第 2 章　Java 新特性

2.1　JDK1.5 新特性

本章主要内容：

- 自动装箱/拆箱
- 类型安全枚举
- 静态导入
- annotation
- 增强 for 循环
- 可变长参数
- 格式化输出
- 泛型

2.1.1　自动装箱/拆箱

一般来说，当创建一个类的对象实例的时候，可使用如下代码：

```
Class a = new Class(parameter);
```

当创建一个 Integer 对象时，可使用如下代码：

```
Integer i = 100; (注意：不是 int i = 100; )
```

实际上，执行上面的代码的时候，系统执行了 Integer i = Integer.valueOf(100);，这就是基本数据类型的自动装箱功能。

下面来了解一下装箱/拆箱的概念。

1．装箱和拆箱操作

装箱操作：将基本数据类型转换为它所对应的包装器类的操作。

拆箱操作：将包装器类转换为对应的基本数据类型的操作。

2．包装器类

Java 中提倡一切都是面向对象的，数据类型分为基本数据类型和引用类型，位于 java.lang 包下，如表 2-1 所示。

表 2-1　数据类型

8 种基本数据类型	包装器类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

包装器类可分为以下两组。

数值型：Byte、Short、Integer、Long、Float、Double，都是 Number 类的子类。

其他类型：Character、Boolean。

Number 类的结构如下。

```
byteValue()
intValue()
.......
```

```
/*Value()主要用于获取该引用类型所代表的基本类型数据，以便进行+、-、*、/操作*/
charValue()
booleanValue()
```

3．进行装箱和拆箱操作的方法

（1）Integer i = 100;：编译器自动做语法编译并进行装箱操作，即 Integer i = Integer.valueOf(100);。

（2）int t=i;：编译器自动做语法编译并进行拆箱操作，即 int t = i.intValue();。

4．自动装箱和自动拆箱

自动装箱：基本数据类型自动转换为包装器类。

自动拆箱：包装器类自动转换为基本数据类型。

这样就大大简化了基本数据类型和引用类型之间的转换。

举例：BoxingTest.java。

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class BoxingTest {

    /*
     * 自动装箱、拆箱
     */
    public static void main(String[] args) {
        /*
         * 自动装箱，编译器会自动将其解析为 Integer i = Integer.valueOf(2);
         */
        Integer i = 2;
        /*
         * 自动拆箱，编译器会自动将其解析为 int j = i.intValue();
         */
        int j = i;
        System.out.println("i:" + i + " j:" + j);
        System.out.println("***********");

        Integer counter = 0;//自动装箱
        //拆箱-->累加-->装箱
        counter++;
        System.out.println("counter:" + counter);
        System.out.println("***********");

        Integer a = 4;//自动装箱
        Integer b = 5;//自动装箱
        //先自动拆箱，再比较
        System.out.println(a > b);
        System.out.println("***********");

        Boolean f1 = true;//自动装箱
        Boolean f2 = false;//自动装箱
        boolean f3 = true;
```

```
            //自动拆箱-->运算-->自动装箱
            Boolean result = (f1 || f2) && f3;
            System.out.println(result);
            System.out.println("***********");

            List list = new ArrayList();
            //JDK1.5 之前
            list.add(new Integer(1));
            list.add(new Integer(2));
            //JDK1.5 以后
            list.add(3);
            list.add(4);
            outList(list);
        }

        private static void outList(List list) {
            //使用 while 循环遍历集合
            Iterator iter = list.iterator();
            while (iter.hasNext()) {
                Object o = iter.next();
                System.out.println(o);
            }
            System.out.println("**********");
            //使用 for 循环遍历集合
            for (Iterator iter2 = list.iterator(); iter2.hasNext();) {
                Object o = iter2.next();
                System.out.println(o);
            }
        }
    }
```

5．进行装箱和拆箱操作的原因

在 Java 中，所有要处理的元素几乎都是对象，然而基本数据类型不是对象，有时需要将基本数据类型转换为对象，通过装箱即可轻松实现。

自动装箱与拆箱的设计是一种模式，即享元模式，也就是使用共享物件，用来尽可能地减少内存使用量以及分享资讯给尽可能多的相似物件；它适用于只是因重复而导致使用无法令人接受的大量内存的大量物件。因此使用自动装箱和拆箱机制，可以节省常用数值的内存开销和创建对象的开销，提高效率。

6．包装器类的缓存方式

并不是所有的包装器类内部都有缓存池，以下是各包装器类内部的缓存方式。

（1）Boolean：全部缓存。

（2）Byte：全部缓存。

（3）Character：缓存 ASCII/Unicode<127 的所有字符。

（4）Integer：缓存–128～127 内的数值。

（5）Short：缓存–128～127 内的数值。

（6）Long：缓存–128～127 内的数值。

（7）Float：全部不缓存。

（8）Double：全部不缓存。

2.1.2 增强 for 循环

1. 语法

对于之前的 for 循环，可使用如下格式。

```
for(初始化变量;循环条件;变量控制) {
    //操作
}
for(int i=0;i<10;i++) {
    //some operation
}
```

JDK1.5 引入的增强 for 循环简化了对集合或数组的遍历。

```
for(type element:arr) {
    //some operation
}
```

其中，type 为类型，指所遍历的数组或集合中所存放的数据类型；element 为元素，指遍历时的临时变量；arr 为所要遍历的数组或集合的引用。

2. 弊端

增强 for 循环的缺点如下。

（1）程序本身无法明确指向所遍历元素的索引位置。

（2）容易出现 ClassCastException 类型转换异常。

2.1.3 类型安全枚举

JDK1.5 引入了一个全新的“类”，即枚举类型，其中包括 enum、class、interface、annotation。

1. 类型安全枚举的作用

例如，定义一个代表星期/性别的类。

```
int  weekday=0
```

星期类只能取值星期一～星期日，而性别只能取男、女两个值。

使用枚举主要是为了控制变量的取值，使变量只能是若干个固定值中的一个，否则编译器报错。枚举会使编译器在编译期间检查源程序的非法值，普通变量无法实现这一目标。

举例：

```
public  class Gender{
    private String name;
    private static Gender male = new Gender("男");
    private static Gender female = new Gender("女");

private Gender() {}
    private Gender(String name) {
        this.name = name;
    }

public static Gender getInstance(int i) {
    switch(i) {
```

```
            case 0:
                return MALE;
            case 1:
                return FEMALE;
            default:
                return null;
        }
    }

        //getter setter
        }

    Gender  male = new Gender("男");
    Gender  female = new Gender("女");
    Gender  female = new Gender("不男不女");
```

以上程序简单实现了性别的枚举，可以看到，不使用枚举类型时，需要编写很多代码来实现相应的功能。

2．枚举的定义

枚举的定义如下。

```
public  enum  Gender{
    male,female
}
```

其中，每一个枚举元素都是该枚举类型的一个实例。

3．枚举的使用

枚举的语法格式如下。

```
类名  引用变量 = 类名.枚举元素
```

举例：

```
Gender male = Gender.male;
Gender femal  = Gender.femal;
```

4．枚举的遍历

对于任意一个自定义的枚举类型，默认提供了两个有用的静态方法：values()和 valueOf()。

（1）values()：返回一个包含该枚举类中所有枚举元素的数组。

（2）valueOf(String str)：通过枚举类获取该类的一个枚举元素。

5．Enum 类和 enum 关键字的区别

（1）任意一个使用 enum 关键字定义的枚举类都默认继承于 java.lang.Enum 类。

（2）枚举类型中的所有枚举元素都是该枚举类型的实例，它们都被预设为 public static final。

6．Enum 类

Enum 类的语法格式如下。

```
protected  Enum(String name, int ordinal) {}
```

其中，name 代表枚举元素的名称，ordinal 代表枚举元素的编号。

任何一个枚举元素一旦声明都会默认调用此构造方法进行编号，编号采用自动编号方式进行（从 0 开始）。

7．枚举类型的属性和方法

枚举类型可以定义属性和方法，但是必须位于元素列表声明之后。

8．枚举类型的构造方法

（1）构造方法必须位于元素列表声明之后。
（2）构造方法是 private 的，默认也是私有的。
（3）如果要调用有参构造方法，则应直接在元素声明之后加上参数列表“(参数)”。

9．枚举类型的继承性

由于枚举类型默认继承于 java.lang.Enum 类，因此无法再继承于其他类。

10．枚举实现接口

（1）枚举类像普通类一样，在类中实现接口中的所有抽象方法。
（2）每一个枚举元素分别实现接口中的抽象方法。

11．在枚举中定义抽象方法

在枚举类中可以定义一个或多个抽象方法，但是每一个枚举元素必须分别实现这些抽象方法。

12．switch 对枚举的支持

举例：

```
Color color = Color.getColor();//Color 是一个枚举类，getColor()随机获取一个枚举元素
switch (color) {
    case RED:
        System.out.println("RED");
        break;
    case GREEN:
        System.out.println("GREEN");
        break;
    case BLUE:
        System.out.println("BLUE");
        break;
    case YELLOW:
        System.out.println("YELLOW");
        break;
    default:
        System.out.println("unknow!!");
        break;
    }
```

switch 还支持以下类型：byte、int、short、char。

13．类集对枚举的支持

JDK1.5 引入了两个类集操作类：EnumMap、EnumSet，其位于 java.util 包中。

（1）EnumMap：实现了 Map 接口，基本功能和 Map 相似，实例化时需要指定键值类型，并且键值类型只能是枚举类型。

（2）EnumSet：实现了 Set 接口，基本功能和 Set 类似，构造方法私有化，通过提供的一系列静态方法，如 allOf()/noneOf()等，获取实例化对象。

2.1.4　可变长参数

JDK1.5 引入了可变长参数，使得用户可以声明一个可接收可变数目参数的方法。

举例：

```
public double  sum(double a,double b) {
    return a+b;
}
public double  sum(double a,double b,double c) {
    return a+b+c;
}

public double sum(double[] para) {
    //对数组进行遍历相加
    //需要提前声明一个数组
}
double[]  para = new double[]{1.0,2.0,3.0,4.0};
sum(para);
```

通过以上代码可以知道，通过方法重载，也可以实现可变长参数的方法。当然，这种写法是比较麻烦的，而使用可变长参数的新特性，可以很简单地实现。

1．可变长参数语法

```
public double sum(double... para) {
    //JDK1.5 将可变参数改为数组以进行操作
    //不需要声明数组
    //直接引用
}
sum(1.0);
sum(1.0,2.0);
double[] para = new double[]{1.0,2.0};
sum(para);
```

2．可变长参数使用

使用可变长参数后，在调用该可变参数方法时可依据类型传入一个或多个同一类型的参数或者传入一个该类型的数组参数，在方法中可依据数组方式进行处理。

（1）传递离散值：sum(1.0,2.0);。

（2）传递数组：sum(para);。

（3）传递空值：sum();，当传递值为空时，方法内部接收的不是 null，而是一个长度为 0 的数组。

注意：一个方法中最多只能声明一个可变长参数，并且该可变长参数必须位于参数列表的最后一位。

举例：VarargsTest.java。

```
public class VarargsTest {
    public static void oldSum(int[] values) {
        int total = 0;
        for (int i : values) {
```

```
                total += i;
            }
            System.out.println("total:" + total);
        }

        public static void newSum(int... values) {
            int total = 0;
            for (int i : values) {
                total += i;
            }
            System.out.println("total:" + total);
        }

        public static void main(String[] args) {
            int[] values = new int[] { 1, 2, 3, 4 };
            System.out.println("**before jdk1.5**");
            oldSum(values);
            System.out.println("**after jdk1.5**");
            newSum(1, 2, 3, 4, 5);
            System.out.println("**********");
            Student s1 = new Student("briup",10);
            System.out.println(s1);
            System.out.println("**********");
            Student s2 =new Student("briup2",20,"Shanghai");
            System.out.println(s2);
            System.out.println("**********");
            Student s3 = new Student("briup3",30,"Shanghai","Beijing");
            System.out.println(s3);
        }
    }
```

2.1.5 静态导入

在 JDK1.5 之前，要使用静态成员（变量和方法），必须给出提供该静态成员的类，通过“类名.静态成员”来访问静态成员。

JDK1.5 使用了静态导入，使被导入类的静态成员在当前类中直接可见，使用时不需要提供类名，就像使用当前类的成员一样。

1. 静态导入语法

```
import static 类的全名.静态成员;
java.lang.Math;
```

2. 静态导入缺陷

过度地使用静态导入会在一定程度上降低代码的可读性。

2.1.6 格式化输出

C 语言使用 printf 进行格式化输出。

举例：

```
int num=10;
printf("this is %d",num);
```

输出结果：

```
this is 10
```

JDK1.5 引入格式化输出，并且 Java 的格式化输出语法比 C 更加严格。

Java 的格式化输出由 java.util.Formatter 支持。

1．Formatter

Formatter 是 Java 中用来格式化输出的类。其语法格式如下。

```
format(String format,Object... args){}
```

其中，第一个参数是包含格式化说明符的格式化字符串，第二个参数是替换格式化说明符的可变参数列表。

举例：

```
format("this is %s,%d","str",100);
```

输出结果：

```
this is str, 100
```

2．常规说明符语法

常规说明符的语法格式如下。

```
%[argument_index$][flags][width][.precision]conversion
```

其中，argument_index 代表参数在参数列表中的位置，从 1 开始；flags 表示输出格式的标志；width 表示占用的宽度；.precision 表示字符位数，若为浮点数，则表示小数点后保留的位数；conversion 表示具体的转换说明符。

3．时间/日期的格式化说明符语法

```
%[argument_index$][flags][width]conversion
```

其中，conversion 由 t 或 T 加上具体的转换符组成。例如，%tH 或%TH。

String 类和 PrintStream 也增加了对格式化输出的支持。

String 类增加了静态方法：format(String format,Object... args){}。

PrintStream 增加了 printf 方法：printf(String format,Object... args){}。

2.1.7　泛型

泛型的本质是参数化类型，即操作的数据类型被指定为一个参数，该参数可以用在类、接口和方法的定义中，分别称为泛型类、泛型接口和泛型方法。JDK1.5 引入泛型的最主要的目的是安全、简单。

1．泛型的特点

泛型的特点是类型安全，不需要强制类型转换。

2．使用泛型的原因

例如，构建一个类 Circle，提供属性 Radius，构建该类并打印出成员，要求属性 Radius 类型为 Integer、Float、Double。

```
IntegerCircle.java
FloatCircle.java
```

在 JDK1.5 之前，为了解决类的通用性，通常把参数类型、返回类型设置为 Object。

Object 存在的问题如下：要进行类型转换，容易出现 ClassCastException。

3．泛型的定义

通过引入泛型，可以获得编译时类型的安全性和运行时更小地抛出 ClassCastException 的可能性。

4．使用泛型的方法

1．泛型类

定义类时，声明一个类型参数

```
class 类名<类型参数,类型参数,...> {
}
```

类型参数可以是任意合法的标识符，一般约定：T 表示 Type，E 表示 Element，K 表示 Key，V 表示 Value。

例如，ArrayList<E>。

其中，ArrayList 称为 ArrayList<E>的原始类型；E 为类型参数；<>读作 typeof；ArrayList<E>称为 ArrayList 的泛型类。

1）注意事项

① 实例化泛型类时，参数类型只能是引用类型，不能使用基本数据类型来实例化泛型类。

② 使用时如果不指定参数类型，则 T 默认为 Object 类型。

③ 泛型类的引用可以指向原始类型。

④ 原始类型的引用也可以指向泛型类。

⑤ 参数类型之间不具有继承性。

2）限制泛型使用类别

在定义泛型类时，预设可以使用任意类型来实例化泛型类中的类型，但是要限制使用泛型类别，当只能是某个特定类或其子类才能实例化该泛型类别时，可以使用 extends 关键字指定该类型是继承于某个类或实现某个接口的。

如果不使用 extends 关键字指定，则默认是 T extends Object。

举例：

```
Generic<T extends Number>
Generic<Integer> g = new Integer<Integer>();
Generic<String> g = new Integer<String>();
```

3）泛型通配符 ？

？：可以匹配任意类型。

？ extends 类型：代表该类型为指定类型或子类。

？ super 类型：代表该类型为指定类型或父类。

4）泛型上限

泛型上限指参数类型所能操作的最大上限。

T extends Class：定义类时指定所能操作的最大上限。

？ extends Class：定义引用所能接受的最大上限。

5）泛型下限

泛型下限指定具体类，通过“？ super someClass”定义引用所能接受的最小下限。

6）泛型继承类别/泛型实现接口

① 子类的类型参数声明必须和父类一致。

```
public class Sub<T> extends Super<T> {}
```

② 如果父类的参数类型指定，则子类可以是泛型类，也可以不是泛型类。

```
public class Sub extends Super<String> {}
public class Sub<T> extends Super<String> {}
```

2．泛型接口

泛型接口和泛型类的定义相同。

语法：

```
interface interfaceName<类型参数,类型参数...> {
}
```

举例：

```
public interface Test<T> {
    public void fun(T t);
}
```

3．泛型方法

如果一个方法要使用泛型，那么该参数类型必须提前声明；存在于泛型类中时，由于类型参数在类定义时声明过，所以可以直接使用类型参数；存在于非泛型类中时，由于类型参数在类定义时没有提前声明，所以需要在方法的返回类型前声明参数类型；存在于泛型类中时，当泛型方法的类型参数声明和类的类型参数声明不一致时，该泛型方法等同于非泛型类中的泛型方法。

1）不能实例化类型参数

```
class  Generic<T> {}
new T();   //错误
```

2）在创建数组时不能使用泛型类型

```
ArrayList<String>[] list = new ArrayList<String>[3];(X)
ArrayList[] list = new ArrayList[3];
```

3）静态成员变量不能使用泛型参数类型

```
private  static T name; //错误
```

4）静态方法中不能调用类型为参数类型的成员变量。

```
private T t;
public static void method(){
  Test test = new Test();
  test.t;  错误
}
```

2.1.8　Annotation

JDK1.5 引入了 annotation，提供了一些不属于源程序的数据，annotation 实际上表示的是一种注释语法，一种标记。

annotation 根据所起的作用大致分为如下两类。

（1）编译检查：仅作为一个标记，使编译器在编译时进行一些特殊处理。

（2）代码处理：在运行时通过反射获取 annotation 的信息并执行相应的操作。

1．系统内建的 Annotation

在 JDK1.5 之后，系统内建了 3 个 annotation，它们都是用做编译检查的，位于 java.lang 包下。

（1）@Override：表示重写/覆写操作，强制保证 @Override 所标记的方法必须是覆写了父类中同名的方法。

（2）@Deprecated：表示过时的、不建议使用的操作。

（3）@SuppressWarnings：表示抑制/压制警告，不让编译器发出警告信息，可同时限制多个警告。

rawtypes：对泛型类实例化时未指定类型参数发出警告。

unchecked：未经检查的操作。

deprecation：过时的操作。

all：消除所有的警告。

unused：相对闲置的代码。

（4）@Override　@Deprecated 称为 marker annotation，即名称本身给编译器提供信息。

```
@SuppressWarnings({ "rawtypes", "unchecked",deprecation" })
String[]  value1();
```

2．自定义 Annotation 的定义及使用

自定义的 annotation 一般用做代码处理。

语法：

```
修饰符   @interface annotationName {
}
```

自定义的 annotation 本质上继承于 java.lang.annotation.Annotation 接口。

1）annotation 和接口的区别

（1）annotation 是一个接口，因此成员和接口中的一致。

（2）annotation 是一个特殊的接口。

① 所有的方法必须没有参数，不能抛出异常，必须有返回值，并且返回值类型有限制。

② 可以为方法的返回值设定默认值。

③ annotation 的方法称之为 annotation 的属性。

④ 使用的时候以标记的形式使用，如@Override。

⑤ 根据保留时间的不同进行不同的处理（高级特性）。

2）annotation 的属性

语法：

```
type  attributeName() [default value];
```

其中，type 为属性类型，包括基本数据类型、String 类型、Class 类型、枚举类型、Annotation 类型及其一维数组类型。

注意：为 annotation 的属性赋值，相当于为方法设定一个返回值。

3）annotation 的使用

① 如果属性没有默认值，则使用的时候必须为属性赋值。

② 属性的默认名称为 value。

③ 如果包含多个属性，并且除了 value 属性外，其他属性都有默认值，则在使用 annotation 时可以不指定属性名称。

3．高级特性

元注解：标识注解的注解，位于 java.lang.annotation 包中。

（1）@Retention：本身也是一个元注解，主要是限制 annotation 的保留范围，由枚举类 RetentionPolicy 支持。

```
public  enum RetentionPolicy {
    SOURCE //annotation 被保留在 Java 源文件中
    CLASS //保留在编译后的 class 文档中，默认
    RUNTIME //保留在编译后的 class 文档中，并且会被 JVM 通过反射机制读取
}
@Retention(RetentionPolicy.CLASS)
public @interface MyAnnotation{
}
```

一个自定义的 annotation 要想起作用，可以通过反射机制获取该 annotation 信息，RUNTIME 的 annotation 可以通过反射机制获取。

```
java.lang.reflect.AnnotatedElement:
boolean isAnnotationPresent(Class<? extends Annotation> annotationType)
```

这段代码用于判断指定元素上是否有指定类型的 annotation 信息存在。

T　getAnnotation(Class<T> annotationType)：获取元素上指定类型的 annotation 信息。

Annotation[] getAnnotations()：获取元素上所有的 annotation 信息。

Annotation[] getDeclaredAnnotations()：获取元素上所有的声明的 annotation 的信息：

（2）@Target：用来限制 annotation 的使用范围，由枚举类型 ElementType 支持。

```
public enum  ElementType {
    ANNOTATION_TYPE,
    CONSTRUCTOR,
    FIELD,
    LOCAL_VARIABLE,
    METHOD,
    PACKAGE,
    PARAMETER,
    TYPE
}
@Target(ElementType.ANNOTATION_TYPE)
public @interface MyAnnotation{
}
```

注意：如果 annotation 要在包声明上使用，则必须位于 package-info.java 文件中。

（3）@Documented：表示被@Documented 所标识的 annotation 在生成 Javadoc 文档时也会把 annotation 的信息加入。

```
@Documented
public @interface MyAnnotation{
}
```

（4）@Inherited：表示一个 annotation 是否会被使用该 annotation 的类的子类继承。

```
@Inherited
public @interface MyAnnotation{
}
```

2.2 JDK1.6 新特性

1．Desktop 类和 SystemTray 类

在 JDK1.6 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指定的 URL，打开系统默认邮件客户端给指定的邮箱发送邮件，用默认应用程序打开或编辑文件（例如，用记事本打开以.txt 为扩展名的文件），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 对象转换为 XML 格式，反之亦然。

可以把对象与关系数据库之间的映射称为 ORM，也可以把对象与 XML 之间的映射称为 OXM（Object XML Mapping）。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 文档的。除了 JAXB 之外，还可以通过 XMLBeans 和 Castor 等来实现同样的功能。

3．StAX

StAX（JSR 173）是 JDK1.6.0 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）中有两种处理 XML 文档的方法，即 DOM（Document Object Model）和 SAX（Simple API for XML）。

JDK1.6.0 中的 JAXB2（JSR 222）和 JAX-WS 2.0（JSR 224）都会用到 StAX，因此 Sun 决定把 StAX 加入到 JAXP 族中，并将 JAXP 的版本升级到 1.4（JAXP1.4 是 JAXP1.3 的维护版本）。

API.StAX 通过提供一种基于事件迭代器的 API，来使程序员控制 XML 文档解析过程，程序遍历这个事件迭代器并处理每一个解析事件。

StAX 也基于事件处理 XML 文档，但是用推模式来解析，解析器解析完整个 XML 文档后，才产生解析事件，然后交给程序处理这些事件；DOM 采用的方式是将整个 XML 文档映射到一颗内存树中，这样可以很容易地得到父节点和子节点及兄弟节点的数据，但文档很大时，会严重影响性能。

4．使用 Compiler API

现在可以用 JDK1.6 的 Compiler API 动态编译 Java 源文件，Compiler API 结合反射功能可以动态地产生 Java 代码并编译执行这些代码，有动态语言的特征。

这个特性对于某些需要用动态编译的应用程序十分有用，如 JSP Web Server，当手动修改 JSP 后，是不希望重启 Web Server 后才能看到效果的，这时可以用 Compiler API 来动态编译 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 在运行期间通过 Runtime.exe 或 ProcessBuilder 来调用 Javac 来编译代码，这种方式需要产生另一个进程去做编译工作，容易使代码依赖于特定的操作系统；Compiler API 通过一套易用的标准的 API 提供了更加丰富的方式来做动态编译，是跨平台的。

5．轻量级 HTTP Server API

JDK1.6 提供了一个简单的 HTTP Server API，据此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，没有被实现的部分可以通过扩展已有的 HTTP Server API 来实现，程序员自己实现 HttpHandler 接口，HTTP Server 会调用 HttpHandler 类的回调方法来处理客户端请求。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，HTTP Server 负责将 HttpExchange 传给 HttpHandler 以实现类的回调方法。

6．插入式注解处理 API

插入式注解处理 API（JSR 269）提供了一套标准 API 来处理 Annotations（JSR 175）。实际上，JSR 269 不仅仅用来处理 Annotation，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，从而将 Java 语言的语义映射为对象。可以在 javax.lang.model 包中可以看到这些类，也可以利用 JSR 269 提供的 API 来构建一个功能丰富的元编程环境。

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，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，则会再次调用 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”，因此，整个 Annotation processing 过程可以看做一个 round 的序列。

JSR 269 主要被设计为针对 Tools 或者容器的 API。如果想建立一套基于 Annotation 的单元测试框架（如 TestNG），则在测试类中可以用 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

JDK1.6 中提供了 java.io.Console 类，此类专用来访问基于字符的控制台设备。若程序要与 Windows 下的 CMD 或者 Linux 下的 Terminal 交互，则可以用 Console 类代替。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（如 Windows 的 CMD）中启动的，并且输入输出没有重定向到其他的地方，则可以得到一个可用的 Console 实例。

8．对脚本语言的支持

JDK1.6 加入了对 Script（JSR 223）的支持。这是一个脚本框架，提供了使用脚本语言访问 Java 内部的方法。可以在运行的时候找到脚本引擎，然后调用这个引擎来执行脚本。这个脚本 API 允许用户为脚本语言提供 Java 支持。另外，Web Scripting Framework 允许脚本代码在任何的 Servlet 容器（如 Tomcat）中生成 Web 内容。

举例：

示例 1：演示脚本语言如何使用 JDK 平台下的类，调用一个 JDK 平台的 Swing 窗口。

```
    private static void testUsingJDKClasses(ScriptEngine engine)throws Exception {
        //Packages 是脚本语言里的一个全局变量,专用于访问 JDK 的 package
        String js = "function doSwing(t){
        var f=new Packages.javax.swing.JFrame(t);f.setSize(400,300);f.
        setVisible(true);}";
        engine.eval(js);
        //Invocable 接口: 允许 Java 平台调用脚本程序中的函数或方法
        Invocable inv = (Invocable) engine;
        /*invokeFunction()中的第一个参数就是被调用的脚本程序中的函数，第二个参数是传递
          给被调用函数的参数*/
        inv.invokeFunction("doSwing", "Scripting Swing");
    }
```

示例 2：演示脚本语言如何实现 Java 的接口，启动线程来运行 Script 提供的方法。

```
private static void testScriptInterface(ScriptEngine engine)throws ScriptException {
    String script = "var obj = new Object(); obj.run = function() {
    println('run method called'); }";
    engine.eval(script);
    Object obj = engine.get("obj");
    Invocable inv = (Invocable) engine;

    Runnable r = inv.getInterface(obj, Runnable.class);
    Thread th = new Thread(r);
    th.start();
}
```

示例 3：演示如何在 Java 中调用脚本语言的方法，通过 JDK 平台给 Script 方法中的形参赋值。

```
private static void testInvokeScriptMethod(ScriptEngine engine)throws Exception {
    String script = "function helloFunction(name) { return 'Hello everybody,' +
    name;}";
    engine.eval(script);
    Invocable inv = (Invocable) engine;
    String res = (String) inv.invokeFunction("helloFunction", "Scripting");
    System.out.println("res:" + res);
}
```

示例 4：演示如何暴露 Java 对象为脚本语言的全局变量，同时演示将 File 实例赋给脚本语言，并通过脚本语言取得它的各种属性值。

```
private static void testScriptVariables(ScriptEngine engine)throws ScriptException {
    File file = new File("test.txt");
    engine.put("f", file);
    engine.eval("println('Total Space:'+f.getTotalSpace())");
    engine.eval("println('Path:'+f.getPath())");
}
```

9．Common Annotations

Common Annotations 原本是 Java EE 5.0（JSR 244）规范的一部分，现在 Sun 将其一部分放到了 Java SE 6.0 中。

随着 Annotation 元数据功能（JSR 175）加入到 Java SE 5.0 中，很多 Java 技术（如 EJB、Web Services）都会用 Annotation 部分代替 XML 文件，以配置运行参数（或者说支持声明式编程，如 EJB 的声明式事务），如果这些技术为通用目的单独定义了自己的 Annotations，则为其他相关的 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，保证 Java SE 和 Java EE 技术的一致性。

2.3 JDK7 新特性

1．对集合类的语言支持

Java 包含对创建集合类的第一类语言支持。这意味着集合类的创建可以像 Ruby 和 Perl 那样简单。为实现此功能，原本需要如下代码：

```
List<String> list = new ArrayList<String>();
list.add("item");
```

```
String item = list.get(0);
Set<String> set = new HashSet<String>();
set.add("item");
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("key", 1);
int value = map.get("key");
```

现在编写以下代码即可：

```
List<String> list = ["item"];
String item = list[0];
Set<String> set = {"item"};
Map<String, Integer> map = {"key" : 1};
int value = map["key"];
```

2．自动资源管理

Java 中的某些资源是需要手动关闭的，如 InputStream、Writes、Sockets、Sql classes 等。这个新的语言特性允许 try 语句本身申请更多的资源，这些资源作用于 try 代码块，并自动关闭。

例如，如下代码：

```
BufferedReader br = new BufferedReader(new FileReader(path));
try {
return br.readLine();
    } finally {
         br.close();
}
```

现在可简化为

```
try (BufferedReader br = new BufferedReader(new FileReader(path)) {
   return br.readLine();
}
```

也可以定义关闭多个资源，代码如下。

```
try (
   InputStream in = new FileInputStream(src);
   OutputStream out = new FileOutputStream(dest))
{
//code
}
```

为了支持这个行为，所有可关闭的类将被修改为可以实现一个 Closable（可关闭的）的接口。

3．改进的通用实例创建类型推断

类型推断是一个特殊的烦恼，例如，下面的代码：

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

通过类型推断后变成：

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

这个<>被称为 diamond（钻石）运算符，这个运算符从引用的声明中推断类型。

4．数字字面量下画线支持

很长的数字可读性不好，在 Java 7 中可以使用下画线分隔长 int 及 long 类型的数据，例如：

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

运算时先去除下画线，如 1_1 * 10 = 110，120 - 1_0 = 110。

5．在 switch 中使用 string

以前在 switch 语句中只能使用 number 或 enum，现在可以使用 string，即：

```
String s = ...
switch(s) {
case "quux":
    processQuux(s);
//fall-through
case "foo":
case "bar":
    processFooOrBar(s);
    break;
case "baz":
    processBaz(s);
//fall-through
default:
processDefault(s);
   break;
  }
```

6．二进制字面量

由于继承于 C 语言，因此 Java 代码在传统上迫使程序员只能使用十进制、八进制或十六进制来表示数字。

由于很少有域是以 bit 为导向的，因此这种限制可能导致错误。现在可以使用 0b 前缀创建二进制字面量，即：

```
int binary = 0b1001_1001;
```

现在可以使用二进制字面量，并且使用非常简短的代码，可将二进制字符转换为数据类型，如 byte 或 short。

```
byte aByte = (byte)0b001;
short aShort = (short)0b010;
```

7．简化可变参数方法调用

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。

JDK 7 将警告从 call 转移到了方法声明的过程中。这样，API 设计者即可使用 varargs，因为警告的数量大大减少了。

2.4 JDK8 新特性

2.4.1 接口的默认方法

Java 8 允许用户为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，这个特征又称扩展方法，示例如下。