**Java 泛型的约束与局限性**

 @author ixenos

* **不能用基本类型实例化类型参数**
  + 不能用类型参数代替基本类型：例如，没有Pair<double>,只有Pair<Double>，其原因是类型擦除。擦除之后，Pair类含有**Object类型的域**，而Object不能存储double值。这体现了Java语言中基本类型的独立状态。

* **运行时类型查询只适用于原始类型(raw type)**
  + **运行时**：通常指在Classloader装载之后，JVM执行之时
  + **类型查询**：instanceof、getClass、强制类型转换
  + **原始类型**：即（raw type）,**泛型类型**经编译器类型擦除后是Object或泛型参数的限定类型（例如Pair<T extends Comparable>，Comparable就是T的限定类型，转化后泛型的原始类型就是Comparable，所以Pair类不带泛型是Pair<Comparable>），即Pair类含有Comparable类型的域
  + JVM中没有泛型
  + eg:
  + if(a instanceof Pair<String>) //ERROR,仅测试了a是否是任意类型的一个Pair，会看到编译器ERROR警告
  + if(a instanceof Pair<T>) //ERROR
  + Pair<String> p = (Pair<String>) a;//WARNING,仅测试a是否是一个Pair
  + Pair<String> stringPair = ...;
  + Pair<Employee> employeePair = ...;

if(stringPair.getClass() == employeePair.getClass()) //会得到true，因为两次调用getClass都将返回Pair.class

* **不能创建参数化类型的数组(泛型数组)**
  + **参数化类型的数组**：指类型带有泛型参数的数组，也即泛型数组，如Pair<T>[] 、 T[]
  + 不能实例化参数化类型的数组，例如:
  + Pair<String> table = new Pair<String>[10]; //ERROR
    - 经**编译器**类型擦除后，table的类型是Pair[]，可以协变数组为Object[]，例如：//注意这里Pair是静态类型了，不是泛型参数，所以是Pair[]
    - Object[] objArray = table;
    - 数组会记住他的元素类型Pair，**如果试图存储其他类型的元素**，就会抛出异常（**数组存储检查**），例如：
    - objArray[0] = "Hello"; //ERROR--component type is Pair
    - 但是，对于**泛型类型**Pair<String>，类型擦除会使这种不同类检查机制无效，例如：
    - objArray[0] = new Pair<Employee>();
      * 数组存储只会检查擦除的类型，又因为Java语言设计数组可以**协变**，所以可以通过编译
      * 能够通过数组存储检查，不过仍会导致一个类型错误，故不允许创建参数化类型的数组
      * 注意，声明类型为Pair<String>[]的变量是合法的，只是不能创建这些实例（直接用new Pair<String>[10]来初始化这个变量）
  + **泛型数组的间接实现**：通过泛型数组包装器，如ArrayList类，维护一个Object数组，然后通过进出口方法set、get来限定类型和强制转换数组类型，从而间接实现泛型数组，例如：ArrayList: ArrayList<Pair<T>>、ArrayList<T>

* **不能实例化类型变量T**
  + 即不能使用new T(..) , new T[..] 或 T.class 这样的表达式中的类型变量
    - 例如： public Pair() { first = new T(); } //ERROR! **类型擦除**将T改变成Object，调用非本意的new Object()
  + 不能使用new T(..)
    - 但是，可通过反射调用Class.newInstance方法来构造泛型对象（要注意表达式T.class是非法的）
  + public static <T> Pair<T> makePair(Class<T> cl){
  + try{ return new Pair<>(cl.newInstance() , cl.newInstance()); }
  + catch(Exception ex) { return null; }
  + }
  + //这个方法可以按照下列方式调用：
  + Pair<String> p = Pair.makePair(String.class);
    - 注意：Class类本身是泛型。String.class是一个Class<String>的实例，因此makePair方法能够推断出pair的类型
  + 不能使用new T[..]
    - 即不能(直接)创建泛型数组，参考我另一篇博文(http://www.cnblogs.com/ixenos/p/5648519.html)，这里不再赘述
    - 解决方案：使用**泛型数组包装器**，例如ArrayList
      * 然而，当在设计一个泛型数组包装器时，例如方法minmax返回一个T[]数组，则泛型数组包装器无法施展，因为类型擦除，return (T [])new Object是没有意义的强转不了。此时只好利用反射，调用Array.newInstance：
        + import java.lang.reflect.\*;
        + ...
        + public static <T extends Comparable> T[] minmax(T... a){
        + T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType() , 2);
        + ...

}

【API文档描述】public Class<?> **getComponentType**() 返回表示数组组件类型的 Class。如果此类不表示数组类，则此方法返回 null。

* + - * 而ArrayList类中的toArray方法的实现就麻烦了
        + public Object[] toArray() 无参，返回Object[]数组即可
        + public Object[] toArray() {
        + return Arrays.copyOf(elementData, size);

}

【API文档描述】public static <T> T[] **copyOf**(T[] original,int newLength)

　　复制指定的数组，截取或用 null 填充（如有必要），以使副本具有指定的长度。对于在原数组和副本中都有效的所有索引，这两个数组将包含相同的值。对于在副本中有效而在原数组无效的所有索引，副本将包含 null。当且仅当指定长度大于原数组的长度时，这些索引存在。所得数组和原数组属于完全相同的类。

* + - * + public <T> T[] toArray(T[] a) a - 要存储列表元素的T[]数组（如果它足够大）否则分配一个具有相同**运行时类型**的新数组，返回该T[]数组
        + @SuppressWarnings("unchecked")
        + public <T> T[] toArray(T[] a) {
        + if (a.length < size)
        + // Make a new array of a's runtime type, but my contents:
        + return (T[]) Arrays.copyOf(elementData, size, a.getClass()); //a.getClass()得**运行时目的数组的运行时类型**
        + System.arraycopy(elementData, 0, a, 0, size);
        + if (a.length > size)
        + a[size] = null;
        + return a;
        + }

　【API文档描述】

public static <T,U> T[] **copyOf**(U[] original,int newLength, [Class](http://www.cnblogs.com/)<? extends T[]> newType)

复制指定的数组，截取或用 null 填充（如有必要），以使副本具有指定的长度。对于在原数组和副本中都有效的所有索引，这两个数组将包含相同的值。对于在副本中有效而在原数组无效的所有索引，副本将包含 null。当且仅当指定长度大于原数组的长度时，这些索引存在。所得数组属于 newType 类。

* **泛型类的静态上下文中类型变量无效**
  + 泛型类不能在静态域或静态方法中引用类型变量
  + public class Singleton<T>{
  + private static T singleInstance; //ERROR
  + public static T getSingleInstance(){...} //ERROR
  + }
    - 类型擦除后只剩下Singleton类，因为静态所以他只包含一个singleInstance域，如果能运行则以Singleton类为模板生成不同类型的域，因此产生了冲突
    - 故，在一个泛型类中禁止使用带有类型变量的静态域和静态方法

* **不能throws或catch泛型类的实例（有关异常）**
  + 泛型类继承Throwable类不合法，如public class Problem<T> extends Exception {...} //ERROR 不能通过编译
  + catch子句不能使用类型变量
    - public static <T extends Throwable> void doWork(Class<T> t){
    - try{
    - do work
    - }catch (T e){ // ERROR
    - Logger.global.info(...)
    - }
    - }

* + **不过，在异常规范中使用类型变量是允许的：**
    - public static <T extends Throwable> void doWork(T t) throws T { //此时可以throws T
    - try{
    - do work
    - }catch (Throwable realCause){ //捕获到具体实例
    - t.initCause(realCause);
    - throw t; //这时候抛具体实例，所以throw t 和 throws T 是可以的!
    - }
    - }
      * **此特性作用：可以利用泛型类、类型擦除、SuppressWarnings标注，来消除对已检查(checked)异常的检查，**
        + **unchecked和checked异常**: Java语言规范将派生于Error类或RuntimeException的所有异常称为**未检查(unchecked)异常**，其他的是**已检查(checked)异常**
        + **Java异常处理原则**：必须为所有已检查(checked)异常提供一个处理器，即一对一个，多对多个
        + @SuppressWarnings("unchecked") //**SuppressWarning标注**很关键，使得编译器认为T是unchecked异常从而不强迫为每一个异常提供处理器
        + public static <T extends Throwable> void throwAs(Throwable e) throws T{ //因为**泛型**和**类型擦除**，可以传递任意checked异常，例如RuntimeException类异常
        + throw (T) e;
        + }

假设该方法放在类Block中，如果调用 Block.<RuntimeException>throwAs(t); 编译器就会认为t是一个未检查的异常

public abstract class Block{

public abstract void body() throws Exception;

public Thread toThread(){

return new Thread(){

public void run(){

try{

body();

}catch(Throwable t){

Block.<RuntimeException>throwAs(t);

}

}

};

}

@SuppressWarnings("unchecked")

public static <T extends Throwable> void throwAs(Throwable e) throws T{

throw (T) e ;

}

}

再写个测试类

public class Test{

public static void main(String[] args){

new Block(){

public void body() throws Exception{

//不存在ixenos文件将产生IOException，checked异常！

Scanner in = new Scanner(new File("ixenos"));

while(in.hasNext())

System.out.println(in.next());

}

}.toThread().start();

}

}

启动线程后，throwAs方法将捕获线程run方法所有checked异常，“处理”成unchecked Exception(其实只是骗了编译器)后抛出；

**有什么意义**？正常情况下，因为run()方法声明为不抛出任何checked异常，所以必须捕获所有checked异常并“包装”到未检查的异常中；**意义**：而我们这样处理后，就不必去捕获所有并包装到unchecked异常中，我们只是抛出异常并“哄骗”了编译器而已

* + **注意擦除后的冲突**
    - Java泛型规范有个原则：“要想支持擦除的转换，就需要强行限制一**个泛型类或类型变量T不能同时成为两个接口类型的子类**，**而这两个接口是统一接口的不同参数化**”
      * 注意：非泛型类可以同时实现同一接口，毕竟没有泛型，很好处理
    - class Calender implements Comparable<Calender>{...}
    - class GGCalender extends Calender implements Comparable<GGCalender>{...} //ERROR

* + - * 在这里GGCalender类会同时实现Comparable<Calender> 和 Comparable<GGCalender>，这是同一接口的不同参数化