**JDK各个版本的新特性**

对于很多刚接触java语言的初学者来说，要了解一门语言，最好的方式就是要能从基础的版本进行了解，升级的过程，以及升级的新特性，这样才能循序渐进的学好一门语言。今天先为大家介绍一下JDK1.5版本到JDK1.7版本的特性。希望能给予帮助。

**JDK1.5新特性：**

**1.自动装箱与拆箱：**

自动装箱的过程：每当需要一种类型的对象时，这种基本类型就自动地封装到与它相同类型的包装中。

自动拆箱的过程：每当需要一个值时，被装箱对象中的值就被自动地提取出来，没必要再去调用intValue()和doubleValue()方法。

自动装箱，只需将该值赋给一个类型包装器引用，java会自动创建一个对象。

自动拆箱，只需将该对象值赋给一个基本类型即可。

java——类的包装器

类型包装器有：Double,Float,Long,Integer,Short,Character和Boolean

2.枚举

把集合里的对象元素一个一个提取出来。枚举类型使代码更具可读性，理解清晰，易于维护。枚举类型是强类型的，从而保证了系统安全性。而以类的静态字段实现的类似替代模型，不具有枚举的简单性和类型安全性。

简单的用法：JavaEnum简单的用法一般用于代表一组常用常量，可用来代表一类相同类型的常量值。

复杂用法：Java为枚举类型提供了一些内置的方法，同事枚举常量还可以有自己的方法。可以很方便的遍历枚举对象。

3.静态导入

通过使用 import static，就可以不用指定 Constants 类名而直接使用静态成员，包括静态方法。

import xxxx 和 import static xxxx的区别是前者一般导入的是类文件如import java.util.Scanner;后者一般是导入静态的方法，import static java.lang.System.out。

4.可变参数（Varargs）

可变参数的简单语法格式为：

methodName([argumentList], dataType...argumentName);

5.内省（Introspector）

是 Java语言对Bean类属性、事件的一种缺省处理方法。例如类A中有属性name,那我们可以通过getName,setName来得到其值或者设置新 的值。通过getName/setName来访问name属性，这就是默认的规则。Java中提供了一套API用来访问某个属性的getter /setter方法，通过这些API可以使你不需要了解这个规则（但你最好还是要搞清楚），这些API存放于包java.beans中。

一 般的做法是通过类Introspector来获取某个对象的BeanInfo信息，然后通过BeanInfo来获取属性的描述器 （PropertyDescriptor），通过这个属性描述器就可以获取某个属性对应的getter/setter方法，然后我们就可以通过反射机制来 调用这些方法。

6.泛型(Generic)

C++ 通过模板技术可以指定集合的元素类型，而Java在1.5之前一直没有相对应的功能。一个集合可以放任何类型的对象，相应地从集合里面拿对象的时候我们也 不得不对他们进行强制得类型转换。猛虎引入了泛型，它允许指定集合里元素的类型，这样你可以得到强类型在编译时刻进行类型检查的好处。

7.For-Each循环

For-Each循环得加入简化了集合的遍历。假设我们要遍历一个集合对其中的元素进行一些处理。

**JDK 1.6新特性**

有关JDK1.6的新特性reamerit的博客文章已经说的很详细了。

1.Desktop类和SystemTray类

在JDK6中 ,AWT新增加了两个类:Desktop和SystemTray。

前者可以用来打开系统默认浏览器浏览指定的URL,打开系统默认邮件客户端给指定的邮箱发邮件,用默认应用程序打开或编辑文件(比如,用记事本打开以txt为后缀名的文件),用系统默认的打印机打印文档;后者可以用来在系统托盘区创建一个托盘程序.

2.使用JAXB2来实现对象与XML之间的映射

JAXB是Java Architecture for XML Binding的缩写，可以将一个Java对象转变成为XML格式，反之亦然。

我 们把对象与关系数据库之间的映射称为ORM, 其实也可以把对象与XML之间的映射称为OXM(Object XML Mapping). 原来JAXB是Java EE的一部分，在JDK6中，SUN将其放到了Java SE中，这也是SUN的一贯做法。JDK6中自带的这个JAXB版本是2.0, 比起1.0(JSR 31)来，JAXB2(JSR 222)用JDK5的新特性Annotation来标识要作绑定的类和属性等，这就极大简化了开发的工作量。

实 际上，在Java EE 5.0中，EJB和Web Services也通过Annotation来简化开发工作。另外,JAXB2在底层是用StAX(JSR 173)来处理XML文档。除了JAXB之外，我们还可以通过XMLBeans和Castor等来实现同样的功能。

3.理解StAX

StAX(JSR 173)是JDK6.0中除了DOM和SAX之外的又一种处理XML文档的API。

StAX 的来历 ：在JAXP1.3(JSR 206)有两种处理XML文档的方法:DOM(Document Object Model)和SAX(Simple API for XML).

由 于JDK6.0中的JAXB2(JSR 222)和JAX-WS 2.0(JSR 224)都会用到StAX所以Sun决定把StAX加入到JAXP家族当中来，并将JAXP的版本升级到1.4(JAXP1.4是JAXP1.3的维护版 本). JDK6里面JAXP的版本就是1.4. 。

StAX是The Streaming API for XML的缩写，一种利用拉模式解析(pull-parsing)XML文档的API.StAX通过提供一种基于事件迭代器(Iterator)的API让 程序员去控制xml文档解析过程,程序遍历这个事件迭代器去处理每一个解析事件，解析事件可以看做是程序拉出来的，也就是程序促使解析器产生一个解析事件 然后处理该事件，之后又促使解析器产生下一个解析事件，如此循环直到碰到文档结束符；

SAX也是基于事件处理xml文档，但却 是用推模式解析，解析器解析完整个xml文档后，才产生解析事件，然后推给程序去处理这些事件；DOM 采用的方式是将整个xml文档映射到一颗内存树，这样就可以很容易地得到父节点和子结点以及兄弟节点的数据，但如果文档很大，将会严重影响性能。

4.使用Compiler API

现在我 们可以用JDK6 的Compiler API(JSR 199)去动态编译Java源文件，Compiler API结合反射功能就可以实现动态的产生Java代码并编译执行这些代码，有点动态语言的特征。

这 个特性对于某些需要用到动态编译的应用程序相当有用， 比如JSP Web Server，当我们手动修改JSP后，是不希望需要重启Web Server才可以看到效果的，这时候我们就可以用Compiler API来实现动态编译JSP文件，当然，现在的JSP Web Server也是支持JSP热部署的，现在的JSP Web Server通过在运行期间通过Runtime.exec或ProcessBuilder来调用javac来编译代码，这种方式需要我们产生另一个进程去 做编译工作，不够优雅而且容易使代码依赖与特定的操作系统；Compiler API通过一套易用的标准的API提供了更加丰富的方式去做动态编译,而且是跨平台的。

5.轻量级Http Server API

JDK6 提供了一个简单的Http Server API,据此我们可以构建自己的嵌入式Http Server,它支持Http和Https协议,提供了HTTP1.1的部分实现，没有被实现的那部分可以通过扩展已有的Http Server API来实现,程序员必须自己实现HttpHandler接口,HttpServer会调用HttpHandler实现类的回调方法来处理客户端请求,在 这里,我们把一个Http请求和它的响应称为一个交换,包装成HttpExchange类,HttpServer负责将HttpExchange传给 HttpHandler实现类的回调方法.

6.插入式注解处理API(Pluggable Annotation Processing API)

插入式注解处理API(JSR 269)提供一套标准API来处理Annotations(JSR 175)

实 际上JSR 269不仅仅用来处理Annotation,我觉得更强大的功能是它建立了Java 语言本身的一个模型,它把method, package, constructor, type, variable, enum, annotation等Java语言元素映射为Types和Elements(两者有什么区别?), 从而将Java语言的语义映射成为对象, 我们可以在javax.lang.model包下面可以看到这些类. 所以我们可以利用JSR 269提供的API来构建一个功能丰富的元编程(metaprogramming)环境.

JSR 269用Annotation Processor在编译期间而不是运行期间处理Annotation, Annotation Processor相当于编译器的一个插件,所以称为插入式注解处理.如果Annotation Processor处理Annotation时(执行process方法)产生了新的Java代码,编译器会再调用一次Annotation Processor,如果第二次处理还有新代码产生,就会接着调用Annotation Processor,直到没有新代码产生为止.每执行一次process()方法被称为一个"round",这样整个Annotation processing过程可以看作是一个round的序列.

JSR 269主要被设计成为针对Tools或者容器的API. 举个例子,我们想建立一套基于Annotation的单元测试框架(如TestNG),在测试类里面用Annotation来标识测试期间需要执行的测试方法。

7.用Console开发控制台程序

JDK6 中提供了java.io.Console 类专用来访问基于字符的控制台设备. 你的程序如果要与Windows下的cmd或者Linux下的Terminal交互,就可以用Console类代劳. 但我们不总是能得到可用的Console, 一个JVM是否有可用的Console依赖于底层平台和JVM如何被调用. 如果JVM是在交互式命令行(比如Windows的cmd)中启动的,并且输入输出没有重定向到另外的地方,那么就可以得到一个可用的Console实 例.

8.对脚本语言的支持如: ruby, groovy, javascript.

9.Common Annotations

Common annotations原本是Java EE 5.0(JSR 244)规范的一部分，现在SUN把它的一部分放到了Java SE 6.0中.

随 着Annotation元数据功能(JSR 175)加入到Java SE 5.0里面，很多Java 技术(比如EJB,Web Services)都会用Annotation部分代替XML文件来配置运行参数（或者说是支持声明式编程,如EJB的声明式事务）, 如果这些技术为通用目的都单独定义了自己的Annotations,显然有点重复建设, 所以,为其他相关的Java技术定义一套公共的Annotation是有价值的，可以避免重复建设的同时，也保证Java SE和Java EE 各种技术的一致性.

下面列举出Common Annotations 1.0里面的10个Annotations Common Annotations

Annotation Retention Target Description

Generated Source ANNOTATION\_TYPE, CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE 用于标注生成的源代码

Resource Runtime TYPE, METHOD, FIELD 用于标注所依赖的资源,容器据此注入外部资源依赖，有基于字段的注入和基于setter方法的注入两种方式

Resources Runtime TYPE 同时标注多个外部依赖，容器会把所有这些外部依赖注入

PostConstruct Runtime METHOD 标注当容器注入所有依赖之后运行的方法，用来进行依赖注入后的初始化工作，只有一个方法可以标注为PostConstruct

PreDestroy Runtime METHOD 当对象实例将要被从容器当中删掉之前，要执行的回调方法要标注为PreDestroy RunAs Runtime TYPE 用于标注用什么安全角色来执行被标注类的方法，这个安全角色必须和Container 的Security角色一致的。RolesAllowed Runtime TYPE, METHOD 用于标注允许执行被标注类或方法的安全角色，这个安全角色必须和Container 的Security角色一致的

PermitAll Runtime TYPE, METHOD 允许所有角色执行被标注的类或方法

DenyAll Runtime TYPE, METHOD 不允许任何角色执行被标注的类或方法，表明该类或方法不能在Java EE容器里面运行

DeclareRoles Runtime TYPE 用来定义可以被应用程序检验的安全角色，通常用isUserInRole来检验安全角色

注意:

1.RolesAllowed,PermitAll,DenyAll不能同时应用到一个类或方法上

2.标注在方法上的RolesAllowed,PermitAll,DenyAll会覆盖标注在类上的RolesAllowed,PermitAll,DenyAll

3.RunAs,RolesAllowed,PermitAll,DenyAll和DeclareRoles还没有加到Java SE 6.0上来

4. 处理以上Annotations的工作是由Java EE容器来做, Java SE 6.0只是包含了上面表格的前五种Annotations的定义类,并没有包含处理这些Annotations的引擎,这个工作可以由Pluggable Annotation Processing API(JSR 269)来做

改动的地方最大的就是java GUI界面的显示了，JDK6.0（也就是JDK1.6）支持最新的windows vista系统的Windows Aero视窗效果，而JDK1.5不支持！！！

你要在vista环境下编程的话最好装jdk6.0，否则它总是换到windows basic视窗效果.

**JDK 1.7 新特性**

|  |
| --- |
| 1，switch中可以使用字串了  String s = "test"; switch (s) {  case "test" :    System.out.println("test"); case "test1" :    System.out.println("test1"); break ; default :   System.out.println("break"); break ; } 2，"<>"这个玩意儿的运用List<String> tempList = new ArrayList<>(); 即泛型实例化类型自动推断。  [复制代码](javascript:void(0);)  mport java.util.\*;  public class JDK7GenericTest {  public static void main(String[] args) {  // Pre-JDK 7  List<String> lst1 = new ArrayList<String>();  // JDK 7 supports limited type inference for generic instance creation  List<String> lst2 = new ArrayList<>();    lst1.add("Mon");  lst1.add("Tue");  lst2.add("Wed");  lst2.add("Thu");    for (String item: lst1) {  System.out.println(item);  }    for (String item: lst2) {  System.out.println(item);  }  }  }  [复制代码](javascript:void(0);)    3. 自定义自动关闭类  以下是jdk7 api中的接口，（不过注释太长，删掉了close()方法的一部分注释）  [复制代码](javascript:void(0);)  /\*\*  \* A resource that must be closed when it is no longer needed.  \*  \* @author Josh Bloch  \* @since 1.7  \*/  public interface AutoCloseable {  /\*\*  \* Closes this resource, relinquishing any underlying resources.  \* This method is invoked automatically on objects managed by the  \* {@code try}-with-resources statement.  \*  \*/  void close() throws Exception;  }  [复制代码](javascript:void(0);)  只要实现该接口，在该类对象销毁时自动调用close方法，你可以在close方法关闭你想关闭的资源，例子如下  [复制代码](javascript:void(0);)  class TryClose implements AutoCloseable {  @Override  public void close() throw Exception {  System.out.println(" Custom close method …  close resources ");  }  } //请看jdk自带类BufferedReader如何实现close方法（当然还有很多类似类型的类）    public void close() throws IOException {         synchronized (lock) {             if (in == null)                 return;             in.close();             in = null;             cb = null;         }     }  [复制代码](javascript:void(0);)    4. 新增一些取环境信息的工具方法   File System.getJavaIoTempDir() // IO临时文件夹  File System.getJavaHomeDir() // JRE的安装目录  File System.getUserHomeDir() // 当前用户目录  File System.getUserDir() // 启动java进程时所在的目录  .......  5. Boolean类型反转，空指针安全,参与位运算  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 Booleans.and(Boolean[] array)  boolean Booleans.or(Boolean[] array)  boolean Booleans.xor(Boolean[] array)  6. 两个char间的equals  boolean Character.equalsIgnoreCase(char ch1, char ch2)  7，安全的加减乘除  int Math.safeToInt(long value)  int Math.safeNegate(int value)  long Math.safeSubtract(long value1, int value2)  long Math.safeSubtract(long value1, long value2)  int Math.safeMultiply(int value1, int value2)  long Math.safeMultiply(long value1, int value2)  long Math.safeMultiply(long value1, long value2)  long Math.safeNegate(long value)  int Math.safeAdd(int value1, int value2)  long Math.safeAdd(long value1, int value2)  long Math.safeAdd(long value1, long value2)  int Math.safeSubtract(int value1, int value2) |

|  |
| --- |
| 1.对Java集合（Collections）的增强支持  在JDK1.7之前的版本中，Java集合容器中存取元素的形式如下：  以List、Set、Map集合容器为例：        //创建List接口对象     List<String> list=new ArrayList<String>();     list.add("item"); //用add()方法获取对象     String Item=list.get(0); //用get()方法获取对象        //创建Set接口对象     Set<String> set=new HashSet<String>();     set.add("item"); //用add()方法添加对象        //创建Map接口对象     Map<String,Integer> map=new HashMap<String,Integer>();     map.put("key",1); //用put()方法添加对象     int value=map.get("key");    在JDK1.7中，摒弃了Java集合接口的实现类，如：ArrayList、HashSet和HashMap。而是直接采用[]、{}的形式存入对象，采用[]的形式按照索引、键值来获取集合中的对象，如下：          List<String> list=["item"]; //向List集合中添加元素       String item=list[0]; //从List集合中获取元素          Set<String> set={"item"}; //向Set集合对象中添加元素          Map<String,Integer> map={"key":1}; //向Map集合中添加对象       int value=map["key"]; //从Map集合中获取对象    2.在Switch中可用String  在之前的版本中是不支持在Switch语句块中用String类型的数据的，这个功能在C#语言中早已被支持，好在JDK1.7中加入了。    String s = "test";         switch (s) {               case "test" :              System.out.println("test");         case "test1" :              System.out.println("test1");              break ;           default :                   System.out.println("break");                   break ; }      3.数值可加下划线  例如：int one\_million = 1\_000\_000;      4.支持二进制文字  例如：int binary = 0b1001\_1001;     5.简化了可变参数方法的调用  当程序员试图使用一个不可具体化的可变参数并调用一个\*varargs\* （可变）方法时，编辑器会生成一个“非安全操作”的警告。    6、在try catch异常扑捉中，一个catch可以写多个异常类型，用"|"隔开，  jdk7之前：  [复制代码](javascript:void(0);)  try {  ......  } catch(ClassNotFoundException ex) {  ex.printStackTrace();  } catch(SQLException ex) {  ex.printStackTrace();  }  [复制代码](javascript:void(0);)  jdk7例子如下  try {  ......  } catch(ClassNotFoundException|SQLException ex) {  ex.printStackTrace();  }    7、jdk7之前，你必须用try{}finally{}在try内使用资源，在finally中关闭资源，不管try中的代码是否正常退出或者异常退出。jdk7之后，你可以不必要写finally语句来关闭资源，只要你在try()的括号内部定义要使用的资源。请看例子：  jdk7之前  [复制代码](javascript:void(0);)  import java.io.\*;  // Copy from one file to another file character by character.  // Pre-JDK 7 requires you to close the resources using a finally block.  public class FileCopyPreJDK7 {  public static void main(String[] args) {  BufferedReader in = null;  BufferedWriter out = null;  try {  in = new BufferedReader(new FileReader("in.txt"));  out = new BufferedWriter(new FileWriter("out.txt"));  int charRead;  while ((charRead = in.read()) != -1) {  System.out.printf("%c ", (char)charRead);  out.write(charRead);  }  } catch (IOException ex) {  ex.printStackTrace();  } finally { // always close the streams  try {  if (in != null) in.close();  if (out != null) out.close();  } catch (IOException ex) {  ex.printStackTrace();  }  }    try {  in.read(); // Trigger IOException: Stream closed  } catch (IOException ex) {  ex.printStackTrace();  }  }  }  [复制代码](javascript:void(0);)  jdk7之后  [复制代码](javascript:void(0);)  import java.io.\*;  // Copy from one file to another file character by character.  // JDK 7 has a try-with-resources statement, which ensures that  // each resource opened in try() is closed at the end of the statement.  public class FileCopyJDK7 {  public static void main(String[] args) {  try (**BufferedReader in = new BufferedReader(new FileReader("in.txt")**);  **BufferedWriter out = new BufferedWriter(new FileWriter("out.txt")**)) {  int charRead;  while ((charRead = in.read()) != -1) {  System.out.printf("%c ", (char)charRead);  out.write(charRead);  }  } catch (IOException ex) {  ex.printStackTrace();  }  }  }  [复制代码](javascript:void(0);) |

**jAVA8 十大新特性**

投稿：junjie 字体：[增加 减小] 类型：转载

本教程将Java8的新特新逐一列出，并将使用简单的代码示例来指导你如何使用默认接口方法，lambda表达式，方法引用以及多重Annotation，之后你将会学到最新的API上的改进，比如流，函数式接口，Map以及全新的日期API

“Java is still not dead—and people are starting to figure that out.”

本教程将用带注释的简单代码来描述新特性，你将看不到大片吓人的文字。

**一、接口的默认方法**  
Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法，示例如下：

 代码如下:

interface Formula {  
    double calculate(int a);

    default double sqrt(int a) {  
        return Math.sqrt(a);  
    }  
}

Formula接口在拥有calculate方法之外同时还定义了sqrt方法，实现了Formula接口的子类只需要实现一个calculate方法，默认方法sqrt将在子类上可以直接使用。

代码如下:

Formula formula = new Formula() {  
    @Override  
    public double calculate(int a) {  
        return sqrt(a \* 100);  
    }  
};

formula.calculate(100);     // 100.0  
formula.sqrt(16);           // 4.0

文中的formula被实现为一个匿名类的实例，该代码非常容易理解，6行代码实现了计算 sqrt(a \* 100)。在下一节中，我们将会看到实现单方法接口的更简单的做法。

译者注： 在Java中只有单继承，如果要让一个类赋予新的特性，通常是使用接口来实现，在C++中支持多继承，允许一个子类同时具有多个父类的接口与功能，在其他 语言中，让一个类同时具有其他的可复用代码的方法叫做mixin。新的Java 8 的这个特新在编译器实现的角度上来说更加接近Scala的trait。 在C#中也有名为扩展方法的概念，允许给已存在的类型扩展方法，和Java 8的这个在语义上有差别。  
**二、Lambda 表达式**  
首先看看在老版本的Java中是如何排列字符串的：

复制代码 代码如下:

List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {  
    @Override  
    public int compare(String a, String b) {  
        return b.compareTo(a);  
    }  
});

只需要给静态方法 Collections.sort 传入一个List对象以及一个比较器来按指定顺序排列。通常做法都是创建一个匿名的比较器对象然后将其传递给sort方法。

在Java 8 中你就没必要使用这种传统的匿名对象的方式了，Java 8提供了更简洁的语法，lambda表达式：

复制代码 代码如下:

Collections.sort(names, (String a, String b) -> {  
    return b.compareTo(a);  
});

看到了吧，代码变得更段且更具有可读性，但是实际上还可以写得更短：

复制代码 代码如下:

Collections.sort(names, (String a, String b) -> b.compareTo(a));

对于函数体只有一行代码的，你可以去掉大括号{}以及return关键字，但是你还可以写得更短点：

复制代码 代码如下:

Collections.sort(names, (a, b) -> b.compareTo(a));

Java编译器可以自动推导出参数类型，所以你可以不用再写一次类型。接下来我们看看lambda表达式还能作出什么更方便的东西来：  
**三、函数式接口**  
Lambda 表达式是如何在java的类型系统中表示的呢？每一个lambda表达式都对应一个类型，通常是接口类型。而“函数式接口”是指仅仅只包含一个抽象方法的 接口，每一个该类型的lambda表达式都会被匹配到这个抽象方法。因为 默认方法 不算抽象方法，所以你也可以给你的函数式接口添加默认方法。

我们可以将lambda表达式当作任意只包含一个抽象方法的接口类型，确保你的接口一定达到这个要求，你只需要给你的接口添加 @FunctionalInterface 注解，编译器如果发现你标注了这个注解的接口有多于一个抽象方法的时候会报错的。

示例如下：

复制代码 代码如下:

@FunctionalInterface  
interface Converter<F, T> {  
    T convert(F from);  
}  
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);  
Integer converted = converter.convert("123");  
System.out.println(converted);    // 123

需要注意如果@FunctionalInterface如果没有指定，上面的代码也是对的。

译者注 将lambda表达式映射到一个单方法的接口上，这种做法在Java 8之前就有别的语言实现，比如Rhino JavaScript解释器，如果一个函数参数接收一个单方法的接口而你传递的是一个function，Rhino 解释器会自动做一个单接口的实例到function的适配器，典型的应用场景有 org.w3c.dom.events.EventTarget 的addEventListener 第二个参数 EventListener。

**四、方法与构造函数引用**  
前一节中的代码还可以通过静态方法引用来表示：

复制代码 代码如下:

Converter<String, Integer> converter = Integer::valueOf;  
Integer converted = converter.convert("123");  
System.out.println(converted);   // 123

Java 8 允许你使用 :: 关键字来传递方法或者构造函数引用，上面的代码展示了如何引用一个静态方法，我们也可以引用一个对象的方法：

复制代码 代码如下:

 converter = something::startsWith;  
String converted = converter.convert("Java");  
System.out.println(converted);    // "J"

接下来看看构造函数是如何使用::关键字来引用的，首先我们定义一个包含多个构造函数的简单类：

复制代码 代码如下:

class Person {  
    String firstName;  
    String lastName;

    Person() {}

    Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}

接下来我们指定一个用来创建Person对象的对象工厂接口：

复制代码 代码如下:

interface PersonFactory<P extends Person> {  
    P create(String firstName, String lastName);  
}

这里我们使用构造函数引用来将他们关联起来，而不是实现一个完整的工厂：

复制代码 代码如下:

PersonFactory<Person> personFactory = Person::new;  
Person person = personFactory.create("Peter", "Parker");

我们只需要使用 Person::new 来获取Person类构造函数的引用，Java编译器会自动根据PersonFactory.create方法的签名来选择合适的构造函数。

**五、Lambda 作用域**  
在lambda表达式中访问外层作用域和老版本的匿名对象中的方式很相似。你可以直接访问标记了final的外层局部变量，或者实例的字段以及静态变量。

六、访问局部变量

我们可以直接在lambda表达式中访问外层的局部变量：

复制代码 代码如下:

final int num = 1;  
Converter<Integer, String> stringConverter =  
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3

但是和匿名对象不同的是，这里的变量num可以不用声明为final，该代码同样正确：

复制代码 代码如下:

int num = 1;  
Converter<Integer, String> stringConverter =  
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3

不过这里的num必须不可被后面的代码修改（即隐性的具有final的语义），例如下面的就无法编译：

复制代码 代码如下:

int num = 1;  
Converter<Integer, String> stringConverter =  
        (from) -> String.valueOf(from + num);  
num = 3;

在lambda表达式中试图修改num同样是不允许的。  
七、访问对象字段与静态变量

和本地变量不同的是，lambda内部对于实例的字段以及静态变量是即可读又可写。该行为和匿名对象是一致的：

复制代码 代码如下:

class Lambda4 {  
    static int outerStaticNum;  
    int outerNum;

    void testScopes() {  
        Converter<Integer, String> stringConverter1 = (from) -> {  
            outerNum = 23;  
            return String.valueOf(from);  
        };

        Converter<Integer, String> stringConverter2 = (from) -> {  
            outerStaticNum = 72;  
            return String.valueOf(from);  
        };  
    }  
}

**八、访问接口的默认方法**  
还记得第一节中的formula例子么，接口Formula定义了一个默认方法sqrt可以直接被formula的实例包括匿名对象访问到，但是在lambda表达式中这个是不行的。  
Lambda表达式中是无法访问到默认方法的，以下代码将无法编译：

复制代码 代码如下:

Formula formula = (a) -> sqrt( a \* 100);  
Built-in Functional Interfaces

JDK 1.8 API包含了很多内建的函数式接口，在老Java中常用到的比如Comparator或者Runnable接口，这些接口都增加了@FunctionalInterface注解以便能用在lambda上。  
Java 8 API同样还提供了很多全新的函数式接口来让工作更加方便，有一些接口是来自Google Guava库里的，即便你对这些很熟悉了，还是有必要看看这些是如何扩展到lambda上使用的。  
Predicate接口

Predicate 接口只有一个参数，返回boolean类型。该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）：

复制代码 代码如下:

Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");              // true  
predicate.negate().test("foo");     // false

Predicate<Boolean> nonNull = Objects::nonNull;  
Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;  
Predicate<String> isNotEmpty = isEmpty.negate();

Function 接口

Function 接口有一个参数并且返回一个结果，并附带了一些可以和其他函数组合的默认方法（compose, andThen）：

复制代码 代码如下:

Function<String, Integer> toInteger = Integer::valueOf;  
Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123");     // "123"

Supplier 接口  
Supplier 接口返回一个任意范型的值，和Function接口不同的是该接口没有任何参数

复制代码 代码如下:

Supplier<Person> personSupplier = Person::new;  
personSupplier.get();   // new Person

Consumer 接口  
Consumer 接口表示执行在单个参数上的操作。

复制代码 代码如下:

Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);  
greeter.accept(new Person("Luke", "Skywalker"));

Comparator 接口  
Comparator 是老Java中的经典接口， Java 8在此之上添加了多种默认方法：

复制代码 代码如下:

Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");  
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);             // > 0  
comparator.reversed().compare(p1, p2);  // < 0

Optional 接口

Optional 不是函数是接口，这是个用来防止NullPointerException异常的辅助类型，这是下一届中将要用到的重要概念，现在先简单的看看这个接口能干什么：

Optional 被定义为一个简单的容器，其值可能是null或者不是null。在Java 8之前一般某个函数应该返回非空对象但是偶尔却可能返回了null，而在Java 8中，不推荐你返回null而是返回Optional。

复制代码 代码如下:

Optional<String> optional = Optional.of("bam");

optional.isPresent();           // true  
optional.get();                 // "bam"  
optional.orElse("fallback");    // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"

Stream 接口

java.util.Stream 表示能应用在一组元素上一次执行的操作序列。Stream 操作分为中间操作或者最终操作两种，最终操作返回一特定类型的计算结果，而中间操作返回Stream本身，这样你就可以将多个操作依次串起来。 Stream 的创建需要指定一个数据源，比如 java.util.Collection的子类，List或者Set， Map不支持。Stream的操作可以串行执行或者并行执行。

首先看看Stream是怎么用，首先创建实例代码的用到的数据List：

复制代码 代码如下:

List<String> stringCollection = new ArrayList<>();  
stringCollection.add("ddd2");  
stringCollection.add("aaa2");  
stringCollection.add("bbb1");  
stringCollection.add("aaa1");  
stringCollection.add("bbb3");  
stringCollection.add("ccc");  
stringCollection.add("bbb2");  
stringCollection.add("ddd1");

Java 8扩展了集合类，可以通过 Collection.stream() 或者 Collection.parallelStream() 来创建一个Stream。下面几节将详细解释常用的Stream操作：

Filter 过滤

过滤通过一个predicate接口来过滤并只保留符合条件的元素，该操作属于中间操作，所以我们可以在过滤后的结果来应用其他Stream操作 （比如forEach）。forEach需要一个函数来对过滤后的元素依次执行。forEach是一个最终操作，所以我们不能在forEach之后来执行 其他Stream操作。

复制代码 代码如下:

stringCollection  
    .stream()  
    .filter((s) -> s.startsWith("a"))  
    .forEach(System.out::println);

// "aaa2", "aaa1"

Sort 排序

排序是一个中间操作，返回的是排序好后的Stream。如果你不指定一个自定义的Comparator则会使用默认排序。

复制代码 代码如下:

stringCollection  
    .stream()  
    .sorted()  
    .filter((s) -> s.startsWith("a"))  
    .forEach(System.out::println);

// "aaa1", "aaa2"

需要注意的是，排序只创建了一个排列好后的Stream，而不会影响原有的数据源，排序之后原数据stringCollection是不会被修改的：

复制代码 代码如下:

System.out.println(stringCollection);  
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

Map 映射  
中间操作map会将元素根据指定的Function接口来依次将元素转成另外的对象，下面的示例展示了将字符串转换为大写字符串。你也可以通过map来讲对象转换成其他类型，map返回的Stream类型是根据你map传递进去的函数的返回值决定的。

复制代码 代码如下:

stringCollection  
    .stream()  
    .map(String::toUpperCase)  
    .sorted((a, b) -> b.compareTo(a))  
    .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"

Match 匹配

Stream提供了多种匹配操作，允许检测指定的Predicate是否匹配整个Stream。所有的匹配操作都是最终操作，并返回一个boolean类型的值。

复制代码 代码如下:

boolean anyStartsWithA =   
    stringCollection  
        .stream()  
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);      // true

boolean allStartsWithA =   
    stringCollection  
        .stream()  
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);      // false

boolean noneStartsWithZ =   
    stringCollection  
        .stream()  
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);      // true

Count 计数  
计数是一个最终操作，返回Stream中元素的个数，返回值类型是long。

复制代码 代码如下:

long startsWithB =   
    stringCollection  
        .stream()  
        .filter((s) -> s.startsWith("b"))  
        .count();

System.out.println(startsWithB);    // 3

Reduce 规约

这是一个最终操作，允许通过指定的函数来讲stream中的多个元素规约为一个元素，规越后的结果是通过Optional接口表示的：

复制代码 代码如下:

Optional<String> reduced =  
    stringCollection  
        .stream()  
        .sorted()  
        .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);  
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"

并行Streams

前面提到过Stream有串行和并行两种，串行Stream上的操作是在一个线程中依次完成，而并行Stream则是在多个线程上同时执行。

下面的例子展示了是如何通过并行Stream来提升性能：

首先我们创建一个没有重复元素的大表：

复制代码 代码如下:

int max = 1000000;  
List<String> values = new ArrayList<>(max);  
for (int i = 0; i < max; i++) {  
    UUID uuid = UUID.randomUUID();  
    values.add(uuid.toString());  
}

然后我们计算一下排序这个Stream要耗时多久，  
串行排序：

复制代码 代码如下:

long t0 = System.nanoTime();

long count = values.stream().sorted().count();  
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);  
System.out.println(String.format("sequential sort took: %d ms", millis));

// 串行耗时: 899 ms  
并行排序：

复制代码 代码如下:

long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();  
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);  
System.out.println(String.format("parallel sort took: %d ms", millis));

// 并行排序耗时: 472 ms  
上面两个代码几乎是一样的，但是并行版的快了50%之多，唯一需要做的改动就是将stream()改为parallelStream()。

Map

前面提到过，Map类型不支持stream，不过Map提供了一些新的有用的方法来处理一些日常任务。

复制代码 代码如下:

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

for (int i = 0; i < 10; i++) {  
    map.putIfAbsent(i, "val" + i);  
}

map.forEach((id, val) -> System.out.println(val));  
以上代码很容易理解， putIfAbsent 不需要我们做额外的存在性检查，而forEach则接收一个Consumer接口来对map里的每一个键值对进行操作。

下面的例子展示了map上的其他有用的函数：

复制代码 代码如下:

map.computeIfPresent(3, (num, val) -> val + num);  
map.get(3);             // val33

map.computeIfPresent(9, (num, val) -> null);  
map.containsKey(9);     // false

map.computeIfAbsent(23, num -> "val" + num);  
map.containsKey(23);    // true

map.computeIfAbsent(3, num -> "bam");  
map.get(3);             // val33

接下来展示如何在Map里删除一个键值全都匹配的项：

复制代码 代码如下:

map.remove(3, "val3");  
map.get(3);             // val33

map.remove(3, "val33");  
map.get(3);             // null

另外一个有用的方法：

复制代码 代码如下:

map.getOrDefault(42, "not found");  // not found

对Map的元素做合并也变得很容易了：

复制代码 代码如下:

map.merge(9, "val9", (value, newValue) -> value.concat(newValue));  
map.get(9);             // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));  
map.get(9);             // val9concat

Merge做的事情是如果键名不存在则插入，否则则对原键对应的值做合并操作并重新插入到map中。

**九、Date API**  
Java 8 在包java.time下包含了一组全新的时间日期API。新的日期API和开源的Joda-Time库差不多，但又不完全一样，下面的例子展示了这组新API里最重要的一些部分：  
Clock 时钟

Clock类提供了访问当前日期和时间的方法，Clock是时区敏感的，可以用来取代 System.currentTimeMillis() 来获取当前的微秒数。某一个特定的时间点也可以使用Instant类来表示，Instant类也可以用来创建老的java.util.Date对象。

复制代码 代码如下:

Clock clock = Clock.systemDefaultZone();  
long millis = clock.millis();

Instant instant = clock.instant();  
Date legacyDate = Date.from(instant);   // legacy java.util.Date

**Timezones 时区**

在新API中时区使用ZoneId来表示。时区可以很方便的使用静态方法of来获取到。 时区定义了到UTS时间的时间差，在Instant时间点对象到本地日期对象之间转换的时候是极其重要的。

复制代码 代码如下:

System.out.println(ZoneId.getAvailableZoneIds());  
// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");  
ZoneId zone2 = ZoneId.of("Brazil/East");  
System.out.println(zone1.getRules());  
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]  
// ZoneRules[currentStandardOffset=-03:00]

**LocalTime 本地时间**

LocalTime 定义了一个没有时区信息的时间，例如 晚上10点，或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。之后比较时间并以小时和分钟为单位计算两个时间的时间差：

复制代码 代码如下:

LocalTime now1 = LocalTime.now(zone1);  
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);  
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween);       // -3  
System.out.println(minutesBetween);     // -239

LocalTime 提供了多种工厂方法来简化对象的创建，包括解析时间字符串。

复制代码 代码如下:

LocalTime late = LocalTime.of(23, 59, 59);  
System.out.println(late);       // 23:59:59

DateTimeFormatter germanFormatter =  
    DateTimeFormatter  
        .ofLocalizedTime(FormatStyle.SHORT)  
        .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);  
System.out.println(leetTime);   // 13:37

**LocalDate 本地日期**

LocalDate 表示了一个确切的日期，比如 2014-03-11。该对象值是不可变的，用起来和LocalTime基本一致。下面的例子展示了如何给Date对象加减天/月/年。另外要注意的是这些对象是不可变的，操作返回的总是一个新实例。

复制代码 代码如下:

LocalDate today = LocalDate.now();  
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);  
LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);  
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();

System.out.println(dayOfWeek);    // FRIDAY  
从字符串解析一个LocalDate类型和解析LocalTime一样简单：

复制代码 代码如下:

DateTimeFormatter germanFormatter =  
    DateTimeFormatter  
        .ofLocalizedDate(FormatStyle.MEDIUM)  
        .withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);  
System.out.println(xmas);   // 2014-12-24

**LocalDateTime 本地日期时间**

LocalDateTime 同时表示了时间和日期，相当于前两节内容合并到一个对象上了。LocalDateTime和LocalTime还有LocalDate一样，都是不可变的。LocalDateTime提供了一些能访问具体字段的方法。

复制代码 代码如下:

LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();  
System.out.println(dayOfWeek);      // WEDNESDAY

Month month = sylvester.getMonth();  
System.out.println(month);          // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY);  
System.out.println(minuteOfDay);    // 1439

只要附加上时区信息，就可以将其转换为一个时间点Instant对象，Instant时间点对象可以很容易的转换为老式的java.util.Date。

复制代码 代码如下:

Instant instant = sylvester  
        .atZone(ZoneId.systemDefault())  
        .toInstant();

Date legacyDate = Date.from(instant);  
System.out.println(legacyDate);     // Wed Dec 31 23:59:59 CET 2014

格式化LocalDateTime和格式化时间和日期一样的，除了使用预定义好的格式外，我们也可以自己定义格式：

复制代码 代码如下:

DateTimeFormatter formatter =  
    DateTimeFormatter  
        .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);  
String string = formatter.format(parsed);  
System.out.println(string);     // Nov 03, 2014 - 07:13

和java.text.NumberFormat不一样的是新版的DateTimeFormatter是不可变的，所以它是线程安全的。  
关于时间日期格式的详细信息：http://download.java.net/jdk8/docs/api/java/time/format/DateTimeFormatter.html

**十、Annotation 注解**  
在Java 8中支持多重注解了，先看个例子来理解一下是什么意思。  
首先定义一个包装类Hints注解用来放置一组具体的Hint注解：

复制代码 代码如下:

@interface Hints {  
    Hint[] value();  
}

@Repeatable(Hints.class)  
@interface Hint {  
    String value();  
}

Java 8允许我们把同一个类型的注解使用多次，只需要给该注解标注一下@Repeatable即可。

例 1: 使用包装类当容器来存多个注解（老方法）

复制代码 代码如下:

@Hints({@Hint("hint1"), @Hint("hint2")})  
class Person {}

**例 2：使用多重注解（新方法）**

复制代码 代码如下:

@Hint("hint1")  
@Hint("hint2")  
class Person {}

第二个例子里java编译器会隐性的帮你定义好@Hints注解，了解这一点有助于你用反射来获取这些信息：

复制代码 代码如下:

Hint hint = Person.class.getAnnotation(Hint.class);  
System.out.println(hint);                   // null

Hints hints1 = Person.class.getAnnotation(Hints.class);  
System.out.println(hints1.value().length);  // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);  
System.out.println(hints2.length);          // 2

即便我们没有在Person类上定义@Hints注解，我们还是可以通过 getAnnotation(Hints.class) 来获取 @Hints注解，更加方便的方法是使用 getAnnotationsByType 可以直接获取到所有的@Hint注解。  
另外Java 8的注解还增加到两种新的target上了：

复制代码 代码如下:

@Target({ElementType.TYPE\_PARAMETER, ElementType.TYPE\_USE})  
@interface MyAnnotation {}

关于Java 8的新特性就写到这了，肯定还有更多的特性等待发掘。JDK 1.8里还有很多很有用的东西，比如Arrays.parallelSort, StampedLock和CompletableFuture等等。