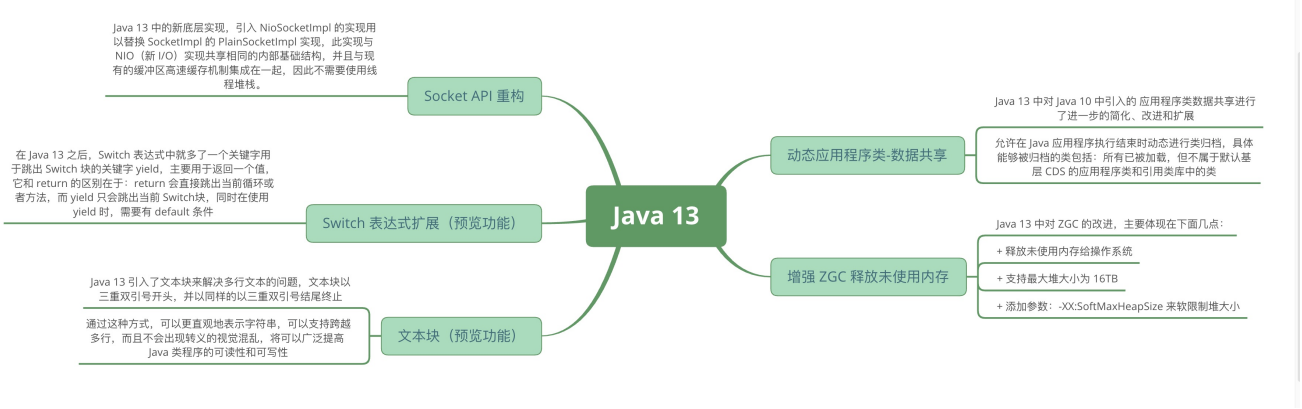
Java 13 新特性概述

知识体系



Java 13 已如期于 9 月 17 日正式发布，此次更新是继半年前 Java 12 这大版本发布之后的一次常规版本更新，在这一版中，主要带来了 ZGC 增强、更新 Socket 实现、Switch 表达式更新等方面的改动、增强。

动态应用程序类-数据共享

在 Java 10 中，为了改善应用启动时间和内存空间占用，通过使用 APP CDS，加大了 CDS 的使用范围，允许自定义的类加载器也可以加载自定义类给多个 JVM 共享使用，具体介绍可以参考 Java 10 新特性介绍一文详细介绍，在此就不再继续展开。

Java 13 中对 Java 10 中引入的 应用程序类数据共享进行了进一步的简化、改进和扩展，即：允许在 Java 应用程序执行结束时动态进行类归档，具体能够被归档的类包括：所有已被加载，但不属于默认基层 CDS 的应用程序类和引用类库中的类。通过这种改进，可以提高应用程序类-数据使用上的简易性，减少在使用类-数据存档中需要为应用程序创建类加载列表的必要，简化使用类-数据共享的步骤，以便更简单、便捷地使用 CDS 存档。

在 Java 中，如果要执行一个类，首先需要将类编译成对应的字节码文件，以下是 JVM 装载、执行等需要的一系列准备步骤：假设给定一个类名，JVM 将在磁盘上查找到该类对应的字节码文件，并将其进行加载，验证字节码文件，准备，解析，初始化，根据其内部数据结构加载到内存中。当然，这一连串的操作都需要一些时间，这在 JVM 启动并且需要加载至少几百个甚至是数千个类时，加载时间就尤其明显。

Java 10 中的 App CDS 主要是为了将不变的类数据，进行一次创建，然后存储到归档中，以便在应用重启之后可以对其进行内存映射而直接使用，同时也可以在运行的 JVM 实例之间共享使用。但是在 Java 10 中使用 App CDS 需要进行如下操作：

- 创建需要进行类归档的类列表
- 创建归档
- 使用归档方式启动

在使用归档文件启动时，JVM 将归档文件映射到其对应的内存中，其中包含所需的大多数类，而

需要使用多么复杂的类加载机制。甚至可以在并发运行的 JVM 实例之间共享内存区域，通过这种方式可以释放需要在每个 JVM 实例中创建相同信息时浪费的内存，从而节省了内存空间。

在 Java 12 中，默认开启了对 JDK 自带 JAR 包类的存档，如果想关闭对自带类库的存档，可以在启动参数中加上：

```
-Xshare:off
```

而在 Java 13 中，可以不用提供归档类列表，而是通过更简洁的方式来创建包含应用程序类的归档。具体可以使用参数 `-XX:ArchiveClassesAtExit` 来控制应用程序在退出时生成存档，也可以使用 `-XX:SharedArchiveFile` 来使用动态存档功能，详细使用见如下示例。

清单 1. 创建存档文件示例

```
$ java -XX:ArchiveClassesAtExit=helloworld.jsa -cp helloworld.jar Hello
```

清单 2. 使用存档文件示例

```
$ java -XX:SharedArchiveFile=hello.jsa -cp helloworld.jar Hello
```

上述就是在 Java 应用程序执行结束时动态进行类归档，并且在 Java 10 的基础上，将多条命令进行了简化，可以更加方便地使用类归档功能。

增强 ZGC 释放未使用内存

ZGC 是 Java 11 中引入的最为瞩目的垃圾回收特性，是一种可伸缩、低延迟的垃圾收集器，不过在 Java 11 中是实验性的引入，主要用来改善 GC 停顿时间，并支持几百 MB 至几个 TB 级别大小的堆，并且应用吞吐能力下降不会超过 15%，目前只支持 Linux/x64 位平台的这样一种新型垃圾收集器。

通过在实际中的使用，发现 ZGC 收集器中并没有像 Hotspot 中的 G1 和 Shenandoah 垃圾收集器一样，能够主动将未使用的内存释放给操作系统的功能。对于大多数应用程序来说，CPU 和内存都属于有限的紧缺资源，特别是现在使用的云上或者虚拟化环境中。如果应用程序中的内存长期处于空闲状态，并且还不能释放给操作系统，这样会导致其他需要内存的应用无法分配到需要的内存，而这边应用分配的内存还处于空闲状态，处于”忙的太忙，闲的太闲”的非公平状态，并且也容易导致基于虚拟化的环境中，因为这些实际并未使用的资源而多付费的情况。由此可见，将未使用内存释放给系统主内存是一项非常有用且亟需的功能。

ZGC 堆由一组称为 ZPages 的堆区域组成。在 GC 周期中清空 ZPages 区域时，它们将被释放并返回到页面缓存 ZPageCache 中，此缓存中的 ZPages 按最近最少使用（LRU）的顺序，并按照大小进行组织。在 Java 13 中，ZGC 将向操作系统返回被标识为长时间未使用的页面，这样它们将可以被其他进程重用。同时释放这些未使用的内存给操作系统不会导致堆大小缩小到参数设置的最小大小以下，如果将最小和最大堆大小设置为相同的值，则不会释放任何内存给操作系统。

Java 13 中对 ZGC 的改进，主要体现在下面几点：

- 释放未使用内存给操作系统
- 支持最大堆大小为 16TB
- 添加参数：-XX:SoftMaxHeapSize 来软限制堆大小

这里提到的是软限制堆大小，是指 GC 应努力是堆大小不要超过指定大小，但是如果实际需要，也还是允许 GC 将堆大小增加到超过 SoftMaxHeapSize 指定值。主要用在下面几种情况：当希望降低堆占用，同时保持应对堆空间临时增加的能力，亦或想保留充足内存空间，以能够应对内存分配，而不会因为内存分配意外增加而陷入分配停滞状态。不应将 SoftMaxHeapSize 设置为大于最大堆大小（-Xmx 的值，如果未在命令行上设置，则此标志应默认为最大堆大小。

Java 13 中，ZGC 内存释放功能，默认情况下是开启的，不过可以使用参数：-XX：-ZUncommit 显式关闭，同时如果将最小堆大小 (-Xms) 配置为等于最大堆大小 (-Xmx)，则将隐式禁用此功能。

还可以使用参数：`-XX：ZUncommitDelay = <seconds>`（默认值为 300 秒）来配置延迟释放，此延迟时间可以指定释放多长时间之前未使用的内存。

Socket API 重构

Java 中的 Socket API 已经存在了二十多年了，尽管这么多年来，一直在维护和更新中，但是在实际使用中遇到一些局限性，并且不容易维护和调试，所以要对其进行大修大改，才能跟得上现代技术的发展，毕竟二十多年来，技术都发生了深刻的变化。Java 13 为 Socket API 带来了新的底层实现方法，并且在 Java 13 中是默认使用新的 Socket 实现，使其易于发现并在排除问题同时增加可维护性。

Java Socket API（java.net.ServerSocket 和 java.net.Socket）包含允许监听控制服务器和发送数据的套接字对象。可以使用 ServerSocket 来监听连接请求的端口，一旦连接成功就返回一个 Socket 对象，可以使用该对象读取发送的数据和进行数据写回操作，而这些类的繁重工作都是依赖于 SocketImpl 的内部实现，服务器的发送和接收两端都基于 SOCKS 进行实现的。

在 Java 13 之前，通过使用 PlainSocketImpl 作为 SocketImpl 的具体实现。

Java 13 中的新底层实现，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。除了这些更改之外，还有其他一些更便利的更改，如使用 java.lang.ref.Cleaner 机制来关闭套接字（如果 SocketImpl 实现在尚未关闭的套接字上被进行了垃圾收集），以及在轮询时套接字处于非阻塞模式时处理超时操作等方面。

为了最小化在重新实现已使用二十多年的方法时出现问题的风险，在引入新实现方法的同时，之前版本的实现还未被移除，可以通过使用下列系统属性以重新使用原实现方法：

```
-Djdk.net.usePlainSocketImpl = true
```

另外需要注意的是，SocketImpl 是一种传统的 SPI 机制，同时也是一个抽象类，并未指定具体的实现，所以，新的实现方式尝试模拟未指定的行为，以达到与原有实现兼容的目的。但是，在使用新实现时，有些基本情况可能会失败，使用上述系统属性可以纠正遇到的问题，下面两个除外。

- 老版本中，PlainSocketImpl 中的 getInputStream() 和 getOutputStream() 方法返回的 InputStream 和 OutputStream 分别来自于其对应的扩展类型 FileInputStream 和 FileOutputStream，而这个在新版实现中则没有。
- 使用自定义或其它平台的 SocketImpl 的服务器套接字无法接受使用其他（自定义或其它平台）类型 SocketImpl 返回 Sockets 的连接。

通过这些更改，Java Socket API 将更易于维护，更好地维护将使套接字代码的可靠性得到改善。同时 NIO 实现也可以在基础层面完成，从而保持 Socket 和 ServerSocket 类层面上的不变。

Switch 表达式扩展（预览功能）

在 Java 12 中引入了 Switch 表达式作为预览特性，而在 Java 13 中对 Switch 表达式做了增强改进，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，而在此之前，想要在 Switch 中返回内容，还是比较麻烦的，只不过目前还处于预览状态。

在 Java 13 之后，Switch 表达式中就多了一个关键字用于跳出 Switch 块的关键字 yield，主要用于返回一个值，它和 return 的区别在于：return 会直接跳出当前循环或者方法，而 yield 只会跳出当前 Switch块，同时在使用 yield 时，需要有 default 条件。

在 Java 12 之前，传统 Switch 语句写法为：

清单 3. 传统形式

```
private static String getText(int number) {
    String result = "";
    switch (number) {
        case 1, 2:
            result = "one or two";
            break;
        case 3:
            result = "three";
            break;
        case 4, 5, 6:
            result = "four or five or six";
            break;
        default:
            result = "unknown";
            break;
    };
    return result;
}
```

在 Java 12 之后，关于 Switch 表达式的写法改进为如下：

清单 4. 标签简化形式

```
private static String getText(int number) {
    String result = switch (number) {
        case 1, 2 -> "one or two";
        case 3 -> "three";
        case 4, 5, 6 -> "four or five or six";
        default -> "unknown";
    };
    return result;
}
```

而在 Java 13 中，value break 语句不再被编译，而是用 yield 来进行值返回，上述写法被改为如下写法：

清单 5. yield 返回值形式

```
private static String getText(int number) {
    return switch (number) {
        case 1, 2:
            yield "one or two";
        case 3:
            yield "three";
        case 4, 5, 6:
            yield "four or five or six";
        default:
            yield "unknown";
    };
}
```

文本块（预览功能）

一直以来，Java 语言在定义字符串的方式是有限的，字符串需要以双引号开头，以双引号结尾，这导致字符串不能够多行使用，而是需要通过换行转义或者换行连接符等方式来变通支持多行，但这样会增加编辑工作量，同时也会导致所在代码段难以阅读、难以维护。

Java 13 引入了文本块来解决多行文本的问题，文本块以三重双引号开头，并以同样的以三重双引号结尾终止，它们之间的任何内容都被解释为字符串的一部分，包括换行符，避免了对大多数转义序列的需要，并且它仍然是普通的 java.lang.String 对象，文本块可以在 Java 中可以使用字符串文字的任何地方使用，而与编译后的代码没有区别，还增强了 Java 程序中的字符串可读性。并且通过这种方式，可以更直观地表示字符串，可以支持跨越多行，而且不会出现转义的视觉混乱，将可以广泛提高 Java 类程序的可读性和可写性。

在 Java 13 之前，多行字符串写法为：

清单 6. 多行字符串写法

```
String html ="<html>\n" +
             "   <body>\n" +
             "      <p>Hello, World</p>\n" +
             "   </body>\n" +
             "</html>\n";


 String json ="{\n" +
             "   \"name\":\"mkyong\",\n" +
             "   \"age\":38\n" +
             "}\n";
```

@pdai: 代码已经复制到剪贴板

在 Java 13 引入文本块之后，写法为：

清单 7. 多行文本块写法

```
String html = """
                <html>
                    <body>
                        <p>Hello, World</p>
                    </body>
                </html>
                """;
```

```
String json = """
              {
                  "name":"mkyong",
                  "age":38
              }
              """;
```

文本块是作为预览功能引入到 Java 13 中的，这意味着它们不包含在相关的 Java 语言规范中，这样做的好处是方便用户测试功能并提供反馈，后续更新可以根据反馈来改进功能，或者必要时甚至删除该功能，如果该功能立即成为 Java SE 标准的一部分，则进行更改将变得更加困难。重要的是要意识到预览功能不是 beta 形式。

由于预览功能不是规范的一部分，因此有必要为编译和运行时明确启用它们。需要使用下面两个命令行参数来启用预览功能：

清单 8. 启用预览功能

```
$ javac --enable-preview --release 13 Example.java
$ java --enable-preview Example
```

结束语

Java 在更新发布周期为每半年发布一次之后，在合并关键特性、快速得到开发者反馈等方面，做得越来越好。从 Java 11 到 Java 13，目前确实是严格保持半年更新的节奏。Java 13 版本的发布带来了些新特性和功能增强、性能提升和改进尝试，不过 Java 13 不是 LTS 版本，本文针对其中对使用人员影响重大的以及主要的特性做了介绍，如有兴趣，您可以自行下载相关代码，继续深入研究。