

## 18 | 当反射、注解和泛型遇到OOP时，会有哪些坑？

2020-04-23 朱晔

Java业务开发常见错误100例

[进入课程 >](#)



讲述：王少泽

时长 16:56 大小 15.52M



你好，我是朱晔。今天，我们聊聊 Java 高级特性的话题，看看反射、注解和泛型遇到重载和继承时可能会产生的坑。

你可能说，业务项目中几乎都是增删改查，用到反射、注解和泛型这些高级特性的机会少之又少，没啥好学的。但我要说的是，只有学好、用好这些高级特性，才能开发出更简洁易读的代码，而且几乎所有的框架都使用了这三大高级特性。比如，要减少重复代码，就得用到反射和注解（详见第 21 讲）。



如果你从来没用过反射、注解和泛型，可以先通过官网有一个大概了解：

[Java Reflection API](#) & [Reflection Tutorials](#);

[🔗 Annotations](#) & [🔗 Lesson: Annotations](#);

[🔗 Generics](#) & [🔗 Lesson: Generics](#)。

接下来，我们就通过几个案例，看看这三大特性结合 OOP 使用时会有哪些坑吧。

## 反射调用方法不是以传参决定重载

反射的功能包括，在运行时动态获取类和类成员定义，以及动态读取属性调用方法。也就是说，针对类动态调用方法，不管类中字段和方法怎么变动，我们都可以用相同的规则来读取信息和执行方法。因此，几乎所有的 ORM（对象关系映射）、对象映射、MVC 框架都使用了反射。

反射的起点是 Class 类，Class 类提供了各种方法帮我们查询它的信息。你可以通过这个 [🔗 文档](#)，了解每一个方法的作用。

接下来，我们先看一个反射调用方法遇到重载的坑：有两个叫 age 的方法，入参分别是基本类型 int 和包装类型 Integer。

 复制代码

```
1 @Slf4j
2 public class ReflectionIssueApplication {
3     private void age(int age) {
4         log.info("int age = {}", age);
5     }
6
7     private void age(Integer age) {
8         log.info("Integer age = {}", age);
9     }
10 }
```

如果不通过反射调用，走哪个重载方法很清晰，比如传入 36 走 int 参数的重载方法，传入 Integer.valueOf(“36”) 走 Integer 重载：

 复制代码

```
1 ReflectionIssueApplication application = new ReflectionIssueApplication();
2 application.age(36);
3 application.age(Integer.valueOf("36"));
```

但使用反射时的误区是，认为反射调用方法还是根据入参确定方法重载。比如，使用 `getDeclaredMethod` 来获取 `age` 方法，然后传入 `Integer.valueOf( "36" )`：

 复制代码

```
1 getClass().getDeclaredMethod("age", Integer.TYPE).invoke(this, Integer.valueOf
```

输出的日志证明，走的是 `int` 重载方法：

 复制代码

```
1 14:23:09.801 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
```

其实，要通过反射进行方法调用，第一步就是通过方法签名来确定方法。具体到这个案例，`getDeclaredMethod` 传入的参数类型 `Integer.TYPE` 代表的是 `int`，所以实际执行方法时无论传的是包装类型还是基本类型，都会调用 `int` 入参的 `age` 方法。

把 `Integer.TYPE` 改为 `Integer.class`，执行的参数类型就是包装类型的 `Integer`。这时，无论传入的是 `Integer.valueOf( "36" )` 还是基本类型的 `36`：

 复制代码

```
1 getClass().getDeclaredMethod("age", Integer.class).invoke(this, Integer.valueOf
2 getClass().getDeclaredMethod("age", Integer.class).invoke(this, 36);
```

都会调用 `Integer` 为入参的 `age` 方法：

 复制代码

```
1 14:25:18.028 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
2 14:25:18.029 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
```

现在我们非常清楚了，反射调用方法，是以反射获取方法时传入的方法名称和参数类型来确定调用方法的。接下来，我们再来看一下反射、泛型擦除和继承结合在一起会碰撞出什么坑。

## 泛型经过类型擦除多出桥接方法的坑

泛型是一种风格或范式，一般用于强类型程序设计语言，允许开发者使用类型参数替代明确的类型，实例化时再指明具体的类型。它是代码重用的有效手段，允许把一套代码应用到多种数据类型上，避免针对每一种数据类型实现重复的代码。

Java 编译器对泛型应用了强大的类型检测，如果代码违反了类型安全就会报错，可以在编译时暴露大多数泛型的编码错误。但总有一部分编码错误，比如泛型类型擦除的坑，在运行时才会暴露。接下来，我就和你分享一个案例吧。

有一个项目希望在类字段内容变动时记录日志，于是开发同学就想到定义一个泛型父类，并在父类中定义一个统一的日志记录方法，子类可以通过继承重用这个方法。代码上线后业务没啥问题，但总是出现日志重复记录的问题。开始时，我们怀疑是日志框架的问题，排查到最后才发现是泛型的问题，反复修改多次才解决了这个问题。

父类是这样的：有一个泛型占位符 T；有一个 AtomicInteger 计数器，用来记录 value 字段更新的次数，其中 value 字段是泛型 T 类型的，setValue 方法每次为 value 赋值时对计数器进行 +1 操作。我重写了 toString 方法，输出 value 字段的值和计数器的值：

 复制代码

```
1 class Parent<T> {
2     //用于记录value更新的次数，模拟日志记录的逻辑
3     AtomicInteger updateCount = new AtomicInteger();
4     private T value;
5     //重写toString，输出值和值更新次数
6     @Override
7     public String toString() {
8         return String.format("value: %s updateCount: %d", value, updateCount.get());
9     }
10    //设置值
11    public void setValue(T value) {
12        this.value = value;
13        updateCount.incrementAndGet();
14    }
15 }
```

子类 Child1 的实现是这样的：继承父类，但没有提供父类泛型参数；定义了一个参数为 String 的 setValue 方法，通过 super.setValue 调用父类方法实现日志记录。我们也能明白，开发同学这么设计是希望覆盖父类的 setValue 实现：

```
1 class Child1 extends Parent {
2     public void setValue(String value) {
3         System.out.println("Child1.setValue called");
4         super.setValue(value);
5     }
6 }
```

[复制代码](#)

在实现的时候，子类方法的调用是通过反射进行的。实例化 Child1 类型后，通过 getClass().getMethods 方法获得所有的方法；然后按照方法名过滤出 setValue 方法进行调用，传入字符串 test 作为参数：

```
1 Child1 child1 = new Child1();
2 Arrays.stream(child1.getClass().getMethods())
3     .filter(method -> method.getName().equals("setValue"))
4     .forEach(method -> {
5         try {
6             method.invoke(child1, "test");
7         } catch (Exception e) {
8             e.printStackTrace();
9         }
10    });
11 System.out.println(child1.toString());
```

[复制代码](#)

运行代码后可以看到，虽然 Parent 的 value 字段正确设置了 test，但父类的 setValue 方法调用了两次，计数器也显示 2 而不是 1：

```
1 Child1.setValue called
2 Parent.setValue called
3 Parent.setValue called
4 value: test updateCount: 2
```

[复制代码](#)

显然，两次 Parent 的 setValue 方法调用，是因为 getMethods 方法找到了两个名为 setValue 的方法，分别是父类和子类的 setValue 方法。

这个案例中，子类方法重写父类方法失败的原因，包括两方面：

一是，子类没有指定 String 泛型参数，父类的泛型方法 setValue(T value) 在泛型擦除后是 setValue(Object value)，子类中入参是 String 的 setValue 方法被当作了新方法；

二是，**子类的 setValue 方法没有增加 @Override 注解，因此编译器没能检测到重写失败的问题。这就说明，重写子类方法时，标记 @Override 是一个好习惯。**

但是，开发同学认为问题出在反射 API 使用不当，却没意识到重写失败。他查文档后发现，getMethods 方法能获得当前类和父类的所有 public 方法，而 getDeclaredMethods 只能获得当前类所有的 public、protected、package 和 private 方法。

于是，他就用 getDeclaredMethods 替代了 getMethods：

 复制代码

```
1 Arrays.stream(child1.getClass().getDeclaredMethods())
2     .filter(method -> method.getName().equals("setValue"))
3     .forEach(method -> {
4         try {
5             method.invoke(child1, "test");
6         } catch (Exception e) {
7             e.printStackTrace();
8         }
9     });
```

这样虽然能解决重复记录日志的问题，但没有解决子类方法重写父类方法失败的问题，得到如下输出：

 复制代码

```
1 Child1.setValue called
2 Parent.setValue called
3 value: test updateCount: 1
```

其实这治标不治本，其他人使用 Child1 时还是会发现有两个 setValue 方法，非常容易让人困惑。

幸好，架构师在修复上线前发现了这个问题，让开发同学重新实现了 Child2，继承 Parent 的时候提供了 String 作为泛型 T 类型，并使用 @Override 关键字注释了 setValue 方法，实现了真正有效的方法重写：

 复制代码

```
1 class Child2 extends Parent<String> {
2     @Override
3     public void setValue(String value) {
4         System.out.println("Child2.setValue called");
5         super.setValue(value);
6     }
7 }
```

但很可惜，修复代码上线后，还是出现了日志重复记录：

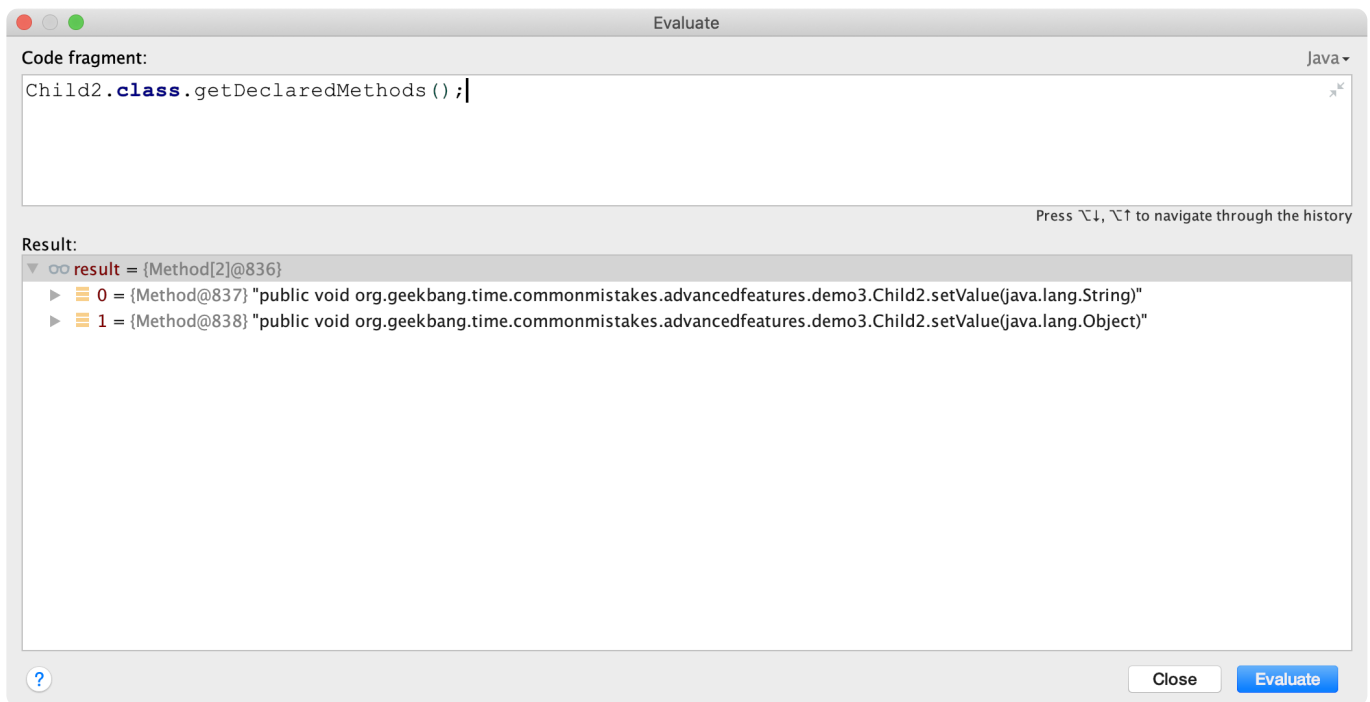
 复制代码

```
1 Child2.setValue called
2 Parent.setValue called
3 Child2.setValue called
4 Parent.setValue called
5 value: test updateCount: 2
```

可以看到，这次是 Child2 类的 setValue 方法被调用了两次。开发同学惊讶地说，肯定是反射出 Bug 了，通过 getDeclaredMethods 查找到的方法一定是来自 Child2 类本身；而且，怎么看 Child2 类中也只有一个 setValue 方法，为什么还会重复呢？

调试一下可以发现，Child2 类其实有 2 个 setValue 方法，入参分别是 String 和 Object。





如果不通过反射来调用方法，我们确实很难发现问题。**其实，这就是泛型类型擦除导致的问题。**我们来分析一下。

我们知道，Java 的泛型类型在编译后擦除为 Object。虽然子类指定了父类泛型 T 类型是 String，但编译后 T 会被擦除成为 Object，所以父类 setValue 方法的入参是 Object，value 也是 Object。如果子类 Child2 的 setValue 方法要覆盖父类的 setValue 方法，那入参也必须是 Object。所以，编译器会为我们生成一个所谓的 bridge 桥接方法，你可以使用 javap 命令来反编译编译后的 Child2 类的 class 字节码：

 复制代码

```
1  javap -c /Users/zhuye/Documents/common-mistakes/target/classes/org/geekbang/ti
2  Compiled from "GenericAndInheritanceApplication.java"
3  class org.geekbang.time.commonmistakes.advancedfeatures.demo3.Child2 extends o
4  org.geekbang.time.commonmistakes.advancedfeatures.demo3.Child2();
5  Code:
6      0: aload_0
7      1: invokespecial #1                  // Method org/geekbang/time/common
8      4: return
9
10
11  public void setValue(java.lang.String);
12  Code:
13      0: getstatic     #2                  // Field java/lang/System.out:Ljavi
14      3: ldc          #3                  // String Child2.setValue called
15      5: invokevirtual #4                  // Method java/io/PrintStream.prin
16      8: aload_0
17      9: aload_1
18     10: invokespecial #5                  // Method org/geekbang/time/common
```



```

19         13: return
20
21
22     public void setValue(java.lang.Object);
23     Code:
24         0: aload_0
25         1: aload_1
26         2: checkcast    #6                // class java/lang/String
27         5: invokevirtual #7                // Method setValue:(Ljava/lang/Str
28         8: return
29 }

```

可以看到，入参为 Object 的 setValue 方法在内部调用了入参为 String 的 setValue 方法（第 27 行），也就是代码里实现的那个方法。如果编译器没有帮我们实现这个桥接方法，那么 Child2 子类重写的是父类经过泛型类型擦除后、入参是 Object 的 setValue 方法。这两个方法的参数，一个是 String 一个是 Object，明显不符合 Java 的语义：

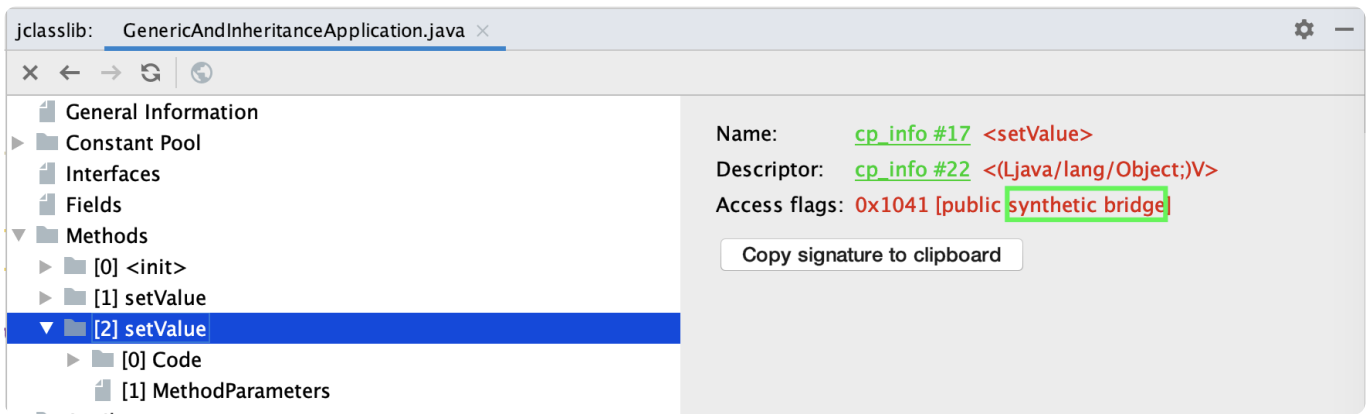
 复制代码

```

1  class Parent {
2
3      AtomicInteger updateCount = new AtomicInteger();
4      private Object value;
5      public void setValue(Object value) {
6          System.out.println("Parent.setValue called");
7          this.value = value;
8          updateCount.incrementAndGet();
9      }
10 }
11
12 class Child2 extends Parent {
13     @Override
14     public void setValue(String value) {
15         System.out.println("Child2.setValue called");
16         super.setValue(value);
17     }
18 }

```

使用 jclasslib 工具打开 Child2 类，同样可以看到入参为 Object 的桥接方法上标记了 public + synthetic + bridge 三个属性。synthetic 代表由编译器生成的不可见代码，bridge 代表这是泛型类型擦除后生成的桥接代码：



知道这个问题之后，修改方式就明朗了，可以使用 method 的 isBridge 方法，来判断方法是不是桥接方法：

通过 getDeclaredMethods 方法获取到所有方法后，必须同时根据方法名 setValue 和非 isBridge 两个条件过滤，才能实现唯一过滤；

使用 Stream 时，如果希望只匹配 0 或 1 项的话，可以考虑配合 ifPresent 来使用 findFirst 方法。

修复代码如下：

 复制代码

```
1 Arrays.stream(child2.getClass().getDeclaredMethods())
2     .filter(method -> method.getName().equals("setValue") && !method.isBridge())
3     .findFirst().ifPresent(method -> {
4         try {
5             method.invoke(child2, "test");
6         } catch (Exception e) {
7             e.printStackTrace();
8         }
9     });
```

这样就可以得到正确输出了：

 复制代码

```
1 Child2.setValue called
2 Parent.setValue called
3 value: test updateCount: 1
```

**最后小结下，使用反射查询类方法清单时，我们要注意两点：**

getMethods 和 getDeclaredMethods 是有区别的，前者可以查询到父类方法，后者只能查询到当前类。

反射进行方法调用要注意过滤桥接方法。


## 注解可以继承吗？

注解可以为 Java 代码提供元数据，各种框架也都会利用注解来暴露功能，比如 Spring 框架中的 @Service、@Controller、@Bean 注解，Spring Boot 的 @SpringBootApplication 注解。

框架可以通过类或方法等元素上标记的注解，来了解它们的功能或特性，并以此来启用或执行相应的功能。通过注解而不是 API 调用来配置框架，属于声明式交互，可以简化框架的配置工作，也可以和框架解耦。

开发同学可能会认为，类继承后，类的注解也可以继承，子类重写父类方法后，父类方法上的注解也能作用于子类，但这些观点其实是错误或者说的不全面的。我们来验证下吧。

首先，定义一个包含 value 属性的 MyAnnotation 注解，可以标记在方法或类上：

 复制代码

```
1 @Target({ElementType.METHOD, ElementType.TYPE})
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface MyAnnotation {
4     String value();
5 }
```

然后，定义一个标记了 @MyAnnotation 注解的父类 Parent，设置 value 为 Class 字符串；同时这个类的 foo 方法也标记了 @MyAnnotation 注解，设置 value 为 Method 字符串。接下来，定义一个子类 Child 继承 Parent 父类，并重写父类的 foo 方法，子类的 foo 方法和类上都没有 @MyAnnotation 注解。

 复制代码

```
1 @MyAnnotation(value = "Class")
2 @Slf4j
```

```

3  static class Parent {
4
5      @MyAnnotation(value = "Method")
6      public void foo() {
7      }
8  }
9
10 @Slf4j
11 static class Child extends Parent {
12     @Override
13     public void foo() {
14     }
15 }

```

再接下来，通过反射分别获取 Parent 和 Child 的类和方法的注解信息，并输出注解的 value 属性的值（如果注解不存在则输出空字符串）：

 复制代码

```

1  private static String getAnnotationValue(MyAnnotation annotation) {
2      if (annotation == null) return "";
3      return annotation.value();
4  }
5
6
7  public static void wrong() throws NoSuchMethodException {
8      //获取父类的类和方法上的注解
9      Parent parent = new Parent();
10     log.info("ParentClass:{}", getAnnotationValue(parent.getClass().getAnnotation(MyAnnotation.class)));
11     log.info("ParentMethod:{}", getAnnotationValue(parent.getClass().getMethod("foo")));
12
13     //获取子类的类和方法上的注解
14     Child child = new Child();
15     log.info("ChildClass:{}", getAnnotationValue(child.getClass().getAnnotation(MyAnnotation.class)));
16     log.info("ChildMethod:{}", getAnnotationValue(child.getClass().getMethod("foo")));
17 }

```

输出如下：

 复制代码


```

1  17:34:25.495 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.demo
2  17:34:25.501 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.demo
3  17:34:25.504 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.demo
4  17:34:25.504 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.demo

```


可以看到，父类的类和方法上的注解都可以正确获得，但是子类的类和方法却不能。这说明，**子类以及子类的方法，无法自动继承父类和父类方法上的注解。**

如果你详细了解过注解应该知道，在注解上标记 `@Inherited` 元注解可以实现注解的继承。那么，把 `@MyAnnotation` 注解标记了 `@Inherited`，就可以一键解决问题了吗？

 复制代码

```
1 @Target({ElementType.METHOD, ElementType.TYPE})
2 @Retention(RetentionPolicy.RUNTIME)
3 @Inherited
4 public @interface MyAnnotation {
5     String value();
6 }
```

重新运行代码输出如下：


 复制代码

```
1 17:44:54.831 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
2 17:44:54.837 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
3 17:44:54.838 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
4 17:44:54.838 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
```

可以看到，子类可以获得父类类上的注解；子类 `foo` 方法虽然是重写父类方法，并且注解本身也支持继承，但还是无法获得方法上的注解。


如果你再仔细阅读一下 [@Inherited 的文档](#) 就会发现，`@Inherited` 只能实现类上的注解继承。要想实现方法上注解的继承，你可以通过反射在继承链上找到方法上的注解。但，这样实现起来很繁琐，而且需要考虑桥接方法。

好在 Spring 提供了 `AnnotatedElementUtils` 类，来方便我们处理注解的继承问题。这个类的 `findMergedAnnotation` 工具方法，可以帮助我们找出父类和接口、父类方法和接口方法上的注解，并可以处理桥接方法，实现一键找到继承链的注解：

 复制代码

```
1 Child child = new Child();
2 log.info("ChildClass:{}", getAnnotationValue(AnnotatedElementUtils.findMergedAi
3 log.info("ChildMethod:{}", getAnnotationValue(AnnotatedElementUtils.findMerged,
```

修改后，可以得到如下输出：

 复制代码

```
1 17:47:30.058 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
2 17:47:30.059 [main] INFO org.geekbang.time.commonmistakes.advancedfeatures.dem
```

可以看到，子类 foo 方法也获得了父类方法上的注解。

## 重点回顾

今天，我和你分享了使用 Java 反射、注解和泛型高级特性配合 OOP 时，可能会遇到的一些坑。

第一，反射调用方法并不是通过调用时的传参确定方法重载，而是在获取方法的时候通过方法名和参数类型来确定的。遇到方法有包装类型和基本类型重载的时候，你需要特别注意这一点。

第二，反射获取类成员，需要注意 getXXX 和 getDeclaredXXX 方法的区别，其中 XXX 包括 Methods、Fields、Constructors、Annotations。这两类方法，针对不同的成员类型 XXX 和对象，在实现上都有一些细节差异，详情请查看 [官方文档](#)。今天提到的 getDeclaredMethods 方法无法获得父类定义的方法，而 getMethods 方法可以，只是差异之一，不能适用于所有的 XXX。

第三，泛型因为类型擦除会导致泛型方法 T 占位符被替换为 Object，子类如果使用具体类型覆盖父类实现，编译器会生成桥接方法。这样既满足子类方法重写父类方法的定义，又满足子类实现的方法有具体的类型。使用反射来获取方法清单时，你需要特别注意这一点。

第四，自定义注解可以通过标记元注解 @Inherited 实现注解的继承，不过这只适用于类。如果要继承定义在接口或方法上的注解，可以使用 Spring 的工具类 AnnotatedElementUtils，并注意各种 getXXX 方法和 findXXX 方法的区别，详情查看 [Spring 的文档](#)。

最后，我要说的是。编译后的代码和原始代码并不完全一致，编译器可能会做一些优化，加上还有诸如 AspectJ 等编译时增强框架，使用反射动态获取类型的元数据可能会和我们编写的源码有差异，这点需要特别注意。你可以在反射中多写断言，遇到非预期的情况直接抛异常，避免通过反射实现的业务逻辑不符合预期。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

## 思考与讨论

1. 泛型类型擦除后会生成一个 bridge 方法，这个方法同时又是 synthetic 方法。除了泛型类型擦除，你知道还有什么情况编译器会生成 synthetic 方法吗？
2. 关于注解继承问题，你觉得 Spring 的常用注解 @Service、@Controller 是否支持继承呢？

你还遇到过与 Java 高级特性相关的其他坑吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你  
攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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。



## 精选留言 (2)

写留言



**终结者999号**

2020-04-23

老师您好，我听我们架构师说生产上最好不要使用反射会对性能有影响，有依据吗？

作者回复：一般情况下这些不会成为性能瓶颈，除非并发特别大，一次处理又涉及几千几万次反射，各种框架内部也大量使用反射，不必这么绝对



3



**Geek\_3b1096**

2020-04-23

谢谢老师期待21讲

展开 ▾



1