[**Java总结篇系列：Java泛型**](http://www.cnblogs.com/lwbqqyumidi/p/3837629.html)

**一. 泛型概念的提出（为什么需要泛型）？**

首先，我们看下下面这段简短的代码:

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4 List list = new ArrayList();

5 list.add("qqyumidi");

6 list.add("corn");

7 list.add(100);

8

9 for (int i = 0; i < list.size(); i++) {

10 String name = (String) list.get(i); // 1

11 System.out.println("name:" + name);

12 }

13 }

14 }

[复制代码](javascript:void(0);)

定义了一个List类型的集合，先向其中加入了两个字符串类型的值，随后加入一个Integer类型的值。这是完全允许的，因为此时list默认的类型为Object类型。在之后的循环中，由于忘记了之前在list中也加入了Integer类型的值或其他编码原因，很容易出现类似于//1中的错误。因为编译阶段正常，而运行时会出现“java.lang.ClassCastException”异常。因此，导致此类错误编码过程中不易发现。

 在如上的编码过程中，我们发现主要存在两个问题：

1.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。

2.因此，//1处取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易出现“java.lang.ClassCastException”异常。

**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。**

**二.什么是泛型？**

**泛型，即“参数化类型”。一提到参数，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。那么参数化类型怎么理解呢？顾名思义，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（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。**

 看着好像有点复杂，首先我们看下上面那个例子采用泛型的写法。

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4 /\*

5 List list = new ArrayList();

6 list.add("qqyumidi");

7 list.add("corn");

8 list.add(100);

9 \*/

10

11 List<String> list = new ArrayList<String>();

12 list.add("qqyumidi");

13 list.add("corn");

14 //list.add(100); // 1 提示编译错误

15

16 for (int i = 0; i < list.size(); i++) {

17 String name = list.get(i); // 2

18 System.out.println("name:" + name);

19 }

20 }

21 }

[复制代码](javascript:void(0);)

采用泛型写法后，在//1处想加入一个Integer类型的对象时会出现编译错误，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//2处无须进行强制类型转换，因为此时，集合能够记住元素的类型信息，编译器已经能够确认它是String类型了。

结合上面的泛型定义，我们知道在List<String>中，String是类型实参，也就是说，相应的List接口中肯定含有类型形参。且get()方法的返回结果也直接是此形参类型（也就是对应的传入的类型实参）。下面就来看看List接口的的具体定义：

[复制代码](javascript:void(0);)

1 public interface List<E> extends Collection<E> {

2

3 int size();

4

5 boolean isEmpty();

6

7 boolean contains(Object o);

8

9 Iterator<E> iterator();

10

11 Object[] toArray();

12

13 <T> T[] toArray(T[] a);

14

15 boolean add(E e);

16

17 boolean remove(Object o);

18

19 boolean containsAll(Collection<?> c);

20

21 boolean addAll(Collection<? extends E> c);

22

23 boolean addAll(int index, Collection<? extends E> c);

24

25 boolean removeAll(Collection<?> c);

26

27 boolean retainAll(Collection<?> c);

28

29 void clear();

30

31 boolean equals(Object o);

32

33 int hashCode();

34

35 E get(int index);

36

37 E set(int index, E element);

38

39 void add(int index, E element);

40

41 E remove(int index);

42

43 int indexOf(Object o);

44

45 int lastIndexOf(Object o);

46

47 ListIterator<E> listIterator();

48

49 ListIterator<E> listIterator(int index);

50

51 List<E> subList(int fromIndex, int toIndex);

52 }

[复制代码](javascript:void(0);)

我们可以看到，在List接口中采用泛型化定义之后，<E>中的E表示类型形参，可以接收具体的类型实参，并且此接口定义中，凡是出现E的地方均表示相同的接受自外部的类型实参。

自然的，ArrayList作为List接口的实现类，其定义形式是：

[复制代码](javascript:void(0);)

1 public class ArrayList<E> extends AbstractList<E>

2 implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

3

4 public boolean add(E e) {

5 ensureCapacityInternal(size + 1); // Increments modCount!!

6 elementData[size++] = e;

7 return true;

8 }

9

10 public E get(int index) {

11 rangeCheck(index);

12 checkForComodification();

13 return ArrayList.this.elementData(offset + index);

14 }

15

16 //...省略掉其他具体的定义过程

17

18 }

[复制代码](javascript:void(0);)

由此，我们从源代码角度明白了为什么//1处加入Integer类型对象编译错误，且//2处get()到的类型直接就是String类型了。

**三.自定义泛型接口、泛型类和泛型方法**

从上面的内容中，大家已经明白了泛型的具体运作过程。也知道了接口、类和方法也都可以使用泛型去定义，以及相应的使用。是的，在具体使用时，可以分为泛型接口、泛型类和泛型方法。

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：

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4

5 Box<String> name = new Box<String>("corn");

6 System.out.println("name:" + name.getData());

7 }

8

9 }

10

11 class Box<T> {

12

13 private T data;

14

15 public Box() {

16

17 }

18

19 public Box(T data) {

20 this.data = data;

21 }

22

23 public T getData() {

24 return data;

25 }

26

27 }

[复制代码](javascript:void(0);)

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？**

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4

5 Box<String> name = new Box<String>("corn");

6 Box<Integer> age = new Box<Integer>(712);

7

8 System.out.println("name class:" + name.getClass()); // com.qqyumidi.Box

9 System.out.println("age class:" + age.getClass()); // com.qqyumidi.Box

10 System.out.println(name.getClass() == age.getClass()); // true

11

12 }

13

14 }

[复制代码](javascript:void(0);)

由此，我们发现，在使用泛型类时，虽然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，在逻辑上我们可以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出，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。

**对此总结成一句话：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。**

**四.类型通配符**

接着上面的结论，我们知道，Box<Number>和Box<Integer>实际上都是Box类型，现在需要继续探讨一个问题，那么在逻辑上，类似于Box<Number>和Box<Integer>是否可以看成具有父子关系的泛型类型呢？

为了弄清这个问题，我们继续看下下面这个例子:

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4

5 Box<Number> name = new Box<Number>(99);

6 Box<Integer> age = new Box<Integer>(712);

7

8 getData(name);

9

10 //The method getData(Box<Number>) in the type GenericTest is

11 //not applicable for the arguments (Box<Integer>)

12 getData(age); // 1

13

14 }

15

16 public static void getData(Box<Number> data){

17 System.out.println("data :" + data.getData());

18 }

19

20 }

[复制代码](javascript:void(0);)

我们发现，在代码//1处出现了错误提示信息：The method getData(Box<Number>) in the t ype GenericTest is not applicable for the arguments (Box<Integer>)。显然，通过提示信息，我们知道Box<Number>在逻辑上不能视为Box<Integer>的父类。那么，原因何在呢？

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4

5 Box<Integer> a = new Box<Integer>(712);

6 Box<Number> b = a; // 1

7 Box<Float> f = new Box<Float>(3.14f);

8 b.setData(f); // 2

9

10 }

11

12 public static void getData(Box<Number> data) {

13 System.out.println("data :" + data.getData());

14 }

15

16 }

17

18 class Box<T> {

19

20 private T data;

21

22 public Box() {

23

24 }

25

26 public Box(T data) {

27 setData(data);

28 }

29

30 public T getData() {

31 return data;

32 }

33

34 public void setData(T data) {

35 this.data = data;

36 }

37

38 }

[复制代码](javascript:void(0);)

这个例子中，显然//1和//2处肯定会出现错误提示的。在此我们可以使用反证法来进行说明。

假设Box<Number>在逻辑上可以视为Box<Integer>的父类，那么//1和//2处将不会有错误提示了，那么问题就出来了，通过getData()方法取出数据时到底是什么类型呢？Integer? Float? 还是Number？且由于在编程过程中的顺序不可控性，导致在必要的时候必须要进行类型判断，且进行强制类型转换。显然，这与泛型的理念矛盾，因此，**在逻辑上Box<Number>不能视为Box<Integer>的父类。**

好，那我们回过头来继续看“类型通配符”中的第一个例子，我们知道其具体的错误提示的深层次原因了。那么如何解决呢？总部能再定义一个新的函数吧。这和Java中的多态理念显然是违背的，因此，我们需要一个在逻辑上可以用来表示同时是Box<Integer>和Box<Number>的父类的一个引用类型，由此，类型通配符应运而生。

**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。由此，我们依然可以定义泛型方法，来完成此类需求。**

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4

5 Box<String> name = new Box<String>("corn");

6 Box<Integer> age = new Box<Integer>(712);

7 Box<Number> number = new Box<Number>(314);

8

9 getData(name);

10 getData(age);

11 getData(number);

12 }

13

14 public static void getData(Box<?> data) {

15 System.out.println("data :" + data.getData());

16 }

17

18 }

[复制代码](javascript:void(0);)

有时候，我们还可能听到**类型通配符上限和类型通配符下限**。具体有是怎么样的呢？

在上面的例子中，如果需要定义一个功能类似于getData()的方法，但对类型实参又有进一步的限制：只能是Number类及其子类。此时，需要用到类型通配符上限。

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4

5 Box<String> name = new Box<String>("corn");

6 Box<Integer> age = new Box<Integer>(712);

7 Box<Number> number = new Box<Number>(314);

8

9 getData(name);

10 getData(age);

11 getData(number);

12

13 //getUpperNumberData(name); // 1

14 getUpperNumberData(age); // 2

15 getUpperNumberData(number); // 3

16 }

17

18 public static void getData(Box<?> data) {

19 System.out.println("data :" + data.getData());

20 }

21

22 public static void getUpperNumberData(Box<? extends Number> data){

23 System.out.println("data :" + data.getData());

24 }

25

26 }

[复制代码](javascript:void(0);)

此时，显然，在代码//1处调用将出现错误提示，而//2 //3处调用正常。

**类型通配符上限通过形如Box<? extends Number>形式定义，相对应的，类型通配符下限为Box<? super Number>形式，其含义与类型通配符上限正好相反**，在此不作过多阐述了。

**五.话外篇**

本文中的例子主要是为了阐述泛型中的一些思想而简单举出的，并不一定有着实际的可用性。另外，一提到泛型，相信大家用到最多的就是在集合中，其实，在实际的编程过程中，自己可以使用泛型去简化开发，且能很好的保证代码质量。**并且还要注意的一点是，Java中没有所谓的泛型数组一说。**

对于泛型，最主要的还是需要理解其背后的思想和目的。