Chapter 19 泛型

19.1 基本概念

泛型（Generic）指可以把类型参数化，这个能力使得我们可以定义带类型参数的泛型类、泛型接口、泛型方法，随后编译器会用**唯一的具体类型替换**它；

主要优点是在**编译时检测出错误**。泛型类或方法允许用户指定可以和这些类或方法一起工作的对象类型。如果试图使用一个不相容的对象，编译器就会检测出这个错误。

Java的泛型通过擦除法实现，和C++模板生成多个实例类不同。编译时会用类型实参代替类型形参进行严格的语法检查，然后擦除类型参数、生成所有实例类型共享的唯一原始类型。这样使得泛型代码能兼容老的使用原始类型的遗留代码。

```
// 泛型类
public class Wrapper<T>  {
    // do something
}

Wrapper<String>  stringWrapper = new Wrapper<String>();
Wrapper<Circle>  circleWrapper = new Wrapper<Circle>();
```

T是一个类型变量，它可以是Java中的任何**引用类型**，例如String，Integer，Double等。当把一个具体的类型实参传递给类型形参T时，就得到了一系列的参数化类型(Parameterized Types)，如Wrapper，Wrapper ，这些参数化类型是泛型类Wrapper的实例类型。

19.2 Class类和Class对象

类型信息是通过Class类（类名为Class的类）的对象表示的，Java利用Class对象来执行RTTI（Run-Time Type Identification运行时类型是别）。通过运行时类型信息，程序在运行时能够检查父类引用所指的对象的实际派生类型。

每个类都有一个对应的Class对象，每当编写并编译了一个类，就会产生一个Class对象，这个对象当JVM加载这个类时就产生了。

19.2.1 获取Class对象

1. 通过Class.forName方法获取

```
public class Person {
    public static void main(String[] args) {
        try {
            Class clz = Class.forName("www.learnjava.garfield.ch19.Manager");
            System.out.println(clz.getName());
            // 获取完全限定名 www.learnjava.garfield.ch19.Manager
            System.out.println(clz.getSimpleName());
            // 获取简单名     Manager

            Class superclz = clz.getSuperclass();
            // 获取直接父类
            System.out.println(superclz.getName());
```

```
            // 获取完全限定名 www.learnjava.garfield.ch19.Employee
            System.out.println(superclz.getSimpleName());
            // 获取简单名     Employee
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

class Employee extends Person{

}

class Manager extends Employee{

}
```

编译器是无法检查字符串"ch13.Manager"是否为一个正确的类的完全限定名，因此在运行时可能抛出异常，比如当不小心把类名写错了时。

2. 通过类字面量获取

```
public class Person {
    public static void main(String[] args) {
        Class clz = Manager.class;
        System.out.println(clz.getName());              // 获取完全限定名
www.learnjava.garfield.ch19.Manager
        System.out.println(clz.getSimpleName());        // 获取简单名     Manager

        Class superclz = clz.getSuperclass();           // 获取直接父类
        System.out.println(superclz.getName());         // 获取完全限定名
www.learnjava.garfield.ch19.Employee
        System.out.println(superclz.getSimpleName());   // 获取简单名     Employee

    }
}

class Employee extends Person {

}

class Manager extends Employee {

}
```

类字面常量不仅可以用于类，也可用于数组(int[].class)，接口，基本类型，如int.class

相比Class.forName方法，这种方法更安全，在编译时就会被检查，因此不需要放在Try/Catch块里(见上面的标注里说明)

Class.forName会引起类的静态初始化块的执行，T.class不会引起类的静态初始化块的执行

3. 通过对象获取

```
public class Person {
    public static void main(String[] args) {
        Object o = new Manager();
        Class clz = o.getClass()
        System.out.println(clz.getName());              // 获取完全限定名
www.learnjava.garfield.ch19.Manager
        System.out.println(clz.getSimpleName());        // 获取简单名     Manager

        Class superclz = clz.getSuperclass();           // 获取直接父类
        System.out.println(superclz.getName());         // 获取完全限定名
www.learnjava.garfield.ch19.Employee
        System.out.println(superclz.getSimpleName());   // 获取简单名     Employee

    }
}

class Employee extends Person {

}

class Manager extends Employee {

}
```

19.2.2 反射（reflection）

利用反射，我们可以在运行时动态地创建对象，调用对象的方法。

```
public class ReflectDemo {
    public static void main(String[] args) {
        Class clz = Student.class;
        Constructor[] constructors = clz.getConstructors();     // 获得构造函数
        Method[] methods = clz.getMethods();

        try {
            //实例化对象
            //1：如有缺省构造函数，调用Class对象的newInstance方法
            Student s1 = (Student)clz.newInstance();
            //2. 调用带参数的构造函数，
            //    到参数类型为String的构造函数对象，然后调用它的newInstance方法调用构造函
数，参数为“John”。等价于：
            //    Student s2 = new Student(“John”);
            //    但是是通过反射机制调用的
            Student s2 =
(Student)clz.getConstructor(String.class).newInstance("John");

            // invoke method.得到方法名为setName,参数为String的方法对象m，类型是
Method。
            // 然后通过m.invoke去调用该方法，第一个参数为对象，第二个参数是传递给被调方法的
实参。
            // 这二条语句等价于s1.setName(“Marry)，但是是通过反射去调的
            Method m = clz.getMethod("setName", String.class);
            m.invoke(s1, "Marry"); //调用s1对象的setName方法，实参"Marry"
            System.out.println(s1.toString());
            System.out.println(s2.toString());
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }


}

class Student{
    private String name;

    public Student(){

    }

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

19.3 定义泛型类和接口

19.3.1 用泛型定义栈类

```
public class GenericStack<T> {
    private ArrayList<T> list = new ArrayList<>();

    //注意泛型类的构造函数不带泛型参数，连<>都不能有
    public GenericContainer(){}

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public T peek() {
        return list.get(list.size() - 1);
```

```
    }

    public T pop() {
        return list.remove(list.size() - 1);
    }

    public void push(T t) {
        list.add(t);
    }
}
```

注意：

- `GenericStack<E>`构造函数形式是擦除参数类型后的`GenericStack()`,不是`GenericStack<>()`
- 泛型类或者泛型接口的一个实例类型，可以作为其它类的父类或者类要实现的接口，

  例如`public class Circle implements Comparable<Circle>`

19.4 泛型方法

```
public class GenericMethodDemo {

    public static void main(String[] args) {
        Integer[] integers = {1,2,3,4,5};
        GenericMethodDemo.<Integer>print(integers);
    }

    public static <T> void print(T[] arr) {
        for (T t : arr) {
            System.out.println(t.toString());
        }
    }
}
```

调用泛型方法，将实际类型放于<>之中方法名之前；也可以不显式指定实际类型，而直接给实参调用，如`print(integers); print(strings);`由编译器自动发现实际类型。

声明泛型方法，将类型参数置于返回类型之前，方法的类型参数可以作为形参类型，方法返回类型，也可以用在方法体内其他类型可以用的地方。

19.5 原始类型

没有指定具体类型实参的泛型类和泛型接口称为原始类型（raw type）。如：

`GenericStack stack = new GenericStack( );`等价于`GenericStatck<Object> stack = new GenericStack<Object>( );`

这种**不带类型参数的泛型类或泛型接口称为原始类型。**使用原始类型可以向后兼容Java的早期版本。如Comparable类型。**尽量不要用。**

```
//从JDK1.5开始，Comparable就是泛型接口Comparable<T>的原始类型(raw type)
public class Max {
    public static Comparable findMax(Comparable o1, Comparable o2){
        return (o1.compareTo(o2) > 0)?o1:o2;
    }
}
```

上例中，`Comparable o1`和`Comparable o2`都是原始类型声明，但是，**原始类型是不安全的。**`Max.findMax("Welcome",123)`；编译通过，但会引起运行时错误。安全的办法是使用泛型，现在将`findMax`方法改成泛型方法。

```
public class Max {
    public static <E extends Comparable<E>> E findMax(E o1, E o2){
        return (o1.compareTo(o2) > 0)?o1:o2;
    }
}
// E extends Comparable<E>>指定类型E必须实现Comparable接口，而且接口比较对象类型必须是E
// 注意：在指定受限的类型参数时，不管是继承父类还是实现接口，都用extends
public class Circle implements Comparable<Circle> {…}

Max.findMax(new Circle(),new Circle(10.0);
//编译上面这条语句时，编译器会自动发现findMax的类型实参为Circle，用Circle替换E
```

19.6 通配泛型

19.6.1 通配泛型

通配泛型有三种形式：

- ？，非受限通配，等价于？extends Object，注意，`GenericStack<?>`不是原始类型，`GenericStack`是原始类型
- ? extends T, 受限通配, 表示T或者T的子类，上界通配符，T定义了类型上限
- ? super T, 下限通配, 表示T或者T的父类型，下界通配符，T定义了类型下限

19.6.2 协变性问题

数组的协变性：如果类A是类B的父类，那么A[]就是B[]的父类。

```
class Fruit{}
class Apple extends Fruit{}
class Jonathan extends Apple{} //一种苹果
class Orange extends Fruit{}

//由于数组的协变性，可以把Apple[]类型的引用赋值给Friut[]类型的引用
Fruit[] fruits = new Apple[10];
fruits[0] = new Apple();
fruits[1] = new Jonathan(); // Jonathan是Apple的子类

try{
    //下面语句fruits的声明类型是Fruit因此编译通过，但运行时将Fruit转型为Apple错误
    //数组是在运行时才去判断数组元素的类型约束；
    //而泛型正好相反，在运行时，泛型的类型信息是会被擦除的，编译的时候去检查类型约束
```

```
    fruits[2] = new Fruit();//运行时抛出异常 java.lang.ArrayStoreException，这是数组协变性导致的问题
}catch(Exception e){
    System.out.println(e);
}
```

为了解决数组协变性导致的问题，Java编译器规定泛型容器（任何泛型类）没有协变性。

```
ArrayList<Fruit> list = new ArrayList<Apple>(); //编译错误
//Type mismatch: cannot convert from ArrayList<Apple> to ArrayList<Fruit>
```

我们在谈论容器的类型，而不是容器持有对象的类型

A是B父类型，但泛型类(比如容器），`ArrayList<A>`不是`ArrayList <B>`的父类型。因此，上面语句报错。

为什么数组有协变性而泛型没有协变性：

数组具有协变性是因此在运行时才去判断数组元素的类型约束（前一页PPT例子），这将导致有时发生运行时错误，抛出异常 java.lang.ArrayStoreException。这个功能在Java中是一个公认的"瑕疵"

泛型没有协变性：泛型设计者认为**与其在运行失败，不如在编译时就失败**（禁止泛型的协变性就是为了杜绝数组协变性带来的问题，即如果泛型有协变性，面临可协变的数组一样的问题）——静态类型语言（Java,C++）的全部意义在于代码运行前找出错误。Python, JavaScript之类的语言是动态类型语言。

但有时希望像数组一样，一个父类型容器引用变量指向子类型容器，这时要使用**通配符**

- **采用上界通配泛型？ extends**

```
ArrayList<? extends Fruit> list = new  ArrayList<Apple>(); //左边类型是右边类型的父类型
```

- **上面语句编译通过，但是这样的list不能加入任何东西。下面语句都会编译出错**

```
list.add(new Apple()); list.add(new Fruit()); //编译都报错
//可加入null
list.add(null);

//但是从这个list取对象没有问题，编译时都解释成Fruit，运行时可以是具体的类型如Apple（有多态性）
Fruit f = list.get(0);
```

- **因为ArrayList<? extends Fruit>意味着该list集合中存放的都是Fruit的子类型（包括Fruit自身），Fruit的子类型可能有很多，但list只能存放其中的某一种类型。编译器只能知道元素类型的上限是Fruit，而无法知道list引用会指向什么具体的ArrayList，可以是ArrayList<Apple>,也可能是ArrayList<Jonathan>，为了安全，Java泛型只能将其设计成不能添加元素。**
- **虽然不能添加元素，但从里面获取元素的类型都是Fruit类型（编译时）**
- **因此带<? extends>类型通配符的泛型类不能往里存内容（不能set），只能读取（只能get）**
- **那这样声明的容器类型有什么意义？它的意义是作为一个只读（只从里面取对象）的容器**

- **采用下界通配泛型？ super**

```
//采用下界通配符 ? super T 的泛型类引用，可以指向所有以T及其父类型为类型参数的实例类型
ArrayList<? super Fruit> list = new ArrayList<Fruit>(); //这时new后边的Fruit可以省略
ArrayList<? super Fruit> list = new ArrayList<Object>(); //允许，Object是Fruit父类
ArrayList<? super Fruit> list = new ArrayList<Apple>(); //但是不能指向Fruit子类的容器
```

- **可以向list里面添加T及T的子类对象**

```
list.add(new Fruit());        //OK
list.add(new Apple());        //OK
list.add(new Jonathan());     //OK
list.add(new Orange());       //OK
//list.add(new Object()); //添加Fruit父类则编译器禁止，报错
```

- **但是从list里get数据只能被编译器解释成Object**

```
Object o1 = list.get(0); //OK
Fruit o2 = list.get(0);  //报错，Object不能赋给Fruit，需要强制类型转换，
```

- **因此这种泛型类和采用？extends的泛型类正好相反：只能存数据，获取数据至少部分失效（编译器解释成Object）**

- **？extends 和？ super的理解**

```
//现在看看通配泛型 ？ extends，注意右边的new ArrayList的类型参数必须是Fruit的子类型
//? extends Fruit指定了类型上限，因此下面的都成立：
ArrayList<? extends Fruit> list1 = new ArrayList<Fruit>(); //=号右边，如果是Fruit，可以不写，等价于new ArrayList<>();
ArrayList<? extends Fruit> list2 = new ArrayList<Apple>();  //=号右边，如果是Fruit的子类，则必须写
ArrayList<? extends Fruit> list3 = new ArrayList<Jonathan>(); //=号右边，如果是Fruit的子类，则必须写
ArrayList<? extends Fruit> list4 = new ArrayList<Orange>();  //=号右边，如果是Fruit的子类，则必须写

ArrayList<? extends Fruit> list可指向ArrayList<Fruit>|ArrayList<Apple>|ArrayList<Jonathan>| ArrayList<Orange>|…
一个ArrayList<Fruit>容器可以加入Fruit、Apple、Jonathan、Orange，
一个ArrayList<Apple>容器可以加入Apple、Jonathan，
一个ArrayList<Orange>容器可以加入Orange，
假如当ArrayList<? extends Fruit> list为方法形参时，如果方法内部调list.add，
由于编译时，编译器无法知道ArrayList<? extends Fruit>类型的引用变量会指向哪一个具体容器类型，编译器无法知道该怎么处理add。
例如当add的对象类型是Orange，如果list指向ArrayList<Apple>，加不进去。但如果list指向为ArrayList<Orange>，就可以加进去。
为了安全，编译器干脆禁止ArrayList<? extends Fruit>类型的list添加元素。
但从list里get元素，都解释成Fruit类型
```

- **？extends 和？ super的理解**

```
//? super Fruit指定了类型下限，因此下面二行都成立
ArrayList<? super Fruit> list1 = new ArrayList<Fruit>();          //=号右边，这时Fruit可以省略，等价于new ArrayList<>();
ArrayList<? super Fruit> list2 = new ArrayList<Object>();         //允许。=号右边，如果是Fruit的父类，必须写出类型
//ArrayList<? super Fruit> list3 = new ArrayList<Apple>();        //但是不能指向Fruit子类的容器

因此ArrayList<? super Fruit> list引用可以指向ArrayList<Fruit>|Fruit父类型的容器如ArrayList<Object>。
当ArrayList<? super Fruit> list为方法形参时,编译器知道list指向的具体容器的类型参数至少是Fruit。当向list里add对象o时，分析几种可能的情况：
1 o是Fruit及其子类类型，这里面又分二种情况
  1.1 ArrayList<? super Fruit> list实际指向ArrayList<Fruit>，可以加入
  1.2 ArrayList<? super Fruit> list实际指向ArrayList<Object>,可以加入
2 o是Fruit的父类型如Object，这里面又分二种情况
  2.1 ArrayList<? super Fruit> list实际指向ArrayList<Fruit>，这时编译器不允许加入，Object不能转型为Fruit
  2.2 ArrayList<? super Fruit> list实际指向ArrayList<Object>，可以加入
综合以上四种情况，可以看到，只要对象o的类型是Fruit及其子类型，这时将对象o加入list一定是安全的（1.1, 1.2）；
如果对象是Fruit父类型，则不允许加入最安全（因为可能出现2.1的情况）。由于? super Fruit规定了list元素类型的下限，因此取元素时编译器只能全部解释成Object

list1.add(new Fruit()); list1.add(new Apple()); list1.add(new Jonathan());//只要加入Fruit及其子类对象都OK
//list1.add(new Object()); //添加Fruit父类则编译器禁止，报错

取对象时都必须解释成Object类型。因此我们说带<? super>通配符的泛型类的get方法至少是部分失效
Object o1 = list.get(0);
//Fruit o2 = list.get(0); //报错，Object不能赋给Fruit，需要强制类型转换，但是引入泛型就是想去掉强制类型转换
```

19.6.3 使用原则

？ extends 和 ？ super 的使用原则为PECS: Producer Extends，Consumer Super

- Producer Extends：如果需要一个只读泛型类，用来Produce T，那么用 ？ extends T
- Consumer Super：如果需要一个只写泛型类，用来Consume T，那么用？ super T

19.7 泛型擦除和对泛型的限制

19.7.1 泛型擦除

泛型是用类型擦除（type erasure）方法实现的。泛型的作用就是使得编译器在编译时通过类型参数来检测代码的类型匹配性。**当编译通过，意味着代码里的类型都是匹配的。因此，所有的类型参数使命完成而全部被擦除。**因此，泛型信息(类型参数)在运行时是不可用的，这种方法使得泛型代码向后兼容使用原始代码的遗留代码。

泛型存在于编译时，当编译器认为泛型类型是安全的，就会将其转化为原始类型。这时(a)所示的源代码编译后变成(b)所示的代码。**注意在(b)里，由于list.get(0)返回的对象运行时类型一定是String，因此强制类型转换一定是安全的。**

```
ArrayList<String> list = new ArrayList<>();
list.add("Oklahoma");
String state = list.get(0);
```

a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

b)

当编译泛型类、接口和方法时，会用Object代替非受限类型参数E。`<E extends Object>`

```
public static <E> void print(E[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

a)

```
public static void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

b)

如果一个泛型的参数类型是受限的，编译器会用该受限类型来替换它。

```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

a)

```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

b)

19.7.2 泛型限制

- 不能使用new E( ); //只能想办法得到E的类型实参的Class信息，再newInstance(...)

  不能用泛型的类型参数创建实例，如： E object = new E( ); //错误

- 不能使用new E[ ]

  不能用泛型的类型参数创建数组，如： E[ ] element = new E[cpacity]; //错误

注意：泛型类型参数在运行时不可用！！new是运行是发生的，因此new 后面一定不能出现类型形参E，运行时类型参数早没了

- 强制类型转换可以用类型形参E，通过类型转换实现无法确保运行时类型转换是否成功

  `E[ ] element = (E[ ])new Object[cpacity];` 编译可通过(所谓编译通过就是指编译时uncheck，至于运行时是否出错，那是程序员自己的责任)

- **静态上下文中**不允许使用泛型的类型参数。由于泛型类的所有实例类型都共享相同的运行时类，**所以泛型类的静态变量和方法都被它的所有实例类型所共享。**因此，在静态方法、数据域或者初始化语句中，使用泛型的参数类型是非法的。

```
public class Test<E> {
  public static void m(E o1) { // Illegal
  }

  public static E o1; // Illegal

  static {
    E o2; // Illegal
  }
}
```

Test<String>和Test<Integer>这二个实例类型共享同一个运行时类型，如果静态上下文可以使用类型参数E, 会导致矛盾：
m方法的形参类型到底是String还是Integer?

- 异常类（Exception类）是不能泛型的。因为JVM在运行时捕获异常必须检查类型，但是在运行时，泛型都已经被擦除了！

泛型限制举例：

```
public class GenericOneDimensionArrayUncheck<T> {   //实现一维数组的泛型包装类。不可能实现泛型数组
    private T[] elements;  //T[]类型数组存放元素
    public GenericOneDimensionArrayUncheck(int size){
        //new Object[]强制类型转换。强制类型转换就是uncheck，就是强烈要求编译器把=右边的类型解释成T[]
        elements = (T[])new Object[size]; //注意：在运行时，elements引用变量指向的是Object[]
    }
    //这里value的类型是T，这点非常重要，保证了放进去的元素类型必须是T及子类型。否则编译报错
    public void put(T value,int index){ elements[index] = value; }
    public T get(int index){ return elements[index]; }//elements声明类型就是T[]，因此类型一致
    public T[] getElements() {return elements;} //这个方法非常危险，编译没问题
    public static void main(String[] args){
        GenericOneDimensionArrayUncheck<String> strArray = new
                    GenericOneDimensionArrayUncheck<>(10);
        strArray.put("Hello",0);
//        strArray.put(new Fruit(),0);  //不是String对象放不进去
        String s = strArray.get(0);  //strArray.get(0)返回对象的运行时类型一定是String，由put保证的
        //但是下面的语句抛出运行时异常：java.lang.ClassCastException
        //因为运行时，elements引用变量指向的是Object[]，无法转成String[]
        String[] a = strArray.getElements(); //返回内部数组，但为String[]类型
    }
}
```

```
public class GenericOneDimensionArray<T> {
    private T[] elements = null;  //T[]类型

    public GenericOneDimensionArray(Class<? extends T> clz,int size){
        elements = (T[])Array.newInstance(clz,size);
        // 这里第一个参数是Class<? extends T> clz，表示一个T类型及其子类的Class对象。
        // 通过Class对象，可以通过反射创建运行时类型为T[]的数组。
        // 但是Array.newInstance方法返回的是Object，因此需要强制类型转换。
        // 但这里的强制类型转换是安全的，因为创建的数组的运行时类型就是T[]
        // Array.newInstance(数组元素类型的Class对象, size)
        // 通过反射机制创建运行时类型为T[]的数组


    }

    //get, put等其他方法省略

    public T[] getElements(){ return elements; }

    public static void main(String[] args){
        GenericOneDimensionArray<String> stringArray =
                new GenericOneDimensionArray(String.class,10);
        String[] a = stringArray.getElements();  //这里不会抛出运行时异常了
//        a[0] = new Fruit(); //不是String类型的对象，编译报错
        a[1] = "Hello";
    }
}
```

19.7.3 实现带泛型参数的对象工厂

```
public class ObjectFactory<T> {
    private Class<T> type;      // 保存要创建的对象的类型信息
    public ObjectFactory(Class<T> type) {
        this.type = type;
    }
    public T create() {
        T o = null;
        try {
            o= type.newInstance();
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
        }
        return o;
    }
}


public class TestFactory {
    public static void main(String[] args) {
        //首先创建一个负责生产Car的对象工厂，传进去需要创建对象的类的Class信息
        ObjectFactory<Car> carFactory = new ObjectFactory<Car>(Car.class);
        Car o = carFactory.create();  //由对象工厂负责产生car对象
        System.out.println(carFactory.create().toString());
```

```
    }
}

class Car {
    private String s = null;
    public Car() {
        s = "Car";
    }
    public String toString() {
        return s;
    }
}
```