链滴

Java 的新特性

作者：xunxiake

原文链接：https://ld246.com/article/1476153851755

来源网站：链滴

Java5:
</san>1、泛型 Generics:
引用泛型之后，允指定集合里元素的类型，免去了强制类型转换，并<spa>且能在编译时刻进行类型检查的好处。Parameterized Type作为参数和返值，Generic是vararg、annotation、enumeration、collection的基石。</span
A、类型安全
抛弃List、Map，使用List<T>、Map<K,V>给它们添加元素或者使用Iterato<T>遍历时，编译期就可以给你检查出类型错误
B、方法参数和返回值加上了Type
抛弃List、Map，使用List<T>、Map<K,V>
C、不需要类型转换
List<String> list=new ArrayList<String>();
String str=list.get(i);
D、类型通配符“?”
假设一个打印List<T>中元素的方法printList,我们希望任何类型T的List<T>可以被打印：
代码：
public void printList(List<?> list,PrintStream out)trows IOException{
for(Iterator<?> i=list.iterator();i.hasNext();){
System.out.println(i.next.toString());
}
}
如果通配符？让我们的参数类型过于广泛，我们可以把List<?>、Iterator<?&gt 修改为
List<? Extends Number>、Iterator<? Extends Nmber>限制一下它。
2、枚举类型 Enumeration:
3、自动装箱拆箱（自动类型包装和解包）autoboxing &amp unboxing:
 简单的说是类型自动转换。
自动装包：基本类型自动转为包装类（int ——Integer）
自动拆包：包装类自动转为基本类型（Integer——int）
4、可变参数varargs(varargs number of arguents)
参数类型相同时，把重载函数合并到一起了。
如：public void test(object... objs){
for(Object obj:objs){
System.out.println(obj);
}
}
5、Annotations 它是Java中的metadata
A、Tiger中预定义的三种标准annotation
a 、Override
指出某个method覆盖了superclass 的method当你要覆盖的方法名拼写错时编不通过
b、Deprecated
指出某个method或element类型的使用是被阻止的，子类将不能覆盖该方法
c、SupressWarnings
关闭class、method、field、variable 初始化的编译期警告，比如：List没有使

 Generic,则@SuppressWarnings("unchecked")去掉编译期警告。
B、自定义annotation
public @interface Marked{}
C、meta-annotation
或者说annotation的annotation
四种标准的meta-annotation全部定义在java.lang.annotaion包中:<b>a, Target
指定所定义的annotation可以用在哪些程序单元上
如果Target没有指定，则表示该annotation可以使用在任意程序单元上
代码br>   1. @Target({ElementType.ANNOTATION_TYP,
   2.   ElementType.CONSTRUCTOR, nbsp;
   3.   &nbp;   ElementType.FIELD,
   4.   &nsp;   ElementType.LOCAL_VARIABLE,
   5.   &nbp; ElementType.METHOD,
nbsp; 6.   ElementType.PACKAGE,
  &bsp;7.   ElementType.PRAMETER,
   8.   ElementType.TYPE}) nbsp;
   9. public @inteface TODO {}

span>b, Retention
指出Java编译期如何对待annotation
annotation可以被编译期丢掉，或者保留在编译过的class文件中
在annotation被保留时，它也指定是否会在JVM加载lass时读取该annotation
代码
   1. @Retention(RetentionPolicy.SOURCE) / Annotation会被编译期丢弃
    2. public @interface TODO1 {} </span
   3. @Retention(RetentionPolicy.CLAS)   // Annotation保留在class文件中，但会被JVM忽略 /span>
   4. public @interface nbsp;TODO2 {}
  &nbp;5. @Retention(RetentionPolicy.RUNTIME) // Annotation保留在class文件且会被JVM读取
   6.&bsp;public @interface TODO3 {}
</spn>

c, Documentedbr>指出被定义的annotation被视为所熟悉的程序单元的公开API之一<spn>
被@Documented标注的annotation会在javadoc中显示，这在annotatio对它标注的元素被客户端使用时有影响时起作用
d, Inerited
该meta-annotation应用于目标为class的annotatio类型上，被此annotattion标注的class会自动继承父类的annotation
</span
D, Annotation的反射
<spa>我们发现java.lang.Class有许多与Annotation的反射相关的方法，如getAnnotations、isAnnotatinpresent
我们可以利用Annotation反射来做许多事情，比自定义Annotation来做Model对象验证
代码<spa>
   1. @Retention(RetentionPolicy.RUNTIME
   2. @Target(ElementType.FIELD, ElementType.METHOD }) <spa>
   3. public @interface RejectEpty {
   4.

    /** hint title used in error
message */
   5
     String value() default "";

   6. }
span>
   7.   <b
>   8. @Retention(RetentionPolicy.RUNTIME)&nbs
;
   9. @Target(
 ElementType.FIELD, ElementType.METHOD }) <spa
>
  10. public @interface AcceptInt&nb
p;{
  11.
  int min() default Integer.MIN_VALUE; </span

  12.   int
max() default Integer.MAX_VALUE;
span> 13.   String hint() default
nbsp;"";
  14. }

使用@RejectEmpty和@AcceptInt标注我们的Mod
l的field，然后利用反射来做Model验证
6、新的迭代语句（for(int n:numbers)）
7、静态导入（import static ）
8、新的格式化方法（java.util.Formatter）
formatter.format("Remaining account balance: $%.2f",
alance);
9、新的线程模型和并发库Thread Framework
HashMap的替代者<a href="https://ld246.com/forward?goto=http%3A%
F%2Fjava.sun.com%2Fj2se%2F1.5.0%2Fdocs%2Fapi%2Fjava%2Futil%2Fconcurrent%2FConcur
entHashMap.html" target="_blank" rel="nofollow ugc">ConcurrentHashMap</span
和ArrayList的替代者<a href="https://ld246.com/forward?goto=http%3
%2F%2Fjava.sun.com%2Fj2se%2F1.5.0%2Fdocs%2Fapi%2Fjava%2Futil%2Fconcurrent%2FCop
OnWriteArrayList.html" target="_blank" rel="nofollow ugc">CopyOnWriteArrayList</
pan>
在大并发量读取时采用java.util.concurrent包里的一些类
让大家满意BlockingQueue、Callable、Executor、Semaphore...

Java6:
1、引入了一个支持脚本引擎的新框架
2、UI的增强
3、对WebService支持的增强（JAX-WS2.0和JAXB2.0）</
>
4、一系列新的安全相关的增强
5、JDBC4.0
6、Compiler API
7、通用的Annotations支持

Java7:
1，switch中可以使用字串了
String s = "test";
switch (s) {
case "
est" :
System.out.println("test");
case "test1" :
System.out.p
intln("test1");
break ;
default :
System.out.println("break");
nbsp;
break ;
}

2.运用List<String> tempList = new ArrayList&lt
>(); 即泛型实例化类型自动推断

3.语法上支持集合，而不一定是数组

final Lis
<Integer> piDigits = [1,2,3,4,5,8];
4.新增一些取环境信息的工具方法

ile System.getJavaIoTempDir() // IO临时文件夹

File System.getJavaHomeDir() // JRE
安装目录

File System.getUserHomeDir() // 当前用户目录

File System.getUse
Dir() // 启动java进程时所在的目录5

5.Boolean类型反转，空指针安全,参与位运算
<b
>Boolean Booleans.negate(Boolean booleanObj)

True => False , False => True
Null => Null

boolean Booleans.and(boolean[] array)

boolean Booleans

or(boolean[] array)

boolean Booleans.xor(boolean[] array)

boolean Boolea
s.and(Boolean[] array)

boolean Booleans.or(Boolean[] array)

boolean Bool
ans.xor(Boolean[] array)

6.两个char间的equals
boolean Character.equalsI
noreCase(char ch1, char ch2)

7.安全的加减乘除
int Math.safeToInt(long v
lue)

int Math.safeNegate(int value)

long Math.safeSubtract(long value1, in
value2)

long Math.safeSubtract(long value1, long value2)

int Math.safeMu
tiply(int value1, int value2)

long Math.safeMultiply(long value1, int value2)

ong Math.safeMultiply(long value1, long value2)

long Math.safeNegate(long value)
br>
int Math.safeAdd(int value1, int value2)

long Math.safeAdd(long value1, int
value2)

long Math.safeAdd(long value1, long value2)

int Math.safeSubtrac
(int value1, int value2)

8.map集合支持并发请求，且可以写成 Map map = {name:"xxx",
ge:18};

<h2>Java 8的新特性</
></h2>

前言： <a href="https://ld246.com/forward?goto=http%3A%2
%2Flib.csdn.net%2Fbase%2F17" class="replace_word" title="Java EE知识库" target="_blank" re
="nofollow ugc">Java 8 已经发布很久了，很多报道表明Java 8 是一次重大的版本升
。在Java Code Geeks上已经有很多介绍Java 8新特性的文章，例如<a href="https://ld246.com/fo
ward?goto=http%3A%2F%2Fwww.javacodegeeks.com%2F2014%2F04%2Fplaying-with-java-8
lambdas-and-concurrency.html" target="_blank" rel="nofollow ugc">Playing with Java 8 – L
mbdas and Concurrency、<a href="https://ld246.com/forward?goto=http%3A%2F%2F
ww.javacodegeeks.com%2F2014%2F04%2Fjava-8-date-time-api-tutorial-localdatetime.html"
arget="_blank" rel="nofollow ugc">Java 8 Date Time API Tutorial : LocalDateTime和<a h
ef="https://ld246.com/forward?goto=http%3A%2F%2Fwww.javacodegeeks.com%2F2014%2F
4%2Fabstract-class-versus-interface-in-the-jdk-8-era.html" target="_blank" rel="nofollow ugc
>Abstract Class Versus Interface in the JDK 8 Era。本文还参考了一些其他资料，例如：<a h
ef="https://ld246.com/forward?goto=http%3A%2F%2Fwww.javacodegeeks.com%2F2014%2F
4%2F15-must-read-java-8-tutorials.html" target="_blank" rel="nofollow ugc">15 Must Read
ava 8 Tutorials和<a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.javac
degeeks.com%2F2014%2F04%2Fjava-8-friday-the-dark-side-of-java-8.html" target="_blank"
el="nofollow ugc">The Dark Side of Java 8。本文综合了上述资料，整理成一份关于Java 8
特性的参考教材，希望你有所收获。
<h2>1. 简介</h2>
毫无疑问，<a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.oracle.com
2Ftechnetwork%2Fjava%2Fjavase%2F8train-relnotes-latest-2153846.html" target="_blank" re
="nofollow ugc">Java 8是Java自Java 5（发布于2004年）之后的最重要的版本。这个版本包
语言、编译器、库、工具和JVM等方面的十多个新特性。在本文中我们将学习这些新特性，并用实际
例子说明在什么场景下适合使用。
这个教程包含Java开发者经常面对的几类问题：

语言
编译器
库
工具
运行时（JVM）

<h2>2. Java语言的新特性</h2>
Java 8是Java的一个重大版本，有人认为，虽然这些新特性领Java开发人员十分期待，但同时也
要花不少精力去学习。在这一小节中，我们将介绍Java 8的大部分新特性。

<h2>2.1 Lambda表达式和函数式接口</h2>
Lambda表达式（也称为闭包）是Java 8中最大和最令人期待的语言改变。它允许我们将函数当参数传递给某个方法，或者把代码本身当作数据处理：<a href="https://ld246.com/forward?goto
http%3A%2F%2Fwww.javacodegeeks.com%2F2014%2F03%2Ffunctional-programming-with-java-8-lambda-expressions-monads.html" target="_blank" rel="nofollow ugc">函数式开发者</
>非常熟悉这些概念。很多JVM平台上的语言（Groovy、<a href="https://ld246.com/forward?got
=http%3A%2F%2Fwww.javacodegeeks.com%2Ftutorials%2Fscala-tutorials%2F" target="_blan
" rel="nofollow ugc">Scala等）从诞生之日就支持Lambda表达式，但是Java开发者没有选择只能使用匿名内部类代替Lambda表达式。
Lambda的设计耗费了很多时间和很大的社区力量，最终找到一种折中的实现方案，可以实现简而紧凑的语言结构。最简单的Lambda表达式可由逗号分隔的参数列表、->
号和语句块组成，例如：
<pre class="hljs cs"><code>Arrays.asList("a", <span clas
="hljs-string">"b", "d").forEach(e -> System.
span class="hljs-keyword">out.println(e));</code></pre>
在上面这个代码中的参数e的类型是由编译器推理得出的，你也可以显式指该参数的类型，例如：
<pre class="hljs coffeescript"><code>Arrays.asList("a", <
pan class="hljs-string">"b", "d").forEach(<span c
ass="hljs-function">(String e) -> System.out
println(e));</code></pre>
如果Lambda表达式需要更复杂的语句块，则可以使用花括号将该语句块括起来，类似于Java中函数体，例如：
<pre class="hljs php"><code>Arrays.asList("a", <span cla
s="hljs-string">"b", "d").<span class="hljs-keywo
d">forEach(e -> {
    System.out.print(e);
    System.out.print(e);
});</code></pre>
Lambda表达式可以引用类成员和局部变量（会将这些变量隐式得转换成final</stron
>的），例如下列两个代码块的效果完全相同：
<pre class="hljs coffeescript"><code>String separator = ","</spa
>;
Arrays.asList("a", "b",
span class="hljs-string">"d").forEach(
    (String e) ->
System.out.print(e + separator));</code></pre>
和
<pre class="hljs coffeescript"><code>final String separator = ","</
pan>;
Arrays.asList("a", "b",
span class="hljs-string">"d").forEach(
    (String e) ->
System.out.print(e + separator));</code></pre>
Lambda表达式有返回值，返回值的类型也由编译器推理得出。如果Lambda表达式中的语句块有一行，则可以不用使用return语句，下列两个代码片段效果相同：
<pre class="hljs coffeescript"><code>Arrays.asList("a", <
pan class="hljs-string">"b", "d").sort(<span clas
="hljs-function">(e1, e2) -> e1.compareTo(
e2));</code></pre>
和
<pre class="hljs coffeescript"><code>Arrays.asList("a", <
pan class="hljs-string">"b", "d").sort(<span clas
="hljs-function">(e1, e2) -> {

```
    int result = e1.compareTo( e2 );
    return result;
} );
```

Lambda的设计者们为了让现有的功能与Lambda表达式良好兼容，考虑了很多方法，于是产生**函数接口**这个概念。函数接口指的是只有个函数的接口，这样的接口可以隐式转换为Lambda表达式。**java.lang.Runnable**和**java.util.concurrent.Callable**是函数式接口的最佳例子。在实践中，函数接口非常脆弱：只要某个开发者在该接口中添加一个函数，则该接口就不再是函数式接口进而导致编失败。为了克服这种代码层面的脆弱性，并显式说明某个接口是函数式接口，Java 8 提供了一个特殊注解**@FunctionalInterface**（Java 库中的所有相关接口都已经带有这个注解了，举个简单的函数式接口的定义：

```
@FunctionalInterface
public interface Functional {
    void method();
}
```

不过有一点需要注意，默认方法和静态方法不会破坏函数式接口的定义，因此如下代码是合法的。

```
@FunctionalInterface
public interface FunctionalDefaultMethods {
    void method();
```

```
}
```

Lambda表达式作为Java 8的最大卖点，它有潜力吸引更多的开发者加入到JVM平台，并在纯Jav编程中使用函数式编程的概念。如果你需要了解更多Lambda表达式的细节，可以参考官方文档。

2.2 接口的默认方法和静态方法

Java 8使用两个新概念扩展了接口的含义：默认方法和静态方法。默认方法使得接口有点类似traits，不过要实现的目标不一。默认方法使得开发者可以在 不破坏二进制兼容性的前提下，往现存接口中添加新的方法，即不强制些实现了该接口的类也同时实现这个新加的方法。

默认方法和抽象方法之间的区别在于抽象方法需要实现，而默认方法不需要。接口提供的默认方会被接口的实现类继承或者覆写，例子代码如下：

```
private interface Defaulable {
    // Interfaces now allow default methods, the implementer may or 
    // may not implement (override) them.
```

```
    <span class="hljs-function"><span class="hljs-keyword">default</span> String <span cla
s="hljs-title">notRequired</span><span class="hljs-params">()</span> </span>{
        <span class="hljs-keyword">return</span> <span class="hljs-string">"Default implem
ntation"</span>;
    }
}
<p><span class="hljs-keyword">private</span> <span class="hljs-keyword">static</span>
<span class="hljs-class"><span class="hljs-keyword">class</span> <span class="hljs-title"
DefaultableImpl</span> <span class="hljs-keyword">implements</span> <span class="hljs
title">Defaulable</span> </span>{<br>
}</p>
</code><p><code><span class="hljs-keyword">private</span> <span class="hljs-keyword
>static</span> <span class="hljs-class"><span class="hljs-keyword">class</span> <span cl
ss="hljs-title">OverridableImpl</span> <span class="hljs-keyword">implements</span> <s
an class="hljs-title">Defaulable</span> </span>{<br>
<span class="hljs-annotation">@Override</span><br>
<span class="hljs-function"><span class="hljs-keyword">public</span> String <span class=
hljs-title">notRequired</span><span class="hljs-params">()</span> </span>{<br>
<span class="hljs-keyword">return</span> <span class="hljs-string">"Overridden impleme
tation"</span>;<br>
}<br>
}</code></p></pre><p></p>
<p><strong>Defaulable</strong>接口使用关键字<strong>default</strong>定义了一个默认
法<strong>notRequired()</strong>。<strong>DefaultableImpl</strong>类实现了这个接口
同时默认继承了这个接口中的默认方法；<strong>OverridableImpl</strong>类也实现了这个接口
但覆写了该接口的默认方法，并提供了一个不同的实现。</p>
<p>Java 8带来的另一个有趣的特性是在接口中可以定义静态方法，例子代码如下：</p>
<pre class="hljs cs"><code><span class="hljs-keyword">private</span> <span class="hljs-
eyword">interface</span> <span class="hljs-title">DefaulableFactory</span> {
    <span class="hljs-comment">// Interfaces now allow static methods</span>
    <span class="hljs-function"><span class="hljs-keyword">static</span> Defaulable <span
lass="hljs-title">create</span>(<span class="hljs-params"> Supplier&amp;lt; Defaulable &amp;gt; su
plier </span>) </span>{
        <span class="hljs-keyword">return</span> supplier.<span class="hljs-keyword">get</
pan>();
    }
}</code></pre>
<p>下面的代码片段整合了默认方法和静态方法的使用场景：</p>
<pre class="hljs cpp"><code><span class="hljs-function"><span class="hljs-keyword">publ
c</span> <span class="hljs-keyword">static</span> <span class="hljs-keyword">void</sp
n> <span class="hljs-title">main</span><span class="hljs-params">( String[] args )</span>
</span>{
    Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::<span class="hljs-keyw
rd">new</span> );
    System.out.println( defaulable.notRequired() );
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">defaulable = DefaulableFactory.create( OverridableImpl::&amp;lt;span class="hljs-keyword"&amp;gt
new&amp;lt;/span&amp;gt; );
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println
 defaulable.notRequired() );
</span></span></code></pre>
</code><p><code>}</code></p></pre><p></p>
<p>这段代码的输出结果如下：</p>
<pre class="hljs php"><code><span class="hljs-keyword">Default</span> implementation
```

Overridden implementation</code></pre>
由于JVM上的默认方法的实现在字节码层面提供了支持，因此效率非常高。默认方法允许在不打现有继承体系的基础上改进接口。该特性在官方库中的应用是：给java.util.Collection</strng>接口添加新方法，如stream()、parallelStream()、<srong>forEach()和removeIf()等等。
尽管默认方法有这么多好处，但在实际开发中应该谨慎使用：在复杂的继承体系中，默认方法可引起歧义和编译错误。如果你想了解更多细节，可以参考官方文档。
<h2>2.3 方法引用</h2>
方法引用使得开发者可以直接引用现存的方法、Java类的构造方法或者实例对象。方法引用和Lambda表达式配合使用，使得java类的构造方法看起来紧凑而简洁，没有很多复杂的模板代码。
西门的例子中，Car类是不同方法引用的例子，可以帮助读者区分四种类型方法引用。
<pre class="hljs cs"><code>public static class Car {
    public static Car create(final Supplier< Car > supplier) {
        return supplier.get</pan>();
    }
<pre><code class="highlight-chroma">publiclt;span class="hljs-keyword">static void&lt/span> collide(<span class="hljs-params"&gt final Car car) {
    System.out.println("Collided "lt;/span> + car.toString());
}

public void follow(final Car another) {
    System.out.println("Following he " + another.toString());
}

public void repair()

    System.out.println("Repaired + this.toString());
}
</code></pre>
</code> <code>}</code> </pre>
第一种方法引用的类型是构造器引用，语法是Class::new，或者更一般的式：Class<T>::new。注意：这个构造器没有参数。

<pre class="hljs php"><code>final Car car = Car.create
 Car::new);
final List< Car
gt; cars = Arrays.asList(car);</code></pre>
第二种方法引用的类型是静态方法引用，语法是Class::static_method。注
：这个方法接受一个Car类型的参数。
<pre class="hljs css"><code>cars.f
rEach(Car:<span clas
="hljs-value">:collide);</code></pre>
第三种方法引用的类型是某个类的成员方法的引用，语法是Class::method</strong
，注意，这个方法没有定义入参：
<pre class="hljs css"><code>cars.f
rEach(Car:<span clas
="hljs-value">:repair);</code></pre>
第四种方法引用的类型是某个实例对象的成员方法的引用，语法是instance::method<
strong>。注意：这个方法接受一个Car类型的参数：
<pre class="hljs php"><code>final Car police = Car.cre
te(Car::new);
cars.forEach(police::follow);</code></pre>
运行上述例子，可以在控制台看到如下输出（Car实例可能不同）：
<pre class="hljs perl"><code>Collided com.javacodegeeks.java8.method.references.Method
eferences$Car@7a81197d<
span>
Repaired com.javacodegeeks.java8.method.references.MethodReferences<span class="hljs-va
iable">$Car@7a81197d
Following the com.javacodegeeks.java8.method.references.MethodReferences<span class="hl
s-variable">$Car@7a81197d</code></pre>
如果想了解和学习更详细的内容，可以参考<a href="https://ld246.com/forward?goto=http
3A%2F%2Fdocs.oracle.com%2Fjavase%2Ftutorial%2Fjava%2FjavaOO%2Fmethodreferences.h
ml" target="_blank" rel="nofollow ugc">官方文档
<h2>2.4 重复注解</h2>
自从Java 5中引入<a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.javaco
egeeks.com%2F2012%2F08%2Fjava-annotations-explored-explained.html" target="_blank" re
="nofollow ugc">注解以来，这个特性开始变得非常流行，并在各个框架和项目中被广泛使用
不过，注解有一个很大的限制是：在同一个地方不能多次使用同一个注解。Java 8打破了这个限制，
入了重复注解的概念，允许在同一个地方多次使用同一个注解。
在Java 8中使用@Repeatable注解定义重复注解，实际上，这并不是语言
面的改进，而是编译器做的一个trick，底层的技术仍然相同。可以利用下面的代码说明：
<pre class="hljs java"><code>package com.javacodeg
eks.java8.repeatable.annotations;
import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
public <span class="hljs-
eyword">class RepeatingAnnotations {<b
>
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface</spa
> Filters {
Filter[] value();
}

<pre><code class="highlight-chroma"><span class="highlight
cl">@Target(ElementType.TYPE)
<span class="hl
s-annotation">@Retention(RetentionPolicy.RUNTIME)
<span class="hl
s-annotation">@Repeatable(Filters.class)
<span class="hl
s-keyword">public @interface</spa
> Filter {
    <span class=
hljs-function">String value<span class="hljs
params">();
};

<span class="hl
s-annotation">@Filter("filter1"

<span class="hl
s-annotation">@Filter("filter2"

<span class="hl
s-keyword">public <span class="hljs-keywo
d">interface Filterable </span
gt;{
}

<span class="hl
s-function">public <span class="hljs-ke
yword">static void <span cl
ss="hljs-title">main(String[] args)</span
gt; {
    <span class=
hljs-keyword">for(Filter filter: Filterable.class.getAnnotationsByType(Filter.cl
ss)) {
        System.out.pr
ntln(filter.value());
    }
}
</code></pre>
</code> <code>}</code> </pre>
正如我们所见，这里的Filter类使用@Repeatable(Filters.class)注解修饰而Filters是存放Filter注解的容器，编译器尽量对开发者蔽这些细节。这样，Filterable接口可以用两个Filter注解释（这里并没有提到任何关于Filters的信息）。
另外，反射API提供了一个新的方法：getAnnotationsByType()，可以返某个类型的重复注解，例如<code>Filterable.class.getAnnoation(Filters.class)</code>将返回两个ilter实例，输出到控制台的内容如下所示：
<pre class="hljs undefined"><code>filter1
filter2</code></pre>
如果你希望了解更多内容，可以参考官方文档。
<h2>2.5 更好的类型推断</h2>
Java 8编译器在类型推断方面有很大的提升，在很多场景下编译器可以推导出某个参数的数据类

，从而使得代码更为简洁。例子代码如下：
<pre class="hljs java"><code>package com.javacodegeeks.java8.type.inference;
public class Value< T > {
public static< T > T defaultValue() {
return null;
}
<pre><code class="highlight-chroma">public TgetOrDefault(Talue, T defaultValue) {
    return (value != null</spn>) ? value : defaultValue;
}
</code></pre>
</code> <code>}</code> </pre>
下列代码是Value<String>类型的应用：
<pre class="hljs cs"><code>package com.javacodegeeks.java8.type.inference;
</code> <code>public class TypeInference {
public static void man(String[] args) {
final Value< String > value = new Value<>();
value.getOrDefault("22"</san>, Value.defaultValue());
}
}</code> </pre>
参数Value.defaultValue()的类型由编译器推导得出，不需要显式指明。在Java 7中这段代码会有编译错误，除非使用<code>Value.<String>defaultValue()</code>。</>
<h2>2.6 拓宽注解的应用场景</h2>
Java 8拓宽了注解的应用场景。现在，注解几乎可以使用在任何元素上：局部变量、接口类型、类和接口实现类，甚至可以用在函数的异常定义上。下面是一些例子：
<pre class="hljs java"><code>package com.javacodegeeks.java8.annotations;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.util.ArrayList;
import java.util.Collection;
public class Annotations {
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE_USE, ElementType.TYP_PARAMETER })
public @interface</spa

> NonEmpty {
}
<pre><code class="highlight-chroma"><span class="highlight
cl">public
tatic class</
pan> Holder< @<span class="hljs-titl
">NonEmpty T > extends @NonEmpty<
span> Object {
    <span class=
hljs-function">public <span class="hljs-
eyword">void method<span cl
ss="hljs-params">() throws
NonEmpty Exception {
    }
}

<span class="hl
s-annotation">@SuppressWarnings("unused
)
<span class="hl
s-function">public <span class="hljs-ke
word">static void <span cl
ss="hljs-title">main(String[] args)</span
gt; {
    <span class=
hljs-keyword">final Holder< String > holder = <span class=
hljs-keyword">new @NonEmpty</spn> Holder< String >();
    <span class=
hljs-annotation">@NonEmpty Collection< <span class="hljs-annot
tion">@NonEmpty String > strings = <span class="hljs-keyword"
gt;new ArrayList<>();
}
</code></pre>
</code> <code>}</code> </pre>
ElementType.TYPE_USER和ElementType.TYPE_PARAMETER
/strong>是Java 8新增的两个注解，用于描述注解的使用场景。Java 语言也做了对应的改变，以识
这些新增的注解。
<h2>3. Java编译器的新特性</h2>
<h2>3.1 参数名称</h2>
为了在运行时获得Java程序中方法的参数名称，老一辈的Java程序员必须使用不同方法，例如<a
href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2Fpaul-hammant%2F
aranamer" target="_blank" rel="nofollow ugc">Paranamer liberary。Java 8终于将这个特
规范化，在语言层面（使用反射API和Parameter.getName()方法）和字节码
面（使用新的javac编译器以及-parameters参数）提供
持。
<pre class="hljs java"><code>package com.javacodeg
eks.java8.parameter.names;
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
</code> <code>public <
pan class="hljs-keyword">class ParameterNames <
span>{

public static void main(String[] args) throws Exception {
Method method = ParameterNames.class.getMethod("main", String[].class);
for(final Parameter parameter: method.getParameters()) {
System.out.println("Parameter: " + parameter.getName());
}
}
}</code> </pre>
在Java 8中这个特性是默认关闭的，因此如果不带-parameters参数编译上述代码并运行，则会输出如下结果：
<pre class="hljs mathematica"><code>Parameter: arg</code></pre>
如果带-parameters参数，则会输出如下结果（正确的结果）：
<pre class="hljs mathematica"><code>Parameter: arg</code></pre>
如果你使用Maven进行项目管理，则可以在maven-compiler-plugin编译的配置项中配置-parameters参数：
<pre class="hljs xml"><code><plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <compilerArgument>-parameters</compilerArgument>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin></code></pre>
<h2>4. Java官方库的新特性</h2>
Java 8增加了很多新的工具类（date/time类），并扩展了现存的工具类，以支持现代的并发编、函数式编程等。
<h2>4.1 Optional</h2>
Java应用中最常见的bug就是空值异常。在Java 8之前，Google Guava引入了Optionals类来解决NullPointerException，从而避免源码各种null检查污染，以便开发者写出更加整洁的代码。Java 8也将Opt

onal加入了官方库。
Optional仅仅是一个容易：存放T类型的值或者null。它提供了一些有用的口来避免显式的null检查，可以参考Jav 8官方文档了解更多细节。
接下来看一点使用Optional的例子：可能为空的值或者某个类型的值：</
>
<pre class="hljs cs"><code>Optional< String > fullName = Optional.ofNullable(null);
System.out.println("Full Na e is set? " + fullName.isPresent());
System.out.println("Full Na e: " + fullName.orElseGet(() -> "[none]"));
System.out.println(fullName.map(s -> "Hey " + s + "!").orElse("Hey Stranger!"));</code></pre>
<p>如果Optional实例持有一个非空值，则isPresent() 法返回true，否则返回false；orElseGet()方法，Optional</strong 实例持有null，则可以接受一个lambda表达式生成的默认值；map()方法可以 现有的Opetional实例的值转换成新的值；orElse()方法与 strong>orElseGet()方法类似，但是在持有null的时候返回传入的默认值。</p>
上述代码的输出结果如下：
<pre class="hljs sql"><code>Full Name is set? false
Full Name: [<spa class="hljs-keyword">none]
Hey Stranger!</code></pre>
再看下另一个简单的例子：
<pre class="hljs swift"><code>Optional< String > firstName = Optional.of(<sp n class="hljs-string">"Tom");
System.out.println(<s an class="hljs-string">"First Name is set? " + firstName.isPresent());
System.out.println(<s an class="hljs-string">"First Name: " + firstName.orElseGet(() -> "[none]"));
System.out.println(firs Name.map(s -> "Hey "</s an> + s + "!").orElse("Hey Stra ger!"));
System.out.println();</ ode></pre>
这个例子的输出是：
<pre class="hljs sql"><code>First Name is set? true
First Name: Tom Hey Tom!</code></pre>
如果想了解更多的细节，请参考官方文档。
<h2>4.2 Streams</h2>
新增的Stream API（java.util.stream）将生成环

的函数式编程引入了Java库中。这是目前为止最大的一次对Java库的完善，以便开发者能够写出更加效、更加简洁和紧凑的代码。
Steam API极大得简化了集合操作（后面我们会看到不止是集合），首先看下这个叫Task的类：
<pre class="hljs java"><code>public class Streams /span>{
    private enum Status {
        OPEN, CLOSED
    };
<pre><code class="highlight-chroma">private <span class="hljs-keyword"&gt static final class Tas k {
    private final Status status;
    private final Integer points;

    Task(final Status status, fina Integer points) {
        this.status = status;
        this.points = points;
    }

    public Integer getPoints() </ pan>{
        return points;
    }

    public Status getStatus() </sp n>{
        return status;
    }

    @Override
    public String toString() </spa >{

        <span clas
="hljs-keyword">return String.format("[%s,
d]", status, points);
    }
}
</code></pre>
</code> <code>}</code> </pre>
Task类有一个分数（或伪复杂度）的概念，另外还有两种状态：OPEN或者CLOSED。现在假设一个task集合：
<pre class="hljs cpp"><code>final Collection< Task > tasks = Arrays.asList(
    new Task(Status.OPEN,
 ),
    new Task(Status.OPEN,
3),
    new Task(Status.CLOSED, <span class="hljs-number
>8)
);</code></pre>
首先看一个问题：在这个task集合中一共有多少个OPEN状态的点？在Java 8之前，要解决这个题，则需要使用foreach循环遍历task集合；但是在Java 8中可以利用steams决：包括一系列元素的列表，并且支持顺序和并行处理。
<pre class="hljs php"><code>// Calculate total points of all act
ve tasks using sum()
final long totalPointsOfOpenTasks = tasks
    .stream()
    .filter(task -> task.getStatus() == Status.OPEN)
    .mapToInt(Task::getPoints)
    .sum();
</code> <code>System.out.println("Total points: " +
otalPointsOfOpenTasks);</code> </pre>
运行这个方法的控制台输出是：
<pre class="hljs mathematica"><code>Total points: <
pan class="hljs-number">18</code></pre>
这里有很多知识点值得说。首先，tasks集合被转换成steam表示；其次，在steam上的
ilter操作会过滤掉所有CLOSED的task；第三，mapToInt操作基于
个task实例的Task::getPoints方法将task流转换成Integer集合；最后，通过<st
ong>sum方法计算总和，得出最后的结果。
在学习下一个例子之前，还需要记住一些steams（<a href="https://ld246.com/forward?goto
http%3A%2F%2Fdocs.oracle.com%2Fjavase%2F8%2Fdocs%2Fapi%2Fjava%2Futil%2Fstream%
Fpackage-summary.html%23StreamOps" target="_blank" rel="nofollow ugc">点此更多细节</
>）的知识点。Steam之上的操作可分为中间操作和晚期操作。
中间操作会返回一个新的steam——执行一个中间操作（例如filter）并不
执行实际的过滤操作，而是创建一个新的steam，并将原steam中符合条件的元素放入新创建的stea
。
晚期操作（例如forEach或者sum），会遍历steam
得出结果或者附带结果；在执行晚期操作之后，steam处理线已经处理完毕，就不能使用了。在几乎
有情况下，晚期操作都是立刻对steam进行遍历。
steam的另一个价值是创造性地支持并行处理（parallel processing）。对于上述的tasks集合，
们可以用下面的代码计算所有任务的点数之和：
<pre class="hljs cpp"><code>// Calculate total points of all ta
ks
final double totalPoints = tasks
  .stream()
  .parallel()
  .map(task -> task.getPoints()) <span class="hljs-c

mment">// or map(Task::getPoints)
   .reduce(0, Integer::sum);
</code> <code>System.out.println("Total points (all tasks): "<
span> + totalPoints);</code> </pre>
这里我们使用parallel方法并行处理所有的task，并使用reduce<
strong>方法计算最终的结果。控制台输出如下：
<pre class="hljs mathematica"><code>Total points（al
 tasks）: 26.0</code></pre>
对于一个集合，经常需要根据某些条件对其中的元素分组。利用steam提供的API可以很快完成
类任务，代码如下：
<pre class="hljs php"><code>// Group tasks by their status</
pan>
final Map< Status, List<
span>< Task > > map = tasks
    .stream()
    .collect(Collectors.groupingBy(Task::getStatus));
System.out.println(map);</code></pre>
控制台的输出如下：
<pre class="hljs cpp"><code>{CLOSED=[[CLOSED, 8]],
PEN=[[OPEN, 5], [OPEN, 13
/span>]]}</code></pre>
最后一个关于tasks集合的例子问题是：如何计算集合中每个任务的点数在集合中所占的比重，
体处理的代码如下：
<pre class="hljs cpp"><code>// Calculate the weight of each t
sks (as percent of total points)
final Collection< String > result = tasks
    .stream()   // Stream< String ></
pan>
    .mapToInt(Task::getPoints)   // IntStream
    .asLongStream()   // LongStream
    .mapToDouble(points -> points / totalPoints)   // Doubl
Stream
    .boxed()   // Stream< Double ><
span>
    .mapToLong(weigth -> (long)(weigth * <span cl
ss="hljs-number">100)) // LongStream
    .mapToObj(percentage -> percentage + "%")   <s
an class="hljs-comment">// Stream< String>
    .collect(Collectors.toList());   // List< String > <
span>
</code> <code>System.out.println(result);</code> </pre>
控制台输出结果如下：
<pre class="hljs json"><code>[19%, <span class="hljs-
umber">50%, 30%]</code></pre>
最后，正如之前所说，Steam API不仅可以作用于Java集合，传统的IO操作（从文件或者网络一
一行得读取数据）可以受益于steam处理，这里有一个小例子：
<pre class="hljs php"><code>final Path path = <span
lass="hljs-keyword">new File(filename).toPath();
try(Stream< String > lines = Files.lines(path, Stan
ardCharsets.UTF_8)) {
    lines.onClose(() -> System.out.println("Done!")).<sp
n class="hljs-keyword">forEach(System.out::println);
}</code></pre>

Stream的方法onClose 返回一个等价的有额外句柄的Stream，当Stream的close（）方法被调用的时候这个句柄会被执行。Stream API、Lambda表达式还有接口默认法和静态方法支持的方法引用，是Java 8对软件开发的现代范式的响应。
<h2>4.3 Date/Time API(JSR 310)</h2>
Java 8引入了新的Date-Time API(JSR 31)来改进时间、日期的处理。时间和日期的管理一直是最令Java开发者痛苦的问题。jav.util.Date和后来的java.util.Calendar一直没有解决这个问题（甚至开发者更加迷茫）。
因为上面这些原因，诞生了第三方库Joda-Time，以替代Java的时间管理API。Java 8中新的时间和日期管理API深受Joda-Time影响，并吸收了很多Jod-Time的精华。新的java.time包包含了所有关于日期、时间、时区、Instant（跟日期类似但是精确到秒）、duration（持续时间）和时钟操作的类。新设计的API认真考虑了这些类的不变性（从java.util.alendar吸取的教训），如果某个实例需要修改，则返回一个新的对象。
我们接下来看看java.time包中的关键类和各自的使用例子。首先，Clock使用时区来返回当前的纳秒时间和日期。Clock可以替代System.currntTimeMillis()和TimeZone.getDefault()。
<pre class="hljs cs"><code>// Get the system clock as UTC offet
final Clock clock = Clock.systemUTC();
System.out.println(clock.instant());
System.out.println(clock.millis());</code></pre>
这个例子的输出结果是：
<pre class="hljs cpp"><code>2014-04-12T15:<span class="hljs-number"19:29.282Z
1397315969360</code></pre>
第二，关注下LocalDate和LocalTime类。ocalDate仅仅包含ISO-8601日历系统中的日期部分；LocalTime则仅包含该日历系统中的时间部分。这两个类的对象都可以使用Clock对象构建得到。
<pre class="hljs perl"><code>// Get the local date <spn class="hljs-keyword">and local <span class=hljs-keyword">time
final LocalDate date = LocalDate.now();
final LocalDate dateFromClock = LocalDate.now(clock);
System.out.println(date);
System.out.println(dateFromClock);
// Get the local date and local <san class="hljs-keyword">time
final LocalTime time = LocalTime.now();
final LocalTime timeFromClock = LocalTime.now(clock);
</code> <code>System.out.println(time);
System.out.println(timeFromClock);</code> </pre>
上述例子的输出结果如下：
<pre class="hljs cpp"><code>2014-04-12
2014-04-12
11:25:<span class=hljs-number">54.568
15:25:<span class=hljs-number">54.568</code></pre>

LocalDateTime类包含了LocalDate和LocalTime的信息，但是不包含ISO-601日历系统中的时区信息。这里有一些关于LocalDate和LocalTime的例子：
<pre class="hljs scala"><code>// Get the local date/time</spn>
final LocalDateTime dtetime = LocalDateTime.now();
final LocalDateTime dtetimeFromClock = LocalDateTime.now(clock);
</code> <code>System.out.println(datetime);
System.out.println(datetimeFromClock);</code> </pr>
上述这个例子的输出结果如下：
<pre class="hljs cpp"><code>2014-04-12T11:37:52.309
2014-04-12T15:37:52.309</code></pre>
如果你需要特定时区的data/time信息，则可以使用ZoneDateTime，它存有ISO-8601日期系统的日期和时间，而且有时区信息。下面是一些使用不同时区的例子：
<pre class="hljs scala"><code>// Get the zoned date/time</san
final ZonedDateTime onedDatetime = ZonedDateTime.now();
final ZonedDateTime onedDatetimeFromClock = ZonedDateTime.now(clock);
final ZonedDateTime onedDatetimeFromZone = ZonedDateTime.now(ZoneId.of("America/Los_Angeles"));
</code> <code>System.out.println(zonedDatetime);<r>
System.out.println(zonedDatetimeFromClock);
System.out.println(zonedDatetimeFromZone);</code></></pre>
这个例子的输出结果是：
<pre class="hljs cpp"><code>2014-04-12T11:47:01.017-04:00[America/New_York]
2014-04-12T15:47:01.017Z
2014-04-12T08:47:01.017-07:0[America/Los_Angeles]</code></pre>
最后看下Duration类，它持有的时间精确到秒和纳秒。这使得我们可以很易得计算两个日期之间的不同，例子代码如下：
<pre class="hljs cpp"><code>// Get duration between two dats
final LocalDateTime from = LocalDateTime.of(2014, Mo

th.APRIL, 16, 0, <s
an class="hljs-number">0, 0);
final LocalDateTime to = LocalDateTime.of(2015, Month
APRIL, 16, 23, <sp
n class="hljs-number">59, 59);
</code> <code>final Duration duration = Duration.between(from, to);
System.out.println("Duration in days: " + duration.toDays()
);
System.out.println("Duration in hours: " + duration.toHou
s());</code> </pre>
这个例子用于计算2014年4月16日和2015年4月16日之间的天数和小时数，输出结果如下：
<pre class="hljs objectivec"><code>Duration in days:
span class="hljs-number">365
Duration in hours: 8783</
pan></code></pre>
对于Java 8的新日期时间的总体印象还是比较积极的，一部分是因为Joda-Time的积极影响，另
部分是因为官方终于听取了开发人员的需求。如果希望了解更多细节，可以参考<a href="https://ld2
6.com/forward?goto=http%3A%2F%2Fdocs.oracle.com%2Fjavase%2Ftutorial%2Fdatetime%2
index.html" target="_blank" rel="nofollow ugc">官方文档。
<h2>4.4 Nashorn JavaScript引擎</h2>
Java 8提供了新的<a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.javaco
egeeks.com%2F2014%2F02%2Fjava-8-compiling-lambda-expressions-in-the-new-nashorn-js
engine.html" target="_blank" rel="nofollow ugc">Nashorn JavaScript引擎，使得我们可
在JVM上开发和运行JS应用。Nashorn<a href="https://ld246.com/forward?goto=http%3A%2F
2Flib.csdn.net%2Fbase%2F18" class="replace_word" title="JavaScript知识库" target="_blank"
el="nofollow ugc">JavaScript引擎是javax.script.ScriptEngine的另一个实现版本，这类Scrip
引擎遵循相同的规则，允许Java和JavaScript交互使用，例子代码如下：
<pre class="hljs cs"><code>ScriptEngineManager manager = n
w ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript"</
pan>);
</code> <code>System.out.println(engine.getClas
().getName());
System.out.println("Result:"
/span> + engine.eval("function f() { return 1; }; f() + 1;"));
/code> </pre>
这个代码的输出结果如下：
<pre class="hljs css"><code>jdk.n
shorn.api.scripting</spa
>.NashornScriptEngine
Result:<span class="hljs-value
> 2</code></pre>
<h2>4.5 Base64</h2>
<a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.javacodegeeks.com%2
2014%2F04%2Fbase64-in-java-8-its-not-too-late-to-join-in-the-fun.html" target="_blank" rel
"nofollow ugc">对Base64编码的支持已经被加入到Java 8官方库中，这样不需要使用第三方
就可以进行Base64编码，例子代码如下：
<pre class="hljs java"><code>package com.javacodeg
eks.java8.base64;
import java.nio.charset.StandardCharsets;
import java.util.Base64;
public <span class="hljs-
eyword">class Base64s {
public <span class="hljs-k

yword">static void ma
n(String[] args) {
final String text = "Base64 fi
ally in Java 8!";
<pre><code class="highlight-chroma"><span class="highlight
cl">   final String encoded = Base64
        .getEncoder()
        .encodeToStr
ng(text.getBytes(StandardCharsets.UTF_8));
    System.out.print
n(encoded);

    <span class=
hljs-keyword">final String decoded = ne
 String(
        Base64.getD
coder().decode(encoded),
        StandardCha
sets.UTF_8);
    System.out.print
n(decoded);
}
</code></pre>
</code> <code>}</code> </pre>
这个例子的输出结果如下：
<pre class="hljs python"><code>QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
Base64 finally in J
va 8!</code></pre>
新的Base64API也支持URL和MINE的编码解码。
(Base64.getUrlEncoder
/em>() / Base64.getUrlDecoder(),&
bsp;Base64.getMimeEncoder() /Base64.<e
>getMimeDecoder())。
<h2>4.6 并行数组</h2>
Java8版本新增了很多新的方法，用于支持并行数组处理。最重要的方法是parallelSort
)，可以显著加快多核机器上的数组排序。下面的例子论证了parallexXxx</stro
g>系列的方法：
<pre class="hljs cs"><code>package com.javacodegeeks.java8.parallel.arrays;
import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;
public class
span class="hljs-title">ParallelArrays {
public <span class="hljs-k
yword">static void ma
n(String[] args) {
long[] arrayOfLong = new
/span> long [20000</spa
>];
<pre><code class="highlight-chroma"><span class="highlight
cl">   Arrays.parallelSetAll(arrayOfLong,
        index -&amp
gt; ThreadLocalRandom.current().nextInt(1000000</spa
>));
    Arrays.stream(a
rayOfLong).limit(10).forEach(

        i -> System.out.print(i + " "));
    System.out.println();

    Arrays.parallelSort(arrayOfLong);
    Arrays.stream(arrayOfLong).limit(10).forEach(
        i -> System.out.print(i + " "));
    System.out.println();
}
</code></pre>
</code> <code>}</code> </pre>
上述这些代码使用parallelSetAll()方法生成20000个随机数，然后使用<strng>parallelSort()方法进行排序。这个程序会输出乱序数组和排序数组的前10个元素。上例子的代码输出的结果是：
<pre class="hljs cpp"><code>Unsorted: 591217 891976 443951 424479 766825 351964 242997 642839 119108 552378
Sorted: 39 220 <san class="hljs-number">263 268 325 607 655 678 723</san> 793</code></pre>
<h2>4.7 并发性</h2>
基于新增的lambda表达式和steam特性，为Java 8中为java.util.concurrent.ConcurrntHashMap类添加了新的方法来支持聚焦操作；另外，也为java.util.concurrenForkJoinPool类添加了新的方法来支持通用线程池操作（更多内容可以参考我们的并发编程课程）
Java 8还添加了新的java.util.concurrent.locks.StampedLock类，用于支基于容量的锁——该锁有三个模型用于支持读写操作（可以把这个锁当做是java.util.concurent.locks.ReadWriteLock的替代者）。
在java.util.concurrent.atomic包中也新增了不少工具类，列举如下：

DoubleAccumulator
DoubleAdder
LongAccumulator
LongAdder

<h2>5. 新的Java工具</h2>
Java 8提供了一些新的命令行工具，这部分会讲解一些对开发者最有用的工具。
<h2>5.1 Nashorn引擎：jjs</h2>
jjs是一个基于标准Nashorn引擎的命令行工具，可以接受js源码并执行。如，我们写一个func.js文件，内容如下：

<pre class="hljs php"><code>function f() {
    return 1;
};
</code> <code>print(f() + 1);</code> </pre>
可以在命令行中执行这个命令：<code>jjs func.js</code>，控制台输出结果是：
<pre class="hljs cpp"><code>2</code></pre>
如果需要了解细节，可以参考官方文档。
<h2>5.2 类依赖分析器：jdeps</h2>
jdeps是一个相当棒的命令行工具，它可以展示包层级和类层级的Java类依关系，它以.class文件、目录或者Jar文件为输入，然后会把依赖关系输出到控台。
我们可以利用jedps分析下Spring Framework库，为了让结果少一点，仅仅分析一个JAR文件：org.springframework.core-3.0.5.RELEASE.jar。
<pre class="hljs css"><code>jdeps org.springframework.core-3.0.5.RELEASE.jar</code></pre>
这个命令会输出很多结果，我们仅看下其中的一部分：依赖关系按照包分组，如果在classpath找不到依赖，则显示"not found".
<pre class="hljs cpp"><code>org.springframework.core-3.0.5.RELEASE.jar -> C:\Program Files\Java\jdk1.8.0\jre\lib\rt.jar
   org.springframework.core (org.springframework.core-3.0.5.RELEASE.jar)
      -> java.io
      -> java.lang
      -> java.lang.annotation
      -> java.lang.ref
      -> java.lang.reflect
      -> java.util
      -> java.util.concurrent
      -> org.apache.commons.logging   not found
      -> org.springframework.asm   not found
      -> org.springframework.asm.commons   not found
   org.springframework.core.annotation (org.springframework.core-3.0.5.RELEASE.jar)
      -> java.lang
      -> java.lang.annotation
      -> java.lang.reflect
      -> java.util</code></pre>
更多的细节可以参考官方文档。
<h2>6. JVM的新特性</h2>
使用<a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.javacode

eeks.com%2F2013%2F02%2Fjava-8-from-permgen-to-metaspace.html" target="_blank" rel=nofollow ugc">Metaspace（JEP 122）代替持久代（PermGen space）。在JVM参数方面，使用-X:MetaSpaceSize和-XX:MaxMetaspaceSize代替原来的XX:PermSize和-XX:MaxPermSize。
<h2>7. 结论</h2>
通过为开发者提供很多能够提高生产力的特性，Java 8使得Java平台前进了一大步。现在还不太合将Java 8应用在生产系统中，但是在之后的几个月中Java 8的应用率一定会逐步提高（PS:原文时间2014年5月9日，现在在很多公司Java 8已经成为主流，我司由于体量太大，现在也在一点点上Java 8虽然慢但是好歹在升级了）。作为开发者，现在应该学习一些Java 8的知识，为升级做好准备。
关于spring</>：对于企业级开发，我们也应该关注Spring社区对Java 8的支持，可以参考这篇文章——Spring 4支持的Java 8新特性一览
<h2>8. 参考资料</h2>

hat’ s New in JDK 8
The Java Tutorials
WildFly 8, JDK 8, NetBeans 8, Java EE
Java 8 Tutorial</>
JDK 8 Command-line Staticependency Checker
The Iluminating Javadoc of JDK
TheDark Side of Java 8
Installing Java™ 8 Support in Eclipse Kpler SR2
Java 8
racle Nashorn. A Next-Generation JavaScript Engine for the JVM

<a href="https://ld246.com/forward?goto=http%3A%2F%2Fblog.csdn.net%2Fycz%2Farticle%2Fdetails%2F50896975" class="bds_tsina" title="分享到新浪微博" target="_blank"

el="nofollow ugc">

