Java 5-11 各个版本新特性最全总结

讲师：尚硅谷 – 宋红康

Java 5

Java5 开发代号为 Tiger(老虎),于 2004-09-30 发行

特性列表

- 泛型
- 枚举
- 自动装箱拆箱
- 可变参数
- 注解
- foreach 循环（增强 for、for/in）
- 静态导入
- 格式化（System.out.println 支持%s %d 等格式化输出）
- 线程框架/数据结构 JUC
- Arrays 工具类/StringBuilder/instrument

1、泛型

所谓类型擦除指的就是 Java 源码中的范型信息只允许停留在编译前期，而编译后的字节码文件中将不再保留任何的范型信息。也就是说，范型信息在编译时将会被全部删除，其中范型类型的类型参数则会被替换为 Object 类型，并在实际使用时强制转换为指定的目标数据类型。而 C++中的模板则会在编译时将模板类型中的类型参数根据所传递的指定数据类型生成相对应的目标代码。

```
Map<Integer, Integer> squares = new HashMap<Integer, Integer>();
```

通配符类型：避免 unchecked 警告，问号表示任何类型都可以接受

```
public void printList(List<?> list, PrintStream out) throws
IOException {
 for (Iterator<?> i = list.iterator(); i.hasNext(); ) {
 out.println(i.next().toString());
 }
 }
```

限制类型

```
public static <A extends Number> double sum(Box<A> box1,Box<A>
box2){
 double total = 0;
 for (Iterator<A> i = box1.contents.iterator(); i.hasNext(); ) {
 total = total + i.next().doubleValue();
 }
```

```
for (Iterator<A> i = box2.contents.iterator(); i.hasNext(); ) {
total = total + i.next().doubleValue();
}
return total;
}
```

2、枚举

EnumMap

```
public void testEnumMap(PrintStream out) throws IOException {
// Create a map with the key and a String message
EnumMap<AntStatus, String> antMessages =
new EnumMap<AntStatus, String>(AntStatus.class);
// Initialize the map
antMessages.put(AntStatus.INITIALIZING, "Initializing Ant...");
antMessages.put(AntStatus.COMPILING, "Compiling Java
classes...");
antMessages.put(AntStatus.COPYING, "Copying files...");
antMessages.put(AntStatus.JARRING, "JARring up files...");
antMessages.put(AntStatus.ZIPPING, "ZIPping up files...");
antMessages.put(AntStatus.DONE, "Build complete.");
antMessages.put(AntStatus.ERROR, "Error occurred.");
// Iterate and print messages
for (AntStatus status : AntStatus.values() ) {
out.println("For status " + status + ", message is: " +
antMessages.get(status));
}
}
```

switch 枚举

```
public String getDescription() {
 switch(this) {
 case ROSEWOOD: return "Rosewood back and sides";
 case MAHOGANY: return "Mahogany back and sides";
 case ZIRICOTE: return "Ziricote back and sides";
 case SPRUCE: return "Sitka Spruce top";
 case CEDAR: return "Wester Red Cedar top";
 case AB_ROSETTE: return "Abalone rosette";
 case AB_TOP_BORDER: return "Abalone top border";
 case IL_DIAMONDS:
 return "Diamonds and squares fretboard inlay";
 case IL_DOTS:
 return "Small dots fretboard inlay";
 default: return "Unknown feature";
 }
 }
```

3、自动拆箱/装箱

将 primitive 类型转换成对应的 wrapper 类型：Boolean、Byte、Short、Character、Integer、Long、Float、Double

4、可变参数

```
private String print(Object... values) {
 StringBuilder sb = new StringBuilder();
 for (Object o : values) {
 sb.append(o.toString())
 .append(" ");
 }
 return sb.toString();
 }
```

5、注解

Inherited 表示该注解是否对类的子类继承的方法等起作用

```
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface InProgress { }
```

Target 类型

Rentation 表示 annotation 是否保留在编译过的 class 文件中还是在运行时可读。

6、增强 for 循环 for/in

for/in 循环办不到的事情：

（1）遍历同时获取 index

（2）集合逗号拼接时去掉最后一个

（3）遍历的同时删除元素

7、静态导入

```
import static java.lang.System.err;
import static java.lang.System.out;
err.println(msg);
```

8、print 输出格式化

```
System.out.println("Line %d: %s%n", i++, line);
```

9、并发支持（JUC）

线程池

uncaught exception（可以抓住多线程内的异常）

```
class SimpleThreadExceptionHandler implements
 Thread.UncaughtExceptionHandler {
 public void uncaughtException(Thread t, Throwable e) {
 System.err.printf("%s: %s at line %d of %s%n",
 t.getName(),
 e.toString(),
 e.getStackTrace()[0].getLineNumber(),
 e.getStackTrace()[0].getFileName());
 }
```

blocking queue(BlockingQueue)

JUC 类库

10、Arrays、Queue、线程安全 StringBuilder

Arrays 工具类

```
Arrays.sort(myArray);
Arrays.toString(myArray)
Arrays.binarySearch(myArray, 98)
Arrays.deepToString(ticTacToe)
Arrays.deepEquals(ticTacToe, ticTacToe3)
```

Queue

避开集合的 add/remove 操作，使用 offer、poll 操作（不抛异常）

Queue 接口与 List、Set 同一级别，都是继承了 Collection 接口。LinkedList 实现了 Deque 接口。

```
Queue q = new LinkedList(); //采用它来实现 queue
```

Override 返回类型

单线程 StringBuilder

java.lang.instrument

Java 6

Java6 开发代号为 Mustang(野马),于 2006-12-11 发行。评价：鸡肋的版本，有 JDBC4.0 更新、Complier API、WebSevice 支持的加强等更新。

1、Web Services

优先支持编写 XML web service 客户端程序。你可以用过简单的 annotaion 将你的 API 发布成.NET 交互的 web services. Mustang 添加了新的解析和 XML 在 Java object-mapping APIs 中，之前只在 Java EE 平台实现或者 Java Web Services Pack 中提供.

2、Scripting（开启 JS 的支持，算是比较有用的）

现在你可以在 Java 源代码中混入 JavaScript 了，这对开发原型很有有用，你也可以插入自己的脚本引擎。

3、Database

Mustang 将联合绑定 Java DB (Apache Derby). JDBC 4.0 增加了许多特性例如支持 XML 作为 SQL 数据类型 ,更好的集成 Binary Large OBjects (BLOBs) 和 Character Large OBjects (CLOBs) .

4、More Desktop APIs

GUI 开发者可以有更多的技巧来使用 SwingWorker utility ,以帮助 GUI 应用中的多线程。, JTable 分类和过滤，以及添加 splash 闪屏。

很显然，这对于主攻服务器开发的 Java 来说，并没有太多吸引力

5、Monitoring and Management

绑定了不是很知名的 memory-heap 分析工具 Jhat 来查看内核导出。

6、Compiler Access（这个很厉害）

compiler API 提供编程访问 javac ,可以实现进程内编译 ,动态产生 Java 代码。

7、Pluggable Annotation

8、Desktop Deployment

Swing 拥有更好的 look-and-feel , LCD 文本呈现, 整体 GUI 性能的提升。Java 应用程序可以和本地平台更好的集成，例如访问平台的系统托盘和开始菜单。Mustang 将 Java 插件技术和 Java Web Start 引擎统一了起来。

9、Security

XML-数字签名(XML-DSIG) APIs 用于创建和操纵数字签名); 新的方法来访问本地平台的安全服务

10、The -ilities（很好的习惯）

质量，兼容性，稳定性。 80,000 test cases 和数百万行测试代码(只是测试活动中的一个方面). Mustang 的快照发布已经被下载 15 个月了 ,每一步中的 Bug 都被修复了，表现比 J2SE 5 还要好。

Java 7

特性列表

- switch 中添加对 String 类型的支持
- 数字字面量的改进 / 数值可加下划
- 异常处理（捕获多个异常） try-with-resources
- 增强泛型推断
- JSR203 NIO2.0（AIO）新 IO 的支持
- JSR292 与 InvokeDynamic 指令
- Path 接口、DirectoryStream、Files、WatchService（重要接口更新）
- fork/join framework

1、switch 中添加对 String 类型的支持

```
String title = "";
switch (gender) {
case "男":
title = name + " 先生";
break;
```

```
 case "女":
 title = name + " 女士";
 break;
 default:
 title = name;
 }
 return title;
}
```

编译器在编译时先做处理：

①case 仅仅有一种情况。直接转成 if。

②假设仅仅有一个 case 和 default，则直接转换为 if...else...。

③有多个 case。先将 String 转换为 hashCode 然后相应的进行处理 ,JavaCode 在底层兼容 Java7 曾经版本号。

2、数字字面量的改进

Java7 前支持十进制（123）、八进制（0123）、十六进制（0X12AB）

Java7 添加二进制表示（0B11110001、0b11110001）

数字中可加入分隔符

Java7 中支持在数字量中间添加’_'作为分隔符。更直观，如（12_123_456）。下划线仅仅能在数字中间。编译时编译器自己主动删除数字中的下划线。

```
int one_million = 1_000_000;
```

3、异常处理（捕获多个异常） try-with-resources

catch 子句能够同一时候捕获多个异常

```
public void testSequence() {
 try {
 Integer.parseInt("Hello");
 }
 catch (NumberFormatException | RuntimeException e) { //使用'|'
切割，多个类型，一个对象 e
 }
 }
```

try-with-resources 语句

Java7 之前须要在 finally 中关闭 socket、文件、数据库连接等资源；

Java7 中在 try 语句中申请资源，实现资源的自己主动释放（资源类必须实现 java.lang.AutoCloseable 接口，一般的文件、数据库连接等均已实现该接口，close 方法将被自己主动调用）。

```
public void read(String filename) throws IOException {
 try (BufferedReader reader = new BufferedReader(new
FileReader(filename))) {
 StringBuilder builder = new StringBuilder();
 String line = null;
 while((line=reader.readLine())!=null){
 builder.append(line);
 builder.append(String.format("%n"));
 }
 return builder.toString();
 }
 }
```

4、增强泛型推断

```
Map<String, List<String>> map = new HashMap<String,
List<String>>();
```

Java7 之后可以简单的这么写

```
Map<String, List<String>> anagrams = new HashMap<>();
```

5、NIO2.0（AIO）新 IO 的支持

bytebuffer

```
public class ByteBufferUsage {
 public void useByteBuffer() {
 ByteBuffer buffer = ByteBuffer.allocate(32);
 buffer.put((byte)1);
 buffer.put(new byte[3]);
 buffer.putChar('A');
 buffer.putFloat(0.0f);
 buffer.putLong(10, 100L);
 System.out.println(buffer.getChar(4));
 System.out.println(buffer.remaining());
 }
 public void byteOrder() {
 ByteBuffer buffer = ByteBuffer.allocate(4);
 buffer.putInt(1);
 buffer.order(ByteOrder.LITTLE_ENDIAN);
 buffer.getInt(0); //值为 16777216
 }
 public void compact() {
 ByteBuffer buffer = ByteBuffer.allocate(32);
 buffer.put(new byte[16]);
 buffer.flip();
 buffer.getInt();
 buffer.compact();
```

```
 int pos = buffer.position();
 }
 public void viewBuffer() {
 ByteBuffer buffer = ByteBuffer.allocate(32);
 buffer.putInt(1);
 IntBuffer intBuffer = buffer.asIntBuffer();
 intBuffer.put(2);
 int value = buffer.getInt(); //值为 2
 }
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 ByteBufferUsage bbu = new ByteBufferUsage();
 bbu.useByteBuffer();
 bbu.byteOrder();
 bbu.compact();
 bbu.viewBuffer();
 }
}
```

filechannel

```
public class FileChannelUsage {
 public void openAndWrite() throws IOException {
 FileChannel channel = FileChannel.open(Paths.get("my.txt"),
StandardOpenOption.CREATE, StandardOpenOption.WRITE);
 ByteBuffer buffer = ByteBuffer.allocate(64);
 buffer.putChar('A').flip();
 channel.write(buffer);
 }
 public void readWriteAbsolute() throws IOException {
 FileChannel channel =
FileChannel.open(Paths.get("absolute.txt"),
StandardOpenOption.READ, StandardOpenOption.CREATE,
StandardOpenOption.WRITE);
 ByteBuffer writeBuffer =
ByteBuffer.allocate(4).putChar('A').putChar('B');
 writeBuffer.flip();
 channel.write(writeBuffer, 1024);
```

```
 ByteBuffer readBuffer = ByteBuffer.allocate(2);
 channel.read(readBuffer, 1026);
 readBuffer.flip();
 char result = readBuffer.getChar(); //值为'B'
 }
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) throws IOException {
 FileChannelUsage fcu = new FileChannelUsage();
 fcu.openAndWrite();
 fcu.readWriteAbsolute();
 }
}
```

6、JSR292 与 InvokeDynamic

JSR 292: Supporting Dynamically Typed Languages on the JavaTM Platform，支持在 JVM 上运行动态类型语言。在字节码层面支持了 InvokeDynamic。

```
public class ThreadPoolManager {
 private final ScheduledExecutorService stpe = Executors
 .newScheduledThreadPool(2);
 private final BlockingQueue<WorkUnit<String>> lbq;
 public ThreadPoolManager(BlockingQueue<WorkUnit<String>> lbq_)
{
 lbq = lbq_;
 }
 public ScheduledFuture<?> run(QueueReaderTask msgReader) {
 msgReader.setQueue(lbq);
 return stpe.scheduleAtFixedRate(msgReader, 10, 10,
TimeUnit.MILLISECONDS);
 }
 private void cancel(final ScheduledFuture<?> hndl) {
 stpe.schedule(new Runnable() {
 public void run() {
```

```
 hndl.cancel(true);
 }
 }, 10, TimeUnit.MILLISECONDS);
 }
 /**
 * 使用传统的反射 api
 */
 public Method makeReflective() {
 Method meth = null;
 try {
 Class<?>[] argTypes = new Class[]{ScheduledFuture.class};
 meth = ThreadPoolManager.class.getDeclaredMethod("cancel",
argTypes);
 meth.setAccessible(true);
 } catch (IllegalArgumentException | NoSuchMethodException
 | SecurityException e) {
 e.printStackTrace();
 }
 return meth;
 }
 /**
 * 使用代理类
 * @return
 */
 public CancelProxy makeProxy() {
 return new CancelProxy();
 }
 /**
 * 使用 Java7 的新 api，MethodHandle
 * invoke virtual 动态绑定后调用 obj.xxx
 * invoke special 静态绑定后调用 super.xxx
 * @return
 */
 public MethodHandle makeMh() {
 MethodHandle mh;
 MethodType desc = MethodType.methodType(void.class,
ScheduledFuture.class);
 try {
 mh =
MethodHandles.lookup().findVirtual(ThreadPoolManager.class,
 "cancel", desc);
 } catch (NoSuchMethodException | IllegalAccessException e) {
```

```
 throw (AssertionError) new AssertionError().initCause(e);
 }
 return mh;
 }
 public static class CancelProxy {
 private CancelProxy() {
 }
 public void invoke(ThreadPoolManager mae_, ScheduledFuture<?>
hndl_) {
 mae_.cancel(hndl_);
 }
 }
}
```

调用 invoke

```
public class ThreadPoolMain {
 /**
 * 如果被继承，还能在静态上下文寻找正确的 class
 */
 private static final Logger logger =
LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
 private ThreadPoolManager manager;
 public static void main(String[] args) {
 ThreadPoolMain main = new ThreadPoolMain();
 main.run();
 }
 private void cancelUsingReflection(ScheduledFuture<?> hndl) {
 Method meth = manager.makeReflective();
 try {
 System.out.println("With Reflection");
 meth.invoke(hndl);
 } catch (IllegalAccessException | IllegalArgumentException
 | InvocationTargetException e) {
 e.printStackTrace();
 }
 }
 private void cancelUsingProxy(ScheduledFuture<?> hndl) {
 CancelProxy proxy = manager.makeProxy();
 System.out.println("With Proxy");
```

```
 proxy.invoke(manager, hndl);
 }
 private void cancelUsingMH(ScheduledFuture<?> hndl) {
 MethodHandle mh = manager.makeMh();
 try {
 System.out.println("With Method Handle");
 mh.invokeExact(manager, hndl);
 } catch (Throwable e) {
 e.printStackTrace();
 }
 }
 private void run() {
 BlockingQueue<WorkUnit<String>> lbq = new
LinkedBlockingQueue<>();
 manager = new ThreadPoolManager(lbq);
 final QueueReaderTask msgReader = new QueueReaderTask(100) {
 @Override
 public void doAction(String msg_) {
 if (msg_ != null)
 System.out.println("Msg recvd: " + msg_);
 }
 };
 ScheduledFuture<?> hndl = manager.run(msgReader);
 cancelUsingMH(hndl);
 // cancelUsingProxy(hndl);
 // cancelUsingReflection(hndl);
 }
}
```

7、Path 接口(重要接口更新)

Path

```
public class PathUsage {
 public void usePath() {
 Path path1 = Paths.get("folder1", "sub1");
 Path path2 = Paths.get("folder2", "sub2");
 path1.resolve(path2); //folder1sub1
```

```
older2sub2
 path1.resolveSibling(path2); //folder1
```

Java 8

Java 8 可谓是自 Java 5 以来最具革命性的版本了，她在语言、编译器、类库、开发工具以及 Java 虚拟机等方面都带来了不少新特性。

一、Lambda 表达式

Lambda 表达式可以说是 Java 8 最大的卖点，她将函数式编程引入了 Java。Lambda 允许把函数作为一个方法的参数，或者把代码看成数据。

一个 Lambda 表达式可以由用逗号分隔的参数列表、–>符号与函数体三部分表示。例如：

```
Arrays.asList( "p", "k", "u","f", "o", "r","k").forEach( e ->
System.out.println( e ) );
```

为了使现有函数更好的支持 Lambda 表达式，Java 8 引入了函数式接口的概念。函数式接口就是只有一个方法的普通接口。java.lang.Runnable 与

java.util.concurrent.Callable 是函数式接口最典型的例子。为此，Java 8 增加了一种特殊的注解@FunctionalInterface

二、接口的默认方法与静态方法

我们可以在接口中定义默认方法，使用 default 关键字，并提供默认的实现。所有实现这个接口的类都会接受默认方法的实现，除非子类提供的自己的实现。例如：

```
public interface DefaultFunctionInterface {
 default String defaultFunction() {
 return "default function";
 }
 }
```

我们还可以在接口中定义静态方法，使用 static 关键字，也可以提供实现。例如：

```
public interface StaticFunctionInterface {
 static String staticFunction() {
 return "static function";
 }
 }
```

接口的默认方法和静态方法的引入，其实可以认为引入了 C + + 中抽象类的理念，以后我们再也不用在每个实现类中都写重复的代码了。

三、方法引用（含构造方法引用）

通常与 Lambda 表达式联合使用，可以直接引用已有 Java 类或对象的方法。一般有四种不同的方法引用：

构造器引用。语法是 Class::new，或者更一般的 Class< T >::new，要求构造器方法是没有参数；

静态方法引用。语法是 Class::static_method 要求接受一个 Class 类型的参数；

特定类的任意对象方法引用。它的语法是 Class::method。要求方法是没有参数的；

特定对象的方法引用，它的语法是 instance::method。要求方法接受一个参数，与 3 不同的地方在于，3 是在列表元素上分别调用方法，而 4 是在某个对象上调用方法，将列表元素作为参数传入；

四、重复注解

在 Java 5 中使用注解有一个限制，即相同的注解在同一位置只能声明一次。Java 8 引入重复注解，这样相同的注解在同一地方也可以声明多次。重复注解机制本身需要用@Repeatable 注解。Java 8 在编译器层做了优化，相同注解会以集合的方式保存，因此底层的原理并没有变化。

五、扩展注解的支持（类型注解）

Java 8 扩展了注解的上下文，几乎可以为任何东西添加注解，包括局部变量、泛型类、父类与接口的实现，连方法的异常也能添加注解。

```
private @NotNull String name;
```

六、Optional

Java 8 引入 Optional 类来防止空指针异常，Optional 类最先是由 Google 的 Guava 项目引入的。Optional 类实际上是个容器：它可以保存类型 T 的值，或者保存 null。使用 Optional 类我们就不用显式进行空指针检查了。

七、Stream

Stream API 是把真正的函数式编程风格引入到 Java 中。其实简单来说可以把 Stream 理解为 MapReduce，当然 Google 的 MapReduce 的灵感也是来自函数式编程。她其实是一连串支持连续、并行聚集操作的元素。从语法上看，也很像 linux 的管道、或者链式编程，代码写起来简洁明了，非常酷帅！

八、Date/Time API (JSR 310)

Java 8 新的 Date-Time API (JSR 310)受 Joda-Time 的影响，提供了新的 java.time 包，可以用来替代 java.util.Date 和 java.util.Calendar。一般会用到 Clock、LocaleDate、LocalTime、LocaleDateTime、ZonedDateTime、Duration 这些类，对于时间日期的改进还是非常不错的。

九、JavaScript 引擎 Nashorn

Nashorn 允许在 JVM 上开发运行 JavaScript 应用，允许 Java 与 JavaScript 相互调用。

十、Base64

在 Java 8 中，Base64 编码成为了 Java 类库的标准。Base64 类同时还提供了对 URL、MIME 友好的编码器与解码器。

说在后面

除了这十大特性，还有另外的一些新特性：

更好的类型推测机制：Java 8 在类型推测方面有了很大的提高，这就使代码更整洁，不需要太多的强制类型转换了。

编译器优化：Java 8 将方法的参数名加入了字节码中，这样在运行时通过反射就能获取到参数名，只需要在编译时使用-parameters 参数。

并行（parallel）数组：支持对数组进行并行处理，主要是 parallelSort()方法，它可以在多核机器上极大提高数组排序的速度。

并发（Concurrency）：在新增 Stream 机制与 Lambda 的基础之上，加入了一些新方法来支持聚集操作。

Nashorn 引擎 jjs：基于 Nashorn 引擎的命令行工具。它接受一些 JavaScript 源代码为参数，并且执行这些源代码。

类依赖分析器 jdeps：可以显示 Java 类的包级别或类级别的依赖。

JVM 的 PermGen 空间被移除：取代它的是 Metaspace（JEP 122）。

Java 9

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（距离上个版本足足 3 年半时间）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。但本博文只介绍最重要的十大新特性

特性列表

- 平台级 modularity（原名：Jigsaw） 模块化系统
- Java 的 REPL 工具： jShell 命令
- 多版本兼容 jar 包（这个在处理向下兼容方面，非常好用）
- 语法改进：接口的私有方法
- 语法改进：UnderScore(下划线)使用的限制
- 底层结构：String 存储结构变更（这个很重要）

- 集合工厂方法：快速创建只读集合
- 增强的 Stream API
- 全新的 HTTP 客户端 API
- 其它特性
- 它的新特性来自于 100 于项 JEP 和 40 于项 JSR

Java 10

2018 年 3 月 20 日，Java 10 正式发布，这一次没有跳票。它号称有 109 项新特性，包含 12 个 JEP。需要注意的是，本次 Java10 并不是 Oracle 的官方 LTS 版本，还是坐等 java11 的发布再考虑在生产中使用

特性列表

- 局部变量的类型推断 var 关键字
- GC 改进和内存管理 并行全垃圾回收器 G1
- 垃圾回收器接口
- 线程-局部变量管控
- 合并 JDK 多个代码仓库到一个单独的储存库中
- 新增 API：ByteArrayOutputStream
- 新增 API：List、Map、Set
- 新增 API：java.util.Properties
- 新增 API： Collectors 收集器

- 其它特性

1、局部变量的类型推断 var 关键字

这个新功能将为 Java 增加一些语法糖 - 简化它并改善开发者体验。新的语法将减少与编写 Java 相关的冗长度，同时保持对静态类型安全性的承诺。

这可能是 Java10 给开发者带来的最大的一个新特性。下面主要看例子：

```
public static void main(String[] args) {
 var list = new ArrayList<String>();
 list.add("hello，world! ");
 System.out.println(list);
 }
```

这是最平常的使用。注意赋值语句右边 ,最好写上泛型类型 ,否则会有如下情况：

```
public static void main(String[] args) {
 var list = new ArrayList<>();
 list.add("hello，world! ");
 list.add(1);
 list.add(1.01);
 System.out.println(list);
 }
```

list 什么都可以装，非常的不安全了。和 js 等语言不同的是，毕竟 Java 还是强类型的语言，所以下面语句是编译报错的：

```
public static void main(String[] args) {
 var list = new ArrayList<String>();
 list.add("hello，world! ");
 System.out.println(list);
```

```
list = new ArrayList<Integer>(); //编译报错
}
```

注意：下面几点使用限制

局部变量初始化

for 循环内部索引变量

传统的 for 循环声明变量

```
public static void main(String[] args) {
//局部变量初始化
var list = new ArrayList<String>();
//for 循环内部索引变量
for (var s : list) {
System.out.println(s);
}
//传统的 for 循环声明变量
for (var i = 0; i < list.size(); i++) {
System.out.println(i);
}
}
```

下面这几种情况，都是不能使用 var 的

```
方法参数全局变量
```

public static var list = new ArrayList<String>(); //编译报错

public static List<String> list = new ArrayList<>(); //正常编译通过

构造函数参数方法返回类型字段捕获表达式（或任何其他类型的变量声明）

2、GC 改进和内存管理 并行全垃圾回收器 G1

JDK 10 中有 2 个 JEP 专门用于改进当前的垃圾收集元素。

Java 10 的第二个 JEP 是针对 G1 的并行完全 GC（JEP 307），其重点在于通过完全 GC 并行来改善 G1 最坏情况的等待时间。G1 是 Java 9 中的默认 GC，并且此 JEP 的目标是使 G1 平行。

3、垃圾回收器接口

这不是让开发者用来控制垃圾回收的接口 ;而是一个在 JVM 源代码中的允许另外的垃圾回收器快速方便的集成的接口。

4、线程-局部变量管控

这是在 JVM 内部相当低级别的更改 ,现在将允许在不运行全局虚拟机安全点的情况下实现线程回调。这将使得停止单个线程变得可能和便宜，而不是只能启用或停止所有线程。

5、合并 JDK 多个代码仓库到一个单独的储存库中

在 JDK9 中,有 8 个仓库：root、corba、hotspot、jaxp、jaxws、jdk、langtools 和 nashorn 。在 JDK10 中这些将被合并为一个，使得跨相互依赖的变更集的存储库运行 atomic commit （原子提交）成为可能。

6、新增 API：ByteArrayOutputStream

String toString(Charset): 重载 toString()，通过使用指定的字符集解码字节，将缓冲区的内容转换为字符串。

7、新增 API：List、Map、Set

这 3 个接口都增加了一个新的静态方法，copyOf(Collection）。这些函数按照其迭代顺序返回一个不可修改的列表、映射或包含给定集合的元素的集合。

8、新增 API：java.util.Properties

增加了一个新的构造函数，它接受一个 int 参数。这将创建一个没有默认值的空属性列表，并且指定初始大小以容纳指定的元素数量，而无需动态调整大小。还有一个新的重载的 replace 方法，接受三个 Object 参数并返回一个布尔值。只有在当前映射到指定值时，才会替换指定键的条目。

9、新增 API： Collectors 收集器

toUnmodifiableList():

toUnmodifiableSet():

toUnmodifiableMap(Function, Function):

toUnmodifiableMap(Function, Function, BinaryOperator):

这四个新方法都返回 Collectors ，将输入元素聚集到适当的不可修改的集合中。

10、其它特性

线程本地握手（JEP 312）

其他 Unicode 语言 - 标记扩展（JEP 314）

基于 Java 的实验性 JIT 编译器

根证书颁发认证（CA）

删除工具 javah（JEP 313）

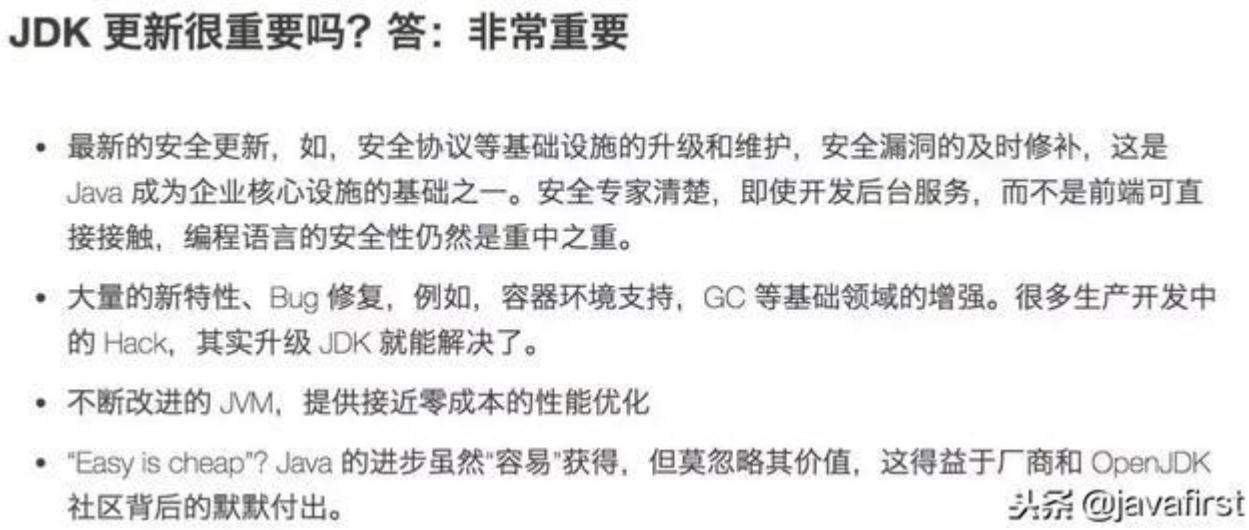
从 JDK 中移除了 javah 工具，这个很简单并且很重要。

最后

JDK10 的升级幅度其实主要还是以优化为主，并没有带来太多对使用者惊喜的特性。所以建议广大开发者还是坐等 Java11 吧 ,预计 2018 年 9 月份就会到来，最重要的是它是 LTS 版本哦，所以是可以运用在生产上的。

Java 11

2018 年 9 月 26 日，Oracle 官方宣布 Java 11 正式发布。这是 Java 大版本周期变化后的第一个长期支持版本（LTS 版本，Long-Term-Support，持续支持到 2026 年 9 月），非常值得关注。Java11 带来了 ZGC、Http Client 等重要特性，一共包含 17 个 JEP(JDK Enhancement Proposals ,JDK 增强提案)。



特性列表

官方新特性：

本文针对于读者对关心、也是最实用的八大新特性做出一些讲解

- 本地变量类型推断
- 字符串加强
- 集合加强
- Stream 加强
- Optional 加强
- InputStream 加强

- HTTP Client API
- 化繁为简，一个命令编译运行源代码

1、本地变量类型推断

什么是局部变量类型推断?

```
var javastack = "javastack";
System.out.println(javastack);
```

大家看出来了，局部变量类型推断就是左边的类型直接使用 var 定义，而不用写具体的类型，编译器能根据右边的表达式自动推断类型，如上面的 String 。

```
var javastack = "javastack";
就等于:
String javastack = "javastack";
```

头条 @javafirst

2、字符串加强

Java 11 增加了一系列的字符串处理方法，如以下所示。

```
// 判断字符串是否为空白
" ".isBlank(); // true
// 去除首尾空格
" Javastack ".strip(); // "Javastack"
// 去除尾部空格
" Javastack ".stripTrailing(); // " Javastack"
// 去除首部空格
" Javastack ".stripLeading(); // "Javastack "
// 复制字符串
"Java".repeat(3);// "JavaJavaJava"
// 行数统计
"A\nB\nC".lines().count(); // 3
```

头条 @javafirst

3、集合加强

自 Java 9 开始，Jdk 里面为集合（List/ Set/ Map）都添加了 of 和 copyOf 方法，它们两个都用来创建不可变的集合，来看下它们的使用和区别。

示例 1：

```
var list = List.of("Java", "Python", "C");
var copy = List.copyOf(list);
System.out.println(list == copy); // true
```

示例 2：

```
var list = new ArrayList<String>();
var copy = List.copyOf(list);
System.out.println(list == copy); // false
```

示例 1 和 2 代码差不多，为什么一个为 true,一个为 false?

来看下它们的源码：

```
static <E> List<E> of(E... elements) {
 switch (elements.length) { // implicit null check of elements
 case 0:
 return ImmutableCollections.emptyList();
 case 1:
 return new ImmutableCollections.List12<>(elements[0]);
 case 2:
```

```
 return new ImmutableCollections.List12<>(elements[0],
elements[1]);
 default:
 return new ImmutableCollections.ListN<>(elements);
 }
}
static <E> List<E> copyOf(Collection<? extends E> coll) {
 return ImmutableCollections.listCopy(coll);
}
static <E> List<E> listCopy(Collection<? extends E> coll) {
 if (coll instanceof AbstractImmutableList && coll.getClass() !=
SubList.class) {
 return (List<E>)coll;
 } else {
 return (List<E>)List.of(coll.toArray());
 }
}
```

可以看出 copyOf 方法会先判断来源集合是不是 AbstractImmutableList 类型的，如果是，就直接返回，如果不是，则调用 of 创建一个新的集合。

示例 2 因为用的 new 创建的集合，不属于不可变 AbstractImmutableList 类的子类，所以 copyOf 方法又创建了一个新的实例，所以为 false.

注意：使用 of 和 copyOf 创建的集合为不可变集合，不能进行添加、删除、替换、排序等操作，不然会报 java.lang.UnsupportedOperationException 异常。

上面演示了 List 的 of 和 copyOf 方法，Set 和 Map 接口都有。

```
public static void main(String[] args) {
 Set<String> names = Set.of("Fred", "Wilma", "Barney", "Betty");
 //JDK11 之前我们只能这么写
 System.out.println(Arrays.toString(names.toArray(new
String[names.size()])));
 //JDK11 之后 可以直接这么写了
 System.out.println(Arrays.toString(names.toArray(size -> new
String[size])));

System.out.println(Arrays.toString(names.toArray(String[]::new
)));
 }
```

Collection.toArray(IntFunction)

在java.util.Collection接口中添加了一个新的默认方法toArray(IntFunction)。

此方法允许将集合的元素传输到新创建的所需运行时类型的数组。

```
public static void main(String[] args) {
 Set<String> names = Set.of("Fred", "Wilma", "Barney", "Betty");
 //JDK11 之前我们只能这么写
 System.out.println(Arrays.toString(names.toArray(new
String[names.size()])));
 //JDK11 之后 可以直接这么写了
 System.out.println(Arrays.toString(names.toArray(size -> new
String[size])));

System.out.println(Arrays.toString(names.toArray(String[]::new
)));
 }
```

4、Stream 加强

1、ofNullable()方法

Stream 是 Java 8 中的新特性，Java 9 开始对 Stream 增加了以下 4 个新方法。

增加单个参数构造方法，可为 null

```
Stream.ofNullable(null).count(); // 0
//JDK8 木有 ofNullable 方法哦
```

源码可看看：

```
/**
 * @since 9
 */
 public static<T> Stream<T> ofNullable(T t) {
 return t == null ? Stream.empty()
 : StreamSupport.stream(new Streams.StreamBuilderImpl<>(t),
false);
 }
```

2、增加 takeWhile 和 dropWhile 方法

```
Stream.of(1, 2, 3, 2, 1)
.takeWhile(n -> n < 3)
.collect(Collectors.toList()); // [1, 2]
```

takeWhile 表示从开始计算，当 n < 3 时就截止。

```
Stream.of(1, 2, 3, 2, 1)
.dropWhile(n -> n < 3)
.collect(Collectors.toList()); // [3, 2, 1]
```

3、iterate 重载

这个 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。

```
public static void main(String[] args) {
// 这构造的是无限流 JDK8 开始
Stream.iterate(0, (x) -> x + 1);
// 这构造的是小于 10 就结束的流 JDK9 开始
Stream.iterate(0, x -> x < 10, x -> x + 1);
}
```

5、Optional 加强

Opthonal 也增加了几个非常酷的方法，现在可以很方便的将一个 Optional 转换成一个 Stream, 或者当一个空 Optional 时给它一个替代的。

```
Optional.of("javastack").orElseThrow(); // javastack
Optional.of("javastack").stream().count(); // 1
Optional.ofNullable(null)
.or(() -> Optional.of("javastack"))
.get(); // javastac
```

or 方法和 stream 方法显然都是新增的

6、InputStream 加强

InputStream 终于有了一个非常有用的方法：transferTo，可以用来将数据直接传输到 OutputStream，这是在处理原始数据流时非常常见的一种用法，如下示例。

```
var classLoader = ClassLoader.getSystemClassLoader();
var inputStream =
classLoader.getResourceAsStream("javastack.txt");
var javastack = File.createTempFile("javastack2", "txt");
try (var outputStream = new FileOutputStream(javastack)) {
 inputStream.transferTo(outputStream);
}
```

7、HTTP Client API(重磅)

在 java9 及 10 被标记 incubator 的模块 jdk.incubator.httpclient，在 java11 被标记为正式，改为 java.net.http 模块。这是 Java 9 开始引入的一个处理 HTTP 请求的的孵化 HTTP Client API ,该 API 支持同步和异步 ,而在 Java 11 中已经为正式可用状态，你可以在 java.net 包中找到这个 API。

来看一下 HTTP Client 的用法：

上面的 .GET() 可以省略，默认请求方式为 Get！

更多使用示例可以看这个 API，后续有机会再做演示。

现在 Java 自带了这个 HTTP Client API，我们以后还有必要用 Apache 的 HttpClient 工具包吗？我觉得没啥必要了

8、化繁为简，一个命令编译运行源代码

看下面的代码。

```
// 编译
javac Javastack.java
```

```
// 运行
java Javastack
```

在我们的认知里面，要运行一个 Java 源代码必须先编译，再运行，两步执行动作。而在未来的 Java 11 版本中，通过一个 java 命令就直接搞定了，如以下所示。

```
java Javastack.java
```

9、移除项

移除了 com.sun.awt.AWTUtilities

移除了 sun.misc.Unsafe.defineClass，

使用 java.lang.invoke.MethodHandles.Lookup.defineClass 来替代

移除了 Thread.destroy()以及 Thread.stop(Throwable)方法移除了

sun.nio.ch.disableSystemWideOverlappingFileLockCheck、sun.locale.formatasdefault 属性

移除了 jdk.snmp 模块

移除了 javafx，openjdk 估计是从 java10 版本就移除了，oracle jdk10 还尚未移除 javafx，而 java11 版本则 oracle 的 jdk 版本也移除了 javafx

移除了 Java Mission Control，从 JDK 中移除之后，需要自己单独下载

移除了这些 Root Certificates ：Baltimore Cybertrust Code Signing CA，SECOM ，AOL and Swisscom

10、废弃项

废弃了 Nashorn JavaScript Engine

废弃了-XX+AggressiveOpts 选项

-XX:+UnlockCommercialFeatures 以及

-XX:+LogCommercialFeatures 选项也不- 再需要

废弃了 Pack200 工具及其 API

说到最后

java11 是 java 改为 6 个月发布一版的策略之后的第一个 LTS(Long-Term Support)版本(oracle 版本才有 LTS)，这个版本最主要的特性是：在模块方面移除 Java EE 以及 CORBA 模块，在 JVM 方面引入了实验性的 ZGC，在 API 方面正式提供了 HttpClient 类。

从 java11 版本开始，不再单独发布 JRE 或者 Server JRE 版本了，有需要的可以自己通过 jlink 去定制 runtime image

备注：ZGC 作为实验性功能包含在内。要启用它，因此需要将-XX：+ UnlockExperimentalVMOptions 选项与-XX：+ UseZGC 选项结合使用。

ZGC 的这个实验版具有以下限制：

- 它仅适用于 Linux / x64。
- 不支持使用压缩的 oops 和/或压缩的类点。默认情况下禁用-XX：+UseCompressedOops 和-XX：+UseCompressedClassPointers 选项。启用它们将不起作用。
- 不支持类卸载。默认情况下禁用-XX：+ ClassUnloading 和-XX：+ - - ClassUnloadingWithConcurrentMark 选项。启用它们将不起作用。
- 不支持将 ZGC 与 Graal 结合使用。

```
older2sub2
 path1.relativize(path2); //....
```

```
older2sub2
 path1.subpath(0, 1); //folder1
 path1.startsWith(path2); //false
 path1.endsWith(path2); //false
 Paths.get("folder1/./../folder2/my.text").normalize();
//folder2my.text
 }
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
 PathUsage usage = new PathUsage();
 usage.usePath();
 }
}
```

DirectoryStream

```
public class ListFile {
 public void listFiles() throws IOException {
 Path path = Paths.get("");
 try (DirectoryStream<Path> stream =
Files.newDirectoryStream(path, "*.*")) {
 for (Path entry: stream) {
 //使用 entry
 System.out.println(entry);
 }
 }
 }
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) throws IOException {
 ListFile listFile = new ListFile();
 listFile.listFiles();
 }
}
```

Files

```
public class FilesUtils {
 public void manipulateFiles() throws IOException {
 Path newFile =
Files.createFile(Paths.get("new.txt").toAbsolutePath());
 List<String> content = new ArrayList<String>();
 content.add("Hello");
 content.add("World");
 Files.write(newFile, content, Charset.forName("UTF-8"));
 Files.size(newFile);
 byte[] bytes = Files.readAllBytes(newFile);
 ByteArrayOutputStream output = new ByteArrayOutputStream();
 Files.copy(newFile, output);
 Files.delete(newFile);
 }
 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) throws IOException {
 FilesUtils fu = new FilesUtils();
 fu.manipulateFiles();
 }
}
```

WatchService

```
public class WatchAndCalculate {
 public void calculate() throws IOException, InterruptedException
{
 WatchService service =
FileSystems.getDefault().newWatchService();
 Path path = Paths.get("").toAbsolutePath();
```

```
path.register(service, StandardWatchEventKinds.ENTRY_CREATE);
while (true) {
WatchKey key = service.take();
for (WatchEvent<?> event : key.pollEvents()) {
Path createdPath = (Path) event.context();
createdPath = path.resolve(createdPath);
long size = Files.size(createdPath);
System.out.println(createdPath + " ==> " + size);
}
key.reset();
}
}
/**
* @param args the command line arguments
*/
public static void main(String[] args) throws Throwable {
WatchAndCalculate wc = new WatchAndCalculate();
wc.calculate();
}
}
```

8、fork/join 计算框架

Java7 提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

该框架为 Java8 的并行流打下了坚实的基础