

目录

第1章 编码

01 开篇词：为什么学习本专栏 [已学完](#)

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确方 [已学完](#)

05 分层领域模型使用解读 [已学完](#)

06 Java属性映射的正确姿势 [已学完](#)

07 过期类、属性、接口的正确处理姿势 [已学完](#)

08 空指针引发的血案 [已学完](#)

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ [最近阅读](#)

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

13 你真得了解可变参数吗？

更新时间：2019-11-15 11:02:53



“

先相信你自己，然后别人才会相信你。

——屠格涅夫

”

1.前言

《手册》第 7 页 有一段关于 Java 变长参数的规约¹：

【强制】相同参数类型，相同业务含义，才可以使用 Java 的可变参数，避免使用 `Object`。说明:可变参数必须放置在参数列表的最后。(提倡同学们尽量不用可变参数编程)
 正例: `public List<User> listUsers(String type, Long... ids) {...}`

那么我们要思考下面几个问题：

- 为什么要有变长参数？
- 可变参数的常见用法是什么？
- 可变参数有哪些诡异的表现？

本节将详细探讨这些问题。

2 变长参数的思考

2.1 初步了解可变参数

我们知道可变参数（`vararg`）方法（又叫 `variable arity method`）语言特性是在 Java 5 出现的。

其核心原理是：创建一个数组，数组大小与可变参数传入的元素个数，最终将数组传递给方法。

目录

第1章 编码

01 开篇词：为什么学习本专栏 [已学完](#)

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确姿势 [已学完](#)

05 分层领域模型使用解读 [已学完](#)

06 Java属性映射的正确姿势 [已学完](#)

07 过期类、属性、接口的正确处理姿势 [已学完](#)

08 空指针引发的血案 [已学完](#)

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ [最近阅读](#)

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

2.2 可变参数的思考

我们学习 Java 一些语言特性时，最好能够思考它为什么会出现？是为了解决什么问题？有哪些优势？没有它会有哪些困难？等。

我们思考这样一个问题：可变参数的目的是什么？

试想一下，如果没有变长参数的语言特性，我们会怎么处理？

- 我们可以通过定义多个相同类型的参数进行重载。但是如果这样做如果参数数量不固定就无法实现。
- 我们还可以通过定义数组的参数进行重载。但是这这就要求调用时要构造数组，又变成了“定长”，而且需要增加构造数组的代码，代码不够简洁。

由此可见，变长参数适应了不定参数个数的情况，避免了手动构造数组，提高语言的简洁性和代码的灵活性。

3.常见变长参数函数

3.1 JDK中变长参数函数举例

包括 JDK 在内的很多库都封装了很多带有变长参数的函数。

`java.lang.String#format(java.lang.String, java.lang.Object...)` 就是 JDK 中非常常见的变长参数函数之一。

其源码如下：

```
/**
 * Returns a formatted string using the specified format string and
 * arguments.
 *
 * <p> The locale always used is the one returned by {@link
 * java.util.Locale#getDefault() Locale.getDefault()}.
 *
 * @param format
 *     A <a href="..../util/Formatter.html#syntax">format string</a>
 *
 * @param args
 *     Arguments referenced by the format specifiers in the format
 *     string. If there are more arguments than format specifiers, the
 *     extra arguments are ignored. The number of arguments is
 *     variable and may be zero. The maximum number of arguments is
 *     limited by the maximum dimension of a Java array as defined by
 *     <cite>The Java&trade; Virtual Machine Specification</cite>.
 *     The behaviour on a
 *     {@code null} argument depends on the <a
 *     href="..../util/Formatter.html#syntax">conversion</a>.
 *
 * @throws java.util.IllegalFormatException
 *     If a format string contains an illegal syntax, a format
 *     specifier that is incompatible with the given arguments,
 *     insufficient arguments given the format string, or other
```

目录

第1章 编码

01 开篇词：为什么学习本专栏 [已学完](#)

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确方式 [已学完](#)

05 分层领域模型使用解读 [已学完](#)

06 Java属性映射的正确姿势 [已学完](#)

07 过期类、属性、接口的正确处理姿势 [已学完](#)

08 空指针引发的血案 [已学完](#)

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ [最近阅读](#)

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

```
*      formatter class specification.
*
* @return A formatted string
*
* @see java.util.Formatter
* @since 1.5
*/
public static String format(String format, Object... args) {
    return new Formatter().format(format, args).toString();
}
```

根据参数名称或源码注释可知：第一个参数是格式定义，第二个参数为变长参数为前面的格式定义占位符对应的参数。

用法如下：

```
@Test
public void format() {
    String pattern = "我喜欢在 %s 上学习 %s";
    String arg0 = "https://www.imooc.com/";
    String arg1 = "编程";
    String format = String.format(pattern, arg0, arg1);

    String expected = "我喜欢在 " + arg0 + " 上学习 " + arg1;
    Assert.assertEquals(expected, format);
}
```

由于第二个参数为变长参数，我们只需要根据前面占位符的个数填充对应个数的参数即可，非常方便。

3.2 第三方库的可变参数函数举例

再如 commons-lang3 的字符串工具类 [org.apache.commons.lang3.StringUtils#isEmpty](#) 函数源码：

```
/**
 * <p>Checks if all of the CharSequences are empty ("" ) or null.</p>
 *
 * <pre>
 * StringUtils.isEmpty(null)      = true
 * StringUtils.isEmpty("", "")    = true
 * StringUtils.isEmpty(new String[] {} ) = true
 * StringUtils.isEmpty(null, "foo") = false
 * StringUtils.isEmpty("", "bar") = false
 * StringUtils.isEmpty("bob", "") = false
 * StringUtils.isEmpty(" bob ", null) = false
 * StringUtils.isEmpty(" ", "bar") = false
 * StringUtils.isEmpty("foo", "bar") = false
 * </pre>
 *
 * @param css the CharSequences to check, may be null or empty
 * @return {@code true} if all of the CharSequences are empty or null
 * @since 3.6
 */
public static boolean isEmpty(final CharSequence... css) {
    if (ArrayUtils.isEmpty(css)) {
        return true;
    }
}
```

目录

第1章 编码

01 开篇词：为什么学习本专栏 [已学完](#)

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确姿势 [已学完](#)

05 分层领域模型使用解读 [已学完](#)

06 Java属性映射的正确姿势 [已学完](#)

07 过期类、属性、接口的正确处理姿势 [已学完](#)

08 空指针引发的血案 [已学完](#)

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ [最近阅读](#)

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

```
        return false;
    }
}
return true;
}
```

该函数的功能是判断传入的参数（个数不固定）是否都是空字符串或 `null`。

用法非常简单：

```
@Test
public void isAllEmpty(){
    boolean result = StringUtils.isEmpty(null, "foo");
    Assert.assertFalse(result);
}
```

有了变长参数支持，我们不需要根据参数的数量构造定长数组或变长的集合，用法上更加简洁。

我们还看到 `org.apache.commons.lang3.StringUtils` 工具类中还封装了

`StringUtils.isEmpty` 单个参数的判空函数。

通过函数命名和参数列表可以很容易地区分哪个是针对单参数，哪个是针对多参数（变长参数）。

这里也隐含了一个潜规则：虽然变长参数支持 0 到多个参数，但是更多时候是用在 2 个参数及其以上的场景。

大家编写带变长参数函数时可以借鉴这种写法，即为单个参数和不定数量参数编写两个不同的函数。

如果大家平时使用三方工具包时能够留心看其源码，还会发现很多类似的变长参数函数。

4.可变参数诡异问题分析

通过上面的两个例子，我们了解了变长参数函数的优势。

接下来我们通过下面一个示例并结合 `commons-lang` 包的布尔工具类：`org.apache.commons.lang3.BooleanUtils` 来学习和分析可变参数导致的一个诡异问题。

示例代码：

```
public class BooleanDemo {

    public static void main(String[] args) {
        boolean result = and(true, true, true);
        System.out.println(result);
        justPrint(true);
    }

    // 函数1
    private static void justPrint(boolean b) {
        System.out.println(b);
    }
}
```

目录

第1章 编码

01 开篇词：为什么学习本专栏 [已学完](#)

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确方 [已学完](#)

05 分层领域模型使用解读 [已学完](#)

06 Java属性映射的正确姿势 [已学完](#)

07 过期类、属性、接口的正确处理姿势 [已学完](#)

08 空指针引发的血案 [已学完](#)

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗? [最近阅读](#)

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题?

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

```
System.out.println(b);
}

// 函数3
private static boolean and(boolean... booleans) {
    System.out.println("boolean");
    for (boolean b : booleans) {
        if (!b) {
            return false;
        }
    }
    return true;
}

// 函数4
private static boolean and(Boolean... booleans) {
    System.out.println("Boolean");
    for (Boolean b : booleans) {
        if (!b) {
            return false;
        }
    }
    return true;
}
}
```

请问上面程序的结果是什么呢？

相信很多人会回答 `true` 、 `true` 。

回答的依据应该是：示例中 `main` 函数调用的可变参数都是基本类型，因此和函数 3 最贴合，应该会选择函数 3 来执行。

实际是这样的吗？

将代码输入到 IDEA，就会发现 IDEA 就会给出下面这段提示：

Ambiguous method call. Both `and (boolean...)` in `BooleanDemo` and `and (Boolean...)` in `BooleanDemo` match.

模糊的函数调用。该函数调用和 `and (boolean...)` 和 `and (Boolean...)` 两个函数签名都匹配。

4.1 为啥会提示 ambiguous method call ?

很多人看到这里可能会毫无头绪，我们该怎么学习和分析这个问题呢？

按照我们的传统，我们从 JLS2中寻找答案。我们发现其中 [15.12.2 节 Compile - Time Step 2 : Determine Method Signature](#) 中提到：

为了兼容Java SE 5.0 之前的版本，方法签名的选择分为 3 个阶段。

目录

第1章 编码

01 开篇词：为什么学习本专栏 [已学完](#)

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确方式 [已学完](#)

05 分层领域模型使用解读 [已学完](#)

06 Java属性映射的正确姿势 [已学完](#)

07 过期类、属性、接口的正确处理姿势 [已学完](#)

08 空指针引发的血案 [已学完](#)

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ [最近阅读](#)

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

由于不允许自动拆箱、拆箱和可变参数，这一条保证了Java SE 5.0 之前的函数调用的合法性。

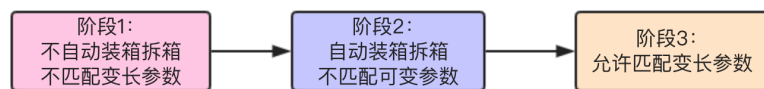
如果在第一阶段可变参数生效，如果在一个已经声明了 `m(Object)` 函数的类中声明 `m(Obejct...)` 函数，会导致即使有更适合的表达式（如 `m(null)`）也不会选择 `m(Object)`。

第二阶段：允许自动装箱和拆箱，但是仍然排除变长参数的重载。如果仍然无法选择合适的方法，则进入第三阶段。

这是为了保证，如果定义了定长参数的函数情况下，不会选择变长参数。

第三阶段：允许自动装箱、拆箱和变长参数的重载。

由此可见，在选择函数签名时，有以下几个阶段：



我们再回头看下示例代码。

第一阶段，选择了函数1。

第二阶段，允许自动装箱和拆箱，但是仍然不匹配可变参数的函数，仍然无法确认使用哪个 `and` 函数，因为自动装箱仍然没有找到 3 个 `boolean` 参数的 `and` 函数。

第三阶段，允许自动装箱和拆箱，允许匹配变长参数。

问题就出现在第三个阶段，允许匹配变长参数时就要允许自动拆箱和装箱，这样函数 3 和函数 4 都可匹配到，因此无法通过编译。

4.2 变长参数的本质是什么？

4.2.1 反编译

我们对项目进行编译，来到 IDEA 的 `target` 目录，查看编译后的 `class` 文件。

也可以直接用 `javac BooleanDemo.java` 对该类进行编译，然后通过前面介绍的 JD-GUI 反编译工具查看。

下面是反编译后的代码：

```

// 函数3
private static boolean and(boolean... booleans) {
    System.out.println("boolean");
    boolean[] var1 = booleans;
    int var2 = booleans.length;
  }

```

目录

第1章 编码

01 开篇词：为什么学习本专栏 已学完

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确方 已学完

05 分层领域模型使用解读 已学完

06 Java属性映射的正确姿势 已学完

07 过期类、属性、接口的正确处理姿 已学完

08 空指针引发的血案 已学完

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ 最近阅读

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

```
        if (!b) {
            return false;
        }
    }

    return true;
}

// 函数4
private static boolean and(Boolean... booleans) {
    System.out.println("Boolean");
    Boolean[] var1 = booleans;
    int var2 = booleans.length;

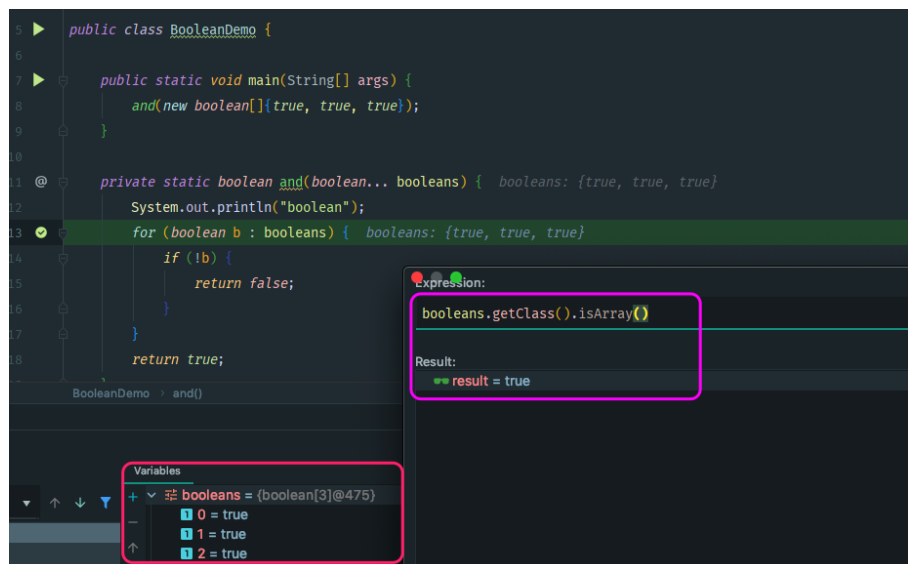
    for(int var3 = 0; var3 < var2; ++var3) {
        Boolean b = var1[var3];
        if (!b) {
            return false;
        }
    }

    return true;
}
```

我们可以清楚地看到，变长参数编译后内部通过数组来处理。

4.2.2 调试

我们还可以在函数 3 中打断点，来观察 `booleans` 这个参数对象的各种属性。



通过“variables”可预览到参数的类型和数据，可以看到 `booleans` 为 `boolean` 类型的数组，长度为 3。

我们还可以通过在“variables”选项卡的 `booleans` 上右键，选择“Evaluate Expression”，然后通过调用 `booleans.getClass().isArray()` 来验证其是否为数组，查看其长度等。

未来有类似的场景，大家都可以通过断点调试来观察数据，还可以通过表达式来研究对象的一些属性。

目录

第1章 编码

01 开篇词：为什么学习本专栏 [已学完](#)

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确方式 [已学完](#)

05 分层领域模型使用解读 [已学完](#)

06 Java属性映射的正确姿势 [已学完](#)

07 过期类、属性、接口的正确处理姿势 [已学完](#)

08 空指针引发的血案 [已学完](#)

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ [最近阅读](#)

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

4.3 如何群伏？

我们如果使用 commons-lang3 的 `org.apache.commons.lang3.BooleanUtils` 工具类中 `and` 函数，也会遇到类似的错误。

下面源码取自 commons-lang3 的 3.9版本。

```
<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3 -->
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.9</version>
</dependency>
```

该类中有两个重载的变长参数函数：

`org.apache.commons.lang3.BooleanUtils#and(boolean...)`

```
/**
 * <p>Performs an and on a set of booleans.</p>
 *
 * <pre>
 * BooleanUtils.and(true, true)      = true
 * BooleanUtils.and(false, false)   = false
 * BooleanUtils.and(true, false)    = false
 * BooleanUtils.and(true, true, false) = false
 * BooleanUtils.and(true, true, true) = true
 * </pre>
 *
 * @param array an array of {@code boolean}s
 * @return {@code true} if the and is successful.
 * @throws IllegalArgumentException if {@code array} is {@code null}
 * @throws IllegalArgumentException if {@code array} is empty.
 * @since 3.0.1
 */
public static boolean and(final boolean... array) {
    // Validates input
    if (array == null) {
        throw new IllegalArgumentException("The Array must not be null");
    }
    if (array.length == 0) {
        throw new IllegalArgumentException("Array is empty");
    }
    for (final boolean element : array) {
        if (!element) {
            return false;
        }
    }
    return true;
}
```

`org.apache.commons.lang3.BooleanUtils#and(java.lang.Boolean...)` 的源码和注释如下：

```
/**
 * <p>Performs an and on an array of Booleans.</p>
 *
```


目录

第1章 编码

01 开篇词：为什么学习本专栏 已学完

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确方式 已学完

05 分层领域模型使用解读 已学完

06 Java属性映射的正确姿势 已学完

07 过期类、属性、接口的正确处理姿势 已学完

08 空指针引发的血案 已学完

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ 最近阅读

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

```
* BooleanUtils.and(Boolean.TRUE, Boolean.FALSE) = Boolean.FALSE
* BooleanUtils.and(Boolean.TRUE, Boolean.TRUE, Boolean.TRUE) = Boolean.TRUE
* BooleanUtils.and(Boolean.FALSE, Boolean.FALSE, Boolean.TRUE) = Boolean.FALSE
* BooleanUtils.and(Boolean.TRUE, Boolean.FALSE, Boolean.TRUE) = Boolean.FALSE
* </pre>
*
* @param array an array of {@code Boolean}s
* @return {@code true} if the and is successful.
* @throws IllegalArgumentException if {@code array} is {@code null}
* @throws IllegalArgumentException if {@code array} is empty.
* @throws IllegalArgumentException if {@code array} contains a {@code null}
* @since 3.0.1
*/
public static Boolean and(final Boolean... array) {
    if (array == null) {
        throw new IllegalArgumentException("The Array must not be null");
    }
    if (array.length == 0) {
        throw new IllegalArgumentException("Array is empty");
    }
    try {
        final boolean[] primitive = ArrayUtils.toPrimitive(array);
        return and(primitive) ? Boolean.TRUE : Boolean.FALSE;
    } catch (final NullPointerException ex) {
        throw new IllegalArgumentException("The array must not contain any null elements")
    }
}
```

错误的原因和前面的示例所分析的一致，都是在选择函数签名时，在前两个阶段没找到匹配的函数，允许变长参数匹配时，允许自动装箱和拆箱，却找到了两个可以匹配的函数。

我们如果直接参考两个工具函数注释上的例子，会发现编译无法通过。从这一点来看，如果注释中的用法和实际使用无法对应，会对使用者造成极大地困扰。

那么到底如何解决这个问题呢？

正如前面讲到的，我们可以看源码的单元测试，也可以通过 [codota](#) 来学习其他优秀的开源项目关于此函数的用法。

接下来我们实践一下。

4.3.1 查看源码的单元测试

我们拉取 [commons-lang](#) 源码，找到了 [BooleanUtilsTest](#) 关于 [and](#) 函数相关的单元测试代码。

[org.apache.commons.lang3.BooleanUtilsTest#testAnd_primitive_validInput_2items](#)

```
@Test
public void testAnd_primitive_validInput_2items() {
    assertTrue(
        BooleanUtils.and(new boolean[] { true, true }),
        "False result for (true, true)");

    assertTrue(
        ! BooleanUtils.and(new boolean[] { false, false }),
```

目录

第1章 编码

01 开篇词：为什么学习本专栏 已学完

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确姿势 已学完05 分层领域模型使用解读 已学完06 Java属性映射的正确姿势 已学完07 过期类、属性、接口的正确处理姿势 已学完08 空指针引发的血案 已学完

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗？ 最近阅读

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题？

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

```
! BooleanUtils.and(new boolean[] { true, false }),
"True result for (true, false)");

assertTrue(
    ! BooleanUtils.and(new boolean[] { false, true }),
    "True result for (false, true)");
}
// 省略其他
```

通过单元测试的代码，我们发现相关的测试代码的参数都是通过数组传入。

`org.apache.commons.lang3.BooleanUtils#and(java.lang.Boolean...)` 相关的单测亦然。

因此我们可以放弃“变长参数”的好处，“回归自然”，我们可以仿照类似写法，使用数组传参。

4.3.2 codota大法

我们在 [codota](#) 上找到该函数的相关范例，可以很好地解决本节所提到的问题。

第一个范例是自定义工具类来包装 `org.apache.commons.lang3.BooleanUtils#and(boolean...)` 函数：

```
BooleanUtil.and(...)
origin: springside/springside

1 /**
2  * 多个值的and
3  */
4 public static boolean and(final boolean... array) {
5     return BooleanUtils.and(array);
6 }
7
```

因为此工具类只包装了其中基本类型变长函数，如果传入基本类型的变长参数可以匹配，如果传入包装类型可以在第二阶段拆箱匹配到该工具函数。

第二个示例也是自定义工具类，但是参数是集合，实际使用时将集合转成数组再调用 `org.apache.commons.lang3.BooleanUtils#and(java.lang.Boolean...)`。

```
HtmlConverters.and(...)
origin: com.shazam.fork/fo...

1 private static Boolean and(final Collection<Boolean> booleans) {
2     return BooleanUtils.and(booleans.toArray(new Boolean[booleans.size()]));
3 }
4
```

通过该示例我们发现作者是用集合来替代不定长参数解决此问题的。

注：通过 [codota](#) 我们还可以看到该工具类的其他函数的一些常见用法。

以上两种方法都是通过自定义工具类的包装，巧妙地避免了直接调用该工具类导致函数签名选择的冲突问题。

5.总结

目录

第1章 编码

01 开篇词：为什么学习本专栏 [已学完](#)

02 Integer缓存问题分析

03 Java序列化引发的血案

04 学习浅拷贝和深拷贝的正确方式 [已学完](#)

05 分层领域模型使用解读 [已学完](#)

06 Java属性映射的正确姿势 [已学完](#)

07 过期类、属性、接口的正确处理姿势 [已学完](#)

08 空指针引发的血案 [已学完](#)

09 当switch遇到空指针

10 枚举类的正确学习方式

11 ArrayList的subList和Arrays的asList学习

12 添加注释的正确姿势

13 你真得了解可变参数吗? [最近阅读](#)

14 集合去重的正确姿势

15 学习线程池的正确姿势

16 虚拟机退出时机问题研究

17 如何解决条件语句的多层嵌套问题?

加餐1：工欲善其事必先利其器

第2章 异常日志

18 一些异常处理建议

19 日志学习和使用的正确姿势

希望大家遇到类似问题时，能够通过本文提供的方法来快速分析原因，并找到应对的办法。

下一节我们将讲述集合去重的正确姿势，会对不同去重方式的性能差异的原因进行分析，并对其性能进行对比。

课后练习

- 结合之前空指针章节所讲的内容，思考示例程序有啥隐患？该如何避免呢？
- 结合本节学的内容，请封装一个工具类，包装 `org.apache.commons.lang3.BooleanUtils#or(java.lang.Boolean...)` 函数，避免选择函数签名时的冲突问题。

参考资料

1. 阿里巴巴与 Java 社区开发者.《Java 开发手册 1.5.0》华山版. 2019. 7 [↩](#)
2. Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley.《Java Language Specification: Java SE 8 Edition》. 2015 [↩](#)

[← 12 添加注释的正确姿势](#)

[14 集合去重的正确姿势 →](#)

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示



目前暂无任何讨论

千学不如一看，千看不如一练