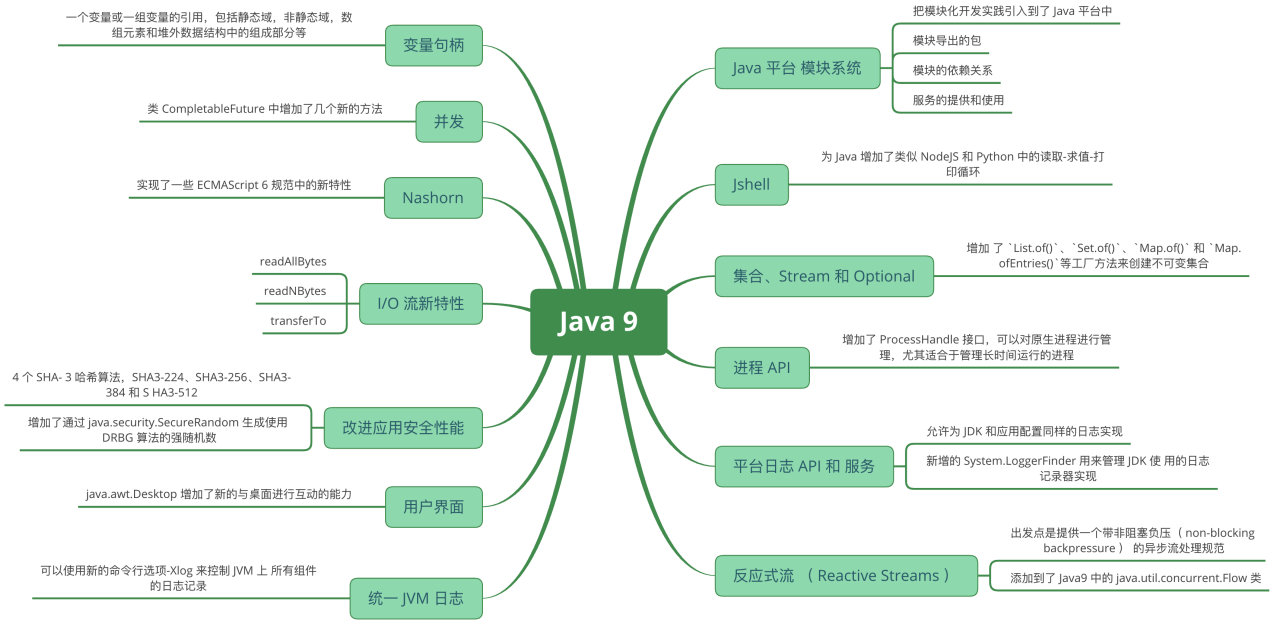
Java 9 新特性概述

Java 9 正式发布于 2017 年 9 月 21 日。作为 Java8 之后 3 年半才发布的新版本，Java 9 带来了很多重大的变化。其中最重要的改动是 Java 平台模块系统的引入。除此之外，还有一些新的特性。

知识体系



Java 平台 模块系统

Java 平台模块系统，也就是 Project Jigsaw，把模块化开发实践引入到了 Java 平台中。在引入了模块系统之后，JDK 被重新组织成 94 个模块。Java 应用可以通过新增的 jlink 工具，创建出只包含所依赖的 JDK 模块的自定义运行时镜像。这样可以极大的减少 Java 运行时环境的大小。这对于目前流行的不可变基础设施的实践来说，镜像的大小的减少可以节省很多存储空间和带宽资源 。

模块化开发的实践在软件开发领域并不是一个新的概念。Java 开发社区已经使用这样的模块化实践有相当长的一段时间。主流的构建工具，包括 Apache Maven 和 Gradle 都支持把一个大的项目划分成若干个子项目。子项目之间通过不同的依赖关系组织在一起。每个子项目在构建之后都会产生对应的 JAR 文件。 在 Java9 中 ， 已有的这些项目可以很容易的升级转换为 Java 9 模块 ， 并保持原有的组织结构不变。

Java 9 模块的重要特征是在其工件（artifact）的根目录中包含了一个描述模块的 module-info.class 文 件。 工件的格式可以是传统的 JAR 文件或是 Java 9 新增的 JMOD 文件。这个文件由根目录中的源代码文件 module-info.java 编译而来。该模块声明文件可以描述模块的不同特征。模块声明文件中可以包含的内容如下：

- **模块导出的包**：使用 exports 可以声明模块对其他模块所导出的包。包中的 public 和 protected 类型，以及这些类型的 public 和 protected 成员可以被其他模块所访问。没有声明为导出的包相当于模块中的私有成员，不能被其他模块使用。

- **模块的依赖关系**：使用 requires 可以声明模块对其他模块的依赖关系。使用 requires transitive 可 以把一个模块依赖声明为传递的。传递的模块依赖可以被依赖当前模块的其他模块所读取。 如果一个模块所导出的类型的型构中包含了来自它所依赖的模块的类型，那么对该模块的依赖应该声明为传递的。
- **服务的提供和使用**：如果一个模块中包含了可以被 ServiceLocator 发现的服务接口的实现 ， 需要使用 provides with 语句来声明具体的实现类 ； 如果一个模块需要使用服务接口，可以使用 uses 语句来声明。

如下代码中给出了一个模块声明文件的示例。在该声明文件中，模块 `com.mycompany.sample` 导出了 Java 包 `com.mycompany.sample` 。 该 模 块 依 赖 于 模 块 `com.mycompany.common` 。 该 模 块 也 提 供 了 服 务 接 口 `com.mycompany.common.DemoService` 的实现类 `com.mycompany.sample.DemoServiceImpl` 。

```
module com.mycompany.sample {
    exports com.mycompany.sample;
    requires com.mycompany.common;
    provides com.mycompany.common.DemoService with
        com.mycompany.sample.DemoServiceImpl;
}
```

模块系统中增加了模块路径的概念。模块系统在解析模块时，会从模块路径中进行查找。为了保持与之前 Java 版本的兼容性，CLASSPATH 依然被保留。所有的类型在运行时都属于某个特定的模块。对于从 CLASSPATH 中加载的类型，它们属于加载它们的类加载器对应的未命名模块。可以通过 Class 的 getModule()方法来获取到表示其所在模块的 Module 对象。

在 JVM 启动时，会从应用的根模块开始，根据依赖关系递归的进行解析，直到得到一个表示依赖关系的图。如果解析过程中出现找不到模块的情况，或是在模块路径的同一个地方找到了名称相同的模块，模块解析过程会终止，JVM 也会退出。Java 也提供了相应的 API 与模块系统进行交互。

Jshell

jshell 是 Java 9 新增的一个实用工具。jshell 为 Java 增加了类似 NodeJS 和 Python 中的读取-求值-打印循环（ Read-Evaluation-Print Loop ） 。 在 jshell 中 可以直接 输入表达式并查看其执行结果。当需要测试一个方法的运行效果，或是快速的对表达式进行求值时，jshell 都非常实用。只需要通过 jshell 命令启动 jshell，然后直接输入表达式即可。每个表达式的结果会被自动保存下来 ， 以数字编号作为引用，类似 $1 和$2 这样的名称 。可以在后续的表达式中引用之前语句的运行结果。 在 jshell 中 ， 除了表达式之外，还可以创建 Java 类和方法。jshell 也有基本的代码完成功能。

在 如下代码 中，我们直接创建了一个方法 add。

```
jshell> int add(int x, int y) {
    ...> return x + y;
    ...> }
 | created method add(int,int)
```

接着就可以在 jshell 中直接使用这个方法，如下代码 所示。

```
jshell> add(1, 2)
$19 ==> 3
```

集合、Stream 和 Optional

在集合上，Java 9 增加 了 `List.of()`、`Set.of()`、`Map.of()` 和 `Map.ofEntries()`等工厂方法来创建不可变集合 ， 如 如下 所示。

```
List.of();
List.of("Hello", "World");
List.of(1, 2, 3);
Set.of();
Set.of("Hello", "World");
Set.of(1, 2, 3);
Map.of();
Map.of("Hello", 1, "World", 2);
```

Stream 中增加了新的方法 ofNullable、dropWhile、takeWhile 和 iterate。在 如下代码 中，流中包含了从 1 到 5 的 元素。断言检查元素是否为奇数。第一个元素 1 被删除，结果流中包含 4 个元素。

```
@Test
public void testDropWhile() throws Exception {
    final long count = Stream.of(1, 2, 3, 4, 5)
        .dropWhile(i -> i % 2 != 0)
        .count();
    assertEquals(4, count);
}
```

Collectors 中增加了新的方法 filtering 和 flatMapping。在 如下代码 中，对于输入的 String 流 ， 先通过 flatMapping 把 String 映射成 Integer 流 ， 再把所有的 Integer 收集到一个集合中。

```
@Test
public void testFlatMapping() throws Exception {
    final Set<Integer> result = Stream.of("a", "ab", "abc")
        .collect(Collectors.flatMapping(v -> v.chars().boxed(),
            Collectors.toSet()));
    assertEquals(3, result.size());
}
```

Optional 类中新增了 ifPresentOrElse、or 和 stream 等方法。在 如下代码 中，Optiona l 流中包含 3 个 元素，其中只有 2 个有值。在使用 flatMap 之后，结果流中包含了 2 个值。

```
@Test
public void testStream() throws Exception {
    final long count = Stream.of(
        Optional.of(1),
        Optional.empty(),
        Optional.of(2)
    ).flatMap(Optional::stream)
        .count();
    assertEquals(2, count);
}
```

进程 API

Java 9 增加了 ProcessHandle 接口，可以对原生进程进行管理，尤其适合于管理长时间运行的进程。在使用 ProcessBuilder 来启动一个进程之后，可以通过 Process.toHandle()方法来得到一个 ProcessHandl e 对象的实例。通过 ProcessHandle 可以获取到由 ProcessHandle.Info 表 示的进程的基本信息，如命令行参数、可执行文件路径和启动时间等。ProcessHandle 的 onExit()方法返回一个 CompletableFuture对象，可以在进程结束时执行自定义的动作。 如下代码中给出了进程 API 的使用示例。

```
final ProcessBuilder processBuilder = new ProcessBuilder("top")
    .inheritIO();
final ProcessHandle processHandle = processBuilder.start().toHandle();
processHandle.onExit().whenCompleteAsync((handle, throwable) -> {
    if (throwable == null) {
        System.out.println(handle.pid());
    } else {
        throwable.printStackTrace();
    }
});
```

平台日志 API 和 服务

Java 9 允许为 JDK 和应用配置同样的日志实现。新增的 System.LoggerFinder 用来管理 JDK 使 用的日志记录器实现。JVM 在运行时只有一个系统范围的 LoggerFinder 实例。LoggerFinder 通 过服务查找机制来加载日志记录器实现。默认情况下，JDK 使用 java.logging 模块中的 java.util.logging 实现。通过 LoggerFinder 的 getLogger()方法就可以获取到表示日志记录器的 System.Logger 实现。应用同样可以使用 System.Logger 来记录日志。这样就保证了 JDK 和应用使用同样的日志实现。我们也可以通过添加自己的 System.LoggerFinder 实现来让 JDK 和应用使用 SLF4J 等其他日志记录框架。 代码清单 9 中给出了平台日志 API 的使用示例。

```
public class Main {
    private static final System.Logger LOGGER = System.getLogger("Main");
    public static void main(final String[] args) {
        LOGGER.log(Level.INFO, "Run!");
    }
}
```

反应式流 （ Reactive Streams ）

反应式编程的思想最近得到了广泛的流行。 在 Java 平台上有流行的反应式 库 RxJava 和 R eactor。反应式流规范的出发点是提供一个带非阻塞负压 （ non-blocking backpressure ） 的异步流处理规范。反应式流规范的核心接口已经添加到了 Java9 中的 java.util.concurrent.Flow 类中。

Flow 中包含了 Flow.Publisher、 Flow.Subscriber、 Flow.Subscription 和 F low.Processor 等 4 个核心接口。Java 9 还提供了 SubmissionPublisher 作为 Flow.Publisher 的一个实现。RxJava 2 和 Reactor 都可以很方便的 与 Flow 类的核心接口进行互操作。

变量句柄

变量句柄是一个变量或一组变量的引用，包括静态域，非静态域，数组元素和堆外数据结构中的组成部分等。变量句柄的含义类似于已有的方法句柄。变量句柄由 J ava 类 java.lang.invoke.VarHandle 来表示。可以使用类 java.lang.invoke.MethodHandles.Looku p 中的静态工厂方法来创建 VarHandle 对 象。通过变量句柄，可以在变量上进行各种操作。这些操作称为访问模式。不同的访问模式尤其在内存排序上的不同语义。目前一共有 31 种 访问模式，而每种访问模式都 在 VarHandle 中 有对应的方法。这些方法可以对变量进行读取、写入、原子更新、数值原子更新和比特位原子操作等。VarHandle 还 可以用来访问数组中的单个元素，以及把 byte[]数组 和 ByteBuffer 当成是不同原始类型的数组来访问。

在 如下代码 中，我们创建了访问 HandleTarget 类中的域 count 的变量句柄，并在其上进行读取操作。

```
public class HandleTarget {
    public int count = 1;
}
public class VarHandleTest {
    private HandleTarget handleTarget = new HandleTarget();
    private VarHandle varHandle;
    @Before
    public void setUp() throws Exception {
        this.handleTarget = new HandleTarget();
        this.varHandle = MethodHandles
            .lookup()
            .findVarHandle(HandleTarget.class, "count", int.class);
    }
    @Test
    public void testGet() throws Exception {
        assertEquals(1, this.varHandle.get(this.handleTarget));
        assertEquals(1, this.varHandle.getVolatile(this.handleTarget));
        assertEquals(1, this.varHandle.getOpaque(this.handleTarget));
        assertEquals(1, this.varHandle.getAcquire(this.handleTarget));
    }
}
```

改进方法句柄（Method Handle）

类 java.lang.invoke.MethodHandles 增加了更多的静态方法来创建不同类型的方法句柄。

- arrayConstructor：创建指定类型的数组。
- arrayLength：获取指定类型的数组的大小。
- varHandleInvoker 和 varHandleExactInvoker：调用 VarHandle 中的访问模式方法。
- zero：返回一个类型的默认值。
- empty：返 回 MethodType 的返回值类型的默认值。
- loop、countedLoop、iteratedLoop、whileLoop 和 doWhileLoop：创建不同类型的循环，包括 for 循环、while 循环和 do-while 循环。
- tryFinally：把对方法句柄的调用封装在 try-finally 语句中。

在 如下代码 中，我们使用 iteratedLoop 来创建一个遍历 S tring 类型迭代器的方法句柄，并计算所有字符串的长度的总和。

```
public class IteratedLoopTest {
    static int body(final int sum, final String value) {
        return sum + value.length();
    }
    @Test
    public void testIteratedLoop() throws Throwable {
        final MethodHandle iterator = MethodHandles.constant(
            Iterator.class,
            List.of("a", "bc", "def").iterator());
        final MethodHandle init = MethodHandles.zero(int.class);
        final MethodHandle body = MethodHandles
            .lookup()
            .findStatic(
                IteratedLoopTest.class,
                "body",
                MethodType.methodType(
                    int.class,
                    int.class,
                    String.class));
```

```
        final MethodHandle iteratedLoop = MethodHandles
            .iteratedLoop(iterator, init, body);
        assertEquals(6, iteratedLoop.invoke());
    }
}
```

并发

在并发方面，类 CompletableFuture 中增加了几个新的方法。completeAsync 使用一个异步任务来获取结果并完成该 CompletableFuture。orTimeout 在 CompletableFuture 没有在给定的超时时间之前完成，使用 TimeoutException 异常来完成 CompletableFuture。completeOnTimeout 与 o rTimeout 类似，只不过它在超时时使用给定的值来完成 CompletableFuture。新的 Thread.onSpinWai t 方法在当前线程需要使用忙循环来等待时，可以提高等待的效率。

Nashorn

Nashorn 是 Java 8 中引入的新的 JavaScript 引擎。Java 9 中的 Nashorn 已经实现了一些 ECMAScript 6 规范中的新特性，包括模板字符串、二进制和八进制字面量、迭代器 和 for..of 循环和箭头函数等。Nashorn 还提供了 API 把 ECMAScript 源代码解析成抽象语法树（ Abstract Syntax Tree，AST ），可以用来对 ECMAScript 源代码进行分析。

I/O 流新特性

类 java.io.InputStream 中增加了新的方法来读取和复制 InputStream 中包含的数据。

- readAllBytes：读取 InputStream 中的所有剩余字节。
- readNBytes： 从 InputStream 中读取指定数量的字节到数组中。
- transferTo：读取 InputStream 中的全部字节并写入到指定的 OutputStream 中 。

```
public class TestInputStream {
    private InputStream inputStream;
    private static final String CONTENT = "Hello World";
    @Before
    public void setUp() throws Exception {
        this.inputStream =
            TestInputStream.class.getResourceAsStream("/input.txt");
    }
    @Test
    public void testReadAllBytes() throws Exception {
        final String content = new String(this.inputStream.readAllBytes());
        assertEquals(CONTENT, content);
    }
    @Test
    public void testReadNBytes() throws Exception {
        final byte[] data = new byte[5];
        this.inputStream.readNBytes(data, 0, 5);
        assertEquals("Hello", new String(data));
    }
    @Test
    public void testTransferTo() throws Exception {
        final ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        this.inputStream.transferTo(outputStream);
```

```
        assertEquals(CONTENT, outputStream.toString());
    }
}
```

ObjectInputFilter 可以对 ObjectInputStream 中 包含的内容进行检查，来确保其中包含的数据是合法的。可以使用 ObjectInputStream 的方法 setObjectInputFilter 来设置。ObjectInputFilter 在 进行检查时，可以检查如对象图的最大深度、对象引用的最大数量、输入流中的最大字节数和数组的最大长度等限制，也可以对包含的类的名称进行限制。

改进应用安全性能

Java 9 新增了 4 个 SHA- 3 哈希算法，SHA3-224、SHA3-256、SHA3-384 和 S HA3-512。另外也增加了通过 java.security.SecureRandom 生成使用 DRBG 算法的强随机数。如下代码中给出了 SHA-3 哈希算法的使用示例。

```
import org.apache.commons.codec.binary.Hex;
public class SHA3 {
    public static void main(final String[] args) throws NoSuchAlgorithmException {
        final MessageDigest instance = MessageDigest.getInstance("SHA3-224");
        final byte[] digest = instance.digest("".getBytes());
        System.out.println(Hex.encodeHexString(digest));
    }
}
```

用户界面

类 java.awt.Desktop 增加了新的与桌面进行互动的能力。可以使用 addAppEventListener 方法来添加不同应用事件的监听器，包括应用变为前台应用、应用隐藏或显示、屏幕和系统进入休眠与唤醒、以及 用户会话的开始和终止等。还可以在显示关于窗口和配置窗口时，添加自定义的逻辑。在用户要求退出应用时，可以通过自定义处理器来接受或拒绝退出请求。在 A WT 图像支持方面，可以在应用中使用多分辨率图像。

统一 JVM 日志

Java 9 中 ， JVM 有了统一的日志记录系统，可以使用新的命令行选项-Xlog 来控制 JVM 上 所有组件的日志记录。该日志记录系统可以设置输出的日志消息的标签、级别、修饰符和输出目标等。Java 9 移除了在 Java 8 中 被废弃的垃圾回收器配置组合，同时 把 G1 设为默认的垃圾回收器实现。另外，CMS 垃圾回收器已经被声明为废弃。Java 9 也增加了很多可以通过 jcmd 调用的诊断命令。

其他改动方面

在 Java 语言本身，Java 9 允许在接口中使用私有方法。 在 try-with-resources 语句中可以使用 effectively-final 变量。类 java.lang.StackWalker 可 以对线程的堆栈进行遍历，并且支持过滤和延迟访问。Java 9 把对 Unicode 的支持升级到了 8.0。ResourceBundle 加载属性文件的默认编码从 ISO-8859-1 改成了 UTF-8，不再需要使用 native2ascii 命 令来对属性文件进行额外处理。注解@Deprecated 也得到了增强，增加了 since 和 forRemoval 两 个属性，可以分别指定一个程序元素被废弃的版本，以及是否会在今后的版本中被删除。

在如下代码中，buildMessage 是接口 SayHi 中的私有方法，在默认方法 sayHi 中被使用。

```
public interface SayHi {
    private String buildMessage() {
        return "Hello";
    }
    void sayHi(final String message);
    default void sayHi() {
        sayHi(buildMessage());
    }
}
```

结束语

作为 Java 平台最新的一个重大更新，Java 9 中的很多新特性，尤其模块系统，对于 Java 应用的开发会产生深远的影响。本文对 Java 9 中的新特性做了概括的介绍，可以作为了解 Java 9 的基础。这些新特性的相信内容，可以通过官方文档来进一步的了解。