



# 第12章 泛型

讲师：宋红康  
新浪微博：尚硅谷-宋红康



# 目录



1

为什么要有泛型

2

在集合中使用泛型

3

自定义泛型结构

4

泛型在继承上的体现

5

通配符的使用

6

泛型应用举例



## 12-1 为什么要有泛型



## 12.1 为什么要有泛型(Generic)

- 泛型：标签

- 举例：

- 中药店，每个抽屉外面贴着标签

- 超市购物架上很多瓶子，每个瓶子装的是什么，有标签

- 泛型的设计背景

集合容器类在设计阶段/声明阶段不能确定这个容器到底实际存的是什么类型的对象，所以在JDK1.5之前只能把元素类型设计为Object，JDK1.5之后使用泛型来解决。因为这个时候除了元素的类型不确定，其他的部分是确定的，例如关于这个元素如何保存，如何管理等是确定的，因此此时把元素的类型设计成一个参数，这个类型参数叫做泛型。Collection<E>，List<E>，ArrayList<E> 这个<E>就是类型参数，即泛型。





# 泛型的概念

- 所谓泛型，就是允许在定义类、接口时通过一个标识表示类中某个属性的类型或者是某个方法的返回值及参数类型。这个类型参数将在使用时（例如，继承或实现这个接口，用这个类型声明变量、创建对象时）确定（即传入实际的类型参数，也称为类型实参）。
- 从JDK1.5以后，Java引入了“参数化类型（Parameterized type）”的概念，允许我们在创建集合时再指定集合元素的类型，正如：`List<String>`，这表明该List只能保存字符串类型的对象。
- JDK1.5改写了集合框架中的全部接口和类，为这些接口、类增加了泛型支持，从而可以在声明集合变量、创建集合对象时传入类型实参。

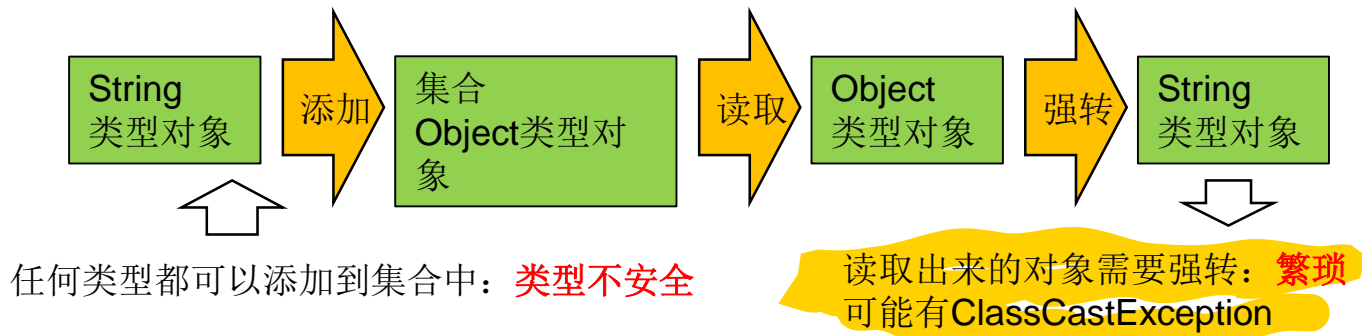


## 12.1 为什么要有泛型(Generic)

那么为什么要有泛型呢，直接Object不是也可以存储数据吗？

1. 解决元素存储的安全性问题，好比商品、药品标签，不会弄错。
2. 解决获取数据元素时，需要类型强制转换的问题，好比不用每回拿商品、药品都要辨别。

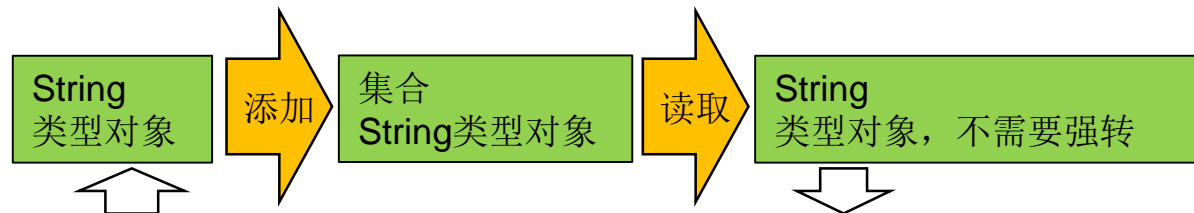
在集合中没有泛型时





## 12.1 为什么要有泛型(Generic)

在集合中有泛型时



只有指定类型才可以添加到集合中：**类型安全**    读取出来的对象不需要强转：**便捷**

Java泛型可以保证如果程序在编译时没有发出警告，运行时就不会产生ClassCastException异常。同时，代码更加简洁、健壮。





## 12-2 在集合中使用泛型



## 12.2 在集合中使用泛型

```
ArrayList<Integer> list = new ArrayList<>(); //类型推断
```

```
list.add(78);
```

```
list.add(88);
```

```
list.add(77);
```

```
list.add(66);
```

能够在编译的时候  
保证数据的安全性

//遍历方式一：

```
//for(Integer i : list){
```

```
//不需要强转
```

```
    //System.out.println(i);
```

```
//}
```

注意这里的泛型不能是基本类型，  
我们只能使用它的包装类Integer进  
行初始化

源码中的E next()代表  
next()返回的也是一个泛  
型

Iterator<Integer>能这样写的原因在于定义  
的时候就有泛型

//遍历方式二：

```
Iterator<Integer> iterator = list.iterator();
```

```
while(iterator.hasNext()){
```

```
    System.out.println(iterator.next());
```

```
}
```



## 12.2 在集合中使用泛型

```
Map<String,Integer> map = new HashMap<String,Integer>();

map.put("Tom1",34);
map.put("Tom2",44);
map.put("Tom3",33);
map.put("Tom4",32);
//添加失败
//map.put(33, "Tom");

Set<Entry<String,Integer>> entrySet = map.entrySet();

Iterator<Entry<String,Integer>> iterator = entrySet.iterator();

while(iterator.hasNext()){
    Entry<String,Integer> entry = iterator.next();
    System.out.println(entry.getKey() + "--->" + entry.getValue());
}
```



## 12-3 自定义泛型结构

- 自定义泛型类
- 自定义泛型接口
- 自定义泛型方法

`add(E e)` ---> 实例化之后: `add(Integer e)`

```

class Order<T>{
    String orderName;
    int orderId;
    //类的内部结构可以使用类的泛型
    T orderT;
    public Order(){};
    public Order(String orderName,int orderId,T orderT){
        this.orderName = orderName;
        this.orderId = orderId;
        this.orderT = orderT;
    }

    public void setOrderT(T orderT)
    {
        this.orderT = orderT;
    }
    public void setOrderId(int orderId)
    {
        this.orderId = orderId;
    }
}

```

```

class Suborder extends Order<Integer>
//不再是泛型类
{}

```

```

class Suborder1<T> extends Order<T>
//仍然是泛型类
{}

```

```

public class generic_define_by_myself {
    public static void main(String[] args) {
        //如果定义了泛型类，实例化没有指明类的泛型，则认为此泛
        //型类型为Object类型
        //要求：如果大家定义了类是带泛型的，建议在实例化时要指
        //明类的泛型
        Order order = new Order();
        order.setOrderT(123);
        order.setOrderT("ABC");
        //上面调用的为错误的,因为没有指定，泛型类型被擦除

        Order<String> order1 = new
        Order<String>("orderAA",1001,"orderAA");
        order1.setOrderT("hello");
    }
}

```



### 1. 泛型的声明



interface List<T> 和 class GenTest<K,V>

其中，T,K,V不代表值，而是表示类型。这里使用任意字母都可以。  
常用T表示，是Type的缩写。

可以查看map之中的定义

### 2. 泛型的实例化:

一定要在类名后面指定类型参数的值（类型）。如：

```
List<String> strList = new ArrayList<String>();
```

```
Iterator<Customer> iterator = customers.iterator();
```

- T只能是类，不能用基本数据类型填充。但可以使用包装类填充
- 把一个集合中的内容限制为一个特定的数据类型，这就是generics背后的核心思想

泛型的不同引用不能相互赋值

```
ArrayList<String> list1 = null;
```

```
ArrayList<Integer> list2 = new ArrayList<Integer>();
```

```
list1 = list2;//会报错，因为不同引用不能相互赋值
```



```
Comparable c = new Date();  
System.out.println(c.compareTo("red"));
```

JDK 1.5 之前



```
Comparable<Date> c = new Date();  
System.out.println(c.compareTo("red"));
```

JDK 1.5

体会：使用泛型的主要优点是能够在编译时而不是在运行时检测错误。





## 12.3 自定义泛型结构：泛型类、泛型接口

1. 泛型类可能有多个参数，此时应将多个参数一起放在尖括号内。比如：

`<E1,E2,E3>`

在实例化的时候这个位置必须要有数值：

2. 泛型类的构造器如下：`public GenericClass(){}。`

`SubOrder1<String> sub2 = new`

而下面是错误的：`public GenericClass<E>(){}。`

`SubOrder1<>();`

3. 实例化后，操作原来泛型位置的结构必须与指定的泛型类型一致。
4. 泛型不同的引用不能相互赋值。

>尽管在编译时`ArrayList<String>`和`ArrayList<Integer>`是两种类型，但是，在运行时只有一个`ArrayList`被加载到JVM中。

5. 泛型如果不指定，将被擦除，泛型对应的类型均按照`Object`处理，但不等价于`Object`。经验：泛型要使用一路都用。要不用，一路都不要用。

6. 如果泛型结构是一个接口或抽象类，则不可创建泛型类的对象。

7. jdk1.7，泛型的简化操作：`ArrayList<Fruit> flist = new ArrayList<>();`

8. 泛型的指定中不能使用基本数据类型，可以使用包装类替换。



## 12.3 自定义泛型结构：泛型类、泛型接口

```
class GenericTest {  
    public static void main(String[] args) {  
        // 1、使用时：类似于Object，不等同于Object  
        ArrayList list = new ArrayList();  
        // list.add(new Date()); // 有风险  
        list.add("hello");  
  
        test(list); // 泛型擦除，编译不会类型检查  
  
        // ArrayList<Object> list2 = new ArrayList<Object>();  
        // test(list2); // 一旦指定Object，编译会类型检查，必须按照Object处理  
    }  
  
    public static void test(ArrayList<String> list) {  
        String str = "";  
        for (String s : list) {  
            str += s + ",";  
        }  
        System.out.println("元素:" + str);  
    }  
}
```



## 12.3 自定义泛型结构：泛型类、泛型接口

9. 在类/接口上声明的泛型，在本类或本接口中即代表某种类型，可以作为非静态属性的类型、非静态方法的参数类型、非静态方法的返回值类型。但在静态方法中不能使用类的泛型。

10. 异常类不能是泛型的

11. 不能使用`new E[]`。但是可以：`E[] elements = (E[])new Object[capacity];`

参考：ArrayList源码中声明：`Object[] elementData`，而非泛型参数类型数组。

12. 父类有泛型，子类可以选择保留泛型也可以选择指定泛型类型：

- 子类不保留父类的泛型：按需实现

- 没有类型 擦除
- 具体类型

- 子类保留父类的泛型：泛型子类

- 全部保留
- 部分保留

结论：子类必须是“富二代”，子类除了指定或保留父类的泛型，还可以增加自己的泛型



## 12.3 自定义泛型结构：泛型类、泛型接口

```
class Father<T1, T2> {  
}  
// 子类不保留父类的泛型  
// 1)没有类型 擦除  
class Son1 extends Father {  
} // 等价于 class Son extends Father<Object, Object> {  
}  
// 2)具体类型  
class Son2 extends Father<Integer, String> {  
} // 子类指定父类的具体类型  
// 子类保留父类的泛型  
// 1)全部保留  
class Son3<T1, T2> extends Father<T1, T2> {  
}  
// 2)部分保留  
class Son4<T2> extends Father<Integer, T2> {  
}
```



## 12.3 自定义泛型结构：泛型类、泛型接口

```
class Father<T1, T2> {  
}  
// 子类不保留父类的泛型  
// 1)没有类型 擦除  
class Son<A, B> extends Father{//等价于class Son extends Father<Object,Object>{  
}  
  
// 2)具体类型  
class Son2<A, B> extends Father<Integer, String> {  
}  
// 子类保留父类的泛型  
// 1)全部保留  
class Son3<T1, T2, A, B> extends Father<T1, T2> {  
}  
// 2)部分保留  
class Son4<T2, A, B> extends Father<Integer, T2> {  
}
```



## 12.3 自定义泛型结构：泛型类

```
class Person<T> {  
    // 使用T类型定义变量  
    private T info;  
    // 使用T类型定义一般方法  
    public T getInfo() {  
        return info;  
    }  
    public void setInfo(T info) {  
        this.info = info;  
    }  
    // 使用T类型定义构造器  
    public Person() {  
    }  
    public Person(T info) {  
        this.info = info;  
    }  
}
```

```
// static的方法中不能声明泛型  
//public static void show(T t) {  
//  
//}  
// 不能在try-catch中使用泛型定义  
//public void test() {  
//try {  
//  
//} catch (MyException<T> ex) {  
//  
//}  
//}  
}
```



## 12.3 自定义泛型结构：泛型方法

- 方法，也可以被泛型化，不管此时定义在其中的类是不是泛型类。在泛型方法中可以定义泛型参数，此时，参数的类型就是传入数据的类型。
- 泛型方法的格式：

**[访问权限] <泛型> 返回类型 方法名([泛型标识 参数名称]) 抛出的异常**

- 泛型方法声明泛型时也可以指定上限(在12.5中讲)

```
public class DAO {  
  
    public <E> E get(int id, E e) {  
  
        E result = null;  
  
        return result;  
    }  
}
```

前面的<E>表示E为泛型参数

泛型方法：在方法中出现了泛型的结构，泛型参数与类的泛型参数没有关系。

换句话说，泛型方法所属的类是不是泛型类都没有关系



## 12.3 自定义泛型结构：泛型方法

```
public static <T> void fromArrayToCollection(T[] a, Collection<T> c) {  
    for (T o : a) {  
        c.add(o);  
    }  
}
```

```
public static void main(String[] args) {  
    Object[] ao = new Object[100];  
    Collection<Object> co = new ArrayList<Object>();  
    fromArrayToCollection(ao, co);  
  
    String[] sa = new String[20];  
    Collection<String> cs = new ArrayList<>();  
    fromArrayToCollection(sa, cs);  
  
    Collection<Double> cd = new ArrayList<>();  
    // 下面代码中T是Double类，但sa是String类型，编译错误。  
    // fromArrayToCollection(sa, cd);  
    // 下面代码中T是Object类型，sa是String类型，可以赋值成功。  
    fromArrayToCollection(sa, co);  
}
```





## 12.3 自定义泛型结构：泛型方法

```
class Creature{}
class Person extends Creature{}
class Man extends Person{}
class PersonTest {
    public static <T extends Person> void test(T t){
        System.out.println(t);
    }

    public static void main(String[] args) {
        test(new Person());
        test(new Man());
        //The method test(T) in the type PersonTest is not
        //applicable for the arguments (Creature)
        test(new Creature());
    }
}
```

```
class Order
{
    public <E> List<E> copyFromArrayToList(E[] arr)
    //泛型方法可以声明为静态的。原因：泛型参数是在调用方法时确定的，并非在实例化时确定的
    {
        ArrayList<E> list = new ArrayList<>();
        for(E e:arr){
            list.add(e);
        }
        return list;
    }
}
```

```
public void test4()
{
    Order<String> order = new Order<>();
    Integer[] arr = new Integer[]{1,2,3,4};
    //泛型方法的调用，指明泛型参数的类型
    List<Integer> list = order.copyFromArrayToList(arr);
    System.out.println(list);
}
```



## 12-4 泛型在继承上的体现



## 12.4 泛型在继承上的体现

请输出如下来两段代码有何不同

```
public void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (int k = 0; k < c.size(); k++) {  
        System.out.println(i.next());  
    }  
}
```

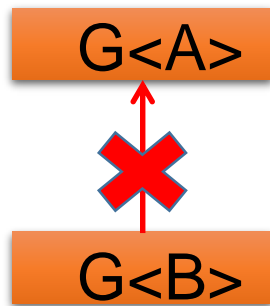
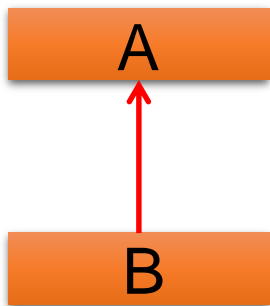
```
public void printCollection(Collection<Object> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```



## 12.4 泛型在继承上的体现

如果B是A的一个子类型（子类或者子接口），而G是具有泛型声明的类或接口， $G<B>$ 并不是 $G<A>$ 的子类型！

比如：String是Object的子类，但是 $List<String>$ 并不是 $List<Object>$ 的子类。





## 12.4 泛型在继承上的体现

```
public void testGenericAndSubClass() {  
    Person[] persons = null;  
    Man[] mans = null;  
    // 而 Person[] 是 Man[] 的父类。  
    persons = mans;  
  
    Person p = mans[0];  
  
    // 在泛型的集合上  
    List<Person> personList = null;  
    List<Man> manList = null;  
    // personList = manList;(报错)  
}
```

虽然Person和Man属于父子类关系，但是List<Person>和List<Man>不属于父子类的关系

变换一种写法就可以

```
AbstractList<String> list1 = null;
```

```
List<String> list2 = null;
```

```
ArrayList<String> list3 = null;
```

```
list1 = list3;
```

```
list2 = list3;
```

此时上面运行的代码能够正常运转，原因在于List是AbstractList的父类



## 12-5 通配符的使用



类A是类B的父类，G<A>和G<B>是没有关系的，二者共同的父类是G<?>

```
List<Object> list1 = null; List<String> list2 = null; List<?> list = null;
```

```
list=list1;
```

```
list = list2;
```

```
public void print(List<?> list)
{
    Iterator<?> iterator = list.iterator();
    while(iterator.hasNext())
    {
        Object obj = iterator.next();
        System.out.println(obj);
    }
}
```



### 1. 使用类型**通配符**：?

比如：List<?> ， Map<?,?>

List<?>是List<String>、List<Object>等各种泛型List的父类。

2. **读取**List<?>的对象list中的元素时，永远是安全的，因为不管list的真实类型是什么，它包含的都是Object。

3. **写入**list中的元素时，不行。因为我们不知道c的元素类型，我们不能向其中添加对象。

➤ 唯一的例外是null，它是所有类型的成员。



- 将任意元素加入到其中不是类型安全的：

```
Collection<?> c = new ArrayList<String>();  
c.add(new Object()); // 编译时错误
```

因为我们不知道c的元素类型，我们不能向其中添加对象。**add**方法有类型参数**E**作为集合的元素类型。我们传给**add**的任何参数都必须是一个未知类型的子类。因为我们不知道那是什么类型，所以我们无法传任何东西进去。

- 唯一的例外的是**null**，它是所有类型的成员。
- 另一方面，我们可以调用**get()**方法并使用其返回值。返回值是一个未知的类型，但是我们知道，它总是一个**Object**。



## 12.5 通配符的使用

```
public static void main(String[] args) {  
    List<?> list = null;  
    list = new ArrayList<String>();  
    list = new ArrayList<Double>();  
    // list.add(3); //编译不通过  
    list.add(null);  
  
    List<String> l1 = new ArrayList<String>();  
    List<Integer> l2 = new ArrayList<Integer>();  
    l1.add("尚硅谷");  
    l2.add(15);  
    read(l1);  
    read(l2);  
}
```

对于List<?>就不能向其内添加数据了，除了添加null之外。

对于list<?>获取：允许读取数据，读取数据为Object类型

Object o = list.get(0);

System.out.println(o);

```
public static void read(List<?> list) {  
    for (Object o : list) {  
        System.out.println(o);  
    }  
}
```



## 12.5 通配符的使用：注意点

//注意点1：编译错误：不能用在泛型方法声明上，返回值类型前面<>不能使用？

```
public static <?> void test(ArrayList<?> list){  
  
}
```

//注意点2：编译错误：不能用在泛型类的声明上

```
class GenericTypeClass<?>{  
  
}
```

//注意点3：编译错误：不能用在创建对象上，右边属于创建集合对象

```
ArrayList<?> list2 = new ArrayList<?>();
```



## 12.5 通配符的使用：有限制的通配符

- `<?>`

允许所有泛型的引用调用

- 通配符指定上限

上限**extends**：使用时指定的类型必须是继承某个类，或者实现某个接口，即`<=`

- 通配符指定下限

下限**super**：使用时指定的类型不能小于操作的类，即`>=`

- 举例：

➤ **`<? extends Number>` (无穷小, `Number`]**

只允许泛型为`Number`及`Number`子类的引用调用

➤ **`<? super Number>` [`Number`, 无穷大)**

只允许泛型为`Number`及`Number`父类的引用调用

➤ **`<? extends Comparable>`**

只允许泛型为实现`Comparable`接口的实现类的引用调用



## 12.5 通配符的使用：有限制的通配符

```
public static void printCollection3(Collection<? extends Person> coll) {  
    //Iterator只能用Iterator<?>或Iterator<? extends Person>.why?  
    Iterator<?> iterator = coll.iterator();  
    while (iterator.hasNext()) {  
        System.out.println(iterator.next());  
    }  
}  
  
public static void printCollection4(Collection<? super Person> coll) {  
    //Iterator只能用Iterator<?>或Iterator<? super Person>.why?  
    Iterator<?> iterator = coll.iterator();  
    while (iterator.hasNext()) {  
        System.out.println(iterator.next());  
    }  
}
```

```
List<? extends Person> list1 = null;  
List<? super Person> list2 = null;
```

//extends可以理解为小于等于，super可以理解为大于等于

```
List<Student> list3 = null;  
List<Person> list4 = null;  
List<Object> list5 = null;
```

```
list1 = list3;  
list1 = list4;  
list1 = list5;//这里会报错
```

```
list2 = list3;  
list2 = list4;  
list2 = list5;
```

## 有限制条件的通配符的使用

? extends A:

G<? extends A> 可以作为G<A>和G<B>的父亲，其中B是A的子类

? super A:

G<? super A> 可以作为G<A>和G<B>的父亲，其中B是A的父类





## 12.5 通配符的使用：有限制的通配符

练习题：为什么编译如下的操作会报错？

```
59
60 public static void addStrings(List<? extends Object> list) {
61
62     // list.add("aaa");
63     // list.add("aaa");
64
65     list.add(new A());
66     list.add(new Circle());
67     list.add(new GeometricObject());
68     list.add(new Object());
69 }
70
71 public void addString(List<? super A> list) {
72
73     list.add(new A());
74     list.add(new Circle());
75     list.add(new GeometricObject());
76     list.add(new Object());
77 }
```



## 12-6 泛型应用举例



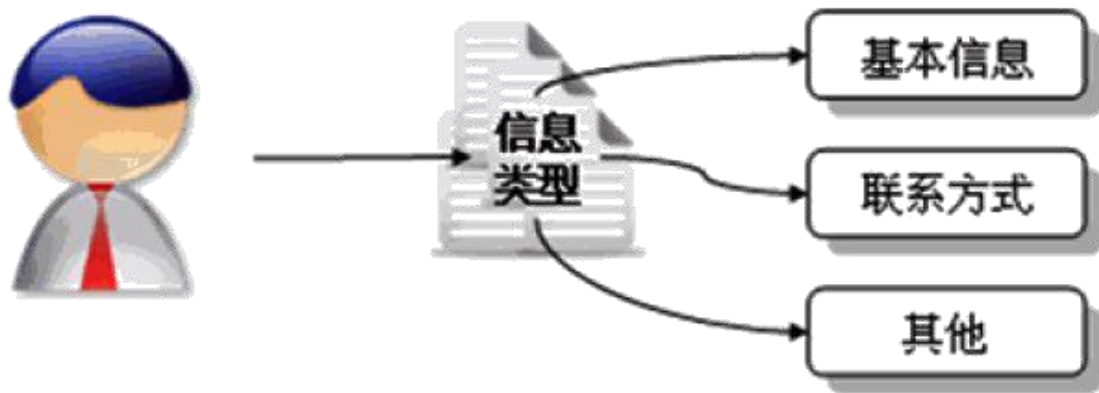
## 12.6 泛型应用举例：泛型嵌套

```
public static void main(String[] args) {  
    HashMap<String, ArrayList<Citizen>> map = new HashMap<String, ArrayList<Citizen>>();  
    ArrayList<Citizen> list = new ArrayList<Citizen>();  
    list.add(new Citizen("刘恺威"));  
    list.add(new Citizen("杨幂"));  
    list.add(new Citizen("小糯米"));  
    map.put("刘恺威", list);  
  
    Set<Entry<String, ArrayList<Citizen>>> entrySet = map.entrySet();  
    Iterator<Entry<String, ArrayList<Citizen>>> iterator = entrySet.iterator();  
    while (iterator.hasNext()) {  
        Entry<String, ArrayList<Citizen>> entry = iterator.next();  
        String key = entry.getKey();  
        ArrayList<Citizen> value = entry.getValue();  
        System.out.println("户主：" + key);  
        System.out.println("家庭成员：" + value);  
    }  
}
```



## 12.6 泛型应用举例：实际案例

用户在设计类的时候往往会使用类的关联关系，例如，一个人中可以定义一个信息的属性，但是一个人可能有各种各样的信息（如联系方式、基本信息等），所以此信息属性的类型就可以通过泛型进行声明，然后只要设计相应的信息类即可。



**GenericPerson.java**

让天下没有难学的技术



尚硅谷