笔记2

12 章 泛型程序设计

泛型编程设计意味着编写的代码可以被很多不同类型的对象所重用

在java增加泛型之前 是实现继承来达到这一功能的

加这个意义是 使程序具有更好的安全性以及可读性

假如：
Arraylist<Manager>可以使用addall方法将Arraylist<Empoyer>这个集合全部添加过来 但是倒过来就不行 为了解决这个问题 提出了一个解决方法
类型参数 使得泛型程序员写出尽可能灵活的方法

当不同的泛型类混合在一起的时候 或是在于对类型参数一无所知的遗留的代码进行衔接时 可能会看到含糊不清的错误

关于实现泛型类的泛型程序员
对于类型参数 使用这段代码的程序员可能想要内置所有的类 他们希望在没有过多的现在以及混乱的错误信息的状态下 做所有的任务 因为 一个泛型程序员的任务就是预测所用类的未来可能有的所有用途
这一任务非常难 这里涉及到了一个新的概念 是**通配符**来进行通用的匹配

一个泛型类就是具有一个或多个类型变量的类

```
public class Pair<T> {


    private  T first;
    private  T second;

    public Pair() {


    }

public Pair(T first, T second) {
this.first = first;
        this.second = second;
}

public T getFirst() {
return first;
}

public void setFirst(T first) {
this.first = first;
}

public T getSecond() {
return second;
}

public void setSecond(T second) {
this.second = second;
}
}
```

这个就是一个典型的泛型类
主要是使用 了 T关键字

泛型可以有多个类型变量

列如：
public class Pair<T,U>{.....}

第一个域与第二个域可以使用不同的范围

类型模式使用大写形式 而且比较短 在java库使用变量E表示集合的元素类型 K和V分别表示表的关键字与值的类型 T（需要的时候还可以表现为U ,S） 表示为任意类型

```
Pair<String> pair=new Pair<>();
pair.setFirst("好");
pair.getFirst();
```

这样用具体的类型替换变量就可以实现实例化泛型类型
泛型类可以看做普通类的工厂

```
public class PairTest {

public static void main(String[] args) {
        String[] words={"MARY","HAD","A","LITTER","LAMB"};
Pair<String> mm=ArrayAlg.minmax(words);
System.out.println("min=  "+mm.getFirst());
System.out.println("max=  "+mm.getSecond());
}

}

class ArrayAlg{
public  static  Pair<String> minmax(String[] a){
if(a.length==0||a==null) return null;
String min=a[0];
String max=a[0];
       for (int i = 0; i <a.length ; i++) {
if(min.compareTo(a[i]) >0) min =a[i] ;
           if(max.compareTo(a[i]) <0) max =a[i];
}
return  new Pair<String>(min,max);
}

}
```

这里是一个列子 比较俩个字符串的compareto方法是个比较有意思的方法 是进行比较的是
java中的compareto方法，返回参与比较的前后两个字符串的asc码的差值

实际上我们可以定义一个带类型参数的简单方法

```
public  static  <T>  T getMiddle(T...a){

return a[a.length /2];
}
```

这个泛型方法可以定义在普通类中 也可以定义在泛型类中

类型变量的限制

```
class  ArrayAlg2{


publicstatic<T extendsComparable> T  min(T[] a)
{
if(a == null|| a.length==0)returnnull;
T smallest =a[0];
for(inti = 0; i <a.length ; i++)
if(smallest.compareTo(a[i]) >0) smallest=a[i];
returnsmallest;
```

```
}


}
```

这个extends关键字
表示T应该是绑定类型 T和绑定类型可以是类 也可以是接口 之所以选择extends关键字的原因是更接近子类的概念
一个类型变量或者通配符可以有多个限定
T extends Comparable & Serializable
限定类型用 & 分隔 而逗号用来分隔类型变量

可以有多个接口超类型 限定中**至多只有一个类** 如果用一个类进行限定 就必须是限定列表中的**第一个**

无论何时定义一个泛型类型 都自动提供了一个相应的原始类型（raw type） 原始类型的名字就是删去类型参数后的泛型类型名 擦除（erased）类型变量 并替换为限定类型 （无限定的变量用Object）
原始类型用第一个限定变量来替换 如果没有给出限定就用Object
什么是擦除 就是在编译之后 运行的时候 因为泛型只是**在编译有效的** 在编译即将到运行的时候 **编译会自动的翻译**
class Interval<Serializable &Comparable> 这样是一个泛型类 **原始类型用Serializable替换T 而编译器在必要的时候向Comparable插入强制类型转换** 为了**提高效率** 应该将标签**(tagging)接口(即没有方法的接口**)放在边界列表的末尾
泛型代码与虚拟机
虚拟机是没有泛型对象的 所有的对象都是普通类

https://segmentfault.com/a/1190000003831229关于java的泛型的 类型擦除

类型擦除也会出现在泛型方法中
方法擦除带来了问题
类型擦除有可能与多态发生了冲突
比如 在父类的泛型类中是设置为Date类型 然后被擦除之后 就会出现一个Object 而Object是不符合这个我们需要的
要解决这个问题 就必须编译器在这个类中生成一个桥方法

关于桥方法 下面的这就是一种
public void setSecond(Object second){setSecond((Date) second)}
在虚拟机中 用参数类型和返回类型来确定一个方法

总之需要记住的是
1 虚拟机中没有泛型 只有普通类和方法
2 所有的类型参数都用他们的限定类型来进行转换
3 桥方法来合成保持多态
4 为保持类型安全性 必要时插入强制类型转换

调用遗留代码 分为俩种情况 一种将这个参数传进来使用 一种是通过返回值来使用相关代码
http://blog.csdn.net/pacosonswjtu/article/details/50209729这里有相关的代码讲解
说实话就是抄书。。

约束与局限性
也就是我们在使用泛型的时候注意的地方

1 不能用基本类型类型实例化类型参数：没有Pair<double> 只有Pair<Double>
2 运行时类型检测只适用于原始类型
3 **不能创建参数化类型的数组**（涉及到一个擦除的问题）比如:Pair<String> [] table =new Pair<String>[10]; 擦除之后就变成了Pair[]
这样就有可能导致出错
只是不允许创建这些数组 而声明类型为Pair<String>[]的变量仍然是合法的 不过不能用new Pair<String>[10]初始化这个变量
4 Varargs警告
{
java本来是不能用泛型类型的数组的 向参数可变的方法加入一个泛型的实例

```
public static  <T> void addAll(Collection<T> coll,T ...ts){

for (T t:ts) coll.add(t);
}
```

这样就会出现一个关于泛型XX<Type>的数组了 这样是可以创建的
但是这样会出现警告 我们有俩种方法可以避免

1用的是

```
@SuppressWarnings("unchecked")
```

@SafeVarargs注解只能用在参数长度可变的方法或构造方法上，且方法必须声明为static或final，否则会出现编译错误。一个方法使用@SafeVarargs注解的前提是，开发人员必须确保这个方法的实现中对泛型类型参数的处理不会引发类型安全问题。
2 用的是关于Varargs警告方面的了

```
@SafeVarargs
public static  <T> void addAll(Collection<T> coll,T ...ts){

for (T t:ts) coll.add(t);
}
```

对于只需要读取参数数组元素的所有方法 都可以使用这个标注

但是需要注意这个警告隐藏着一个BUG 如果假如我们使用的时候是这么使用的

```
Pair<String> [] table=array(pair1,pair2);
Object[] objarray=table;
objarray[0]=new Pair<Employe>();
```

这样调用table[0]的时候就会出错 运行不会出错 （因为数组存储只会检查擦除的类型）
}

5 不能实例化类型对象
不能使用像new T(...)， new T[....]这样的表达式中的类型参数

但是可以利用Class.newInstance来构造泛型对象

```
public  static  <T>  Pair<T> makePair(Class<T> cl)
{
try{
return  new Pair<>(cl.newInstance(),cl.newInstance());
}catch (Exception e){
        e.printStackTrace();
}
}
```

然后这样传递进去就可以获得了实例对象 实际上 String.class就是Class<String>的实例
不能构造泛型数组
http://blog.csdn.net/PacosonSWJTU/article/details/50216303这里还是那个抄书的人的博客。。就当书看吧 比较方便

6 泛型类的静态上下文中类型变量无效
不能在静态域或方法中引用类型变量
private static T singleInstance;//error
具体原因还是因为类型擦除
7不能抛出或捕获泛型类的实例 甚至是不能抛出异常的 哪怕是Thowable也不行
Catch子句中是不能使用类型变量的
Catch（T e）//ERROR
不过
可以消除对已检查异常的检查

```
@SuppressWarnings("unchecked")
public  static  <T extends Throwable> void throwAs(Throwable e) throws  T{
throw (T) e;
}
```

```
try {

}catch (Throwable t){
    FanXinTest.<RuntimeException>throwAs(t);
}
}
```

编译器就会认为t是一个未检测的异常

```
public abstract class Blook {
```

```
public  abstract void body() throws Exception;

    public  Thread toThread()
    {
             return  new Thread()
        {
    public  void run()
            {
             try {
                   body();
            }catch (Throwable t){
                FanXinTest.<RuntimeException>throwAs(t);
             }
            }
        };
}
}
```

```
public static void main(String[] args) {
new Blook(){
@Override
public void body() throws Exception {
            Scanner  in=new Scanner(new File("ququx"));
            while (in.hasNext())
                System.out.println(in.next());
}
    }.toThread().start();
}
```

正常情况下 你必须捕获线程run方法中所有已检查的异常 把它们包装到未检测的异常中 因为run方法声明为不抛出任何已检查的异常
通过使用泛型类，擦除 和@SupperWarnings标注 就能消除java类型系统的部分基本限制

8注意擦除后的冲突
补救的办法是重新命名引发错误的方法
泛型规范说明 ：要想支持擦除的转换 就需要**强行限制一个类或类型变量不能同时成为俩个接口类型的子类** 而这俩个接口式同一接口的不同参数化
至于原因 就非常巧妙了 有可能是合成的桥方法冲突

泛型类型的继承规则
无论S与T有什么关系 通常 Pair<S>与Pair<T>之间没有什么联系
泛型与数组是有区别的 如果S与T之间存在着继承的关系的话 如果试图将低的S存到高的T中 会自动报异常

永远都可以将参数化类型装换为一个原始类型
但是转换为原始类型之后 会出现类型错误（失去了泛型设计提供的附加安全性）

泛型类可以扩展为其他的泛型类 比如 Arraylist<T>转换为List<T>

通配符类型
比如：Pair<? extends Employee>表示任何泛型Pair 类型
我们写一个表示雇员的方法

```
public  static  void printBuddies(Pair<Employee> p){
    Employee first=p.getFirst();
Employee second=p.getSecond();
System.out.println(first.getFirst()+"and"+second.getSecond()+"are buddies");

}
```

Pair<Manager>本来是不能传进这个方法 这一点很受限制 但是 解决的方法很简单 就是使用通配符

```
public  static  void printBuddies(Pair<? extends Employee> p){
```

使用通配符会通过Pair<? extends Employee>（具有任何从Employee继承类型的列表）
，编译器无法确定Pair所持有的类型，所以无法安全的向其中添加对象
的引用有可能破坏Pair<Manager>
使用setFirst的时候会出现错误 因为编译器只知道是需要某个Employee子类型 但不知道具体是什么类型 ？不能用来匹配 所以出错
但是使用getFirst方法 就不会出现这个问题

通配符的超类型限定 就也是**<? super ClassName> 这个通配符限制为Classname的所有超类型**

这个与上面的用法正好相反 比如 <? super manager>(具有任何manager超类型的列表)
列表的类型至少是一个 manager类型，因此可以安全的向其中添加manager及其子类型。
setFIrst方法编译器不知道具体的类型 但是可以被任意的Manager的子类调用
但是不能被Employee的类调用 因为这个是属于Manager的父类
getFirst就是不能保证返回对象类型 只能赋给一个Object

带有超类型限定的可以向泛型类型对象写入 带有子类型限定的通配符可以从泛型对象读取

```
public interface Comparable<T> {

public  int compareTo(T other);

}
```

假如我们声明这么一个接口

```
public  static  <T extends  Comparable> T min(T[] a){
```

一 般是这样的 但是我们可以这样

```
public  static  <T extends  Comparable<T>> T min(T[] a){
```

这样更好 更彻底 工作的更好 但是对于一个（假如有一个类是A A继承于B B实现了Compareable<B> 因此 A实现的是Compareable<B> 而不是Compareable<A>）
这样的就会出现问题 我们可以这个时候使用一个超类型来进行救助

```
public  static  <T extends  Comparable< ? super T>> Tmin(T[] a){
```

这样就完美了 足够安全
这个声明的意义在于为程序员排除调用参数的不必要限制

无限定的通配符
pair<?>就是一个无限定的通配符 get方法只能返回一个Object set方法只能不能被调用（严格意义上只能使用setxx（ null ）） 这个的目的就是来进行检查一个pair是否包含一个null引用

关于参数类型与通配符的区别http://www.ciaoshen.com/2016/08/21/wildcards/

通配符捕获

通配符不是类型变量 因此 不能在编写代码中使用"?"作为一种类型
？ t=p.getFirst();
这是非法的

```
public  static  <T> void swap(Pair<?> p )
```

这是个编写pair元素交换的方法 但是因为？不能作为一个类型 所以我们需要一个辅助方法
swapHelper 这是一个泛型方法 而上面的swap方法不是泛型方法 它具有固定的Pair<?>类型参数 **由swap调用swapHelper**

```
public  static  <T> void swapHelper(Pair<T> p){
T  t=p.getFirst();
p.setFirst(p.getSecond());
p.setSecond(t);

}
```

swapHelper方法的参数捕获通配符 它不知道是哪种类型的通配符 但是 这是一个明确的类型
并且<T>swapHelper的定义只有在T指出类型时才有明确的含义
通配符捕获机制在许多情况满足的情况下才是合法的
编译器必须能够确信通配符表达的是单个确定的类型 比如 Arraylist<Pair<T>>中的T永远不能捕获ArrayList<Pair<?>>的通配符
数组列表可以保存俩个Pair<?> 分别针对?的不同类型
反射与泛型
在反射中
列如在 Class<T>（ Class类是泛型的）【【【
【
类型参数是十分有用的 这是因为它允许Class<T> 方法的返回类型更加具有针对性

http://www.cnblogs.com/whitewolf/p/4355541.html这里有一个博客（反射处理java泛型）

在这本书里涉及到了几个方法

T newInstance()
返回默认构造器的一个新实例

T cast(Object obj)
如果obj为null 或有可能转换类型T 则返回Obj 否则就抛出BadCastException

T[] getEnumConstants()
如果T是枚举类型 则返回所有值组成的数组 否则返回null

Class<? super T>getSuperclass()
返回这个类的超类 如果T不是一个类或Object类 则返回NULL

Constructor<T> getConstructor(Class ...parameterTypes)
Constructor<T> getDeclaredConstructor<Class ...parameterTypes>
获得公有的构造器 或带有给定参数类型的构造

T newInstance(Object ...parameters)
返回用指定参数构造的新实例

使用Class<T>参数进行类型匹配
有时 匹配泛型方法中的Class<T>参数的类型变量将会变得很有实用价值

```
public static  <T> Pair<T> makePair(Class<T> c)throws
InstantiationException ,IllegalAccessException{

return  new Pair<>(c.newInstance(),c.newInstance());
}
```

```
makePair(Employee.class);
```

这样就能返回一个Pair<Employee>

虚拟机的泛型类型信息
虚拟机对泛型的擦除并不是特别的干净 还保留着一些泛型祖先的简单信息
可以用反射API来确定一些关于泛型方法的擦除之前的信息
具体原因如下：
虚拟机需要重新构造实现这声明的泛型类以及方法中的所有内容
但是 **不会知道对于特定的对象或方法调用（ 所以被反射API获取 ） 如何解释类型参数**
这些信息是包含在类文件中的 这些泛型反射可用的类型信息是旧的虚拟机不兼容的 所以才会被获取

<1>泛型参数类型限制

(1) 泛型的**参数类型只能是类（ class ）类型，而不能是简单类型。**

(2) 可以声明多个泛型参数类型，比如 <T, P,Q…>，同时还可以**嵌套泛型**，例如： <List<String>>

(3) 泛型 的参数 类 型可以使用 extends 语 句，例如 <T extends superclass> 。

(4) 泛型的参数类型可以使用 super 语句，例如 < T super childclass> 。

(5) 泛型还可以使用通配符，例如 <? e xtends ArrayList>

<E extends ClassA>表示后续都只能使用E进行某些判断或操作，而<? extends ClassA>? 表示后续使用时可以是任意的。

参数化类型是不可变的。所以我们的通配符类型就派上用场了
http://ifeve.com/difference-between-super-t-and-extends-t-in-java/ 关于泛型中 ？super T 和 ？ extends T的区别