

Java 11 VS Java 17

JDK 17的必要性

- JDK 11 作为一个 LTS版本，它的商业支持时间框架比 JDK 8 短，JDK 11 的 LTS 会提供技术支持直至 2023 年 9 月，对应的补丁和安全警告等支持将持续至 2026 年。JDK 17 作为下一代 LTS 将提供至少到 2026 年的支持时间框架；
- Java系最为重要的开发框架Spring Framework 6 和 Spring Boot 3对JDK版本的最低要求是JDK 17；所以可以预见，为了使用Spring最新框架，很多团队和开发者将被迫从Java 11（甚至Java 8)直接升级到Java 17版本。

新特性

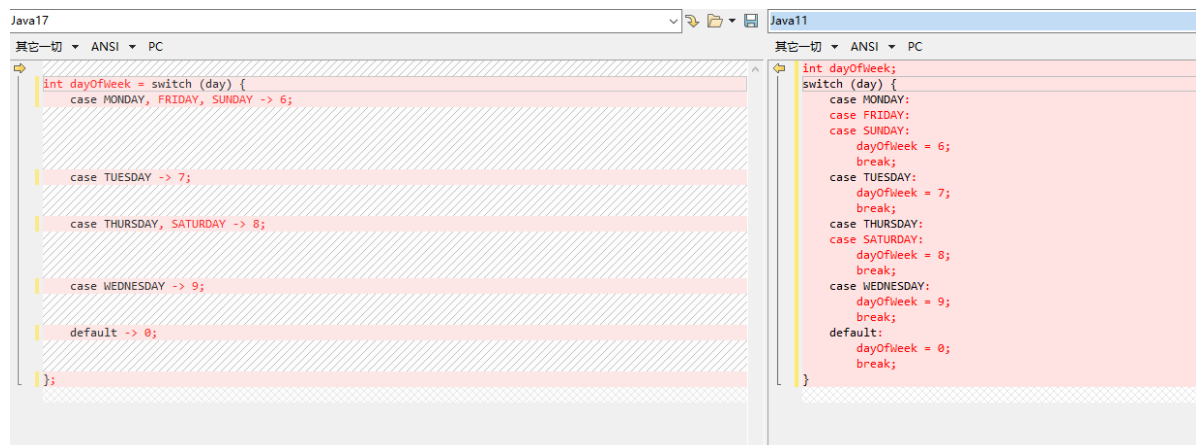
Switch表达式

Java 11

```
int dayOfWeek;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        dayOfWeek = 6;
        break;
    case TUESDAY:
        dayOfWeek = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        dayOfWeek = 8;
        break;
    case WEDNESDAY:
        dayOfWeek = 9;
        break;
    default:
        dayOfWeek = 0;
        break;
}
```

Java 17

```
int dayOfWeek = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY -> 7;
    case THURSDAY, SATURDAY -> 8;
    case WEDNESDAY -> 9;
    default -> 0;
};
```



- 之前需要用变量来接收返回值，而现在直接使用 yield 关键字来返回 case 分支需要返回的结果。
- 现在的 switch 表达式中不再需要显式地使用 return、break 或者 continue 来跳出当前分支。

- 现在不需要像之前一样，在每个分支结束之前加上 `break` 关键字来结束当前分支，如果不加，则会默认往后执行，直到遇到 `break` 关键字或者整个 `switch` 语句结束，在 `Java 14` 表达式中，表达式默认执行完之后自动跳出，不会继续往后执行。
- 对于多个相同的 `case` 方法块，可以将 `case` 条件并列，而不需要像之前一样，通过每个 `case` 后面故意不加 `break` 关键字来使用相同方法块。

文本块

Java 11

```
String textBlock = """
    This is a text block
    in JDK 17.
    It supports
    leading indentation.
    """;
```

Java 17

- 转义字符的使用

```
String textBlock = """
    This is a text block
    in JDK 17.
    It supports \n newline
    and \t tab characters.
    """;
```

- 字符串格式化

```
String name = "Alice";
String textBlock = """
    This is a text block
    in JDK 17.
    It can embed expressions, like this: %s.
    """.formatted(name);
```

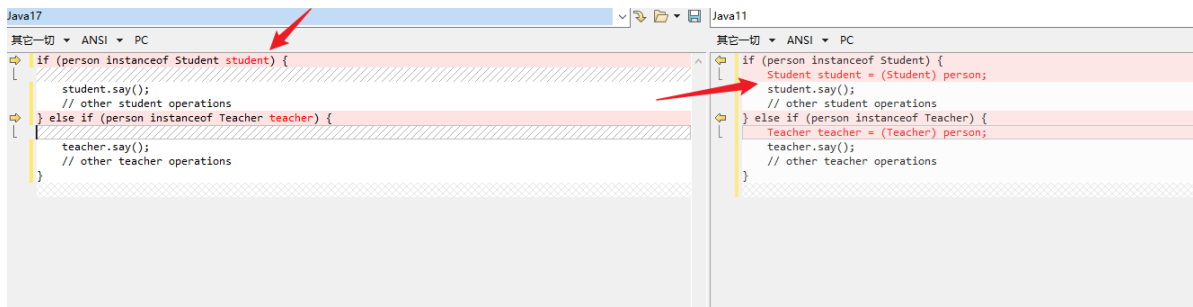
模式匹配

在以往实际使用中，`instanceof` 主要用来检查对象的类型，然后根据类型对目标对象进行类型转换，之后进行不同的处理、实现不同的逻辑，具体可以参考如下：

```
if (person instanceof Student) {
    Student student = (Student) person;
    student.say();
    // other student operations
} else if (person instanceof Teacher) {
    Teacher teacher = (Teacher) person;
    teacher.say();
    // other teacher operations
}
```

- 类型判断成功后不需要强制转换

```
if (person instanceof Student student) {
    student.say();
    // other student operations
} else if (person instanceof Teacher teacher) {
    teacher.say();
    // other teacher operations
}
```



- instanceof 类型匹配成功，模式局部变量的作用范围也可以相应延长

```
if (obj instanceof String s && s.length() > 5) {  
    // s 是 String 类型，并且在此范围内可以使用 s 的方法  
}
```

Records

Records是Java 14引入的一种新的类类型，它是一种轻量级的数据封装类型，用于简化创建不可变数据对象的过程。Records类提供了一种更简洁的方式来定义类，不需要显式地编写字段、构造函数、equals()、hashCode()和toString()等方法。这样，我们可以更专注于描述类的数据成员，而不用担心繁琐的重复代码。

- 定义Records类

要定义一个Records类，只需要使用 `record` 关键字，后面跟着类名和类的成员变量。

```
record Person(String name, int age) {  
}
```

上面的例子定义了一个简单的Person类，它有两个成员变量：`name` 和 `age`。

- 自动生成方法

Records类自动生成了以下方法：

1. 构造函数：Records类的构造函数会自动接收传入的成员变量，并将其赋值给相应的字段。
2. equals()和hashCode()：Records类会自动根据所有成员变量生成equals()和hashCode()方法，以便进行对象的相等性比较。
3. toString()：Records类会自动生成toString()方法，以便输出对象的字符串表示。

```
Person person = new Person("Alice", 30);  
System.out.println(person); // 输出: Person[name=Alice, age=30]
```

- 不可变性

Records类是不可变的，一旦创建了一个Records对象，其字段值不能再被修改。如果要修改字段值，需要创建一个新的Records对象。

- 继承

Records类可以继承其他类和实现接口，与普通的类一样。

```
record Employee(String name, int age, String department) extends Person {  
    // Subclass constructor  
    public Employee(String name, int age, String department) {  
        super(name, age);  
        this.department = department;  
    }  
}
```

- 模式匹配

Records类和模式匹配一起使用，可以更方便地处理对象类型。

```
if (person instanceof Person p) {
    System.out.println(p.name());
    System.out.println(p.age());
}
```

- 内部类用法

```
public class OuterClass {
    // Records类作为内部类
    public record Person(String name, int age) {
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        Person person = outer.new Person("Alice", 30);
        System.out.println(person); // 输出: Person[name=Alice, age=30]
    }
}
```

总结: Records类是一种简化数据对象创建的方式, 提供了一种轻量级的数据封装类型, 自动生成构造函数、equals()、hashCode()和toString()等方法。它可以增强代码的可读性, 减少了冗余代码的编写, 特别适用于表示数据的简单类。Records类在Java 14中被引入, 并在后续的Java版本中得到了改进和增强。

Sealed 关键字 -Java17独有

```
// 定义一个抽象类Animal, 使用sealed关键字来限制允许继承的子类
public sealed abstract class Animal permits Cat, Dog {

    // 定义一个抽象方法, 子类必须实现该方法
    public abstract void makeSound();

    // 定义一个静态方法, 用于创建具体的Animal实例
    public static Animal create(String animalType) {
        switch (animalType) {
            case "cat":
                return new Cat();
            case "dog":
                return new Dog();
            default:
                throw new IllegalArgumentException("Unknown animal type: " +
animalType);
        }
    }
}

// 具体的子类Cat, 继承自Animal
final class Cat extends Animal {

    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}

// 具体的子类Dog, 继承自Animal
final class Dog extends Animal {

    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

动态切换

- 低成本的切换
- 开关的存在
- 可扩展可维护的考量
- 前段后端和运维之间的优劣

前端

提供一种开关



后端

```
51 # tcc 配置
52 tcc:
53     model-name: account
54     retry-max: 3
55     db-config:
56         url: jdbc:mysql://localhost:3306/tcc_log?useUnicode=true&characterEncoding=utf8
57         username: root
58         password: 123456
59         minimum-idle: 20
60         maximum-pool-size: 100
61         idle-timeout: 30000
62         connection-timeout: 30000
63         connection-test-query: SELECT 1
64         max-lifetime: 1800000
```

Document 1/1 > tcc: > db-config: > max-lifetime: > 1800000

```
1 package com.scs.springcloud.start.config;
2
3 import com.scs.common.config.FrameConfig;
4 import org.springframework.boot.context.properties.ConfigurationProperties;
5 import org.springframework.boot.context.properties.EnableConfigurationProperties;
6 import org.springframework.stereotype.Component;
7
8
9 @Component
10 @ConfigurationProperties(prefix = "tcc",
11     ignoreInvalidFields = true)
12 public class ConfigProperties extends FrameConfig {
13
14 }
15
```

@ConditionalOnProperty

```
@Service
@ConditionalOnProperty(name = "app.feature.enabled", havingValue = "true",
    matchIfMissing = true)
public class OriginalService {
    // 原有功能
}

@Service
@ConditionalOnProperty(name = "app.feature.enabled", havingValue = "false")
public class NewService {
    // 新功能
}
```

```

@Configuration
public class FeatureConfig {
    @Bean
    @ConditionalOnProperty(name = "app.feature.enabled", havingValue = "true",
matchIfMissing = true)
    public MyService originalService() {
        return new OriginalService();
    }

    @Bean
    @ConditionalOnProperty(name = "app.feature.enabled", havingValue = "false")
    public MyService newService() {
        return new NewService();
    }
}

```

```

app:
  feature:
    enabled: true

```

运维

ConfigMap

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config-map
data:
  property1: value1
  property2: value2

```

spring.yml

```

# application.yaml
property1: ${property1}
property2: ${property2}

```

Spring Cloud Kubernetes Config

不需要应用重启

<https://juejin.cn/post/7166916547351412743>