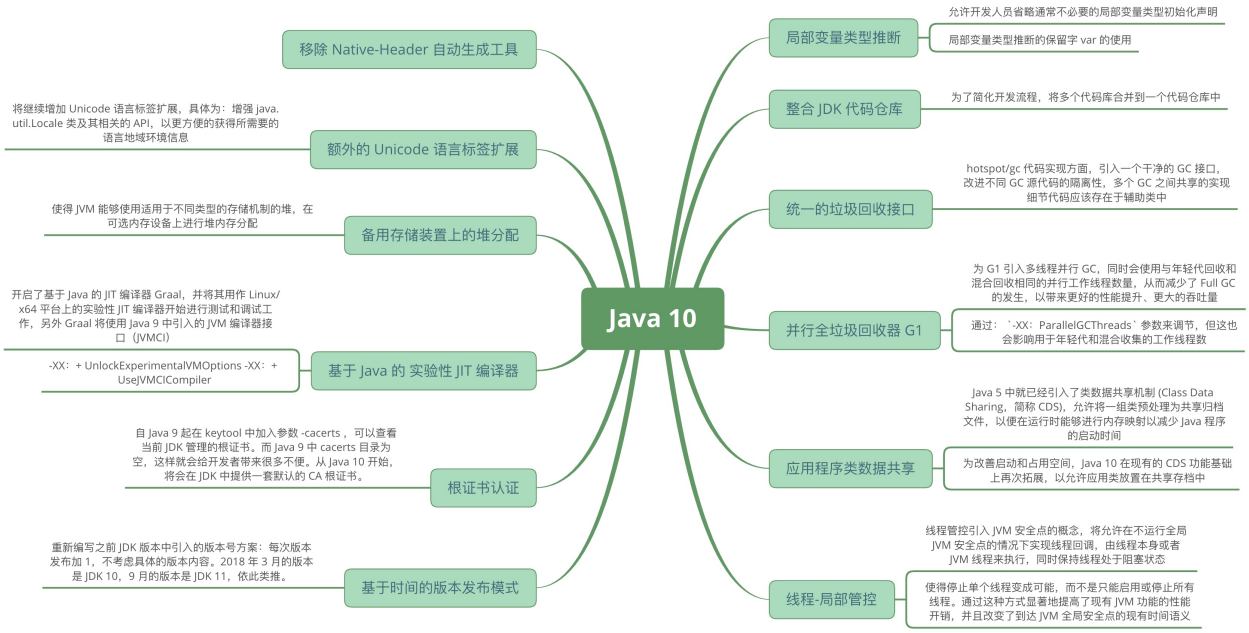
Java 10 新特性概述

知识体系



作为当今使用最广泛的编程语言之一的 Java 在 2018 年 3 月 21 日发布了第十个大版本。为了更快地迭代、更好地跟进社区反馈，Java 语言版本发布周期调整为每隔 6 个月发布一次。Java 10 是这一新规则之后，采用新发布周期的第一个大版本。Java 10 版本带来了很多新特性，其中最备受广大开发者关注的莫过于局部变量类型推断。除此之外，还有其他包括垃圾收集器改善、GC 改进、性能提升、线程管控等一批新特性。

局部变量类型推断

局部变量类型推断是 Java 10 中最值得开发人员注意的新特性，这是 Java 语言开发人员为了简化 Java 应用程序的编写而进行的又一重要改进。

这一新功能将为 Java 增加一些新语法，允许开发人员省略通常不必要的局部变量类型初始化声明。新的语法将减少 Java 代码的冗长度，同时保持对静态类型安全性的承诺。局部变量类型推断主要是向 Java 语法中引入在其他语言（比如 C#、JavaScript）中很常见的保留类型名称 var 。但需要特别注意的是： var 不是一个关键字，而是一个保留字。只要编译器可以推断此种类型，开发人员不再需要专门声明一个局部变量的类型，也就是可以随意定义变量而不必指定变量的类型。这种改进对于链式表达式来说，也会很方便。以下是一个简单的例子：

清单 1. 局部变量类型推断示例

```
var list = new ArrayList<String>(); // ArrayList<String>
var stream = list.stream(); // Stream<String>
```

看着是不是有点 JS 的感觉？有没有感觉越来越像 JS 了？虽然变量类型的推断在 Java 中不是一个崭新的概念，但在局部变量中确是很大的一个改进。说到变量类型推断，从 Java 5 中引进泛型，到 Java 7 的 <> 操作符允许不绑定类型而初始化 List，再到 Java 8 中的 Lambda 表达式，再到现在 Java 10 中引入的局部变量类型推断，Java 类型推断正大刀阔斧地向前进步、发展。

而上面这段例子，在以前版本的 Java 语法中初始化列表的写法为：

清单 2. Java 类型初始化示例

```
List<String> list = new ArrayList<String>();
Stream<String> stream = getStream();
```

在运算符允许在没有绑定 `ArrayList <>` 的类型的情况下初始化列表的写法为：

清单 3. Java 7 之后版本类型初始化示例

```
List<String> list = new LinkedList<>();
Stream<String> stream = getStream();
```

但这种 var 变量类型推断的使用也有局限性，仅局限于具有初始化器的局部变量、增强型 for 循环中的索引变量以及在传统 for 循环中声明的局部变量，而不能用于推断方法的参数类型，不能用于构造函数参数类型推断，不能用于推断方法返回类型，也不能用于字段类型推断，同时还不能用于捕获表达式（或任何其他类型的变量声明）。

不过对于开发者而言，变量类型显式声明会提供更加全面的程序语言信息，对于理解和维护代码有很大的帮助。Java 10 中新引入的局部变量类型推断能够帮助我们快速编写更加简洁的代码，但是局部变量类型推断的保留字 var 的使用势必会引起变量类型可视化缺失，并不是任何时候使用 var 都能容易、清晰的分辨出变量的类型。一旦 var 被广泛运用，开发者在没有 IDE 的支持下阅读代码，势必会对理解程序的执行流程带来一定的困难。所以还是建议尽量显式定义变量类型，在保持代码简洁的同时，也需要兼顾程序的易读性、可维护性。

整合 JDK 代码仓库

为了简化开发流程，Java 10 中会将多个代码库合并到一个代码仓库中。

在已发布的 Java 版本中，JDK 的整套代码根据不同功能已被分别存储在多个 Mercurial 存储库，这八个 Mercurial 存储库分别是：root、corba、hotspot、jaxp、jaxws、jdk、langtools、nashorn。

虽然以上八个存储库之间相互独立以保持各组件代码清晰分离，但同时管理这些存储库存在许多缺点，并且无法进行相关联源代码的管理操作。其中最重要的一点是，涉及多个存储库的变更集无法进行原子提交 （atomic commit）。例如，如果一个 bug 修复时需要对独立存储两个不同代码库的代码进行更改，那么必须创建两个提交：每个存储库中各一个。这种不连续性很容易降低项目和源代码管理工具的可跟踪性和加大复杂性。特别是，不可能跨越相互依赖的变更集的存储库执行原子提交这种多次跨仓库的变化是常见现象。

为了解决这个问题，JDK 10 中将所有现有存储库合并到一个 Mercurial 存储库中。这种合并的一个次生效应是，单一的 Mercurial 存储库比现有的八个存储库要更容易地被镜像(作为一个 Git 存储库)，并且使得跨越相互依赖的变更集的存储库运行原子提交成为可能，从而简化开发和管理过程。虽然在整合过程中，外部开发人员有一些阻力，但是 JDK 开发团队已经使这一更改成为 JDK 10 的一部分。

统一的垃圾回收接口

在当前的 Java 结构中，组成垃圾回收器（GC）实现的组件分散在代码库的各个部分。尽管这些惯例对于使用 GC 计划的 JDK 开发者来说比较熟悉，但对新的开发人员来说，对于在哪里查找特定 GC 的源代码，或者实现一个新的垃圾收集器常常会感到困惑。更重要的是，随着 Java modules 的出现，我们希望在构建过程中排除不需要的 GC，但是当前 GC 接口的横向结构会给排除、定位问题带来困难。

为解决此问题，需要整合并清理 GC 接口，以便更容易地实现新的 GC，并更好地维护现有的 GC。Java 10 中，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，多个 GC 之间共享的实现细节代码应该存在于辅助类中。这种方式提供了足够的灵活性来实现全新 GC 接口，同时允许以混合搭配方式重复使用现有代码，并且能够保持代码更加干净、整洁，便于排查收集器问题。

并行全垃圾回收器 G1

大家如果接触过 Java 性能调优工作，应该会知道，调优的最终目标是通过参数设置来达到快速、低延时的内存垃圾回收以提高应用吞吐量，尽可能的避免因内存回收不及时而触发的完整 GC（Full GC 会带来应用出现卡顿）。

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。

应用程序类数据共享

在 Java 5 中就已经引入了类数据共享机制 (Class Data Sharing，简称 CDS)，允许将一组类预处理为共享归档文件，以便在运行时能够进行内存映射以减少 Java 程序的启动时间，当多个 Java 虚拟机（JVM）共享相同的归档文件时，还可以减少动态内存的占用量，同时减少多个虚拟机在同一个物理或虚拟的机器上运行时的资源占用。简单来说，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。

CDS 特性在原来的 bootstrap 类基础之上，扩展加入了应用类的 CDS (Application Class-Data Sharing) 支持。

其**原理**为：在启动时记录加载类的过程，写入到文本文件中，再次启动时直接读取此启动文本并加载。设想如果应用环境没有大的变化，启动速度就会得到提升。

可以想像为类似于操作系统的休眠过程，合上电脑时把当前应用环境写入磁盘，再次使用时就可以快速恢复环境。

对大型企业应用程序的内存使用情况的分析表明，此类应用程序通常会将数以万计的类加载到应用程序类加载器中，如果能够将 AppCDS 应用于这些应用，将为每个 JVM 进程节省数十乃至数百兆字节的内存。另外对于云平台上的微服务分析表明，许多服务器在启动时会加载数千个应用程序类，AppCDS 可以让这些服务快速启动并改善整个系统响应时间。

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。

移除 Native-Header 自动生成工具

自 Java 9 以来便开始了一些对 JDK 的调整，用户每次调用 javah 工具时会被警告该工具在未来的版本中将会执行的删除操作。当编译 JNI 代码时，已不再需要单独的 Native-Header 工具来生成头文件，因为这可以通过 Java 8（JDK-7150368）中添加的 javac 来完成。在未来的某一时刻，JNI 将会被 Panama 项目的结果取代，但是何时发生还没有具体时间表。

额外的 Unicode 语言标签扩展

自 Java 7 开始支持 BCP 47 语言标记以来， JDK 中便增加了与日历和数字相关的 Unicode 区域设置扩展，在 Java 9 中，新增支持 ca 和 nu 两种语言标签扩展。而在 Java 10 中将继续增加 Unicode 语言标签扩展，具体为：增强 java.util.Locale 类及其相关的 API，以更方便的获得所需要的语言地域环境信息。同时在这次升级中还带来了如下扩展支持：

表 1.Unicode 扩展表

编码	注释
cu	货币类型
fw	一周的第一天
rg	区域覆盖
tz	时区

如 Java 10 加入的一个方法：

清单 4. Unicode 语言标签扩展示例

```
java.time.format.DateTimeFormatter::localizedBy
```

通过这个方法，可以采用某种数字样式，区域定义或者时区来获得时间信息所需的语言地域本地环境信息。

备用存储装置上的堆分配

硬件技术在持续进化，现在可以使用与传统 DRAM 具有相同接口和类似性能特点的非易失性 RAM。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。

一些操作系统中已经通过文件系统提供了使用非 DRAM 内存的方法。例如：NTFS DAX 模式和 ext4 DAX。这些文件系统中的内存映射文件可绕过页面缓存并提供虚拟内存与设备物理内存的相互映射。与 DRAM 相比，NV-DIMM 可能具有更高的访问延迟，低优先级进程可以为堆使用 NV-DIMM 内存，允许高优先级进程使用更多 DRAM。

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。

基于 Java 的 实验性 JIT 编译器

Java 10 中开启了基于 Java 的 JIT 编译器 Graal，并将其用作 Linux/x64 平台上的实验性 JIT 编译器开始进行测试和调试工作，另外 Graal 将使用 Java 9 中引入的 JVM 编译器接口（JVMCI）。

Graal 是一个以 Java 为主要编程语言、面向 Java bytecode 的编译器。与用 C++实现的 C1 及 C2 相比，它的模块化更加明显，也更加容易维护。Graal 既可以作为动态编译器，在运行时编译热点方法；亦可以作为静态编译器，实现 AOT 编译。在 Java 10 中，Graal 作为试验性 JIT 编译器一同发布（JEP 317）。将 Graal 编译器研究项目引入到 Java 中，或许能够为 JVM 性能与当前 C++ 所写版本匹敌（或有幸超越）提供基础。

Java 10 中默认情况下 HotSpot 仍使用的是 C2 编译器，要启用 Graal 作为 JIT 编译器，请在 Java 命令行上使用以下参数：

清单 5. 启用 Graal 为 JIT 编译器示例

```
-XX：+ UnlockExperimentalVMOptions -XX：+ UseJVMCICompiler
```

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。

作为 JDK 一部分的 cacerts 密钥库旨在包含一组能够用于在各种安全协议的证书链中建立信任的根证书。但是，JDK 源代码中的 cacerts 密钥库至目前为止一直是空的。因此，在 JDK 构建中，默认情况下，关键安全组件（如 TLS）是不起作用的。要解决此问题，用户必须使用一组根证书配置和 cacerts 密钥库下的 CA 根证书。

基于时间的版本发布模式

虽然 JEP 223 中引入的版本字符串方案较以往有了显著的改进。但是，该方案并不适合以后严格按照六个月的节奏来发布 Java 新版本的这种情况。

按照 JEP 223 的语义中，每个基于 JDK 构建或使用组件的开发者（包括 JDK 的发布者）都必须提前敲定版本号，然后切换过去。开发人员则必须在代码中修改检查版本号的相关代码，这对所有参与者来说都很尴尬和混乱。

Java 10 中将重新编写之前 JDK 版本中引入的版本号方案，将使用基于时间模型定义的版本号格式来定义新版本。保留与 JEP 223 版本字符串方案的兼容性，同时也允许除当前模型以外的基于时间的发布模型。使开发人员或终端用户能够轻松找出版本的发布时间，以便开发人员能够判断是否将其升级到具有最新安全修补程序或可能的附加功能的新版本。

Oracle Java 平台组的首席架构师 Mark Reinhold 在博客上介绍了有关 Java 未来版本的一些想法（你能接受 Java 9 的下一个版本是 Java 18.3 吗？）。他提到，Java 计划按照时间来发布，每半年一个版本，而不是像之前那样按照重要特性来确定大版本，如果某个大的特性因故延期，这个版本可能一拖再拖。

当时，Mark 也提出来一种基于时间命名版本号的机制，比如下一个将于 2018 年 3 月发布的版本，就是 18.3，再下一个版本是 18.9，以后版本依此类推。

不过经过讨论，考虑和之前版本号的兼容等问题，最终选择的命名机制是：

```
$FEATURE.$INTERIM.$UPDATE.$PATCH
```

$FEATURE：每次版本发布加 1，不考虑具体的版本内容。2018 年 3 月的版本是 JDK 10，9 月的版本是 JDK 11，依此类推。 $INTERIM：中间版本号，在大版本中间发布的，包含问题修复和增强的版本，不会引入非兼容性修改。

结束语

尽管距离 Java 9 发布仅有六个月的时间，Java 10 的发布也带来了不少新特性和功能增强，以上只是针对其中对开发人员影响重大的主要的一些特性做了介绍，同时也希望下一个 Java 版本能够带来更多、更大的变化。