

JDK11&JDK12新特性课程

第一章 JDK 11新特性介绍

1.1 初识JDK11新特性

北京时间 2018年9月26日，Oracle官方宣布**JDK 11 (18.9 LTS)**正式发布。这是Java 8以后支持的首个长期版本。非常值得关注。从官网即可下载，最新发布的JDK 11将带来ZGC、Http Client等重要特性，一共包含17个JEP (JDK Enhancement Proposals, JDK增强提案)。

本次发布的Java 11和2017年9月份发布的Java 9以及2018年3月份发布的Java 10相比，其最大的区别就是：在长期支持(Long-Term-Support)方面，Oracle表示会对JDK 11提供大力支持。

下图是详细的发行时间和支持的周期时间：

Examples of some key product dates for Oracle Java SE product offerings include:

Oracle Java SE Support Roadmap ^{*†}				
Release	GA Date	Premier Support Until	Extended Support Until	Sustaining Support
6	December 2006	December 2015	December 2018	Indefinite
7	July 2011	July 2019	July 2022 ^{*****}	Indefinite
8 ^{**}	March 2014	March 2022	March 2025	Indefinite
9 (non-LTS)	September 2017	March 2018	Not Available	Indefinite
10 (non-LTS)	March 2018	September 2018	Not Available	Indefinite
11 (LTS)	September 2018	September 2023	September 2026	Indefinite
12 (non-LTS)	March 2019	September 2019	Not Available	Indefinite
13 (non-LTS)	September 2019 ^{***}	March 2020	Not Available	Indefinite

1.2 JDK 11新特性更新列表

我们可以在如下网站中查看JDK11的新特性 增强提案详情：

<http://openjdk.java.net/projects/jdk/11/>

JDK 11新特性增强提案如下。共17个JEP (JDK Enhancement Proposals, JDK 增强提案)：

Features

181: Nest-Based Access Control
309: Dynamic Class-File Constants
315: Improve Aarch64 Intrinsics
318: Epsilon: A No-Op Garbage Collector
320: Remove the Java EE and CORBA Modules
321: HTTP Client (Standard)
323: Local-Variable Syntax for Lambda Parameters
324: Key Agreement with Curve25519 and Curve448
327: Unicode 10
328: Flight Recorder
329: ChaCha20 and Poly1305 Cryptographic Algorithms
330: Launch Single-File Source-Code Programs
331: Low-Overhead Heap Profiling
332: Transport Layer Security (TLS) 1.3
333: ZGC: A Scalable Low-Latency Garbage Collector
(Experimental)
335: Deprecate the Nashorn JavaScript Engine
336: Deprecate the Pack200 Tools and API

181：基于嵌套的访问控制(Nest-Based Access Control) 309：动态类文件常量(Dynamic Class-File Constants)
315：改进 Aarch64 内在函数(Improve Aarch64 Intrinsics) 318：Epsilon：无操作垃圾收集器(Epsilon: A No-Op Garbage Collector) 320：移除Java EE和CORBA模块(Remove the Java EE and CORBA Modules) 321：HTTP客户端(标准) 323：用于 Lambda 参数的局部变量语法(Local-Variable Syntax for Lambda Parameters) 324：Curve25519 和 Curve448 算法的密钥协议(Key Agreement with Curve25519 and Curve448) 327：Unicode 10
328：Flight Recorder 329：ChaCha20 和 Poly1305 加密算法(ChaCha20 and Poly1305 Cryptographic Algorithms) 330：启动单文件源代码程序(Launch Single-File Source-Code Programs) 331：低开销堆分析 (Low-Overhead Heap Profiling) 332：传输层安全性 (TLS) 1.3 333：ZGC:可扩展的低延迟垃圾收集器 (实验性)
335：弃用Nashorn JavaScript引擎 336：弃用 Pack200 工具和 API(Deprecate the Pack200 Tools and API)

1.3 安装JDK 11

- 去官网下载JDK 11
- 下载地址：<https://www.oracle.com/technetwork/java/javase/downloads/index.html>
- 安装
- 启动IDEA创建一个模块集成JDK 11

第二章 JDK 11新特性详解

2.1 基于嵌套的访问控制

JEP 181：基于嵌套的访问控制(Nest-Based Access Control)

目标：如果你在一个类中嵌套了多个类，各类中可以直接访问彼此的私有成员。因为JDK 11开始在private,public,protected的基础上，JVM又提供了一种新的访问机制：Nest。



我们先来看下JDK 11之前的如下案例：

```
class Outer {
    private int outerInt;

    class Inner {
        public void test() {
            System.out.println("Outer int = " + outerInt);
        }
    }
}
```

在JDK 11之前 执行编译的最终结果的class文件形式如下：

```
class Outer {
    private int outerInt;
    public int access$000() {
        return outerInt;
    }
}

class Inner$Outer {
    Outer outer;
    public void test() {
        System.out.println("Outer int = " + outer.access$000());
    }
}
```

以上方案虽然从逻辑上讲，内部类是与外部类相同的代码实体的一部分，但它被编译为一个单独的类。因此，它需要编译器创建合成桥接方法，以提供对外部类的私有字段的访问。

这种方案一个很大的坑是反射的时候会有问题。当使用反射在内部类中访问外部类的私有成员outerInt时会报IllegalAccessError错误。这个是让人不能理解的，因为反射还是源码级访问的权限。

```
class Outer {
    private int outerInt;
    class Inner {
        public void test() throws Exception {
            System.out.println("Outer int = " + outerInt);
            // JDK 11之前，如下代码报出异常：IllegalAccessError
            Class c = Outer.class;
            Field f = c.getDeclaredField("outerInt");
            f.set(Outer.this, 23);
        }
    }

    public static void main(String[] args) throws Exception {
        new Outer().new Inner().test();
    }
}
```

JDK 11开始，嵌套是一种访问控制上下文，它允许多个class同属一个逻辑代码块，但是被编译成多个分散的class文件，它们访问彼此的私有成员无需通过编译器添加访问扩展方法，而是可以直接进行访问，如果上述代码可以直接通过反射访问外部类的私有成员，而不会出现权限问题，请看如下代码：

```
class Outer {
    private int outerInt;

    class Inner {
        public void test() throws Exception {
            System.out.println("Outer int = " + outerInt);
            // JDK 11之后，如下代码不会出现异常
            Class c = Outer.class;
            Field f = c.getDeclaredField("outerInt");
            f.set(Outer.this, 23);
        }
    }

    public static void main(String[] args) throws Exception {
        new Outer().new Inner().test();
    }
}
```

2.2 局部变量var类型推断

JEP 323: 用于 Lambda 参数的局部变量语法(Local-Variable Syntax for Lambda Parameters)

首先了解什么是局部变量类型推断？

```
var rs = "itheima";
System.out.println(rs);
```

大家看出来了，局部变量类型推断就是左边的类型直接使用 var 定义，而不用写具体的类型，编译器能根据右边的表达式自动推断类型，如上面的 String。

```
var rs = "itheima";
System.out.println(rs);
就等于：
String rs = "itheima"
```

局部变量推断对于较复杂的类型也有很好的简化作用，请看如下对Map集合的遍历方式的简化：

```
public class Demo01 {
    public static void main(String[] args) {
        Map<String,Integer> maps = new HashMap<>();
        // 1. 添加元素：添加键值对元素
        maps.put("iphonex" , 1);
        maps.put("huawei" , 8);
        maps.put("Java" , 1);
        System.out.println(maps);

        /*Set<Map.Entry<String, Integer>> entries = maps.entrySet();
```

```
for(Map.Entry<String,Integer> entry : entries){
    String key = entry.getKey();
    Integer value = entry.getValue();
    System.out.println(key + "---->" + value);
}*/

var entries = maps.entrySet();
for(var entry : entries){
    String key = entry.getKey();
    Integer value = entry.getValue();
    System.out.println(key + "---->" + value);
}
}
```

在声明隐式类型的lambda表达式的形参时允许使用var,使用var的好处是在使用lambda表达式时给参数加上注解。Lambda是用于简化函数式接口的匿名内部类的形式。

```
List<Integer> nums = new ArrayList<>();
Collections.addAll(nums , 9 , 10 , 3);
nums.sort((@Deprecated var o2 , @Deprecated var o1) -> o2 - o1);
System.out.println(nums);
```

var局部变量的类型推断.语法注意事项:

- **var a;** 这样不可以, 因为无法推断。
- 类的属性的数据类型不可以使用var
- var不同于js,类型依然是静态类型, var不可以在lambda表达式中混用。

2.3 String新增处理方法

如以下所示,JDK11新增了一些使用的String处理方法。

```
// 判断字符串是否为空白
System.out.println("    ".isBlank()); // true
// 去除首尾空白
System.out.println("    itheima    ".strip()); // 可以去除全角的空白字符
System.out.println("    itheima    ".trim()); // 不能去除全角的空白字符
// 去除尾部空格
System.out.println("    itheima    ".stripTrailing());
// 去除首部空格
System.out.println("    itheima    ".stripLeading());
// 复制字符串
System.out.println("itheima".repeat(3)); // itheimaitheimaitheima
// 行数统计
System.out.println("A\nB\nC".lines().count()); // 3;
```

2.4 集合新增的API

```
List<Integer> list = List.of(10 , 11 , 12);
System.out.println(list);
// list为不可变集合，添加新元素会报出：UnsupportedOperationException
// System.out.println(list.add(13));

// 把List集合转换成数组
// JDK 11前的方式
Integer[] nums1 = list.toArray(new Integer[0]);
// JDK 11开始之后新增方式
Integer[] nums2 = list.toArray(Integer[]::new);
```

2.5 更方便的IO

Path

```
of(String, String...)
```

此前我们需要使用 Paths.get()。现在，我们像其他类一样使用 of()。

Files

writeString(Path, CharSequence) 我们可以使用该方法来保存一个 String 字符串。

```
Files.writeString(Path.of("test.txt"), "Hello!!!")
```

readString(Path)

我们可以使用该方法来读取一个 String 字符串。

```
Files.readString(Path.of("test.txt"), StandardCharsets.UTF_8);
```

Reader

nullReader()

使用该方法，我们可以得到一个不执行任何操作的 Reader。

Writer

nullWriter() 使用该方法，我们可以得到一个不执行任何操作的 Writer。

InputStream

nullInputStream() 使用该方法，我们可以得到一个不执行任何操作的 InputStream。

InputStream 还终于有了一个非常有用的方法：**transferTo**，可以用来将数据直接传输到 OutputStream，这是在处理原始数据流时非常常见的一种用法，如下示例。

```
try (var is = Demo01.class.getResourceAsStream("dlei.txt");
    var os = new FileOutputStream("dlei01.txt")) {
    is.transferTo(os); // 把输入流中的所有数据直接自动地复制到输出流中
}
```

OutputStream

`nullOutputStream()` 使用该方法，我们可以得到一个不执行任何操作的 `OutputStream`。

2.6 标准Java HTTP客户端

JEP 321：标准HTTP客户端

从 Java 9 开始引入了一个处理 HTTP 请求的 HTTP Client API，不过当时一直处于孵化阶段，而在 Java 11 中已经为正式可用状态，作为一个标准API提供在 `java.net.http` 供大家使用，该 API 支持同步和异步请求。

目标

取代繁琐的 `HttpURLConnection` 的请求。

动机

JDK8中的 `HttpURLConnection` API 及其实现存在许多问题：

- `URLConnection` API 是设计时考虑了多种协议，而这些都是现在已经不存在（ftp, gopher, 等）。
- API 早于 HTTP / 1.1 并且过于抽象。
- 难于使用
- 仅能在阻塞模式下工作
- 难于维护

优势

HTTP Client 的优势

- API 必须是易于使用的，包括简单的阻塞模式
- 必须支持通知机制如 HTTP 消息头收到、错误码、HTTP 消息体收到
- 简洁的 API 能够支持 80-90% 的需求
- 必须支持标准和通用身份验证机制
- 必须能够轻松使用 WebSocket
- 必须支持 HTTP 2
- 必须执行与现有网络 API 一致的安全检查
- 必须对 lambda 表达式等新语言功能很友好
- 应该对嵌入式系统友好，避免永久运行的后台线程
- 必须支持 HTTPS / TLS
- 满足 HTTP 1.1 和 HTTP 2 的性能要求

使用

需求：使用 `Http Client` 请求如下网址内容：



```
http://api.k780.com:88/?  
app=life.time&appkey=10003&sign=b59bc3ef6191eb9f747dd4e83c99f2a4&format=json
```

来看一下 HTTP Client 的用法:

```
// 同步  
// 1.创建HttpClient对象。  
var client = HttpClient.newHttpClient();  
// 2.创建请求对象: request,封装请求地址和请求方式get.  
var request = HttpRequest.newBuilder().uri(URI.create("http://api.k780.com:88/?  
app=life.time&appkey=10003&sign=b59bc3ef6191eb9f747dd4e83c99f2a4&format=json"))  
    .GET().build();  
// 3.使用HttpClient对象发起request请求。得到请求响应对象response  
HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());  
// 4.得到响应的状态码信息  
System.out.println(response.statusCode());  
// 5.得到响应的数据信息输出  
System.out.println(response.body());  
  
// 异步  
// 1.创建HttpClient对象。  
var client = HttpClient.newHttpClient();  
// 2.创建请求对象: request,封装请求地址和请求方式get.  
  
var request = HttpRequest.newBuilder().uri(URI.create("http://api.k780.com:88/?  
app=life.time&appkey=10003&sign=b59bc3ef6191eb9f747dd4e83c99f2a4&format=json"))  
    .GET().build();  
// 3.使用HttpClient对象发起request异步请求。得到请求响应对象future  
CompletableFuture<HttpResponse<String>> future =  
    client.sendAsync(request, HttpResponse.BodyHandlers.ofString());  
// 4.监听future对象的响应结果,并通过join方法进行异步阻塞式等待。  
future.whenComplete((resp ,ex) -> {  
    if(ex != null ){  
        ex.printStackTrace();  
    } else{  
        System.out.println(resp.statusCode());  
        System.out.println(resp.body());  
    }  
}).join();  
  
// future.thenApply(t -> t.body()).thenAccept(System.out::println);
```

2.7 Unicode 10

JEP 327: Unicode 10

目标: 升级现有平台的API,支持Unicode 10,Unicode10的标准请参考网站 (<http://unicode.org/versions/Unicode10.0.0>)

目前支持最新的Unicode的类主要有

- java.lang包下的Character, String
- java.awt.font下的相关类。
- java.text包下的Bidi,Normalizer等。

String对Unicode的示例：

```
System.out.println("\uD83E\uDD93");
System.out.println("\uD83E\uDD92");
System.out.println("\uD83E\uDDDA");
System.out.println("\uD83E\uDD99");
System.out.println("\uD83E\uDD11");
System.out.println("\uD83E\uDD88");
System.out.println("\uD83E\uDD95");
System.out.println("\uD83E\uDD2e");
```

2.8 改进Aarch64函数

JEP 315：改进 Aarch64 内在函数(Improve Aarch64 Intrinsics)

目标：改进现有的字符串和数组函数，并在Aarch64处理器上为java.lang.Math 下的sin, cos 和log函数实现新的内联函数。从而实现专用的CPU架构下提高应用程序的性能。

代码示例：

```
public static void main(String[] args) {
    long startTime = System.nanoTime();
    for(int i = 0 ; i < 10000000 ; i++) {
        Math.sin(i);
        Math.cos(i);
        Math.log(i);
    }
    long endTime = System.nanoTime();
    // JDK 11下耗时: 1564ms
    // JDK 8前耗时: 10523ms
    System.out.println(TimeUnit.NANOSECONDS.toMillis(endTime-startTime)+"ms");
}
```

2.9 更简化的编译运行程序

JEP 330：增强java启动器支持运行单个java源代码文件的程序。

目标：使用 java HelloWorld.java运行源文件代码。

一个命令编译运行源代码：**java HelloWorld.java**



```
D:\soft\java\jdk-11.0.4\bin>java C:\Users\dlei\Desktop\HelloWorld.java
输出: 0
输出: 1
输出: 2
输出: 3
输出: 4
输出: 5
输出: 6
输出: 7
输出: 8
输出: 9
D:\soft\java\jdk-11.0.4\bin>
```

2.10 Epsilon垃圾收集器

JEP 318: Epsilon: 无操作垃圾收集器(Epsilon: A No-Op Garbage Collector)

目标

JDK上对这个特性的描述是: 开发一个处理内存分配但不实现任何实际内存回收机制的GC, 一旦java的堆被耗尽, JVM就直接关闭。

如果有System.gc()调用, 实际上什么也不会发生垃圾对象的回收操作(这种场景下和-XX:+DisableExplicitGC效果一样), 因为没有内存回收, 这个实现可能会警告用户尝试强制GC是徒劳。

先使用G1垃圾收集器

```
public class Demo08 {
    public static void main(String[] args) {
        System.out.println("程序开始");
        List<Garbage> list = new ArrayList<>();
        long count = 0;
        while (true) {
            list.add(new Garbage(list.size() + 1));

            if (list.size() == 1000000 && count == 0) {
                list.clear();
                count++;
            }
        }
        System.out.println("程序结束");
    }
}

class Garbage {

    private int number;

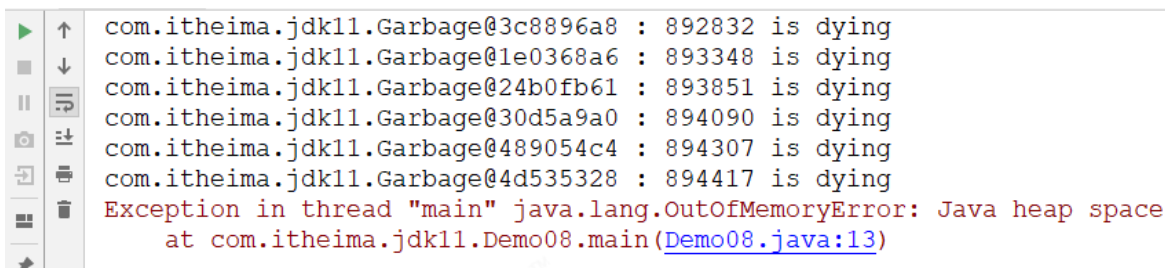
    public Garbage(int number) {
        this.number = number;
    }
}
```

```
/**
 * GC在清除对象时，会调用finalize()方法
 */
@Override
public void finalize() {
    System.out.println(this + " : " + number + " is dying");
}
}
```

启动参数(设置堆内存空间大小):

```
-Xms100m -Xmx100m
```

运行程序后，结果如下:



```
com.itheima.jdk11.Garbage@3c8896a8 : 892832 is dying
com.itheima.jdk11.Garbage@1e0368a6 : 893348 is dying
com.itheima.jdk11.Garbage@24b0fb61 : 893851 is dying
com.itheima.jdk11.Garbage@30d5a9a0 : 894090 is dying
com.itheima.jdk11.Garbage@489054c4 : 894307 is dying
com.itheima.jdk11.Garbage@4d535328 : 894417 is dying
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at com.itheima.jdk11.Demo08.main(Demo08.java:13)
```

会发现G1一直回收对象，直到内存不够用。

使用Epsilon垃圾收集器

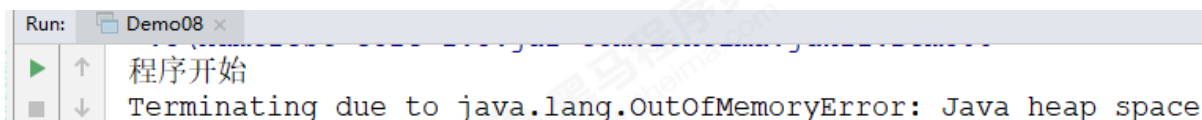
用法

UnlockExperimentalVMOptions: 解锁隐藏的虚拟机参数

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseEpsilonGC
-Xms100m
-Xmx100m
```

如果使用选项-XX:+UseEpsilonGC, 程序很快就因为堆空间不足而退出。

运行程序后，结果如下:



```
Run: Demo08 x
程序开始
Terminating due to java.lang.OutOfMemoryError: Java heap space
```

会发现很快就内存溢出了，因为Epsilon不会去回收对象。

Epsilon垃圾收集器主要用途

- Performance testing: 性能测试(它可以帮助过滤掉GC引起的性能假象)
- Memory pressure testing, 在测试java代码时，确定分配内存的阈值有助于设置内存压力常量值。这时no-op就很有用，它可以简单地接受一个分配的内存分配上限，当内存超限时就失败。例如：测试需要分配小于1G的内存，就使用-Xmx1g参数来配置no-op GC，然后当内存耗尽的时候就直接crash
- 非常短的JOB任务(对象这种任务，接受GC清理堆那都是浪费空间)

- VM interface testing, 以VM开发视角，有一个简单的GC实现，有助于理解VM-GC的最小接口实现。它也用于证明VM-GC接口的健全性。
- Last-drop latency improvements, 对那些极端延迟敏感的应用，开发者十分清楚内存占用，或者是几乎没有垃圾回收的应用，此时耗时较长的GC周期将会是一件坏事。
- Last-drop throughput improvements, 即便对那些无需内存分配的工作，选择一个GC意味着选择了一系列的GC屏障，所有的OpenJDK GC都是分代的，所以他们至少会有一个写屏障。避免这些屏障可以带来一点点的吞吐量提升。

2.11 ZGC 可伸缩低延迟垃圾收集器

JEP 333: ZGC:可伸缩低延迟垃圾收集器 (Experimental实验性)，后面带了Experimental, 说明还不建议用到生产环境。

目标

GC是Java主要优势之一。然而，当GC停顿太长，就会开始影响应用的响应时间。消除或者减少GC停顿时长，Java将对更广泛的应用场景是一个更有吸引力的平台。此外，现代系统中可用内存不断增长，用户和程序员希望JVM能够以高效的方式充分利用这些内存，并且无需长时间的GC暂停时间。

今天，应用程序同时为数千甚至数百万用户提供服务的情况并不少见。这些应用程序需要大量内存。但是，管理所有内存可能会轻易影响应用程序性能。为了解决这个问题，Java 11引入了Z垃圾收集器（ZGC）作为实验性垃圾收集器（GC）实现。

ZGC全称是Z Garbage Collector，是一款可伸缩(scalable)的低延迟(low latency garbage)、并发(concurrent)垃圾回收器，旨在实现以下几个目标：

- 停顿时间不超过10ms
- 停顿时间不随heap大小或存活对象大小增大而增大
- 可以处理从几百G到几TB的内存大小,远剩余前一代的G1。
- 初始只支持64位系统；

用法

运行我们的应用程序时，我们可以使用以下命令行选项启用ZGC：

`-XX: +UnlockExperimentalVMOptions -XX: +UseZGC`

请注意：因为ZGC还处于实验阶段，所以需要通过JVM参数UnlockExperimentalVMOptions 来解锁这个特性。

平台支持

ZGC目前只在Linux/x64上可用，如果有足够的需求，将来可能会增加对其他平台的支持。

目前只支持64位的linux系统。

ZGC和G1停顿时间比较：

ZGC

avg: 1.021ms (+/-0.215ms)

95th percentile: 1.392ms

99th percentile: 1.512ms

99.9th percentile: 1.663ms

99.99th percentile: 1.681ms

max: 1.681ms

G1

avg: 157.202ms (+/-71.126ms)

95th percentile: 316.672ms

99th percentile: 428.095ms

99.9th percentile: 543.846ms

99.99th percentile: 543.846ms

max: 543.846ms

2.12 其他新特性

移除项

- 1、移除了com.sun.awt.AWTUtilities
- 2、移除了sun.misc.Unsafe.defineClass, 使用java.lang.invoke.MethodHandles.Lookup.defineClass来替代
- 3、移除了Thread.destroy()以及 Thread.stop(Throwable)方法
- 4、移除了sun.nio.ch.disableSystemwideOverlappingFileLockCheck、sun.locale.formatasdefault属性
- 5、移除了jdk.snmp模块
- 6、移除了javafx, openjdk估计是从java10版本就移除了, oracle jdk10还尚未移除javafx, 而java11版本则oracle的jdk版本也移除了javafx
- 7、移除了Java Mission Control, 从JDK中移除之后, 需要自己单独下载
- 8、移除了这些Root Certificates : Baltimore Cybertrust Code Signing CA, SECOM , AOL and Swisscom

废弃项

- 1、-XX+AggressiveOpts选项
- 2、-XX:+UnlockCommercialFeatures
- 3、-XX:+LogCommercialFeatures选项也不再需要

JEP : 320: 移除Java EE和CORBA模块(Remove the Java EE and CORBA Modules)

- 1、java.xml.ws,
- 2、java.xml.bind,
- 3、java.xml.ws,
- 4、java.xml.ws.annotation,
- 5、jdk.xml.bind,
- 6、jdk.xml.ws被移除,
只剩下java.xml, java.xml.crypto,jdk.xml.dom这几个模块
- 7、java.corba,
- 8、java.se.ee,
- 9、java.activation,
- 10、java.transaction被移除, 但是java11新增一个java.transaction.xa模块

JEP : 335 : Deprecate the Nashorn JavaScript Engine

废除Nashorn javascript引擎, 在后续版本准备移除掉。

JEP : 336 : Deprecate the Pack200 Tools and API

Java5中帶了一个压缩工具:Pack200, 这个工具能对普通的jar文件进行高效压缩。其实现原理是根据Java类特有的结构, 合并常数池, 去掉无用信息等来实现对Java类的高效压缩。由于是专门对Java类进行压缩的, 所以对普通文件的压缩和普通压缩软件没有什么两样, 但是对于Jar文件却能轻易达到10-40%的压缩率。这在Java应用部署中很有用, 尤其对于移动Java计算, 能够大大减小代码下载量。

Java5中还提供了这一技术的API接口, 你可以将其嵌入到你的程序中使用。使用的方法很简单, 下面的短短几行代码即可以实现jar的压缩和解压:

压缩

```
Packer packer=Pack200.newPacker();
OutputStream output=new BufferedOutputStream(new FileOutputStream(outfile));
packer.pack(new JarFile(jarFile), output);
output.close();
```

解压

```
Unpacker unpacker=Pack200.newUnpacker();
output=new JarOutputStream(new FileOutputStream(jarFile));
unpacker.unpack(pack200File, output);
output.close();
```

Pack200的压缩和解压缩速度是比较快的, 而且压缩率也是很惊人的, 在我是使用 的包4.46MB压缩后成了1.44MB (0.322%), 而且随着包的越大压缩率会根据明显, 据说如果jar包都是class类可以压缩到1/9的大小。其实JavaWebStart还有很多功能, 例如可以按不同的jar包进行lazy下载和单独更新, 设置可以根据jar中的类变动进行class粒度的下载。

但是在java11中废除了pack200以及unpack200工具以及java.util.jar中的Pack200 API。因为Pack200主要是用来压缩jar包的工具, 由于网络下载速度的提升以及java9引入模块化系统之后不再依赖Pack200, 因此这个版本将其移除掉。

JEP 332: Transport Layer Security (TLS) 1.3

实现TLS协议1.3版本。(TLS允许客户端和服务端通过互联网以一种防止窃听, 篡改以及消息伪造的方式进行通信)。

TLS 1.3是TLS协议的重大改进, 与以前的版本相比, 它提供了显着的安全性和性能改进。其他供应商的几个早期实现已经可用。我们需要支持TLS 1.3以保持竞争力并与最新标准保持同步。这个特性的实现动机和Unicode 10一样, 也是紧跟历史潮流。

JEP 328: Flight Recorder

提供一个低开销的, 为了排错Java应用问题, 以及JVM问题的数据收集框架, 希望达到的目标如下:

- 提供用于生产和消费数据作为事件的事件API;
- 提供缓存机制和二进制数据格式;
- 允许事件配置和事件过滤;
- 提供OS, JVM和JDK库的事件;

动机

排错, 监控, 性能分析是整个开发生命周期必不可少的一部分, 但是某些问题只会在大量真实数据压力下才会发生在生产环境。

Flight Recorder记录源自应用程序, JVM和OS的事件。事件存储在一个文件中, 该文件可以附加到错误报告中并由支持工程师进行检查, 允许事后分析导致问题的时期内的问题。工具可以使用API从记录文件中提取信息。

Flight Recorder的名字来源有点像来自于飞机的黑盒子，一种用来记录飞机飞行情况的仪器。而Flight Recorder就是记录Java程序运行情况的工具。

第三章 JDK 12新特性介绍

3.1 初识JDK12新特性

美国当地时间 2019 年 3 月 19 日，也就是北京时间 20 号 JDK12 正式发布了！

发行网站：

<http://openjdk.java.net/projects/jdk/12/>

详情：

Features

- 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)
- 230: Microbenchmark Suite
- 325: Switch Expressions (Preview)
- 334: JVM Constants API
- 340: One AArch64 Port, Not Two
- 341: Default CDS Archives
- 344: Abortable Mixed Collections for G1
- 346: Promptly Return Unused Committed Memory from G1

3.2 JDK 12更新列表

Features: 总共有8个新的JEP(JDK Enhancement Proposals)。

<http://openjdk.java.net/projects/jdk/12/>

189:Shenandoah:A Low-Pause-Time Garbage Collector(Experimental)

低暂停时间的GC

<http://openjdk.java.net/jeps/189>

230:Microbenchmark Suite

微基准测试套件

<http://openjdk.java.net/jeps/230>

325:Switch Expressions(Preview)

switch表达式

<http://openjdk.java.net/jeps/325>

334:JVM Constants API

JVM常量API

<http://openjdk.java.net/jeps/334>

340:One AArch64 Port,Not Two

只保留一个AArch64实现

<http://openjdk.java.net/jeps/340>

341:Default CDS Archives

默认类数据共享归档文件

<http://openjdk.java.net/jeps/341>

344:Abortable Mixed Collections for G1

可中止的G1 Mixed GC

<http://openjdk.java.net/jeps/344>

346:Promptly Return Unused Committed Memory from G1

G1及时返回未使用的已分配内存

<http://openjdk.java.net/jeps/346>

3.3 安装JDK 12

- 去官网下载JDK 12
- 下载地址: <https://www.oracle.com/technetwork/java/javase/downloads/index.html>
- 安装
- 启动IDEA创建一个模块集成JDK12

第四章 JDK 12新特性详解

4.1 switch表达式

JEP 325:Switch Expressions(Preview) : switch表达式

Java的switch语句是一个变化较大的语法（可能是因为Java的switch语句一直不够强大、熟悉swift或者js语言的同学可与swift的switch语句对比一下，就会发现Java的switch相对较弱），因为Java的很多版本都在不断地改进switch语句：

JDK 12扩展了switch语句，使其可以用作语句或者表达式，并且传统的和扩展的简化版switch都可以使用。JDK 12对于switch的增强主要在于简化书写形式，提升功能点。

下面简单回顾一下switch的进化阶段：

- 从Java 5+开始，Java的switch语句可使用枚举了。
- 从Java 7+开始，Java的switch语句支持使用String类型的变量和表达式了。
- 从Java 11+开始，Java的switch语句会自动对省略break导致的贯穿提示警告（以前需要使用-X:fallthrough选项才能显示出来）。
- 但从JDK12开始，Java的switch语句有了很大程度的增强。

JDK 12以前的switch程序

代码如下：

```
public class Demo01{
    public static void main(String[] args){
        // 声明变量score，并为其赋值为'C'
        var score = 'C';
    }
}
```



```
// 执行switch分支语句
switch (score) {
    case 'A':
        System.out.println("优秀");
        break;
    case 'B':
        System.out.println("良好");
        break;
    case 'C':
        System.out.println("中");
        break;
    case 'D':
        System.out.println("及格");
        break;
    case 'E':
        System.out.println("不及格");
        break;
    default:
        System.out.println("数据非法! ");
}
}
```

这是经典的Java 11以前的switch写法，这里不能忘记写break，否则switch就会贯穿、导致程序出现错误（JDK 11会提示警告）。

JDK 12不需要break了

在JDK 12之前如果switch忘记写break将导致贯穿，在JDK 12中对switch的这一贯穿性做了改进。你只要将case后面的冒号（:）改成箭头，那么你即使不写break也不会贯穿了，因此上面程序可改写如下形式：

```
public class Demo02{
    public static void main(String[] args){
        // 声明变量score, 并为其赋值为'C'
        var score = 'C';
        // 执行switch分支语句
        switch (score){
            case 'A' -> System.out.println("优秀");
            case 'B' -> System.out.println("良好");
            case 'C' -> System.out.println("中");
            case 'D' -> System.out.println("及格");
            case 'E' -> System.out.println("不及格");
            default -> System.out.println("成绩数据非法! ");
        }
    }
}
```

上面代码简洁很多了。

JDK 12的switch表达式

Java 12的switch甚至可作为表达式了——不再是单独的语句。例如如下程序。



```
public class Demo03
{
    public static void main(String[] args)
    {
        // 声明变量score, 并为其赋值为'C'
        var score = 'C';
        // 执行switch分支语句
        String s = switch (score)
        {
            case 'A' -> "优秀";
            case 'B' -> "良好";
            case 'C' -> "中";
            case 'D' -> "及格";
            case 'F' -> "不及格";
            default -> "成绩输入错误";
        };
        System.out.println(s);
    }
}
```

上面程序直接将switch表达式的值赋值给s变量，这样switch不再是一个语句，而是一个表达式。

JDK 12中switch的多值匹配

当你把switch中的case后的冒号改为箭头之后，此时switch就不会贯穿了，但在某些情况下，程序本来就希望贯穿比如我就希望两个case共用一个执行体！JDK 12的switch中的case也支持多值匹配，这样程序就变得更加简洁了。例如如下程序。

```
public class Demo04
{
    public static void main(String[] args)
    {
        // 声明变量score, 并为其赋值为'C'
        var score = 'B';
        // 执行switch分支语句
        String s = switch (score)
        {
            case 'A', 'B' -> "上等";
            case 'C' -> "中等";
            case 'D', 'E' -> "下等";
            default -> "成绩数据输入非法! ";
        };
        System.out.println(s);
    }
}
```

小结

从以上案例可以看出JDK 12对switch的功能做了很大的改进，代码也十分的简化，目前来看switch依然是不支持区间匹配的，未来是否可以支持，我们拭目以待。

4.2 微基准测试套件

JEP 230: Microbenchmark Suite 微基准测试套件

JMH是什么

JMH, 即Java Microbenchmark Harness, 是专门用于代码微基准测试的工具套件。何谓Micro Benchmark呢? 简单的来说就是基于方法层面的基准测试, 精度可以达到微秒级。当你希望进一步优化方法执行性能的时候, 就可以使用JMH对优化的结果进行量化的分析。

JMH比较典型的应用场景

- 想定量地知道某个方法需要执行多长时间, 以及执行时间和输入 n 的相关性
- 一个接口有两种不同实现(例如实现 A 使用了 `FixedThreadPool`, 实现 B 使用了 `ForkJoinPool`), 不知道哪种实现性能更好。

JMH的使用案例

如果你使用 maven 来管理你的 Java 项目的话, 引入 JMH 是一件很简单的事情——只需要在 `pom.xml` 里增加 JMH 的依赖即可

```
<properties>
  <jmh.version>1.14.1</jmh.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>${jmh.version}</version>
  </dependency>
  <dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>${jmh.version}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.0</version>
      <configuration>
        <release>12</release>
        <compilerArgs>--enable-preview</compilerArgs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

接下来再创建我们的第一个 Benchmark

```
@BenchmarkMode({Mode.AverageTime})
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State({Scope.Thread})
public class HelloWorld {

    @Benchmark
    public int testSleep01() {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return 0;
    }

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
            .include(HelloWorld.class.getSimpleName())
            .forks(1)
            .warmupIterations(5)
            .measurementIterations(5)
            .build();

        new Runner(opt).run();
    }
}
```

注解

@BenchmarkMode

对应Mode选项，可用于类或者方法上，需要注意的是，这个注解的value是一个数组，可以把几种Mode集合在一起执行，还可以设置为Mode.All，即全部执行一遍。

@State

类注解，JMH测试类必须使用@State注解，State定义了一个类实例的生命周期，可以类比Spring Bean的Scope。由于JMH允许多线程同时执行测试，不同的选项含义如下：

Scope.Thread：默认的State，每个测试线程分配一个实例； Scope.Benchmark：所有测试线程共享一个实例，用于测试有状态实例在多线程共享下的性能； Scope.Group：每个线程组共享一个实例；

@OutputTimeUnit

benchmark 结果所使用的时间单位，可用于类或者方法注解，使用java.util.concurrent.TimeUnit中的标准时间单位。

@Benchmark

方法注解，表示该方法是需要进行 benchmark 的对象。

启动项

```
public static void main(String[] args) throws RunnerException {  
    Options opt = new OptionsBuilder()  
        .include(HelloWorld.class.getSimpleName())  
        .forks(1)  
        .warmupIterations(5)  
        .measurementIterations(5)  
        .build();  
  
    new Runner(opt).run();  
}
```

`include(SimpleBenchmark.class.getSimpleName())`代表我要测试的是哪个类的方法

`exclude("xxx")`代表测试的时候需要排除xxxx方法

`forks(2)`指的是做2轮测试，在一轮测试无法得出最满意的结果时，可以多测几轮以便得出更全面的测试结果，而每一轮都是先预热，再正式计量。

`warmupIterations(5)`代表先预热5次

`measurementIterations(5)` 正式运行测试5次

参数详解

Mode

Mode 表示 JMH 进行 Benchmark 时所使用的模式。通常是测量的维度不同，或是测量的方式不同。目前 JMH 共有四种模式：

Throughput：吞吐量，一段时间内可执行的次数，每秒可执行次数

AverageTime：每次调用的平均耗时时间。

SampleTime：随机进行采样执行的时间，最后输出取样结果的分布

SingleShotTime：在每次执行中计算耗时，以上模式都是默认一次 iteration 是 1s，只有 SingleShotTime 是只运行一次。

Iteration

Iteration 是 JMH 进行测试的最小单位。在大部分模式下，一次 iteration 代表的是一秒，JMH 会在这一秒内不断调用需要 benchmark 的方法，然后根据模式对其采样，计算吞吐量，计算平均执行时间等。

Warmup

Warmup 是指在实际进行 benchmark 前先进行预热的行为。为什么需要预热？因为 JVM 的 JIT 机制的存在，如果某个函数被调用多次之后，JVM 会尝试将其编译成为机器码从而提高执行速度。为了让 benchmark 的结果更加接近真实情况就需要进行预热

测试结果如下

```
Run: HelloWorld x
# Warmup Iteration 4: 300.006 ms/op
# Warmup Iteration 5: 300.026 ms/op
Iteration 1: 300.017 ms/op
Iteration 2: 300.037 ms/op
Iteration 3: 300.005 ms/op
Iteration 4: 300.021 ms/op
Iteration 5: 300.207 ms/op

Result "testSleep01":
  300.057 ±(99.9%) 0.325 ms/op [Average]
  (min, avg, max) = (300.005, 300.057, 300.207), stdev = 0.084
  CI (99.9%): [299.732, 300.382] (assumes normal distribution)
```

JDK12中JMH说明

Java 12 中添加一套新的基本的微基准测试套件 (microbenchmarks suite) :

- 此功能为JDK源代码添加了一套微基准测试套件，简化了现有微基准测试的运行和新基准测试的创建过程。
- 使开发人员可以轻松运行现有的微基准测试并创建新的基准测试，其目标在于提供一个稳定且优化过的基准。它基于Java Microbenchmark Harness (JMH)，可以轻松测试JDK性能，支持JMH更新

4.3 默认生成类数据共享

JEP 341: Default CDS Archives 默认生成类数据共享

我们知道在同一个物理机上启动多个JVM时，如果每个虚拟机都单独装载自己需要的所有类，启动成本和内存占用是比较高的。所以Java团队引入了类数据共享机制 (Class Data Sharing，简称 CDS) 的概念，通过把一些核心类在每个JVM间共享，每个JVM只需要装载自己的应用类即可。

好处是：JVM启动时间减少了，另外核心类是共享的，所以JVM的内存占用也减少了。

Java12新特性

- JDK 12之前，想要利用CDS的用户，即使仅使用JDK中提供的默认类列表，也必须 `java -Xshare:dump` 作为额外的步骤来运行。
- Java 12 针对 64 位平台下的JDK 构建过程进行了增强改进，使其默认生成类数据共享 (CDS) 归档，以进一步达到改进应用程序的启动时间的目的。
- 同时也取消了用户必须手动运行：`java -Xshare:dump` 才能使用CDS的功能。

4.4 Shenandoah GC低停顿时间的GC(预览)

JEP 189:Shenandoah:A Low-Pause-Time Garbage Collector(Experimental) 低暂停时间的GC

目标

添加一个名为Shenandoah的新垃圾收集 (GC) 算法，通过与正在运行的Java线程同时进行疏散工作来减少GC暂停时间，最终目标旨在针对 JVM 上的内存回收实现低停顿的需求。

使用Shenandoah的暂停时间与堆大小无关，这意味着无论堆是200MB还是200GB，您都将具有相同的一致暂停时间。

与 ZGC 类似，Shenandoah GC 主要目标是 99.9% 的暂停小于 10ms，暂停与堆大小无关等

使用

```
-XX:+UnlockExperimentalVMOptions
```

在命令行中要求。作为实验性功能，Shenandoah构建系统会自动禁用不受支持的配置。

要启用/使用Shenandoah GC，需要以下JVM选项：

```
-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC
```

4.5 G1垃圾收集器功能增强

344:Abortable Mixed Collections for G1 可中止的G1 Mixed GC

如果G1垃圾收集器有可能超过预期的暂停时间，则可以使用终止选项。

该垃圾收集器设计的主要目标之一是满足用户设置的预期的 JVM 停顿时间，可以终止可选部分的回收已到达停顿时间的目标。

346:Promptly Return Unused Committed Memory from G1 G1及时返回未使用的已分配内存

如果应用程序活动非常低，G1应该在合理的时间段内释放未使用的Java堆内存。

G1可以使其能够在空闲时自动将Java堆内存返还给操作系统

4.6 JDK 12其他新特性

增加项：String新增方法

1. transform(Function)：对字符串进行处理后返回。

```
var rs = "itheima".transform(s -> s+"学习Java").transform(s -> s.toUpperCase());  
System.out.println(rs); // ITHEIMA学习JAVA
```

2. indent：该方法允许我们调整String实例的缩进

```
System.out.println("=====");  
String result = "Java\nMySQL\nMyBatis".indent(3);  
System.out.println(result);
```

执行输出结果：

```
=====
  Java
  MySQL
  MyBatis
```

新增项：Files新增mismatch方法

返回内容第一次不匹配的字符位置索引

```
Writer fw1 = new FileWriter("a.txt");
fw1.write("acc");
fw1.write("b");
fw1.write("c");
fw1.close();
Writer fw2 = new FileWriter("b.txt");
fw2.write("acc");
fw2.write("ddd");
fw2.write("ddd");
fw2.write("c");
fw2.close();
System.out.println(Files.mismatch(Path.of("a.txt"), Path.of("b.txt")));
```

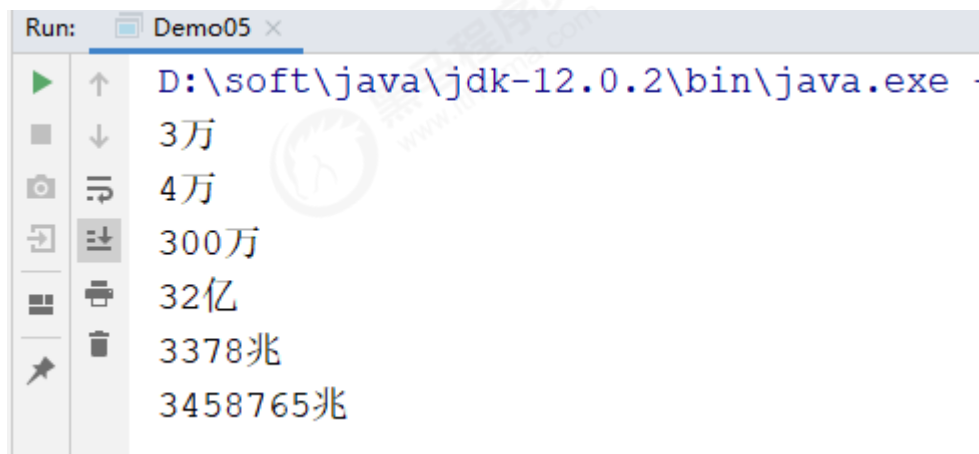
核心库java.text支持压缩数字格式

NumberFormat添加了对以紧凑形式格式化数字的支持。紧凑数字格式是指以简短或人类可读形式表示的数字。例如，在en_US语言环境中，1000可以格式化为“1K”，1000000可以格式化为“1M”，具体取决于指定的样式NumberFormat.Style。紧凑数字格式由LDML的Compact Number格式规范定义。要获取实例，请使用NumberFormat紧凑数字格式所给出的工厂方法之一。

```
//例如:
NumberFormat fmt = NumberFormat.getCompactNumberInstance(Locale.US,
NumberFormat.Style.SHORT);
String result = fmt.format(1000);
// 上面的例子导致“1K”。

var cnf = NumberFormat.getCompactNumberInstance(Locale.CHINA,
NumberFormat.Style.SHORT);
System.out.println(cnf.format(3_0000));
System.out.println(cnf.format(3_9200));
System.out.println(cnf.format(3_000_000));
System.out.println(cnf.format(3L << 30));
System.out.println(cnf.format(3L << 50));
System.out.println(cnf.format(3L << 60));
```

输出结果:



```
Run: Demo05 x
D:\soft\java\jdk-12.0.2\bin\java.exe
3万
4万
300万
32亿
3378兆
3458765兆
```

核心库java.lang中支持Unicode 11

JDK 12版本包括对Unicode 11.0.0的支持。在发布支持Unicode 10.0.0的JDK 11之后，Unicode 11.0.0引入了以下JDK 12中包含的新功能：

- 1、684个新角色
 - 1.1、66个表情符号字符
 - 1.2、Copyleft符号
 - 1.3、评级系统的半星
 - 1.4、额外的占星符号
 - 1.5、象棋中国象棋符号
- 2、11个新区块
 - 2.1、格鲁吉亚扩展
 - 2.2、玛雅数字
 - 2.3、印度Siyaa数字
 - 2.4、国际象棋符号
- 3、7个新脚本
 - 3.1、Hanifi Rohingya
 - 3.2、Old Sogdian
 - 3.3、Sogdian
 - 3.4、Dogra
 - 3.5、Gunjala Gondi
 - 3.6、Makasar
 - 3.7、Medefaidrin

移除项

- 移除com.sun.awt.SecurityWarnin;
- 移除FileInputStream、FileOutputStream、- Java.util.ZipFile/Inflator/Deflator的finalize方法;
- 移除GTE CyberTrust Global Root;
- 移除javac的-source, -target对6及1.6的支持，同时移除--release选项;

废弃项

- 废弃的API列表见deprecated-list
- 废弃-XX:+/-MonitorInUseLists选项
- 废弃Default Keytool的-keyalg值