前言

目前项目已经从JDK8直接升级到Open JDK11，因为是两个LTS版本跳跃，所以JDK9、JDK10以及JDK11的部分新功能都可以尝试着使用。JDK9共有91个JEP，JDK10包含12个JEP，JDK11包含17个JEP。本文未详尽描述全部的JEP，都是个人认为价值较高且需要了解的新特性。 --------
----刘一达

一、Java平台模块化系统（Jigsaw项目）

JDK9最耀眼的新特性就是Java平台模块化系统（JPMS，Java Platform Module System），通过Jigsaw项目实施。Jigsaw项目是Java发展过程的一个巨大里程碑，Java模块系统对Java系统产生非常深远的影响。与JDK的函数式编程和 Lamda表达式存在本质不同 ，Java模块系统是对整个Java生态系统做出的改变。

同时也是JDK7到JDK9的第一跳票王。。Jigsaw项目本计划于在2010年伴随着JDK7发布，随着Sun公司的没落及Oracle公司的接手，Jigsaw项目从JDK7一直跳票到JDK9才发布。前后经历了前后将近10年的时间。即使在2017JDK9发布前夕，Jigsaw项目还是差点胎死腹中。原因是以IBM和Redhat为首的13家企业在JCP委员会上一手否决了Jigsaw项目作为Java模块化规范进入JDK9发布范围的规划。原因无非就是IBM希望为自己的OSGI技术在Java模块化规范中争取一席之地。但是Oracle公司没有任何的退让，不惜向JCP发去公开信，声称如果Jigsaw提案无法通过，那么Oracle将直接自己开发带有Jigsaw项目的java新版本。经历了前后6次投票，最终JDK9还是带着Jigsaw项目最终发布了。但是，令人失望的是，Java模块化规范中还是给Maven、Gradle和OSGI等项目保留了一席之地。对于用户来说，想要实现完整模块化项目，必须使用多个技术相互合作，还是增加了复杂性。

1、什么是Java模块化?

简单理解，Java模块化就是将目前多个包（package）组成一个封装体，这个封装体有它的逻辑含义，同时也存在具体实例。同时模块遵循以下三个核心原则：

1. **强封装性**：一个模块可以选择性的对其他模块隐藏部分实现细节。
2. **定义良好的接口**：一个模块只有封装是不够的，还要通过对外暴露接口与其他模块交互。因此，暴露的接口必须有良好的定义。
3. **显示依赖**：一个模块通常需要协同其他模块一起工作，该模块必须显示的依赖其他模块 ，这些依赖关系同时也是模块定义的一部分。

2、为什么要做模块化?

模块化是分而治之的一个重要实践机制，微服务、OSGI和DDD都可以看到模块化思想的影子。现在很多大型的Java项目都是通过maven或者gradle进行版本管理和项目构建，模块的概念在Maven和gradle中早就存在，两者的不同下文也会说到。现在让我们一起回顾一下目前在使用JDK搭建复杂项目时遇到的一些问题：

2.1 如何使得Java SE应用程序更加轻量级的部署?

java包的本质只不过是类的限定名。jar包的本质就是将一组类组合到一起。一旦将多个Jar包放入ClassPath，最终得到只不过是一大堆文件而已。如何维护这么庞大的文件结构？目前最有效的方式，也是只能依赖mave或者gradle等项目构建工具。那最底层的Java平台的Jar包如何维护？如果我只是想部署一个简答的 helloworld应用，我需要一个JRE和一个用户编译的Jar包，并将这个Jar包放到classpath中去。JDK9以前，JRE的运行依赖我们的核心java类库-rt.jar。rt.jar是一个开箱即用的全量java类库，要么不使用，要么使用全部。直到JDK8，rt.jar的大小为60M，随着JDK的持续发展，这个包必然会越来越大。而且全量的java类库，给JRE也带来了额外的性能损耗。Java应用程序如果能选择性的加载rt.jar中的文件该多好?

2.2 在暴露的JAR包中，如何隐藏部分API和类型?

在使用Dubbo等RPC框架中，provider需要提供调用的接口定义Jar包，在该Jar包中包含一个共该Jar包内部使用的常量聚合类Constannt，放在constant包内。如何才能暴露JAR包的同时，隐藏常量聚合类Constant?

2.3 一直遭受NoClassDefFoundError的折磨

通过什么方式，可以知道一个Jar包依赖了哪些其他的 Jar包？JDK本身目前没有提供，可以通过Maven工具完成。那为什么不让Java平台自身就提供这些功能?

4、JPMS如何解决现有问题?

JPMS具有两个重要的目标：

1. **强封装（Strong encapsulation)**: 每一个模块都可以声明了哪些包是对外暴露的，java编译和运行时就可以实施这些规则来确保外部模块无法使用内部类型。
2. **可靠配置（Reliable configuration）**：每一模块都声明了哪些是它所需的，那么在运行时就可以检查它所需的所有模块在应用启动运行前是否都有。

Java平台本身就是必须要进行模块化改造的复杂项目，通过Jigsaw项目落地。

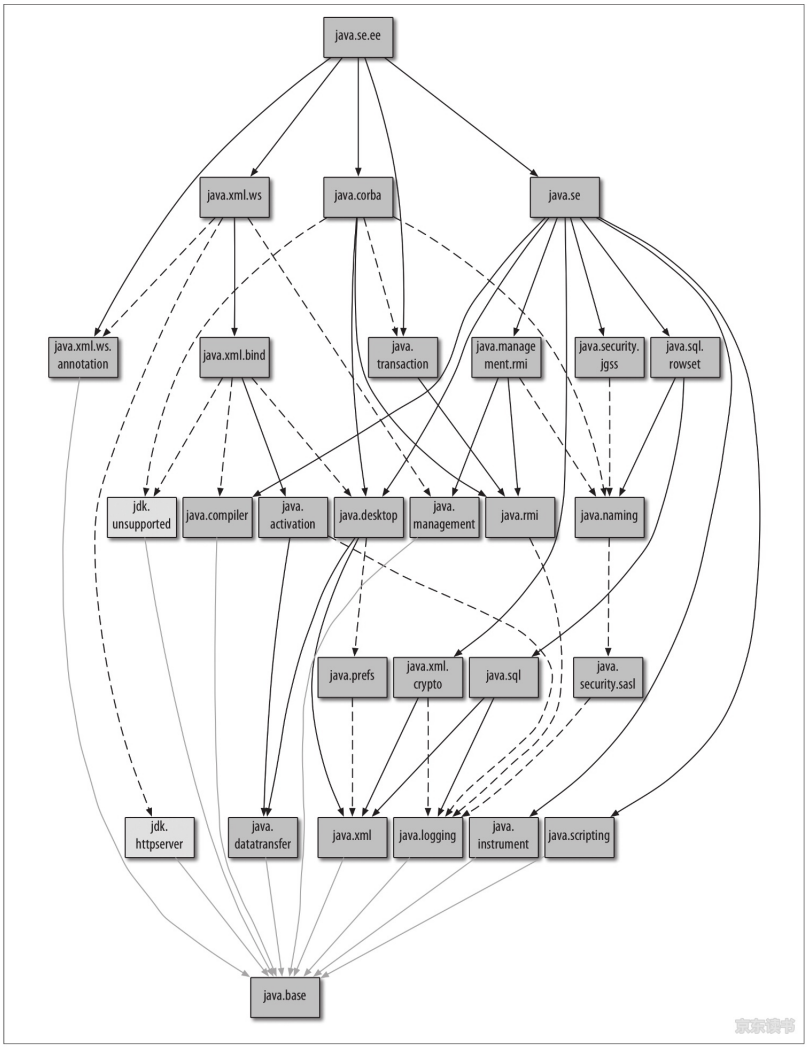
4.1 Project Jigsaw

```
Modular development starts with a modular platform. —Alan Bateman 2016.9
```

模块化开始于模块化平台 。Project Jigsaw 有如下几个目标：

1. **可伸缩平台（Scalable platform）**：逐渐从一个庞大的运行时平台到有有能力缩小到更小的计算机设备。
2. **安全性和可维护性（Security and maintainability）**：更好的组织了平台代码使得更好维护。隐藏内部API和更明确的接口定义提升了平台的安全性。
3. **提升应用程序性能（Improved application performance）**：只有必须的运行时runtimes的更小的平台可以带来更快的性能。
4. **更简单的开发体验Easier developer experience**：模块系统与模块平台的结合使得开发者更容易构建应用和库。

对Java平台进行模块化改造是一个巨大工程，JDK9之前，rt.jar是个巨大的Java运行时类库，大概有60MB左右。JDK9将其拆分成90个模块左右 ，如下图所示：



5 创建第一个Java模块

创建一个Java模块其实非常的简单。在目前Maven结构的项目下，只需要在java目录下，新建一个module-info.java文件即可。此时，当前目录就变成了一个Java模块及Maven模块。

```
--moudule1
---src
----main
-----java
------com.company.package1
------moudule-info.java
---pom.xml
```

6 模块化对现有应用的影响

6.1 你可以不用但是不能不懂

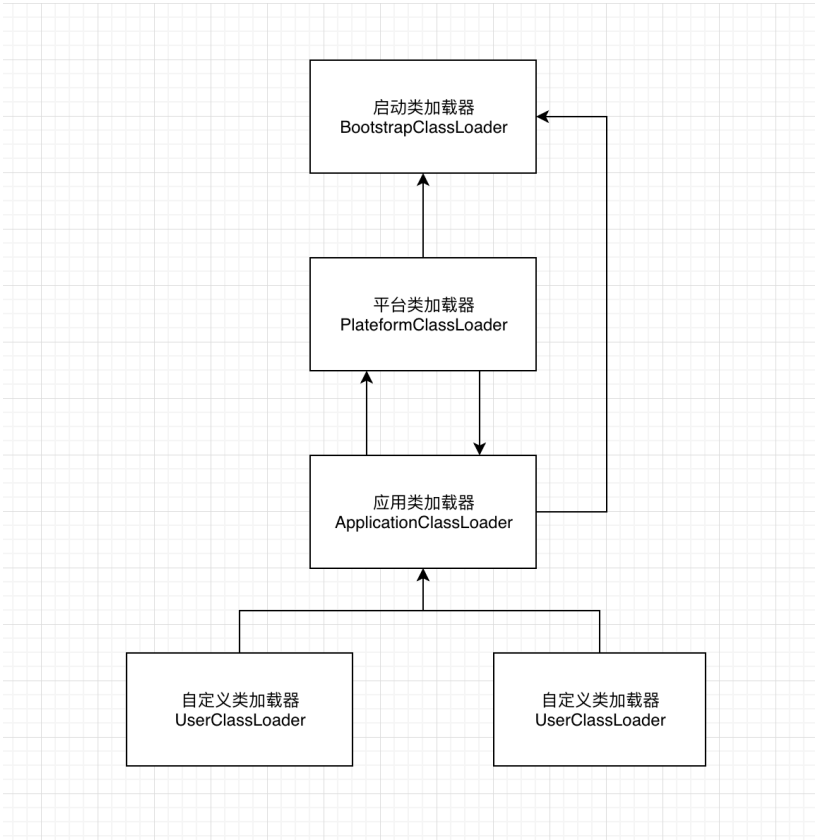
Java模块化目前并没有展现出其宣传上的影响，同时也鲜有类库正在做模块化的改造。甚至，本人在穿件第一个模块的时候，就遇到了Lombook失效、深度反射失败、Spring启动失败以及无法动态部署的影响。因此，**尽量不要尝试在线上环境使用模块化技术**！不用，但是不代表你可以不懂！随着Java平台模块化的完成，运行在JDK9环境的Java程序就已经面临着Jar包和模块的协作问题。未雨绸缪，在发现问题的时候，模块化技术可以帮你快速的定位问题并解决问题。

6.2 Java模块、Maven模块和OSGI模块的之间的关系。

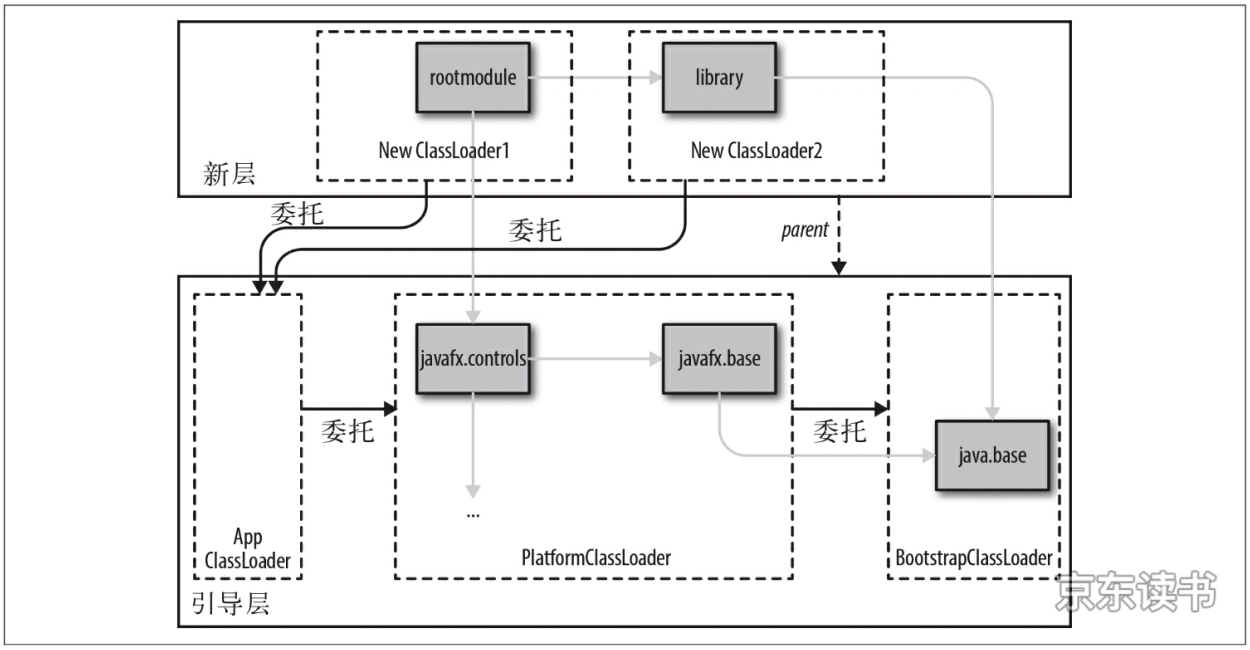
Java模块化技术，理论上可以从Java底层解决模块和模块之间的模块依赖、多版本、动态部署等问题。前文所述，在2017JDK9发布前夕，以IBM和Redhat为首的13家企业在JCP委员会上一手否决了Jigsaw项目作为Java模块化规范进入JDK9发布范围的规划。经过众多权衡，Java模块化规范中还是给Maven、Gradle和OSGI等项目保留了一席之地。目前，可以通过Java模块+Maven模块或者Java模块+OSGI模块的方式构建项目，可惜的是，使用多个技术相互合作，还是增加了复杂性。

6.3 模块化对类加载机制的影响

JDK9之后，首先取消了之前的扩展类加载器，这是清理之中，因为本身JRE扩展目录都已经不存在，取而代之的是平台类加载器。然后，类加载器的双亲委派模型机制进行了破坏，在子类将类委派给父类加载之前，会优先将当前类交给当前模块（Moudle）或层（Layer）的类加载器加载。所以会形成如下的类加载模型：



同时，在JDK9之后，引入的层（Layer）的概念，在Java程序启动时，会解析当前模块路径中的依赖关系，并形成一个依赖关系图包含在引导层（Bootstrap Layer）中，这个依赖关系图从此开始不再改变。因此，现在动态的新增模块要创建新的层，不同的层之间可以包含相同的模块。会形成如下所示的依赖关系图：



综上所述，模块化改造对于使用自定义类加载器进行功能动态变化的程序还是巨大的，一旦使用模块化，必然会导致这类功能受到巨大影响。当然模块化技术普及还需要很长一段时间，会晚但是不会不来，提前掌握相关技术还是很必要。

二、JShell[JEP222]

JShell 是 Java 9 新增的一个交互式的编程环境工具。它允许你无需使用类或者方法包装来执行 Java 语句。它与 Python 的解释器类似，可以直接 输入表达式并查看其执行结果。

三、集合工厂方法[JEP269]

List，Set 和 Map 接口中，新的静态工厂方法可以创建**不可变集合**。不可变集合的合理使用，可以优化代码和提高性能。

```
// 创建只有一个值的可读list，底层不使用数组
static <E> List<E> of(E e1) {
    return new ImmutableCollections.List12<>(e1);
}
// 创建有多个值的可读list，底层使用数组
static <E> List<E> of(E e1, E e2, E e3) {
    return new ImmutableCollections.List12<>(e1, e2,e3);
}
// 创建单例长度为0的Set结合
static <E> Set<E> of() {
    return ImmutableCollections.emptySet();
}
static <E> Set<E> of(E e1) {
    return new ImmutableCollections.Set12<>(e1);
}
```

四、接口私有方法[JEP213]

Java 8, 接口可以有默认方法。Java9之后，可以在接口内实现私有方法实现。

```
public interface HelloService {
    public void sayHello();
    // 默认方法
    default void saySomething(){
        syaEngHello();
        sayHello();
    };
    // 私有方法
    private void syaEngHello(){
        System.out.println("Hello!");
    }
}
```

五、改进的 Stream API

Java 9 为 Stream 新增了几个方法：dropWhile、takeWhile、ofNullable，为 iterate 方法新增了一个重载方法。

```
// 循环直到第一个满足条件后停止
default Stream<T> takeWhile(Predicate<? super T> predicate);
// 循环直到第一个满足条件后开始
default Stream<T> dropWhile(Predicate<? super T> predicate);
// 根据表达式生成迭代器
static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext,
UnaryOperator<T> next);
// 使用空值创建空的Stream,避免空指针
static <T> Stream<T> ofNullable(T t);
```

六、统一 JVM 日志（JEP158）

在以往的低版本中很难知道导致JVM性能问题和导致JVM崩溃的根本原因。不同的JVM对日志的使用是不同的机制和规则，这就使得JVM难以进行调试。

解决这个问题最佳的方法：对所有的JVM组件引入一个统一的日志框架，这些JVM组件支持细粒度的和易配置的JVM日志。JDK8以前常用的打印GC日志方式：

```
-Xloggc:/export/Logs/gc.log //输出GC日志到指定文件
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
```

目标：

1. 所有日志记录的通用命令行选项。
2. 通过tag对日志进行分类，例如：compiler, gc, classload, metaspace, svc, jfr等。一条日志可能会含有多个 tag
3. 日志包含多个日志级别：error, warning, info, debug, trace, develop。
4. 可以将日志重定向到控制台或者文件。
5. error, warning级别的日志重定向到标准错误stderr.
6. 可以根据日志大小或者文件数对日志文件进行滚动。
7. 一次只打印一行日志，日志之间无交叉。
8. 日志包含装饰器，默认的装饰器包括：**uptime, level, tags**，且装饰可配置。

如何使用

日志通用命令

```
-Xlog[:option]
    option          :=  [<what>][:[<output>][:[<decorators>][:<output-
options>]]]
                        'help'
                        'disable'
    what            :=  <selector>[,...]
    selector        :=  <tag-set>[*][=<level>]
    tag-set         :=  <tag>[+...]
                        'all'
    tag             :=  name of tag
    level           :=  trace
                        debug
                        info
                        warning
                        error
    output          :=  'stderr'
                        'stdout'
                        [file=]<filename>
    decorators      :=  <decorator>[,...]
                        'none'
    decorator       :=  time
                        uptime
                        timemillis
                        uptimemillis
                        timenanos
                        uptimenanos
                        pid
                        tid
                        level
                        tags
    output-options :=  <output_option>[,...]
    output-option  :=  filecount=<file count>
```

```
                        filesize=<file size>
                        parameter=value
```

1. 可以通过配置-Xlog:help参数，获取常用的JVM日志配置方式。
2. 可以通过-Xlog:disable参数关闭JVM日志。
3. 默认的JVM日志配置如下：

```
-Xlog:all=warning:stderr:uptime,level,tags
    - 默认配置
    - 'all' 即是包含所有tag
    - 默认日志输出级别warning，位置stderr
    - 包含uptime,level,tags三个装饰
```

4. 可以参考使用如下配置：

   JDK9之前参数-XX:+PrintGCDetails可参考：

```
-Xlog:safepoint,classhisto*=trace,age*,gc*=info:file=/export/Logs/gc-
%t.log:time,tid,level,tags:filecount=5,filesize=50MB
   - safepoint表示打印用户线程并发及暂停执行时间
   - classhisto表示full gc时打印堆快照信息
   - age*,gc* 表示打印包括gc及其细分过程日志，日志级别info，文
件：/export/Logs/gc.log。
    - 日志格式包含装饰符：time,tids,level,tags
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect
    - 保存日志个数5个，每个日志50M大小
```

   查看GC前后堆、方法区可用容量变化，在JDK9之前，可以使用-XX::+PrintGeapAtGC,现在可参考：

```
-
Xlog:gc+heap=debug:file=/export/Logs/gc.log:time,tids,level,tags:filecount=
5,filesize=1M
    - 打印包括gc及其细分过程日志，日志级别info，文件：/export/Logs/gc.log。
    - 日志格式包含装饰符：time,tids,level,tags
    - default output of all messages at level 'warning' to 'stderr'
    will still be in effect
    - 保存日志个数5个，每个日志1M大小
```

JDK9之前的GC日志：

```
2014-12-10T11:13:09.597+0800: 66955.317: [GC concurrent-root-region-scan-start]
2014-12-10T11:13:09.597+0800: 66955.318: Total time for which application
threads were stopped: 0.0655753 seconds
2014-12-10T11:13:09.610+0800: 66955.330: Application time: 0.0127071 seconds
2014-12-10T11:13:09.614+0800: 66955.335: Total time for which application
threads were stopped: 0.0043882 seconds
2014-12-10T11:13:09.625+0800: 66955.346: [GC concurrent-root-region-scan-end,
0.0281351 secs]
2014-12-10T11:13:09.625+0800: 66955.346: [GC concurrent-mark-start]
2014-12-10T11:13:09.645+0800: 66955.365: Application time: 0.0306801 seconds
2014-12-10T11:13:09.651+0800: 66955.371: Total time for which application
threads were stopped: 0.0061326 seconds
2014-12-10T11:13:10.212+0800: 66955.933: [GC concurrent-mark-end, 0.5871129
secs]
2014-12-10T11:13:10.212+0800: 66955.933: Application time: 0.5613792 seconds
2014-12-10T11:13:10.215+0800: 66955.935: [GC remark 66955.936: [GC ref-proc,
0.0235275 secs], 0.0320865 secs]
```

JDK9统一日志框架输出的日志格式：

```
[2021-02-09T21:12:50.870+0800][258][info][gc] Using G1
[2021-02-09T21:12:51.751+0800][365][info][gc] GC(0) Pause Young (Concurrent
Start) (Metadata GC Threshold) 60M->5M(4096M) 7.689ms
[2021-02-09T21:12:51.751+0800][283][info][gc] GC(1) Concurrent Cycle
[2021-02-09T21:12:51.755+0800][365][info][gc] GC(1) Pause Remark 13M-
>13M(4096M) 0.959ms
[2021-02-09T21:12:51.756+0800][365][info][gc] GC(1) Pause Cleanup 13M-
>13M(4096M) 0.127ms
[2021-02-09T21:12:51.758+0800][283][info][gc] GC(1) Concurrent Cycle 7.208ms
[2021-02-09T21:12:53.232+0800][365][info][gc] GC(2) Pause Young (Normal) (G1
Evacuation Pause) 197M->15M(4096M) 17.975ms
[2021-02-09T21:12:53.952+0800][365][info][gc] GC(3) Pause Young (Concurrent
Start) (GCLocker Initiated GC) 114M->17M(4096M) 15.383ms
[2021-02-09T21:12:53.952+0800][283][info][gc] GC(4) Concurrent Cycle
```

七、废弃CMS垃圾回收器[JEP291]

CMS垃圾回收器在JDK9彻底被废弃，在JDK12直接被删除。目前，G1垃圾回收器是代替CMS的最优选择。

八、Java Flow API

Reactive Streams是一套非阻塞背压的异步数据流处理规范。从Java9开始，Java原生支持Reactive Streams编程规范。Java Flow API是对Reactive Streams编程规范的1比1复刻，同时意味着从Java9开始，JDK本身开始在Reactive Streams方向上进行逐步改造。

九、局部类型推断（JEP286）

JDK10推出了局部类型推断功能，可以使用var作为局部变量类型推断标识符，减少模板代码的生成，本质还是一颗语法糖。同时var关键字的用于与lombok提供的局部类型推断功能也基本相同。

```
public static void main(String[] args) throws Exception {
    var lists = List.of("a", "b", "c");
    for (var word : lists) {
        System.out.println(word);
    }
}
```

var关键字只能用于可推断类型的代码位置，不能使用于方法形式参数，构造函数形式参数，方法返回类型等。标识符var不是关键字，它是一个保留的类型名称。这意味着var用作变量，方法名或则包名称的代码不会受到影响。但var不能作为类或则接口的名字。

var关键字的使用确实可以减少很多没必要的代码生成。但是，也存在自己的缺点：1. 现在很多ide都存在自动代码生成的快捷方式，所以使不使用var关键字区别不大。2. 局部类型推断，不光是编译器在编译时期要推断，后面维护代码的人也要推断，会在一定程度上增加理解成本。

十、G1垃圾回收器的Full GC从串行改为并行（JEP307）

G1垃圾回收器，在 Mix GC回收垃圾的速度小于新对象分配的速度时，会发生Full GC。之前，发生Full GC时采用的是Serial Old算法，该算法使用单线程标记-清除-压缩算法，垃圾回收吞吐量较高，但是Stop-The-World时间变长。JDK10，为了减少G1垃圾回收器在发生Full GC时对应用造成的影响，Full GC采用并行标记-清除-压缩算法。该算法可以通过多线程协作，减少Stop-The-World时间。线程的数量可以由-XX:ParallelGCThreads选项来配置，但是这也会影响Young GC和Mixed GC线程数量。

十一、标准Java HTTP Cliet[JEP110]

使用过Python或者其他语言的HTTP访问工具的人，都知道JDK提供的HttpURLConnection或者Apache提供的HttpClient有多么的臃肿。简单对比一下。

python 自带的urllib工具：

```
response=urllib.request.urlopen('https://www.python.org')  #请求站点获得一个
HTTPResponse对象
print(response.read().decode('utf-8'))   #返回网页内容
```

JDK:

```
HttpURLConnection connection = (HttpURLConnection) new
URL("http://localhost:8080/demo/list?name=HTTP").openConnection();
connection.setRequestMethod("GET");
connection.connect();
int responseCode = connection.getResponseCode();
log.info("response code : {}", responseCode);
// read response
try (BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()))) {
```

```
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} finally {
    connection.disconnect();
}
```

Apache HttpClient:

```
CloseableHttpClient httpClient = HttpClientBuilder.create().build();
// 创建Get请求
HttpGet httpGet = new HttpGet("http://localhost:12345/doGetControllerOne");
// 响应模型
CloseableHttpResponse response = null;
// 由客户端执行(发送)Get请求
response = httpClient.execute(httpGet);
// 从响应模型中获取响应实体
HttpEntity responseEntity = response.getEntity();
System.out.println("响应状态为:" + response.getStatusLine());
```

Java 9 中引入了标准Http Client API 。并在 Java 10 中进行了更新的。 到了Java11，在前两个版本中进行孵化的同时，Http Client 几乎被完全重写，并且现在完全支持异步非阻塞。与此同时它是 Java 在 Reactive-Stream 方面的第一个生产实践，其中广泛使用了 Java Flow API。

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
      .uri(URI.create("http://openjdk.java.net/"))
      .build();
client.sendAsync(request, BodyHandlers.ofString())
      .thenApply(HttpResponse::body)
      .thenAccept(System.out::println)
      .join();
```

十二、新一代JIT编译器 Graal

即时编译器在提高JVM性能上扮演着非常重要的角色。目前存在两JIT编译器：编译速度较快但对编译后的代码优化较低的C1编译器；编译速度较慢但编译后的代码优化较高的C2编译器。两个编译器在服务端程序及分层编译算法中扮演着非常重要的角色。但是，C2编译器已经存在将近20年了，其中混乱的代码以及部分糟糕的架构使其难以维护。JDK10推出了新一代JIT编译器Graal（JEP317）。Graal作为C2的继任者出现，完全基于Java实现。Graal编译器借鉴了C2编译器优秀的思想同时，使用了新的架构。这让Graal在性能上很快追平了C2，并且在某些特殊的场景下还有更优秀的表现。遗憾的是，目前Graal编译器还属于实验性的功能，在生产环境中使用时应额外注意。

十三、Epsilon：低开销垃圾回收器

Epsilon 垃圾回收器的目标是开发一个控制内存分配，但是不执行任何实际的垃圾回收工作。下面是该垃圾回收器的几个使用场景：性能测试、内存压力测试、极度短暂 job 任务、延迟改进、吞吐改进。

十四、低延迟垃圾回收器-ZGC

JDK11中，最耀眼的新特性就是ZGC垃圾回收器。作为实验性功能，ZGC的特点包括：

1. GC停顿时间不会超过10ms。
2. 停顿时间不会随着堆的大小，或者活跃对象的大小而增加；
3. 相对于G1垃圾回收器而言，吞吐量降低不超过15%；
4. 仅支持Linux/x64平台；
5. 支持8MB~4TB级别的堆。

ZGC通过有色指针和读屏障技术，解决了回收对象的并发问题，并且ZGC在整个垃圾回收过程中几乎都是并发执行的。虽然目前ZGC还是在实验状态，但是通过SPECjbb® 2015 性能比较结果看，ZGC有着非常出色的性能表现：

1. 在仅关注吞吐量指标下，ZGC超过了G1；
2. 在最大延迟不超过某个设定值（10到100ms）下关注吞吐量，ZGC较G1性能更加突出。
3. 在仅关注低延迟指标下，ZGC的性能高出G1将近两个数量级。99.9th仅为G1的百分之一。

当然，ZGC的性能表现还是与线上生产环境息息相关。即使是这样，ZGC对于低延迟要求非常高的SLA服务有着不可抵挡的吸引力。

十五、升级JDK11后，与提升性能相关的新特性总结

a. Segmented Code Cache

b. Compact Strings

c. Application Class-Data Sharing

d. Thread-Local Handshakes

e. Lazy Allocation of Compiler Threads

f. Variable Handles

e. Spin-Wait Hints