# 前言

Java 11 已于 2018 年 9 月 25 日正式发布，Java 11 仅将提供长期支持服务（LTS, Long-Term-Support）。

本文档编写不是很详细，只希望起个抛砖引玉的作用吧。

# Java 8 新特性（LTS）

主要讨论以下几个：

\*1、Lambda 表达式 − Lambda 允许把函数作为一个方法的参数（函数作为参数传递到方法中）。

\*2、方法引用 − 方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

\*3、默认方法 − 默认方法就是一个在接口里面有了一个实现的方法。

4、新工具 − 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。

\*5、Stream API −新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中。

\*6、Date Time API − 加强对日期与时间的处理。

\*7、Optional 类 − Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。

8、Nashorn, JavaScript 引擎 − Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM上运行特定的javascript应用。

# Java 9 新特性

1、模块系统：模块是一个包的容器，Java 9 最大的变化之一是引入了模块系统（Jigsaw 项目）。

2、REPL (JShell)：交互式编程环境。

3、HTTP 2 客户端：HTTP/2标准是HTTP协议的最新版本，新的 HTTPClient API 支持 WebSocket 和 HTTP2 流以及服务器推送特性。

4、改进的 Javadoc：Javadoc 现在支持在 API 文档中的进行搜索。另外，Javadoc 的输出现在符合兼容 HTML5 标准。

5、多版本兼容 JAR 包：多版本兼容 JAR 功能能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本。

\*6、集合工厂方法：List，Set 和 Map 接口中，新的静态工厂方法可以创建这些集合的不可变实例。

7、私有接口方法：在接口中使用private私有方法。我们可以使用 private 访问修饰符在接口中编写私有方法。

8、进程 API: 改进的 API 来控制和管理操作系统进程。引进 java.lang.ProcessHandle 及其嵌套接口 Info 来让开发者逃离时常因为要获取一个本地进程的 PID 而不得不使用本地代码的窘境。

9、改进的 Stream API：改进的 Stream API 添加了一些便利的方法，使流处理更容易，并使用收集器编写复杂的查询。

10、改进 try-with-resources：如果你已经有一个资源是 final 或等效于 final 变量,您可以在 try-with-resources 语句中使用该变量，而无需在 try-with-resources 语句中声明一个新变量。

11、改进的弃用注解 @Deprecated：注解 @Deprecated 可以标记 Java API 状态，可以表示被标记的 API 将会被移除，或者已经破坏。

12、改进钻石操作符(Diamond Operator) ：匿名类可以使用钻石操作符(Diamond Operator)。

13、改进 Optional 类：java.util.Optional 添加了很多新的有用方法，Optional 可以直接转为 stream。

14、多分辨率图像 API：定义多分辨率图像API，开发者可以很容易的操作和展示不同分辨率的图像了。

15、改进的 CompletableFuture API ： CompletableFuture 类的异步机制可以在 ProcessHandle.onExit 方法退出时执行操作。

16、轻量级的 JSON API：内置了一个轻量级的JSON API

17、响应式流（Reactive Streams) API: Java 9中引入了新的响应式流 API 来支持 Java 9 中的响应式编程。

\*18、G1 垃圾回收器是 Java 9 中 Hotspot 的默认垃圾回收器。

# java10新特性

\*1、局部变量类型推断，var list = new ArrayList<String>(); // ArrayList<String>

2、整合 JDK 代码仓库，将所有现有存储库合并到一个 Mercurial 存储库中

3、统一的垃圾回收接口，hotspot/gc 代码实现方面，引入一个干净的 GC 接口，改进不同 GC 源代码的隔离性

4、并行全垃圾回收器 G1，G1 垃圾回收器是 Java 9 中 Hotspot 的默认垃圾回收器，是以一种低延时的垃圾回收器来设计的，旨在避免进行 Full GC，但是当并发收集无法快速回收内存时，会触发垃圾回收器回退进行 Full GC。

之前 Java 版本中的 G1 垃圾回收器执行 GC 时采用的是基于单线程标记扫描压缩算法（mark-sweep-compact）。为了最大限度地减少 Full GC 造成的应用停顿的影响，Java 10 中将为 G1 引入多线程并行 GC，同时会使用与年轻代回收和混合回收相同的并行工作线程数量，从而减少了 Full GC 的发生，以带来更好的性能提升、更大的吞吐量。

Java 10 中将采用并行化 mark-sweep-compact 算法，并使用与年轻代回收和混合回收相同数量的线程。具体并行 GC 线程数量可以通过：-XX：ParallelGCThreads 参数来调节，但这也会影响用于年轻代和混合收集的工作线程数。

5、应用程序类数据共享，Java 安装程序会把 rt.jar 中的核心类提前转化成内部表示，转储到一个共享存档（shared archive）中。多个 Java 进程（或者说 JVM 实例）可以共享这部分数据。为改善启动和占用空间，Java 10 在现有的 CDS 功能基础上再次拓展，以允许应用类放置在共享存档中。

\*6、线程-局部管控，在已有的 Java 版本中，JVM 线程只能全部启用或者停止，没法做到对单独某个线程的操作。为了能够对单独的某个线程进行操作，Java 10 中线程管控引入 JVM 安全点的概念，将允许在不运行全局 JVM 安全点的情况下实现线程回调，由线程本身或者 JVM 线程来执行，同时保持线程处于阻塞状态，这种方式使得停止单个线程变成可能，而不是只能启用或停止所有线程。通过这种方式显著地提高了现有 JVM 功能的性能开销，并且改变了到达 JVM 全局安全点的现有时间语义。

增加的参数为：-XX:ThreadLocalHandshakes (默认为开启)，将允许用户在支持的平台上选择安全点。

7、移除 Native-Header 自动生成工具

8、额外的 Unicode 语言标签扩展

9、备用存储装置上的堆分配，Java 10 中将使得 JVM 能够使用适用于不同类型的存储机制的堆，在可选内存设备上进行堆内存分配。要在这样的备用设备上进行堆分配，可以使用堆分配参数 -XX：AllocateHeapAt = <path>，这个参数将指向文件系统的文件并使用内存映射来达到在备用存储设备上进行堆分配的预期结果。

10、基于 Java 的 实验性 JIT 编译器

11、根证书认证，自 Java 9 起在 keytool 中加入参数 -cacerts，可以查看当前 JDK 管理的根证书。而 Java 9 中 cacerts 目录为空，这样就会给开发者带来很多不便。从 Java 10 开始，将会在 JDK 中提供一套默认的 CA 根证书。

# Java 11 新特性介绍（LTS）

1、基于嵌套的访问控制，Java 11 中引入了两个新的属性：一个叫做 NestMembers 的属性，用于标识其它已知的静态 nest 成员；另外一个是每个 nest 成员都包含的 NestHost 属性，用于标识出它的 nest 宿主类。

\*2、标准 HTTP Client 升级，对Http Client API 进行了标准化，现在完全支持异步非阻塞。通过 CompleteableFutures 提供非阻塞请求和响应语义，主流开源 API（如：Apache HttpClient、Jetty、OkHttp 等）类似甚至拥有更高的性能。

3、Epsilon：低开销垃圾回收器，Epsilon 垃圾回收器和其他 OpenJDK 的垃圾回收器一样，可以通过参数 -XX:+UseEpsilonGC 开启。

4、简化启动单个源代码文件的方法

$ java HelloWorld.java

也等同于：

$ javac HelloWorld.java

$ java -cp . hello.World

5、用于 Lambda 参数的局部变量语法，@Nonnull var x = new Foo(); (@Nonnull var x, @Nullable var y) -> x.process(y)

6、低开销的 Heap Profiling，Java 11 中提供一种低开销的 Java 堆分配采样方法，能够得到堆分配的 Java 对象信息，并且能够通过 JVMTI 访问堆信息。

7、支持 TLS 1.3 协议

\*8、ZGC：可伸缩低延迟垃圾收集器。ZGC 即 Z Garbage Collector（垃圾收集器或垃圾回收器），这应该是 Java 11 中最为瞩目的特性，没有之一。

9、飞行记录器，Java 语言中的飞行记录器类似飞机上的黑盒子，是一种低开销的事件信息收集框架，主要用于对应用程序和 JVM 进行故障检查、分析。启用飞行记录器参数如下：-XX:StartFlightRecording

10、动态类文件常量，Java 11 引入了类似invokedynamic 的机制，扩展了 Java 文件格式，以支持新的常量池：CONSTANT\_Dynamic

11、Nashorn JavaScript 在jdk11中移除

# Java 12 新特性

1、Shenandoah：一个低停顿垃圾收集器（实验阶段）

2、增加一套微基准套件

3、Switch 表达式扩展（预览功能）

int dayNumber = switch (day) {

case MONDAY, FRIDAY, SUNDAY -> 6;

case TUESDAY -> 7;

case THURSDAY, SATURDAY -> 8;

case WEDNESDAY -> 9;

default -> throw new IllegalStateException("Huh? " + day);

}

Java 11 以及之前版本中，Switch 表达式支持下面类型： byte、char、short、int、Byte、Character、Short、Integer、enum、tring，在未来的某个 Java 版本有可能会允许支持 float、double 和 long （以及上面类型的封装类型）。

4、引入 JVM 常量 API

5、改进 AArch64 实现

6、使用默认类数据共享（CDS）存档

7、改善 G1 垃圾收集器，使其能够中止混合集合

8、增强 G1 垃圾收集器，使其能自动返回未用堆内存给操作系统

# Java 13 新特性

1、动态应用程序类-数据共享

2、增强 ZGC 释放未使用内存

3、Socket API 重构，引入 NioSocketImpl 的实现用以替换 SocketImpl 的 PlainSocketImpl 实现，此实现与 NIO（新 I/O）实现共享相同的内部基础结构，并且与现有的缓冲区高速缓存机制集成在一起，因此不需要使用线程堆栈。

4、Switch 表达式扩展（预览功能），在块中引入了 yield 语句来返回值，而不是使用 break。这意味着，Switch 表达式（返回值）应该使用 yield，而 Switch 语句（不返回值）应该使用 break

5、文本块（预览功能）

# Java 14 新特性

1、增强 instanceOf 类型推断

2、增强 switch case 代码块

3、新增构造函数关键字 record

4、字符串块

5、增强NPE NullPointerException 错误推断

6、Non-Volatile Mapped Byte Buffers

7、打包工具

8、G1 的非均匀存储器访问

9、JFR 事件流

10、其他特性

废除了 Solaris / SPARC, Solaris / x64, and Linux / SPARC， 并会在以后的版本中彻底移除

移除了CMS垃圾回收器

给windows和macOS新增了特定垃圾回收器

废除了并行清楚和串行GC（ParallelScavenge + SerialOld GC）组合

移除了pack200 相关工具和api

# 示例

说明：示例代码基于jdk11

|  |
| --- |
| package com.test.demo;  import java.io.IOException;  import java.net.URI;  import java.net.http.HttpClient;  import java.net.http.HttpRequest;  import java.net.http.HttpResponse;  import java.nio.charset.StandardCharsets;  import java.time.\*;  import java.util.\*;  import java.util.stream.Collectors;  import java.util.stream.Stream;  public class DemoApplication {  public static void main(String[] args) {  var list = new ArrayList<String>(); // var 定义局部变量，自动推断类型，jdk10中增加  list.add("11");  list.add("22");  list.add("33");  list.add("44");  //list.forEach(item-> System.out.println(item)); // lambda表达式输出，jdk8中增加  list.parallelStream().forEach(System.out::println); // 流，并行输出，输出方式为方法引用，jdk8中增加  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  List<String> list1 = Arrays.asList("aa","bb","","cc","dd","","ee","cc","","cc","dd",""); // 集合设值, List，Set 和 Map 接口创建集合方法，jdk9中增加  System.out.println(list1.stream().filter(item -> item.isEmpty()).count()); // 流，过滤空的并统计数量  //System.out.println(list1.parallelStream().filter(item -> item.isEmpty()).count()); // 并行过滤空的并统计数量  list1.stream().filter(item -> !item.isEmpty()).collect(Collectors.toList()).forEach(System.out::println); // 过滤不为空的并转化为新的集合并输出  String filters = list1.stream().filter(item -> !item.isEmpty()).collect(Collectors.joining(",")); // 过滤不为空的并合并成字符串  System.out.println(filters);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  Random random = new Random();  //random.ints().limit(5).forEach(System.out::println); // 限制5个随机int型数据并输出  random.ints().limit(6).sorted().forEach(System.out::println); // 限制6个随机int型数据并升序输出  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  List<Integer> numbers = Arrays.asList(2,4,3,2,1,3,5);  List<Integer> seqNumbers = numbers.stream().map(item -> item\*item).distinct().collect(Collectors.toList()); // 集合元素平方去重之后再转化为新集合  seqNumbers.forEach(System.out::println);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  var nums = Arrays.asList(3, 2, 2, 3, 7, 3, 5);  IntSummaryStatistics stats = nums.stream().mapToInt((x) -> x).summaryStatistics(); // 统计  System.out.println("列表中元素数量：" + stats.getCount() + "，列表中最大的数：" + stats.getMax() + ", 列表中最小的数：" + stats.getMin() + ", 列表中数之和：" + stats.getSum() + ", 列表中数的平均数：" + stats.getAverage());  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  new Thread(() -> System.out.println("线程id：" + Thread.currentThread().getId() + " 启动了")).start(); // 创建线程  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  java.util.Collections.sort(nums,(s1, s2) -> s1.compareTo(s2)); // 升序排序  System.out.println(nums);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // 函数式接口，jdk8中增加  GreetingService greetService = message -> System.out.println("Hello " + message);  greetService.sayMessage("abc");  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // 默认方法，jdk8中增加；私有方法，jdk9中增加  GreetingService greeting = new Car();  greeting.print();  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // Nashorn JavaScript 在jdk8中加入，又在jdk11中移除  Optional<Integer> a = Optional.ofNullable(2); // 设置允许为空的值，jdk8中增加  System.out.println(a.isPresent()); // 判断值是否存在  Integer b = a.orElse(Integer.valueOf(0)); // 如果存在返回他，否则返回默认值  System.out.println(b);  System.out.println(a.get()); // 获取值，值需要存在  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // 日期时间api，jdk8中增加  LocalDateTime ldt = LocalDateTime.now();  System.out.println(ldt);  System.out.println(ldt.getYear() + "-" + ldt.getMonth().getValue() + "-" + ldt.getDayOfMonth() + " " + ldt.getHour() + ":" + ldt.getMinute() + ":" + ldt.getSecond());  System.out.println(ldt.withDayOfMonth(6).withYear(2018));  LocalDate ld = ldt.toLocalDate();  System.out.println(ld);  LocalDate ld1 = LocalDate.now();  System.out.println(ld1);  System.out.println(LocalDate.of(2014, Month.DECEMBER, 12));  System.out.println(LocalTime.parse("15:16:23"));  // LocalDateTime转换为Date  ZoneId zoneId = ZoneId.systemDefault();  System.out.println("当前时区：" + zoneId);  ZonedDateTime zdt = ldt.atZone(zoneId);  Instant instant = zdt.toInstant();  Date date = Date.from(instant);  System.out.println(date);  // Date转换为LocalDateTime  Date date1 = new Date();  Instant instant1 = date1.toInstant(); // 表示时刻，不直接对应年月日信息，需要通过时区转换  ZoneId zoneId1 = ZoneId.systemDefault(); // 表示时区  ZonedDateTime zdt1 = instant1.atZone(zoneId1); // 表示特定时区的日期和时间  LocalDateTime ldt1 = zdt1.toLocalDateTime(); // 表示与时区无关的日期和时间信息，不直接对应时刻，需要通过时区转换  System.out.println(ldt1);  System.out.println("Date转换为LocalDateTime:" + date2LocalDateTime(new Date()));  System.out.println("LocalDateTime转换为Date:" + localDateTime2Date(LocalDateTime.now()));  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // Base64，jdk8中增加，三种方式：getEncoder()，getMimeEncoder()，getUrlEncoder()  String encode = base64Encoded("hello?java11");  System.out.println("base64加密之后：" + encode);  String decode = base64Decoded(encode);  System.out.println("base64解密之后：" + decode);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // 新增进程api，jdk9中增加  ProcessBuilder pb = new ProcessBuilder("notepad.exe");  try {  // 打开记事本  Process p = pb.start();  System.out.printf("Process ID : %s%n", p.pid());  String np = "Not Present";  ProcessHandle.Info info = p.info();  System.out.printf("Command name : %s%n", info.command().orElse(np));  System.out.printf("Command line : %s%n", info.commandLine().orElse(np));  System.out.printf("Start time: %s%n",  info.startInstant().map(i -> i.atZone(ZoneId.systemDefault())  .toLocalDateTime().toString()).orElse(np));  System.out.printf("Arguments : %s%n",  info.arguments().map(c -> Stream.of(c).collect(  Collectors.joining(" "))).orElse(np));  System.out.printf("User : %s%n", info.user().orElse(np));  } catch (IOException e) {  e.printStackTrace();  }  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // jdk11中Http Client API 进行了标准化，通过 CompleteableFutures 提供异步非阻塞  /\*HttpClient client = HttpClient.newHttpClient();  HttpRequest request = HttpRequest.newBuilder()  .uri(URI.create("http://openjdk.java.net/"))  .build();  client.sendAsync(request, HttpResponse.BodyHandlers.ofString())  .thenApply(HttpResponse::body)  .thenAccept(System.out::println)  .join();\*/  }  /\*\*  \* LocalDateTime转换为Date  \* @param localDateTime  \* @return  \*/  public static Date localDateTime2Date(LocalDateTime localDateTime){  return Date.from(localDateTime.atZone(ZoneId.systemDefault()).toInstant());  }  /\*\*  \* Date转换为LocalDateTime  \* @param date  \* @return  \*/  public static LocalDateTime date2LocalDateTime(Date date){  return date.toInstant().atZone(ZoneId.systemDefault()).toLocalDateTime();  }  /\*\*  \* Base64加密  \* @param encode  \* @return  \*/  public static String base64Encoded(String encode){  return Base64.getEncoder().encodeToString(encode.getBytes(StandardCharsets.UTF\_8));  }  /\*\*  \* Base64解密  \* @param decode  \* @return  \*/  public static String base64Decoded(String decode){  return new String(Base64.getDecoder().decode(decode), StandardCharsets.UTF\_8);  }  }  @FunctionalInterface //函数接口，jdk1.8增加  interface GreetingService {  String MYSQL = "MySql\_Database";  void sayMessage(String message);  // 默认方法，jdk1.8增加  default void print(){  System.out.println("我是一辆车!");  log(MYSQL, "INFO");  }  // 静态方法，jdk1.8增加  static void blowHorn(){  System.out.println("按喇叭!!!");  }  // 私有方法，jdk1.9增加  private void log(String message, String prefix) {  getConnection();  System.out.println("Log Message : " + prefix);  closeConnection();  }  // 私有静态方法，jdk1.9增加  private static void getConnection() {  System.out.println("Open Database connection");  }  // 私有静态方法，jdk1.9增加  private static void closeConnection() {  System.out.println("Close Database connection");  }  }  class Car implements GreetingService{  @Override  public void sayMessage(String message) {  }  @Override  public void print() {  GreetingService.super.print();  GreetingService.blowHorn();  System.out.println("我是一辆汽车!");  }  } |