new Undefined();
3
4     record ReturnValue<T>(T returnValue) implements Returned {
5     }
6
7     record Undefined() implements Returned {
8     }
9 }
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！

注：本文使用的完整的代码可以从 [🔗 GitHub](#) 下载，你可以通过修改 [🔗 GitHub](#) 上 [🔗 review template](#) 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在 [🔗 空指针专用的代码评审目录](#) 下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 nullp/review/xuelel 的目录下面。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

[生成海报并分享](#)

[👍 赞 1](#) [🔗 提建议](#)

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 13 | 外部函数接口，能不能取代 Java 本地接口？

## 精选留言 (4)

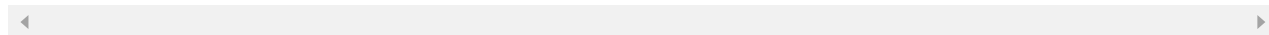
[写留言](#)**aoe**

2021-12-17

开始使用 Optional 的时候确实也遇到了像文中说的空指针（以为使用后就能神奇的消除空指针），后来还错用了 Optional.of() 又遇到空指针，直到最后才发现 Optional.ofNullable() 才是能最大限度避免空指针的方法。

如果有需要返回 null 的地方，可以通过一个中间层封装其状态，避免直接返回 null；或者今后 JDK 升级功能后像 Go 一样没有空指针异常了，但是 Go 的 panic 一样能让程...  
展开 ∨

作者回复: Optional一言难尽啊



共 2 条评论 >

👍 1

**bigben**

2021-12-17

用Returned并不比Optional有优势，代码比较冗长，整个公司统一比较难。java语言级为啥不提供更简单的方式呢？其它语言有可借鉴的好的方法吗？

展开 ∨



2021-12-17

optional我是体验不出强大之处，代码量还是那么多，没怎么降，不过Optional.ifPresent()有时还是有点用，但没优势

展开 ∨

**kimoti**

2021-12-17

感觉那个Optional就是个鸡肋,你不还是得调一下isPresent方法吗？这和判断一下是否为NULL不是一样吗？

展开 ∨

共 1 条评论 >



