# 泛型介绍

英文：Generic

什么是泛型，就是广泛的类型，如ArrayList<T>集合中的T可以接受很多的类型，如String、Integer等一切对象都可以。几乎所有的集合类都定义了泛型。

泛型是jdk1.5的所有新特性中最难深入掌握的部分，张孝详老师也不敢说所有的细节都掌握，泛型那些一堆堆的数学公式是难以看懂的，不过，我们在实际应用中没有需要掌握得那么深入，掌握泛型中一些最基本的内容就差不多了。

<>是用来传递引用数据的类型，如String类型、Date类型，又如ArrayList<String>，这就说明我们给ArrayList传递了一个引用数据类型，这个类型就是String，这告诉JDK说ArrayList里所装的引用元素的类型都是String类型的，如果你装的元素不是String类型，那么JDK在编译时就会给你报错，如果调用get方法，编译知道取出来的只能是String，所以不需要强转，使用方便。

在JDK1.5中，你还可以按原来的方式将各种不同类型的数据装到一个集合中，但是编译器会报告unckecked警告。如下：

这时我们可以使用注解的方式消除警告。

泛型是提供给javac编译器使用的，可以限定集合中的输入类型，让编译器挡住源程序中的非法输入，编译器在编译带泛型的集合时会去除掉泛型信息，使程序运行效率不受影响，对于带泛型的类，getClass()方法的返回值和原始类型完全一样，示例如下：

ArrayList<String> stringList = **new** ArrayList<String>();  
ArrayList<Integer> integerList = **new** ArrayList<Integer>();  
System.***out***.println(stringList.getClass() == ArrayList.**class**); *// true*System.***out***.println(stringList.getClass() == integerList.getClass()); *// true*

在Jdk文档中的<T>、<E>，这里的T、E只是个变量名，而它们的值是什么呢？就是引用的数据类型，如String、Date等，这些都可以是变量T、E的值。T代表Type，E代Element。其实这个字母可以随便取，只是一个变量名字而已。又如Map集合中的<K,V>，K代表key，V代表value。只是一个变量名而已。也可以用多个字母，如下：

**class** Entry<Key, Value> {  
 Key **key**;  
 Value **value**;  
  
 **public** Entry(Key key, Value value) {  
 **this**.**key** = key;  
 **this**.**value** = value;  
 }  
}

泛型的变量名一般用一个字母表示，方便理解，一看到只有一个的大写字母，就能知道这个字母代表的是一个泛型变量，如果用多个字母表示泛型变量，有时候会分不清这是一个类名还是一个泛型变量名。

# 什么时候定义泛型类

当类中要操作的引用数据类型不确定的时候，早期定义Object来完成扩展，现在定义泛型来完成扩展。

# 泛型中的一些术语

class ArrayList<E> { }和ArrayList<Integer>中涉及如下术语：

* + ArrayList<E>中的E称为类型变量或类型参数，int变量用于装int数据，类型变量则用于装类型（String、Date、Person等类型，只是类型，不是具体对象）
  + 整个ArrayList<E>称为泛型类型，因为现在ArrayList中装什么还不确定。而整个ArrayList<Interger>称为参数化类型，参数化的英语为：parameterized [pə'ræmitəraizd] adj. [数] 参数化的。参数化意思就是已经把上面说的类型变量E变成真实的参数类型 (Integer) 了
  + ArrayList<Integer>中的Integer称为类型参数的实例或实际类型参数
  + ArrayList<Integer>中的<>读为typeof，这里ArrayList<Integer>整体念为：ArrayList typeof Integer，可理解为：Integer类型的ArrayList
  + ArrayList称为原始类型

# 泛型变量的使用

可以把泛型理解成变量，只是泛型变量的值比较特殊，它的值只能是对象的类型（如String、Date、Person、Integer等）

使用普通变量分3步：

* 声明变量（可以在很多地方声明）
* 给变量赋值（可以声明的时候直接赋值，也可在其它地方赋值，可以多次赋值）
* 使用变量（除了声明和赋值，其它任何出现变量的地方都是在使用变量）  
  int i;  
  i = 3;  
  System.out.println(i);

使用泛型变量跟使用普通变量一样，也分3步：

* 声明泛型变量（只能在固定的地方声明：在类名后面声明，或者在方法的返回值前声明）
* 给泛型变量赋值（只能在固定的地方赋值，且只能赋值一次）
* 使用泛型变量（除了声明和赋值，其它任何出现泛型变量的地方都是在使用泛型变量，可以用到泛型类上，也可以用来声明变量）

# 泛型类

## 泛型类定义

一个类上可以定义多个泛型变量，用逗号分隔，如下：

**class** Util<T, K> {  
  
 **public void** print(T t) {  
 System.***out***.println(t);  
 }  
}  
  
**public class** GenericDemo {  
 **public static void** main(String[] args) {  
 Util<String, Integer> util = **new** Util<>();  
 util.print(**"haha"**);  
 }  
}

在类中定义泛型变量，这样的类叫“泛型类”，在类中定义的泛型变量，可以在整个类中使用。

类上的泛型变量不能用于静态变量和静态方法，因为使用这些静态成员时不需要创建对象，所以在使用时泛型都还没创建，如何知道泛型的类型是什么。

## 类型推导、泛型变量默认值

Java 1.7时出现了类型推导功能，所以可以这样使用泛型类：

Util<String> util = **new** Util<>();

通过前面的声明（Util<String> util）就能推断出new Util时必须是Util<String>类型的，所以在new的时候可以省略String不写。如果没有声明，则在new的时候必须写明泛型，如下代码是错误的：

甚至这样写都是可以的：

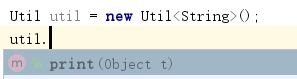
Util<String> util = **new** Util();

但是会报一个警告，如下：



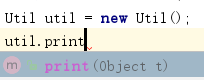
所以，最好还是加上< >，这代表了创建的是一个泛型类，只是类型我没写，由系统帮我推导出来。

但是如果声明时没加泛型，而new的时候加了，则系统不会进行推导，系统会认为没有给泛型变量赋值，不赋值默泛型变量的值为Object，如下：



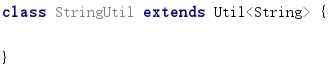
因为在声明时没有指定泛型导致了泛型丢失，如果不写声明，则不会丢失，如下：  


甚至一个泛型都不写也是可以的，如下：

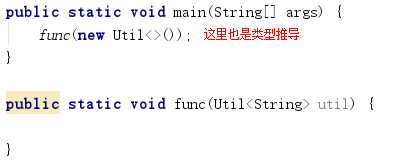
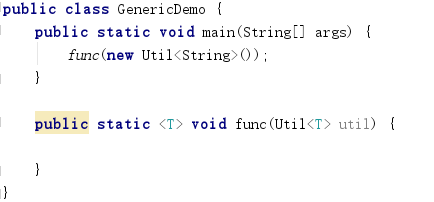
  

可以看到，如果没有给泛型变量赋值，则泛型变量的值为默认值——Object

## 泛型类中泛型变量的赋值

1. 在继承的时候赋值：  
    
2. 在创建对象的时候赋值  
    

 这里new Util< >虽然没有填入泛型，但是通过类型推导系统会自动帮我们填入String

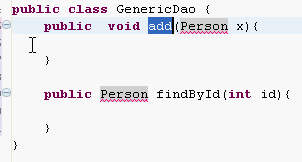
  
  这里同时给类上的泛型变量和方法上的泛型变量都赋了值

## 泛型类应用

**//dao 是data access object的缩写，这个类的功能主要是crud (create、read或retrieve、update、delete的缩写)，即增加、查询、更新、删除数据。**

**一说到J2EE，它工作的事情都是CRUD。J2EE是13种技术。**

**定义一个DAO如下：**

****

很明显这个DAO是专门操作Person的，怎样可以这个DAO也可以操作其他的类型呢，如可以操作产品，用泛型就可以解决，如下：

**public** **class** **GenericDao**<T> {

**public** **void** add(T t) { //增

}

**public** T findById(**int** id) //查

{

**return** **null**;

}

**public** **static** **void** main(**String** args[])

{

**GenericDao**<**Student**> dao = **new** **GenericDao**<**Student**>();

dao.add(**new** **Student**());

**Student stu** = dao.findById(1);

}

}

# 泛型方法

## 泛型方法定义

普通方法、构造方法和静态方法中都可以使用泛型

**class** Util {  
 **public** <T> **void** print(T t) {  
 System.***out***.println(t);  
 }  
}

方法里的泛型变量只能在返回值前面声明，且必须紧挨着返回值

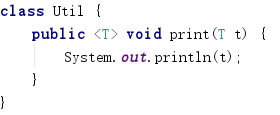
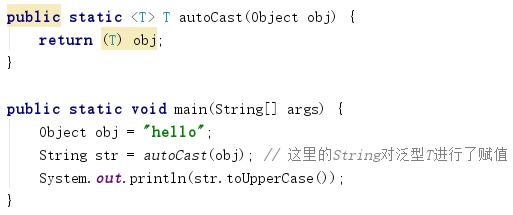
在方法中声明泛型变量，这样的方法叫“泛型方法”。在方法中定义的泛型变量，只能在这个方法中使用，所以同一个类中的多个方法里的泛型变量名可以相同，如：

**public** <T> **void** func\_1() { }  
**public** <T> **void** func\_2() { }

也可以声明多个泛型变量，如下：

**public** <K, V> **void** print(K k, V v)

## 泛型方法中泛型变量的赋值

1. 在传参的时候赋值  
       
    这里传参为String，则泛型T的值为String
2. 如果方法的返回值是就是泛型T，则我们在我们声明变量时就对泛型进行赋值了，如下：  
    

## 静态方法的泛型

静态方法不可以访问类上定义的泛型变量

**class** Util {  
 **public static void** print(T t) {  
 System.***out***.println(t);  
 }  
}

这样编写语法不正确，<T>中T是在new对象的时候才明确的，也就是说在new对象的时候才知道T是什么类型，而静态方法调用的时候不需要new对象，这时候T就不知道是什么类型了。

## C++中的模板函数

Java中的泛型是从C++中借鉴过来的，Java中的泛型没有C++中的强大，这是由于Java虚拟机设计的原因，但是Java还是尽量去模仿C++中的泛型。

如下函数的结构很相似，仅类型不同：

int add(int x , int y) { return x+y ; }

float add(float x , float y) { return x+y ; }

double add(double x , double y) { return x+y ; }

C++用模板函数解决，只写一个通用的方法，它可适应各种类型（这就是泛型了），示意代码如下：

Template<class T>

T add(T x, T y) { return (T) ( x +y ) }

Java中的泛型类型类似于C++中的模板，但是这种相似性仅限于表面，Java语言中的泛型基本上完全是在编译器中实现，用于编译器执行类型检查和类型推断。然后生成普通的非泛型的字节码，这种实现称为擦除（erasure）（编译器使用泛型类型信息保证类型安全，然后在生成字节码之前将其清除）。这是因为扩展虚拟机指令集来支持泛型被认为是无法接受的，这会为Java厂商升级其JVM造成难以逾越的障碍。所以，java的泛型采用了可以完全在编译器中实现的擦除方法。

**使用Java的泛型模拟上面C++的模板函数：**

**public static void** main(String[] args) {  
 Integer i = *add*(1, 2);  
 Number n = *add*(1.5, 3);  
 Object obj = *add*(3, **"abc"**);  
}  
  
**public static** <T> T add(T x, T y) {  
 **return** y; // 此处不能写成return x + y;因为x和y这两个变量不一定能使用加法操作符，这里也可以看出Java的泛型没有C++的强大。  
}

这很奇怪啊，按方法的定义，参数x和y肯定是同一个类型的，而上面调用了3次add方法，第二第三个调用明显它们的参数类型不一致，但是编译器竟然没有报错，这里编译器到底把方法中的两个参数当成了什么类型呢？答：取它们的交集，如3.5和3都是Number类型，则这两个参数被当成Number类型处理，因此：

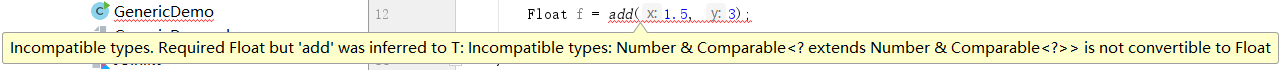
Number x = add(3.5, 3); // 正确

Float x = add(3.5, 3); // 错误

这样的实现是类型推断，在Java中的实现是使用一大堆的表达式计算来实现推断的，很难看懂。

那add(3,”abc”); 两个参数的交集是Object，因为3是Number，“abc”是String，Number和Object都是Object的子类。

在使用时，如果你不知道是用什么类型保存，可以随便用一个类型，在报错时就能从警告信息中看出来应该用什么类型，如下：



警告信息说：不兼容的类型。需要Float，但是'add'方法中的T被推断为Number & Comparable。 因为1.5的类型为Float，3的类型为Integer，而Float和Integer的这两个类都是继承了Number并且实现了Comparable。

## 泛型方法实例

Object类中的一个泛型方法使用：

**public static** <T> T requireNonNull(T obj) {  
 **if** (obj == **null**)  
 **throw new** NullPointerException();  
 **return** obj;  
}

此方法判断对象是空则抛出异常，给的是什么类型，返回去的还是什么类型。

# 接口中的泛型

**public interface** Person<T> {  
 **void** show(T t);  
}  
  
**class** Student **implements** Person<String> {  
  
 @Override  
 **public void** show(String str) {  
 System.***out***.println(str);  
 }  
  
 **public static void** main(String[] args) {  
 Student stu = **new** Student(); // 因为在Student上没有声明泛型，所以这里就new的时候也不需要泛型  
 stu.show(**"abc"**);  
 }  
}

上面的例子中Student实现了Person，那么必须覆盖接口中的show方法，在class Student implements Person<String> {}中传递的泛型变量值是String，所以覆盖**void** show(T t)方法中的T就是String，也就是必须覆盖public void show(String t)方法。如果Student在实现Person接口时，也不知道会传给**void** show(T t)方法中的参数是什么类型，那么Student类也可以定义为泛型，如下：

**class** Student<T> **implements** Person<T> {  
  
 @Override  
 **public void** show(T t) {  
 System.***out***.println(t);  
 }  
  
 **public static void** main(String[] args) {  
 Student<String> stu = **new** Student<>(); *// 因为Student类上声明了泛型，所以new的时候也要泛型* stu.show(**"abc"**);  
 }  
}

注：这里Person<T>中的T就不是声明泛型变量了，而是使用泛型变量T。这个变量T是在Student<T>这里声明的。

# 泛型中的？通配符

**public class** GenericDemo {  
 **public static void** main(String[] args) {  
 List<String> strings = Arrays.*asList*(**"a"**, **"b"**, **"c"**);  
 List<Integer> numbers = Arrays.*asList*(1, 2, 3);  
 *printCollection*(strings);  
 *printCollection*(numbers);*//会报错* }  
  
 **public static void** printCollection(List<String> list){  
 list.forEach(e -> System.***out***.println(e));  
 }  
}

上面例子中定义的printCollection方法只能接受的参数是：List<String>类型，所以*printCollection*(numbers)会报错，可不可以把printCollection中的List声明成List<Object>呢？答：不行，这样的话，在传参时相当于下面的语句：

List<Object> list = new ArrayList<String>(); // 这样的语句编译就通不过

右边的声明是可以存任意对象，而右边的声明是只能存String对象，这就有冲突了，会出问题的，问题示例如下：

**public static void** main(String[] args) {  
 List<String> strings = **new** ArrayList<>();  
 *printCollection*(strings); // 这行代码编译不通过，假设能通过，则下一行代码会挂类转换异常。这里的传参相当于：List<Object> list = new ArrayList<String>();   
 String str = strings.get(0)  
}  
  
**public static void** printCollection(List<Object> list){  
 list.add(System.**class**);  
}

那要想printCollection这个方法能接受的参数比较通用的话可以用“？”通配符，把printColleciton方法中的List的泛型参数改成“？”，这样*printCollection*(numbers);就不会报错了。

Object代表了可以接受任何类型，而？代表不知道接受的是什么类型，这是不一样的概念。举例如下：

**public static void** printCollection(List<Object> list){  
 list.add(System.**class**); // 编译通过  
}

**public static void** printCollection(List<?> list){  
 list.add(System.**class**); // 编译失败，？代表不知道什么类型，既然都不知道是什么类型你怎么能存东西进去呢。万一别人传进来的是一个new ArrayList<Person>()的集合，这就代表了只能装Person，所以这里装了一个System.class肯定不行的，装Object对象也是不行的，就算装Person也是不行的，说白了就是泛型为？的集合不能调用add方法，不论你添加的是什么都是不可以的。  
}

如果想要表示一个集合什么对象都可以装，可以使用：List<Object> list = new ArrayList<Object>();

?类型的集合不能调用add添加元素，但是可以调用get方法获取对象，但是必须强转，因为？代表不知道是什么类型，示例如下：

List<String> strings = **new** ArrayList<>();  
strings.add(**"haha"**);  
List<?> list = strings;  
// list.add("hehe"); // 这行代码编译不通过// ? str = list.get(0); // 这行代码编译不通过，？代表不知道是什么类型，它会把泛型实例化为Object，所以可以这样：Object obj = list.get(0);String str = (String) list.get(0); // 因为不知道是什么类型，所以要强转System.***out***.println(str);

其实除了有关存东西的方法是不可以调用的，其它的任何方法应该都是可以调用的。

其实想在方法中调用add，是有解决方案的，如下：

**public static void** main(String[] args) **throws** Exception {  
 List<String> list = **new** ArrayList<>();  
 *printCollection*(list, **"Haha"**);  
}  
  
**public static** <T> **void** printCollection(List<T> list, T element) {  
 list.add(element);  
 list.forEach(e -> System.***out***.println(e));  
}

这样的话T还可以用来声明变量呢！一般来说使用？的方式会比较方便，除非你真的需要使用T来声明变量，如上例子中，T element就是声明变量，当然在方法体中也可以使用T来声明变量。

## ？通配符的应用

前面的printCollection是一个应用，下面再来看Class类上对于？的应用，Class是一个泛型类，所以我们在使用Class时最好带上泛型。Class中有一个forName方法，如下：





用法：

Class<?> clazz = Class.*forName*(**"java.lang.String"**);

forName方法的功能为：给它传一个完整类名，它就能返回这个类的字节码对象。

因为Class是一个泛型类，所以这个方法中要返回Class也应该带上泛型，但是泛型的类型是没办法确定的，所以用？来表示。 下面语句编译失败：

Class<String> clazz = Class.*forName*(**"java.lang.String"**);

这其实相当于Class<String> clazz = Class<?>，这样的赋值明显是没法通过的，因为？代表不知道是什么类型，假如上体是一个Integer类型，你这里用String类型来接收，这肯定是不行的。如果反过来则可以的：Class<?> class = Class<String>;

## ？通配符用在什么地方

只能用在声明变量时使用，在声明泛型的地方不可以使用，未例如下：

**class** Person<?> { // 不可以  
**public** <?> **void** fun() {} // 不可以}

**public static void** main(String[] args) {  
 List<?> list\_1 = **new** ArrayList<String>(); // 可以，但没人会这么用的，因为这里右边都有具体类型String了，左边就很奇怪了，开发人员是傻的吗  
List<?> list\_2 = *func*(**new** ArrayList<String>());// 左边声明？是没办法，因为方法中返回值就是这样的，这种例子比如Class的forName方法就是这样的}  
  
**private static** List<?> func(List<?> list) { // 在方法的参数中使用？是最常用的  
**return** list;  
}

# 泛型中的向后兼容

## 泛型不考虑继承关系但向后兼容

泛型是不考虑类型参数的继承关的，在使用泛型时，左右两边的泛型类型一定要相同，如果不相同则会出现矛盾，这是不允许的，但是为了可以向后兼容，左右两边有一边使用原始类型，或者两边都使用原始类型则是OK的，如果两边都写了泛型则一定要相同。示例如下：

List<String> list = new ArrayList<Object>(); // 左边说只能装String，右边说可以装Object，这是矛盾的，编译器不允许不通过。

List<String> list = new ArrayList(); // 左边说只能装String，右边说可以装Object，这是矛盾的。但是为了向后兼容，编译器允许通过

List<Object> list = new ArrayList<String>(); // 左边说可以装Object，右边说只能装String，这是矛盾的，编译器不允许不通过。

List list = new ArrayList<String>(); // 左边说可以装Object，右边说只能装String，这是矛盾的，但是为了向后兼容，编译器允许通过

List list\_1 = new ArrayList<String>();

List<Object> list\_2 = list\_1; // 编译通过，编译器是一句一句编译的，编译器只知道list\_1是一个原始类型

对于左右两边泛型不一样时，为什么编译器不允许编译通过，因为如果允许的话就有可能出现类型转换异常，而泛型的出现就是为了避免出现类型转换异常的，所以，只要你用了泛型，编译器就能给你保证不会出现类转换异常，这是多么好的一个事情啊！！！

**左边说只能装String，右边说可以装Object，这样有可能出现异常，示例如下：**

List<Object> list\_1 = **new** ArrayList<Object>();  
list\_1.add(System.**class**);  
List<String> list\_2 = list\_1; // 这一行相当于：List<String> list\_2 = new ArrayList<Object>(); 这一行编译不通过，如果能编译通过，则下一行代码挂异常  
String str = list\_2.get(0); // list\_2这个变量的声明是表明了装的是String，所以取出来也应该是String，但实际上取出来的是System.class

如果把第一行代码中的泛型删除，编译就通过了，但是运行会抛异常，即使抛异常，编译器也让编译通过，这是为了向后兼容所付出的一点点代价

**左边说可以装Object，右边说只能装String，这样有可能出现异常，示例如下：**

List<String> list\_1 = **new** ArrayList<String>();  
List<Object> list\_2 = list\_1; // 这一行相当于：List<Object> list\_2 = new ArrayList<String>();这一行编译不通过，如果能编译通过则下一行代码就能把System.class装到集合中,则最后一行代码挂异常  
list\_2.add(System.**class**);

String str = list\_1.get(0); // v1这个变量的声明是表明了装的String，所以取出来也应该是String，但实际上取出来的是System.class

如果把第二行代码中的泛型删除，编译就通过了，但是运行会抛异常，即使抛异常，编译器也让编译通过，这是为了向后兼容所付出的一点点代价

**左右两边都用原始类型：**

List list = **new** ArrayList(); // 这里两边都用了原始类型，为了向后兼容，这是OK的  
list.add(System.**class**);  
String str = (String) list.get(0); // 挂类型转换异常异常

编译就通过了，但是运行会抛异常，即使抛异常，编译器也让编译通过，这是为了向后兼容所付出的一点点代价

泛型不考虑类型参数的继承关、向后兼容这些特性不但适用于集合类，对于其他类一样适用，示例如下：

**public static void** main(String[] args) {  
 PrintUtil<String> util\_1 = **new** PrintUtil(); // 把泛型T认为是String  
PrintUtil util\_2 = **new** PrintUtil<String>(); // 把泛型T认为是Object  
PrintUtil util\_3 = **new** PrintUtil(); // 把泛型T认为是Object  
PrintUtil<String> util\_5 = **new** PrintUtil<Object>(); // 编译失败  
PrintUtil<Object> util\_6 = **new** PrintUtil<String>(); // 编译失败  
}  
  
**static class** PrintUtil<T> {  
 **public void** print(T t) {  
 System.***out***.println(t);  
 }  
}

## 泛型为什么要向后兼容

假设别人很早以前就写了一个类库，那时还没有泛型，别人写的这个类库功能很强大，当我们也需要这些功能时就可以直接调用别人的类库，而不要自己去再开发一个类库，假设这个类库中有如下两个方法：

public class ListUtil {

public static void printCollection(List list) { // 打印List代码 }

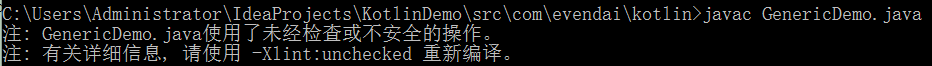
public static List getBooks() { return new ArrayList(); }

}

假设这两个方法是早期开发的，那时候还没有泛型，这两个方法一个是接收参数List，一个是返回List，都没有泛型，可是现在有泛型了，我们在做开发时肯定是要使用泛型的，那我们在使用泛型的同时又想使用上面的类库，这就有了泛型集合和原始集合的兼容性问题，Sun公司肯定会让它们实现兼容的，不然早期开发的类库就没有复用价值了。

### 给老类库中传递泛型集合

**public class** GenericDemo {  
 **public static void** main(String[] args) {  
 List<String> list = **new** ArrayList<>();  
 list.add(**"haha"**);  
 ListUtil.*printCollection*(list); // 这里的传参操作相当于：List list = new ArrayList<String>();  
}  
}  
  
// 假设这是一个旧类库，别人已经编译成了jar，我们没有源代码，所以只能调用不能修改**class** ListUtil {  
 **public static void** printCollection(List list) {  
 list.add(**"hehe"**);  
 String str\_1 = (String) list.get(0);  
 String str\_2 = (String) list.get(1);  
 System.***out***.println(str\_1);  
 System.***out***.println(str\_2);  
 }  
}

编译如下：  
   
按照提示使用-Xlint:unchecked重新编译，如下：  
 

这里显示调用list.add("hehe")是未经过检查的，对于调list.get(0)的调用却没有检查警告。为什么对于原始类型的add方法会提示警告呢？原因如下：

**public static void** main(String[] args) {  
 List<String> list = **new** ArrayList<>();  
 *printCollection*(list);   
 String str = list.get(0); // 挂类型转换异常  
}  
  
**public static void** printCollection(List list) {  
 list.add(Integer.*valueOf*(55));  
}

对于get方法不会提示警告，因为get如果有异常会在get调用处直接抛出类转换异常。而在调用add的时候，如果存入了非法数据，并不会在add时报异常，所以给add一个未经检查警告。那

编译时对于add方法报出未经检查警告，如何去检查呢？在printCollection方法上加注解：@SuppressWarnings("unchecked") ，这是什么意思呢，对于上面的例子，可以这样理解：加上@SuppressWarnings("unchecked") 代表向编译器说：我已经检查了我的add方法了，存入的就是合法的值，我确定没有存入非法的值。所以此时编译器就不会再警告你了。

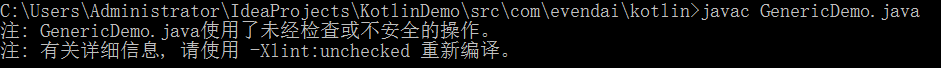
那为什么get方法没有报出未经检查警告呢？因为：String str\_1 = (String) list.get(0); 这里必须加上强转，否则会编译不通过，加上强转其实就是在跟编译器说：我检查过了，list中装的就是String。所以编译器就不需要给出警告了。

在编译出现这种警告时，大多数情况是因为你使用了一些陈旧的类库，而你也没有办法去修改这些类（因为你只有jar文件，没有源代码），没办法在这些类上加上@SuppressWarnings("unchecked")注解，所以编译时看到警告时没什么大不了的，因为你也没办法去消除这些警告。 所以，在编译后一定要看看有没有警告信息，如果有的话，是不是我们自己写的类导致的，如果是的话则一定要去修改一下，肯定是因为使用了原始集合了，改成使用泛型集合即可。

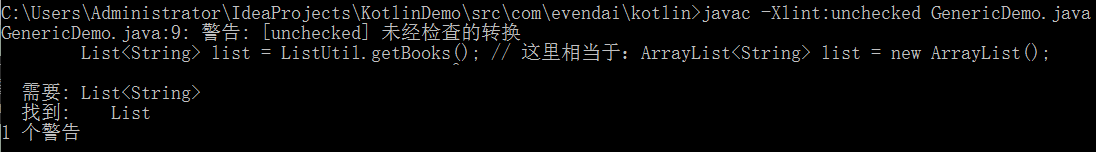
### 获取老类库中返回的原始集合

**public class** GenericDemo {  
 **public static void** main(String[] args) {  
 List<String> list = ListUtil.*getBooks*(); *// 这里相当于：ArrayList<String> list = new ArrayList();* }  
}  
  
**class** ListUtil {  
 **public static** List getBooks() {  
 **return new** ArrayList();  
 }  
}

编译如下：



重新编译如下：



因为在List<String> list = ListUtil.getBooks(); 这行代码中，左边声明了list只能存入String，而右边返回的List可以装任意类型，万一右边的集合装的是Integer呢，所以报了一个未经检查的转换警告。如何消除警告呢？可以在这个语句上面加上注解，如下：

@SuppressWarnings(**"unchecked"**)  
List<String> list = ListUtil.*getBooks*();

也可以加到main方法上，区别是加到语句上，只对这个语句生效，加到方法上，对方法中所有的未经检查语句生效（包括调用add方法）。

修改代码如下：

**public class** GenericDemo {  
 **public static void** main(String[] args) {  
 @SuppressWarnings(**"unchecked"**)  
 List<String> list = ListUtil.*getBooks*();  
 list.add(**"abc"**);

String str = list.get(0); // 编译通过，运行挂掉  
 }  
}  
  
**class** ListUtil {  
 **public static** List getBooks() {  
 List list = **new** ArrayList();  
 list.add(System.**class**);  
 **return** list;  
 }  
}

编译：



对于list.add(System.class)出现了警告，这就有点奇怪了，getBooks( )方法中的List list = new ArrayList();这里就没有加任何的泛型，说明就是可以存任意对象的，还警告个舍呀！因为：

List list\_1 = **new** ArrayList<String>();  
List list\_2 = **new** ArrayList();

编译器是一句一句代码编译的，对于编译器来说list\_1和list\_2都是原始类型，没有任何区别，所以，只要在原始类型上调用add方法就会报出未经检查警告。

**对于原始类总结如下：**

List<String> list = **new** ArrayList(); // 这行代码编译时就会有警告，因为右边的集合中装的可能不是String，而左边声明了只能装String； 在调用add方法时没有警告。  
List list = **new** ArrayList<String>(); // 这行代码编译时没有警告，因为右边的集合不管装什么类型对于左边的声明都是适用的，但是在调用add方法时有警告，因为add时可能装的不是String，而右边声明了只能装String

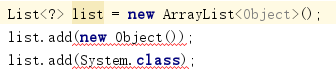
List list = **new** ArrayList(); // 这行代码编译时没有警告，但是在调用add时有警告，其实这种方式跟第2种差不多，虽然第2种的右边有泛型，但是左边没有泛型，因此在使用时对于编译器来说第2种和第3种的List没有任何区别，都是原始类型

这3种写法都是OK的，OK是因为要向后兼容。第一第二种是属于我们现在用泛型，要和旧库类的原始类型兼容。 第三种属于旧类库中的用法，这种旧类库代码在现代的JRE中也要能运行吧，这就是兼容。

虽然这3都可以的，但是我们在做开发的时候千万别这么写。

第1种写法肯定是只能装String，开发时可以这样写：List<String> list = new ArrayList<>();在右边加上尖括号，让编译器去推导类型。

第2、3种写法的list什么都能装，在做开发时如果也想实现这样的功能，可以这样写：List<Object> list = new ArrayList<>();

第2、3种写法的list能不能这样代替呢：List<?> list = new ArrayList<Object>(); 这是不可以的，虽然右边声明了可以存任意类型，但是左边的？代表的是不知道存的是什么类型，所以如果调用add方法，不论存什么对象都会报错，如下：  


# 泛型的上限、下限

## ？限制

### ？上限、下限含义

上限：<? extends Person>，这个限制了泛型可以传递的类型为Person或Person的子类。Person下面的子类不固定，也就是下面的子类可以扩展，而上面的父类Person己固定，所以叫上限。

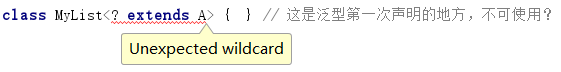
下限：<? super Person>，这个限制了泛型可以传递的类型为Person或Person的父类。Person上面的父类不固定，也就是上面的父类可以扩展，而下面的子类Person己固定，所以叫下限。

这个上限、下限一般用在集合中。

### ？限制细节

?可以用来表示不知道什么类型，也可以用来对泛型的类型进行进一步限制，进一步限制只能用于在类上声明的泛型，不能用于在方法上声明的泛型。

?限制出现在什么地方？ 答：出现在第二次声明的时候，如下是错误的：



那第二次声明的地方是在哪里？答：声明变量的时候，示例如下：

**class** A { }  
**class** B **extends** A { }  
**class** C **extends** B { }  
**class** MyList<T **extends** A> { } // 声明泛型T，且对T的范围进行了限制，T的类型被限制为只能是A或A的子类

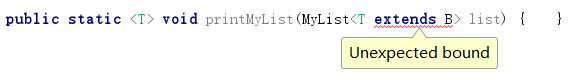
MyList<? **extends** B> list = **new** MyList<C>(); // 这种方式基本不会使用，new的时候就已经有具体类型了，左边的声明就用具体的类型就OK了！

void printMyList(MyList<? extends B> list) // ?限制就是专门用在方法里的，代表我接收的MyList中的元素可以是B或者B的子类。注意：这里泛型的类型是“？”，虽然我们知道“？”的类型可以是B或者B的子类，但是具体是哪一个类型，这是未知的，所以，在这个方法中，不能用？来声明变量，也不能往这个集合中存入数据，因为类型是未知的，你怎么知道可以存入什么类型的数据呢！示例如下：

**public static void** printMyList(MyList<? **extends** B> list) {  
 // ? item = list.get(0); // 编译失败，如果你把item当成B类型，然后调用了一个B对象的方法，万一别人传入的集合是new MyList<C>呢！  
 // list.add(new B()); // 编译失败，万一别人传入的集合是new MyList<C>呢！  
B b = list.get(0); // 编译通过，虽然不知道取出来的是B类型还是C类型，但是肯定属于B类型。  
Object obj = list.get(0);// 编译通过，虽然不知道取出来的是B类型还是C类型，但是肯定属于Object类型。  
}

**public static void** printMyList(List<? **super** B> list) {  
 // ? item = list.get(0); // 编译失败，如果你把item当成B类型，然后调用了一个B对象的方法，万一别人传入的集合是new MyList<C>呢！  
 // list.add(new B()); // 编译失败，万一别人传入的集合是new MyList<C>呢！  
A b = list.get(0); // 编译失败，虽然A是B的父类，但是万一A的父类是Q，我们传入的集合是new MyList<Q>呢，不能把Q赋值给A  
Object obj = list.get(0);// 编译通过，虽然不知道取出来的是B类型还是C类型，但是肯定属于Object类型。}

在？通配符中，可以使用泛型方法替代？通配符，但是在这里行不通，如下是编译不能通过的：



?限制使用示例：

**class** Util {  
 // 这里过?进一步限制了T的类型只能是B或者B的子类，范围比之前小了**public static void** printMyList(MyList<? **extends** B> list) { }  
}

**public static void** main(String[] args) {  
 MyList<A> listA = **new** MyList<>();  
 MyList<B> listB = **new** MyList<>();  
 MyList<C> listC = **new** MyList<>();  
 Util.printMyList(listA); // 这行代码编译不通过  
Util.printMyList(listB); // 这行代是OK的  
Util.printMyList(listC); // 这行代是OK的  
}

注意，下面的代码竟然可以编译通过：

**public static void** printMyList(MyList<? **extends** Person> list) { }

我们在声明MyList的时候已经明确了泛型只能是A或者A的子类，这里第二次声明时尽然可以指定为其它类型，编译器疯了吧，给Sun发邮件，看会不会有什么奖励呢^\_^

在真实开发中，我们不会写出这样的方法来的，因为这是一个没有用的方法，因为你是不可能创建出这个方法需要的集合对象的，如下：

MyList<Person> list = **new** MyList<Person>(); // 这个代码编译失败，MyList只能装A或者A的子类

### ？限制的上限实例

**public class** GenericDemo {  
 **static class** Grandpa { }  
 **static class** Father **extends** Grandpa { }  
 **static class** Son **extends** Father { }  
  
 **public static void** print(List<? **extends** Father> list) {*//这个方法只能接受Father或Father的子类对象* Iterator<? **extends** Father> it = list.iterator();  
 **while** (it.hasNext()) {  
 System.***out***.println(it.next());  
 }  
 }  
  
 **public static void** main(String[] args) {  
 List<Grandpa> grandpas = **new** ArrayList<>();  
 List<Father> fathers = **new** ArrayList<>();  
 List<Son> sons = **new** ArrayList<>();  
  
 grandpas.add(**new** Grandpa());  
 fathers.add(**new** Father());  
 sons.add(**new** Son());  
  
 *print*(grandpas); *// 这个不符合要求  
 print*(fathers);  
 *print*(sons);  
 }  
}

### ？限制的下限实例

**public class** GenericDemo {  
 **static class** Grandpa { String **name**; **public** Grandpa(String name) {**this**.**name** = name;} }  
 **static class** Father **extends** Grandpa { **public** Father(String name) {**super**(name);} }  
 **static class** Son **extends** Father { **public** Son(String name) {**super**(name);} }  
  
 **static class** SonComparator **implements** Comparator<Son> {  
 **public int** compare(Son s1, Son s2) {  
 **return** s1.**name**.compareTo(s2.**name**);  
 }  
 }  
  
 **public static void** main(String[] args) {  
 TreeSet<Son> sons = **new** TreeSet<>(**new** SonComparator());  
 sons.add(**new** Son(**"son-06"**));  
 sons.add(**new** Son(**"son-05"**));  
 sons.add(**new** Son(**"son-07"**));  
 sons.forEach(son -> System.***out***.println(son.**name**));  
 }  
}

这段代码的功能为把3个Son对象存到TreeSet中，TreeSet是有序的，而顺序的决定由Comparator决定，最后输入TreeSet中的对象以验证是否排序。

这时我们再创建一个Father的集合，如下：

TreeSet<Father> fathers = **new** TreeSet<>(**new** SonComparator()); *// 这行代码报错*fathers.add(**new** Father(**"father-04"**));  
fathers.add(**new** Father(**"father-03"**));  
fathers.add(**new** Father(**"father-05"**));  
fathers.forEach(father -> System.***out***.println(father.**name**));

这是因为SonComarator这个比较器无法在Father的TreeSet中使用，解决方案——为Father创建一个比较器，但是这样做不好，如果再来一个Grandpa，或者更多的类，难道每来一个类就创建一个对应的比较器吗，这样就太费尽了。查看一下TreeSet的文档，类声明和构造函数声明如下：





如果我们创建一个Grandpa的比较器：class GrandpaComparator implements Comparator<Grandpa>，这个比较器可以用于Grandpa和它的所有子类，分析如下：

TreeSet<Son> sons = **new** TreeSet<>(**new** GrandpaComparator());

这里TreeSet中的泛型比较简单，没什么好说的，主要是TreeSet的构造函数中的参数是否是正确的。

因为new TreeSet<Son>把TreeSet的泛型变量E给赋值了，则E的值为Son，所以TreeSet的构造函数变成：TreeSet(Comparator<? super Son> c) ，需要的比较器必须是Son或者Son的父类，而我们的的GrandpaComparator确实是Son的父类，所以GrandpaComparator可以用于TreeSet<Son>集合的比较。对于TreeSet<Father>也同理，如下：

TreeSet<Father> fathers = **new** TreeSet<>(**new** GrandpaComparator());

此时的构造函数为：TreeSet(Comparator<? super Father> c) ，需要的比较器必须是Father或者Father的父类，Grandpa是父类，所以GrandpaComparator也可以用于TreeSet<Father>集合的比较

总结就是GrandpaComparator对Grandpa和它的所有子类都适用，需要注意的是，在GrandpaComarator中只能调用得到Grandpa的方法，调用不到子类的方法。

## 非？限制

“非？限制”只能使用上限，不能使用下限，可能是因为下限在类和方法中没有使用场景。“非？限制”格式如下：

**class** Util<T **extends** Person> { }

“非？限制”中的T可以用来声明变量，这是它与“?限制”的不同点，还有一个不同点是——“非？限制”可以指定多个上限，使用&进行连接，如下：

**class** Util<T **extends** Person & Serializable & Cloneable> { }

注意：第一个上限可以是类也可以是接口，但是从第二个起，必须是接口

“非？限制”在类上使用，示例如下：

**interface** Person { String getName(); }  
  
**class** Student **implements** Person {  
 **public** String getName() { **return "李连杰"**; }  
}  
  
**class** Haha<P **extends** Person> { // 在类的泛型中使用上限  
 P **p**;  
 **public** Haha(P p) { **this**.**p** = p; } // 因为泛型P带了上限，所以这里只能传Person或者Person的子类  
 String get欢迎语() { **return p**.getName() + **", 你好!"** ; } // 因为泛型P带了上限，所以可以调用上限Person中的方法  
}  
  
**public static void** main(String[] args) {  
 Student stu = **new** Student();  
 Haha<Student> haha = **new** Haha<>(stu);

System.***out***.println(haha.get欢迎语());  
}

“非？限制”在方法上使用，示例如下：

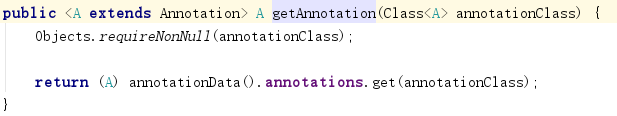
**class** Hehe<T **extends** Person> **extends** Haha<T> { // 没有任何功能，只是继承了Haha  
 **public** Hehe(T t) { **super**(t); }  
}

**class** Util {  
 **public static** <H **extends** Haha> **void** print(H h) { // 在方法的泛型中使用上限  
System.***out***.println(h.get欢迎语()); // 因H带了上限，所以可以调用上限Haha中的方法  
}  
}

**public static void** main(String[] args) {  
 Student stu = **new** Student();  
 Hehe<Student> hehe = **new** Hehe<>(stu);  
 Util.*print*(hehe); // 这里即可传Haha，也可传Hehe  
}

在这个例子中的print方法和“public static void print(Haha h)”没有区别，那是不是在方法中声明“非？上限”的泛型没有用啊？这错误的，这里只是我们举的这个例子没有，在Class类上有一个getAnnotaion方法，声明和源码如下：





# 泛型中的类型擦除

**new** Vector<String>().getClass() == **new** Vector<Date>().getClass() // 结果为true

泛型是提供给javac编译器使用的，编译器编译带泛型的集合时会去除掉“类型”信息（简称为“去类型化”），使程序运行效率不受影响。由于编译生成的字节码会去掉泛型的类型信息，所以，只要能跳过编译器，就可以往某个泛型集合中加入其他类型的数据，例如，用反射得到集合，再调用方法即可添加任意类型的数据。示例如一下：

**public** **static** **void** main(String[] args) **throws** Exception {

ArrayList<Integer> intList = **new** ArrayList<Integer>();

// 通过反射得到集合的add(Object obj)的方法

Method addMethod = intList.getClass().getMethod("add", Object.**class**);

addMethod.invoke(intList, "abc");// 通过反射往集合中装入了字符串

// 编译器只知道它取出来的是Integer，所以编译通过，但运行报错，因为存进去的是字符串

Integer i = intList.get(0);

}

对于这个例子，有一个奇怪的问题，可查看同目录下的“02\_泛型问题-深入理解类型擦除.doc”文档。

不通过反射，通过向后兼容的方式，也可以给泛型集合添加任意类型：

**public static void** main(String[] args) {  
 List<String> list = **new** ArrayList<>();  
 *printCollection*(list);   
 String str = list.get(0);  
}

// 假设这是一个旧类库，别人已经编译成了jar，我们没有源代码，所以只能调用不能修改，JDK肯定是向后兼容的，所以我们现在写的带泛型的集合肯定也可以传到这个方法中  
**public static void** printCollection(List list) {  
 list.add(Integer.*valueOf*(55));  
}

由于类型擦除，所以下面的方法不属于重载，在编译时将会报错：

**public void** a(List<String> list) { }  
**public void** a(List<Date> list) { }

# 数组中不能使用泛型

张老师也不知道为什么不可以。

例如，下面语句有编译失败：

Vector<Integer> vectorArray[] = new Vecotr<Integer>[10]；

# 内部类中的泛型

Map是一个泛型类，如下：



它有一个内部类，如下：



当我们要使用内部类Entry时，如下：

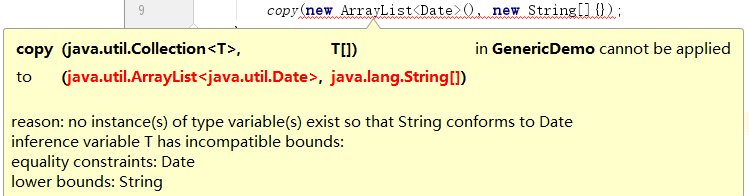
Map.Entry<String, Intenger> entry; // 正确

Map<String, Integer>.Entry<String, Intenger> entry; // 错误，虽然Map类有泛型，但是这里是使用Entry类，Map.Entry是用于表示Entry是Map的一个内部类，这里Map不需要带上泛型。

# 泛型的类型推断

List<String> str = new ArrayList<>(); // 右边推断为String

public <T> void copy(List<T> list, T[] array); // 在调用这个方法时，如果传入：copy( <List<Date>, String[] )，这个调用编译不通过，报错如下：



大致意思是推断变量T有不兼容的的界限

public <T> void copy(T[] arr\_1, T[] arr\_2); // 这个方法和上一个很像，调用：copy( Date[], String[] )，这个调用编译通过，这就很奇怪了，在这种调用下，编译器把Date和String都当成Object对象了。这说明在方法里，如果泛型有用到集合上，则编译器在进行类型推断时不会取交集，也就是说给方法传参时只能用同一个相同的类型。而如果在方法上泛型没有用到集合中，则编译器会取交集，这里证明一下刚刚说的编译器把Date和String都当成了Object：

**public static** <T> T[] copy2(T[] arr\_1, T[] arr\_2) {  
 **return** arr\_1;  
}  
  
**public static void** main(String[] args) **throws** Exception {  
 Object[] arr = *copy2*(**new** Date[]{}, **new** String[]{});  
}

能用Object[]接收结果，说明编译器是把T当成Object。

对于取交集的详细列子可查看 “泛型方法-->C++中的模板函数”这一知识点。

# 泛型应用实例

## 1、包装异常

用泛型变量表示异常，这个泛型变量可以用于方法throws列表中，但是不能用于catch中。如：

**public static** <E **extends** Exception> **void** fun() **throws** E {  
 **try** {  
   
 } **catch** (Exception exc) { *// 这里不能泛型使用：E e* **throw** (E) exc; *// 这个地方具体怎么把e包装成E的类型呢？ 需要去百度一下学习* }  
}

这里具体怎么把exc这个异常包装成E类型呢，百度了没有找到答案，我自己的推测如下：

泛型变量它是个变量，肯定是要赋值的，方法中泛型变量的赋值方式有两种，一种是方法的参数赋值，一种是使用返回值的方式赋值，一个方法返回值是个异常，这不太合理，所以只能是使用方法参数的方式给泛型变量赋值了，示例如下：

**class** PersonalException **extends** Exception { }  
  
**class** ImageLoader{  
  
 **public static** <E **extends** Exception> **void** loadImage(String imageUrl, E e) **throws** E {  
 **try** {  
 **int** x = 5 / 0; *// 这设这里下载图片出现异常了* } **catch** (Exception exc) {  
 e.initCause(exc); *// 对异常进行包装* **throw** e;  
 }  
 }  
}

**public static void** main(String[] args) **throws** PersonalException {  
 String imageUrl = **"http://www.abc.com/flower.jpg"**;  
 ImageLoader.*loadImage*(imageUrl, **new** PersonalException());  
}

对异常进行包装的好处： 在分层设计里面是很经常应用的一种应用。 因为没有实际这么搞过，所以不理解这到底是什么意思，我自己想的一个好处：比如我的App分了多个模块，像上面的例子，我们一看抛出的是PersonalExcption，则知道穿上异常是在个人中心里产生。当然异常也可以不抛出，包装后写到文件中去，或者上传到服务器，还可以进行异常统计，如哪个模块抛出的异常最多。有了这些统计我们就可以去修复这些模块。

## 2、自动强转

**public static void** main(String[] args) **throws** Exception {  
 Object obj = **new** String(**"abc"**);  
 String str = *autoConvert*(obj);  
}  
  
**private static** <T> T autoConvert(Object obj) {  
 **return** (T) obj;  
}

## 把任意类型的数组转换为对应类型的集合

需求是这个方法对任意类型的数组通用

**public static** <T> List<T> arrayToList(T[] array) {  
 **if** (array == **null**) **return null**;  
 List<T> list = **new** ArrayList<>();  
 **for** (T element: array) {  
 list.add(element);  
 }  
 **return** list;  
}

假设集合是需要别人传进来的，如：copy(List<T> list, T[] array)，使用泛型的好处是可以确保别人传进来的集合的类型和数组的类型一致。

## 把任意类型的集合转换为对应类型的数组

**public static** <T> T[] listToArray(List<T> list) {  
 **if** (list == **null** || list.isEmpty()) **return null**;  
 Class<?> clazz = list.get(0).getClass();  
 T[] array = (T[]) Array.*newInstance*(clazz, list.size());  
 **for** (**int** i = 0; i < list.size(); i++) {  
 array[i] = list.get(i);  
 }  
 **return** array;  
}

List<String> list = Arrays.*asList*(**"a"**, **"b"**, **"c"**);  
String[] array = *listToArray*(list);  
System.***out***.println(Arrays.*toString*(array));

## 交换任意类型数组中两个元素的位置

**public static void** main(String[] args) {  
 String[] strs = **new** String[] {**"a"**,**"b"**,**"c"**};  
 **int**[] nums = **new int**[] {1,2,3};  
 *swap*(strs,1,2);  
 //swap(nums,1,2);//这句代码会报错，虽然基本数据类型有包装类，但是现在是int[]，这是数组，不是基本类型，所以没法自动装箱，T不能接受int类型}  
  
**public static** <T> **void** swap(T[] array, **int** i, **int** j) {  
 T temp = array[i];  
 array[i] = array[j];  
 array[j] = temp;  
}

## 获取方法中的参数的泛型

框架经常就需要做这个事情，需要获取到泛型的类型是什么，然后创建该类型的对象。

泛型也有称之为签名，如定义在方法上的泛型称为方法上的签名，定义在类上的泛型，称为类上的签名。

List<User> list = **new** ArrayList<User>();

通过变量list本身是无法获得这个变量上的泛型参数是什么类型的，因为在编译后会去掉泛型信息，但是确实是有的框架能知道它是什么类型，他们是怎么做的呢？答：通过把变量声明在方法参数中，然后通过反射获取Method，再通过Method来获取到方法中的参数，拿到了参数就可以拿到参数中的泛型，示例如下：

**public void** fun(List<User> list) **throws** NoSuchMethodException {  
 Method method = getClass().getMethod(**"fun"**, List.**class**);  
 *// Type[] types = method.getParameterTypes(); // 获取方法的参数列表（不带泛型）* Type[] types = method.getGenericParameterTypes(); *// 获取方法的参数列表（带泛型）* ParameterizedType firstParam = (ParameterizedType) types[0];*// 获取方法中的第一个参数，并且这个参数是一个带泛型的参数* System.***out***.println(firstParam); *// 这就是参数，它包含原始类型和泛型* System.***out***.println(firstParam.getRawType()); *// 获取参数的原始类型（即不包括泛型）* System.***out***.println(firstParam.getActualTypeArguments()[0]); *// 获取参数中的第一个泛型，因为有可能有多个泛型，如Map<K,V>*}

打印结果如下：

java.util.List<com.evendai.kotlin.User> // 完整参数类型

interface java.util.List // 原始参数类型

class com.evendai.kotlin.User // 参数上的泛型

这是张老师讲课时的一个知识点，我不明白这个技术的应用场景，因为方法参数中写明了User，说明方法的创建人知道是User对象了，用的时候就直接声明User就可以了，为什么需要用反射去获取这么麻烦。

**public** <T> **void** fun(List<T> list) { }

在这个方法中应用上面的代码，然后调用：fun(new ArrayList<User>()) ，打印结果如下：

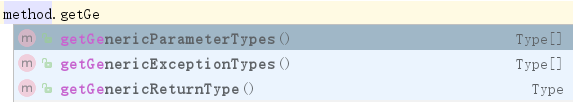
java.util.List<T>

interface java.util.List

T

拿到的泛型是T，其实我们想要的结果是User

对于Method，可以拿到方法的所有信息，如返回值类型，抛出的异常类型，如下：



对这3样内容，只能获取到已经有具体类型的泛型，如List<String>中的String是可以获取到的，而List<T>是不能获取到真实类型的，从Gson这个库就可知，如果可以的话，Gson这个这么强大的库肯定会去实现的，Gson的Json解析方法如下：



方法使用如下：



如果能获取到返回类型是User的话，则fromJson方法中就不需要传入User.class这个参数了。

## 解决泛型T不能使用intanceof的问题

**class** A { }  
**class** B **extends** A { }  
**class** C **extends** B { }  
  
**class** Util<T **extends** A> {  
 T **element**;  
 **public** Util(T element) { **this**.**element** = element; }  
  
 **public boolean** func(Object obj) {  
 **boolean** result;  
 *// result = obj instanceof T; // 编译失败，编译后T的类型是Object* Class<?> clazz = **element**.getClass();  
 result = clazz.isInstance(obj);  
 **return** result;  
 }  
}

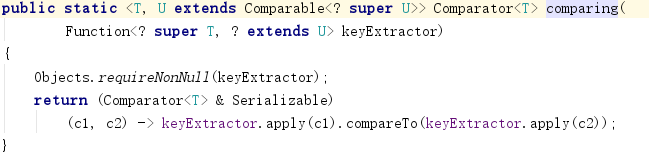
B student = **new** B();  
Util<B> util = **new** Util<>(student);  
  
System.***out***.println(util.func(**new** A()));*// false，相当于：A instanceof B*System.***out***.println(util.func(**new** B()));*// true, 相当于：B instanceof B*System.***out***.println(util.func(**new** C()));*// true, 相当于：C instanceof B*

## 看不懂的泛型用法

找名字最短的人

List<Person> list = Arrays.*asList*(**new** Person(**"黄蓉"**), **new** Person(**"西门吹雪"**), **new** Person(**"欧阳峰"**));  
Person p = list.stream()  
 .min(Comparator.*comparing*(person -> person.**name**.length()))  
 .get();  
System.***out***.println(p.**name**);

Comartor中的comparing方法如下：



# 我对泛型的一些理解

为什么泛型T可以用来声明变量却不能用来new，如下：

T t; // OK，这里编译器在编译成字节码时变成Object t;

new T() ; // 错误，这里不能把T当成Object，因为T这个类型可能没有空构造方法。

对于自动强转方法的理解 ：

**private static** <T> T autoConvert(Object obj) {  
 **return** (T) obj;  
}

User user = autoConvert(obj);

如果我们不使用泛型方法的话，自己写代码为：User user = (User) obj;

用了泛型方法后，是变成这样了吗：

**private static** <User> User autoConvert(Object obj) {  
 **return** (User) obj;  
}

不是的，编译器还是会把泛型编译成Object的，而且会把强转删除，如下：

**private static** <Object> Object autoConvert(Object obj) {  
 **return** obj;  
}

加上User user = autoConvert(obj); 的调用相当于：User user = obj;

虽然User user = obj; 看起来不合法，但是在字节码中，不会出现有强转的语句的。如Student stu = (Student) mPerson; 在编译成字节码后也是变成：Student stu = mPerson; 当然，在进行赋值时，编译器会判断左右两边的类型是否兼容，如果不兼容则会编译出一个检查类型是否兼容的字码，如果检查不兼容，则抛出类转换异常，如果检查兼容，则执行赋值语句Student stu = mPerson。

所以Student stu = mPerson这是OK的，只是编译器要警告你说“mPerson的类型可能和Student的类型不兼容，你必须检查一下以确保安全！”，那编译器怎么知道你去做了检查这个事情了呢？就是加上强转代码以表示你做了检查了。所以强转代码只是给编译器看的，编译器在编译成字节码时会直接把强转的代码丢掉。对于这个细节可参考相同目录下的“02\_泛型问题-深入理解类型擦除.doc”；

对于上面的User user = autoConvert(obj)相当于User user = obj;为什么编译不报错？ 给不给编译通过就是编译器一句话的问题，这没什么。

**总结就是编译器会把所有的泛型编译成Object类型，有一个例外，就是对于有上限的编译，编译器会把泛型编译成上限，如下：**

**class** Haha<P **extends** Person> {   
 P **p**;  
 **public** Haha(P p) { **this**.**p** = p; }   
 String get欢迎语() { **return p**.getName() + **", 你好!"** ; }   
}

编译后为：

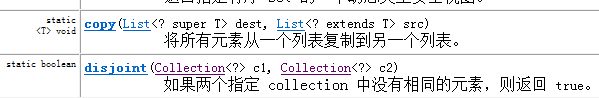
**class** Haha<Person> {   
 Person **p**;  
 **public** Haha(Person p) { **this**.**p** = p; }   
 String get欢迎语() { **return p**.getName() + **", 你好!"** ; }   
}

注：虽然把泛型P编译成了Person，但是在代码中不能new，因为我们可以传一个Person的子类，但是这个子类可能没有空构造方法，所以不能new。

# 泛型的学习

可以到JDK文档中去学习，如下：

Collections类的方法：



Collection类的方法：





