JAVA 应用与开发
泛　型

让我们愉快的 Coding 起来吧...

王晓东

中国海洋大学信息学院计算机系

October 3, 2018

学习目标

1. 泛型的概念与基本应用
   - 集合框架中的泛型
   - 泛型的向后兼容性
2. 泛型进阶
   - 类型参数 / 定义自己的泛型类
   - 类型通配符
   - 泛型方法
   - 受限制的类型参数

大纲

泛型概念

什么是泛型

泛型（Generics）

泛型机制自 JDK 5.0 开始引入，其实质是**将原本确定不变的数据类型参数化**。

作为对原有 Java 类型体系的扩充，使用泛型可以提高 Java 应用程序的类型安全、可维护性和可靠性。

什么是泛型

❖ 集合框架中的数据造型问题

传统的集合容器为了提供广泛的适用性，会将所有加入其中的元素当作 Object 类型来处理。基于此原因，在实际使用时，我们必须将从集合中取出的元素值再强制转换（造型）为所期望的类型。

什么是泛型

❖ 集合框架中的数据造型问题

传统的集合容器为了提供广泛的适用性，会将所有加入其中的元素当作 Object 类型来处理。基于此原因，在实际使用时，我们必须将从集合中取出的元素值再强制转换（造型）为所期望的类型。

☛ 无泛型机制的集合容器

```
1    Vector v = new Vector();
2    v.addElement(new Person("Tom", 18));
3    Person p = (Person) v.elementAt(0);
4    p.showInfo();
```

集合框架中的泛型

- 泛型允许编译器实施由开发者设定的附加类型约束，将类型检查从**运行时挪到编译时**进行，这样类型错误就可以在编译时暴露出来，而不是在运行时才发作（抛出 ClassCastException 运行异常）。

集合框架中的泛型

- 泛型允许编译器实施由开发者设定的附加类型约束，将类型检查从**运行时挪到编译时**进行，这样类型错误就可以在编译时暴露出来，而不是在运行时才发作（抛出 ClassCastException 运行异常）。
- 创建集合容器时规定其允许保存的元素类型，然后由编译器负责添加元素的类型合法性检查，在取用集合元素时则不必再进行造型处理。

集合框架中的泛型

❖ 在 Vector 中使用泛型

课程配套代码 ➧ sample.generics.VectorGenericsSample.java

❖ 在 Hashtable 中使用泛型

课程配套代码 ➧ sample.generics.HashtableGenericsSample.java

泛型的向后兼容性

- Java 语言中的泛型是维护向后兼容的，完全可以不采用泛型、而继续沿用过去的做法。
- 这些未加改造的旧式代码将无法使用泛型带来的便利和安全性。

未启用泛型机制的代码在高版本编译器中会输出如下形式的编译提示信息：

output

```
注：VectorGenericsSample.java 使用了未经检查或不安全的操作。
注：有关详细信息，请使用 -Xlint:unchecked 重新编译。
```

泛型的向后兼容性

- Java 语言中的泛型是维护向后兼容的，完全可以不采用泛型、而继续沿用过去的做法。
- 这些未加改造的旧式代码将无法使用泛型带来的便利和安全性。

未启用泛型机制的代码在高版本编译器中会输出如下形式的编译提示信息：

output

```
注：VectorGenericsSample.java 使用了未经检查或不安全的操作。
注：有关详细信息，请使用 -Xlint:unchecked 重新编译。
```

可以使用 SuppressWarnings 注解关闭编译提示或警告信息：

```
1  @SuppressWarnings({"unchecked"})
2  public class VectorGenericsSample {
3    ...
4  }
```

泛型类与泛型方法

类型参数

类型参数和泛型类

形如 Vector<String>，其中，尖括号括起来的部分被称为**类型参数**，而这种由类型参数修饰的类型则被称为**泛型类**。

❖ 注意

应在声明泛型类变量和创建对象时均给出类型参数，且两者应保持一致。

类型参数

使用类型参数 *E* 进行泛型化处理的 java.util.Vector 类的定义代码摘要如下：

```
1    public class Vector<E> --- {
2      public void addElement(E obj) { --- }
3      public E elementAt(int index) { --- }
4    }
```

类型参数

使用类型参数 *E* 进行泛型化处理的 java.util.Vector 类的定义代码摘要如下：

```
1    public class Vector<E> --- {
2      public void addElement(E obj) { --- }
3      public E elementAt(int index) { --- }
4    }
```

这里的 *E* 也称为“形式类型”参数。在实际使用该泛型类时，我们需要指定相应的具体类型，即实际类型参数。

```
1    Vector<String> v = new Vector<String>();
```

类型参数

使用类型参数 *E* 进行泛型化处理的 java.util.Vector 类的定义代码摘要如下：

```
1  public class Vector<E> --- {
2    public void addElement(E obj) { --- }
3    public E elementAt(int index) { --- }
4  }
```

这里的 *E* 也称为“形式类型”参数。在实际使用该泛型类时，我们需要指定相应的具体类型，即实际类型参数。

```
1  Vector<String> v = new Vector<String>();
```

编译器遇到 Vector<String> 类型变量时，即知道此 Vector 变量/对象的类型参数 *E* 已经被绑定为 String 类型，进而也就确定其 addElement() 方法的参数和 elementAt() 方法的返回值均为 String 类型。

定义泛型类

课程配套代码 ➧ package sample.generics.userdefined

代码示例中的泛型类 PersonG 可以在使用时通过类型参数 T 指定其属性 secrecy 的具体类型（以及该属性相应存/取方法的参数和返回值类型），进而提供了通用的信息存储能力。

定义泛型类

课程配套代码 ▸ package sample.generics.userdefined

代码示例中的泛型类 PersonG 可以在使用时通过类型参数 T 指定其属性 secrecy 的具体类型（以及该属性相应存/取方法的参数和返回值类型），进而提供了通用的信息存储能力。

❖ 形式类型参数的编程惯例

符号	意义
K	键，比如映射的键
V	值，比如 List 和 Set 的内容，或者 Map 中的值
E	元素，比如 Vector<E>
T	泛型

定义泛型类

❖ 使用受限制的类型参数

泛型机制允许开发者对类型参数进行附加约束。

```
1   import java.utils.Number;
2
3   public class Point<T extends Number> {
4     private T x;
5     private T y;
6     public Point() {}
7     public Point(T x, T y) {
8       this.x = x;
9       this.y = y;
10    }
11    ...
12  }
```

定义泛型类

❖ 使用受限制的类型参数

泛型机制允许开发者对类型参数进行附加约束。

```
1   import java.utils.Number;
2
3   public class Point<T extends Number> {
4     private T x;
5     private T y;
6     public Point() {}
7     public Point(T x, T y) {
8       this.x = x;
9       this.y = y;
10    }
11    ...
12  }
```

类型参数不能为基本数据类型，而 java.lang.Number 是所有数值型封装类（如 Integer、Float、Double 等）的父类型，于是限制泛型类 Point 的类型参数必须为 Number 或其子类类型，并使用 extends 关键字来标明这种继承层次上限制。

定义泛型方法

与泛型类的情况类似，**方法也可以被泛型化，且无论其所属的类是否为泛型类。**

CODE ▸ 泛型方法示例

```
1  public class Tool {
2    public <T>T evaluate(T a, T b) {
3      if(a.equals(b))
4      return a;
5      else
6      return null;
7    }
8  }
```

定义泛型方法

与泛型类的情况类似，**方法也可以被泛型化，且无论其所属的类是否为泛型类。**

CODE ◗ 泛型方法示例

```
1  public class Tool {
2    public <T>T evaluate(T a, T b) {
3      if(a.equals(b))
4      return a;
5      else
6      return null;
7    }
8  }
```

- 上述代码中方法 evaluate() 声明中的“<T>”用于标明这是一个**泛型方法**。
- 类型 T 是可变的，不必显示的告知编译器 T 具体取何值，但出现多处（两个形参、一个返回值类型）的这些值必须都相同。

定义泛型方法

❖ 使用泛型方法，而不是定义泛型类的原则

1. 不涉及到类中的其他方法时，则应将之定义为泛型方法，因为泛型方法的类型参数是局部性的，这样可以简化其所在类型的声明和处理开销。

定义泛型方法

❖ 使用泛型方法，而不是定义泛型类的原则

1. 不涉及到类中的其他方法时，则应将之定义为泛型方法，因为泛型方法的类型参数是局部性的，这样可以简化其所在类型的声明和处理开销。
2. 要施加类型约束的方法为静态方法时，只能将之定义为泛型方法，因为静态方法不能使用其所在类的类型参数。

对泛型的理解

- 泛型类可以理解为具有广泛适用性、尚未最终定型的类型。
- Person<String> 和 Person<Double> 属于同一个类，但确是不同的类型。
- 同一个泛型类与不同的类型参数复合而成的类型间并不存在继承关系，即使是类型参数间存在继承关系时也是如此。
  如：Vector<String> 不是 Vector<Object> 的子类。

处理泛型类型

遍历泛型 Vector 集合

❖ 遍历 Vector<String> 类型集合

```
1     public void overview(Vector<String> v) {
2       for(String o: v) {
3         String.out.println(o);
4       }
5     }
```

遍历泛型 Vector 集合

❖ 遍历 Vector<String> 类型集合

```
1     public void overview(Vector<String> v) {
2       for(String o: v) {
3         String.out.println(o);
4       }
5     }
```

☛ 遍历其他类型参数的 Vector 集合元素该怎么办？

难道要定义 overview(Vector<Person> v)、
overview(Vector<Integer> v) ...
显然过于繁琐，引用泛型机制后代码的通用性似乎不如从前？

泛型类型的处理方法

❖ 可能的处理方法（不要使用）

将遍历方法的形参定义为不带任何类型参数的原型类型 Vector，但这样会破坏已有的类型安全性。

```
1     public void overview(Vector v) {
2       for(Object o: v) {
3         String.out.println(o);
4       }
5     }
```

泛型类型的处理方法

❖ 可能的处理方法（不要使用）

将遍历方法的形参定义为不带任何类型参数的原型类型 Vector，但这样会破坏已有的类型安全性。

```
1    public void overview(Vector v) {
2      for(Object o: v) {
3        String.out.println(o);
4      }
5    }
```

❖ 使用通配符

为了解决类似泛型遍历的问题，Java 泛型机制中引入了通配符“?”。

```
1    public void overview(Vector<?> v) {
2      for(Object o : v) {
3        System.out.println(o);
4      }
5    }
```

类型通配符

❖ 使用类型通配符的好处

1. Vector<?> 是任何泛型 Vector 的父类型，因此可以将 Vector<String>、Vector<Integer>、Vector<Object> 等作为实参传给 overview(Vector<?> v) 方法处理；
2. Vector<?> 类型的变量在调用方法时是受到限制的——凡是必须知道具体类型参数才能进行的操作均被禁止。

```
1 Vector<String> vs = new Vector<String>();
2 vs.add("Tom");
3 Vector<?> v = vs;
4 v.add("Billy"); //非法，编译器不知道具体类型参数
5 Object e = v.elementAt(0); //合法，允许检索元素，此时读取的元素均当作 Object 类型处理
6 System.out.println(e);
```

类型通配符

上述限制不等同于将 Vector<?> 变为“只读”，在不需要编译器确定类型参数的情况下也是可以修改集合内容的，例如：

```
1 Vector<String> vs = new Vector<String>();
2 vs.add("Tom");
3 vs.add("Billy");
4 Vector<?> v = vs;
5 v.remove(new Integer(200)); //形参为 Object 类型，运行不受影响
6 v.clear(); //不需要参数，运行不受影响
```

THE END

WANGXIAODONG@OUC.EDU.CN