

Java 9 新特性

讲师：尚硅谷 - 宋红康

一、Java 9 概述

1. jdk 9 的发布

经过 4 次跳票，历经曲折的 java 9 终于终于在 2017 年 9 月 21 日发布。



2. 哪些人适合看这套视频？

- 已经熟悉或熟练运用 java 8 及 之前 java 版本的开发人员、科研人员、学生及兴趣爱好者
- 本套视频不会从头介绍 java 的基本语法、面向对象等核心内容，这里讲解基于 java 8 之后 java 9 中的新特性
- 有兴趣学习 java 语言的朋友，可以下载学习《[尚硅谷 java 基础经典 20 天视频](#)》

3. Java 9 中哪些不得不说的新特性？

java 9 提供了超过 150 项新功能特性，包括备受期待的模块化系统、可交互的 REPL 工具：jshell，JDK 编译工具，Java 公共 API 和私有代码，以及安全增强、扩展提升、性能管理改善等。可以说 Java 9 是一个庞大的系统工程，完全做了一个整体改变。

具体来讲：

- 模块化系统
- jShell 命令
- 多版本兼容 jar 包
- 接口的私有方法
- 钻石操作符的使用升级
- 语法改进：try 语句
- 下划线使用限制
- String 存储结构变更
- 便利的集合特性：of()
- 增强的 Stream API
- 多分辨率图像 API
- 全新的 HTTP 客户端 API
- Deprecated 的相关 API
- 智能 Java 编译工具
- 统一的 JVM 日志系统
- javadoc 的 HTML 5 支持
- Javascript 引擎升级：Nashorn
- java 的动态编译器

4. java 语言后续版本的更迭

- 从Java 9 这个版本开始，Java 的计划发布周期是 6 个月，下一

个 Java 的主版本将于 2018 年 3 月发布，命名为 **Java 18.3**，紧接着再过六个月将发布 **Java 18.9**。

- 这意味着java的更新从传统的以**特性驱动**的发布周期，转变为以**时间驱动**的（6 个月为周期）发布模式，并逐步的将 Oracle JDK 原商业特性进行开源。
- 针对企业客户的需求，Oracle 将以**三年为周期**发布长期支持版本（long term support）。

5. 如何看待 java 9 的更新

Java 更快的发布周期意味着开发者将不需要像以前一样为主要发布版本望眼欲穿。这也意味着开发者将可能跳过 Java 9 和它的不成熟的模块化功能，只需要再等待 6 个月就可以迎来新版本，这将可能解决开发者的纠结。

oracle 理念 与 小步快跑，快速迭代

二、 java 9 的安装和官网说明

1.jdk 9 的下载

<http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html>

Overview Downloads Documentation Community Technologies Training

Java SE Downloads


DOWNLOAD 
Java Platform (JDK) 9


DOWNLOAD 
NetBeans with JDK 8

Java Platform, Standard Edition

Java SE 9.0.1
Java SE 9.0.1 includes important bug fixes. Oracle strongly recommends that all Java SE 9 users upgrade to this release.
[Learn more](#) 






- Installation Instructions
- Release Notes
- Oracle License

JDK
DOWNLOAD 

Java SE Development Kit 9.0.1

You must accept the [Oracle Binary Code License Agreement for Java SE](#) to download this software.

☒ Accept License Agreement ☐ Decline License Agreement

Product / File Description	File Size	Download
Linux	304.99 MB	 jdk-9.0.1_linux-x64_bin.rpm
Linux	338.11 MB	 jdk-9.0.1_linux-x64_bin.tar.gz
macOS	382.11 MB	 jdk-9.0.1_osx-x64_bin.dmg
Windows	375.51 MB	 jdk-9.0.1_windows-x64_bin.exe
Solaris SPARC	206.85 MB	 jdk-9.0.1_solaris-sparcv9_bin.tar.gz

下载安装完毕，需要配置环境变量：

① 新建 JAVA_HOME 的环境变量，变量值为 jdk 路径。如下：



② 将 JAVA_HOME 配置到 path 环境变量下：

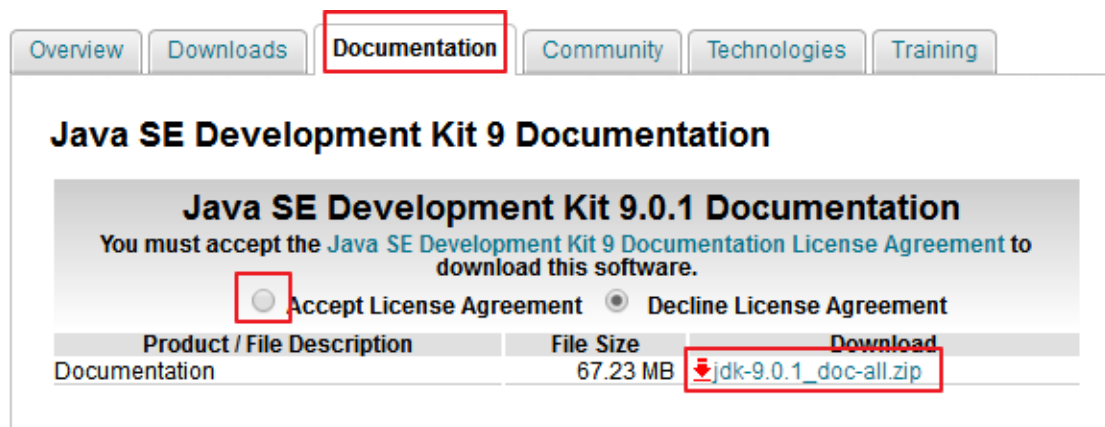


之后在命令行校验是否配置成功。成功，则显示如下：

```
C:\Users\Administrator>java -version
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
C:\Users\Administrator>
```

2. java 9 帮助文档的下载

<http://www.oracle.com/technetwork/java/javase/documentation/jdk9-doc-downloads-3850606.html>



3. 在线 Oracle JDK 9 Documentation

<https://docs.oracle.com/javase/9/>

三、Java 9 的新特性

官方提供的新特性列表：

<https://docs.oracle.com/javase/9/whatsnew/toc.htm#JSNEW-GUID-C23AFD78-C777-460B-8ACE-58BE5EA681F6>

或参考 Open JDK

<http://openjdk.java.net/projects/jdk9/>

Features	特性
JEP 102: Process API Updates	102:API 更新过程
JEP 110: HTTP 2 Client	110:2 HTTP 客户端
JEP 143: Improve Contended Locking	143:改善竞争锁
JEP 158: Unified JVM Logging	158:统一 JVM 日志记录
JEP 165: Compiler Control	165:编译器控制
JEP 193: Variable Handles	193:变量处理
JEP 197: Segmented Code Cache	197:分段代码缓存
JEP 199: Smart Java Compilation, Phase Two	199:聪明的 Java 编译,第二阶段
JEP 200: The Modular JDK	200:模块化 JDK
JEP 201: Modular Source Code	201:模块化的源代码
JEP 211: Elide Deprecation Warnings on Import Statements	211: 导入语句时省略警告信息
JEP 212: Resolve Lint and Doclint Warnings	212:解决 Lint 和 Doclint 警告
JEP 213: Milling Project Coin	213:研磨项目 Coin
JEP 214: Remove GC Combinations Deprecated in JDK 8	214:移除 JDK 8 过时的 GC 组合
JEP 215: Tiered Attribution for javac	215:javac 分层归因
JEP 216: Process Import Statements Correctly	216:正确处理导入语句
JEP 217: Annotations Pipeline 2.0	217:2.0 注释管道
JEP 219: Datagram Transport Layer Security (DTLS)	219:数据报传输层安全性(DTLS)
JEP 220: Modular Run-Time Images	220:模块化运行时图像
JEP 221: Simplified Doclet API	221:简化 Doclet API
JEP 222: jshell: The Java Shell (Read-Eval-Print Loop)	222:jshell:Java Shell(Read-Eval-Print-Loop)
JEP 223: New Version-String Scheme	223: 字符串新版本方案
JEP 224: HTML5 Javadoc	224:Javadoc 支持 HTML5
JEP 225: Javadoc Search	225:Javadoc 搜索
JEP 226: UTF-8 Property Files	226:utf - 8 属性文件
JEP 227: Unicode 7.0	227:Unicode 7.0

JEP 228: Add More Diagnostic Commands	228:添加更多的诊断命令
JEP 229: Create PKCS12 Keystores by Default	229:创建 PKCS12 默认密钥存储库
JEP 231: Remove Launch-Time JRE Version Selection	231:删除启动 JRE 版本选择
JEP 232: Improve Secure Application Performance	232:提高安全应用程序性能
JEP 233: Generate Run-Time Compiler Tests Automatically	233:自动生成运行时编译器测试
JEP 235: Test Class-File Attributes Generated by javac	235:测试 javac 生成的类文件属性
JEP 236: Parser API for Nashorn	236:Nashorn 解析器 API
JEP 237: Linux/AArch64 Port	237:Linux / AArch64 端口
JEP 238: Multi-Release JAR Files	238:Multi-Release JAR 文件
JEP 240: Remove the JVM TI hprof Agent	240:删除 JVM TI hprof 代理
JEP 241: Remove the jhat Tool	241:删除 jhat 工具
JEP 243: Java-Level JVM Compiler Interface	243:java 级别 JVM 编译器接口
JEP 244: TLS Application-Layer Protocol Negotiation Extension	244:TLS 应用层协议谈判扩展
JEP 245: Validate JVM Command-Line Flag Arguments	245:JVM 命令行标记参数进行验证
JEP 246: Leverage CPU Instructions for GHASH and RSA	246:利用 CPU 指令 GHASH 和 RSA
JEP 247: Compile for Older Platform Versions	247:老平台版本编译
JEP 248: Make G1 the Default Garbage Collector	248:设置 G1 为默认的垃圾收集器
JEP 249: OCSP Stapling for TLS	249:OCSP 装订 TLS
JEP 250: Store Interned Strings in CDS Archives	250:CDS 档案中存储实际字符串
JEP 251: Multi-Resolution Images	251:多分辨率图像
JEP 252: Use CLDR Locale Data by Default	252:使用系统默认语言环境数据
JEP 253: Prepare JavaFX UI Controls & CSS APIs for Modularization	253 年:准备 JavaFX UI 控件和 CSS api 用于模块化
JEP 254: Compact Strings	254:紧凑的字符串
JEP 255: Merge Selected Xerces 2.11.0 Updates into JAXP	255:合并选定的 Xerces 2.11.0 更新到 JAXP
JEP 256: BeanInfo Annotations	256:BeanInfo 注释
JEP 257: Update JavaFX/Media to Newer Version of GStreamer	257:更新 JavaFX /Media 到 GStreamer 的新版本
JEP 258: HarfBuzz Font-Layout Engine	258:HarfBuzz 文字编排引擎
JEP 259: Stack-Walking API	259:提供 Stack – Walking API

JEP 260: Encapsulate Most Internal APIs	260:封装内部 api
JEP 261: Module System	261:模块系统
JEP 262: TIFF Image I/O	262:TIFF 图像 I/O
JEP 263: HiDPI Graphics on Windows and Linux	263:Windows 和 Linux 上的 HiDPI 图形
JEP 264: Platform Logging API and Service	264:日志 API 和服务平台
JEP 265: Marlin Graphics Renderer	265:Marlin 图形渲染器
JEP 266: More Concurrency Updates	266:更多的并发更新
JEP 267: Unicode 8.0	267:Unicode 8.0
JEP 268: XML Catalogs	268:XML 目录
JEP 269: Convenience Factory Methods for Collections	269:方便的集合工厂方法
JEP 270: Reserved Stack Areas for Critical Sections	270:保留堆栈领域至关重要的部分
JEP 271: Unified GC Logging	271:统一的 GC 日志记录
JEP 272: Platform-Specific Desktop Features	272:特定于平台的桌面功能
JEP 273: DRBG-Based SecureRandom Implementations	273:基于 DRBG 的 SecureRandom 实现
JEP 274: Enhanced Method Handles	274:增强的方法处理
JEP 275: Modular Java Application Packaging	275:模块化 Java 应用程序包装
JEP 276: Dynamic Linking of Language-Defined Object Models	276:语言定义对象模型的动态链接
JEP 277: Enhanced Deprecation	277:增强的弃用
JEP 278: Additional Tests for Humongous Objects in G1	278:为 G1 的极大对象提供额外的测试
JEP 279: Improve Test-Failure Troubleshooting	279:改善测试失败的故障排除
JEP 280: Indify String Concatenation	280:Indify 字符串连接
JEP 281: HotSpot C++ Unit-Test Framework	281:热点 c++ 的单元测试框架
JEP 282: jlink: The Java Linker	282:jlink:Java 连接器
JEP 283: Enable GTK 3 on Linux	283:在 Linux 上启用 GTK 3
JEP 284: New HotSpot Build System	284:新热点的构建系统
JEP 285: Spin-Wait Hints	285:循环等待提示
JEP 287: SHA-3 Hash Algorithms	287:SHA-3 散列算法
JEP 288: Disable SHA-1 Certificates	288:禁用 sha - 1 证书

JEP 289: <u>Deprecate the Applet API</u>	289:标记过时的 Applet API
JEP 290: <u>Filter Incoming Serialization Data</u>	290:过滤传入的序列化数据
291: <u>Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector</u>	291:反对并发标记清理垃圾收集器(CMS)
JEP 292: <u>Implement Selected ECMAScript 6 Features in Nashorn</u>	292:实现选定的 ECMAScript Nashorn 6 特性
JEP 294: <u>Linux/s390x Port</u>	294:Linux / s390x 端口
JEP 295: <u>Ahead-of-Time Compilation</u>	295:提前编译
JEP 297: <u>Unified arm32/arm64 Port</u>	297:统一的 arm32 / arm64 端口
JEP 298: <u>Remove Demos and Samples</u>	298:删除演示和样本
JEP 299: <u>Reorganize Documentation</u>	299:整理文档

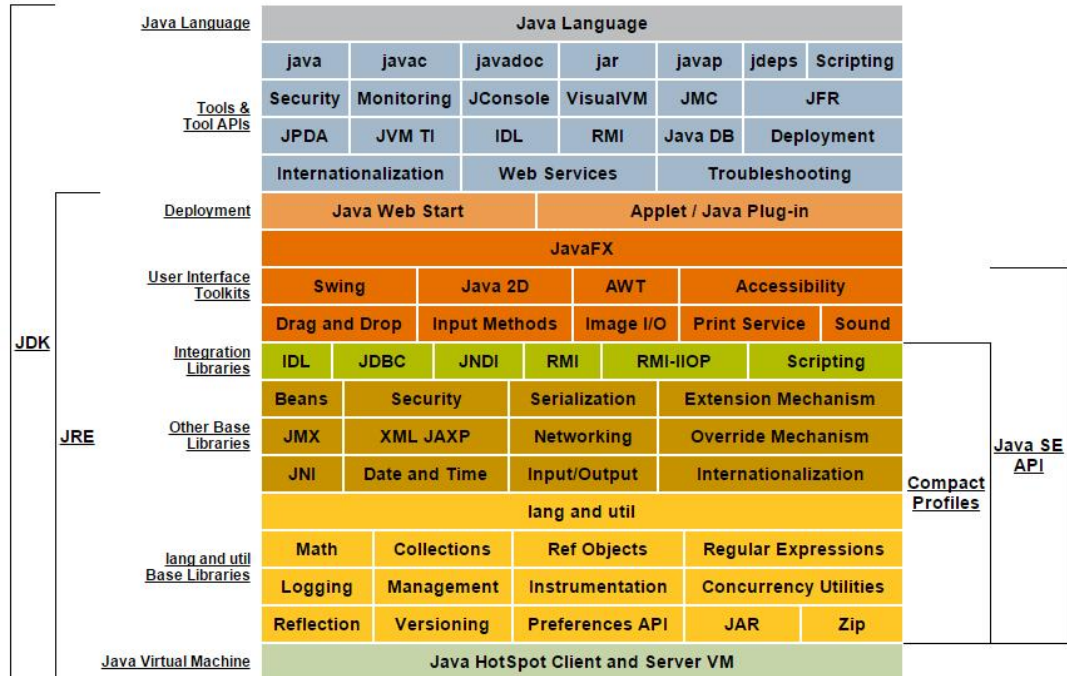
- **JEP(JDK Enhancement Proposals)**: jdk 改进提案，每当需要有新的设想时候，JEP 可以在 JCP(java community Process)之前或者同时提出**非正式的规范(specification)**，被正式认可的 JEP 正式写进 JDK 的发展路线图并分配版本号。
- **JSR(Java Specification Requests)**: java 规范提案，新特性的规范出现在这一阶段，是指向 JCP(Java Community Process)提出新增一个**标准化技术规范的正式请求**。请求可以来自于小组/项目、JEP、JCP 成员或者 java 社区(community)成员的提案，每个 java 版本都由相应的 JSR 支持。
 - 小组：对特定技术内容，比如安全、网络、HotSpot 等有共同兴趣的组织和个人
 - 项目：编写代码、文档以及其他工作，至少由一个小组赞助和支持，比如最近的 Lambda 计划，JigSaw 计划等

1. JDK 和 JRE 的改变

1.1 JDK 与 JRE 的关系

JDK : **J**ava **D**evelopment **K**it (Java 开发工具包)

JRE : Java Runtime Environment (Java 运行环境)

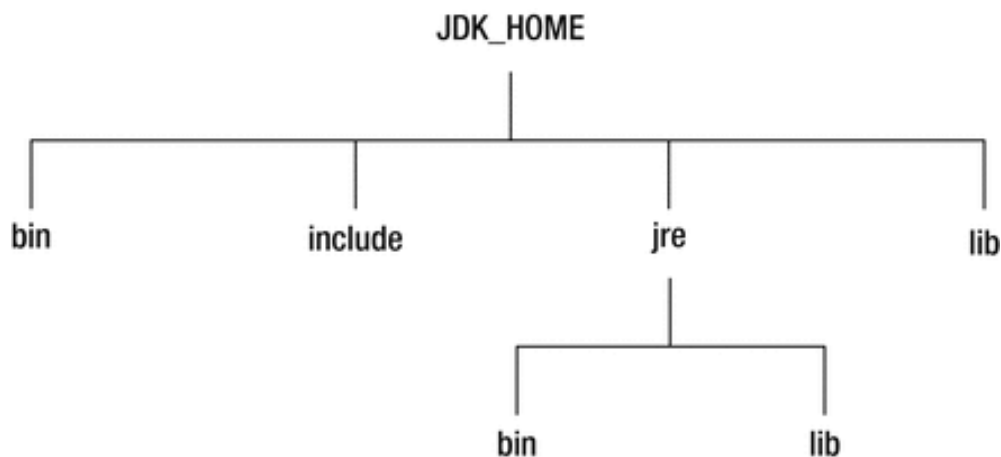


说明:

JDK = JRE + 开发工具集 (例如 `Javac` 编译工具等)

JRE = JVM + Java SE 标准类库

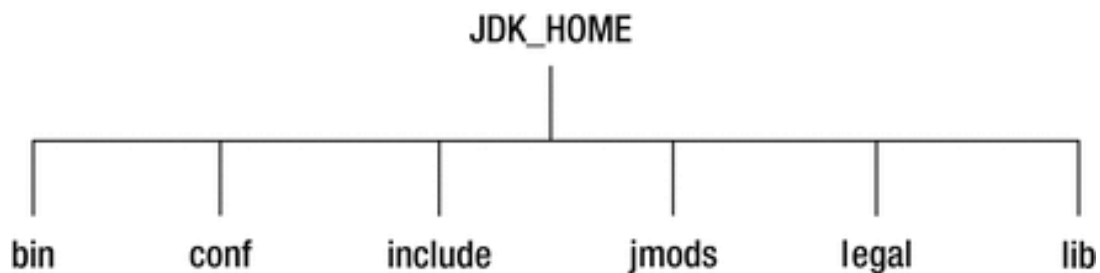
1.2 JDK 8 的目录结构



说明:

bin 目录	包含命令行开发和调试工具，如 javac, jar 和 javadoc。
include 目录	包含在编译本地代码时使用的 C/C++头文件
lib 目录	包含 JDK 工具的几个 JAR 和其他类型的文件。 它有一个 tools.jar 文件，其中包含 javac 编译器的 Java 类
jre/bin 目录	包含基本命令，如 java 命令。 在 Windows 平台上，它包含系统的运行时动态链接库（DLL）。
jre/lib 目录	包含用户可编辑的配置文件，如.properties 和.policy 文件。包含几个 JAR。 rt.jar 文件包含运行时的 Java 类和资源文件。

1.3 JDK 9 的目录结构



说明：

没有名为 jre 的子目录	
bin 目录	包含所有命令。 在 Windows 平台上，它继续包含系统的运行时动态链接库。
conf 目录	包含用户可编辑的配置文件，例如以前位于 jre\lib 目录中的.properties 和.policy 文件
include 目录	包含要在以前编译本地代码时使用的 C/C++头文件。它只存在于 JDK 中
jmods 目录	包含 JMOD 格式的平台模块。创建自定义运行时映像时需要它。 它只存在于 JDK 中

legal 目录	包含法律声明
lib 目录	包含非 Windows 平台上的动态链接本地库。其子目录和文件不应由开发人员直接编辑或使用

2. 模块化系统: Jigsaw → Modularity

2.1 官方 Feature

200: [The Modular JDK](#)

201: [Modular Source Code](#)

220: [Modular Run-Time Images](#)

260: [Encapsulate Most Internal APIs](#)

261: [Module System](#)

282: [jlink: The Java Linker](#)

2.2 产生背景及意义

- 谈到 Java 9 大家往往第一个想到的就是 Jigsaw 项目。众所周知，Java 已经发展超过 20 年（95 年最初发布），Java 和相关生态在不断丰富的同时也越来越暴露出一些问题：
 - Java 运行环境的膨胀和臃肿。每次JVM启动的时候，至少会有 30~60MB的内存加载，主要原因是JVM需要加载rt.jar，不管其中的类是否被classloader加载，第一步整个jar都会被JVM加载到内存当中去（而模块化可以根据模块的需要加载程序运行需要的class）
 - 当代码库越来越大，创建复杂，盘根错节的“意大利面条式代码”的几率呈指数级的增长。不同版本的类库交叉依赖导致让人头疼的问题，这些都阻碍了 Java 开发和运行效率的提升。
 - 很难真正地对代码进行封装，而系统并没有对不同部分（也就是 JAR 文件）之间的依赖关系有个明确的概念。每一个公共

类都可以被类路径之下任何其它的公共类所访问到，这样就会导致无意中使用了并不想被公开访问的 API。

- 类路径本身也存在问题：你怎么知晓所有需要的 JAR 都已经有了，或者是不是会有重复的项呢？
- 同时，由于兼容性等各方面的掣肘，对 Java 进行大刀阔斧的革新越来越困难，Jigsaw 从 Java 7 阶段就开始筹备，Java 8 阶段进行了大量工作，终于在 Java 9 里落地，一种千呼万唤始出来的意味。
- Jigsaw项目（后期更名为Modularity）的工作量和难度大大超出了初始规划。JSR 376 Java 平台模块化系统（JPMS, Java Platform Module System）作为 Jigsaw 项目的核心，其主体部分被分解成 6 个 JEP(JDK Enhancement Proposals)。
- 作为java 9 平台最大的一个特性，随着 Java 平台模块化系统的落地，开发人员无需再为不断膨胀的 Java 平台苦恼，例如，您可以使用 jlink 工具，根据需要定制运行时环境。这对于拥有大量镜像的容器应用场景或复杂依赖关系的大型应用等，都具有非常重要的意义。
- 本质上讲，模块(module)的概念，其实就是package外再裹一层，也就是说，用模块来管理各个package，通过声明某个package暴露，不声明默认就是隐藏。因此，模块化使得代码组织上更安全，因为它可以指定哪些部分可以暴露，哪些部分隐藏。

2.3 设计理念

模块独立、化繁为简

模块化（以 Java 平台模块系统的形式）将 JDK 分成一组模块，可以在编译时，运行时或者构建时进行组合。

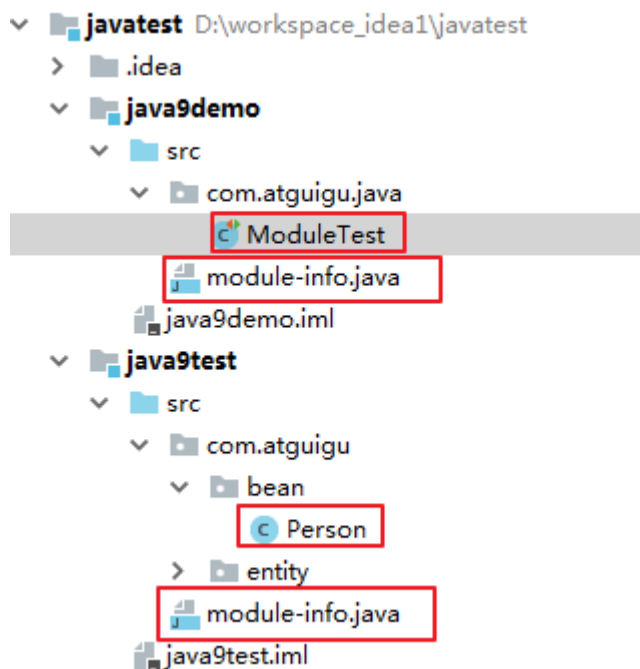
2.4 实现目标

- 主要目的在于减少内存的开销
- 只须必要模块，而非全部jdk模块，可简化各种类库和大型应用的开发和维护
- 改进 Java SE 平台，使其可以适应不同大小的计算设备
- 改进其安全性，可维护性，提高性能

2.5 使用举例

注： IntelliJ IDEA 2017.3 支持模块化特性，这里选择此开发环境。

模块将由通常的类和新的模块声明文件（module-info.java）组成。该文件是位于 java 代码结构的顶层，该模块描述符明确地定义了我们的模块需要什么依赖关系，以及哪些模块被外部使用。在 exports 子句中未提及的所有包默认情况下将封装在模块中，不能在外部使用。



java 9demo 模块中的 ModuleTest 类使用如下：

```
package com.atguigu.java;
```



```
import com.atguigu.bean.Person;
import org.junit.Test;

import java.util.logging.Logger;

/**
 * Created by songhongkang on 2017/12/27 0027.
 */
public class ModuleTest {

    private static final Logger LOGGER =
        Logger.getLogger("java9test");

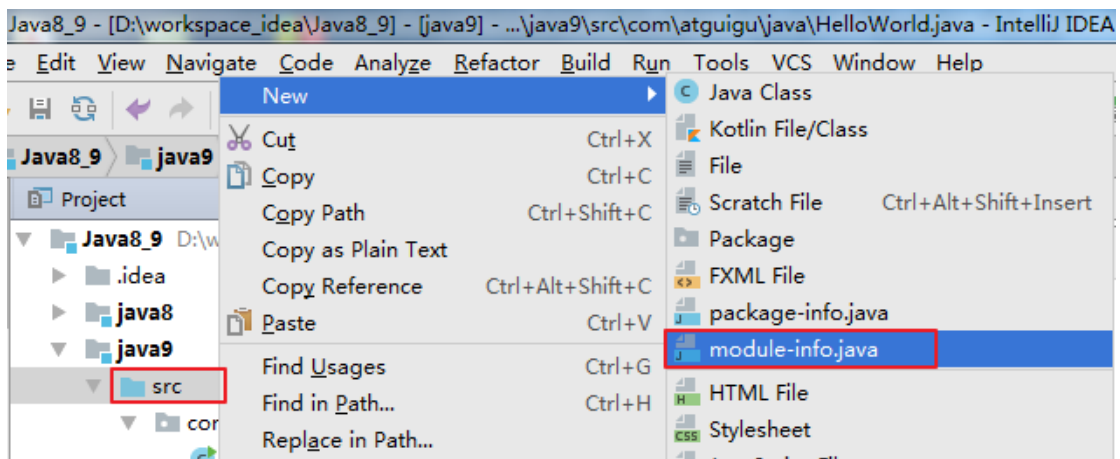
    public static void main(String[] args) {
        Person p = new Person("马云", 40);

        System.out.println(p);

        LOGGER.info("aaaaaa");
    }

    @Test
    public void test1(){
        System.out.println("hello");
    }
}
```

对应 java 9demo 模块的 src 下创建 module-info.java 文件:




```
/**
 * Created by songhongkang on 2017/12/27 0027.
 */
module java9demo {
    requires java9test;
    requires java.logging;
    requires junit;
}
```

requires: 指明对其它模块的依赖。

在 java9test 模块的指定包下提供类 Person:

```
package com.atguigu.bean;

/**
 * Created by songhongkang on 2017/12/27 0027.
 */
public class Person {

    private String name;
    private int age;

    public Person() {

    }

    public Person(String name, int age) {

        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
```

```
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
    }
}
```

要想在 java9demo 模块中调用 java9test 模块下包中的结构，需要在 java9test 的 module-info.java 中声明：

```
/**
 * Created by songhongkang on 2017/12/27 0027.
 */
module java9test {
    //package we export
    exports com.atguigu.bean;
}
```

exports: 控制着哪些包可以被其它模块访问到。所有不被导出的包默认都被封装在模块里面。

2.6 补充说明

关于更多 Java 9 模块编程的内容请参考一本书：《[Java 9 Modularity](#)》里面讲的比较详细，介绍了当前 Java 对 jar 之间以来的管理是多么的混乱，引入 modularity 之后的改变会是很明显的差别。

3. Java 的 REPL 工具： jShell 命令

3.1 官方 Feature

222: jshell: The Java Shell (Read-Eval-Print Loop)

3.2 产生背景

像 Python 和 Scala 之类的语言早就有交互式编程环境 REPL (read - evaluate - print - loop)了，以交互式的方式对语句和表达式进行求值。开发者只需要输入一些代码，就可以在编译前获得对程序的反馈。而之前的 Java 版本要想执行代码，必须创建文件、声明类、提供测试方法方可实现。

3.3 设计理念

即写即得、快速运行

3.4 实现目标

- Java 9 中终于拥有了 REPL 工具：jShell。利用 jShell 在没有创建类的情况下直接声明变量，计算表达式，执行语句。即开发时可以在命令行里直接运行 java 的代码，而无需创建 Java 文件，无需跟人解释“public static void main(String[] args)”这句废话。
- jShell 也可以从文件中加载语句或者将语句保存到文件中。
- jShell 也可以是 tab 键进行自动补全和自动添加分号。

3.5 使用举例

调出 jShell

```
C:\Users\Administrator>jshell
: 欢迎使用 JShell -- 版本 9.0.1
: 要大致了解该版本, 请键入: /help intro
```

获取帮助

```
jsHELL> /help intro
:
: intro
:
: 使用 jsHELL 工具可以执行 Java 代码, 从而立即获取结果。
: 您可以输入 Java 定义 (变量, 方法, 类, 等等), 例如: int x = 8
: 或 Java 表达式, 例如: x + x
: 或 Java 语句或导入。
: 这些小块 of Java 代码称为 '片段'。
:
: 这些 jsHELL 命令还可以让您了解和
: 控制您正在执行的操作, 例如: /list
```

基本使用

```
jsHELL> System.out.println("你好! world");
你好! world

jsHELL> int i = 10;
i ==> 10

jsHELL> int j = 20;
j ==> 20

jsHELL> int k = i + j;
k ==> 30

jsHELL> System.out.println(k);
30
```

```
jsshell> public int add<int m,int n>{  
    ...> return m + n;  
    ...> }  
! 已创建 方法 add<int,int>  
  
jsshell> int k = add<1,2>;  
k ==> 3  
  
jsshell> System.out.println(k);  
3
```

Tips: 在 JShell 环境下, 语句末尾的 “;” 是可选的。但推荐还是最好加上。提高代码可读性。

导入指定的包

```
jsshell> import java.util.*;
```

默认已经导入如下的所有包: (包含 java.lang 包)

```
jsshell> /imports  
!  
! import java.io.*  
! import java.math.*  
! import java.net.*  
! import java.nio.file.*  
! import java.util.*  
! import java.util.concurrent.*  
! import java.util.function.*  
! import java.util.prefs.*  
! import java.util.regex.*  
! import java.util.stream.*
```

只需按下 **Tab** 键，就能自动补全代码

```
jsshell> Sy
SyncFailedException   SynchronousQueue   System

jsshell> System.out
out

jsshell> System.out.
append(               checkError()   close()               equals(               flush()
format(               getClass()   hashCode()           notify()             notifyAll()
print(                printf(      println(             toString()           wait(
write(

jsshell> System.out.
```

列出当前 **session** 里所有有效的代码片段

```
jsshell> /list

1 : public int add(int m,int n){
    return m + n;
}
2 : int k = add(1,2);
3 : System.out.println(k);

jsshell>
```

查看当前 **session** 下所有创建过的变量

```
jsshell> /var
!      int k = 3

jsshell>
```

查看当前 **session** 下所有创建过的方法

```
jsshell> /methods
!      int add(int,int)

jsshell>
```

Tips: 我们还可以重新定义相同方法名和参数列表的方法，即为对现有方法的修改（或覆盖）。

使用外部代码编辑器来编写 Java 代码

```
jshe11> /edit add
! 已修改 方法 add(int,int)

jshe11> add(1,2);
a

jshe11> int j = 5;
j ==> 5

jshe11> /edit j

jshe11> _半.
```

从外部文件加载源代码

指定目录下提供 HelloWorld.java 文件:

```
/**
 * Created by songhongkang on 2017/12/27 0020.
 */
public void printHello() {
    System.out.println("马上 2018 年了，尚硅谷祝所有的谷粉元旦
快乐!");
}
printHello();
```

使用/open 命令调用:

```
jshe11> /open E:\teach\01_Java9\HelloWorld.java
马上2018年了，尚硅谷祝所有的谷粉元旦快乐!

jshe11>
```

没有受检异常（编译时异常）

```
jshell> URL url = new URL("http://www.atguigu.com");  
url ==> http://www.atguigu.com  
  
jshell>
```

说明：本来应该强迫我们捕获一个 `IOException`，但却没有出现。因为 `jShell` 在后台为我们隐藏了。

退出 `jShell`

```
jshell> /exit  
! 再见
```

4. 多版本兼容 jar 包

4.1 官方 Feature

238: [Multi-Release JAR Files](#)

4.2 使用说明

当一个新版本的 `Java` 出现的时候，你的库用户要花费数年时间才会切换到这个新的版本。这就意味着库得去向后兼容你想要支持的最老的 `Java` 版本（许多情况下就是 `Java 6` 或者 `Java7`）。这实际上意味着未来的很长一段时间，你都不能在库中运用 `Java 9` 所提供的新特性。幸运的是，多版本兼容 `jar` 功能能让你创建仅在特定版本的 `Java` 环境中运行库程序选择使用的 `class` 版本。

举例 1:

```
jar root  
- A.class
```



```
- B.class
- C.class
- D.class
- META-INF
  - versions
    - 9
      - A.class
      - B.class
```

说明：

在上述场景中，root.jar 可以在 Java 9 中使用，不过 A 或 B 类使用的不是顶层的 root.A 或 root.B 这两个 class，而是处在“META-INF/versions/9”下面的这两个。这是特别为 Java 9 准备的 class 版本，可以运用 Java 9 所提供的特性和库。同时，在早期的 Java 诸版本中使用这个 JAR 也是能运行的，因为较老版本的 Java 只会看到顶层的 A 类或 B 类。

举例 2：

```
jar root
- A.class
- B.class
- C.class
- D.class
- META-INF
  - versions
    - 9
      - A.class
```

- B.class

- 10

- A.class

官方说明：

By this scheme, it is possible for versions of a class designed for a later Java platform release to override the version of that same class designed for an earlier Java platform release.

4.3 使用举例

步骤一：提供必要的类

在指定目录(E:\teach\01_Java9\multijar\src\main\java\com\atguigu)下提供如下的类：

【Generator.java】

```
package com.atguigu;

import java.util.Set;
import java.util.HashSet;
/**
 * Created by songhongkang on 2017/12/28 0028.
 */
public class Generator {

    public Set<String> createStrings() {
        Set<String> strings = new HashSet<String>();
        strings.add("Java");
        strings.add("8");
        return strings;
    }
}
```

【Application.java】

```
package com.atguigu;

import com.atguigu.Generator;

import java.io.IOException;
import java.util.List;
import java.util.ArrayList;
import java.util.Set;

/**
 * Created by songhongkang on 2017/12/28 0028.
 */
public class Application {

    public static void testMultiJar() {
        com.atguigu.Generator gen = new Generator();
        System.out.println("Generated strings: " +
gen.createStrings());
    }

}
```

在如下目录(E:\teach\01_Java9\multijar\src\main\java-9\com\atguigu)下提供同名的类:

【Generator.java】

```
package com.atguigu;

import java.util.Set;

/**
 * Created by songhongkang on 2017/12/28 0028.
 */
public class Generator {

    public Set<String> createStrings() {
        return Set.of("Java", "9");
    }

}
```

步骤二：打包

指令如下：

```
javac -d build --release 8 src/main/java/com/atguigu/*.java
```

```
javac -d build9 --release 9 src/main/java-9/com/atguigu/*.java
```

```
jar --create --main-class=Application --file multijar.jar -C build . --release 9 -C build9 .
```

步骤三：在 **java 9** 及之前版本的环境下进行测试即可

5. 语法改进：接口的私有方法

5.1 官方 Feature

213: [Milling Project Coin](#)

Support for private methods in interfaces was briefly in consideration for inclusion in Java SE 8 as part of the effort to add support for Lambda Expressions, but was withdrawn to enable better focus on higher priority tasks for Java SE 8. It is now proposed that support for private interface methods be undertaken thereby enabling non abstract methods of an interface to share code between them.

5.2 使用说明

Java 8 中规定接口中的方法除了抽象方法之外，还可以定义静态方法和默认的方法。一定程度上，扩展了接口的功能，此时的接口更像一个抽象类。

在 Java 9 中，接口更加的灵活和强大，连方法的访问权限修饰符都可以声明为 **private** 的了，此时方法将不会成为你对外暴露的 API 的一部分。

5.3 使用举例

```
/**
 * Created by songhongkang
 */

interface MyInterface {

    void normalInterfaceMethod();

    default void methodDefault1() {
        init();
    }

    public default void methodDefault2() {
        init();
    }

    // This method is not part of the public API exposed by
    MyInterface
    private void init() {
        System.out.println("默认方法中的通用操作");
    }
}

class MyInterfaceImpl implements MyInterface{

    @Override
    public void normalInterfaceMethod() {
        System.out.println("实现接口的方法");
    }

}
```

```
public class MyInterfaceTest{
    public static void main(String[] args) {
        MyInterfaceImpl impl = new MyInterfaceImpl();
        impl.methodDefault1();
        //    impl.init();//不能调用
    }
}
```

6. 语法改进:钻石操作符(Diamond Operator)使用升级

6.1 使用说明

我们将能够与匿名实现类共同使用钻石操作符 (diamond operator)
在 java 8 中如下的操作是会报错的:

```
private List<String> flattenStrings(List<String>... lists) {

    Set<String> set = new HashSet<>(){};
    for(List<String> list : lists) {
        set.addAll(list);
    }
    return new ArrayList<>(set);
}
```

编译报错信息: '<>' cannot be used with anonymous classes

6.2 使用举例

```
private List<String> flattenStrings(List<String>... lists) {

    // anonymous classes can now use type inference
    Set<String> set = new HashSet<>(){};
    for(List<String> list : lists) {
        set.addAll(list);
    }
    return new ArrayList<>(set);
}
```

```
}
```

7. 语法改进: try 语句

7.1 使用举例

在 java 8 之前, 我们习惯于这样处理资源的关闭:

```
InputStreamReader reader = null;
try{
    reader = new InputStreamReader(System.in);

    // 流的操作

    reader.read();
}catch (IOException e){
    e.printStackTrace();
}finally{
    if(reader != null){
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

java 8 中, 可以实现资源的自动关闭, 但是要求执行后必须关闭的所有资源必须在 try 子句中初始化, 否则编译不通过。如下例所示:

```
try(InputStreamReader reader = new
InputStreamReader(System.in)){

}catch (IOException e){
    e.printStackTrace();
}
```

java 9 中，用资源语句编写 try 将更容易，我们可以在 try 子句中使用已经初始化过的资源，此时的资源是 final 的：

```
InputStreamReader reader = new InputStreamReader(System.in);
OutputStreamWriter writer = new OutputStreamWriter(System.out);
try(reader;writer){
    //reader 是 final 的，不可再被赋值
    // reader = null;
}catch (IOException e){
    e.printStackTrace();
}
```

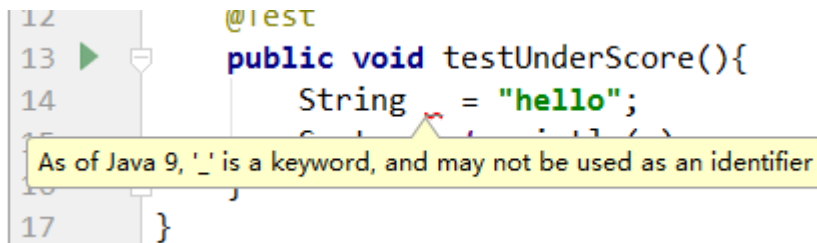
8. 语法改进：UnderScore(下划线)使用的限制

8.1 使用说明

在 java 8 中，标识符可以独立使用 “_” 来命名：

```
String _ = "hello";
System.out.println(_);
```

但是，在 java 9 中规定 “_” 不再可以单独命名标识符了，如果使用，会报错：



```
12
13  @Test
14  public void testUnderScore(){
15      String _ = "hello";
16  }
17
```

As of Java 9, '_' is a keyword, and may not be used as an identifier

9. String 存储结构变更

9.1 官方 Feature

JEP 254: [Compact Strings](#)

9.2 产生背景

Motivation

The current implementation of the `String` class stores characters in a `char` array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, that most `String` objects contain only Latin-1 characters. Such characters require only one byte of storage, hence half of the space in the internal `char` arrays of such `String` objects is going unused.

9.3 使用说明

Description

We propose to change the internal representation of the `String` class from a UTF-16 `char` array to a `byte` array plus an encoding-flag field. The new `String` class will store characters encoded either as ISO-8859-1/Latin-1 (one byte per character), or as UTF-16 (two bytes per character), based upon the contents of the string. The encoding flag will indicate which encoding is used.

结论: `String` 再也不用 `char[]` 来存储啦, 改成了 `byte[]` 加上编码标记, 节约了一些空间。

```
public final class String
```

```
implements java.io.Serializable, Comparable<String>,
CharSequence {
    @Stable
    private final byte[] value;
```

9.4 拓展: StringBuffer 与 StringBuilder

那 StringBuffer 和 StringBuilder 是否仍无动于衷呢?

String-related classes such as AbstractStringBuilder, StringBuilder, and StringBuffer will be updated to use the same representation, as will the HotSpot VM's intrinsic string operations.

10. 集合工厂方法: 快速创建只读集合

10.1 官方 Feature

269: Convenience Factory Methods for Collections

10.2 产生背景

要创建一个只读、不可改变的集合, 必须构造和分配它, 然后添加元素, 最后包装成一个不可修改的集合。

比如:

```
List<String> namesList = new ArrayList <>();
namesList.add("Joe");
namesList.add("Bob");
namesList.add("Bill");

namesList = Collections.unmodifiableList(namesList);
System.out.println(namesList);
```

缺点: 我们一下写了五行。即: 它不能表达为单个表达式。

当然，我们也可以稍微简单点处理：

```
List<String> list =  
Collections.unmodifiableList(Arrays.asList("a", "b", "c"));  
Set<String> set = Collections.unmodifiableSet(new  
HashSet<>(Arrays.asList("a", "b", "c")));  
  
//如下操作不适用于 jdk 8 及之前版本, 适用于 jdk 9  
  
Map<String,Integer> map = Collections.unmodifiableMap(new  
HashMap<>(){  
    {  
        put("a", 1);  
        put("b", 2);  
        put("c", 3);  
    }  
});  
  
map.forEach((k,v) -> System.out.println(k + ":" + v));
```

10.3 使用说明

Java 9 因此引入了方便的方法，这使得类似的事情更容易表达。

static <E> List<E>	of()	Returns an immutable list containing zero elements.
static <E> List<E>	of(E e1)	Returns an immutable list containing one element.
static <E> List<E>	of(E... elements)	Returns an immutable list containing an arbitrary number of elements
static <E> List<E>	of(E e1, E e2)	Returns an immutable list containing two elements.
static <E> List<E>	of(E e1, E e2, E e3)	Returns an immutable list containing three elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4)	Returns an immutable list containing four elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5)	Returns an immutable list containing five elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6)	Returns an immutable list containing six elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)	Returns an immutable list containing seven elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)	Returns an immutable list containing eight elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)	Returns an immutable list containing nine elements.
static <E> List<E>	of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)	Returns an immutable list containing ten elements.

```
List firstnamesList = List.of("Joe","Bob","Bill");
```

调用集合中静态方法 `of()`，可以将不同数量的参数传输到此工厂方法

中。此功能可用于 Set 和 List，也可用于 Map 的类似形式。此时得到的集合，是不可变的：在创建后，继续添加元素到这些集合会导致“`UnsupportedOperationException`”。

由于 Java 8 中接口方法的实现，可以直接在 List，Set 和 Map 的接口内定义这些方法，便于调用。

10.4 使用举例

```
List<String> list = List.of("a", "b", "c");
Set<String> set = Set.of("a", "b", "c");

Map<String, Integer> map1 = Map.of("Tom", 12, "Jerry", 21,
    "Lilei", 33, "HanMeimei", 18);

Map<String, Integer> map2 = Map.ofEntries(
    Map.entry("Tom", 89),
    Map.entry("Jim", 78),
    Map.entry("Tim", 98)
);
```

11. 增强的 Stream API

11.1 使用说明

Java 的 **Stream API** 是java标准库最好的改进之一，让开发者能够快速运算，从而能够有效的利用数据并行计算。Java 8 提供的 Stream 能够利用多核架构实现声明式的数据处理。

在 Java 9 中，Stream API 变得更好，Stream 接口中添加了 4 个新的方法：**dropWhile, takeWhile, ofNullable**，还有个 **iterate** 方法的新重载方法，可以让你提供一个 Predicate (判断条件)来指定什么时候结束迭代。（见下例）

除了对 Stream 本身的扩展，Optional 和 Stream 之间的结合也得到了改进。现在可以通过 Optional 的新方法 stream() 将一个 Optional 对象转换为一个(可能是空的) Stream 对象。（见下例）

11.2 使用举例

takeWhile()的使用:

用于从 Stream 中获取一部分数据，接收一个 Predicate 来进行选择。在有序的 Stream 中，takeWhile 返回从开头开始的尽量多的元素。

```
List<Integer> list =  
Arrays.asList(45,43,76,87,42,77,90,73,67,88);  
list.stream().takeWhile(x -> x < 50)  
    .forEach(System.out::println);  
  
System.out.println();  
  
list = Arrays.asList(1,2,3,4,5,6,7,8);  
list.stream().takeWhile(x -> x < 5)  
    .forEach(System.out::println);
```

dropWhile()的使用:

dropWhile 的行为与 takeWhile 相反，返回剩余的元素。

```
List<Integer> list =  
Arrays.asList(45,43,76,87,42,77,90,73,67,88);  
list.stream().dropWhile(x -> x < 50)  
    .forEach(System.out::println);  
  
System.out.println();  
  
list = Arrays.asList(1,2,3,4,5,6,7,8);  
list.stream().dropWhile(x -> x < 5)  
    .forEach(System.out::println);
```

ofNullable()的使用:

Java 8 中 Stream 不能完全为 null，否则会报空指针异常。而 Java 9 中的

ofNullable 方法允许我们创建一个单元素 Stream，可以包含一个非空元素，也可以创建一个空 Stream。

```
//报 NullPointerException
//Stream<Object> stream1 = Stream.of(null);
//System.out.println(stream1.count());

//不报异常，允许通过
Stream<String> stringStream = Stream.of("AA", "BB", null);
System.out.println(stringStream.count());//3

//不报异常，允许通过
List<String> list = new ArrayList<>();
list.add("AA");
list.add(null);
System.out.println(list.stream().count());//2

//ofNullable(): 允许值为null
Stream<Object> stream1 = Stream.ofNullable(null);
System.out.println(stream1.count());//0

Stream<String> stream = Stream.ofNullable("hello world");
System.out.println(stream.count());//1
```

iterator()重载的使用:

原来的控制终止方式:

```
Stream.iterate(1,i -> i + 1).limit(10)
    .forEach(System.out::println);
```

现在的终止方式:

```
Stream.iterate(1,i -> i < 100,i -> i + 1)
    .forEach(System.out::println);
```

Optional 类中 stream()的使用:

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

```
list.add("Tom");  
list.add("Jerry");  
list.add("Tim");  
  
Optional<List<String>> optional =  
Optional.ofNullable(list);  
Stream<List<String>> stream = optional.stream();  
stream.flatMap(x ->  
x.stream()).forEach(System.out::println);
```

12. 多分辨率图像 API

12.1 官方 Feature

251: [Multi-Resolution Images](#)

263: [HiDPI Graphics on Windows and Linux](#)

12.2 产生背景

在 Mac 上, JDK 已经支持视网膜显示, 但在 Linux 和 Windows 上, 它并没有。在那里, Java 程序在当前的高分辨率屏幕上可能看起来很小, 不能使用它们。这是因为像素用于这些系统的大小计算 (无论像素实际有多大)。毕竟, 高分辨率显示器的有效部分是像素非常小。

JEP 263 以这样的方式扩展了 JDK, 即 Windows 和 Linux 也考虑到像素的大小。为此, 使用比现在更多的现代 API: Direct2D for Windows 和 GTK +, 而不是 Xlib for Linux。图形, 窗口和文本由此自动缩放。

JEP 251 还提供处理多分辨率图像的能力, 即包含不同分辨率的相同图像的文件。根据相应屏幕的 DPI 度量, 然后以适当的分辨率使用图像。

12.3 使用说明

- 新的 API 定义在 `java.awt.image` 包下
- 将不同分辨率的图像封装到一张（多分辨率的）图像中，作为它的变体
- 获取这个图像的所有变体
- 获取特定分辨率的图像变体-表示一张已知分辨率单位为 DPI 的特定尺寸大小的逻辑图像，并且这张图像是最佳的变体。
- 基于当前屏幕分辨率大小和运用的图像转换算法，`java.awt.Graphics` 类可以从接口 `MultiResolutionImage` 获取所需的变体。
- `MultiResolutionImage` 的基础实现是 `java.awt.image.BaseMultiResolutionImage`。

13. 全新的 HTTP 客户端 API

13.1 官方 Feature

110: [HTTP 2 Client](#)

13.2 使用说明

HTTP，用于传输网页的协议，早在 1997 年就被采用在目前的 1.1 版本中。直到 2015 年，HTTP2 才成为标准。



HTTP/1.1 和 HTTP/2 的主要区别是如何在客户端和服务端之间构建和传输数据。**HTTP/1.1** 依赖于请求/响应周期。**HTTP/2** 允许服务器“push”数据：它可以发送比客户端请求更多的数据。这使得它可以优先处理并发送对于首先加载网页至关重要的数据。

Java 9 中有新的方式来处理 **HTTP** 调用。它提供了一个新的 **HTTP** 客户端（**HttpClient**），它将替代仅适用于 **blocking** 模式的 **URLConnection**（**URLConnection** 是在 **HTTP 1.0** 的时代创建的，并使用了协议无关的方法），并提供对 **WebSocket** 和 **HTTP/2** 的支持。

此外，**HTTP** 客户端还提供 **API** 来处理 **HTTP/2** 的特性，比如流和服务端推送等功能。

全新的 **HTTP** 客户端 **API** 可以从 **jdk.incubator.httpclient** 模块中获取。因为在默认情况下，这个模块是不能根据 **classpath** 获取的，需要使用 **add modules** 命令选项配置这个模块，将这个模块添加到 **classpath** 中。

13.3 使用举例

举例：

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest req =

HttpRequest.newBuilder(URI.create("http://www.atguigu.com"
))
    .GET()
    .build();
HttpResponse<String> response = client.send(req,
HttpRequest.BodyHandler.asString());
System.out.println(response.statusCode());
System.out.println(response.version().name());
System.out.println(response.body());
```

14. Deprecated 的相关 API

14.1 官方 Feature

211: [Elide Deprecation Warnings on Import Statements](#)

214: [Remove GC Combinations Deprecated in JDK 8](#)

277: [Enhanced Deprecation](#)

289: [Deprecate the Applet API](#)

291: [Deprecate the Concurrent Mark Sweep \(CMS\) Garbage Collector](#)

14.2 使用说明

Java 9 废弃或者移除了几个不常用的功能。其中最主要的是 **Applet API**，现在是标记为废弃的。随着对安全要求的提高，主流浏览器已经取消对 Java 浏览器插件的支持。HTML5 的出现也进一步加速了它的消亡。开发者现在可以使用像 Java Web Start 这样的技术来代替 Applet，它可以实现从浏览器启动应用程序或者安装应用程序。

同时，appletviewer 工具也被标记为废弃。

15. 智能 Java 编译工具

15.1 官方 Feature

139: [Enhance javac to Improve Build Speed.](#)

199: [Smart Java Compilation, Phase Two](#)

15.2 使用说明

智能 java 编译工具(**sjavac**)的第一个阶段始于 JEP139 这个项目，用于在多核处理器情况下提升 JDK 的编译速度。如今，这个项目已经进入第二阶段，即 JEP199，其目的是改进 Java 编译工具，并取代目前 JDK 编译工具 **javac**，继而成为 Java 环境默认的通用的智能编译工具。

JDK 9 还更新了 **javac** 编译器以便能够将 java 9 代码编译运行在低版本 Java 中。

16. 统一的 JVM 日志系统

16.1 官方 Feature

158: [Unified JVM Logging](#)

271: [Unified GC Logging](#)

16.2 使用说明

日志是解决问题的唯一有效途径：曾经很难知道导致 JVM 性能问题和导致 JVM 崩溃的根本原因。不同的 JVM 日志的碎片化和日志选项（例如：JVM 组件对于日志使用的是不同的机制和规则），这使得 JVM 难以进行调试。

解决该问题最佳方法：对所有的 JVM 组件引入一个单一的系统，这些 JVM 组件支持细粒度的和易配置的 JVM 日志。

17. javadoc 的 HTML 5 支持

17.1 官方 Feature

224: [HTML5 Javadoc](#)

225: [Javadoc Search](#)

17.2 使用说明

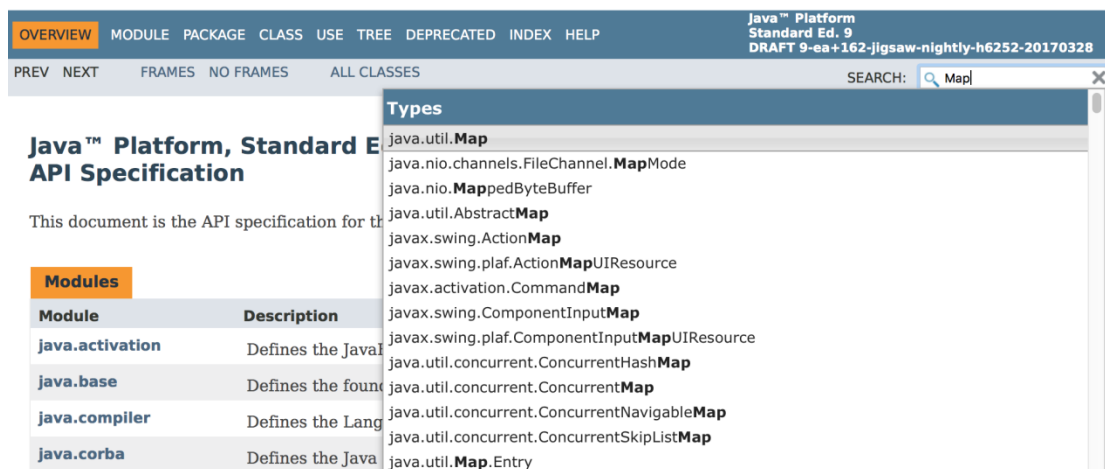
jdk 8 : 生成的 java 帮助文档是在 HTML 4 中, 而 HTML 4 已经是很久的标准了。

jdk 9 : javadoc 的输出, 现在符合兼容 HTML 5 标准。

下图是 java8 中生成的 html 页面, 如果想要找到一些类文档, 必须在 google 中搜索。



下图是在 java 9 中, 添加了一个搜索框。



18. Javascript 引擎升级：Nashorn

18.1 官方 Feature

236: [Parser API for Nashorn](#)

292: [Implement Selected ECMAScript 6 Features in Nashorn](#)

18.2 使用说明

Nashorn 项目在 JDK 9 中得到改进，它为 Java 提供轻量级的 Javascript 运行时。Nashorn 项目跟随 Netscape 的 Rhino 项目，目的是为了在 Java 中实现一个高性能但轻量级的 Javascript 运行时。Nashorn 项目使得 Java 应用能够嵌入 Javascript。它在 JDK 8 中为 Java 提供一个 Javascript 引擎。

JDK 9 包含一个用来解析 Nashorn 的 ECMAScript 语法树的 API。这个 API 使得 IDE 和服务端框架不需要依赖 Nashorn 项目的内部实现类，就能够分析 ECMAScript 代码。

19. java 的动态编译器

19.1 官方 Feature

243: [Java-Level JVM Compiler Interface](#)

295: [Ahead-of-Time Compilation](#)

19.2 产生背景

Oracle 一直在努力提高 Java 启动和运行时性能，希望其能够在更广泛的场景达到或接近本地语言的性能。但是，直到今天，谈到 Java，很多 C/C++ 开发者还是会不屑地评价为启动慢，吃内存。

简单说，这主要是因为 Java 编译产生的类文件是 Java 虚拟机可以理解的二进制代码，而不是真正的可执行的本地代码，需要 Java 虚拟机进行解释和编译，这带来了额外的开销。

19.3 使用说明

JIT (Just-in-time) 编译器可以在运行时将热点编译成本地代码，速度很快。但是 Java 项目现在变得很大很复杂，因此 JIT 编译器需要花费较长时间才能热身完，而且有些 Java 方法还没法编译，性能方面也会下降。AoT 编译就是为了解决这些问题而生的。

在 JDK 9 中，AOT (JEP 295: Ahead-of-Time Compilation) 作为实验特性被引入进来，开发者可以利用新的 jaotc 工具将重点代码转换成类似类库一样的文件。虽然仍处于试验阶段，但这个功能使得 Java 应用在被虚拟机启动之前能够先将 Java 类编译为原生代码。此功能旨在改进小型和大型应用程序的启动时间，同时对峰值性能的影响很小。

但是 Java 技术供应商 Excelsior 的营销总监 Dmitry Leskov 担

心 **AoT 编译技术不够成熟**，希望 Oracle 能够等到 **Java 10** 时有个更稳定版本才发布。

另外 JVMCI (JEP 243: Java-Level JVM Compiler Interface) 等特性，对于整个编程语言的发展，可能都具有非常重要的意义，虽然未必引起了广泛关注。目前 Graal Core API 已经被集成进入 Java 9，虽然还只是初始一小步，但是完全用 Java 语言来实现的可靠的、高性能的动态编译器，似乎不再是遥不可及，这是 Java 虚拟机开发工程师的福音。

与此同时，随着 Truffle 框架和 Substrate VM 的发展，已经让个别信心满满的工程师高呼 “**One VM to Rule Them All!**”，也许就在不久的将来 Ploygot 以一种另类的方式成为现实。

四、总结

1. 在 java 9 中看不到什么？

1.1 一个标准化和轻量级的 JSON API

一个**标准化和轻量级的 JSON API** 被许多 java 开发人员所青睐。但是由于资金问题无法在 Java 9 中见到，但并不会削减掉。Java 平台首席架构师 Mark Reinhold 在 JDK 9 邮件列中说：“这个 JEP 将是平台上的一个有用的补充，但是在计划中，它并不像 Oracle 资助的其他功能那么重要，可能会重新考虑 JDK 10 或更高版本中实现。”

1.2 新的货币 API

对许多应用而言货币价值都是一个关键的特性，但 JDK 对此却几乎没有任何支持。严格来讲，现有的 `java.util.Currency` 类只是代表了

当前 ISO 4217 货币的一个数据结构,但并没有关联的值或者自定义货币。JDK 对货币的运算及转换也没有内建的支持,更别说有一个能够代表货币值的标准类型了。

此前, Oracle 公布的 JSR 354 定义了一套新的 Java 货币 API: **JavaMoney**, 计划会在 Java 9 中正式引入。但是目前没有出现 JDK 9 中。

不过,如果你用的是 Maven 的话,可以做如下的添加,即可使用相关的 API 处理货币:

```
<dependency>

  <groupId>org.javamoney</groupId>

  <artifactId>moneta</artifactId>

  <version>0.9</version>

</dependency>
```

代码参考,可以访问 <https://github.com/JavaMoney>,里面已经给出了使用说明和示例。

2. 展望

- 随着云计算和 AI 等技术浪潮,当前的计算模式和场景正在发生翻天覆地的变化,不仅对 Java 的发展速度提出了更高要求,也深刻影响着 Java 技术的发展方向。传统的大型企业或互联网应用,正在被云端、容器化应用、模块化的微服务甚至是函数(FaaS, Function-as-a-Service)所替代。
- Java虽然标榜面向对象编程,却毫不顾忌的加入面向接口编程思想,又扯出匿名对象之概念,每增加一个新的东西,对Java的根

本所在的面向对象思想的一次冲击。反观Python，抓住面向对象的本质，又能在函数编程思想方面游刃有余。Java对标C/C++，以抛掉内存管理为卖点，却又陷入了JVM优化的噩梦。选择比努力更重要，选择Java的人更需要对它有更清晰的认识。

- Java 需要新的计算场景下，改进开发效率。这话说的有点笼统，我谈一些自己的体会，Java 代码虽然进行了一些类型推断等改进，更易用的集合 API 等，但仍然给开发者留下了过于刻板、形式主义的印象，这是一个长期的改进方向。