# 1. 集合

## 1.1 什么是集合

集合是存储对象的容器。集合中可以存储任意类型的变量。Java提供了多种集合类，不同集合类的数据结构不同，分别适合在不同的场景使用。

## 1.2 集合和数组

数组也用来存储数据，如果定义一个对象数组，例如Object[] objArr；那么对象数组也能存储对象。但是集合和数组有很大区别：

（1）数组长度是固定的，集合的长度是可变的。如果无法预先知道需要多少存储多少个数据，那么很难使用数组。

（2）数组一般只能存储同一类型的元素，集合可存储不同类型的元素。

（3）集合只能存储引用类型的元素。JDK 5实现了自动装箱功能，所以集合形式上也能存储基本数据类型的元素。

## 1.3 Collection接口

集合有很多共性，因此集合抽象出了一个Collection接口，下面讲的单例集合都是Collection接口的实现类。下面通过一个实现类来讲Collection接口中的常用方法。

# 2. Collection接口

为了了解Collection接口中方法，首先要创建一个集合对象，那么需要用多态方式来创建一个Collection接口的实现类对象。

先找到一个实现Collection接口的类。查看API，有如下说明：

“JDK 不提供此接口的任何直接实现：它提供更具体的子接口（如 Set 和 List）的实现。”

也就是说Collection没有直接的实现类，我们可以找Collection子接口Set或者List接口的实现类。其中，ArrayList是List接口的一个实现类，我们就用它来创建集合对象，以此介绍Collection接口中的方法。【提示，目前看API时会遇到<E>、<T>等字样，而且在IDE中会有警告，这涉及到泛型，暂时不用管它】

## 2.1 添加和删除对象方法

（1）public boolean add(Object o)：添加一个元素到集合中。此方法总是返回true，因此返回值不重要。

（2）public boolean addAll(Collection c)：将一个集合的元素全部添加到该集合中。

（3）public boolean remove(Object o)：将元素从集合中删除（如果有重复，只会删除一个）。如果集合中没有该元素，则返回false（不会出现异常）。

（4）public boolean removeAll(Collection c)：移除集合中存在于集合c中的元素，包括重复的元素。只要有一个元素被移除了就返回true。

（5）public void clear()：移除集合中所有的元素，集合变为一个空集合。

## 2.2 其他方法

（1）public boolean contains(Object o)：判断集合中是否包含指定的元素。对于引用类型，contains方法中是调用对象的equals方法来判断的，因此当集合存储自定义对象时，注意重写好自定义类的equals方法。

（2）public boolean isEmpty()：判断集合是否为空。

（3）public int size()：返回集合中元素个数。

（4）public boolean containsAll(Collection c)：判断集合中是否包含了集合c的所有元素。必须全部包含才返回true。

（5）public boolean retainAll(Collection c)：将该集合与集合c进行交集，结果保存在本集合中，如果本集合改变了就返回true，否则返回false。

（6）public Object toArray()：将集合转成Object数组。学了泛型，这个方法更好用。

（7）public String toString()：Collection重写了toString()方法，可以直接将集合以类似数组的方式输出。

下面是将自定义对象存储再集合中的例子。

（1）Student类

|  |
| --- |
| **package** com.zhang.test;  **public class** Student {  **private** String **name**;  **private int** age;   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public int** getAge() {  **return** age;  }   **public void** setAge(**int** age) {  **this**.age = age;  }  *// 构造方法* **public** Student(){}  **public** Student(String name, **int** age) {  **this**.**name** = name;  **this**.age = age;  }  *// 下面重写equals、hashCode和toString方法* **public boolean** equals(Object obj) {  **if**(!(obj **instanceof** Student)) {  **return false**;  }  **if**(obj == **this**) {  **return true**;  }  Student stu = (Student) obj;  **if**(stu.getName().equals(**this**.getName()) && stu.getAge() == **this**.getAge()) {  *// 若姓名和年龄相同，就假定他们是同一对象* **return true**;  } **else** {  **return false**;  }  }   **public int** hashCode() {  **return this**.getName().hashCode() + **this**.getAge();  }   **public** String toString() {  **return this**.getName() + **":"** + **this**.getAge();  } } |

（2）主类

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList; **import** java.util.Collection;  **public class** Demo {  **public static void** main(String[] args) {  Collection c = **new** ArrayList();  c.add(**new** Student(**"张三"**, 12));  c.add(**new** Student(**"李四"**, 13));  c.add(5); *//可以加任意类型  // 判断，底层调用的是equals方法* System.***out***.println(c.contains(**new** Student(**"张三"**, 12)));  *// 得到元素个数* System.***out***.println(c.size());  *// 输出集合，调用的是toString()方法，显示很友好，把其中自定义对象equals方法也调用了* System.***out***.println(c);  } } |

## 2.3 Iterator迭代器

迭代器用于对集合元素的遍历。迭代器以内部类的形式存在于集合类中，通过迭代器能够拿到集合中元素，也能通过迭代器来操作元素。

调用集合的iterator()方法能得到此集合的迭代器。Collection接口继承自Iterable接口，正是Iterable接口定义了iterator()方法。iterator()方法返回的是Iterator接口，该接口有如下方法以便遍历和获取元素：

（1）boolean hasNext()：判断是否还有可迭代的元素。

（2）Object next()：获取下一个元素，同时指针再向后移动。如果没有下一个元素，则抛出NoSuchElementException异常。

小知识点：如果一个方法能返回任意类型或者接受任意类型，那么返回值类型或者参数类型就可设置为Object，因为Object是所有类的基类。

因此常用上面的方法实现遍历，例子：（Student类还是上面的代码）

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList; **import** java.util.Collection; **import** java.util.Iterator;  **public class** Demo {  **public static void** main(String[] args) {  Collection c = **new** ArrayList();  c.add(**new** Student(**"张三"**, 12));  c.add(**new** Student(**"李四"**, 13));  *// 迭代* Iterator it = c.iterator();  **while**(it.hasNext()) {  *// 先拿出对象。由于存储的都是Student，直接转换* Student stu = (Student) it.next();  System.***out***.println(stu);  */\*  注意，不能这样写：  System.out.println( ((Student) it.next()).getName() );  System.out.println( ((Student) it.next()).getAge() );  这样每次循环都会执行两次next()，会导致跳过元素和出现  NoSuchElementException的异常  \*/* }  } } |

推荐在for循环中创建迭代器并迭代，这样可以尽早释放迭代器资源。代码如下：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList; **import** java.util.Collection; **import** java.util.Iterator;  **public class** Demo {  **public static void** main(String[] args) {  Collection c = **new** ArrayList();  c.add(**new** Student(**"张三"**, 12));  c.add(**new** Student(**"李四"**, 13));  *// 迭代* **for**(Iterator it = c.iterator(); it.hasNext(); ) {  Student stu = (Student) it.next();  System.***out***.println(stu.getName());  System.***out***.println(stu.getAge());  }  } } |

迭代器中还有remove()方法，可在遍历时删除元素，但是使用remove()方法之前必须先调用next()方法，否则会出现IllegalStateException异常。此外，在用迭代器迭代过程中，只能通过迭代器操作元素，不能使用集合本身提供的方法或者其他方法操作集合元素，否则会有安全隐患，会抛出并发修改异常ConcurrentModificationException。这点在List集合中还会强调和加深。

总结：上述讲的Collection方法和迭代器，在下面讲的单例集合（List和Set集合）中都能使用，因为单例集合就是Collection的实现类。

# 3. List集合

## 3.1 简介

List也是接口，最常用的实现类是ArrayList。List集合有如下特点：

（1）List集合是有序的。List集合会保留元素的存储顺序，这样取出元素时是顺序取出。

（2）List集合中元素可以重复，即一个集合能存储多个相同的元素。

## 3.2 List集合特有方法

（1）public void add(int index, Object o)：在指定索引处添加元素。

（2）public Object get(int index)：获取指定位置的元素。

（3）public Object remove(int index)：删除指定索引处的元素，并返回此元素

（4）public Object set(int index, Object o)：将指定索引处的元素修改为新值o，返回值是原本此位置的元素。若索引超出范围，则抛出IndexOutOfBoundsException。

（5）public boolean addAll(int index, Collection c)：在指定索引处添加集合中所有元素。

（6）public int indexOf(Object o)：返回集合中第一次出现该对象的索引，如果没有，就返回-1。

（7）public int lastIndexOf(Object o)：返回集合中最后一次出现该对象的索引，如果没有，就返回-1。

（8）public List subList(int fromIndex, int toIndex)：获得指定区间的子集合。不包含toIndex。

说明：上述介绍的很多方法，比如remove()、indexOf()等都要先找到集合中与目标相同的对象，调用的方法都是调用equals()方法。

List集合有get(int index)方法，这样可以结合Collection的size()方法自行实现遍历，而不需用迭代器。例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList; **import** java.util.List;  **public class** Demo {  **public static void** main(String[] args) {  List list = **new** ArrayList();  list.add(**new** Student(**"张三"**, 12));  list.add(**new** Student(**"李四"**, 13));  *// 利用get()和size()遍历List集合* **for**(**int** i = 0; i < list.size(); i++) {  Student stu = (Student) list.get(i);  System.***out***.println(stu);  }  } } |

## 3.3 List特有迭代器——ListIterator

List集合可以直接用Iterator迭代器，还能用List特有的迭代器ListIterator。通过调用List集合对象的listIterator()方法，能得到ListIterator对象。

ListIterator继承自Iterator，基本的用法和Iterator相同，只是ListIterator中增加了如下方法：

（1）public boolean hasPrevious()：判断是否有上一个元素

（2）public Object previous()：获取上一个元素，并将指针再指向前一个元素。

通过上述方法，能够实现逆向遍历List集合。当然，一般先要使用next()方法，将指针指向后面，才可能逆向遍历，因为一开始的游标（指针）都是在第一个元素的前面。

另外，ListIterator还有方法：

（1）public void add(Object o)：将指定的元素插入列表（可选操作）。该元素直接插入到 next 返回的下一个元素的前面（如果有），或者 previous 返回的下一个元素之后（如果有）；如果列表没有元素，那么新元素就成为列表中的唯一元素。新元素被插入到隐式光标前：不影响对 next 的后续调用，并且对 previous 的后续调用会返回此新元素。

借此机会着重讲“并发修改异常”。

前面提到过：在用迭代器过程中，只能通过迭代器操作元素，否则会有安全隐患，产生并发修改异常。例子：集合中存储“hello”，“world”和“java”三个字符串，要求遍历集合时，一旦遇到“world”字符串，就在集合中添加一个“c++”字符串。

错误示范：在用迭代器过程中，使用集合方法操作集合元素：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.\*;  **public class** Demo {  **public static void** main(String[] args) {  List list = **new** ArrayList();  list.add(**"hello"**);  list.add(**"world"**);  list.add(**"java"**);  *// 迭代器遍历* **for**(Iterator it = list.iterator(); it.hasNext(); ) {  String str = (String) it.next();  **if**(str.equals(**"world"**)) {  *// 用集合的add()方法添加元素* list.add(**"c++"**);  }  }  *// 最后输出* System.***out***.println(list);  } } |

运行程序后出现异常：ConcurrentModificationException。为什么出现并发修改异常呢？因为使用迭代器时，如果使用了集合来操作元素，会导致迭代器并不知道集合被修改了，而迭代器的使用是完全依赖于集合的，因此出现并发修改异常。

解决的办法就是只用迭代器来操作集合（ListIterator中有add方法），或者只用集合的方法来遍历和操作集合。下面是例子。只不过根据他们的实现不用，用ListIterator添加时，元素被添加在当前元素下面，而用集合本身的add()方法添加时，新元素被添加在集合的最后。

方法1：使用List迭代器的add()方法。

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.\*;  **public class** Demo {  **public static void** main(String[] args) {  List list = **new** ArrayList();  list.add(**"hello"**);  list.add(**"world"**);  list.add(**"java"**);  *// 迭代器遍历* **for**(ListIterator it = list.listIterator(); it.hasNext(); ) {  String str = (String) it.next();  **if**(str.equals(**"world"**)) {  it.add(**"c++"**);  }  }  *// 最后输出* System.***out***.println(list); *// [hello, world, c++, java]* } } |

方法2：使用List集合的方法来遍历和操作。

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.\*;  **public class** Demo {  **public static void** main(String[] args) {  List list = **new** ArrayList();  list.add(**"hello"**);  list.add(**"world"**);  list.add(**"java"**);  *// 用集合的size()和get()遍历，并修改* **for**(**int** i = 0; i < list.size(); i++) {  **if**(list.get(i).equals(**"world"**)) {  list.add(**"c++"**);  }  }  *// 最后输出* System.***out***.println(list); *// [hello, world, java, c++]* } } |

## 3.4 List集合常用子类

List集合常用的子类有ArrayList（前面一直用的）、LinkedList和Vector。详细解释如下：

（1）ArrayList：底层的数据结构是动态数组。特点是查询速度快（允许直接用索引来查找对应的元素）、增删慢（插入和删除操作涉及元素移动和内存操作）。ArrayList是线程不安全的类，效率较高。

（2）LinkedList：底层数据结构是用链表存储数据。链表的特点是查询慢（需要遍历节点查找），增删快（只需记住上个节点和下个节点）。LinkedList也是线程不安全的类，效率较高。LinkedList中一些特有的方法（需要针对头尾的操作）：

addFirst(E e)、addLast(E e)、getFirst()、getLast()、removeFirst()和removeLast()等。

（3）Vector：是线程安全的ArrayList。用法和ArrayList相似。说明一下，Vector中提供了一个elements()实例方法，返回的是一个Enumeration对象。此对象用法和Iterator一致，只不过方法名不一样，方法原型为：

public boolean hasMoreElement();和public Object nextElement()。

除此之外，List的实现类还有Stack，就是堆栈结构。其实List就是数据结构里面的线性表，使用时，结合数据结构的知识，就能很好的选择使用，并能迅速的了解不同结构中提供的方法的含义。因此推荐学习完C语言之后研究下数据结构和算法。

# 4. 泛型（Generic）

## 4.1 引入泛型

集合能存储不同的数据类型，但这个特点可能会出现安全问题。因为操作集合元素时，要先拿到该元素，用集合（或迭代器）的方法只能拿到Object类型元素，因为集合能存储任意类型，集合本身无法知道自己所存储的究竟是什么类型。因此我们拿到Object类型元素时再将元素强制转换成需要的类型，这时可能发生转换错误。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList;  **public class** Demo {  **public static void** main(String[] args) {  ArrayList list = **new** ArrayList();  list.add(**"hello"**);  list.add(**"world"**);  list.add(**new** Student());  *// 该集合中存储了字符串，最后一个元素存储了学生对象  // 遍历取出元素* **for**(**int** i = 0; i < list.size(); i++) {  *// 强制转换为String类型* String str = (String) list.get(i);  System.***out***.println(str);  }  } } |

运行程序会出现“ClassCastException”，即类型转换异常，这是因为集合中还存储了Student对象。这样取出元素会有安全性问题，这种问题在编译时是无法发现的，因此IDE只能提示警告。虽然能通过类型检查（如instanceof）来判断元素的类型，但是这样效率较低。处理这样的问题，一般是直接限定该集合只能存储一种特定类型的对象。

而泛型就是用来限定该集合只能存储一种特定数据类型的。

对集合使用泛型的格式：

集合类<E> 集合变量 = new 集合类<E>()；这里的<E>就是泛型，E就指定该集合能存储何种数据类型。比如：

ArrayList<String> arrayList = new ArrayList<String>()；就表示arrayList集合只能存储String类型的数据。

需要注意的是，这里的E只能写引用类型，不能写基本数据类型，比如不能写成<int>，而只能写成<Integer>。以后就常用泛型了，使用泛型后，IDE环境中也不会出现警告了。现在也能知道API中<E>等的含义了。

下面体会下泛型的使用：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList;  **public class** Demo {  **public static void** main(String[] args) {  ArrayList<Student> stuList = **new** ArrayList<Student>();  stuList.add(**new** Student(**"张三"**, 13));  stuList.add(**new** Student(**"李四"**, 14));  *// 遍历* **for**(**int** i = 0; i < stuList.size(); i++) {  *// 使用泛型无需类型转换* Student stu = stuList.get(i);  System.***out***.println(stu);  }  } } |

上面的例子中体会到，使用泛型后，集合中只能存特定类型的数据，并且取出元素时，会自动根据泛型得到对应的数据类型，无需类型转换，即如果这是用String泛型的集合，那么用get()方法得到的就会是String类型的数据。

很多地方使用泛型更加友好，比如JDK中提供的一个接口Comparable：

|  |
| --- |
| **package** java.lang; **public interface** Comparable {  **public int** compareTo(Object o); } |

在JDK 1.5之后，此接口被改成了泛型：

|  |
| --- |
| **package** java.lang; **public interface** Comparable<T> {  **public int** compareTo(T o); } |

这样的话，这里的泛型类型<T>就能被以后具体的类型所替换，避免了从特定类型转到Object类型，再从Object类型转到具体类型的过程。（学了后面的泛型方法、泛型类等后理解更深刻）

## 4.2 泛型方法

泛型方法就是在方法中使用泛型，实际也就是在方法形参和返回值中使用泛型，使形参接受指定类型的参数，然后返回该类型的数据，这样可以使方法适应不同的数据类型，而且不用像使用Object那样，进行强制转换。

可以只在形参中使用泛型，返回值可以不用泛型，而是返回特定的一种类型，但是一般不只在返回值中使用泛型，因为没有什么意义。

泛型相当于将类型当作变量处理。规范泛型的定义一般是一个大写的任意字母（实际上任意合法的变量名都是可以的）。在方法中使用泛型的格式：

public <声明泛型的变量> 返回值类型 方法名(数据类型和形参...)。尖括号中就是声明泛型的变量，声明好了后，就能在方法中使用该变量作为泛型类型了，比如：

public <T> T getModel(T t) {...}。

上面的方法就先在尖括号中声明了泛型变量T，这个T就相当于一个类型了，然后在返回值、参数和方法体中就能使用这个T类型了。方法说明此方法接受一个泛型类型T参数，方法返回值是也是类型T。这个T到底是什么类型呢，只有在调用方法时，传递参数时才知道。也就是调用方法时传递什么类型，这个T就是什么类型。

实际上泛型就是实现参数化类型，把类型当作参数一样的传递。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.Date;  **public class** Demo {  **public static void** main(String[] args) {  Student stu = *getObj*(**new** Student(**"张三"**, 12));  System.***out***.println(stu);  *// 传递什么类型进去，就返回什么类型* Date date = *getObj*(**new** Date(System.*currentTimeMillis*()));  System.***out***.println(date);  }  *// 这是一个静态的泛型方法，直接返回传递进来的参数* **public static** <E> E getObj(E e) {  **return** e;  } } |

## 4.3 泛型类

当一个类中多个方法都用到了泛型方法时，特别是在类中也需要使用泛型时，那么就可将泛型声明到类上，使用泛型类。

泛型类是在类名后声明泛型，也是在尖括号中。这时，泛型的类型就是在创建此对象时使用的泛型类型。当然，类中的静态成员不能使用类中定义的泛型，如果静态成员要使用泛型，只能独立声明泛型，因为静态成员是在类对象存在之前就存在的，不能根据创建对象时使用的类型来决定静态泛型的类型。

泛型类的格式：

public class 类名<声明泛型的变量> {

...

}

例子：

（1）GenericDemo类（泛型类）

|  |
| --- |
| **package** com.zhang.test;  *// 类上声明泛型* **public class** GenericDemo <E> {  *// 类中可以使用泛型类型* **private** E **obj**; *// 泛型成员  // get和set方法* **public void** setObj(E obj) {  **this**.**obj** = obj;  }  **public** E getObj() {  **return this**.**obj**;  }   *// 实例方法，展示其中维护的泛型类型对象* **public void** showObj() {  System.***out***.println(**this**.getObj().toString());  }   *// 如果是静态方法，同样还要定义泛型类型，不能使用实例成员* **public static** <T> **void** showArr(T[] arr) {  *// 用于展示任意类型的数组* **for**(**int** i = 0; i < arr.**length**; i++) {  System.***out***.println(arr[i]);  }  } } |

（2）主类

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.Date;  **public class** Demo {  **public static void** main(String[] args) {  *// 使用泛型类。  // 创建对象时，new后面的泛型类型可省略，只要声明时带有泛型类型即可* GenericDemo<Student> stu = **new** GenericDemo<>();  stu.setObj(**new** Student(**"张三"**, 14));  stu.showObj();  *// 使用静态方法* String[] strArr = {**"hello"**, **"world"**, **"java"**};  GenericDemo.*showArr*(strArr);  } } |

创建对象时，如果不指定泛型的具体类型，那么默认类型是Object(和创建集合对象时一样)。

### 4.3.1 泛型类的继承

如果一个类继承了泛型类，那么下面的方式：

（1）子类不使用父类泛型，那么直接：

public class SubClass extends GenericDemo {

...

}

这时在子类中调用父类的泛型方法，其类型是Object。

或者子类使用自己定义的泛型，那么语法是：

public class SubClass<T> extends GenericDemo {...}

（2）子类使用和父类一样的泛型：

public class SubClass<E> extends GenericDemo<E> {...}；

（3）子类继承父类的指定类型：

public class SubClass extends GenericDemo<String>{...}

当然这时子类也能使用自己的泛型：

public class SubClass<T> extends GenericDemo<String>{...}

## 4.4 泛型接口

接口泛型的声明也是在接口名后面的尖括号中。比如：

|  |
| --- |
| **package** com.zhang.test;  **public interface** GenericInterface <E> {  **void** show(E e); } |

那么泛型接口的实现类，可以还是用泛型，也可以处理具体的类型，比如：

（1）方法1：还是用泛型，处理任意类：

|  |
| --- |
| **package** com.zhang.test; *// 声明泛型类型的名字可以和接口的名字不一样。只要他们一致就表示处理的还是泛型* **public class** ImplDemo1<T> **implements** GenericInterface<T> {  **public void** show(T t) {  System.***out***.println(t);  } } |

然后主类创建实现类对象时指定具体类即可。

（2）方法2：处理具体的类型：

|  |
| --- |
| **package** com.zhang.test;  **public class** ImplDemo2 **implements** GenericInterface<String> {  @Override  **public void** show(String s) {  System.***out***.println(s);  } } |

当然，如果直接implements GenericInterface，那么具体的类就是Object，此外，实现类也能使用自己的泛型，即：

public class ImplDemo2<T> implements GenericInterface<String>。但是就是不能同时使用接口的泛型，又使用自己的泛型，泛型类中也是这样。

## 4.5 泛型通配符

先看一个需求：写一个方法，此方法能接受任意泛型的集合对象，然后将对象打印。

由于能接受任意类型，那么方法参数可定义为Collection<Object>，即：

|  |
| --- |
| **public static void** print(Collection<Object> c) {  **for**(Iterator<Object> it = c.iterator(); it.hasNext(); ) {  Object obj = it.next();  System.***out***.println(obj);  } } |

我们的想法是，传递一个ArrayList<String>对象或者一个ArrayList<Integer>对象进去，都能打印出集合中的元素，即能：

|  |
| --- |
| **public static void** main(String[] args) {  ArrayList<String> arr = **new** ArrayList<String>();  arr.add(**"hello"**);  *print*(arr); } |

但是print(arr)这行报错，并不能通过编译，原因是print函数已经限定了类型是Object，因此arr是不能作为参数的。

解决的办法有：1.就传递给print方法Coolection<Object>对象，但是这样可能要将原数据进行转换。2.print方法不使用泛型，即形参直接是Collection c。但是这样没有使用泛型会有警告，JDK不推荐。

而通配符?就是用于解决这个问题的，可以将?作为泛型类型，这个?就代表任意类型。如果传递进来的泛型是String，那么?就代表String，如果传递进来的泛型是Integer，那么?就代表Integer。因此最终可用这样的代码：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList; **import** java.util.Collection; **import** java.util.Iterator;  **public class** Demo {  **public static void** main(String[] args) {  ArrayList<String> arr = **new** ArrayList<String>();  arr.add(**"hello"**);  ArrayList<Integer> num = **new** ArrayList<>();  num.add(1);  num.add(23);  *print*(arr);  *print*(num);  }   **public static void** print(Collection<?> c) {  **for**(Iterator<?> it = c.iterator(); it.hasNext(); ) {  Object obj = it.next();  *// 这里还是用Object是理所当然的，因为不知道具体类型  // 使用?为了使用泛型并且不用将原来对象进行转换* System.***out***.println(obj);  }  } } |

通配符用来限定类型的范围。单单的?没有类型的限定，是没有边界的，可以用下面的方式进行限定：

<? extends E>：限定类型的上边界，即只接收E或者E的子类类型。（需要继承E或者本身）；

<? super E>：限定类型的下边界，即只接收E或者E的父类类型。（需要是E的父类或者本身）。

例如：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList; **import** java.util.Collection;  **public class** Demo {  **public static void** main(String[] args) {  *//？通配符可以是任意类型* Collection<?> e1 = **new** ArrayList<Animal>();  Collection<?> e2 = **new** ArrayList<Dog>();  *//? extends E，可以是E和E的子类* Collection<? **extends** Animal> e3 = **new** ArrayList<Animal>();  Collection<? **extends** Animal> e4 = **new** ArrayList<Dog>();  *//? super E，可以是E及其父类* Collection<? **super** Animal> e5 = **new** ArrayList<Animal>();  Collection<? **super** Animal> e6 = **new** ArrayList<Dog>();*//错误* } } *//先写两个具有继承关系的类* **class** Animal{ } **class** Dog **extends** Animal{ } |

# 5. 补充

泛型允许程序员在编写代码时，就限制集合的处理类型，从而把原来程序运行时可能发生的问题，转变为编译时问题，以此提高程序的可读性和稳定性。

注意：泛型是提供给javac编译器使用的，它用于限定集合的输入类型，让编译器在源代码级别上就阻止向集合中插入非法数据。但编译器编译完带有泛形的java程序后，生成的class文件中将不再带有泛形信息，以此使程序运行效率不受影响，这个过程称为“擦除”。

泛型的基本术语：

以ArrayList<E>为例：<>念：typeof。

ArrayList<E>中的E称为类型参数变量

ArrayList<Integer>中的Integer称为实际类型参数

整个称为ArrayList<E>泛型类型

整个ArrayList<Integer>称为参数化的类型ParameterizedType