链滴

Java 中如何避免空指针异常?

作者：laoma

原文链接：https://ld246.com/article/1612322705891

来源网站：链滴

"null很恶心。" -Doug Lea(道格·利)

"Null 引用一直是个坏主意，从来没发挥过什么正面作用。这是一个令我追悔莫及的错误。" - Sir C. A. R. Hoare(托尼·霍尔), 在评价他对null的发明时说。

编程语言中null引用的十亿美元错误

后悔发明Null：堪称CS史上最严重错误，至少造成10亿美金损失

1. 什么是NPE?

NPE是**空指针异常** NullPointerException的缩写，是一个影响非常广泛，破坏性非常强的东西。对于个Java开发工程师来说，避免NPE是一个值得研究的课题。

作为一名合格的Java开发工程师都，我们需要严肃认真的对待NPE问题，NPE不除，软件质量不提。

2. Java中的null代表什么?

null表示啥都没有，无。

现实生活中， 我在对一个人说话时，我们会看看这个人在不在？如果不在， 我们就不说了。写代码， 我们要做的也是：在对一个对象做一个操作时，先检查一下这个对象在不在。

然而，我们在写代码的过程中经常范的错误就是：**对着空气说话。**

每次操作一个对象前对其进行空检查是可行的， 但是这种高强度的检查频率会带来两个问题：

- 增加代码长度

当我们把大量的代码放在检查空值上面时，会大大的增加代码的长度；

如：一个方法100行代码，有50行是检查各种参数，30行进行异常处理，20行调用方法进行业务处理

这样的代码将来如何维护、阅读呢？岂不是刚写出来就知道日后定会被其它程序员骂出翔。

- 减少写代码的乐趣

注意力是非常保贵的资源，在混乱的办公场景中，嘀嘀响的工作群、一会儿不看就有几百条未读的划群、吵闹的产品经理、测出Bug的测试人员、分派任务的Leader... ... 我们的注意力无时无刻被打扰，不容易集中注意力开始写业务代码，大半的时间还要编写各种对空的检查，我们写代码的乐趣全都被磨掉了。

所以Java中的Null代表着问题、代表着麻烦、代表着各种坑，代表着你在半夜被电话叫醒解决线上问，代表着你的技术水平被拉低几个段位。

那么如何解决NPE?

3. 如何解决NPE?

答案是：**没法解决！！！** 或者说没法从根本上解决。

我们能做的是：认真面对每一个业务需求，认真面对每一行代码，写代码时小心谨慎，从**态度**上重视PE问题，用各种限制手段降低NPE问题的影响，一定不能在写代码时放飞自我，放飞自我的同时，也NPE一起放飞了出去。

一定要记住我们是工程师，我们做的是工程，不是玩具。

工程师要严谨、认真，具有三心：细心、耐心、责任心。

下面列出一些经验之谈，虽然不能完全解决NPE问题，用的好的话，还是能最大程度的减少NPE的发。

3.1 遵守一些开发约定

所有集合对象在声明时即进行实例化

```
// 使用Guava中的工具类实例化List
private List<UserProfile> userList = Lists.newArrayList();

// 直接实例化
private List<UserProfile> userList1 = new ArrayList<>();

// 使用Guava中的工具类实例化Map
private Map<String, UserProfile> userMap = Maps.newHashMap();

// 直接实例化
private Map<String, UserProfile> userMap1 = new HashMap<>();
```

返回集合类型时，如果没有数据，返回空集合对象。

```
public List<String> doSomething() {
  List<String> returnValue;
   ....
  return returnValue == null? Lists.newArrayList(): returnValue;
}
```

为了方便，可以自己写一个简单的返回空集合的方法

```
public <T> T nonNullVal(T val, T def) {
    return Objects.isNull(val)? def: val;
}

// 或者这样
public <T> T nonNullList(T val) {
    return val == null? Lists.newArrayList(): val;
}
```

上面的代码就变为下面这样了

```
public List doSomething() {
    List returnValue;
    ....
    return nonNullVal(returnValue, Lists.newArrayList());

    // 如果返回的List不会被做插入数据等操作，也可以直接返回空列表
    return nonNullVal(returnValue, Lists.emptyList());

    return nonNullList(returnValue);
}
```

将变量设置为[有业务含意]的默认值

这样会最大程度上减少对一个对象进行初始化的复杂度：你不用手动设置一些默认值了。

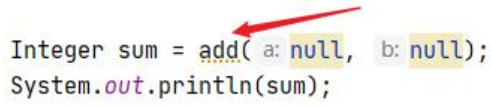
```
// 姓名默认为空字符串，如果在实际业务代码编写时没有填写姓名，
// 空字符串也能表明没有填写姓名
private String name = "";

// 如果业务上规定0为没有填写年龄，可以默认将年龄设置为0
private Integer age = 0;

// 创建时间在对象实例时，默认赋值为当前
private LocalDateTime createTime = LocalDateTime.now();
```

对可能为空的变量增加提示信息

1. 增加Spring注解　@NonNull@Nullable，IDE会作异常提示；
2. 在注释中标明参数不可为空，提醒调用者小心NPE

```
/**
 * 两个整数相加
 * @param a 不能为空
 * @param b 不能为空
 * @return 相加结果
 */
private Integer add(@NonNull Integer a, @NonNull Integer b) {
    return a + b;
}
```

在Idea中会进行提示，如下图所示：



集合中不存储 null；使用 Map时，不将 null作为Key

一般情况下，列表对象中不存储null，这样就不会给处理列表元素的程序埋下深坑。

Map对象的元素中，Key和Value都不使用null。

使用null有明确的业务含意时，在任何时候、任何地方都可以使用null

3.2 集合为空的检查问题

通常集合检查的方式为：1. 检查集合对象是否为空， 2. 检查集合内元素数量是否为0

```
List list = null;
if (list == null || list.isEmpty()) {
    System.out.println("List为空");
}

Map map = new HashMap();
if (map == null || map.isEmpty()) {
    System.out.println("Map为空");
}

Set set = new HashSet();
if (set == null || set.isEmpty()) {
    System.out.println("Set为空");
}
```

因为List、Set都是Collection，所以我们可以统一写验证为空的方法：

```
public boolean isEmpty(Collection collection) {
    return collection == null || collection.isEmpty();
}

public boolean isEmpty(Map map) {
    return map == null || map.isEmpty();
}
```

3.3 使用JDK8的Objects来操作对象

因为NPE问题实在是太严重了，所以JDK中出现了Objects，提供了一组避免产生NPE的API。

下面两种情况，Java程序员应该都遇到过：

1. 在将一个对象转为字符串时，对象不存在；

```
Integer age;
.....一堆操作
```

```
age.toString(); // 咦，此处NPE
```

2. 比对两个对象是否相等时，主比较对象为空；

```
User me = new User();
User loginUser = getLoginUser(); // 如果此方法返回了null
if (loginUser.equals(me)) { // 咦，此处NPE

}
```

Objects针对上面的情况，给了一个解决方案，从一定程度上避免了NPE问题。如：

```
String val1 = null;
String val2 = "hello";
if (Objects.equals(val1, val2)) { // 你不用再对null作检查了
    // 如果两个值等，进入这里。如果两个值都是null，判定两个值是相等的。
} else {
    // 如果不相等，会进入这里。
};

String val4 = null;
String val5 = Objects.toString(val4);// 如果val4是null，返回 ” null “字符串

// 如果不想变成 "null"，可以使用下面的方法，会返回一个替换
String val5 = Objects.toString(val4, "");
```

除了 Objects.equals() 和 Objects.toString()外，Objects还提供了下面一些方法来对null作检查，在常的开发中可以尝试一下。

判断一个对象是否为空：

```
// 这是一个null变量
String value = null;
// 检查变量是否为空
if (Objects.isNull(value)) {
    // 对象为空,进入这里
}

// 检查变量是否不为空
if (Objects.nonNull(value)) {
    // 对象不为空,进入这里
}
```

【检查一个对象是否为空，如果为空抛出异常】：

这个方法在验证参数是否为空时比较有效，但是文章后面的断言部分会**更有效。**

```
// 这是一个null变量
String value = null;

// 过滤一下变量,如果为空就抛出异常
value = Objects.requireNonNull(value);

// 过滤一下变量,如果为空就抛出异常,异常带个消息
value = Objects.requireNonNull(value, "抛出的异常带着这个消息");
```

```
// 可以在方法中作校验参数使用
public List<String> doSomething(String param) {
    Objects.requireNonNull(param);
    // ....
}
```

计算HashCode值，避免对空对象计算Hash值抛NPE：

```
String value = null;
int hashCode = Objects.hashCode(value);
// hasCode 结果为1
```

3.4 使用Optional来处理空对象

Optional是JDK提供的一个用于处理空对象的实践，使用合理的话能够在一定程度上避免NPE的产生。

可以把 Optional看作一个对象的包装对象，通过这个包装对象来操作原对象，一方面Optional强制原对象作判空检查，另一方面强制开发人员重视对 null的处理，从技术与态度两方面来避免NPE的发。

下面用实例来讲一下Optional的用法：

Optional对象的实例化：

```
Integer value = 6;
Optional<Integer> op = Optional.of(value);
```

上面的代码中，如果value是null，会抛出NPE，为了避免这种无谓的NPE，可以使用下面的方式来实化：

```
// 如果value==null,返回 Optional.empty()
Optional<Integer> op = Optional.ofNullable(value);
```

如何从Optional中取出原对象?

```
// 直接取值，虽然简单，但是如果为空时，还是避免不了NPE
Integer i = op.get()

// 增加默认值的取法，如果op不为空，返回op中的原值，如果op为空，返回999
Integer i = op.orElse(999);

// 如果默认值需要经过一系列的操作，那么可以使用lambda表达式来完成
Integer i = op1.orElseGet(() -> {
    .... 一堆操作
    return Optional.of(999); // 返回一个Optional对象
});

// 如果为空时不想返回默认值，想直接抛出一个自定义异常呢？按下面的来
Integer i = op.orElseThrow(() -> {
    return new BusinessException("如果为空, 抛出我");
});
```

如何检查一个对象是否为空

```
// 最初级的检查方式，比较复杂
if (op1.isPresent()) {
   // 不为空时进入这里做逻辑处理
}

// 使用lambda表示式，简单处理
op1.ifPresent(v -> {
    // 如果value不为null, 执行这里面的代码段. 可以替换 if(op1.isPresent()){....}
    System.out.println(v);
});
```

Optional的的使用方法已经讲述清楚了，下面看看如何在业务开发中应用Optional呢?

未改造前代码：

```
public User getLoginUser() {
    // ... 一堆业务处理
    return user;
}

// 调用代码
User user = getLoginUser();
if (user != null) {
    // 正常业务逻辑
} else {
    // 用户为空的处理，比如抛出异常
}
```

改造后的代码：

```
public Optional<User> getLoginUser() {
    // ... 一堆业务处理
    return Optional.ofNullable(user);
}

// 调用代码
Optional<User> user = getLoginUser();
user.XXXX();  // 不能直接操作，因为Optional对User做了包装，强制使用下面的几种方式来处理

// 使用方法1，跟原来相比更复杂了
if (user.isPresent()) {
    // 正常业务逻辑
    User u = user.get();
    u.XXXX();
} else {
    // 用户为空的处理，比如抛出异常
    throw new RuntimeException("没有找到用户");
}

// 使用方法2，用户为空， 抛出异常
User u = user.orElseThrow(() -> {
    return BusinessException("没有找到用户");
});
u.XXXX();
```

```
// 使用方法3，用户为空，使用默认用户，me是一个默认User对象。
User u = user.orElse(me);
u.XXXX();
```

3.5 使用断言类来校验参数

什么是断言

不知道别人是怎么理解断言的，我是很长一段时间都不能理解什么是断言，想理解断言，得从它的英单词说起：Assert，中文翻译是：

明确肯定; 断言; 坚持自己的主张; 表现坚定; 维护自己的权利(或权威);

我理解断言就是：**在程序中明确肯定的一些事情，如果没成功，程序中断。**

什么是需要明确肯定的呢？一些事关业务成败的条件是要明确肯定的。如果达不到这个条件，业务就法顺利完成，需要中断业务处理（反正也执行不成功，也没有必要执行下去了。）

具体表现在程序中就是：**在业务处理之前的前置条件校验。**

现在有很多三方库提供了方便的断言工具类，下面挑选出两个应用广泛的，大家在项目中几乎默认引的两个库：

Guava中的PreConditions断言类

PreCondition类提供的主要的静态方法列表为：

方法名	描述	失败时抛出异常
checkArgument(boolean)	检查参数 boolean 是否为 true 。 用验证方法的参数是否有效。	IllegalArgumentException
checkNotNull(T)	检查参数值是否为非null。直接返回这个值，因你可以内联地使用 checkNotNull(value) 。	NullPointerException
checkState(boolean)	检查object的某些状态，不依赖方法的参数例如： Iterator 在调用 remove 前可以用这个方法检查 next 是否被调用过。	llegalStateException
checkElementIndex(int index, int size)	检查参数 index 是否是定size的list或string、array等的合法index。一个合法index必须>=0且=0且<=size。不用把list、strng或array直接传参进来；只要传它的size即可。返回 index 。	IndxOutOfBoundsException
checkPositionIndexes(int start, int end, int size)	检查参数构成的[start, end) 是给定size的list或string、array等得合法子集。抛出exception时带有自己的错误信息。	ndexOutOfBoundsException

下面给出一个使用的示例：

```
public List<String> doSomething(String param) {
    // 如果param为空，抛出异常
    Preconditions.checkArgument(Objects.nonNull(param), "参数不能为空");
```

PreConditions的所有方法都没有明确指明是以 true 还是 false 为标准， 以至于我每次使用时都需要一下代码的实现。所以从个人角度来说PreConditions容易产生误解，而Spring中的Assert类则没有个问题，下面大家可以瞅瞅Spring的Assert类。

Spring中的 Assert断言类

Assert类提供的主要的静态方法列表：

方法名	描述	失败时抛出异常
isNull(Object object, String message)	object不为空，抛出异常	llegalArgumentException
notNull(Object object, String message)	object为空，抛出异常	llegalArgumentException
hasLength(String text, String message)	text是空字符串，抛出	常 IllegalArgumentException
hasText(String text, String message)	不包含空白字符串，抛出	常 IllegalArgumentException
doesNotContain(String textToSearch, String substring, String message)	extToSearch中包含substring，抛出异常	IllegalArgumentExcepti n
notEmpty(Object[] array, String message)	array为空或长度为1	抛出异常 IllegalArgumentException
noNullElements(Object[] array, String message)	array中包含nul	元系，抛异常 IllegalArgumentException
notEmpty(Collection collection, String message)	collection不	含元素，抛出异常 IllegalArgumentException
notEmpty(Map map, String message)	map中包含null，抛出异常	llegalArgumentException
isInstanceOf(Class type, Object obj, String message)	如果obj	是type类型，抛出异常 IllegalArgumentException
isAssignable(Class superType, Class subType, String message)	ubType不是superType子类，抛出异常	IllegalArgumentException
state(boolean expression, String message)	expression不为tru	抛出异常 IllegalStateException
isTrue(boolean expression, String message)	expression不为tru	抛出异常 IllegalArgumentException

```
public List<String> doSomething(String param) {
    // 如果param为空，抛出异常
    Assert.notNull(param, "param不能为空");
```

总结

避免NPE问题的法宝不是工具，而是态度。我们应该对代码有敬畏之心，重视每一行代码，每一个需，千万不能怀有“这个很简单”的想法，就像潘加宇老师说的：“所有卖钱的系统就没有简单的”。们拿着工资写的代码都是要给公司赚钱的，都是不简单的。