此文章已于 15:50:34 2017/11/20 重新发布到 HealerJean梦想博客

泛型

类别 无

# 1、泛型的标识

|  |
| --- |
| E - Element (在集合中使用，因为集合中存放的是元素)  T - Type（Java 类）  K - Key（键）  V - Value（值）  N - Number（数值类型）  ？ - 表示不确定的java类型 |

T表示泛型，new的时候要加入泛型，更方便通用

? 表示不确定的类型，一般用在通配

Object表示java中所有类的父类，在集合中使用时要格外注意。

jdk为了便于理解，用K表示键，V表示值，T表示type类型，E表示enum枚举，

其实这四个都只是符号，都是表示泛型名称。换成其他字母都没关系，但是都要在之前声明。

# 2、[java泛型中<?>和<T>有什么区别？](http://www.cnblogs.com/Vcanccc/p/5701351.html)

|  |
| --- |
| **public** **class** OneTestMain {    /\*\*  ? 的使用  \*/  **public** **static** **void** printColl(ArrayList<?> al){  Iterator<?> it = al.iterator();  **while**(it.hasNext())  {  System.*out*.println(it.next().toString());  }  }  /\*\*  T 的使用，可以直接当做方法使用哦，函数里面可以对T进行操作  \*/  **public** **static** <T> **void** printT(ArrayList<T> al){  Iterator<T> it = al.iterator();  **while**(it.hasNext())  {  T it1 = it.next();  System.*out*.println(it1.toString());  }  }      } |

# 3、泛型的简单实用List、ArrayList

//1、可以打印  
List list = new ArrayList();  
list.add(1);  
list.add("String");  
Iterator iterator = list.iterator();  
while (iterator.hasNext()){  
 System.***out***.println(iterator.next());//正常打印  
}

## 1、真正的反射概念

### 解释：

\* 所有反射的操作都是在运行时的，所以使用class来验证测试，

结果为：Java中编译后的class不会包含泛型信息

\* 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

\* 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。

|  |
| --- |
| **private** **static** **void** testEquals() {  ArrayList<String> a = **new** ArrayList<String>();  ArrayList b = **new** ArrayList();  Class c1 = a.getClass();  Class c2 = b.getClass();    System.*out*.println(c1 == c2); //true 表示已经经过编译了  System.*out*.println(a == b); //false  } |

## 2、利用反射调用arrylist

|  |
| --- |
| **private** **static** **void** invokeMethod() {  ArrayList<String> a = **new** ArrayList<String>();  Class c = a.getClass();    a.add("CSDN\_SEU\_Cavin");  **try**{  Method method = c.getMethod("add",Object.**class**);  method.invoke(a,100);  System.*out*.println(a); //[CSDN\_SEU\_Cavin, 100]  }**catch**(Exception e){  e.printStackTrace();  }  } |

# 4、自建反射对象的使用

## 1、具有泛型，T

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

## 2、new的实现，没有泛型，使用的是Object

|  |
| --- |
| **public** **class** ThreeGenericsNoUse {  **public** **static** **void** main(String[] args) {  FX intOb = **new** FX(**new** Integer(100));  intOb.showTyep();  System.*out*.println("value= " + intOb.getOb());  System.*out*.println("----------------------------------");    FX strOb = **new** FX("CSDN\_SEU\_Calvin");  strOb.showTyep();  System.*out*.println("value= " + strOb.getOb());  }  **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** showTyep() {  System.*out*.println("T的实际类型是: " + ob.getClass().getName());  }  }      } |

# 5、确定的泛型类型，传参必须确定类型才能成功

|  |
| --- |
| **public** **class** ThreeMain {  **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  }  //此行若把Number换为“？”或者“T” getData(ex\_int);编译通过  **public** **static** **void** getData(FX<Number> temp) {  //do something...  }    **public** **static** **class** FX<T> {  **private** T ob;  **public** FX(T ob) {  **this**.ob = ob;  }  }  } |

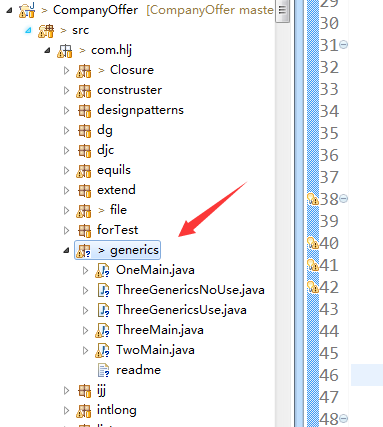
# 6、上下边界<? extends Number> 和下界FX<? supers Number>

## 1、<? extends Number> 表示 继承Number 的都能够使用

## 2、<? supers Number> 表示父类以上都能够使用

|  |
| --- |
| **public** **class** ThreeMain {  **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;  }  } |

# 7、代码位置



# 8、注意事项

## 1、不能对确切的泛型类型使用instanceof操作

|  |
| --- |
| **public** **class** FourMain {    **public** **static** **void** main(String[] args) {  FX<Number> ex\_num = **new** FX<Number>(100);  FX<Integer> ex\_int = **new** FX<Integer>(200);    /\* 报错，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。不确定的应该使用？  if(ex\_num instanceof FX<Number>){  }  \*/  **if**(ex\_num **instanceof** FX<?>){ //使用T也是错误的，以为T也是已知的java类型  }  }    **public** **static** **class** FX<T> {  **private** T ob;  **public** FX(T ob) {  **this**.ob = ob;  }  }  } |

# 9、泛型的使用

## 1、泛型的用法，可以强制转型

|  |
| --- |
| @Component  **public** **class** ApplicationContextHolder **implements** ApplicationContextAware,BeanPostProcessor {  /\*\*  \* Spring上下文  \*/  **private** **static** ApplicationContext *applicationContext*;    /\*\*  \* **@return**  \* **@Description**:  \* 获取 ApplicationContext  \*/  **public** **static** ApplicationContext getApplicationContext() {  **return** *applicationContext*;  }    /\*\*  \* **@param** <T>  \* **@param** name  \* **@return**  \* **@Description**:  \* 获取Bean  \*/  @SuppressWarnings("unchecked")  **public** **static** <T> T getBean(String name) {  **return** (T) *applicationContext*.getBean(name);  }    /\*\*  \* **@param** <T>  \* **@param** name  \* **@return**  \* **@Description**:  \* 获取Bean  \*/  **public** **static** <T> T getBean(Class<T> clazz) {  **return** (T) *applicationContext*.getBean(clazz);  }    /\*\*  \* **@param** <T>  \* **@param** name  \* **@return**  \* **@Description**:  \* 获取Bean  \*/  **public** **static** <T> T getBean(Class<T> clazz, String name) {  **return** (T) *applicationContext*.getBean(name, clazz);  }    @Override  **public** **void** setApplicationContext(ApplicationContext context) **throws** BeansException {  *applicationContext* = context;  }  @Override  **public** Object postProcessAfterInitialization(Object arg0, String arg1)  **throws** BeansException {  //  **return** arg0;  }  @Override  **public** Object postProcessBeforeInitialization(Object arg0, String arg1)  **throws** BeansException {  //  **return** arg0;  }  } |

## 2、泛型的使用

|  |
| --- |
| **private** AbstractOATransactionAssistantProcessor getProcessor(OAXMLHeadVO headerVo,String managercode ) {      AbstractOATransactionAssistantProcessor obj=**null**;  **try**{  obj=ApplicationContextHolder.*getBean*("assistantProcessor\_"+provinceName+"\_"+buscd.toUpperCase());  }**catch**(Exception ex){      }  **return** obj;  } |