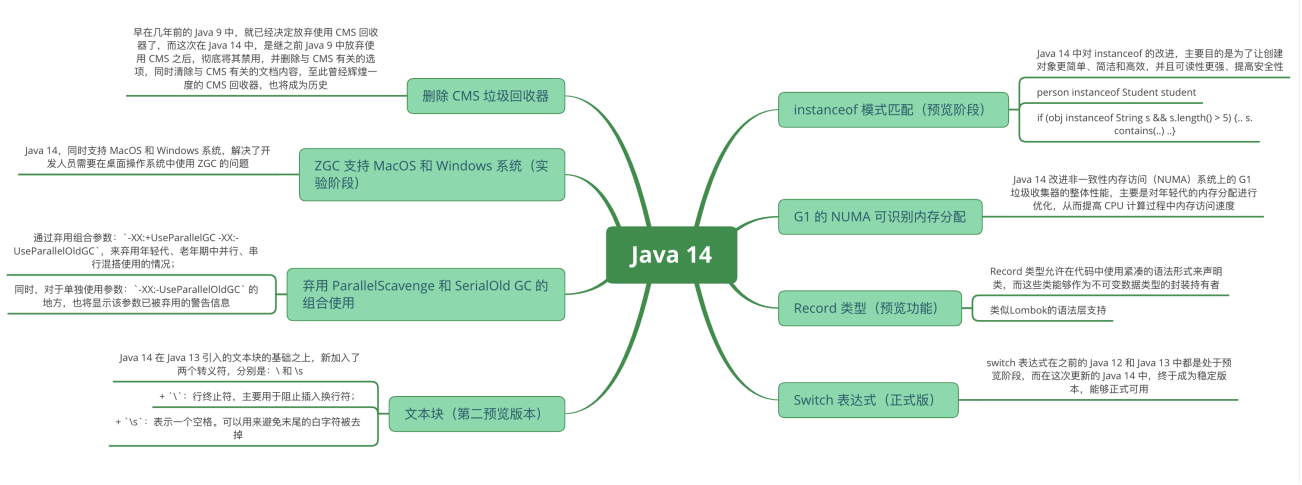
Java 14 新特性概述

知识体系



Java 14 已如期于 2020 年 3 月 17 日正式发布，此次更新是继半年前 Java 13 这大版本发布之后的又一次常规版本更新，即便在全球疫情如此严峻形势下，依然保持每六个月的版本更新频率，为大家及时带来改进和增强，这一点值得点赞。在这一版中，主要带来了 ZGC 增强、instanceof 增强、Switch 表达式更新为标准版等方面的改动、增强和新功能。

instanceof 模式匹配（预览阶段）

Java 14 中对 instanceof 的改进，主要目的是为了让创建对象更简单、简洁和高效，并且可读性更强、提高安全性。

在以往实际使用中，instanceof 主要用来检查对象的类型，然后根据类型对目标对象进行类型转换，之后进行不同的处理、实现不同的逻辑，具体可以参考清单 1：

清单 1. instanceof 传统使用方式

```
if (person instanceof Student) {
    Student student = (Student) person;
    student.say();
   // other student operations
} else if (person instanceof Teacher) {
    Teacher teacher = (Teacher) person;
    teacher.say();
    // other teacher operations
}
```

上述代码中，我们首先需要对 person 对象进行类型判断，判断 person 具体是 Student 还是 Teacher，因为这两种角色对应不同操作，亦即对应到的实际逻辑实现，判断完 person 类型之后，然后强制对 person 进行类型转换为局部变量，以方便后续执行属于该角色的特定操作。

上面这种写法，有下面两个问题：

- 每次在检查类型之后，都需要强制进行类型转换。
- 类型转换后，需要提前创建一个局部变量来接收转换后的结果，代码显得多余且繁琐。

Java 14 中，对 instanceof 进行模式匹配改进之后，上面示例代码可以改写成：

清单 2. instanceof 模式匹配使用方式

```
if (person instanceof Student student) {
    student.say();
   // other student operations
} else if (person instanceof Teacher teacher) {
    teacher.say();
    // other teacher operations
}
```

清单 2 中，首先在 if 代码块中，对 person 对象进行类型匹配，校验 person 对象是否为 Student 类型，如果类型匹配成功，则会转换为 Student 类型，并赋值给模式局部变量 student，并且只有当模式匹配表达式匹配成功是才会生效和复制，同时这里的 student 变量只能在 if 块中使用，而不能在 else if/else 中使用，否则会报编译错误。

注意，如果 if 条件中有 && 运算符时，当 instanceof 类型匹配成功，模式局部变量的作用范围也可以相应延长，如下面代码：

清单 3. Instanceof 模式匹配 && 方式

```
if (obj instanceof String s && s.length() > 5) {.. s.contains(..) ..}
```

另外，需要注意，这种作用范围延长，并不适用于或 || 运算符，因为即便 || 运算符左边的 instanceof 类型匹配没有成功也不会造成短路，依旧会执行到||运算符右边的表达式，但是此时，因为 instanceof 类型匹配没有成功，局部变量并未定义赋值，此时使用会产生问题。

与传统写法对比，可以发现模式匹配不但提高了程序的安全性、健壮性，另一方面，不需要显式的去进行二次类型转换，减少了大量不必要的强制类型转换。模式匹配变量在模式匹配成功之后，可以直接使用，同时它还被限制了作用范围，大大提高了程序的简洁性、可读性和安全性。instanceof 的模式匹配，为 Java 带来的有一次便捷的提升，能够剔除一些冗余的代码，写出更加简洁安全的代码，提高码代码效率。

G1 的 NUMA 可识别内存分配

Java 14 改进非一致性内存访问（NUMA）系统上的 G1 垃圾收集器的整体性能，主要是对年轻代的内存分配进行优化，从而提高 CPU 计算过程中内存访问速度。

NUMA 是 **non-unified memory access** 的缩写，主要是指在当前的多插槽物理计算机体系中，比较普遍是多核的处理器，并且越来越多的具有 NUMA 内存访问体系结构，即内存与每个插槽或内核之间的距离并不相等。同时套接字之间的内存访问具有不同的性能特征，对更远的套接字的访问通常具有更多的时间消耗。这样每个核对于每一块或者某一区域的内存访问速度会随着核和物理内存所在的位置的远近而有不同的时延差异。

Java 中，堆内存分配一般发生在线程运行的时候，当创建了一个新对象时，该线程会触发 G1 去分配一块内存出来，用来存放新创建的对象，在 G1 内存体系中，其实就是一块 region（大对象除外，大对象需要多个 region），在这个分配新内存的过程中，如果支持了 NUMA 感知内存分配，将会优先在与当前线程所绑定的 NUMA 节点空闲内存区域来执行 allocate 操作，同一线程创建的对象，尽可能的保留在年轻代的同一 NUMA 内存节点上，因为是基于同一个线程创建的对象大部分是短存活并且高概率互相调用的。

具体启用方式可以在 JVM 参数后面加上如下参数:

```
-XX:+UseNUMA
```

通过这种方式来启用可识别的内存分配方式，能够提高一些大型计算机的 G1 内存分配回收性能。 改进 NullPointerExceptions 提示信息

Java 14 改进 NullPointerException 的可查性、可读性，能更准确地定位 null 变量的信息。该特性能够帮助开发者和技术支持人员提高生产力，以及改进各种开发工具和调试工具的质量，能够更加准确、清楚地根据动态异常与程序代码相结合来理解程序。

相信每位开发者在实际编码过程中都遇到过 NullPointerException，每当遇到这种异常的时候，都需要根据打印出来的详细信息来分析、定位出现问题的原因，以在程序代码中规避或解决。例如，假设下面代码出现了一个 NullPointerException：

```
book.id = 99;
```

打印出来的 NullPointerException 信息如下：

清单 4. NullPointerException 信息

```
Exception in thread "main" java.lang.NullPointerException
    at Book.main(Book.java:5)
```

像上面这种异常，因为代码比较简单，并且异常信息中也打印出来了行号信息，开发者可以很快速定位到出现异常位置：book 为空而导致的 NullPointerException，而对于一些复杂或者嵌套的情况下出现 NullPointerException 时，仅根据打印出来的信息，很难判断实际出现问题的位置，具体见下面示例：

```
shoopingcart.buy.book.id = 99;
```

对于这种比较复杂的情况下，仅仅单根据异常信息中打印的行号，则比较难判断出现 NullPointerException 的原因。

而 Java 14 中，则做了对 NullPointerException 打印异常信息的改进增强，通过分析程序的字节码信息，能够做到准确的定位到出现 NullPointerException 的变量，并且根据实际源代码打印出详细异常信息，对于上述示例，打印信息如下：

清单 5. NullPointerException 详细信息

```
Exception in thread "main" java.lang.NullPointerException:
        Cannot assign field "book" because "shoopingcart.buy" is null
    at Book.main(Book.java:5)
```

对比可以看出，改进之后的 NullPointerException 信息，能够准确打印出具体哪个变量导致的 NullPointerException，减少了由于仅带行号的异常提示信息带来的困惑。该改进功能可以通过如下参数开启：

```
-XX:+ShowCodeDetailsInExceptionMessages
```

该增强改进特性，不仅适用于属性访问，还适用于方法调用、数组访问和赋值等有可能会导致 NullPointerException 的地方。

Record 类型（预览功能）

Java 14 富有建设性地将 Record 类型作为预览特性而引入。Record 类型允许在代码中使用紧凑的语法形式来声明类，而这些类能够作为不可变数据类型的封装持有者。Record 这一特性主要用在特定领域的类上；与枚举类型一样，Record 类型是一种受限形式的类型，主要用于存储、保存数据，并且没有其它额外自定义行为的场景下。

在以往开发过程中，被当作数据载体的类对象，在正确声明定义过程中，通常需要编写大量的无实际业务、重复性质的代码，其中包括：构造函数、属性调用、访问以及 equals() 、hashCode()、toString() 等方法，因此在 Java 14 中引入了 Record 类型，其效果有些类似 Lombok 的 @Data 注解、Kotlin 中的 data class，但是又不尽完全相同，它们的共同点都是类的部分或者全部可以直接在类头中定义、描述，并且这个类只用于存储数据而已。对于 Record 类型，具体可以用下面代码来说明：

清单 6. Record 类型定义

```
public record Person(String name, int age) {
    public static String address;

    public String getName() {
        return name;
    }
}
```

对上述代码进行编译，然后反编译之后可以看到如下结果：

清单 7. Record 类型反编译结果

```
public final class Person extends java.lang.Record {
    private final java.lang.String name;
    private final java.lang.String age;

    public Person(java.lang.String name, java.lang.String age) { /* compiled code */ }

    public java.lang.String getName() { /* compiled code */ }

    public java.lang.String toString() { /* compiled code */ }

    public final int hashCode() { /* compiled code */ }

    public final boolean equals(java.lang.Object o) { /* compiled code */ }

    public java.lang.String name() { /* compiled code */ }

    public java.lang.String age() { /* compiled code */ }
}
```

根据反编译结果，可以得出，当用 Record 来声明一个类时，该类将自动拥有下面特征：

- 拥有一个构造方法
- 获取成员属性值的方法：name()、age()
- hashCode() 方法和 euqals() 方法
- toString() 方法
- 类对象和属性被 final 关键字修饰，不能被继承，类的示例属性也都被 final 修饰，不能再被赋值使用。
- 还可以在 Record 声明的类中定义静态属性、方法和示例方法。注意，不能在 Record 声明的类中定义示例字段，类也不能声明为抽象类等。

可以看到，该预览特性提供了一种更为紧凑的语法来声明类，并且可以大幅减少定义类似数据类型时所需的重复性代码。

另外 Java 14 中为了引入 Record 这种新的类型，在 java.lang.Class 中引入了下面两个新方法：

清单 8. Record 新引入至 Class 中的方法

```
RecordComponent[] getRecordComponents()
boolean isRecord()
```

其中 getRecordComponents() 方法返回一组 java.lang.reflect.RecordComponent 对象组成的数组，java.lang.reflect.RecordComponent也是一个新引入类，该数组的元素与 Record 类中的组件相对应，其顺序与在记录声明中出现的顺序相同，可以从该数组中的每个 RecordComponent 中提取到组件信息，包括其名称、类型、泛型类型、注释及其访问方法。

而 isRecord() 方法，则返回所在类是否是 Record 类型，如果是，则返回 true。

Switch 表达式（正式版）

switch 表达式在之前的 Java 12 和 Java 13 中都是处于预览阶段，而在这次更新的 Java 14 中，终于成为稳定版本，能够正式可用。

switch 表达式带来的不仅仅是编码上的简洁、流畅，也精简了 switch 语句的使用方式，同时也兼容之前的 switch 语句的使用；之前使用 switch 语句时，在每个分支结束之前，往往都需要加上 break 关键字进行分支跳出，以防 switch 语句一直往后执行到整个 switch 语句结束，由此造成一些意想不到的问题。switch 语句一般使用冒号：来作为语句分支代码的开始，而 switch 表达式则提供了新的分支切换方式，即 -> 符号右则表达式方法体在执行完分支方法之后，自动结束 switch 分支，同时 -> 右则方法块中可以是表达式、代码块或者是手动抛出的异常。以往的 switch 语句写法如下：

清单 9. Switch 语句

```
int dayOfWeek;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        dayOfWeek = 6;
        break;
    case TUESDAY:
        dayOfWeek = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        dayOfWeek = 8;
        break;
    case WEDNESDAY:
        dayOfWeek = 9;
        break;
    default:
        dayOfWeek = 0;
        break;
}
```

而现在 Java 14 可以使用 switch 表达式正式版之后，上面语句可以转换为下列写法：

清单 10. Switch 表达式

```
int dayOfWeek = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
case WEDNESDAY              -> 9;
    default             -> 0;

};
```

很明显，switch 表达式将之前 switch 语句从编码方式上简化了不少，但是还是需要注意下面几点：

- 需要保持与之前 switch 语句同样的 case 分支情况。
- 之前需要用变量来接收返回值，而现在直接使用 yield 关键字来返回 case 分支需要返回的结果。
- 现在的 switch 表达式中不再需要显式地使用 return、break 或者 continue 来跳出当前分支。
- 现在不需要像之前一样，在每个分支结束之前加上 break 关键字来结束当前分支，如果不加，则会默认往后执行，直到遇到 break 关键字或者整个 switch 语句结束，在 Java 14 表达式中，表达式默认执行完之后自动跳出，不会继续往后执行。
- 对于多个相同的 case 方法块，可以将 case 条件并列，而不需要像之前一样，通过每个 case 后面故意不加 break 关键字来使用相同方法块。

使用 switch 表达式来替换之前的 switch 语句，确实精简了不少代码，提高了编码效率，同时也可以规避一些可能由于不太经意而出现的意想不到的情况，可见 Java 在提高使用者编码效率、编码体验和简化使用方面一直在不停的努力中，同时也期待未来有更多的类似 lambda、switch 表达式这样的新特性出来。

删除 CMS 垃圾回收器

CMS 是老年代垃圾回收算法，通过标记-清除的方式进行内存回收，在内存回收过程中能够与用户线程并行执行。CMS 回收器可以与 Serial 回收器和 Parallel New 回收器搭配使用，CMS 主要通过并发的方式，适当减少系统的吞吐量以达到追求响应速度的目的，比较适合在追求 GC 速度的服务器上使用。

因为 CMS 回收算法在进行 GC 回收内存过程中是使用并行方式进行的，如果服务器 CPU 核数不多的情况下，进行 CMS 垃圾回收有可能造成比较高的负载。同时在 CMS 并行标记和并行清理时，应用线程还在继续运行，程序在运行过程中自然会创建新对象、释放不用对象，所以在这个过程中，会有新的不可达内存地址产生，而这部分的不可达内存是出现在标记过程结束之后，本轮 CMS 回收无法在周期内将它们回收掉，只能留在下次垃圾回收周期再清理掉。这样的垃圾就叫做浮动垃圾。由于垃圾收集和用户线程是并发执行的，因此 CMS 回收器不能像其他回收器那样进行内存回收，需要预留一些空间用来保存用户新创建的对象。由于 CMS 回收器在老年代中使用标记-清除的内存回收策略，势必会产生内存碎片，内存当碎片过多时，将会给大对象分配带来麻烦，往往会出现老年代还有空间但不能再保存对象的情况。

所以，早在几年前的 Java 9 中，就已经决定放弃使用 CMS 回收器了，而这次在 Java 14 中，是继之前 Java 9 中放弃使用 CMS 之后，彻底将其禁用，并删除与 CMS 有关的选项，同时清除与 CMS 有关的文档内容，至此曾经辉煌一度的 CMS 回收器，也将成为历史。

当在 Java 14 版本中，通过使用参数：`-XX:+UseConcMarkSweepGC`，尝试使用 CMS 时，将会收到下面信息：

```
Java HotSpot(TM) 64-Bit Server VM warning: Ignoring option UseConcMarkSweepGC; \
support was removed in <version>
```

ZGC 支持 MacOS 和 Windows 系统（实验阶段）

ZGC 是最初在 Java 11 中引入，同时在后续几个版本中，不断进行改进的一款基于内存 Region，同时使用了内存读屏障、染色指针和内存多重映射等技，并且以可伸缩、低延迟为目标的内存垃圾回收器器，不过在 Java 14 之前版本中，仅仅只支持在 Linux/x64 位平台。

此次 Java 14，同时支持 MacOS 和 Windows 系统，解决了开发人员需要在桌面操作系统中使用 ZGC 的问题。

在 MacOS 和 Windows 下面开启 ZGC 的方式，需要添加如下 JVM 参数：

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

弃用 ParallelScavenge 和 SerialOld GC 的组合使用

由于 Parallel Scavenge 和 Serial Old 垃圾收集算法组合起来使用的情况比较少，并且在年轻代中使用并行算法，而在老年代中使用串行算法，这种并行、串行混搭使用的情况，本身已属罕见同时也很冒险。由于这两 GC 算法组合很少使用，却要花费巨大工作量来进行维护，所以在 Java 14 版本中，考虑将这两 GC 的组合弃用。

具体弃用情况如下，通过弃用组合参数：`-XX:+UseParallelGC -XX:-UseParallelOldGC`，来弃用年轻代、老年期中并行、串行混搭使用的情况；同时，对于单独使用参数：`-XX:-UseParallelOldGC` 的地方，也将显示该参数已被弃用的警告信息。

文本块（第二预览版本）

Java 13 引入了文本块来解决多行文本的问题，文本块主要以三重双引号开头，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文本块字符串的一部分，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 中能够使用字符串的任何地方进行使用，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。

Java 14 在 Java 13 引入的文本块的基础之上，新加入了两个转义符，分别是：\ 和 \s，这两个转义符分别表达涵义如下：

- `\`：行终止符，主要用于阻止插入换行符；
- `\s`：表示一个空格。可以用来避免末尾的白字符被去掉。

在 Java 13 之前，多行字符串写法为：

清单 11. 多行字符串写法

```
String literal = "Lorem ipsum dolor sit amet, consectetur adipiscing " +
                 "elit, sed do eiusmod tempor incididunt ut labore " +
                 "et dolore magna aliqua.";
```

在 Java 14 新引入两个转义符之后，上述内容可以写为：

清单 12. 多行文本块加上转义符的写法

```
String text = """
              Lorem ipsum dolor sit amet, consectetur adipiscing \
              elit, sed do eiusmod tempor incididunt ut labore \
              et dolore magna aliqua.\
              """;
```

上述两种写法，text 实际还是只有一行内容。

对于转义符：`\s`，用法如下，能够保证下列文本每行正好都是六个字符长度：

清单 13. 多行文本块加上转义符的写法

```
String colors = """
              red  \s
             green\s
              blue \s
              """;
```

Java 14 带来的这两个转义符，能够简化跨多行字符串编码问题，通过转义符，能够避免对换行等特殊字符串进行转移，从而简化代码编写，同时也增强了使用 String 来表达 HTML、XML、SQL 或 JSON 等格式字符串的编码可读性，且易于维护。

同时 Java 14 还对 String 进行了方法扩展：

- `stripIndent()`：用于从文本块中去除空白字符
- `translateEscapes()`：用于翻译转义字符
- `formatted(Object... args)`：用于格式化

结束语

Java 在更新版本周期为每半年发布一次之后，目前来看，确实是严格保持每半年更新的节奏。Java 14 版本的发布带来了不少新特性、功能实用性的增强、性能提升和 GC 方面的改进尝试。