> 本文由 简悦 SimpRead 转码， 原文地址 www.imooc.com

《手册》的第 7 页和 25 页有两段关于空指针的描述：

> 【强制】Object 的 equals 方法容易抛空指针异常，应使用常量或确定有值的对象来调用 equals。
>
> 【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景:
>
> 1. 返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE。
>
> 反例: public int f () { return Integer 对象}， 如果为 null，自动解箱抛 NPE。
>
> 2. 数据库的查询结果可能为 null。
> 3. 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。
> 4. 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。
> 5. 对于 Session 中获取的数据，建议进行 NPE 检查，避免空指针。
> 6. 级联调用 obj.getA ().getB ().getC (); 一连串调用，易产生 NPE。

《手册》对空指针常见的原因和基本的避免空指针异常的方式给了介绍，非常有参考价值。

那么我们思考以下几个问题：

- 如何学习 `NullPointerException`（简称为 NPE）？
- 哪些用法可能造 NPE 相关的 BUG?
- 在业务开发中作为接口提供者和使用者如何更有效地避免空指针呢?

2.1 源码注释

前面介绍过源码是学习的一个重要途径，我们一起看看 `NullPointerException` 的源码：

```
public
class NullPointerException extends RuntimeException {
    private static final long serialVersionUID = 5162710183389028792L;


    public NullPointerException() {
        super();
    }


    public NullPointerException(String s) {
        super(s);
    }
}
```

源码注释给出了非常详尽地解释：

> 空指针发生的原因是应用需要一个对象时却传入了 `null`，包含以下几种情况：
>
> 1. 调用 null 对象的实例方法。
> 2. 访问或者修改 null 对象的属性。
> 3. 获取值为 null 的数组的长度。
> 4. 访问或者修改值为 null 的二维数组的列时。

> 5. 把 null 当做 Throwable 对象抛出时。

实际编写代码时，产生空指针的原因都是这些情况或者这些情况的变种。

《手册》中的另外一处描述

> “集合里的元素即使 isNotEmpty，取出的数据元素也可能为 `null`。”

和第 4 条非常类似。

如《手册》中的：

> “级联调用 obj.getA ().getB ().getC (); 一连串调用，易产生 NPE。”

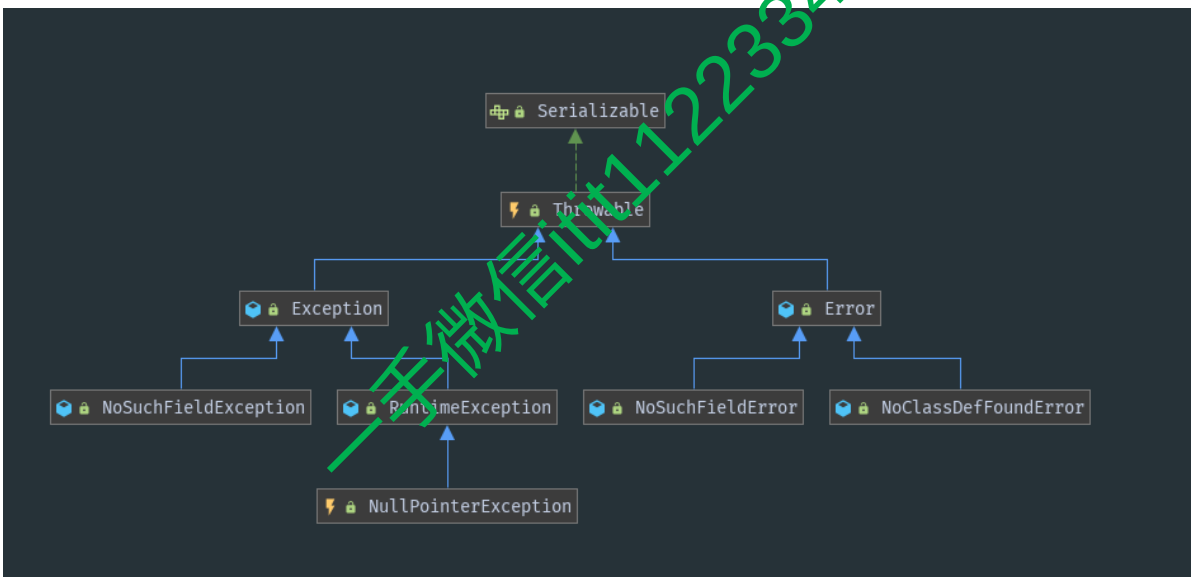
和第 1 条很类似，因为每一层都可能得到 `null` 。

当遇到《手册》中和源码注释中所描述的这些场景时，要注意预防空指针。

另外通过读源码注释我们还得到了“意外发现”，JVM 也可能会通过 `Throwable#Throwable(String, Throwable, boolean, boolean)` 构造函数来构造 `NullPointerException` 对象。

2.2 继承体系

通过源码可以看到 NPE 继承自 `RuntimeException` 我们可以通过 IDEA 的“Java Class Diagram” 来查看类的继承体系。



可以清晰地看到 NPE 继承自 `RuntimeException` ，另外我们选取 `NoSuchFieldException` 和 `NoSuchFieldError` 和 `NoClassDefFoundError` ，可以看到 `Throwable` 的子类型包括 `Error` 和 `Exception`, 其中 NPE 又是 `Exception` 的子类。

那么为什么 `Exception` 和 `Error` 有什么区别？ `Excption` 又分为哪些类型呢?

我们可以分别去 `java.lang.Exception` 和 `java.lang.Error` 的源码注释中寻找答案。

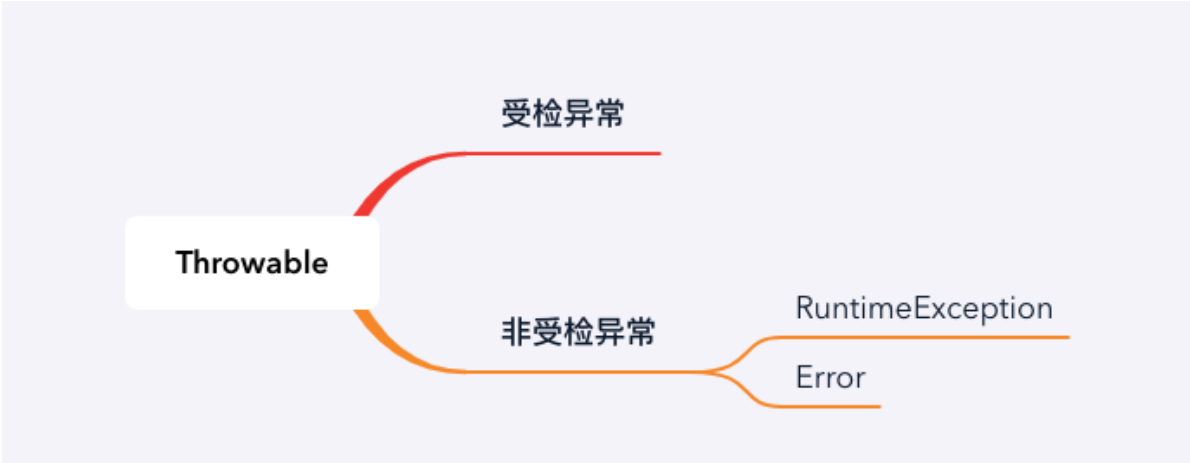
通过 `Exception` 的源码注释我们了解到， `Exception` 分为两类一种是非受检异常（uncheked exceptions）即 `java.lang.RuntimeException` 以及其子类；而受检异常（checked exceptions）的抛出需要再普通函数或构造方法上通过 `throws` 声明。

通过 `java.lang.Error` 的源码注释我们了解到， `Error` 代表严重的问题，不应该被程序 `try-catch` 。编译时异常检测时， `Error` 也被视为不可检异常（uncheked exceptions）。

大家可以在 IDEA 中分别查看 `Exception` 和 `Error` 的子类，了解自己开发中常遇到的异常都属于哪个分类。

其中在异常的类型这里，讲到：

> 不可检异常（**unchecked exception**）包括运行时异常和 error 类。
>
> 可检异常（**checked exception** ）不属于不可检异常的所有异常都是可检异常。除 RuntimeException 和其子类，以及 Error 类以及其子类外的其他 Throwable 的子类。



还有更多关于异常的详细描述，，包括异常的原因、异步异常、异常的编译时检查等，大家可以自己进一步学习。

3.1 最常见的错误姿势

```
@Test
    public void test() {
        Assertions.assertThrows(NullPointerException.class, () -> {
            List<UserDTO> users = new ArrayList<>();
            users.add(new UserDTO(1L, 3));
            users.add(new UserDTO(2L, null));
            users.add(new UserDTO(3L, 3));
            send(users);
        });

    }


    private void send(List<UserDTO> users) {
        for (UserDTO userDto : users) {
            doSend(userDto);
        }
    }

    private static final Integer SOME_TYPE = 2;

    private void doSend(UserDTO userDTO) {
        String target = "default";

        if (!userDTO.getType().equals(SOME_TYPE)) {
            target = getTarget(userDTO.getType());
        }
        System.out.println(String.format("userNo:%s, 发送到%s成功", userDTO,
target));
    }

    private String getTarget(Integer type) {
        return type + "号基地";
    }
```

在第 1 处，如果集合为 `null` 则会抛空指针；

在第 2 处，如果 `type` 属性为 `null` 则会抛空指针异常，导致后续都发送失败。

大家看这个例子觉得很简单，看到输入的参数有 `null` 本能地就会考虑空指针问题，但是自己写代码时你并不知道上游是否会有 `null`。

3. 2 无结果仍返回对象

实际开发中有些同学会有一些非常 “个性” 的写法。

为了避免空指针或避免检查到 null 参数抛异常，直接返回一个空参构造函数创建的对象。

类似下面的做法：

```
public Order getByOrderNo(String orderNo) {

    if (StringUtils.isEmpty(orderNo)) {
        return new Order();
    }

    return doGetByOrderNo(orderNo);
}
```

由于常见的单个数据的查询接口，参数检查不符也会抛异常或者返回 `null`。 极少有上述的写法，因此调用方的惯例是判断结果不为 `null` 就使用其中的属性。

这个哥们这么写之后，上层判断返回值不为 `null`，上层就放心大胆得调用实例函数，导致线上报空指针，就造成了线上 BUG。

3.3 新增 @NonNull 属性反序列化的 BUG

假如有一个订单更新的 RPC 接口，该接口有一个 `OrderUpdateParam` 参数，之前有两个属性一个是 `id` 一个是 `name` 。在某个需求时，新增了一个 extra 属性，且该字段一定不能为 `null` 。

采用 lombok 的 `@NonNull` 注解来避免空指针：

```
import lombok.Data;
import lombok.NonNull;

import java.io.Serializable;

@Data
public class OrderUpdateParam implements Serializable {
    private static final long serialVersionUID = 3240762365557530541L;

    private Long id;

    private String name;
```

```
    @NonNull
    private String extra;
}
```

上线后导致没有使用最新 jar 包的服务对该接口的 RPC 调用报错。

我们来分析一下原因，在 IDEA 的 target - classes 目录下找到 DEMO 编译后的 class 文件，IDEA 会自动帮我们反编译：

```
public class OrderUpdateParam implements Serializable {
    private static final long serialVersionUID = 3240762365557530541L;
    private Long id;
    private String name;
    @NonNull
    private String extra;

    public OrderUpdateParam(@NonNull final String extra) {
        if (extra == null) {
            throw new NullPointerException("extra is marked non-null but is
null");
        } else {
            this.extra = extra;
        }
    }

    @NonNull
    public String getExtra() {
        return this.extra;
    }
    public void setExtra(@NonNull final String extra) {
        if (extra == null) {
            throw new NullPointerException("extra is marked non-null but is
null");
        } else {
            this.extra = extra;
        }
    }


}
```


我们还可以使用反编译工具：JD-GUI 对编译后的 class 文件进行反编译，查看源码。

由于调用方调用的是不含 `extra` 属性的 jar 包，并且序列化编号是一致的，反序列化时会抛出 NPE。

```
Caused by: java.lang.NullPointerException: extra

•       at com.xxx.OrderUpdateParam.<init>(OrderUpdateParam.java:21)
```

RPC 参数新增 lombok 的 `@NonNull` 注解时，要考虑调用方是否及时更新 jar 包，避免出现空指针。

3.4 自动拆箱导致空指针

前面章节讲到了对象转换，如果我们下面的 `GoodCreateDTO` 是我们自己服务的对象， 而 `GoodCreateParam` 是我们调用服务的参数对象。

```
@Data
public class GoodCreateDTO {
    private String title;

    private Long price;

    private Long count;
}

@Data
public class GoodCreateParam implements Serializable {

    private static final long serialVersionUID = -5602221246284162 74L;
    private String title;

    private long price;

    private long count;
}
```


其中 `GoodCreateDTO` 的 `count` 属性在我们系统中是非必传参数，本系统可能为 `null`。

如果我们没有拉取源码的习惯，直接通过前面的转换工具类去转换。

我们潜意识会认为外部接口的对象类型也都是包装类型，这时候很容易因为转换出现 NPE 而导致线上 BUG。

```
public class GoodCreateConverter {

    public static GoodCreateParam convertToParam(GoodCreateDTO goodCreateDTO) {
        if (goodCreateDTO == null) {
            return null;
        }
        GoodCreateParam goodCreateParam = new GoodCreateParam();
        goodCreateParam.setTitle(goodCreateDTO.getTitle());
        goodCreateParam.setPrice(goodCreateDTO.getPrice());
        goodCreateParam.setCount(goodCreateDTO.getCount());
        return goodCreateParam;
    }
}
```

当转换器执行到 `goodCreateParam.setCount(goodCreateDTO.getCount());` 会自动拆箱会报空指针。

当 `GoodCreateDTO` 的 `count` 属性为 `null` 时，自动拆箱将报空指针。

再看一个花样踩坑的例子：

我们作为使用方调用如下的二方服务接口：

```
public Boolean someRemoteCall();
```

然后自以为对方肯定会返回 `TRUE` 或 `FALSE`，然后直接拿来作为判断条件或者转为基本类型，如果返回的是 `null`，则会报空指针异常：

```
if (someRemoteCall()) {

 }
```

大家看示例的时候可能认为这种情况很简单，自己开发的时候肯定会注意，但是往往事实并非如此。

希望大家可以掌握常见的可能发生空指针场景，在开发是注意预防。

3.5 分批调用合并结果时空指针

大家再看下面这个经典的例子。

因为某些批量查询的二方接口在数据较大时容易超时，因此可以分为小批次调用。

下面封装一个将 `List` 数据拆分成每 `size` 个一批数据，去调用 `function` RPC 接口，然后将结果合并。

```
  public static <T, V> List<V> partitionCallList(List<T> dataList, int size,
Function<List<T>, List<V>> function) {

        if (CollectionUtils.isEmpty(dataList)) {
            return new ArrayList<>(0);
        }
        Preconditions.checkArgument(size > 0, "size 必须大于0");

        return Lists.partition(dataList, size)
                .stream()
                .map(function)
                .reduce(new ArrayList<>(),
                        (resultList1, resultList2) -> {
                            resultList1.addAll(resultList2);
                            return resultList1;
                        });


    }
```

看着挺对，没啥问题，其实则不然。

设想一下，如果某一个批次请求无数据，不是返回空集合而是 null，会怎样?

很不幸，又一个空指针异常向你飞来 ...

此时**要根据具体业务场景来判断如何处理这里可能产生的空指针异常。**

如果在某个场景中，返回值为 null 是一定不允许的行为，可以在 function 函数中对结果进行检查，如果结果为 null，可抛异常。

如果是允许的，在调用 map 后，可以过滤 null :

```
// 省略前面代码
.map(function)
.filter(Objects::nonNull)
// 省略后续代码
```

`NPE` 造成的线上 BUG 还有很多种形式，如何预防空指针很重要。

下面将介绍几种预防 NPE 的一些常见方法：

4.1 接口提供者角度

4.1.1 返回空集合

如果参数不符合要求直接返回空集合，底层的函数也使用一致的方式：

```
public List<Order> getByOrderName(String name) {
    if (StringUtils.isNotEmpty(name)) {
        return doGetByOrderName(name);
    }
    return Collections.emptyList();
}
```

4.1.2 使用 Optional

`Optional` 是 Java 8 引入的特性，返回一个 `Optional` 则明确告诉使用者结果可能为空：

```
public Optional<Order> getByOrderId(Long orderId) {
    return Optional.ofNullable(doGetByOrderId(orderId));
}
```

如果大家感兴趣可以进入 `Optional` 的源码，结合前面介绍的 `codota` 工具进行深入学习，也可以结合《Java 8 实战》的相关章节进行学习。

4.1.3 使用空对象设计模式

该设计模式为了解决 NPE 产生原因的第 1 条 “调用 `null` 对象的实例方法”。

在编写业务代码时为了避免 `NPE` 经常需要先判空再执行实例方法：

```
public void doSomeOperation(Operation operation) {
    int a = 5;
    int b = 6;
    if (operation != null) {
        operation.execute(a, b);
    }
}
```


《设计模式之禅》（第二版）554 页在拓展篇中还了 “空对象模式”。

可以构造一个 `NullXXX` 类拓展自某个接口， 这样这个接口需要为 `null` 时，直接返回该对象即可：

```
public class NullOperation implements Operation {

    @Override
    public void execute(int a, int b) {

    }
}
```

这样上面的判空操作就不再有必要， 因为我们在需要出现 `null` 的地方都统一返回 `NullOperation`，而且对应的对象方法都是有的：

```
public void doSomeOperation(Operation operation) {
    int a = 5;
    int b = 6;
    operation.execute(a, b);
}
```

4.2 接口使用者角度

讲完了接口的编写者该怎么做，我们讲讲接口的使用者该如何避免 `NPE` 。

4.2.1 null 检查

正如《代码简洁之道》第 7.8 节 “别传 null 值” 中所要表达的意义：

直接在使用前对不能为 `null` 的和不满足业务要求的条件进行检查，是一种最简单最常见的做法。

通过防御性参数检测，可以极大降低出错的概率，提高程序的健壮性：

```
    @Override
    public void updateOrder(OrderUpdateParam orderUpdateParam) {
        checkUpdateParam(orderUpdateParam);
        doUpdate(orderUpdateParam);
    }

    private void checkUpdateParam(OrderUpdateParam orderUpdateParam) {
        if (orderUpdateParam == null) {
            throw new IllegalArgumentException("参数不能为空");
        }
        Long id = orderUpdateParam.getId();
        String name = orderUpdateParam.getName();
        if (id == null) {
            throw new IllegalArgumentException("id不能为空");
        }
        if (name == null) {
            throw new IllegalArgumentException("name不能为空");
        }
    }
```


JDK 和各种开源框架中可以找到很多这种模式，
`java.util.concurrent.ThreadPoolExecutor#execute` 就是采用这种模式。

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();

     }
```

以及
`org.springframework.context.support.AbstractApplicationContext#assertBeanFactoryActive`

```
protected void assertBeanFactoryActive() {
   if (!this.active.get()) {
      if (this.closed.get()) {
         throw new IllegalStateException(getDisplayName() + " has been closed
already");
      }
      else {
         throw new IllegalStateException(getDisplayName() + " has not been
refreshed yet");
      }
   }
}
```

4.2.2 使用 Objects

可以使用 Java 7 引入的 Objects 类，来简化判空抛出空指针的代码。

使用方法如下：

```
private void checkUpdateParam2(OrderUpdateParam orderUpdateParam) {
    Objects.requireNonNull(orderUpdateParam);
    Objects.requireNonNull(orderUpdateParam.getId());
    Objects.requireNonNull(orderUpdateParam.getName());
}
```


原理很简单，我们看下源码；

```
public static <T> T requireNonNull(T obj) {
    if (obj == null)
        throw new NullPointerException();
    return obj;
}
```

4.2.3 使用 commons 包

我们可以使用 commons-lang3 或者 commons-collections4 等常用的工具类辅助我们判空。

4.2.3.1 使用字符串工具类：org.apache.commons.lang3.StringUtils

```
public void doSomething(String param) {
    if (StringUtils.isNotEmpty(param)) {

    }
}
```

4.2.3.2 使用校验工具类：org.apache.commons.lang3.Validate

```
public static void doSomething(Object param) {
    Validate.notNull(param,"param must not null");
}
public static void doSomething2(List<String> parms) {
    Validate.notEmpty(parms);
}
```

该校验工具类支持多种类型的校验，支持自定义提示文本等。

前面已经介绍了读源码是最好的学习方式之一，这里我们看下底层的源码：

```
public static <T extends Collection<?>> T notEmpty(final T collection, final
String message, final Object... values) {
    if (collection == null) {
        throw new NullPointerException(String.format(message, values));
    }
    if (collection.isEmpty()) {
        throw new IllegalArgumentException(String.format(message, values));
    }
    return collection;
}
```


该如果集合对象为 null 则会抛空 `NullPointerException` 如果集合为空则抛出 `IllegalArgumentException`。

通过源码我们还可以了解到更多的校验函数。

4.2.4 使用集合工具类：`org.apache.commons.collections4.CollectionUtils`

```
public void doSomething(List<String> params) {
    if (CollectionUtils.isNotEmpty(params)) {

    }
}
```

4.2.5 使用 guava 包

可以使用 guava 包的 `com.google.common.base.Preconditions` 前置条件检测类。

同样看源码，源码给出了一个范例。原始代码如下：

```
public static double sqrt(double value) {
    if (value < 0) {
        throw new IllegalArgumentException("input is negative: " + value);
    }

}
```

使用 `Preconditions` 后，代码可以简化为：

```
 public static double sqrt(double value) {
   checkArgument(value >= 0, "input is negative: %s", value);

 }
```


Spring 源码里很多地方可以找到类似的用法，下面是其中一个例子：

`org.springframework.context.annotation.AnnotationConfigApplicationContext#register`

```
public void register(Class<?>... annotatedClasses) {
    Assert.notEmpty(annotatedClasses, "At least one annotated class must be
specified");
    this.reader.register(annotatedClasses);
}
```

`org.springframework.util.Assert#notEmpty(java.lang.Object[], java.lang.String)`

```
public static void notEmpty(Object[] array, String message) {
    if (ObjectUtils.isEmpty(array)) {
        throw new IllegalArgumentException(message);
    }
}
```

虽然使用的具体工具类不一样，核心的思想都是一致的。

4.2.6 自动化 API

4.2.6.1 使用 lombok 的 `@Nonnull` 注解

```
 public void doSomething5(@NonNull String param) {

      proccess(param);
 }
```

查看编译后的代码：

```
 public void doSomething5(@NonNull String param) {
      if (param == null) {
          throw new NullPointerException("param is marked non-null but is
null");
      } else {
          this.proccess(param);
      }
  }
```


4.2.6.2 使用 IntelliJ IDEA 提供的 @NotNull 和 @Nullable 注解

maven 依赖如下：

```
<dependency>
    <groupId>org.jetbrains</groupId>
    <artifactId>annotations</artifactId>
    <version>17.0.0</version>
</dependency>
```

@NotNull 在参数上的用法和上面的例子非常相似。

```
public static void doSomething(@NotNull String param) {

    proccess(param);
}
```

我们可以去该注解的源码 `org.jetbrains.annotations.NotNull#exception` 里查看更多细节，大家也可以使用 IDEA 插件或者前面介绍的 JD-GUI 来查看编译后的 class 文件，去了解 @NotNull 注解的作用。

本节主要讲述空指针的含义，空指针常见的中枪姿势，以及如何避免空指针异常。下一节将为你揭秘 当 switch 遇到空指针，又会发生什么奇妙的事情。

}