Java 泛型

1 什么是泛型

从 jdk1.5 开始，Java 中开始支持泛型了。泛型是一个很有用的编程工具，给我们带来了极大的灵活性。在看了《java 核心编程》之后，我小有收获，写出来与大家分享。

所谓泛型，我的感觉就是，不用考虑对象的具体类型，就可以对对象进行一定的操作，对任何对象都能进行同样的操作。这就是灵活性之所在。但是，正是因为没有考虑对象的具体类型，因此一般情况下不可以使用对象自带的接口函数，因为不同的对象所携带的接口函数不一样，你使用了对象 A 的接口函数，万一别人将一个对象 B 传给泛型，那么程序就会出现错误，这就是泛型的局限性。所以说，泛型的最佳用途，就是用于实现容器类，实现一个通用的容器。该容器可以存储对象，也可以取出对象，而不用考虑对象的具体类型。因此，在学习泛型的时候，一定要了解这一点，你不能指望泛型是万能的，要充分考虑到泛型的局限性。下面我们来探讨一下泛型的原理以及高级应用。首先给出一个泛型类：

```
public class Pair<T>
{
    public Pair() { first = null; second = null; }
    public Pair(T first, T second) { this.first = first;  this.second = second; }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }

    private T first;
    private T second;
}
```

我们看到，上述 Pair 类是一个容器类（我会多次强调，泛型天生就是为了容器类的方便实现），容纳了 2 个数据，但这 2 个数据类型是不确定的，用泛型 T 来表示。关于泛型类如何使用，那是最基本的内容，在此就不讨论了。

2 泛型类跟接口及泛型方法

2.1 泛型类跟接口及继承

2.1.1 泛型类

泛型可以继承自某一个父类，或者实现某个接口，或者同时继承父类并且实现接口

泛型也可用于匿名内部类

```
Holder3<String> hold = new Holder3<String>(){...}  //匿名内部类
```

```
public class Holder3<T> {
  private T a;
  public Holder3(T a) { this.a = a; }
  public void set(T a) { this.a = a; }
  public T get() { return a; }
  public static void main(String[] args) {
    Holder3<Automobile> h3 = new Holder3<Automobile>(new Automobile());
    Automobile a = h3.get(); // No cast needed
    // h3.set("Not an Automobile"); // Error
    // h3.set(1); // Error
  }
} ///:~
```

2.1.2 继承

泛型类也可以继承

```
Public Class A<T,A> extends Holder3<T>{}
```

```
Public Class A<T,A， B> extends Holder3<T，A>{}
```

2.1.3 接口

泛型也可以用于接口

```
Public    interface A<T>{}
```

2.2 泛型方法

2.2.1 方法

是否拥有泛型方法跟其所在的类是否是泛型没有关系。原则：无论何时只要你能做到，你就尽可能的使用泛型方法。对于 static 方法你只能使用泛型方法。

在调用泛型方法时，通常不必指明参数类型 因为编译器会为我找出具体类型，这称之为类型推断。

泛型方法也能跟可变参数共存: public static <T> void f(T...arg){}

```
public class GenericMethods {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f(1.0F);
    gm.f('c');
    gm.f(gm);
  }
} /* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
*///:~
```

2.2.2 类型推断

```
public class New {
  public static <K,V> Map<K,V> map() {
    return new HashMap<K,V>();
  }
  public static <T> List<T> list() {
    return new ArrayList<T>();
  }
  public static <T> LinkedList<T> lList() {
    return new LinkedList<T>();
  }
  public static <T> Set<T> set() {
    return new HashSet<T>();
  }
  public static <T> Queue<T> queue() {
    return new LinkedList<T>();
  }
  // Examples:
```

```
  public static void main(String[] args) {
    Map<String, List<String>> sls = New.map();
    List<String> ls = New.list();
    LinkedList<String> lls = New.lList();
    Set<String> ss = New.set();
    Queue<String> qs = New.queue();
  }
} ///:~
```

编译器能根据Map<String, List<String>>推断出New.*map*()返回的类型

类型推断只能对赋值语句有效：f(New.map())这句不能被编译

```
static void f(Map<Person, List<? extends Pet>> petPeople) {}
  public static void main(String[] args) {
    // f(New.map()); // Does not compile
 }
```

在范型方法中你可以显示指明类型，要指明类型必须在 点操作符与方法名之间插入尖括号，然后把类型置于尖括号中，如果是在类中的方法中使用必须加this.<type> 如果是static方法需要加上类名。f(New.< Person, List<? extends Pet> >map());

3 泛型实现原理

下面我们来讨论一下 Java 中泛型类的实现原理。在 java 中，泛型是在编译器中实现的，而不是在虚拟机中实现的，虚拟机对泛型一无所知。因此，编译器一定要把泛型类修改为普通类，才能够在虚拟机中执行。在 java 中，这种技术称之为“擦除”，也就是用 Object 类型替换泛型。上述代码经过擦除后就变成如下形式：

```
public class Pair
{
  public Pair(Object first, Object second)
  {
    this.first = first;
    this.second = second;
  }

  public Object getFirst() { return first; }
  public Object getSecond() { return second; }
```

```
    public void setFirst(Object newValue) { first = newValue; }
    public void setSecond(Object newValue) { second = newValue; }

    private Object first;
    private Object second;
}
```

大家可以看到，这是一个普通类，所有的泛型都被替换为 Object 类型，他被称之为原生类。每当你用一个具体类去实例化该泛型时，编译器都会在原生类的基础上，通过强制约束和在需要的地方添加强制转换代码来满足需求，但是不会生成更多的具体的类（这一点和 c++完全不同）。我们来举例说明这一点：

```
Pair<Employee> buddies = new Pair<Employee>();

//在上述原生代码中，此处参数类型是 Object，理论上可以接纳各种类型，但编译器通过强制约束
//你只能在此使用 Employee（及子类）类型的参数，其他类型编译器一律报错
buddies.setFirst(new Employee("张三"));

//在上述原生代码中，getFirst()的返回值是一个 Object 类型，是不可以直接赋给类型为 Employee 的 buddy 的
//但编译器在此做了手脚，添加了强制转化代码，实际代码应该是
Employee buddy = (Employee)buddies.getFirst();
//这样就合法了。但编译器做过手脚的代码你是看不到的，他是以字节码的形式完成的。
Employee buddy = buddies.getFirst();
```

4 泛型数组

你不能这样创建泛型数组 T[] array = new T[size]; 因为编译器无法确定 T 实际类型

ArrayList<String>[] array = new ArrayList<String>[size]; 这条语句同样也是不能编译的

原因是数组将将跟踪它们的实际类型，而这个类型是在数组创建时确定的。然而由于擦除泛型的类型信息被移除了，数组就无法保证 ArrayList 的实际类型.

但是你可以这样 ArrayList<String>[] array = (ArrayList<String>[]) new ArrayList[1] 编译器确保 你只能向数组中添加 ArrayList<String>() 对象而不能添加 ArrayList<Integer>等其他类型。

创建泛型数组的二种方式：

第一种：

```
public class GenericArray<T> {
  private T[] array;
  @SuppressWarnings("unchecked")
  public GenericArray(int sz) {
    array = (T[])new Object[sz];
  }
  public T get(int index) { return array[index]; }
  public T[] rep() { return array; }
  public static void main(String[] args) {
    GenericArray<Integer> gai = new GenericArray<Integer>(10);
    //! Integer[] ia = gai.rep();// This causes a ClassCastException:
    Object[] oa = gai.rep();//只能用Object数组接受 因为泛型数组在运行时只能是Object
  }
} ///:~
```

第二种：

```
public GenericArrayWithTypeToken(Class<T> type, int sz) {
    array = (T[]) Array.newInstance(type, sz);//该数组运行时类型是确切的类型T
}
```

5 边界

泛型就像字面意思表达的那样，你可以将代码运用到任何类型 既然是任何类型那么你就不能在类型上调用任何方法。但是 Object 中的方法是可以的，因为编译器知道这个类型至少是一个 Object，任何类就直接或者间接继承了 Object 类。如果什么方法都不能调那么你能用来干什么能，所以如果你想要调用方法那么你就需要边界就像下面 <T extends HasColor>这表示 T 只要是 HasColor 和任何它的子类都可以。那么现在你可以调用 getColor()方法了，因为编译器知道 T 至少是 HasColor 类型，擦除也会擦除到边界这里是 HasColor 如果没有设定边界那么默认会擦除到 Object。如何你想要限定多个边界那么你只需在后面加上&符号即可。<T extends HasColor & Ojbect>

```
interface HasColor { java.awt.Color getColor(); }

class Colored<T extends HasColor> {
  T item;
  Colored(T item) { this.item = item; }
  T getItem() { return item; }
  // The bound allows you to call a method:
  java.awt.Color color() { return item.getColor(); }
```

}

6 通配符

数组的协变：如果 B 是 A 的子类那么 A[] = new B[size] 是可以的。

Number[] num = new Integer[size] 那么如果向 num 数组中加入 Double 类型呢？（既然是 Number 类型那么你就没有理由阻止放入 Number 类型）你向 Integer 数组中加入 Double 这肯定导致错误，数组是 Java 内置类型他能在编译期跟运行进行检查错误。所以我们无需担心。

那么我们来看看这条语句 ArrayList<Number> arr = new ArrayList<Integer>();

这条语句不能编译，原因是因为泛型不能协变，如果你允许的话 那么你就可以往里面加入 Number 子类如 Double 而你引用的却是 ArrayList<Integer>类型，这违反了泛型的初衷编译期类型安全。所以如果我们要想实现向数组那样协变那么就要用到通配符，<? extends Number> 表示我不知道是什么类型为了协办我也不关心。它只要是 Number 的子类或者 Number 就行。我们知道即使你向上转型了(协变)，你也不能保证类型安全。所以泛型向上转型之后你就丢失了向其中添加任何东西的能力，即使 Ojbect 也不行.但是添加 null 是可以的。因为 null 是任何对象的引用但没有实际意义。就是你即使往 Numer 数组中添加 null 一样它并不会报错。Get()方法能够能行是编译器至少知道?至少是一个 Number

```
public static void test(ArrayList<? extends Number> arr) {
    // arr.add(new Integer(2)); 不能编译

    arr.add(null);//OK

    Number obj = arr.get(0);
}
```

如果你想向其中添加内容那么你可以使用<? Supper Number >表示我不知道是什么类型为了协办我也不关心。它只要是 Number 或者 Number 的子类就行，add()方法只能添加 Number 或者 Number 的子类。为什么现在添加内容是安全的呢？因为编译器至少

知道 Number 类型或者 Number 的父类型，所以向里添加子类是安全的 如：ArrayList<Object> 你可以添加任何类型。这当然包括 Number 或者其子类啦，但你却不能再添加其他的类型了 Number 的父类(Object)也不行，如果可以的那么就可添加非 Number 类型了。然而 Number obj = arr.get(0);不能正常运行了。因为这时编译器只知道是 Object 类型

```
    public static void test(ArrayList<? supper Number> arr) {
        arr.add(new Number());//OK
       arr.add(new Integer(2)); OK
        arr.add(null);//OK
        ！ //Number obj = arr.get(0);//无法判断是否是 Number 类型
        Object obj = arr.get(0)
}
```

7 泛型的问题及建议

7.1 问题

1 任何基本类型不能作为泛型参数

2 一个不能实现同一个泛型接口 原因：擦除

3 catch 语句不能捕获泛型异常

7.2 建议

以上就是我学习泛型的所有心得。下面再把《Java 核心编程》中列出的使用泛型时的注意事项列出来（各种操作被禁止的原因就不具体说明了），供大家参考：

```
 //1、不可以用一个本地类型（如 int  float）来替换泛型
 //2、运行时类型检查，不同类型的泛型类是等价的（Pair<String>与 Pair<Employee>是属于同一个类型 Pair），
 //    这一点要特别注意，即如果 a instanceof Pair<String>==true 的话，并不代表 a.getFirst()的返回值是一个 String 类型
```

```
//3、泛型类不可以继承 Exception 类，即泛型类不可以作为异常被抛出
//4、不可以定义泛型数组
//5、不可以用泛型构造对象，即 first = new T(); 是错误的
//6、在 static 方法中不可以使用泛型，泛型变量也不可以用 static 关键字来修饰
//7、不要在泛型类中定义 equals(T x)这类方法，因为 Object 类中也有 equals 方法，当泛型类被擦除后，这两个方法会冲突
//8、根据同一个泛型类衍生出来的多个类之间没有任何关系，不可以互相赋值
//     即 Pair<Number> p1; Pair<Integer> p2;  p1=p2; 这种赋值是错误的。
//9、若某个泛型类还有同名的非泛型类，不要混合使用，坚持使用泛型类
//     Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
//     Pair rawBuddies = managerBuddies; 这里编译器不会报错，但存在着严重的运行时错误隐患
```