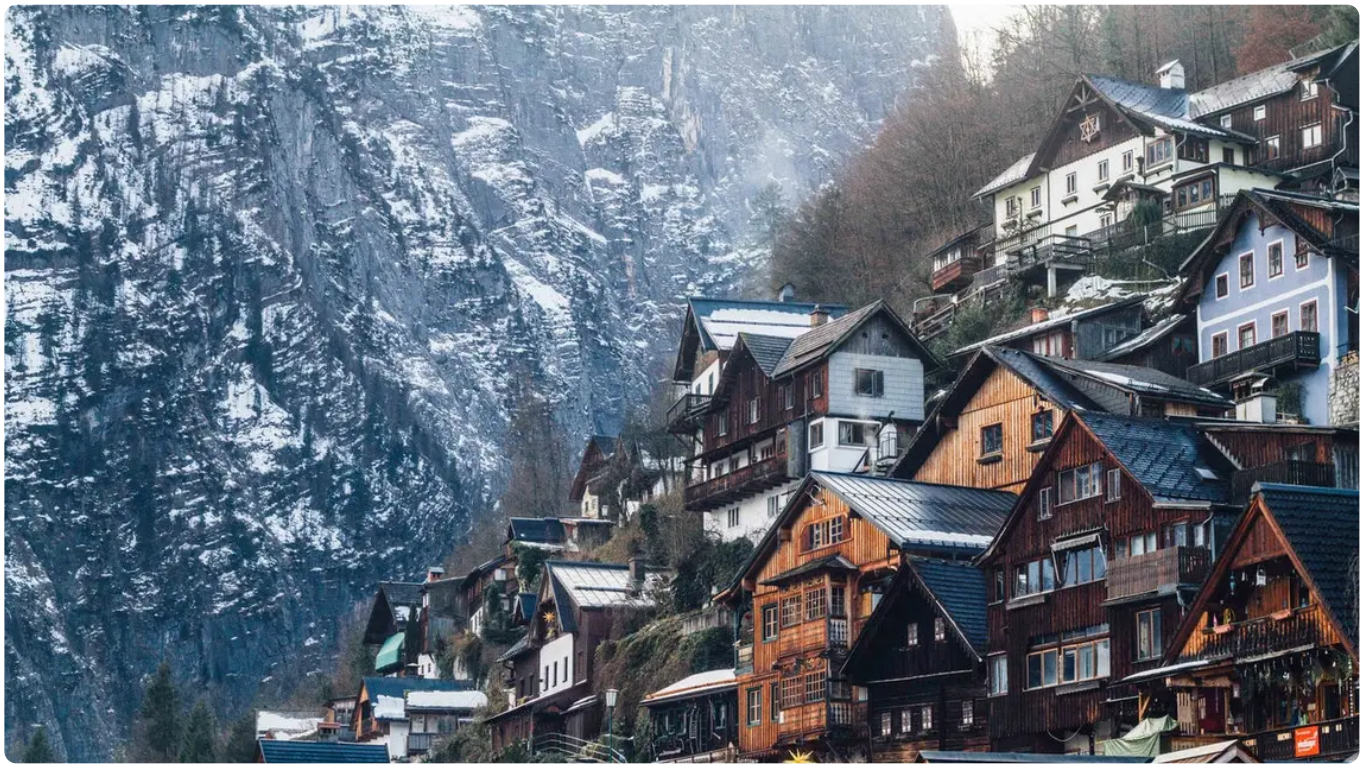
07 | switch匹配：能不能适配不同的类型？

2021-11-29 范学雷

《深入剖析Java新特性》 课程介绍 >



讲述：范学雷

时长 15:01 大小 13.76M

你好，我是范学雷。今天，我们聊一聊 switch 的模式匹配。

switch 的模式匹配这个特性，在 JDK 17 中以预览版的形式发布。按照通常的进度，这个特性可能还需要两到三个版本，才能最终定稿。

这个特性很简单，但是非常重要，可以帮助我们解决不少棘手而且重要的问题。我们不妨在定稿之前，就试着看看它。

前面，我们讨论了类型匹配和 switch 表达式。那 switch 的模式匹配又是什么样子的？为什么说 switch 的模式匹配非常重要？我们还是通过案例和代码，一步一步地了解 switch 的模式匹配吧。

阅读案例

在面向对象的编程语言中，研究表示形状的类，是一个常用的教学案例。今天的阅读案例，会涉及到表示形状的接口和类的定义，以后，我还会给出一个使用案例。通过这个案例，我们可以看到面向对象设计的一个代码在维护和发展时的难题。

假设我们定义了一个表示形状的封闭类，它的名字是 Shape；我们也定义了两个许可类：Circle 和 Square，它们分别表示圆形和正方形。下面的代码，就是一个可供你参考的实现方式。

复制代码

```
public sealed interface Shape
        permits Shape.Circle, Shape.Square {
    record Circle(double radius) implements Shape {
        // blank
    }

    record Square(double side) implements Shape {
        // blank
    }
}
```

接着，我们就要使用形状这个类来处理具体的问题了。你可以先试着回答一下，给定了一个形状的对象，我们该怎么判断这个对象是不是一个正方形呢？

这是一个简单的问题。只要判断这个对象是不是一个正方形类（Square）的实例就可以了。就像下面的代码这样。

复制代码

```
public static boolean isSquare(Shape shape) {
    return (shape instanceof Shape.Square);
}
```

无论是形状类的设计，还是我们处理问题的方式，看起来都没有什么问题。不过，如果我们朝前看，想一想未来的形状类的变化，问题可能就浮现出来了。

假设上面表示形状的封闭类和许可类是版本 1.0，它们被封装在一个基础 API 类库里。而判断一个表示形状的对象是不是正方形的代码，也就是 IsSquare 的实现代码，我们把它封装到另外一个 API 类库里。为了方便后面的讨论，我们把这两个类库称为基础类库和扩展类库（这两个名字并不一定契合实际）。

现在，我们升级表示形状的封闭类和许可类，新加入一个许可类，用来表示长方形。这样，我们就有了下面这样的代码。

复制代码

```
public sealed interface Shape
        permits Shape.Circle, Shape.Rectangle, Shape.Square {
    /**
     * @since 1.0
     */
    record Circle(double radius) implements Shape {
        // blank
    }

    /**
     * @since 1.0
     */
    record Square(double side) implements Shape {
        // blank
    }

    /**
     * @since 2.0
     */
    record Rectangle(double length, double width) implements Shape {
        // blank
    }
}
```

在面向对象的世界里，增加一个新的子类是一种很常见的升级方法。而且，不论是出于理论还是实践，我们都没有充分的理论、也没有应有的能力杜绝掉这样的升级。所以，新加入一个表示长方形的许可类，似乎并没有什么不妥。类似这样的更改，我们也不会期待出现明显的可兼容性问题。

好了，现在我们有了 2.0 版本的基础类库。

然后，我们再来看看扩展类库。我们知道，正方形是一个特殊的长方形。如果一个长方形的长和宽是相等的，那么它也是一个正方形。所以，如果基础类库支持了长方形，我们就需要考虑正方形这个特例。不然的话，这个扩展类库的实现，就不能处理这个特例。

扩展类库的更改也很简单，只要加入处理特例的逻辑就可以了。这样，我们就有了下面这样的升级之后的代码。

复制代码

```
public static boolean isSquare(Shape shape) {
    if (shape instanceof Shape.Rectangle rect) {
        return (rect.length() == rect.width());
    }
    return (shape instanceof Shape.Square);
}
```

然而，意识到扩展类库需要更改，并不是一件容易的事情。甚至，通常情况下，我们可以说它是一件非常艰苦和艰难的事情。

对于需要更改扩展类库这件事，基础类库的作者，不会通知扩展类库的作者。这绝对不是基础类库的作者的懒惰或者不负责任。一般情况下，基础类库和扩展类库是独立的产品，由不同的团队或者社区维护。所以基础类库的作者往往不太可能意识到扩展类库的存在，更不可能去研究扩展类库的实现细节。所以，修改扩展类库这件事，一般来说，是扩展类库维护者的责任。

同样地，扩展类库维护者也不会注意到基础类库的修改，更不容易想到基础类库的修改会影响到扩展类库的行为。通常地，API 的使用者依赖 API 的兼容性。也就是说，API 可以升级，但是这个升级不能影响已有代码的使用。换句话说，1.0 版本的 API 上能跑得通的代码，2.0 版本的 API 上，同样的代码也必须能跑得通。所以，扩展类库维护者，也可以把问题踢给基础类库的维护者。

那么用户呢？有时候，他们找基础类库的维护者抱怨；有时候，他们找扩展类库的维护者抱怨。谁的市场影响大，对用户更友好，谁听到的抱怨就多一点。我们也没有理由责怪用户的抱怨，毕竟是他们的业务系统，也就是现实世界的系统，遇到了真正的问题，遭受了真实的损失。

这样的问题出现的根本原因，就是我们没有在用户抱怨之前发现这样的事实：扩展类库必须做出修改，以适应升级的基础类库。

而解决这样的问题，只依靠基础类库维护者和扩展类库维护者的勤奋，是不可能实现的。

那么，我们该怎么办呢？

其中的一个思路，就是尽可能早地发现这样的兼容性问题。而我给你的其中一条解决办法，就是使用具有类型匹配能力的 switch 表达式。

模式匹配的 switch

具有模式匹配能力的 switch，说的是将模式匹配扩展到 switch 语句和 switch 表达式，允许测试多个模式，而且每一个模式都可以有特定的操作。这样，就可以简洁、安全地表达复杂的面向数据的查询了。

下面的代码，展示了如何使用具有模式匹配能力的 switch，来判断一个对象是不是正方形：

复制代码

```
1 public static boolean isSquare(Shape shape) {
2     return switch (shape) {
3         case null, Shape.Circle c -> false;
4         case Shape.Square s -> true;
5     };
6 }
```

这段简短的代码里面，有几个地方是我们在 JDK 17 之前没有遇到过的。

扩充的匹配类型

第一个地方，就是 switch 要匹配的表达式，或者说数据，而不是我们熟悉的类型。我们可能都知道，JDK 17 之前的 switch 关键字可以匹配的数据类型包括数字、枚举和字符串。本质上，这三种数据类型都是整形的原始类型。而在上面的例子中，这个要匹配的目标数据类型，是一个表示形状的对象，是一个引用类型。

具有模式匹配能力的 switch，提升了 switch 的数据类型匹配能力。switch 要匹配的数据，现在可以是整形的原始类型（数字、枚举、字符串），或者引用类型。

支持 null 情景模式

第二个地方，就是空引用“null”出现在了匹配情景中。以前，switch 要匹配的数据不能是空引用。否则，就会抛出“NullPointerException”这样的运行时异常。所以，规范的、公开接口的代码，通常都要检查匹配数据是不是一个空引用，然后才能接着使用 switch 语句或者 switch 表达式。就像下面的例子这样。

复制代码

```
public static boolean isSquare(Shape shape) {
    if (shape == null) {
      return false;
    }

    return switch (shape) {
        case Shape.Circle c -> false;
        case Shape.Square s -> true;
    };
}
```

然而，对于非公开接口的内部实现代码，是不是需要这样的检查，并不是显而易见的。比如说，如果所有的调用，都不会传入空的引用，当然也就不需要检查空引用。可是，这样的假设过于脆弱。而且，对于代码的阅读者来说，去检查所有可能的内部调用，真的是一件很艰难的事情。

具有模式匹配能力的 switch，支持空引用的匹配。如果我们能够有意识地使用这个特性，可以提高我们的编码效率，降低代码错误。

可类型匹配的情景

第三个地方，就是类型匹配出现在了匹配情景中。也就是说，你既可以检查类型，还可以获得匹配变量。以前，switch 要匹配的数据是一个数值，比如说星期三或者十二月。对类型匹配来说，switch 要匹配的数据是一个引用；这时候，匹配情景要做的主要判断之一，是我们希望知道的这个引用的类型。

比如说吧，如果要匹配的数据是一个表示形状的类的引用，我们希望匹配情景要能够判断出来这个引用是一个圆形类的引用，还是一个正方形类的引用。如果情景能够匹配，我们还希望能够获得匹配变量。这一点，其实就像是我们在第 5 讲说到的类型匹配。现在，类型匹配出现在了 switch 语句和 switch 表达式的使用场景里。

```
1 case Shape.Circle c -> false;
```

复制代码

这样，我们就在 switch 语句和 switch 表达式里获得了类型匹配的好处，如果需要使用转换后的数据类型，我们就不再需要编写强制类型转换的代码了。这就简化了代码逻辑，减少了代码错误，提高了生产效率。

穷举的匹配情景

具有模式匹配能力的 switch，是怎么解决掉阅读案例里讨论的基础类库和扩展类库协同维护问题的呢？到现在，这个问题的答案还不是很明确，虽然答案已经有了。

这就是我们要讨论的第四个地方，使用 switch 表达式，穷举出所有的情景。在 isSquare 这个方法的实现里，我们使用了 switch 表达式，并且穷举出了所有可以匹配的形状类。我们知道，switch 表达式需要穷举出所有的情景。否则，编译器就会报错。使用 switch 表达式这个特点，就是我们解决阅读案例里提到的问题的基本思路。

现在，如果我们使用 2.0 版本的基础类库，也就是新加入了表示长方形的许可类的实现，那么 isSquare 这个方法的实现就不能通过编译了。因为，这个方法的实现遗漏了长方形这个许可类，没有满足 switch 表达式需要穷举所有情景的要求。

如果代码编译期就报错，扩展类库的维护者就能够第一时间知道这个方法的缺陷。这样，他们就不用等到用户遇到真实问题的时候，才意识到要去适应升级的基础类库了。

这种提前暴露问题的方式，大大地降低了代码维护的难度，让我们有更多的精力专注在更有价值的问题上。

意识到代码需要修改，其实是最难的一步。如果已经意识到这个问题，具体的修改就很简单了。如果对实现细节感兴趣，你可以参考下面这段我修改后的代码。

复制代码

```
public static boolean isSquare(Shape shape) {
    return switch (shape) {
        case null, Shape.Circle c -> false;
        case Shape.Square s -> true;
        case Shape.Rectangle r -> r.length() == r.width();
    };
}
```

改进的性能

另外，具有模式匹配能力的 switch（包括 switch 语句和 switch 表达式），还提高了多情景处理性能。

如果使用 if-else 的处理方式，每一个情景，都要至少对应一个 if-else 语句。寻找匹配情景时，需要按照 if-else 的使用顺序来执行，直到遇到条件匹配的情景为止。这样，对于 if-else 语句来说，找到匹配情景的时间复杂度是 O(N)，其中 N 指的是需要处理的情景的数量。换句话说，if-else 语句寻找匹配情景的时间复杂度和需要处理的情景数量成正比。

如果使用 switch 的处理方式，每一个情景，也要至少对应一个 case 语句。但是，寻找匹配情景时，switch 并不需要按照 case 语句的顺序执行。对于 switch 的处理方式，找到匹配的情景的时间复杂度是 O(1)。也就是说，switch 寻找匹配情景的时间复杂度和需要处理的情景数量关系不大。

情景越多，使用 switch 的处理方式获得的性能提升就越大。

什么时候使用 default？

在前面的代码里，我们并没有看到 switch 的缺省选择情景 default 关键字的使用。在 switch 的模式匹配里，我们还可以使用缺省选择情景。比如说，我们可以使用 default 来实现前面讨论的 isSquare 这个方法。

复制代码

```
public static boolean isSquare(Shape shape) {
    return switch (shape) {
        case Shape.Square s -> true;
        case null, default -> false;
    };
}
```

使用了 default，也就意味着这样的 switch 表达式总是能够穷举出所有的情景。遗憾的是，这样的代码丧失了检测匹配情景有没有变更的能力；也丧失了解决阅读案例里提到的问题的能力。

所以，一般来说，**只有我们能够确信，待匹配类型的升级，不会影响 switch 表达式的逻辑的时候，我们才能考虑使用缺省选择情景**。

总结

好，到这里，我来做个小结。从前面的讨论中，我们重点了解了 switch 的模式匹配，以及如何使用 switch 表达式来检测子类扩充出现的兼容性问题。具有模式匹配能力的 switch，提升了 switch 的数据类型匹配能力。switch 要匹配的数据，现在可以是整形的原始类型（数字、枚举、字符串），或者引用类型。

在前面的讨论里，我们把重点放在了 switch 表达式上。实际上，除了情景穷举相关的内容之外，我们的讨论也适用于 switch 语句。

在我们日常的编码实践中，为了尽早暴露子类扩充出现的兼容性问题，降低代码的维护难度，提高多情景处理的性能，我们应该优先考虑使用 switch 的模式匹配，而不是传统的 if-else 语句。

如果你想要丰富你的代码评审清单，有了 switch 的模式匹配以后，你可以加入下面这几条：

> 处理情景选择的 if-else 语句，是不是可以使用 switch 的模式匹配？
> 使用了模式匹配的 switch 表达式，有没有必要使用缺省选择情景 default？
> 使用了模式匹配的 switch 语句和表达式，是不是可以使用 null 选择情景?

另外，我还拎出了几个今天讨论过的技术要点，这些都可能在你们面试中出现哦。通过这一次学习，你应该能够：

知道 switch 能够适配不同的类型，并且能够使用 switch 的模式匹配；

- 面试问题：你知道怎么使用 switch 匹配不同的类型吗？

了解 switch 的模式匹配要解决的问题，以及它的特点；

- 面试问题：使用 switch 的模式匹配有哪些好处？

掌握怎么使用 switch 表达式处理子类扩充带来的兼容性问题。

- 面试问题：子类扩充有可能遇到什么问题，该怎么解决？

子类扩充出现的兼容性问题，是面向对象编程实践中一个棘手、重要、高频的问题。如果你能够有意识地使用 switch 的模式匹配，并且编写的代码能够自动检测到子类扩充出现的变动，就可以降低代码的维护难度和维护成本，提高代码的健壮性。在面试的时候，如果你能够主动地在代码里使用 switch 的模式匹配，而不是传统的 if-else 语句，这会是一个震惊面试官的好机会。

思考题

关于 switch 的模式匹配，还有两个特点我们没有讨论。一个是匹配情景的支配地位，一个是戒备模式的匹配情景。这一次的思考题，主要是一个阅读作业，也是自学这两个特点的一个家庭作业。

希望你可以阅读 switch 的模式匹配的官方文档，然后找出并且改正下面这段代码的错误，尽可能地优化这段代码。

复制代码

```
public static boolean isSquare(Shape shape) {
    if (shape == null) {
        return false;
    }

    return switch (shape) {
        case Shape.Square s -> true;
        case Shape.Rectangle r -> false;
        case Shape.Rectangle r && r.length() == r.width() -> true;
        default ->false;
    };
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及验证的代码和结果。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switch 模式匹配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 pattern/review/xuelei 的目录下面。

注：switch 的模式匹配这个特性，在 JDK 17 还是预览版。你可以现在开始学习这个特性，但是暂时不要把它用在严肃的产品里，直到正式版发布。

上一篇 06 | switch表达式：怎么简化多情景操作？

下一篇 08 | 抛出异常，是不是错误处理的第一选择？

精选留言 (13)

写留言

Jagger Chen 置顶

2021-12-01

1. 匹配情景的支配地位，第一条语句支配了第二条语句
case Shape.Rectangle r -> false;
case Shape.Rectangle r && r.length() == r.width() -> true;
2. 戒备模式的匹配情景
https://docs.oracle.com/en/java/javase/17/language/pattern-matching.html...

展开

作者回复: 我要点赞把这个回答置顶。 小伙伴们，让更多的同学看到这个留言吧！

2

曙光
2021-12-04

哇哇哇！！！Java 是世界上最最最好的编程语言[好的] 不接受任何反驳

作者回复: 哈哈，淡定些

Jxin
2021-12-01

这个 switch表达式 很香啊，感觉可以用来代替 map策略模式 了。虽然相对于 map策略模式 多了每次新增策略时要追加 case ，但基于编译期的检查，其实是能感知到的，所以只要能发包就不会漏（多包一层给调用侧也就隔离了散弹式修改的可能）。但是，原本被map策略模式隐掉的路由或则映射关系就再次显示表达了，我觉得这个显示表达所有路由关系的价值完全可以覆盖每次追加一处(可感知变动)修改的成本。

展开

作者回复: 而且，追加的需求还能感知到，这就太棒了！

共 2 条评论

许灵
2021-12-01

https://github.com/XueleiFan/java-up/pull/11,
现在的代码库有点问题，主要是新特性加了之后。需要开启module, 有一些代码不符合要求，本次作业答案，已经在文章中了。使用类型的匹配主要是为了更早的，自动地发现问题， 而不是等出问题了去定位问题。

展开

作者回复: 谢谢你发现了这个问题："现在的代码库有点问题"。 有一些东西遗漏了，加进来了。
你更新一下再试试看，还有没有问题。PR我稍后再看。

许灵
2021-12-01

https://github.com/XueleiFan/java-up/pull/11

展开

作者回复: 代码很干净。 有一个小小的建议，放在PR的评论里了。

ABC
2021-12-01

思考题：

1.首先可以去掉代码头部的if判断
2.在switch中整合null,default,如果代码作为类库发布，此处可以抛出异常，告知调用者必须明确参数。如果作为业务代码，则可以默认某个类型为返回值。...

展开

作者回复: 两个点都没有问题，但是没有找出来代码里的错误。switch匹配有两个特点，我们没有讨论。这个题，需要进一步阅读switch 的模式匹配的官方文档，才能做的出来。

共 2 条评论 >

ABC
2021-12-01

第五讲的时候简单用了一下，但有一个疑惑，如果有三个子类，单用boolean并不能区分出具体是哪个子类。

判断空这方面，Dart有点激进，所有变量必须是非空的，除非用?问号修饰，表示变量可空。估计很难在Java上看到空安全了。

展开

作者回复: 空安全是一个热门的话题，我们后面也会讨论。

bigben
2021-11-30

想要个语言特性，不知道有没人提过，有点类似别名，比如：我可以定义一个Age类，它是Integer的别名，但取值范围可以限制为0-200

展开

作者回复: 挺有意思的。可以试着提交一个新特性请求。这个主意也可以用来改进档案类，比如Circle(double radius > 0).

aoe
2021-11-30

switch真好用，性能还这么强！对于 switch 的处理方式，找到匹配的情景的时间复杂度是 O(1)

Calvin
2021-11-30

文章中说到：寻找匹配情景时，switch 并不需要按照 case 语句的顺序执行。对于 switch 的处理方式，找到匹配的情景的时间复杂度是 O(1)。也就是说，switch 寻找匹配情景的时间复杂度和需要处理的情景数量关系不大。

请教下老师，这是为什么呢？switch的case项不是从上至下从左至右一个一个匹配直至成...
展开

作者回复: switch不是从上往下一个一个匹配着执行的。相对于if-else来说，switch匹配场景更简单，这样的话，JVM就有办法对场景编码，直接跳转到对应的case，而不需要挨个试。

共 4 条评论 >

哦吼掉了
2021-11-29

咨询一小段代码，为啥这个编译报错呢？是因为没有break导致穿透么
```
static void test(Object obj) {
    switch (obj) {
        case Character c:
            System.out.println("character");...
```
展开

作者回复: 编译器有没有开启预览版支持？—enable-preview

共 4 条评论 >

赤云
2021-11-29

震惊面试官 就怕面试官不学习 ，啊哈哈

展开

作者回复: 这么小看面试官啊。不学习的面试官，不正是我们喜欢的面试官吗？


bigben
2021-11-29

就喜欢这种某地方做了修改可以检测影响其他地方的特性，代码改起来也放心多了。

作者回复: 改代码太难了，这种特性确实招人喜欢。