JEP 440：记录模式

所有者	加文·比尔曼
类型	新特性
范围	SE
地位	已关闭/已交付
发布	21
成分	规格/语言
讨论	openjdk dot org 的 Amber dash 开发人员
涉及到	JEP 432：记录模式（第二次预览）
审阅者	布赖恩·戈茨
受认可	布赖恩·戈茨
已创建	2023/01/18 14:38
更新	2023/08/28 16:51
问题	8300541

概括

增强Java编程语言，引入record patterns (记录模式)以实现对record values的解构。允许嵌套record patterns 和 type patterns，从而实现强大、声明式和可组合的数据导航和处理形式。

历史

记录模式由JEP 405提议作为预览功能并在JDK 19中提供，并由JEP 432进行第二次预览并在 JDK 20中提供。*此功能与模式匹配*`switch` (JEP 441)共同发展，两者之间具有相当大的交互性。该 JEP 建议根据持续的经验和反馈进一步完善该功能。

除了一些小的编辑更改之外，自第二次预览以来的主要更改是删除对增强`for`语句标题中出现的记录模式的支持。此功能可能会在未来的 JEP 中重新提出。

目标

- 扩展模式匹配以解构`record`类的实例，从而实现更复杂的数据查询。
- 添加嵌套模式，支持更多可组合的数据查询。

动机

在 Java 16 中，JEP 394扩展了该`instanceof`运算符以采用 *类型模式*并执行*模式匹配*。这种适度的扩展允许简化熟悉的 instanceof-and-cast 习惯用法，使其更加简洁且不易出错：

```
// Prior to Java 16
if (obj instanceof String) {
    String s = (String)obj;
    ... use s ...
}

// As of Java 16
if (obj instanceof String s) {
    ... use s ...
}
```

在新的代码中，如果在运行时，obj的值是String的实例，那么obj将与类型模式String s匹配。如果模式匹配成功，则instanceof表达式为true，并且模式变量s将被初始化为obj强制转换为String的值，然后可以在包含的代码块中使用。

类型模式一举消除了许多强制转换的发生。然而，它们只是迈向更声明式、以数据为重心的编程风格的第一步。随着Java支持对数据建模的新颖和更富表现力的方式，模式匹配可以通过允许开发人员表达模型的语义意图，从而简化对这些数据的使用。

模式匹配和记录

Records (JEP 395)是数据的透明承载体。接收到record类实例的代码通常会使用内置的组件访问器方法提取数据，也称为组件。例如，我们可以使用类型模式来测试一个值是否是record类Point的实例，如果是，则从该值中提取x和y组件：

```
record Point(int x, int y) {}


static void printSum(Object obj) {
    if (obj instanceof Point p) {
        int x = p.x();
        int y = p.y();
        System.out.println(x + y);
    }
}
```

在这里，模式变量`p`仅用于调用访问器方法`x()`和`y()`，这些方法返回组件`x`和`y`的值。通过引入record patterns，可以更直接地从record实例中提取组件，而无需显式调用访问器方法。

```
// As of Java 16
record Point(int x, int y) {}


static void printSum(Object obj) {
    if (obj instanceof Point p) {
        int x = p.x();
        int y = p.y();
        System.out.println(x+y);
    }
}
```

在这里，模式变量`p`仅用于调用访问器方法`x()`和`y()`，这些方法返回组件`x`和`y`的值。（在每个record类中，其访问器方法和组件之间存在一对一的对应关系。）如果模式不仅能够测试一个值是否是Point的实例，还能够直接从该值中提取x和y组件，以我们的名义调用访问器方法，那将更好。换句话说：

```
// 截至 Java 21
static void printSum(Object obj) {
    if (obj instanceof Point(int x, int y)) {
        System.out.println(x + y);
    }
}
```

`Point(int x, int y)` 是一个record pattern。它将用于提取组件的本地变量的声明提升到模式本身，并在值与模式匹配时通过调用访问器方法来初始化这些变量。实际上，一个record pattern 将一个record实例解构成其组件。这样的语法使得处理record实例的数据更加简洁和直观。

嵌套记录模式

模式匹配的真正力量在于它可以优雅地扩展以匹配更复杂的对象图。例如，考虑以下声明：

```
// As of Java 16
record Point(int x, int y) {}
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

我们已经看到，我们可以使用记录模式提取对象的组成部分。如果我们想从左上角提取颜色，我们可以这样写：

```
// As of Java 21
static void printUpperLeftColoredPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint ul, ColoredPoint lr)) {
         System.out.println(ul.c());
    }
}
```

但该`ColoredPoint`值`ul`本身就是一个记录值，我们可能需要进一步分解它。因此，记录模式支持*嵌套*，这允许记录组件进一步匹配嵌套模式并由嵌套模式分解。我们可以在记录模式中嵌套另一个模式并同时分解外部和内部记录：

```
// As of Java 21
static void printColorOfUpperLeftPoint(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint(Point p, Color c),
                               ColoredPoint lr)) {
        System.out.println(c);
    }
}
```

嵌套模式进一步使我们能够使用与将其组合在一起的代码一样清晰简洁的代码来分解聚合。例如，如果我们要创建一个矩形，我们可能会将构造函数嵌套在单个表达式中：

```
// As of Java 16
Rectangle r = new Rectangle(new ColoredPoint(new Point(x1, y1), c1),
                            new ColoredPoint(new Point(x2, y2), c2));
```

通过嵌套模式，我们可以使用与嵌套构造函数的结构相呼应的代码来解构这样的矩形：

```
// As of Java 21
static void printXCoordOfUpperLeftPointWithPatterns(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint(Point(var x, var y), var c),
                               var lr)) {
        System.out.println("Upper-left corner: " + x);
    }
}
```

嵌套模式当然也有可能匹配失败：

```
// As of Java 21
record Pair(Object x, Object y) {}


Pair p = new Pair(42, 42);


if (p instanceof Pair(String s, String t)) {
    System.out.println(s + ", " + t);
} else {
    System.out.println("Not a pair of strings");
}
```

在这里，record pattern `Pair(String s, String t)` 包含两个嵌套的类型模式，即 `String s` 和 `String t`。当一个值匹配模式 `Pair(String s, String t)` 时，它必须是一个Pair，并且递归地，其组件值必须与类型模式 `String s` 和 `String t` 匹配。在我们上面的示例代码中，这些递归模式匹配失败，因为record组件的值都不是字符串，因此执行了`else`块。

总之，嵌套模式消除了导航对象时的偶发复杂性，使我们能够专注于这些对象所表达的数据。它们还赋予了我们集中处理错误的能力，因为如果子模式中的一个或两个都无法匹配，那么值就无法与嵌套模式 `P(Q)` 匹配。我们无需检查和处理每个单独的子模式匹配失败 — 要么整个模式匹配，要么不匹配。

描述

我们使用可嵌套记录模式扩展了 Java 编程语言。

模式的语法变为：

```
Pattern:
  TypePattern
  RecordPattern


TypePattern:
  LocalVariableDeclaration


RecordPattern:
  ReferenceType ( [ PatternList ] )


PatternList :
  Pattern { , Pattern }
```

记录模式

记录*模式*由记录类类型和（可能为空）模式列表组成，该列表用于匹配相应的记录组件值。

例如，给定声明

```
record Point(int i, int j) {}
```

一个值`v` 与 record 模式`Point(int i, int j)` 匹配，如果它是 record 类型`Point` 的实例；如果匹配成功，模式变量`i` 将被初始化为在值`v` 上调用与`i` 相对应的访问器方法的结果，模式变量`j` 将被初始化为在值`v` 上调用与`j` 相对应的访问器方法的结果。（模式变量的名称不必与 record 组件的名称相同；也就是说，record 模式`Point(int x, int y)` 的行为相同，只是模式变量`x` 和`y` 被初始化。）

null 值不与任何 record 模式匹配。

record 模式可以使用`var` 来匹配 record 组件，而无需指定组件的类型。在这种情况下，编译器会推断由`var` 模式引入的模式变量的类型。例如，模式`Point(var a, var b)` 是模式`Point(int a, int b)` 的简写。

由 record 模式声明的模式变量集合包括在模式列表中声明的所有模式变量。

表达式与 record 模式兼容，如果它可以在不需要未经检查的转换的情况下被转换为模式中的 record 类型。

如果 record 模式指定了泛型 record 类，但没有提供类型参数（即，record 模式使用原始类型），则类型参数总是会被推断。

例如：

```
// As of Java 21
record MyPair<S,T>(S fst, T snd){};
static void recordInference(MyPair<String, Integer> pair){
    switch (pair) {
        case MyPair(var f, var s) ->
            ... // Inferred record pattern MyPair<String,Integer>(var f, var s)
        ...
    }
}
```

`instanceof`所有支持记录模式的构造（即表达式、语句`switch` 和表达式）都支持记录模式的类型参数推断。

推理适用于嵌套记录模式；例如：

```
// As of Java 21
record Box<T>(T t) {}
static void test1(Box<Box<String>> bbs) {
    if (bbs instanceof Box<Box<String>>(Box(var s))) {
        System.out.println("String " + s);
    }
}
```

这里嵌套模式的类型参数`Box(var s)`被推断为 `String`，因此模式本身也被推断为`Box<String>(var s)`。

事实上，也可以删除外部记录模式中的类型参数，从而得到简洁的代码：

```
// As of Java 21
static void test2(Box<Box<String>> bbs) {
    if (bbs instanceof Box(Box(var s))) {
        System.out.println("String " + s);
    }
}
```

这里编译器将推断整个`instanceof`模式是 `Box<Box<String>>(Box<String>(var s))`，

为了兼容性，类型模式不支持类型参数的隐式推断；例如，类型模式`List l`始终被视为原始类型模式。

记录模式并详尽`switch`

JEP 441增强了`switch`表达式和`switch`语句以支持模式标签。无论是switch表达式还是模式switch语句都必须是穷举的(详尽的)：switch块必须包含处理选择器表达式的所有可能值的子句。对于模式标签，这是通过对模式类型的分析来确定的；例如，case标签`case Bar b` 匹配类型为Bar以及Bar的所有可能子类型的值。

对于涉及record patterns的模式标签，分析更为复杂，因为我们必须考虑组件模式的类型并对密封层次结构进行适当处理。例如，考虑以下声明：

```
class A {}
class B extends A {}
sealed interface I permits C, D {}
final class C implements I {}
final class D implements I {}
record Pair<T>(T x, T y) {}
Pair<A> p1;
Pair<I> p2;
```

以下的 switch 不是穷尽的，因为没有匹配包含两个类型为 A 的值的 Pair 的情况：

```
// As of Java 21
switch (p1) {                 // Error!
    case Pair<A>(A a, B b) -> ...
    case Pair<A>(B b, A a) -> ...
}
```

在这个例子中，没有匹配的情况来处理包含两个类型为 A 的值的 Pair。这会导致编译错误，因为该 switch 语句不满足穷尽性要求。

这两个 switch 是穷尽的，因为接口 I 是封闭的，所以类型 C 和 D 涵盖了所有可能的实例：

```
// As of Java 21
switch (p2) {
    case Pair<I>(I i, C c) -> ...
    case Pair<I>(I i, D d) -> ...
}
switch (p2) {
    case Pair<I>(C c, I i) -> ...
    case Pair<I>(D d, C c) -> ...
    case Pair<I>(D d1, D d2) -> ...
}
```

相反，这`switch`并不详尽，因为没有匹配包含两个类型为 D 的值的 Pair 的情况：

```
// As of Java 21
switch (p2) {                         // Error!
    case Pair<I>(C fst, D snd) -> ...
    case Pair<I>(D fst, C snd) -> ...
    case Pair<I>(I fst, C snd) -> ...
}
```

未来的工作

这里描述的记录模式可以在许多方向上扩展：

- Varargs 模式，用于具有可变参数数量的records；
- 匿名模式，可以出现在record模式的模式列表中，匹配任何值但不声明模式变量；
- 可以应用于任意类的值而不仅仅是record类的模式。

我们可能会在未来的 JEP 中考虑其中的一些内容。

依赖关系

该 JEP 基于 JDK 16 中提供的模式匹配`instanceof`(JEP 394)构建。它与*模式匹配*`switch` (JEP 441)共同发展。