目录

Java 5 新特性总结

Java 5（也称为 JDK 1.5 或 J2SE 5.0）是 Java 发展历程中一个革命性的版本，于 2004 年发布。与先前的大多数 Java 升级不同，该版本有很多重要的改进，从根本上扩展了 Java 语言的应用领域、功能和范围。Java 5 引入了大量重要的语言特性和 API，是 Java 发展史上的一个重要里程碑。

主要新特性

1. 泛型（Generics）

Java 5 引入了泛型机制，允许开发者在编译时检查类型安全，从而避免了运行时的 ClassCastException。

```
// 使用泛型前
List list = new ArrayList();
list.add("hello");
String str = (String) list.get(0); // 需要强制类型转换

// 使用泛型后
List<String> list = new ArrayList<String>();
list.add("hello");
String str = list.get(0); // 不需要强制类型转换，类型安全
```

2. 枚举类型（Enums）

Java 5 引入了枚举类型，允许开发者定义一组固定的常量。

```
public enum Color {
    RED, GREEN, BLUE;
}

// 使用枚举
Color color = Color.RED;
switch (color) {
    case RED:
        System.out.println(" 红色");
        break;
    case GREEN:
        System.out.println(" 绿色");
        break;
    case BLUE:
        System.out.println(" 蓝色");
        break;
}
```

3. 自动装箱/拆箱（Autoboxing/Unboxing）

Java 5 引入了自动装箱和自动拆箱机制，允许开发者在基本类型和对象类型之间进行自动转换。

```
// 自动装箱
List<Integer> list = new ArrayList<Integer>();
list.add(5); // 自动将 int 转换为 Integer
```

```java
// 自动拆箱
int i = list.get(0); // 自动将 Integer 转换为 int
```

4. 注解（Annotations）

Java 5 引入了注解机制，提供了一种形式化的方法来为代码添加元数据。

```java
@Override
public String toString() {
    return "MyClass";
}

@SuppressWarnings("deprecation")
public void deprecatedMethod() {
    // 使用被废弃的方法
}
```

5. 增强的 for 循环（Enhanced for Loop）

Java 5 引入了 foreach 循环，允许开发者使用简洁的语法来遍历集合。

```java
// 遍历数组
int[] array = {1, 2, 3, 4, 5};
for (int i : array) {
    System.out.println(i);
}

// 遍历集合
List<String> list = Arrays.asList("a", "b", "c");
for (String item : list) {
    System.out.println(item);
}
```

6. 可变参数（Varargs）

Java 5 引入了可变参数机制，允许方法接受可变数量的参数。

```java
public static void printNumbers(String format, int... numbers) {
    for (int num : numbers) {
        System.out.printf(format, num);
    }
}

// 调用方法
printNumbers("%d ", 1, 2, 3, 4, 5);
```

7. 静态导入（Static Import）

Java 5 引入了静态导入机制，允许开发者直接使用静态成员，而不需要使用类名。

```
import static java.lang.Math.PI;
import static java.lang.Math.max;

public class Example {
    public static void main(String[] args) {
        double area = PI * 5 * 5; // 直接使用 PI
        int maxVal = max(10, 20); // 直接使用 max 方法
    }
}
```

8. 协变返回类型（Covariant Return Types）

Java 5 允许在子类的覆盖方法中返回父类被覆盖方法的子类型。

```
class Animal {}
class Dog extends Animal {}

class AnimalFactory {
    public Animal createAnimal() {
        return new Animal();
    }
}

class DogFactory extends AnimalFactory {
    @Override
    public Dog createAnimal() { // 协变返回类型
        return new Dog();
    }
}
```

API 增强

1. 并发工具类（Concurrency Utilities）

Java 5 引入了 java.util.concurrent 包，提供了丰富的并发编程工具。

```
import java.util.concurrent.*;

// 使用线程池
ExecutorService executor = Executors.newFixedThreadPool(10);
Future<String> future = executor.submit(() -> "Hello World");
String result = future.get(); // 获取结果
```

```
// 使用 Callable 和 Future
Callable<Integer> task = () -> {
    // 执行一些计算
    return 42;
};
Future<Integer> futureResult = executor.submit(task);
```

2. Scanner 类

Java 5 引入了 Scanner 类，提供了更方便的输入解析功能。

```
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);
String input = scanner.nextLine();
int number = scanner.nextInt();
```

3. StringBuilder 类

虽然 StringBuffer 早已存在，但 Java 5 引入了 StringBuilder，提供了非线程安全但性能更好的字符串拼接。

```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("World");
String result = sb.toString();
```

4. 格式化功能

Java 5 增强了字符串格式化功能，引入了 printf 风格的格式化。

```
String formatted = String.format("Name: %s, Age: %d", "John", 25);
System.out.printf("Total: %.2f%n", 123.456);
```

5. 枚举的增强功能

枚举类提供了额外的方法，如 values() 和 valueOf()。

```
Color[] colors = Color.values(); // 获取所有枚举值
Color red = Color.valueOf("RED"); // 从字符串获取枚举值
```

虚拟机改进

1. 自动装箱/拆箱的性能优化

Java 5 对自动装箱/拆箱进行了性能优化。

2. 更好的垃圾收集

改进了垃圾收集算法，提高了性能。

3. Tiger 编译器优化

引入了新的编译器优化技术。

总结

Java 5 是 Java 发展史上的一个重要里程碑，引入了大量重要的语言特性和 API。这些新特性包括泛型、枚举、注解、自动装箱/拆箱、增强的 for 循环、可变参数、静态导入等，极大地提高了 Java 代码的类型安全性、可读性和可维护性。

Java 5 的并发工具包（java.util.concurrent）为多线程编程提供了更强大和易用的工具，使并发编程变得更加安全和高效。这些改进奠定了现代 Java 开发的基础，影响了后续所有 Java 版本的发展方向。

Java 5 的发布标志着 Java 语言从简单面向对象语言向更成熟、更强大的编程平台的转变，为 Java 在企业级应用开发中的广泛应用奠定了坚实基础。

Java 6 新特性总结

Java 6（也称为 Java SE 6）是 Java 平台的一个重要版本，于 2006 年 12 月发布。与 Java 5 相比，Java 6 没有引入太多新的语言特性，而更多的是以稳定、提高性能和质量为目标。尽管如此，Java 6 仍引入了许多重要的 API 和功能增强。

主要新特性

1. 脚本语言支持（Scripting Support）

Java 6 引入了对脚本语言的支持，允许在 Java 应用程序中嵌入脚本引擎。

```
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;

// 使用脚本引擎执行 JavaScript
ScriptEngineManager manager = new ScriptEngineManager();
```

```
ScriptEngine engine = manager.getEngineByName("JavaScript");
Object result = engine.eval("function hello(name) { return 'Hello, ' + name; } hello('World
System.out.println(result); // 输出: Hello, World
```

2. Compiler API

Java 6 提供了编译器 API，允许在运行时动态编译 Java 源代码。

```
import javax.tools.ToolProvider;
import javax.tools.JavaCompiler;
import java.io.File;

// 使用编译器 API 动态编译 Java 源代码
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
int result = compiler.run(null, null, null, "MyClass.java");
if (result == 0) {
    System.out.println(" 编译成功");
} else {
    System.out.println(" 编译失败");
}
```

3. 轻量级 HTTP 服务器（Lightweight HTTP Server)

Java 6 引入了内置的轻量级 HTTP 服务器 API。

```
import com.sun.net.httpserver.HttpServer;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpExchange;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;

// 创建轻量级 HTTP 服务器
HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);
server.createContext("/test", new MyHandler());
server.setExecutor(null);
server.start();

static class MyHandler implements HttpHandler {
    @Override
    public void handle(HttpExchange t) throws IOException {
        String response = "This is the response";
        t.sendResponseHeaders(200, response.length());
        OutputStream os = t.getResponseBody();
        os.write(response.getBytes());
        os.close();
```

```
    }
}
```

4. Common Annotations（JSR 250）

Java 6 引入了对通用注解的支持，包括 @PostConstruct、@PreDestroy、@Resource 等。

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.annotation.Resource;

public class MyService {
    @Resource
    private DataSource dataSource;

    @PostConstruct
    public void init() {
        // 初始化代码
        System.out.println(" 初始化完成");
    }

    @PreDestroy
    public void cleanup() {
        // 清理代码
        System.out.println(" 清理完成");
    }
}
```

5. JDBC 4.0

Java 6 引入了 JDBC 4.0，带来了多项改进：

- 自动驱动程序加载
- RowId 支持
- SQL/XML 支持
- 大对象（LOB）改进

```
// JDBC 4.0 自动加载驱动程序
Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/test", "user", "password");

// 使用 RowId
PreparedStatement pstmt = conn.prepareStatement(
    "INSERT INTO table VALUES (?, ?)");
RowId rowId = pstmt.executeReturningGeneratedKeys().getRowId(1);
```

6. JAXB 2.0（Java Architecture for XML Binding）

JAXB 2.0 提供了更好的 XML 和 Java 对象之间的映射功能。

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Person {
    private String name;
    private int age;

    // getters and setters
}
```

7. StAX（Streaming API for XML）

StAX 提供了流式 XML 处理 API，允许对 XML 文档进行逐事件的解析和生成。

```
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import java.io.FileReader;

// 使用 StAX 解析 XML
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader reader = factory.createXMLStreamReader(
    new FileReader("data.xml"));

while (reader.hasNext()) {
    int event = reader.next();
    if (event == XMLStreamReader.CHARACTERS) {
        System.out.println(reader.getText());
    }
}
```

8. Console 类

Java 6 引入了 Console 类，提供了与控制台交互的更好支持。

```
import java.io.Console;

// 使用 Console 读取密码
Console console = System.console();
if (console != null) {
    char[] password = console.readPassword("Enter password: ");
    String userInput = console.readLine("Enter username: ");
}
```

9. Web Services Metadata（JSR 181）

Java 6 引入了 Web 服务元数据支持，简化了 Web 服务的开发。

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Calculator {
    @WebMethod
    public int add(int a, int b) {
        return a + b;
    }
}
```

10. Java DB (Apache Derby)

Java 6 集成了 Apache Derby 作为内置的纯 Java 数据库。

性能和质量改进

1. 性能提升

Java 6 在服务器端和客户端版本都有了两位数百分比的提高，根据不同领域，性能提高了 20%-40%。

2. 更高的质量

- 兼容性测试包含超过 100,000 个测试程序
- 首个在社区模式下开放开发的 Java 版本
- 更高的稳定性和可靠性

3. 安全功能增强

Java 6 包含了一系列新的安全相关的增强功能。

GUI 增强

Java 6 对 Swing 和 AWT 进行了多项改进，包括更好的外观和感觉支持、性能优化等。

总结

Java 6 虽然不像 Java 5 那样引入了大量新的语言特性，但仍然带来了许多重要的 API 增强和功能改进。它重点关注性能提升、稳定性改进和安全性增强，为后续版本的发展奠定了基础。

Java 6 的发布标志着 Java 平台在企业级应用开发方面更加成熟，特别是在 Web 服务、数据库连接、脚本语言集成等方面提供了更好的支持。这些改进使得 Java 6 成为了当时企业级开发的重要平台。

Java 7 新特性总结

Java 7（代号 Dolphin）是甲骨文在 2011 年发布的第一个 Java 版本，它是 Oracle 工程师和全球 Java 社区成员广泛合作的结果。Java 7 包含了很多变化，虽然比开发人员预期的要少，但它引入了许多重要的语言和 API 改进。

主要新特性

1. 在 switch 语句中使用 String

Java 7 允许在 switch 语句中使用 String 类型，这是非常实用的改进。

```
public String generate(String name, String gender) {
    String title = "";
    switch (gender) {
        case " 男":
            title = name + " 先生";
            break;
        case " 女":
            title = name + " 女士";
            break;
        default:
            title = name;
    }
    return title;
}
```

2. 自动资源管理（Try-with-resources）

Java 7 引入了 try-with-resources 语句，自动管理资源的关闭，无需手动在 finally 块中关闭资源。

```
// 以前的写法
BufferedReader br = new BufferedReader(new FileReader(path));
try {
    return br.readLine();
} finally {
    br.close();
}

// Java 7 的写法
```

```
try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
}
```

3. 改进的泛型实例创建类型推断（Diamond Operator）

Java 7 引入了钻石操作符（<>），可以从声明中推断泛型类型。

```
// 以前的写法
Map<String, List<String>> anagrams = new HashMap<String, List<String>>();

// Java 7 的写法
Map<String, List<String>> anagrams = new HashMap<>();
```

4. 数字字面量中的下划线支持

Java 7 允许在数字字面量中使用下划线来提高可读性。

```
int one_million = 1_000_000;            // 更易读的数字
long creditCardNumber = 1234_5678_9012_3456L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
```

5. 二进制字面量

Java 7 支持直接使用二进制表示整数，使用 0b 或 0B 前缀。

```
int binary = 0b001001;      // 十进制数字 9
int binary2 = 0B001001;     // 十进制数字 9
```

6. 多异常捕获（Multi-catch Exception Handling）

Java 7 允许在一个 catch 块中捕获多个异常类型。

```
try {
    // 一些可能抛出异常的代码
} catch (IOException | SQLException ex) {
    // 处理 IOException 或 SQLException
    logger.error(ex);
}
```

7. 对集合类的语言级支持（Project Coin）

虽然 Java 7 最终没有完全实现对集合类的字面量支持，但这一特性曾被讨论过。
实际上，Java 7 中并未包含此功能。

JVM 和底层改进

1. JSR 292: 动态语言支持

Java 7 通过新的 invoke 动态字面量支持动态语言，使 Java 代码可以使用 Python、Ruby、Perl、JavaScript 和 Groovy 等非 Java 语言实现的代码。

2. 压缩的 64 位指针

JVM 的内部优化，使用压缩的 64 位指针，因此消耗的内存更少。

3. G1 垃圾回收器

引入了新的 G1（Garbage First）垃圾回收器，旨在替代 CMS 垃圾回收器，提供更好的性能和可预测的暂停时间。

并发性改进

Fork/Join 框架

Java 7 引入了 Fork/Join 框架，用于更好地支持并行计算，利用多核处理器的优势。

```
import java.util.concurrent.RecursiveTask;

public class Fibonacci extends RecursiveTask<Integer> {
    final int n;
    Fibonacci(int n) { this.n = n; }
    public Integer compute() {
        if (n <= 1) return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }
}
```

I/O 改进

NIO.2 (New I/O 2)

Java 7 引入了 NIO.2，提供了更好的文件系统支持和异步 I/O 操作。

```
import java.nio.file.*;

// 使用 NIO.2 进行文件操作
Path path = Paths.get("example.txt");
```

```
String content = Files.readAllLines(path).stream()
    .collect(Collectors.joining("\n"));
```

总结

Java 7 虽然不像 Java 8 那样引入了 Lambda 表达式等重大语言特性，但它仍然带来了许多实用的改进，特别是在资源管理、异常处理和代码可读性方面。try-with-resources 语句、钻石操作符、数字字面量改进等特性大大简化了 Java 代码的编写和维护。

尽管 Project Coin 中的一些激进想法（如对集合字面量的支持）没有实现，但 Java 7 的改进为后续版本的发展奠定了基础。Java 7 的发布标志着 Java 平台在现代化道路上的稳步前进，为开发者提供了更简洁、更安全的编程方式。

Java 8 新特性

1. Lambda 表达式

Lambda 表达式是 Java 8 最重要的特性之一，它允许用简洁的方式定义匿名函数。

语法： `(parameters) -> expression` 或 `(parameters) -> { statements }`

示例：

```
// 传统写法
Comparator<Integer> comparator1 = new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo(o2);
    }
};

// Lambda 写法
Comparator<Integer> comparator2 = (o1, o2) -> o1.compareTo(o2);
```

2. 函数式接口（Functional Interface）

函数式接口是只有一个抽象方法的接口，可以使用 Lambda 表达式实现。

常见函数式接口： - `java.util.function.Function<T, R>` - 接收一个参数，返回一个结果 - `java.util.function.Consumer<T>` - 接收一个参数，无返回值 - `java.util.function.Supplier<T>` - 无参数，返回一个结果 - `java.util.function.Predicate<T>` - 接收一个参数，返回 boolean

示例：

```java
Function<String, Integer> function = s -> s.length();
System.out.println(function.apply("hello")); // 输出: 5

Predicate<Integer> predicate = x -> x > 10;
System.out.println(predicate.test(15)); // 输出: true
```

3. Stream API

Stream API 提供了一种高效、优雅的方式来处理集合数据，支持函数式编程风格。

主要特点： - 允许以声明性方式处理数据集合 - 支持并行处理 (Parallel Stream) - 支持链式调用

常见操作： - **中间操作**：`filter`、`map`、`flatMap`、`distinct`、`sorted` 等 (返回 Stream) - **终端操作**：`collect`、`forEach`、`reduce`、`count` 等 (返回非 Stream)

示例：

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

// 找出所有偶数，平方后求和
int result = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .reduce(0, Integer::sum);
System.out.println(result); // 输出: 56

// 并行流处理
numbers.parallelStream()
    .filter(n -> n > 3)
    .forEach(System.out::println);
```

4. 方法引用 (Method Reference)

方法引用是一种更加简洁的 Lambda 表达式形式。

四种方式： 1. **静态方法引用**：`ClassName::staticMethodName` 2. **实例方法引用**：`instance::methodName` 3. **类的任意对象的实例方法引用**：`ClassName::methodName` 4. **构造方法引用**：`ClassName::new`

示例：

```java
// 静态方法引用
Function<String, Integer> function = Integer::parseInt;
System.out.println(function.apply("123")); // 输出: 123

// 实例方法引用
String str = "hello";
```

```
Consumer<String> consumer = System.out::println;
consumer.accept(str); // 输出: hello

// 构造方法引用
Supplier<List<String>> supplier = ArrayList::new;
List<String> list = supplier.get();
```

5. 接口默认方法（Default Method）和静态方法

Java 8 允许在接口中定义默认方法和静态方法。

示例：

```
interface Animal {
    // 默认方法
    default void eat() {
        System.out.println("Animal is eating");
    }

    // 静态方法
    static void staticMethod() {
        System.out.println("Static method in interface");
    }
}

class Dog implements Animal {
    // 可以不实现 eat() 方法，直接使用默认实现
}
```

6. Optional 类

`Optional` 是一个容器类，用于处理可能为 null 的值，能有效地减少 NullPointerException。

常见方法： - `Optional.of(T value)` - 创建一个包含非空值的 Optional - `Optional.ofNullable(T value)` - 创建一个 Optional，可以包含 null - `isPresent()` - 检查值是否存在 - `get()` - 获取值（如果值不存在则抛出异常） - `getOrElse(T other)` - 获取值或返回默认值 - `map(Function<T, U> mapper)` - 对值进行转换 - `filter(Predicate<T> predicate)` - 对值进行过滤

示例：

```
Optional<String> optional = Optional.of("Hello");
System.out.println(optional.isPresent()); // 输出: true
System.out.println(optional.get()); // 输出: Hello

Optional<String> emptyOptional = Optional.empty();
System.out.println(emptyOptional.orElse("Default")); // 输出: Default
```

```java
Optional<Integer> optionalInt = Optional.ofNullable(null);
optionalInt.ifPresent(System.out::println); // 不会输出任何内容
```

7. 日期和时间 API（java.time）

Java 8 引入了新的日期和时间 API，替代了旧的 `Date` 和 `Calendar` 类。

主要类： - `LocalDate` - 表示日期（年月日） - `LocalTime` - 表示时间（时分秒） - `LocalDateTime` - 表示日期和时间 - `Instant` - 表示时间戳 - `Duration` - 表示持续时间 - `Period` - 表示时间段 - `ZonedDateTime` - 表示带时区的日期和时间

示例：

```java
// 当前日期和时间
LocalDate today = LocalDate.now();
System.out.println(today); // 输出：2026-01-14

LocalDateTime now = LocalDateTime.now();
System.out.println(now); // 输出：2026-01-14T16:29:14.123456

// 创建指定日期
LocalDate date = LocalDate.of(2026, 1, 14);
LocalTime time = LocalTime.of(16, 29, 14);

// 日期计算
LocalDate tomorrow = today.plusDays(1);
LocalDate nextMonth = today.plusMonths(1);

// 比较
boolean isAfter = tomorrow.isAfter(today); // true
boolean isBefore = today.isBefore(tomorrow); // true

// 格式化
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
String formattedDate = now.format(formatter);
System.out.println(formattedDate); // 输出：2026-01-14 16:29:14
```

8. 其他特性

1) 重复注解（Repeatable Annotations） 允许在同一个元素上多次使用同一个注解。

```java
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Schedules.class)
public @interface Schedule {
    String value();
```

```
}

@Retention(RetentionPolicy.RUNTIME)
public @interface Schedules {
    Schedule[] value();
}

@Schedule("morning")
@Schedule("afternoon")
public void task() {
    // ...
}
```

2) 类型注解（Type Annotations） 允许在任何使用类型的地方使用注解。

```
@NotEmpty String str = "hello";
List<@NonNull String> list = new ArrayList<>();
```

3) Base64 编码和解码 Java 8 提供了原生的 Base64 编码和解码支持。

```
String originalString = "Hello World";
String encodedString = Base64.getEncoder().encodeToString(originalString.getBytes());
System.out.println("Encoded: " + encodedString); // Encoded: SGVsbG8gV29ybGQ=

String decodedString = new String(Base64.getDecoder().decode(encodedString));
System.out.println("Decoded: " + decodedString); // Decoded: Hello World
```

4) 新的 HashMap 构造函数

```
Map<String, Integer> map = new HashMap<String, Integer>() {{
    put("one", 1);
    put("two", 2);
    put("three", 3);
}};
```

总结

Java 8 是 Java 语言的重要里程碑版本，它引入了函数式编程范式，极大地简化了代码编写，提高了开发效率。核心贡献包括：

- **函数式编程支持**：Lambda 表达式和 Stream API
- **增强的日期和时间 API**：更直观、更易用的时间处理
- **更灵活的接口设计**：默认方法和静态方法
- **更好的空值处理**：Optional 类减少 NullPointerException

这些特性使得 Java 语言更加现代化、易用，同时保持了向后兼容性。

Java 9 新 特性

概述

Java 9 是 Java 平台的一个重要里程碑版本，于 2017 年 9 月发布。这个版本引入了许多革命性的新特性，为 Java 生态系统的现代化奠定了基础。

主要新特性

1. 模块系统 (Project Jigsaw)

Java 9 引入了全新的模块系统，这是 Java 平台历史上最重要的变化之一。

特点：

- **模块化 JDK**：将 JDK 按功能模块化组织
- **模块描述文件**：使用 `module-info.java` 定义模块
- **强封装性**：可以精确控制哪些包对外可见
- **依赖管理**：显式声明模块间的依赖关系

示例代码：

```
// module-info.java
module com.example.app {
    requires java.base;
    requires java.sql;
    exports com.example.app.api;
}
```

2. JShell (REPL 工具)

JShell 是 Java 9 引入的交互式编程工具，允许开发者快速测试代码片段。

特点：

- **即时执行**：无需创建完整的类和方法
- **代码补全**：支持 Tab 键自动补全
- **即时反馈**：立即看到代码执行结果
- **学习友好**：非常适合学习和演示 Java 语法

使用示例：

```
jshell> int add(int a, int b) { return a + b; }
 created method add(int,int)
```

```
jshell> add(5, 3)
$3 ==> 8
```

3. 集合工厂方法

Java 9 为集合创建提供了简洁的工厂方法。

不可变集合创建：

```
// Java 9 之前
List<String> list = Collections.unmodifiableList(Arrays.asList("a", "b", "c"));

// Java 9 之后
List<String> list = List.of("a", "b", "c");
Set<String> set = Set.of("a", "b", "c");
Map<String, Integer> map = Map.of("a", 1, "b", 2);
```

特点：

- **简洁语法**：一行代码创建不可变集合
- **空值限制**：不允许包含 null 元素
- **线程安全**：创建的集合天然不可变
- **性能优化**：针对小集合做了特殊优化

4. 接口私有方法

Java 9 允许在接口中定义私有方法。

用途：

- **代码复用**：在接口内部共享通用逻辑
- **封装实现细节**：隐藏辅助方法
- **保持接口简洁**：避免公开内部实现方法

示例：

```
public interface MyInterface {
    void publicMethod();

    default void defaultMethod() {
        helperMethod();
        // 其他逻辑
    }

    private void helperMethod() {
        // 私有方法，只能在接口内部使用
```

```
        System.out.println("Helper logic");
    }
}
```

5. HTTP/2 客户端 API (孵化)

Java 9 引入了现代化的 HTTP 客户端 API，支持 HTTP/2 和 WebSocket。

特点：

- **HTTP/2 支持**：支持多路复用和头部压缩
- **异步编程**：基于 CompletableFuture 的异步 API
- **WebSocket 支持**：原生支持 WebSocket 协议
- **流式 API**：流畅的构建器模式

示例：

```
HttpClient client = HttpClient.newBuilder()
    .version(HttpClient.Version.HTTP_2)
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.example.com"))
    .GET()
    .build();

HttpResponse<String> response = client.send(request,
    HttpResponse.BodyHandlers.ofString());
```

6. 改进的 Stream API

Java 9 增强了 Stream API，增加了新的中间操作和终止操作。

新增方法：

```
// takeWhile - 获取满足条件的元素
List.of(1, 2, 3, 4, 5, 4, 3, 2, 1)
    .stream()
    .takeWhile(n -> n < 4)
    .collect(Collectors.toList()); // [1, 2, 3]

// dropWhile - 跳过满足条件的元素
List.of(1, 2, 3, 4, 5, 4, 3, 2, 1)
    .stream()
    .dropWhile(n -> n < 4)
```

```
    .collect(Collectors.toList()); // [4, 5, 4, 3, 2, 1]

// ofNullable - 处理可能为 null 的元素
Stream.ofNullable(null).count(); // 0
Stream.ofNullable("hello").count(); // 1

// iterate - 带终止条件的迭代
Stream.iterate(1, n -> n * 2, n -> n <= 64)
    .forEach(System.out::println);
```

7. 响应式流 (Reactive Streams)

Java 9 引入了响应式流 API，为异步流处理提供标准接口。

核心接口：

- `Flow.Publisher` - 数据发布者
- `Flow.Subscriber` - 数据订阅者
- `Flow.Subscription` - 订阅管理
- `Flow.Processor` - 数据处理器

8. 改进的 Process API

改进了进程 API，更好地支持操作系统进程管理。

增强功能：

- **进程句柄**：`ProcessHandle` 接口提供更丰富的进程信息
- **进程信息**：获取进程 PID、命令行、启动时间等
- **进程管理**：创建子进程、等待进程结束、销毁进程

示例：

```
ProcessHandle.current()
    .info()
    .commandLine()
    .ifPresent(System.out::println);
```

9. 多版本兼容 JAR

允许 JAR 文件包含针对不同 Java 版本的类文件。

结构：

```
mylib.jar
   META-INF/
       versions/
           9/
               com/example/MyClass.class
           10/
                com/example/MyClass.class
   com/example/
       MyClass.class（默认版本）
   META-INF/MANIFEST.MF
```

10. 改进的 try-with-resources

Java 9 改进了 try-with-resources 语句，支持使用已声明的资源。

对比：

```
// Java 8
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    // 使用 br
}

// Java 9
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
try (br) {
    // 使用 br
}
```

11. 钻石操作符改进

钻石操作符在匿名内部类中可以更好地推断类型。

```
// Java 9
List<String> list = new ArrayList<>() { // 可以省略泛型参数
    {
        add("hello");
    }
};
```

12. 改进的警告信息

Java 9 改进了编译器警告信息，提供更清晰的错误提示。

- **弃用警告**：更详细的弃用信息
- **模块系统警告**：模块相关的警告信息

- **原始类型警告**：更精确的原始类型使用警告

性能改进

1. GC 改进

- **G1 收集器**：成为默认垃圾收集器
- **并行 Full GC**：改进 G1 的 Full GC 性能
- **堆分配**：更好的大内存分配性能

2. 编译器优化

- **JIT 编译器**：改进的方法内联
- **字符串优化**：更好的字符串处理性能

安全性改进

1. 加密增强

- **更强的加密算法**：支持更多现代加密标准
- **密钥管理**：改进的密钥管理系统

2. TLS 1.3 支持

为未来的 TLS 1.3 标准做准备。

开发工具改进

1. 编译器改进

- **诊断信息**：更丰富的错误和警告信息
- **性能改进**：更快的编译速度

2. 调试工具

- **jcmd**：增强的诊断命令工具
- **jmap**：改进的内存映射工具
- **jstat**：增强的统计信息工具

向后兼容性

1. 保持兼容

Java 9 保持了与 Java 8 的高度向后兼容性，大部分现有代码无需修改即可运行。

2. 弃用的功能

- **Applet API**：标记为弃用（后续版本移除）
- **Java EE 和 CORBA 模块**：从 JDK 中移除

升级建议

1. 模块化迁移

- 评估现有项目的模块化需求
- 逐步引入模块系统
- 优先模块化核心组件

2. API 迁移

- 使用新的集合工厂方法
- 替换过时的 API 调用
- 采用新的 HTTP 客户端

3. 开发实践

- 学习和使用 JShell
- 采用新的 Stream API 特性
- 使用接口私有方法优化代码结构

总结

Java 9 是一个具有里程碑意义的版本，它不仅引入了许多实用的新特性，更重要的是通过模块系统为 Java 平台的未来发展奠定了基础。虽然模块系统的学习曲线相对陡峭，但它带来的强封装性和可维护性是值得的。对于新项目，建议直接采用 Java 9+ 的特性；对于现有项目，可以根据实际需求逐步迁移。

Java 9 的设计理念体现了现代化编程语言的发展趋势：更好的模块化、更强的类型安全、更简洁的语法、更优秀的性能，这些特性为 Java 在云原生时代的发展奠定了坚实基础。

Java 10 新特性

1. 局部变量类型推断（Local Variable Type Inference）

Java 10 引入了 `var` 关键字，允许编译器自动推断局部变量的类型。

优点： - 减少冗余的类型声明 - 提高代码的可读性 - 编译器进行类型检查，保证类型安全

示例：

```java
// Java 9 及之前
List<Integer> list = new ArrayList<Integer>();
Map<String, String> map = new HashMap<String, String>();

// Java 10 使用 var
var list = new ArrayList<Integer>();
var map = new HashMap<String, String>();

// 其他使用场景
var message = "Hello World"; // String
var numbers = new int[]{1, 2, 3, 4, 5}; // int[]
var stream = list.stream(); // Stream<Integer>

// var 的限制条件
// 1. 只能用于局部变量，不能用于成员变量、方法参数、返回类型
// 2. 变量必须在声明时初始化
// 3. 不能初始化为 null
```

2. 不可变集合（Unmodifiable Collections）

Java 10 提供了便捷的方式创建不可变集合。

示例：

```java
// 创建不可变 List
List<String> immutableList = List.of("a", "b", "c");
// immutableList.add("d"); // 会抛出 UnsupportedOperationException

// 创建不可变 Set
Set<String> immutableSet = Set.of("apple", "banana", "orange");

// 创建不可变 Map
Map<String, Integer> immutableMap = Map.of("one", 1, "two", 2, "three", 3);
```

3. 垃圾回收改进

- **G1GC（Garbage First GC）**变为默认垃圾回收器
- 改进了并发标记的线程数量设置
- 优化了完整 GC 的性能

4. 应用类数据共享（Application Class Data Sharing, AppCDS）

允许应用程序类被存档到共享归档文件中，在多个 JVM 实例之间共享。

优点： - 减少应用启动时间 - 降低内存占用

5. 线程本地握手（Thread-Local Handshake）

允许线程相互请求停止，而无需全局虚拟机安全点。

优点： - 减少 STW（Stop-The-World）的停顿时间 - 提高应用的响应性

6. 删除的特性

- **删除 JDK 8 中的 Nashorn JavaScript 引擎**
- **删除 applet 支持**
- **删除 JNLP（Java Network Launch Protocol）支持**

7. 其他改进

- **容器感知（Container Awareness）**：JVM 能更好地识别容器环境的资源限制
- **新增的 API**：`ProcessHandle` 增强，提供获取进程 ID 和进程树的能力
- **Java 源文件的直接执行**：可以直接运行 `.java` 文件而无需先编译

示例：

```
# Java 10 支持直接运行 Java 文件
java HelloWorld.java
```

总结

Java 10 是一个重要的改进版本，主要聚焦于简化开发体验和提升性能：

- `var` **关键字**：简化了局部变量声明，减少样板代码
- **不可变集合**：提供了更便捷的不可变集合创建方式
- **GC 优化**：G1GC 成为默认，提升了垃圾回收性能
- **容器支持**：更好地支持云原生和容器化部署环境
- **开发者体验**：删除过时特性，优化开发流程

Java 10 虽然是一个短期支持版本（仅 6 个月），但它引入的 `var` 关键字等特性在后续版本中得到了保留和广泛应用。

Java 11 新特性总结

Java 11 是 Oracle 于 2018 年 9 月 25 日发布的长期支持（LTS）版本，是 Java 8 之后最重要的版本之一。Java 11 不仅包含许多重要的语言、API 和性能改进，还引入了一些新的功能，让 Java 更加现代化。

主要新特性

1. HTTP Client 标准化

Java 11 将新的 HTTP Client API 正式加入标准库（位于 `java.net.http` 包），取代了之前的 HttpURLConnection API。

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.example.com"))
    .build();
HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println("Response: " + response.body());
```

- 支持 HTTP/1.1 和 HTTP/2
- 支持同步和异步请求
- 支持 WebSocket

2. 字符串增强

Java 11 为 String 类添加了多个实用的新方法：

- `isBlank()` - 检查字符串是否为空白
- `lines()` - 将字符串按行分割为流
- `strip()` - 去除前导和尾随空白
- `stripLeading()` - 去除前导空白
- `stripTrailing()` - 去除尾随空白
- `repeat(int count)` - 重复字符串指定次数

```
String str = "  Hello World  ";
System.out.println(str.strip()); // "Hello World"
System.out.println(str.isBlank()); // false
System.out.println("Hello\nWorld".lines().count()); // 2
System.out.println("Java".repeat(3)); // "JavaJavaJava"
```

3. 局部变量类型推断增强

Java 11 扩展了 Java 10 引入的 `var` 关键字，允许在 Lambda 表达式的参数中使用 `var`：

```
// 在 Lambda 表达式中使用 var
List<String> list = Arrays.asList("apple", "banana", "cherry");
list.forEach((@Nonnull var item) -> System.out.println(item));
```

4. 文件操作增强

新增了两个便捷的文件操作方法：

- `Files.readString(Path)` - 读取文件内容为字符串

- `Files.writeString(Path, String)` - 将字符串写入文件

```
Path path = Path.of("example.txt");
String content = Files.readString(path);
Files.writeString(path, "Hello Java 11!");
```

5. ZGC 垃圾收集器

Java 11 引入了 ZGC（Z Garbage Collector），这是一个可伸缩的低延迟垃圾收集器：

- 最大暂停时间不超过 10 毫秒
- 支持 TB 级别的堆内存
- 适用于需要低延迟的大规模应用程序

6. Epsilon GC

Java 11 引入了 Epsilon GC，一个”无操作”垃圾收集器：

- 仅负责内存分配，不做任何回收操作
- 适用于运行时间很短的应用程序
- 主要用于性能测试和压力测试

7. 动态类文件常量（Dynamic Class-File Constants）

Java 11 支持动态类文件常量，这为未来 Java 语言的发展提供了基础。

8. Flight Recorder 和 Mission Control

Java Flight Recorder 和 Java Mission Control 从 JRockit 移植到 OpenJDK，并开源：

- 用于收集有关正在运行的 JVM 以及应用程序的数据
- 对性能影响极小（小于 1%）
- 有助于分析和诊断问题

9. 单文件源码程序

Java 11 正式支持直接运行单个源文件，无需预先编译：

```
java HelloWorld.java
```

这使得 Java 更适合编写脚本和小型程序。

10. Nest-Based 访问控制

引入了基于 Nest 的访问控制机制：

- 同一个类文件中的嵌套类可以访问彼此的私有成员
- 减少了编译器生成的桥接方法
- 提高了代码的可维护性

已删除的内容

1. 删除的 API

- Java EE 和 CORBA 模块被移除
- JavaFX 被移除（独立项目）
- Applet API 被弃用

2. 其他变化

- 移除了 Nashorn JavaScript 引擎
- 移除了 Pack200 工具和 API

总结

Java 11 作为继 Java 8 之后最重要的 LTS 版本，引入了大量实用的新特性和改进。这些新特性不仅提升了开发效率，也改善了性能和安全性。特别是 HTTP Client 标准化、字符串增强和垃圾收集器改进，为现代 Java 应用提供了更好的支持。

Java 11 的发布标志着 Java 平台进入了一个新的发展阶段，也为后续版本奠定了坚实的基础。

Java 12 新特性总结

Java 12 是 Oracle 于 2019 年 3 月 19 日发布的版本，作为 Java 11 之后的第一个版本，Java 12 带来了许多有用的新特性和改进，虽然它不是长期支持（LTS）版本，但仍包含了一些重要的功能更新。

主要新特性

1. Switch 表达式（预览特性）

Java 12 引入了 Switch 表达式作为预览特性，它允许 switch 不仅可以用作语句，还可以用作表达式，可以直接返回值并使用简洁的”箭头语法”。

```
// 传统的 switch 语句
int numLetters = 0;
switch (day) {
    case MONDAY:
    case FRIDAY:
```

```
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    default:
        numLetters = 10;
}

// Java 12 的 Switch 表达式
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    default                     -> 10;
};
```

2. 字符串增强

Java 12 为 String 类添加了几个新的实用方法：

indent() - 调整字符串每行的缩进

```
String text = "Hello\nWorld";
System.out.println(text.indent(2)); // 为每行添加 2 个空格的缩进
```

transform() - 将字符串通过给定的函数转换为另一种对象

```
String original = "hello";
String result = original.transform(str -> str.toUpperCase());
System.out.println(result); // "HELLO"
```

describeConstable() 和 resolveConstantDesc() 这两个方法用于支持 JVM 常量 API。

3. Files.mismatch() 方法

Java 12 在 java.nio.file.Files 类中新增了 mismatch() 方法，用于高效比较两个文件的内容是否相同。

```
Path file1 = Path.of("file1.txt");
Path file2 = Path.of("file2.txt");

long mismatch = Files.mismatch(file1, file2);
if (mismatch == -1) {
```

```
    System.out.println("Files are identical");
} else {
    System.out.println("Files differ at byte " + mismatch);
}
```

4. Compact Number Formatting（紧凑数字格式）

Java 12 引入了紧凑数字格式（Compact Number Formatting），可以将数字格式化为人类可读的形式。

```
NumberFormat fmt = NumberFormat.getCompactNumberInstance(
    Locale.US, NumberFormat.Style.SHORT);
String result = fmt.format(1000); // "1K" for English locale
System.out.println(result);

NumberFormat fmt2 = NumberFormat.getCompactNumberInstance(
    Locale.US, NumberFormat.Style.LONG);
String result2 = fmt2.format(1000); // "1 thousand" for English locale
System.out.println(result2);
```

5. Collectors.teeing() 收集器

Java 12 增加了一个很有用的 Collector：teeing()，可以让流拆分成两个子流，各自收集后再合并结果。

```
DoubleSummaryStatistics stats = Stream.of(1.0, 2.0, 3.0, 4.0, 5.0)
    .collect(Collectors.teeing(
        Collectors.summingDouble(Double::doubleValue),
        Collectors.counting(),
        (sum, count) -> new DoubleSummaryStatistics(count, sum)
    ));
```

6. Shenandoah GC（实验性）

Java 12 正式引入了 Shenandoah 垃圾收集器（实验性），这是由 Red Hat 开发的低停顿垃圾收集器。

- 设计目标是实现低暂停时间
- 适用于需要低延迟的应用程序

7. G1 垃圾收集器优化

Java 12 对默认的 G1 垃圾收集器进行了改进：- G1 的可中断 mixed GC - 从 G1 立即返回未使用的已提交内存

8. Microbenchmark Harness（微基准测试套件）

Java 12 引入了一个基于 Java 的微基准测试框架（JEP 230），方便开发者进行性能测试。

9. JVM 常量 API

Java 12 提供了一套新的 API，用于在字节码中加载动态常量（JEP 309）。

10. 默认 CDS（Class-Data Sharing）归档

Java 12 实现了默认的 CDS 归档，减少了启动时间和内存占用。

预览和实验性功能

Switch 表达式

- 作为预览特性引入
- 提供了更简洁、安全的 Switch 语法

其他改进

Unicode 11 支持

Java 12 增加了对 Unicode 11 的支持。

instanceof 模式匹配（预览）

Java 12 引入了模式匹配的 instanceof（预览特性），简化了在进行类型检查后再强制转换的常见模式。

总结

Java 12 虽然不是一个长期支持版本，但仍带来了许多实用的改进和新功能。其中最值得注意的是 Switch 表达式的预览，它大大简化了 Switch 语句的语法。此外，字符串增强、Files.mismatch() 方法、紧凑数字格式等功能也为日常开发提供了更多便利。

Java 12 还为后续版本的开发奠定了基础，特别是在语法简化和 API 增强方面，为 Java 语言的演进做出了贡献。

Java 13 新特性总结

Java 13 是 Oracle 于 2019 年 9 月 17 日发布的版本，虽然不是长期支持（LTS）版本，但也带来了不少重要的新功能和改进。Java 13 继续沿用了每六个月发布

一个新版本的策略，并重点增强了性能和语言表达能力。

主要新特性

1. 文本块（Text Blocks，预览特性）

Java 13 引入了文本块作为预览特性，这是一个多行字符串文字，它避免了对大多数转义序列的需要，以可预测的方式自动格式化字符串，并在需要时让开发人员控制格式。

```
// 传统的多行字符串写法
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, world</p>\n" +
              "    </body>\n" +
              "</html>\n";

// Java 13 的文本块写法
String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
              """;
```

文本块使用三个双引号 (""") 作为分隔符，使得多行字符串的编写更加简洁和易读。

2. Switch 表达式增强（二次预览）

Java 13 对在 Java 12 中首次引入的 Switch 表达式进行了增强，引入了 `yield` 语句来返回值，而不是使用 `break`。`yield` 关键字用于从 switch 表达式的 case 块中返回一个值。

```
// 使用 yield 从 switch 表达式返回值
int i = switch (day) {
    case MONDAY, TUESDAY -> 1;
    case WEDNESDAY -> {
        System.out.println("Wednesday");
        yield 3; // 使用 yield 返回值
    }
    default -> 0;
};
```

3. 动态 CDS 归档（Dynamic CDS Archives）

Java 13 扩展了应用程序类-数据共享（AppCDS），以允许在 Java 应用程序执行结束时动态归档类。归档类将包括默认的基础层 CDS 存档中不存在的所有已加载的应用程序类和库类。

- 提升应用程序启动性能
- 减少内存占用
- 延续了在 Java 10 中引入的 AppCDS 功能

4. ZGC：释放未使用的内存（ZGC: Uncommit Unused Memory）

Java 13 增强了 ZGC，使其能够将未使用的堆内存返回给操作系统。ZGC 是一个可扩展的低延迟垃圾收集器，在 Java 13 中进一步优化了内存管理。

- 减少应用程序的内存占用
- 提高内存使用效率
- 适用于需要低延迟的大规模应用程序

5. 重新实现 Socket API（Reimplement the Legacy Socket API）

Java 13 使用更简单、更现代的实现替换了 java.net.Socket 和 java.net.ServerSocket API 使用的底层实现，以提高可维护性和调试性。

- 简化了 Socket API 的底层实现
- 提高了可维护性
- 便于调试和故障排除

预览和实验性功能

文本块（Text Blocks）

- 作为预览特性引入
- 提供了多行字符串字面量的简洁语法
- 自动处理字符串格式化

Switch 表达式（增强版）

- 在 Java 12 的基础上进行了增强
- 引入了 `yield` 语句用于返回值
- 继续作为预览特性提供

其他改进

性能优化

- 通过动态 CDS 归档提升启动性能

- 通过 ZGC 改进内存管理
- 通过 Socket API 重构提高可维护性

JVM 改进

- 更好的内存管理机制
- 优化的垃圾收集行为
- 改进的类加载机制

总结

Java 13 虽然不是长期支持版本，但仍然引入了一些非常有用的功能，特别是文本块和 Switch 表达式的增强，显著改善了 Java 代码的可读性和表达能力。动态 CDS 归档和 ZGC 的改进则进一步提升了性能表现。

Java 13 的发布延续了 Java 平台不断演进的趋势，通过预览特性的引入，允许开发者提前体验并反馈，确保新功能在正式发布前经过充分验证。文本块的引入使得处理多行字符串变得更加直观，而 Switch 表达式的改进则让条件逻辑的编写更加简洁。

Java 14 新特性总结

Java 14 是 Oracle 于 2020 年 3 月 17 日发布的版本，这个版本包含了 16 个新特性（JEP），是自 Java 采用六个月发布周期以来的重要版本之一。Java 14 在语言特性、JVM 改进和 API 增强等方面都有显著的提升。

主要新特性

1. Switch 表达式（正式特性）

经过两个版本的预览，增强型 switch 在 Java 14 中正式成为 Java 语言的一部分。Switch 表达式扩展了 switch 语句，使其不仅可以作为语句，还可以作为表达式，并且支持简化的“case L ->”模式匹配语法。

```
// 传统的 switch 语句
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
    default                     -> throw new IllegalStateException("Unexpected value: " + da
};

// 使用 yield 从复杂的 case 块返回值
int result = switch (day) {
```

```
    case MONDAY -> 0;
    case TUESDAY -> 1;
    default -> {
        int result = days.length;
        yield result;
    }
};
```

2. Record 类（预览特性）

Java 14 引入了一种全新的类型声明：记录类（Record），作为预览特性。Record 提供了一种紧凑的语法来声明类，主要用于创建不可变的数据载体类。

```
// 使用 Record 简化 POJO 类的定义
public record Person(String name, int age) {}

// 使用示例
Person person = new Person("John", 30);
System.out.println(person.name()); // 获取 name
System.out.println(person.age());  // 获取 age
```

3. instanceof 模式匹配（预览特性）

Java 14 中，instanceof 模式匹配作为预览特性再次出现，简化了在进行类型检查后再强制转换的常见模式。

```
// 传统的 instanceof 使用方式
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.toUpperCase());
}

// Java 14 的模式匹配
if (obj instanceof String s) {
    System.out.println(s.toUpperCase()); // 自动转换为 String 类型
}
```

4. 文本块（第二次预览）

文本块在 Java 14 中继续作为预览特性，并引入了新的转义序列。文本块提供了一种简洁的方式来表示多行字符串。

```
// 使用文本块定义多行字符串
String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
```

```
        </body>
    </html>
    """;
```

5. 改进的 NullPointerExceptions 提示信息

Java 14 对长期困扰开发者的 NullPointerException 进行了改进。当发生 NPE 时，错误信息现在会指出具体哪个变量为空。

```
// 改进后的 NullPointerException 会显示更详细的信息
// 如 "Exception in thread "main" java.lang.NullPointerException:
// Cannot assign field "name" because "person" is null"
```

JVM 相关改进

1. ZGC 扩展到 macOS 和 Windows

Java 14 将 ZGC 从仅支持 Linux 扩展到了 macOS 和 Windows 平台 (实验性)。

- 提供低延迟的垃圾收集
- 支持跨平台使用
- 适用于需要低延迟的大规模应用程序

2. G1 的 NUMA 感知内存分配

Java 14 对 G1 垃圾收集器进行了优化，支持 NUMA（Non-Uniform Memory Access）感知的内存分配，以提高多核系统的性能。

3. 移除 CMS 垃圾回收器

并发标记清除（CMS）垃圾收集器自 Java 9 起被弃用，终于在 Java 14 中被完全移除。

工具和 API 增强

1. jpackage 工具（孵化特性）

Java 14 提供了 jpackage 工具的早期版本，可将 Java 应用打包成原生安装包，支持创建独立的可执行文件。

2. 非易失性映射字节缓冲区

新增了特定 JDK 的文件映射模式，可以使用 FileChannel 创建引用非易失性存储器的 MappedByteBuffer。

3. JFR 事件流

Java Flight Recorder (JFR) 现在支持事件流，允许应用程序代码注册回调以接收实时事件。

已删除和弃用的功能

1. 移除 Pack200 工具和 API

Pack200 工具和相关 API 在 Java 14 中被完全移除。

2. 弃用 Solaris 和 SPARC 端口

由于缺乏足够的维护资源，Solaris 和 SPARC 端口在 Java 14 中被弃用。

3. 弃用 ParallelScavenge 和 SerialOld GC 组合

ParallelScavenge 和 SerialOld 垃圾收集器的组合使用在 Java 14 中被弃用。

预览和实验性功能

1. Record 类

- 作为预览特性引入
- 提供了一种简洁的方式来定义数据载体类
- 自动生成构造函数、访问器方法、equals、hashCode 和 toString 方法

2. instanceof 模式匹配

- 简化类型检查和转换过程
- 减少样板代码

3. 文本块（第二次预览）

- 进一步完善多行字符串的处理
- 提供更自然的字符串字面量语法

总结

Java 14 是一个内容相当丰富的版本，其中最引人注目的是 Switch 表达式正式成为标准特性，以及 Record 类和 instanceof 模式匹配的引入。这些语言层面的增强让 Java 变得更简洁和富有表达力，逐步摆脱”样板代码多”的诟病。

Java 14 还在性能和 JVM 改进方面做了很多工作，包括 ZGC 的跨平台支持、G1 的 NUMA 优化等。同时，移除一些老旧的 GC 和工具也表明了 Java 平台向前发展的决心。这些新特性的引入，标志着 Java 语言正在向更现代化的方向发展。

Java 15 新特性总结

Java 15 是 Oracle 于 2020 年 9 月 15 日发布的版本，虽然不是长期支持 (LTS) 版本，但带来了 14 个新功能，其中有不少是十分实用的特性。Java 15 在语言特性、JVM 改进和 API 增强等方面都有显著的提升。

主要新特性

1. 密封类（Sealed Classes，预览特性）

密封类允许开发者对继承进行更精细的控制，提供了一种更加精确地控制类继承的方法。类的设计者可以指定一个类能够被哪些类继承，增强了类的封装性和安全性。

```
// 使用 sealed 关键字限制继承
public sealed class Shape
    permits Circle, Square, Rectangle {
    // ...
}

final class Circle extends Shape {
    // ...
}

final class Square extends Shape {
    // ...
}

non-sealed class Rectangle extends Shape {
    // ...
}
```

2. 文本块（Text Blocks，正式特性）

经过两次预览，Text Blocks 在 Java 15 成为正式特性，使多行字符串处理不再繁琐。

```
// 使用文本块定义多行字符串
String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
              """;
```

3. Records（二次预览）

Records 特性在 Java 15 中继续作为预览特性，可以用来简化对象的创建和访问，自动生成 getter 和其他必要的方法。

```
// 使用 Records 简化数据载体类
public record Person(String name, int age) {
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

4. instanceof 模式匹配（二次预览）

instanceof 模式匹配在 Java 15 中继续作为预览特性，简化了类型检查和转换的过程。

```
// 使用模式匹配简化类型检查
if (obj instanceof String str) {
    System.out.println(str.toUpperCase()); // 自动转换为 String 类型
}
```

5. 隐藏类（Hidden Classes）

Java 15 引入了隐藏类（Hidden Classes）特性，可以用来隐藏类的实现细节。隐藏类不能被其他类直接引用，主要用于框架和库的内部实现。

```
// 隐藏类主要用于框架实现，普通开发者较少直接使用
// 但可以提高框架的性能和灵活性
```

JVM 相关改进

1. ZGC: 可扩展低延迟垃圾收集器（正式发布）

ZGC 在 Java 15 中正式发布，不再处于实验阶段，提供可扩展的低延迟垃圾收集功能。

- 停顿时间不超过 10 毫秒
- 支持 TB 级别的堆内存
- 适用于需要低延迟的大规模应用程序

2. Shenandoah GC（正式发布）

Shenandoah 垃圾收集器在 Java 15 中正式发布，提供低停顿时间的垃圾收集功能。

3. 禁用和废弃偏向锁（Biased Locking）

Java 15 开始禁用和废弃偏向锁（Biased Locking），这是为了未来的性能优化做准备。

4. 重新实现 DatagramSocket API

Java 15 重新实现了 DatagramSocket API，使其更加现代化和易于维护。

API 增强

1. 外部内存访问 API（第二个孵化器）

Java 15 继续提供外部内存访问 API 的孵化器版本，允许更安全地访问 JVM 外部内存。

2. 爱德华曲线算法（EdDSA）

Java 15 增加了一个新的密码学算法，爱德华曲线算法（EdDSA）签名算法。

已删除和移除的功能

1. 移除 Nashorn JavaScript 引擎

Java 15 正式移除了 Nashorn JavaScript 引擎，这是因为在 GraalVM 提供了更好的替代方案。

2. 删除 Solaris 和 SPARC 端口

由于缺乏足够的维护资源，Solaris 和 SPARC 端口在 Java 15 中被完全删除。

预览和孵化器功能

1. 密封类（Sealed Classes）

- 作为预览特性引入
- 提供更精确的类继承控制
- 增强类的封装性和安全性

2. Records（二次预览）

- 简化数据载体类的定义
- 自动生成构造函数、访问器方法、equals、hashCode 和 toString 方法

3. instanceof 模式匹配（二次预览）

- 简化类型检查和转换过程
- 减少样板代码

总结

Java 15 虽然是短期版本，但特性相当丰富。其中最引人注目的是文本块在这一版终于正式发布，使多行字符串处理不再繁琐。密封类的引入为开发者提供了更精确的继承控制机制，隐藏类为框架开发者提供了更灵活的实现方式。

Java 15 还在性能和 JVM 改进方面做了很多工作，包括 ZGC 和 Shenandoah GC 的正式发布，这为低延迟应用场景提供了更多选择。同时，移除 Nashorn JavaScript 引擎等决策也表明了 Java 平台向前发展的决心。

这些新特性的引入，标志着 Java 语言正在向更现代化、更安全和更高效的方向发展。

Java 16 新特性总结

Java 16 是 Oracle 于 2021 年 3 月 16 日发布的版本，虽然不是长期支持（LTS）版本，但作为下一个 LTS 版本 Java 17 的先行版本，带来了 17 个新特性，其中许多重要特性最终成为了正式标准。

主要新特性

1. Record 类（正式特性）

Java 16 将记录类从预览转为正式特性。Records 允许我们以一种简洁的方式定义一个类，我们只需要指定其数据内容。对于每个 Record 类，Java 都会自动地为其成员变量生成 equals, hashCode, toString 方法，以及所有字段的访问器方法（getter）。

```
// 使用 Records 定义数据载体类
public record Person(String name, int age) {
    // 编译器自动生成构造器、访问器、equals、hashCode 和 toString 方法
}

// 使用示例
Person person = new Person("John", 30);
```

```
System.out.println(person.name()); // 获取 name
System.out.println(person.age());  // 获取 age
```

2. instanceof 模式匹配（正式特性）

instanceof 模式匹配在 Java 16 中正式发布，简化了类型检查和转换的过程。

```
// 使用模式匹配简化类型检查
if (obj instanceof String str) {
    System.out.println(str.toUpperCase()); // 自动转换为 String 类型
}
```

3. 密封类（第二次预览）

密封类在 Java 16 中继续作为预览特性，允许开发者对继承进行更精细的控制。

```
// 使用 sealed 关键字限制继承
public sealed class Shape
    permits Circle, Square, Rectangle {
    // ...
}

final class Circle extends Shape {
    // ...
}

final class Square extends Shape {
    // ...
}

non-sealed class Rectangle extends Shape {
    // ...
}
```

4. jpackage 工具（正式特性）

jpackage 工具在 Java 16 中正式发布，不再是孵化器功能，允许将 Java 应用程序打包成原生安装包。

5. Unix 域套接字通道

SocketChannel 和 ServerSocketChannel 现在支持 Unix 域套接字，提供更高效的进程间通信。

JVM 相关改进

1. ZGC：并发线程栈处理

ZGC 通过将其线程堆栈处理从安全点移至并发阶段得到改进，减少了垃圾收集的停顿时间。

2. 弹性元空间（Elastic Metaspace）

通过将未使用的 HotSpot 类元数据或元空间内存快速返回给操作系统来改进元空间内存管理，减少元空间占用空间。

3. 启用 C++14 语言特性

C++ 14 功能可以在带有 JDK 16 的 C++ 源代码中使用，提高了 JDK 内部开发的灵活性。

API 增强

1. Vector API（孵化器）

引入了新的矢量 API，允许开发人员明确执行矢量操作，为在 Java 中进行高性能并行计算提供了基础。

2. Foreign Linker API（孵化器）

Java 代码可以由 C/C++ 调用，反之亦然，使用新的 API 替换 JNI，简化了与本地代码的交互。

3. 基于值的类的警告

如果使用 synchronize 同步基于值的类，则会引发警告，这是为了推动使用更适合的同步机制。

平台支持改进

1. Alpine Linux 端口

现在 JDK 可用于 Alpine Linux 和其他使用 musl 实现的 Linux 发行版。

2. Windows/AArch64 端口

现在 JDK 可以在 AArch64、ARM 硬件服务器或基于 ARM 的笔记本电脑上运行。

开发工具和环境改进

1. 从 Mercurial 迁移到 Git/GitHub

OpenJDK 源代码从 Mercurial 转移到 Git/GitHub，使开发流程更加现代化。

2. 强封装 JDK 内部 API

Java 16 开启了对内部 API 强封装的最后一步，默认情况下禁止访问内部 API。

预览和孵化器功能

1. 密封类（第二次预览）

- 作为预览特性继续提供
- 提供更精确的类继承控制
- 增强类的封装性和安全性

2. Vector API（孵化器）

- 提供矢量操作的能力
- 为高性能并行计算奠定基础

3. Foreign Linker API（孵化器）

- 提供与本地代码交互的新方式
- 简化了与 C/C++ 代码的集成

总结

Java 16 是一个重要的过渡版本，完成了若干预览特性的正式化（如 Records 和 instanceof 模式匹配），并推进了 JDK 内部重构。其中最显著的变化是 Record 类正式发布，这极大地简化了数据载体类的创建；instanceof 模式匹配也成为正式特性，使类型检查和转换更加简洁。

Java 16 还在性能和 JVM 改进方面做了很多工作，包括 ZGC 的改进、弹性元空间的引入等。同时，平台支持的扩展（如 Alpine Linux 和 Windows/AArch64）也表明了 Java 生态的持续扩展。

这些新特性的引入，标志着 Java 语言正在向更现代化、更简洁和更安全的方向发展，为即将到来的 Java 17 LTS 版本奠定了坚实的基础。

Java 17 新特性总结

Java 17 是 Oracle 于 2021 年 9 月 14 日发布的长期支持（LTS）版本，这也是继 Java 11 之后的又一个长期支持版本。根据 Oracle 的支持路线图，Java 17

将获得至少八年的支持，使其成为企业级应用的理想选择。

主要新特性

1. 密封类（Sealed Classes，正式特性）

Java 17 将密封类从预览特性转为正式特性。密封类和接口限制了哪些其他类或接口可以扩展或实现它们，提供了对类继承结构的更好控制。

```
// 使用 sealed 关键字限制继承
public sealed class Shape
    permits Circle, Rectangle, Square {
    // ...
}

final class Circle extends Shape {
    // ...
}

final class Rectangle extends Shape {
    // ...
}

non-sealed class Square extends Shape {
    // ...
}
```

2. Switch 模式匹配（预览特性）

Java 17 提供了 switch 的模式匹配作为预览特性，扩展了 switch 表达式和语句的模式匹配功能。

```
// 使用模式匹配的 switch 表达式
static String formatter(Object obj) {
    return switch (obj) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> obj.toString();
    };
}
```

3. 弃用安全管理器（Security Manager）

Java 17 标记安全管理器（Security Manager）为弃用，因为其复杂性和维护成本较高。

4. 永久性强封装 JDK 内部 API

Java 17 完成了对 JDK 内部 API 强封装的最后工作，默认情况下禁止访问内部 API。

API 增强

1. 伪随机数生成器（Pseudo-Random Number Generators）

Java 17 通过 JEP 356 引入了一系列新的随机数生成器接口和实现，提供了更好的性能和更灵活的选择。

```
// 使用新的随机数生成器接口
RandomGenerator rng = RandomGeneratorFactory.all().findFirst().get().create();

// 生成随机数
int randomInt = rng.nextInt();
```

2. Stream API 增强

Java 17 为 Stream API 添加了新的方法，如 toList()，用于简化流操作。

```
// 使用 Stream.toList() 方法
List<String> list = Stream.of("a", "b", "c")
    .filter(s -> s.length() > 0)
    .toList(); // 替代 collect(Collectors.toList())
```

3. 外部函数和内存 API（孵化特性）

Java 17 引入了外部函数和内存 API（孵化特性），用于替代 JNI，提供更安全和高效的本地代码互操作性。

JVM 相关改进

1. ZGC 改进

ZGC 在 Java 17 中继续得到改进，包括并发栈处理和其他性能优化。

2. 弃用 Applet API

Java 17 完全弃用了 Applet API，因为 Applet 技术已经过时。

3. macOS 渲染改进

针对 macOS 平台，Java 17 引入了基于 Metal 的 Java 2D 渲染管线，提高了图形性能。

语言和语法改进

1. Records 增强

Java 17 进一步增强了 Records 的功能，使其更适合数据载体类的定义。

```
// 使用 Records 定义数据类
public record Person(String name, int age) {
    public Person {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
    }
}
```

2. instanceof 模式匹配增强

Java 17 在已有 instanceof 模式匹配基础上进行了增强，使类型检查和转换更加简洁。

平台支持改进

1. 跨平台渲染管线

Java 17 改进了对不同操作系统的渲染支持，特别是在 macOS 上引入了基于 Metal 的渲染管线。

2. 增强的伪随机数生成器

Java 17 引入了更灵活和高性能的随机数生成器 API。

安全改进

1. 上下文特定的反序列化过滤器

Java 17 引入了上下文特定的反序列化过滤器，增强了反序列化操作的安全性。

已删除和弃用的功能

1. 弃用安全管理器

安全管理器被标记为弃用，计划在未来版本中移除。

2. 移除 Applet API

完全移除了已经过时的 Applet API。

预览和孵化功能

1. Switch 模式匹配（预览特性）

- 扩展了 switch 表达式和语句的模式匹配功能
- 提供了更强大的数据查询能力

2. 外部函数和内存 API（孵化特性）

- 提供了与本地代码互操作的新方式
- 替代了复杂的 JNI 机制

总结

Java 17 是继 Java 11 之后的又一个长期支持版本，也是”现代 Java”功能集的大成者。它巩固了近年来引入的重要语言特性，如密封类、Record、Pattern Matching 等，使其成为正式特性。

Java 17 的发布标志着 Java 平台的一个重要里程碑，它不仅带来了许多新的特性和改进，还为 Java 社区的未来发展奠定了坚实的基础。Java 17 的增强特性和性能改进将使 Java 在各个领域发挥更大的作用，满足现代应用的需求。

对于企业级应用来说，Java 17 提供了稳定性、安全性和性能的完美平衡，是升级到现代 Java 版本的理想选择。

Java 18 新特性总结

Java 18 是 Oracle 于 2022 年 3 月 22 日发布的版本，虽然不是长期支持（LTS）版本，但带来了 9 个新功能。Java 18 在平台与性能、安全与密码等方面都有显著的提升。

主要新特性

1. UTF-8 作为默认字符集（JEP 400）

Java 18 通过 JEP 400 将默认字符集统一为 UTF-8。这意味着所有的 Java API (例如 String、FileReader 和 FileWriter）将默认使用 UTF-8 编码进行读取和写入操作。

```
// 在 Java 18 之前，字符集依赖于平台
// 现在默认使用 UTF-8，提高了跨平台一致性
String text = "Hello, 世界!";
byte[] bytes = text.getBytes(); // 默认使用 UTF-8 编码
```

2. 简易 Web 服务器（JEP 408）

Java 18 引入了一个简单的 Web 服务器，通过命令行工具 jwebserver 可以启动一个只提供静态文件的最小网络服务器，主要用于原型设计、临时编码和测试目的。

```
# 启动简易 Web 服务器
jwebserver

# 或指定端口和目录
jwebserver -p 8080 -d /path/to/directory
```

3. Switch 模式匹配（第二次预览，JEP 420）

Java 18 继续预览 Switch 模式匹配功能，扩展了 switch 表达式和语句的模式匹配能力。

```
// 使用模式匹配的 switch 表达式
static String formatterPatternSwitch(Object obj) {
    return switch (obj) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> obj.toString();
    };
}
```

4. 代码片段标签 @snippet（JEP 413）

为 JavaDoc 的 Standard Doclet 引入了 @snippet 标签，简化了在 API 文档中嵌入示例源代码的难度。

```
/**
 * 计算两个整数的和
 * {@snippet :
 * int sum = add(2, 3);
 * System.out.println(sum); // 输出 5
 * }
 */
public static int add(int a, int b) {
    return a + b;
}
```

5. 使用方法句柄重新实现核心反射（JEP 416）

使用方法句柄重新实现核心反射，以提高性能和安全性。

API 增强

1. Vector API（第三次孵化，JEP 417）

Java 18 中 Vector API 进入第 3 轮孵化，提供了更高级别的 API 来表达向量计算，这些计算在支持的 CPU 架构上编译为相应的 SIMD 指令。

2. 外部函数与内存 API（第二次孵化，JEP 419）

Java 18 中外部函数与内存 API 进入第 2 轮孵化，提供了更安全和高效的方式来与 Java 之外的代码和数据进行互操作。

3. 互联网地址解析 SPI（JEP 418）

引入了互联网地址解析的服务提供商接口（SPI），允许应用程序使用不同的解析协议。

JVM 相关改进

1. 弃用终结器（JEP 421）

Java 18 通过 JEP 421 正式将终结器（finalization）标记为废弃，因为终结器存在很多已知的问题，通常不推荐使用。

2. 安全与密码改进

Java 18 在安全方面也做了一些更新，例如禁用了 TLS 1.0 和 1.1 协议默认支持、更严格的默认信任库等。

语言和语法改进

1. 模式匹配增强

Java 18 在已有模式匹配基础上进行了增强，使代码更加简洁和易读。

2. 密封类支持

继续支持密封类（Sealed Classes），允许类或接口的继承和实现被限制到特定的类或接口。

平台支持改进

1. 默认字符集统一

通过将 UTF-8 作为默认字符集，消除了长期以来跨平台编码不一致的问题，让 Java 更贴合互联网时代的数据交换标准。

2. 简易 Web 服务器

内置的简易 Web 服务器体现了 Java 对开发者体验的重视，即使是小工具也能发挥作用。

预览和孵化功能

1. Switch 模式匹配（第二次预览）

- 扩展了 switch 表达式和语句的模式匹配功能
- 提供了更强大的数据查询能力

2. Vector API（第三次孵化）

- 提供了向量计算的高级 API
- 利用 SIMD 指令提高性能

3. 外部函数与内存 API（第二次孵化）

- 提供与本地代码和数据的互操作
- 更安全和高效的替代 JNI

总结

Java 18 虽然不是长期支持版本，但其新特性对开发者有重要意义。其中最大的变化对普通开发者来说莫过于默认编码 UTF-8 和内置简易 Web 服务器。

UTF-8 默认消除了长期以来跨平台编码不一致的问题，让 Java 更贴合互联网时代的数据交换标准。jwebserver 则体现出 Java 对开发者体验的重视，即使是一个很小的工具，也能发挥作用。

在底层性能上，Java 18 继续推进 Panama 和 Vector 等项目，使 Java 在系统编程和高性能计算上更具竞争力。作为非 LTS 版本，Java 18 提供了一个让社区试水新功能的平台，其反馈将作用于后续的 Java 版本中。

Java 19 新特性总结

Java 19 是 Oracle 于 2022 年 9 月 20 日发布的版本，虽然不是长期支持 (LTS) 版本，但带来了 7 个重要的新特性。Java 19 引入了许多重要的预览和孵化特性，特别是虚拟线程和结构化并发，为 Java 的并发编程带来了革命性的改进。

主要新特性

1. 虚拟线程（Virtual Threads，预览特性）

Java 19 引入了虚拟线程作为预览特性，这是最受期待的特性之一。虚拟线程是由 JVM 管理的轻量级线程，实现了”纤程”的概念，每个虚拟线程由多个虚拟线程映射到一个 OS 线程执行，调度由 JVM 负责。

```
// 传统的平台线程创建方式
try (ExecutorService executor = Executors.newFixedThreadPool(100)) {
    for (int i = 0; i < 10000; i++) {
        executor.submit(() -> {
            // 执行任务
        });
    }
}

// 使用虚拟线程的方式
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 10000; i++) {
        Thread.startVirtualThread(() -> {
            // 执行任务
            // 虚拟线程成本极低，可以轻松创建大量线程
        });
    }
}
```

2. 结构化并发（Structured Concurrency，孵化特性）

结构化并发是一种简化多线程编程的高级 API，它提供了一种更安全和可维护的方式来处理多线程任务。

```
import java.util.concurrent.StructuredTaskScope;

// 使用结构化并发处理多个异步任务
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    var userTask = scope.fork(() -> findUser(userId));
    var orderTask = scope.fork(() -> findOrder(userId));

    scope.joinUntil(Instant.now().plusSeconds(5));
    scope.throwIfFailed();

    User user = userTask.resultNow();
    Order order = orderTask.resultNow();

    return new UserProfile(user, order);
}
```

3. 记录模式（Record Patterns，预览特性）

记录模式扩展了模式匹配的能力，允许在模式匹配中使用记录类型，使解构记录类的组件变得更加简洁。

```
// 使用记录模式进行模式匹配
static String toString(Object obj) {
    return switch (obj) {
        case Point(int x, int y) -> x + ", " + y;
        case null -> "null";
        default -> obj.toString();
    };
}

// 记录类定义
record Point(int x, int y) {}
```

4. Switch 模式匹配（预览特性）

Java 19 进一步增强了 switch 表达式的模式匹配功能，提供了更强大的数据查询能力。

```
// 使用模式匹配的 switch 表达式
static String formatterPatternSwitch(Object obj) {
    return switch (obj) {
        case String s when s.length() > 5 -> s.toUpperCase();
        case Point(int x, int y) -> x + ", " + y;
        case null -> "null";
        default -> obj.toString();
    };
}
```

5. 外部函数与内存 API（预览特性）

Java 19 引入了外部函数与内存 API 作为预览特性，提供了更安全和高效的方式来与 Java 之外的代码和数据进行互操作。

```
import java.lang.foreign.*;

// 使用外部函数与内存 API 调用本地函数
try (Arena arena = Arena.ofConfined()) {
    SymbolLookup libm = SymbolLookup.libraryLookup("libm.so.6", arena);
    // 查找并调用本地函数
    var cosHandle = libm.lookup("cos").orElseThrow();
    // 调用函数
}
```

API 增强

1. 向量 API（第四次孵化，JEP 426）

向量 API 进入第四次孵化，提供了更高级别的 API 来表达向量计算，这些计算在支持的 CPU 架构上编译为相应的 SIMD 指令。

2. Unicode 14.0 支持

Java 19 增加了对 Unicode 14.0 的支持。

平台支持改进

1. Linux/RISC-V 移植（JEP 422）

Java 19 增加了对 Linux/RISC-V 平台的支持，扩展了 Java 的硬件平台兼容性。

安全改进

1. 改进的 TLS 支持

Java 19 引入了对 TLS 1.3 的改进支持，提供了更强的安全性和更快的加密速度。

2. 禁用过时的加密算法

Java 19 禁用了一些过时的加密算法，防止安全漏洞和攻击。

3. 加强的证书验证

Java 19 加强了对证书的验证，以确保 Java 应用程序只与受信任的实体通信。

预览和孵化功能

1. 虚拟线程（预览特性）

- 提供轻量级线程实现
- 大幅提高并发性能
- 降低并发编程复杂性

2. 记录模式（预览特性）

- 扩展模式匹配能力
- 简化记录类组件的解构

3. Switch 模式匹配（预览特性）

- 增强 switch 表达式的模式匹配功能
- 提供更强大的数据查询能力

4. 外部函数与内存 API（预览特性）

- 提供与本地代码和数据的安全互操作
- 替代复杂的 JNI 机制

5. 结构化并发（孵化特性）

- 简化多线程编程
- 提供更安全和可维护的并发模型

6. 向量 API（孵化特性）

- 提供向量计算的高级 API
- 利用 SIMD 指令提高性能

总结

Java 19 虽然不是长期支持版本，但引入了许多重要的预览和孵化特性，特别是虚拟线程和结构化并发，为 Java 的并发编程带来了革命性的改进。

虚拟线程的引入使得 Java 应用程序能够轻松创建数十万计的线程而不会像平台线程那样耗尽资源，这将极大地提高高并发应用程序的性能和可伸缩性。结构化并发则提供了更安全和可维护的方式来处理多线程任务。

Java 19 的发布继续推进了 Java 语言的现代化，通过预览和孵化特性让开发者有机会尝试和反馈新功能，为后续版本的正式发布奠定了基础。

Java 20 新特性总结

Java 20 是 Oracle 于 2023 年 3 月 21 日发布的版本，虽然不是长期支持（LTS）版本，但继续推进了 Java 语言在并发、模式匹配和本地互操作等方面的改进。Java 20 主要是对 Java 19 中引入的预览和孵化特性进行进一步改进和完善，为即将到来的 Java 21 LTS 版本做准备。

主要新特性

1. 虚拟线程（第二次预览，JEP 436）

Java 20 继续预览虚拟线程特性，这是 Project Loom 的核心部分。虚拟线程是由 JVM 管理的轻量级线程，每个虚拟线程映射到一个平台线程。

```
// 使用虚拟线程执行大量并发任务
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000)
        .forEach(i -> executor.submit(() -> {
            // 执行长时间运行的任务
            try {
                Thread.sleep(Duration.ofMillis(100));
                System.out.println("Task " + i + " completed");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }));
}
```

2. 结构化并发（第二次孵化，JEP 437）

Java 20 将结构化并发 API 进行第二轮孵化，提供了更安全和可维护的方式来处理多线程任务。

```
import java.util.concurrent.StructuredTaskScope;

// 使用结构化并发处理多个异步任务
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    var userTask = scope.fork(() -> findUser(userId));
    var orderTask = scope.fork(() -> findOrder(userId));

    scope.join(); // 等待所有任务完成
    scope.throwIfFailed(); // 检查是否有失败的任务

    User user = userTask.resultNow();
    Order order = orderTask.resultNow();

    return new UserProfile(user, order);
}
```

3. 作用域值（孵化特性，JEP 429）

Java 20 引入了作用域值（Scoped Values）孵化特性，允许在线程内和线程间共享不可变数据。

```
import java.lang.ScopedValue;

// 使用作用域值传递上下文信息
static final ScopedValue<Carrier> CARRIER = ScopedValue.newInstance();

// 在作用域中绑定值
```

```
ScopedValue.runWhere(CARRIER, carrier, () -> {
    // 在此作用域内可以访问 CARRIER.get()
    processRequest();
});

// 在方法中访问作用域值
void processRequest() {
    Carrier carrier = CARRIER.get(); // 获取作用域值
    // 使用 carrier 处理请求
}
```

4. 记录模式（第二次预览，JEP 432）

Java 20 对记录模式进行第二次预览，进一步改进了对泛型记录模式的支持，并支持在增强 for 语句中使用记录模式。

```
// 使用记录模式进行解构
static String displayCenter(Object obj) {
    return switch (obj) {
        case Point(int x, int y) ->
            "Point at (%d, %d)".formatted(x, y);
        case Rectangle(Point ul, Point lr) ->
            "Rectangle with center at (%d, %d)"
                .formatted((ul.x() + lr.x()) / 2, (ul.y() + lr.y()) / 2);
        default -> "Unknown shape";
    };
}

// 在增强 for 循环中使用记录模式
void printPoints(Point[] points) {
    for (Point(var x, var y) : points) {
        System.out.println(x + ", " + y);
    }
}
```

5. Switch 模式匹配（第四次预览，JEP 433）

Java 20 的 switch 模式匹配进入第 4 次预览，进一步完善了功能。

```
// 使用模式匹配的 switch 表达式
static double calculateArea(Shape shape) {
    return switch (shape) {
        case Circle(var radius) -> Math.PI * radius * radius;
        case Rectangle(var width, var height) -> width * height;
        case Triangle(var base, var height) -> 0.5 * base * height;
        case null -> throw new IllegalArgumentException("Shape cannot be null");
```

```
    };
}
```

6. 外部函数与内存 API（第二次预览，JEP 434）

Java 20 将外部函数与内存 API 进行第二次预览，提供了与本地代码和数据的安全互操作。

```
import java.lang.foreign.*;

// 使用外部函数与内存 API 调用本地函数
try (Arena arena = Arena.ofConfined()) {
    // 查找并调用本地函数
    SymbolLookup libm = SymbolLookup.loaderLookup();
    var cosHandle = libm.find("cos").orElseThrow();

    // 调用本地函数
    double result = (double) cosHandle.invoke(new Object[]{Math.PI});
}
```

API 增强

1. 向量 API（第五次孵化，JEP 438）

向量 API 进入第五次孵化，继续提供高级 API 来表达向量计算，这些计算在支持的 CPU 架构上编译为相应的 SIMD 指令。

项目改进

1. Project Loom 持续改进

- 虚拟线程的进一步完善
- 结构化并发的改进
- 作用域值的引入

2. Project Amber（模式匹配）

- 记录模式的持续改进
- Switch 模式匹配的完善

3. Project Panama（本地互操作）

- 外部函数与内存 API 的改进

预览和孵化功能

1. 虚拟线程（第二次预览）

- 轻量级线程实现
- 提高高并发应用程序的性能

2. 记录模式（第二次预览）

- 扩展模式匹配能力
- 简化记录类组件的解构

3. Switch 模式匹配（第四次预览）

- 增强 switch 表达式的模式匹配功能
- 提供更强大的数据查询能力

4. 外部函数与内存 API（第二次预览）

- 提供与本地代码和数据的安全互操作
- 替代复杂的 JNI 机制

5. 结构化并发（第二次孵化）

- 简化多线程编程
- 提供更安全和可维护的并发模型

6. 作用域值（孵化特性）

- 在线程内和跨线程共享不可变数据
- 支持虚拟线程的上下文传递

总结

Java 20 在功能上和 Java 19 一脉相承，并无全新重量级特性亮相，但却将之前的创新推进到了最后阶段。虚拟线程更完善、结构化并发和作用域值为简化并发提供了全新思路；记录模式和 switch 模式几乎打磨成熟，为模式匹配全面落地做好准备。

Java 20 是通往 Java 21 LTS 版本的重要里程碑，它延续了 Loom、Amber、Panama 三大项目的创新，为开发者提供了更好的编码体验。对于期待 LTS 版本的开发者来说，Java 20 显得相对平稳，但这正是为了在 Java 21 中释放更强大的功能所做的准备。

Java 21 新特性总结

Java 21 是 Oracle 于 2023 年 9 月 19 日发布的长期支持（LTS）版本，这是继 Java 8、Java 11、Java 17 之后的第 5 个 LTS 版本。Java 21 作为里程碑式的发布，融合了过去几版的预览特性，带来了约 15 项新特性，使 Java 在语法、并发、性能等方面迈上新台阶。

主要新特性

1. 虚拟线程（Virtual Threads，正式特性）

虚拟线程是 Project Loom 的核心成果，现已正式发布。虚拟线程是由 JVM 管理的轻量级线程，可显著提高高并发应用程序的性能和可伸缩性。

```
// 使用虚拟线程执行大量并发任务
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000)
        .forEach(i -> executor.submit(() -> {
            // 执行长时间运行的任务
            try {
                Thread.sleep(Duration.ofMillis(100));
                System.out.println("Task " + i + " completed");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }));
}
```

2. 记录模式（Record Patterns，正式特性）

记录模式正式发布，扩展了模式匹配的能力，使解构记录类的组件变得更加简洁。

```
// 使用记录模式进行解构
static String displayCenter(Object obj) {
    return switch (obj) {
        case Point(int x, int y) ->
            "Point at (%d, %d)".formatted(x, y);
        case Rectangle(Point ul, Point lr) ->
            "Rectangle with center at (%d, %d)"
                .formatted((ul.x() + lr.x()) / 2, (ul.y() + lr.y()) / 2);
        default -> "Unknown shape";
    };
}

// 记录类定义
```

```
record Point(int x, int y) {}
record Rectangle(Point upperLeft, Point lowerRight) {}
```

3. Switch 模式匹配（Pattern Matching for Switch，正式特性）

Switch 模式匹配正式发布，提供了更强大的类型匹配和数据提取能力。

```
// 使用模式匹配的 switch 表达式
static double calculateArea(Shape shape) {
    return switch (shape) {
        case Circle(var radius) -> Math.PI * radius * radius;
        case Rectangle(var width, var height) -> width * height;
        case Triangle(var base, var height) -> 0.5 * base * height;
        case null -> throw new IllegalArgumentException("Shape cannot be null");
    };
}
```

4. 未命名模式和变量（Unnamed Patterns and Variables，正式特性）

Java 21 引入了未命名模式和变量，允许使用 _ 作为未使用的模式变量名，提升代码简洁性。

```
// 使用未命名变量
static boolean isValid(Object obj) {
    return switch (obj) {
        case Point(_, _) -> true;  // 忽略 x 和 y 坐标
        case null -> false;
        default -> false;
    };
}
```

5. 序列化集合（Sequenced Collections，正式特性）

Java 21 引入了序列化集合 API，提供了一个统一的接口来处理有序集合。

```
import java.util.SequencedCollection;
import java.util.SequencedMap;
import java.util.ArrayList;

// 使用序列化集合
List<String> list = new ArrayList<>();
list.addFirst("first");  // 添加到开头
list.addLast("last");    // 添加到结尾
String first = list.getFirst();  // 获取第一个元素
String last = list.getLast();    // 获取最后一个元素
```

API 增强

1. 字符串模板（String Templates，预览特性）

Java 21 引入了字符串模板预览特性，提供更安全和更易读的字符串拼接方式。

```
// 使用字符串模板
Point p = new Point(10, 20);
String message = STR."The point is at (\{p.x()}, \{p.y()})";  // 需启用预览

// 字符串模板处理器
String sql = STR."SELECT * FROM users WHERE id = \{userId} AND active = \{true}";
```

2. 作用域值（Scoped Values，正式特性）

作用域值提供了一种安全地在线程内和线程间共享不可变数据的方式。

```
import java.lang.ScopedValue;

// 使用作用域值传递上下文信息
static final ScopedValue<Carrier> CARRIER = ScopedValue.newInstance();

// 在作用域中绑定值
ScopedValue.runWhere(CARRIER, carrier, () -> {
    // 在此作用域内可以访问 CARRIER.get()
    processRequest();
});

// 在方法中访问作用域值
void processRequest() {
    Carrier carrier = CARRIER.get(); // 获取作用域值
    // 使用 carrier 处理请求
}
```

3. 结构化并发（Structured Concurrency，正式特性）

结构化并发正式发布，提供了一种更安全和可维护的方式来处理多线程任务。

```
import java.util.concurrent.StructuredTaskScope;

// 使用结构化并发处理多个异步任务
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    var userTask = scope.fork(() -> findUser(userId));
    var orderTask = scope.fork(() -> findOrder(userId));

    scope.join(); // 等待所有任务完成
    scope.throwIfFailed(); // 检查是否有失败的任务
```

```
    User user = userTask.resultNow();
    Order order = orderTask.resultNow();

    return new UserProfile(user, order);
}
```

JVM 和性能改进

1. 分代 ZGC（Generational ZGC）

Java 21 引入了分代 ZGC，通过将堆分为年轻代和老年代来减少垃圾收集的停顿时间。

2. 序列化 API

Java 21 引入了序列化集合 API，提供统一的接口来处理有序集合。

3. 更详细的 NullPointerExceptions

Java 21 对 NullPointerException 进行了改进，提供更详细的错误信息，包括异常的来源和路径。

语言和语法改进

1. 模式匹配的进一步增强

- 记录模式的正式发布
- Switch 模式匹配的完善
- 未命名模式和变量的引入

2. 集合 API 的统一

- SequencedCollection 接口的引入
- 统一的首尾元素操作方法

预览特性

1. 字符串模板（String Templates）

- 提供更安全的字符串拼接
- 支持表达式嵌入和自定义处理器

平台支持改进

1. Unicode 15 支持

Java 21 增加了对 Unicode 15 的支持。

2. 更好的性能和稳定性

- 虚拟线程带来的并发性能提升
- 分代 ZGC 的引入
- 更好的内存管理

总结

Java 21 作为最新的长期支持版本，是 Java 语言发展历程中的一个重要里程碑。它不仅固化了多年来预览的特性（如虚拟线程、记录模式、Switch 模式匹配等），还引入了新的功能（如序列化集合、作用域值等）。

虚拟线程的正式发布将彻底改变 Java 的并发编程模型，使得编写高并发应用变得更加容易；记录模式和 Switch 模式匹配的正式发布进一步简化了数据处理代码；分代 ZGC 的引入则显著提升了垃圾收集的性能。

对于企业和开发者来说，Java 21 提供了更好的性能、更高的生产力和更强的安全性，是升级到现代 Java 版本的理想选择。Java 21 的发布标志着 Java 在并发编程、性能优化和代码简洁性方面达到了新的高度。

jpackage 详解

jpackage 是 Java 16 中正式发布的工具，用于将 Java 应用程序打包成原生安装包。它最初作为孵化器功能引入，在 Java 16 中成为标准特性。

概述

jpackage 是一个命令行工具，它可以将 Java 应用程序及其依赖项（包括 JRE）打包成原生的安装包或应用程序包。这使得 Java 应用程序可以像原生应用程序一样安装和运行，无需用户单独安装 JRE。

主要功能

1. 创建原生安装包

- 为不同操作系统创建适当的安装包格式
- Windows: MSI 或 EXE
- macOS: PKG 或 DMG
- Linux: RPM 或 DEB

2. 打包 JRE

- 自动包含最小化的 JRE，无需用户预先安装
- 减少应用程序的依赖复杂性

3. 创建可执行文件

- 生成原生可执行文件
- 提供更自然的用户体验

使用示例

基本用法

```
# 创建简单的应用程序包
jpackage --input input_directory --name MyApp --app-version 1.0 --module com.example.mymodul
```

详细参数示例

```
# 完整的打包命令示例
jpackage \
    --input dist \                      # 包含 JAR 文件的输入目录
    --name MyApplication \              # 应用程序名称
    --app-version 1.0.0 \              # 应用程序版本
    --icon icon.png \                   # 应用程序图标
    --main-class com.example.Main \     # 主类
    --main-jar myapp.jar \              # 主 JAR 文件
    --java-options "-Xmx2g" \           # JVM 选项
    --description "My Application" \    # 应用程序描述
    --vendor "My Company"               # 供应商名称
```

平台特定选项

Windows 选项

```
# Windows 特定选项
jpackage \
    --input dist \
    --name MyApp \
    --win-console                  # 显示控制台窗口
    --win-dir-chooser             # 显示目录选择对话框
    --win-menu                    # 创建开始菜单项
    --win-menu-group "My Apps"    # 菜单组名称
    --win-per-user-install        # 每用户安装
    --win-shortcut                # 创建桌面快捷方式
```

macOS 选项

```
# macOS 特定选项
jpackage \
    --input dist \
    --name MyApp \
    --mac-package-name "My App"           # 包名称
    --mac-package-identifier com.mycompany.myapp  # 包标识符
    --mac-signing-key-user-name "Developer ID Application: Name"  # 签名密钥
    --mac-app-store-compliant             # 符合 App Store 规范
```

Linux 选项

```
# Linux 特定选项
jpackage \
    --input dist \
    --name MyApp \
    --linux-app-category Game       # 应用程序类别
    --linux-deb-maintainer "maintainer@example.com"  # DEB 维护者
    --linux-menu-group "Games"      # 菜单组
    --linux-shortcut                # 创建快捷方式
```

支持的输出格式

Windows

- `.exe` - 可执行安装程序
- `.msi` - Windows Installer 包

macOS

- `.pkg` - Apple Installer 包
- `.dmg` - 磁盘镜像

Linux

- `.deb` - Debian 包
- `.rpm` - Red Hat 包

优势

1. 用户体验

- 无需预先安装 JRE
- 原生安装体验
- 简化应用程序分发

2. 开发者友好

- 简化的部署流程
- 减少兼容性问题
- 自动处理依赖关系

3. 安全性

- 包含固定版本的 JRE
- 避免系统 JRE 变化的影响

限制和注意事项

1. 平台依赖

- 需要在目标平台上构建
- 无法跨平台构建安装包

2. 大小考虑

- 包含 JRE 可能增加包大小
- 需要权衡功能和大小

3. 签名要求

- macOS 和 Windows 可能需要代码签名
- 对于分发到应用商店尤其重要

与其他工具的关系

与 JavaFX Packager 的关系

- jpackage 替代了 JavaFX Packager
- 提供更广泛的功能和平台支持

与 JLink 的关系

- jpackage 可以与 jlink 结合使用
- jlink 用于创建自定义运行时镜像
- jpackage 用于创建原生安装包

最佳实践

1. 应用程序准备

- 确保应用程序具有清晰的主类

- 测试在不同环境下的运行

2. 图标和资源

- 提供适当大小的应用程序图标
- 确保资源文件路径正确

3. 版本管理

- 使用有意义的版本号
- 考虑自动更新机制

未来发展方向

jpackage 的引入标志着 Java 应用程序分发的重大进步，它解决了长期以来 Java 应用程序部署复杂的问题。随着 Java 平台的发展，jpackage 也在不断完善，未来可能会支持更多平台特性和更灵活的配置选项。

总结

jpackage 是 Java 16 中一个重要的新功能，它简化了 Java 应用程序的部署和分发过程。通过将应用程序及其运行时环境打包成原生格式，jpackage 提供了更好的用户体验和更简单的分发机制。对于希望提供专业级 Java 应用程序的开发者来说，jpackage 是一个不可或缺的工具。