1. **为什么需要泛型**

先看下面的代码 例1：

|  |
| --- |
| List list = new ArrayList();  list.add("CSDN\_SEU\_Calvin");  list.add(100);  for (int i = 0; i < list.size(); i++) {  String name = (String) list.get(i); //取出Integer时，运行时出现异常  System.out.println("name:" + name);  } |

本例向list类型集合中加入了一个字符串类型的值和一个Integer类型的值（这样是合法的，因为此时list默认的类型为Object类型）。

在循环中，由于忘记了之前添加了Integer类型的值或其他原因，运行时会出现java.lang.ClassCastException。为了解决这个问题，泛型应运而生。

因为当我们将一个对象放入集合中，集合不会记住此对象的类型，当再次从集合中取出此对象时，改对象的编译类型变成了Object类型，但其运行时类型任然为其本身类型。

那么有没有什么办法可以使集合能够记住集合内元素各类型，且能够达到只要编译时不出现问题，运行时就不会出现“java.lang.ClassCastException”异常呢？答案就是使用泛型。

1. **泛型概念**

泛型，即“参数化类型”。

提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。

参数化类型怎么理解？就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可称之为类型形参），在使用/调用时传入具体的类型（类型实参）。

1. **泛型的使用**

Java泛型编程是JDK1.5版本后引入的。泛型让编程人员能够使用类型抽象，通常用于集合里面。

例1的泛型写法：

|  |
| --- |
| public class GenericTest {  public static void main(String[] args) {  List<String> list = new ArrayList<String>(); //String是类型实参  list.add("qqyumidi");  list.add("corn");  //list.add(100); //提示编译错误  for (int i = 0; i < list.size(); i++) {  String name = list.get(i); // 2  System.out.println("name:" + name);  }  }  } |

通过List<String>，直接限定了list集合中只能含有String类型的元素，从而在上例中的第6行中，无须进行强制类型转换，因为集合能够记住其中元素的类型信息，编译器已经能够确认它是String类型了。

1. **自定义泛型接口、泛型类和泛型方法**

接口、类和方法都可使用泛型定义及使用。

是的，在具体使用时，可分为泛型接口、泛型类和泛型方法。

4.1 自定义泛型接口

|  |
| --- |
| public class Demo {  public static void main(String arsg[]) {  Info<String> obj = new InfoImp<String>("www.weixueyuan.net");  System.out.println("Length Of String: " + obj.getVar().length());  }  }  //定义泛型接口  interface Info<T> {  public T getVar();  }  //实现接口  class InfoImp<T> implements Info<T> {  private T var;  // 定义泛型构造方法  public InfoImp(T var) {  this.setVar(var);  }  public void setVar(T var) {  this.var = var;  }  public T getVar() {  return this.var;  }  } |

4.1 定义泛型类

自定义泛型接口、泛型类和泛型方法与上述Java源码中的List、ArrayList类似。

如下，我们看一个最简单的泛型类和方法定义：

|  |
| --- |
| public class GenericTest {  public static void main(String[] args) {  Box<String> name = new Box<String>("corn");  System.out.println("name:" + name.getData());  }  }  class Box<T> {  private T data;  public Box() {  }  public Box(T data) {  this.data = data;  }  public T getData() {  return data;  }  } |

传值参数由小括号包围，如 (int x, double y)，类型参数（泛型参数）由尖括号包围，多个参数由逗号分隔，如 <T> 或 <T, E>。

K 表键，V 表值，E 表异常或错误，T 表示一般意义上的数据类型

在泛型接口、泛型类和泛型方法的定义过程中，我们常见的如T、E、K、V等形式的参数常用于表示泛型形参，由于接收来自外部使用时候传入的类型实参。那么对于不同传入的类型实参，生成的相应对象实例的类型是不是一样的呢？

|  |
| --- |
| public class GenericTest {  public static void main(String[] args) {  Box<String> name = new Box<String>("corn");  Box<Integer> age = new Box<Integer>(712);  System.out.println("name class:" + name.getClass()); // com.qqyumidi.Box  System.out.println("age class:" + age.getClass()); // com.qqyumidi.Box  System.out.println(name.getClass() == age.getClass()); // true  }  } |

由此发现，使用泛型类时，虽传入了不同的泛型实参，但并没有真正意义上生成不同的类型，传入不同泛型实参的泛型类在内存上只有一个，即还是原来的最基本的类型（本实例中为Box），当然，在逻辑上可理解成多个不同的泛型类型。

究其原因，在于Java中的泛型这一概念提出的目的，导致其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦除，也就是说，成功编译过后的class文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。

总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

1. **泛型只在编译阶段有效**

看下面的代码：

|  |
| --- |
| ArrayList<String> a = new ArrayList<String>();  ArrayList b = new ArrayList();  Class c1 = a.getClass();  Class c2 = b.getClass();  System.out.println(c1 == c2); //true |

上面程序的输出结果为true。因为所有反射的操作都是在运行时的，既然为true，就证明了编译之后，程序会采取去泛型化的措施。

也就是说Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，成功编译过后的class文件中是不包含任何泛型信息的。

上述结论可通过下面反射的例子来印证：

|  |
| --- |
| ArrayList<String> a = new ArrayList<String>();  a.add("CSDN\_SEU\_Calvin");  Class c = a.getClass();  try{  Method method = c.getMethod("add",Object.class);  method.invoke(a,100);    }catch(Exception e){  e.printStackTrace();  }System.out.println(a); |

因为绕过了编译阶段也就绕过了泛型，输出结果为：

|  |
| --- |
| [CSDN\_SEU\_Calvin, 100] |

1. **泛型类和泛型方法**

如下，我们看一个泛型类和方法的使用例子，和未使用泛型的使用方法进行了对比，两者输出结果相同，在这里贴出来方便读者体会两者的差异。泛型接口的例子有兴趣可以去找一些资料，这里就不赘述了。

1. 使用泛型的情况

|  |
| --- |
| public static class FX<T> {  private T ob; // 定义泛型成员变量    public FX(T ob) {  this.ob = ob;  }    public T getOb() {  return ob;  }    public void showType() {  System.out.println("T的实际类型是: " + ob.getClass().getName());  }  }  public static void main(String[] args) {  FX<Integer> intOb = new FX<Integer>(100);  intOb.showType();  System.out.println("value= " + intOb.getOb());  System.out.println("----------------------------------");    FX<String> strOb = new FX<String>("CSDN\_SEU\_Calvin");  strOb.showType();  System.out.println("value= " + strOb.getOb());  } |

1. 不使用泛型的情况

|  |
| --- |
| public static class FX {  private Object ob; // 定义泛型成员变量    public FX(Object ob) {  this.ob = ob;  }    public Object getOb() {  return ob;  }    public void showType() {  System.out.println("T的实际类型是: " + ob.getClass().getName());  }  }    public static void main(String[] args) {  FX intOb = new FX(new Integer(100));  intOb.showType();  System.out.println("value= " + intOb.getOb());  System.out.println("----------------------------------");    FX strOb = new FX("CSDN\_SEU\_Calvin");  strOb.showType();  System.out.println("value= " + strOb.getOb());  } |

两种写法输出结果均为：

|  |
| --- |
| T的实际类型是: java.lang.Integer  value= 100  ----------------------------------  T的实际类型是: java.lang.String  value= CSDN\_SEU\_Calvin |

1. **通配符**

为了引出通配符的概念，先看如下代码：

|  |
| --- |
| List<Integer> ex\_int= new ArrayList<Integer>();  List<Number> ex\_num = ex\_int; //非法的 |

上述第2行会出现编译错误，因为Integer虽然是Number的子类，但List<Integer>不是List<Number>的子类。

假定第2行代码没有问题，那么我们可以使用语句ex\_num.add(newDouble())在一个List中装入了各种不同类型的子类，这显然是不可以的，因为我们在取出List中的对象时，就分不清楚到底该转型为Integer还是Double了。因此，需要一个在逻辑上可以用来同时表示为List<Integer>和List<Number>的父类的一个引用类型，类型通配符应运而生。在本例中表示为List<?>即可。

类型通配符一般是使用 ? 代替具体的类型实参。注意了，此处是类型实参，而不是类型形参！且Box<?>在逻辑上是Box<Integer>、Box<Number>...等所有Box<具体类型实参>的父类。

下面这个例子也说明了这一点，注释已经写的很清楚了。

|  |
| --- |
| public static void main(String[] args) {  FX<Number> ex\_num = new FX<Number>(100);  FX<Integer> ex\_int = new FX<Integer>(200);  getData(ex\_num);  getData(ex\_int);//编译错误  }    public static void getData(FX<Number> temp) { //此行若把Number换为“？”编译通过  //do something...  }    public static class FX<T> {  private T ob;  public FX(T ob) {  this.ob = ob;  }  } |

1. **上下边界**

看了下面这个上边界的例子就明白了，下界FX<? supers Number>的形式就不做过多赘述了。

|  |
| --- |
| public static void main(String[] args) {  FX<Number> ex\_num = new FX<Number>(100);  FX<Integer> ex\_int = new FX<Integer>(200);  getUpperNumberData(ex\_num);  getUpperNumberData(ex\_int);  }    public static void getUpperNumberData(FX<? extends Number> temp){  System.out.println("class type :" + temp.getClass());  }    public static class FX<T> {  private T ob;  public FX(T ob) {  this.ob = ob;  }  } |

1. **泛型的好处**

（1）类型安全。

通过知道泛型定义的变量类型限制，编译器可以更有效地提高Java程序的类型安全。

（2）消除强制类型转换。

消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。所有的强制转换都是自动和隐式的。

1. 提高性能。

|  |
| --- |
| Lits list1 = new ArrayList();  list1.add("CSDN\_SEU\_Calvin ");  String str1 = (String)list1.get(0);  List<String> list2 = new ArrayList<String>();  list2.add("CSDN\_SEU\_Calvin ");  String str2 = list2.get(0); |

对于上面的两段程序，由于泛型所有工作都在编译器中完成，javac编译出来的字节码是一样的（只是更能确保类型安全），那么何谈性能提升呢？是因为在泛型的实现中，编译器将强制类型转换插入生成的字节码中，但是更多类型信息可用于编译器这一事实，为未来版本的 JVM 的优化带来了可能。

1. **泛型使用的注意事项**

（1）泛型的类型参数只能是类类型（包括自定义类），不能是简单类型。

（2）泛型的类型参数可以有多个。

（3）不能对确切的泛型类型使用instanceof操作。如下面的操作是非法的，编译时会出错。

|  |
| --- |
| if(ex\_num instanceof FX<Number>){  } |

1. 不能创建一个确切的泛型类型的数组。下面使用Sun的一篇文档的一个例子来说明这个问题：

|  |
| --- |
| List<String>[] lsa = new List<String>[10]; // Not really allowed.  Object o = lsa;  Object[] oa = (Object[]) o;  List<Integer> li = new ArrayList<Integer>();  li.add(new Integer(3));  oa[1] = li; // Unsound, but passes run time store check  String s = lsa[1].get(0); // Run-time error: ClassCastException. |

这种情况下，由于JVM泛型的擦除机制，在运行时JVM是不知道泛型信息的，所以可以给oa[1]赋上一个ArrayList<Integer>而不会出现异常，但是在取出数据的时候却要做一次类型转换，所以就会出现ClassCastException，如果可以进行泛型数组的声明，上面说的这种情况在编译期将不会出现任何的警告和错误，只有在运行时才会出错。

下面采用通配符的方式是被允许的：

|  |
| --- |
| List<?>[] lsa = new List<?>[10]; // OK, array of unbounded wildcard type.  Object o = lsa;  Object[] oa = (Object[]) o;  List<Integer> li = new ArrayList<Integer>();  li.add(new Integer(3));  oa[1] = li; // Correct.  Integer i = (Integer) lsa[1].get(0); // OK |

9．List与List<?>

（1）List实际上也是List<Object>。List实际上表示持有任何Object类型的原生List。

（2）而List<?>表示具有某种特定类型的非原生List，只是我们不知道那种类型是什么。