# 目录

- 个人公众号：黄小斜

# 泛型概述

泛型在java中有很重要的地位，在面向对象编程及各种设计模式中有非常广泛的应用。
什么是泛型？为什么要使用泛型？
https://docs.oracle.com/javase/tutorial/java/generics/why.html

> 泛型，即"参数化类型"。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。
>
> 泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

## 一个栗子

一个被举了无数次的例子：

```
List arrayList = new ArrayList();
arrayList.add("aaaa");
arrayList.add(100);
```

```
for(int i = 0; i< arrayList.size();i++){
    String item = (String)arrayList.get(i);
    Log.d("泛型测试","item = " + item);
}
```

毫无疑问，程序的运行结果会以崩溃结束：

java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

ArrayList可以存放任意类型，例子中添加了一个String类型，添加了一个Integer类型，再使用时都以String的方式使用，因此程序崩溃了。为了解决类似这样的问题（在编译阶段就可以解决），泛型应运而生。

我们将第一行声明初始化list的代码更改一下，编译器会在编译阶段就能够帮我们发现类似这样的问题。

List arrayList = new ArrayList(); ... //arrayList.add(100); 在编译阶段，编译器就会报错

### 特性

泛型只在编译阶段有效。看下面的代码：

```
List<String> stringArrayList = new ArrayList<String>();
List<Integer> integerArrayList = new ArrayList<Integer>();

Class classStringArrayList = stringArrayList.getClass();
Class classIntegerArrayList = integerArrayList.getClass();

if(classStringArrayList.equals(classIntegerArrayList)){
    Log.d("泛型测试","类型相同");
}
```

> 通过上面的例子可以证明，在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。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行时阶段。

对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

## 泛型的使用方式

泛型有三种使用方式，分别为：泛型类、泛型接口、泛型方法

### 泛型类

> 泛型类型用于类的定义中，被称为泛型类。通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Set、Map。
>
> 泛型类的最基本写法（这么看可能会有点晕，会在下面的例子中详解）：

```
class 类名称 <泛型标识：可以随便写任意标识号，标识指定的泛型的类型>{
  private 泛型标识 /*（成员变量类型）*/ var;
  .....

  }
```

一个最普通的泛型类：

//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型

```
//在实例化泛型类时，必须指定T的具体类型
public class Generic<T>{
    //在类中声明的泛型整个类里面都可以用，除了静态部分，因为泛型是实例化时声明的。
    //静态区域的代码在编译时就已经确定，只与类相关
    class A <E>{
        T t;
    }
    //类里面的方法或类中再次声明同名泛型是允许的，并且该泛型会覆盖掉父类的同名泛型T
    class B <T>{
        T t;
    }
    //静态内部类也可以使用泛型，实例化时赋予泛型实际类型
    static class C <T> {
        T t;
    }
    public static void main(String[] args) {
        //报错，不能使用T泛型，因为泛型T属于实例不属于类
//        T t = null;
    }

    //key这个成员变量的类型为T,T的类型由外部指定
    private T key;

    public Generic(T key) { //泛型构造方法形参key的类型也为T，T的类型由外部指定
        this.key = key;
    }

    public T getKey(){ //泛型方法getKey的返回值类型为T，T的类型由外部指定
        return key;
    }
}
```

> 12-27 09:20:04.432 13063-13063/? D/泛型测试: key is 123456

> 12-27 09:20:04.432 13063-13063/? D/泛型测试: key is key_vlaue

> 定义的泛型类，就一定要传入泛型类型实参么？并不是这样，在使用泛型的时候如果传入泛型实参，则会根据传入的泛型实参做相应的限制，此时泛型才会起到本应起到的限制作用。如果不传入泛型类型实参的话，在泛型类中使用泛型的方法或成员变量定义的类型可以为任何的类型。

看一个例子：

```
Generic generic = new Generic("111111");
Generic generic1 = new Generic(4444);
Generic generic2 = new Generic(55.55);
Generic generic3 = new Generic(false);

Log.d("泛型测试","key is " + generic.getKey());
Log.d("泛型测试","key is " + generic1.getKey());
Log.d("泛型测试","key is " + generic2.getKey());
Log.d("泛型测试","key is " + generic3.getKey());

D/泛型测试: key is 111111
D/泛型测试: key is 4444
D/泛型测试: key is 55.55
D/泛型测试: key is false
```

注意：泛型的类型参数只能是类类型，不能是简单类型。 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。 if(ex_num instanceof Generic){
}

## 泛型接口

泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中，可以看一个例子：

```
//定义一个泛型接口
public interface Generator<T> {
    public T next();
}
```

当实现泛型接口的类，未传入泛型实参时：

```
/**
 * 未传入泛型实参时，与泛型类的定义相同，在声明类的时候，需将泛型的声明也一起加到类中
 * 即：class FruitGenerator<T> implements Generator<T>{
 * 如果不声明泛型，如：class FruitGenerator implements Generator<T>，编译器会报错："Unknown class"
 */
class FruitGenerator<T> implements Generator<T>{
    @Override
```

```
    public T next() {
        return null;
    }
}
```

当实现泛型接口的类，传入泛型实参时：

```
/**
 * 传入泛型实参时：
 * 定义一个生产器实现这个接口,虽然我们只创建了一个泛型接口Generator<T>
 * 但是我们可以为T传入无数个实参，形成无数种类型的Generator接口。
 * 在实现类实现泛型接口时，如已将泛型类型传入实参类型，则所有使用泛型的地方都要替换成传入的实参类型
 * 即：Generator<T>，public T next();中的的T都要替换成传入的String类型。
 */
public class FruitGenerator implements Generator<String> {

    private String[] fruits = new String[]{"Apple", "Banana", "Pear"};

    @Override
    public String next() {
        Random rand = new Random();
        return fruits[rand.nextInt(3)];
    }
}
```

## 泛型通配符

我们知道Ingeter是Number的一个子类，同时在特性章节中我们也验证过Generic与Generic实际上是相同的一种基本类型。那么问题来了，在使用Generic作为形参的方法中，能否使用Generic的实例传入呢？在逻辑上类似于Generic和Generic是否可以看成具有父子关系的泛型类型呢？

为了弄清楚这个问题，我们使用Generic这个泛型类继续看下面的例子：

```
public void showKeyValue1(Generic<Number> obj){
    Log.d("泛型测试","key value is " + obj.getKey());
}

Generic<Integer> gInteger = new Generic<Integer>(123);
Generic<Number> gNumber = new Generic<Number>(456);

showKeyValue(gNumber);

// showKeyValue这个方法编译器会为我们报错：Generic<java.lang.Integer>
// cannot be applied to Generic<java.lang.Number>
// showKeyValue(gInteger);
```

通过提示信息我们可以看到Generic不能被看作为`Generic的子类。由此可以看出:同一种泛型可以对应多个版本（因为参数类型是不确定的），不同版本的泛型类实例是不兼容的。

回到上面的例子，如何解决上面的问题？总不能为了定义一个新的方法来处理Generic类型的类，这显然与java中的多台理念相违背。因此我们需要一个在逻辑上可以表示同时是Generic和Generic父类的引用类型。由此类型通配符应运而生。

我们可以将上面的方法改一下：

```
public void showKeyValue1(Generic<?> obj){
    Log.d("泛型测试","key value is " + obj.getKey());
```

类型通配符一般是使用？代替具体的类型实参，注意， 此处的？和Number、String、Integer一样都是一种实际的类型，可以把？看成所有类型的父类。是一种真实的类型。

可以解决当具体类型不确定的时候，这个通配符就是？； 当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用？通配符来表未知类型

public void showKeyValue(Generic obj){ System.out.println(obj); }

```
Generic<Integer> gInteger = new Generic<Integer>(123);
Generic<Number> gNumber = new Generic<Number>(456);

public void test () {
//        showKeyValue(gInteger);该方法会报错
    showKeyValue1(gInteger);
}

public void showKeyValue1(Generic<?> obj) {
    System.out.println(obj);
}
// showKeyValue这个方法编译器会为我们报错：Generic<java.lang.Integer>
// cannot be applied to Generic<java.lang.Number>
// showKeyValue(gInteger);
```

。

## 泛型方法

在java中,泛型类的定义非常简单，但是泛型方法就比较复杂了。

尤其是我们见到的大多数泛型类中的成员方法也都使用了泛型，有的甚至泛型类中也包含着泛型方法，这样在初学者中非常容易将泛型方法理解错了。 泛型类，是在实例化类的时候指明泛型的具体类型；泛型方法，是在调用方法的时候指明泛型的具体类型 。

```
/**
 * 泛型方法的基本介绍
 * @param tClass 传入的泛型实参
 * @return T 返回值为T类型
 * 说明：
 *     1）public 与 返回值中间<T>非常重要，可以理解为声明此方法为泛型方法。
 *     2）只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型
方法。
 *     3）<T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。
 *     4）与泛型类的定义一样，此处T可以随便写为任意标识，常见的如T、E、K、V等形式
的参数常用于表示泛型。
 */
    public <T> T genericMethod(Class<T> tClass)throws InstantiationException ,
      IllegalAccessException{
            T instance = tClass.newInstance();
            return instance;
    }

Object obj = genericMethod(Class.forName("com.test.test"));
```

## 泛型方法的基本用法

光看上面的例子有的同学可能依然会非常迷糊，我们再通过一个例子，把我泛型方法再总结一下。

```
/**
 * 这才是一个真正的泛型方法。
 * 首先在public与返回值之间的<T>必不可少，这表明这是一个泛型方法，并且声明了一个泛
型T
 * 这个T可以出现在这个泛型方法的任意位置.
 * 泛型的数量也可以为任意多个
 *    如：public <T,K> K showKeyName(Generic<T> container){
 *        ...
 *        }
 */

    public class 泛型方法 {
    @Test
    public void test() {
        test1();
        test2(new Integer(2));
        test3(new int[3],new Object());

        //打印结果
//        null
//        2
//        [I@3d8c7aca
//        java.lang.Object@5ebec15
    }
    //该方法使用泛型T
    public <T> void test1() {
```

```
        T t = null;
        System.out.println(t);
    }
    //该方法使用泛型T
    //并且参数和返回值都是T类型
    public <T> T test2(T t) {
        System.out.println(t);
        return t;
    }

    //该方法使用泛型T,E
    //参数包括T,E
    public <T, E> void test3(T t, E e) {
        System.out.println(t);
        System.out.println(e);
    }
}
```

## 类中的泛型方法

当然这并不是泛型方法的全部，泛型方法可以出现杂任何地方和任何场景中使用。但是有一种情况是非常特殊的，当泛型方法出现在泛型类中时，我们再通过一个例子看一下

```
//注意泛型类先写类名再写泛型，泛型方法先写泛型再写方法名
//类中声明的泛型在成员和方法中可用
class A <T, E>{
    {
        T t1 ;
    }
    A (T t){
        this.t = t;
    }
    T t;

    public void test1() {
        System.out.println(this.t);
    }

    public void test2(T t,E e) {
        System.out.println(t);
        System.out.println(e);
    }
}
@Test
public void run () {
    A <Integer,String > a = new A<>(1);
    a.test1();
    a.test2(2,"ds");
//        1
//        2
```

```
//        ds
}

static class B <T>{
    T t;
    public void go () {
        System.out.println(t);
    }
}
```

## 泛型方法与可变参数

再看一个泛型方法和可变参数的例子：

```
public class 泛型和可变参数 {
    @Test
    public void test () {
        printMsg("dasd",1,"dasd",2.0,false);
        print("dasdas","dasdas", "aa");
    }
    //普通可变参数只能适配一种类型
    public void print(String ... args) {
        for(String t : args){
            System.out.println(t);
        }
    }
    //泛型的可变参数可以匹配所有类型的参数。。有点无敌
    public <T> void printMsg( T... args){
        for(T t : args){
            System.out.println(t);
        }
    }
        //打印结果：
    //dasd
    //1
    //dasd
    //2.0
    //false

}
```

## 静态方法与泛型

静态方法有一种情况需要注意一下，那就是在类中的静态方法使用泛型：静态方法无法访问类上定义的泛型；如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。

即：如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法 。

```
public class StaticGenerator<T> {
    ....
    ....
    /**
     * 如果在类中定义使用泛型的静态方法，需要添加额外的泛型声明（将这个方法定义成泛型方法）
     * 即使静态方法要使用泛型类中已经声明过的泛型也不可以。
     * 如：public static void show(T t){..},此时编译器会提示错误信息：
          "StaticGenerator cannot be refrenced from static context"
     */
    public static <T> void show(T t){

    }
}
```

# 泛型方法总结

泛型方法能使方法独立于类而产生变化，以下是一个基本的指导原则：

无论何时，如果你能做到，你就该尽量使用泛型方法。也就是说，如果使用泛型方法将整个类泛型化，那么就应该使用泛型方法。另外对于一个static的方法而已，无法访问泛型类型的参数。所以如果static方法要使用泛型能力，就必须使其成为泛型方法。

# 泛型上下边界

在使用泛型的时候，我们还可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。

为泛型添加上边界，即传入的类型实参必须是指定类型的子类型

```
public class 泛型通配符与边界 {
    public void showKeyValue(Generic<Number> obj){
        System.out.println("key value is " + obj.getKey());
    }
    @Test
    public void main() {
        Generic<Integer> gInteger = new Generic<Integer>(123);
        Generic<Number> gNumber = new Generic<Number>(456);
        showKeyValue(gNumber);
        //泛型中的子类也无法作为父类引用传入
//        showKeyValue(gInteger);
    }
    //直接使用？通配符可以接受任何类型作为泛型传入
    public void showKeyValueYeah(Generic<?> obj) {
        System.out.println(obj);
    }
    //只能传入number的子类或者number
    public void showKeyValue1(Generic<? extends Number> obj){
```

```
        System.out.println(obj);
    }

    //只能传入Integer的父类或者Integer
    public void showKeyValue2(Generic<? super Integer> obj){
        System.out.println(obj);
    }

    @Test
    public void testup () {
        //这一行代码编译器会提示错误，因为String类型并不是Number类型的子类
        //showKeyValue1(generic1);
        Generic<String> generic1 = new Generic<String>("11111");
        Generic<Integer> generic2 = new Generic<Integer>(2222);
        Generic<Float> generic3 = new Generic<Float>(2.4f);
        Generic<Double> generic4 = new Generic<Double>(2.56);

        showKeyValue1(generic2);
        showKeyValue1(generic3);
        showKeyValue1(generic4);
    }

    @Test
    public void testdown () {

        Generic<String> generic1 = new Generic<String>("11111");
        Generic<Integer> generic2 = new Generic<Integer>(2222);
        Generic<Number> generic3 = new Generic<Number>(2);
//        showKeyValue2(generic1);本行报错，因为String并不是Integer的父类
        showKeyValue2(generic2);
        showKeyValue2(generic3);
    }
}
```

== 关于泛型数组要提一下 ==

看到了很多文章中都会提起泛型数组，经过查看sun的说明文档，在java中是"不能创建一个确切的泛型类型的数组"的。

> 也就是说下面的这个例子是不可以的：
>
> ```
> List<String>[] ls = new ArrayList<String>[10];
> ```
>
> 而使用通配符创建泛型数组是可以的，如下面这个例子：
>
> ```
> List<?>[] ls = new ArrayList<?>[10];
> ```
>
> 这样也是可以的：
>
> ```
> List<String>[] ls = new ArrayList[10];
> ```

下面使用Sun的一篇文档的一个例子来说明这个问题：

```
List<String>[] lsa = new List<String>[10]; // Not really allowed.
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // Unsound, but passes run time store check
String s = lsa[1].get(0); // Run-time error: ClassCastException.
```

> 这种情况下，由于JVM泛型的擦除机制，在运行时JVM是不知道泛型信息的，所以可以给oa[1]赋上一个ArrayList而不会出现异常，但是在取出数据的时候却要做一次类型转换，所以就会出现ClassCastException，如果可以进行泛型数组的声明，上面说的这种情况在编译期将不会出现任何的警告和错误，只有在运行时才会出错。
>
> 而对泛型数组的声明进行限制，对于这样的情况，可以在编译期提示代码有类型安全问题，比没有任何提示要强很多。下面采用通配符的方式是被允许的:数组的类型不可以是类型变量，除非是采用通配符的方式，因为对于通配符的方式，最后取出数据是要做显式的类型转换的。

```
List<?>[] lsa = new List<?>[10]; // OK, array of unbounded wildcard type.
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // Correct.
Integer i = (Integer) lsa[1].get(0); // OK
```

最后

本文中的例子主要是为了阐述泛型中的一些思想而简单举出的，并不一定有着实际的可用性。另外，一提到泛型，相信大家用到最多的就是在集合中，其实，在实际的编程过程中，自己可以使用泛型去简化开发，且能很好的保证代码质量。

## 泛型常见面试题

1. Java中的泛型是什么？使用泛型的好处是什么?

这是在各种Java泛型面试中，一开场你就会被问到的问题中的一个，主要集中在初级和中级面试中。那些拥有Java1.4或更早版本的开发背景的人 都知道，在集合中存储对象并在使用前进行类型转换是多么的不方便。泛型防止了那种情况的发生。它提供了编译期的类型安全，确保你只能把正确类型的对象放入 集合中，避免了在运行时出现ClassCastException。

2. Java的泛型是如何工作的？什么是类型擦除？

这是一道更好的泛型面试题。泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如 List在运行时仅用一个List来表示。这样做的目的，是确保能和Java 5之前的版本开发二进制类库进行兼容。你无法在运行时访问到类型参数，因为编译器已经把泛型类型转换成了原始类型。根据你对这个泛型问题的回答情况，你会 得到一些后续提问，比如为什么泛型是由类型擦除来实现的或者给你展示一些会导致编译器出错的错误泛型代码。请阅读我的Java中泛型是如何工作的来了解更 多信息。

3. 什么是泛型中的限定通配符和非限定通配符 ?

这是另一个非常流行的Java泛型面试题。限定通配符对类型进行了限制。有两种限定通配符，一种是它通过确保类型必须是T的子类来设定类型的上界，另一种是它通过确保类型必须是T的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面表 示了非限定通配符，因为<?>可以用任意类型来替代。更多信息请参阅我的文章泛型中限定通配符和非限定通配符之间的区别。

4. List<? extends T>和List <? super T>之间有什么区别 ?

这和上一个面试题有联系，有时面试官会用这个问题来评估你对泛型的理解，而不是直接问你什么是限定通配符和非限定通配符。这两个List的声明都是 限定通配符的例子，List<? extends T>可以接受任何继承自T的类型的List，而List<? super T>可以接受任何T的父类构成的List。例如List<? extends Number>可以接受List或List。在本段出现的连接中可以找到更多信息。

5. 如何编写一个泛型方法，让它能接受泛型参数并返回泛型类型?

编写泛型方法并不困难，你需要用泛型类型来替代原始类型，比如使用T, E or K,V等被广泛认可的类型占位符。泛型方法的例子请参阅Java集合类框架。最简单的情况下，一个泛型方法可能会像这样:

```
public V put(K key, V value) {

return cache.put(key, value);

}
```

6. Java中如何使用泛型编写带有参数的类?

这是上一道面试题的延伸。面试官可能会要求你用泛型编写一个类型安全的类，而不是编写一个泛型方法。关键仍然是使用泛型类型来代替原始类型，而且要使用JDK中采用的标准占位符。

7. 编写一段泛型程序来实现LRU缓存?

对于喜欢Java编程的人来说这相当于是一次练习。给你个提示，LinkedHashMap可以用来实现固定大小的LRU缓存，当LRU缓存已经满 了的时候，它会把最老的键值对移出缓存。LinkedHashMap提供了一个称为removeEldestEntry()的方法，该方法会被put() 和putAll()调用来删除最老的键值对。当然，如果你已经编写了一个可运行的JUnit测试，你也可以随意编写你自己的实现代码。

8. 你可以把List传递给一个接受List参数的方法吗?

   对任何一个不太熟悉泛型的人来说，这个Java泛型题目看起来令人疑惑，因为乍看起来String是一种Object，所以 List应当可以用在需要List的地方，但是事实并非如此。真这样做的话会导致编译错误。如 果你再深一步考虑，你会发现Java这样做是有意义的，因为List可以存储任何类型的对象包括String, Integer等等，而List却只能用来存储Strings。

   List objectList;

   List stringList;

   objectList = stringList; //compilation error incompatible types

   ix. Array中可以用泛型吗?

   这可能是Java泛型面试题中最简单的一个了，当然前提是你要知道Array事实上并不支持泛型，这也是为什么Joshua Bloch在Effective Java一书中建议使用List来代替Array，因为List可以提供编译期的类型安全保证，而Array却不能。

   x. 如何阻止Java中的类型未检查的警告?

   如果你把泛型和原始类型混合起来使用，例如下列代码，Java 5的javac编译器会产生类型未检查的警告，例如

   List rawList = new ArrayList()

   注意: Hello.java使用了未检查或称为不安全的操作;

   这种警告可以使用@SuppressWarnings("unchecked")注解来屏蔽。

## 参考文章


https://www.cnblogs.com/huajiezh/p/6411123.html
https://www.cnblogs.com/jpfss/p/9929045.html
https://www.cnblogs.com/dengchengchao/p/9717097.html
https://www.cnblogs.com/cat520/p/9353291.html
https://www.cnblogs.com/coprince/p/8603492.html


## 微信公众号

远在 JDK 1.4 版本的时候，那时候是没有泛型的概念的，如果使用 Object 来实现通用、不同类型的处理，有这么两个缺点：

1 每次使用时都需要程序员手动强制转换成想要的类型
2. 编译时编译器并不知道类型转换是否正常(编译成字节码阶段)，运行时才知道（运行字节码），不安全。如这个例子：

```
List list = new ArrayList();
list.add("www.cnblogs.com");
list.add(23);
String name = (String)list.get(0);
String number = (String)list.get(1);     //ClassCastException
```

上面的代码在运行时会发生强制类型转换异常。这是因为我们在存入的时候，第二个是一个 Integer 类型，但是取出来的时候却将其强制转换为 String 类型了。Sun 公司为了使 Java 语言更加安全，减少运行时异常的发生。于是在 JDK 1.5 之后推出了泛型的概念。

根据《Java 编程思想》中的描述，泛型出现的动机在于：有许多原因促成了泛型的出现，而最引人注意的一个原因，就是为了创建容器类。泛型使用场景：容器类中编译阶段提供类型安全检查

使用泛型的好处有以下几点：

https://docs.oracle.com/javase/tutorial/java/generics/why.html

1. 类型安全

在编译时进行更强的类型检查，如果代码违反类型安全，则会发出错误，泛型的主要目标是提高 Java 程序的类型安全，编译时期就可以检查出因 Java 类型不正确导致的 ClassCastException 异常。

2. 消除强制类型转换. 泛型的一个附带好处是，使用时直接得到目标类型，消除许多强制类型转换

```
List list= new ArrayList（）；
list.add（“hello”）；
String s = （String） list.get（0）；
```

使用泛型时，代码不需要强制转换：

```
List <String> list = new ArrayList <String>（）；
list.add（“hello”）；
String s = list.get（0）；
```

反射跳过泛型

```
List<Integer> list = new ArrayList<>();

list.add(12);
//这里直接添加会报错
list.add("a");
Class<? extends List> clazz = list.getClass();
Method add = clazz.getDeclaredMethod("add", Object.class);
//但是通过反射添加，是可以的
```

反射是在运行时执行方法、修改属性等，跳过泛型的编译器检查

```
add.invoke(list, "kl");

System.out.println(list)
```

> 写在前面:最近在看泛型,研究泛型的过程中,发现了一个比较令我意外的情况,Java中的泛型基本上都是在编译器这个层次来实现的。在**生成的Java字节代码中是不包含泛型中的类型信息**的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉。 其实编译器通过Code sharing方式为每个泛型类型创建唯一的字节码表示，并且将该泛型类型的实例都映射到这个唯一的字节码表示上。将多种泛型类形实例映射到唯一的字节码表示是通过**类型擦除**（`type erasue`）实现的。

类型擦除,嘿嘿,第一次听说的东西,很好奇,于是上网查了查,把官方解释贴在下面,应该可以看得懂JavaDoc (http://docs.oracle.com/javase/tutorial/java/generics/erasure.html)

> Java语言引入了泛型，以在编译时提供更严格的类型检查并支持泛型编程。为了实现泛型，Java编译器将类型擦除应用于:
> 1 如果类型参数不受限制，则将通用类型中的所有类型参数替换为其边界或对象。因此，产生的字节码仅包含普通的类，接口和方法。
> 2 必要时插入类型转换，以保持类型安全。
> 3 生成桥接方法以在扩展的泛型类型中保留多态。
> 类型擦除可确保不会为参数化类型创建新的类；因此，泛型不会产生运行时开销。

---

**一、各种语言中的编译器是如何处理泛型的**

通常情况下，一个编译器处理泛型有两种方式:

1. `Code specialization`。在实例化一个泛型类或泛型方法时都产生一份新的目标代码（字节码or二进制代码）。例如，针对一个泛型 `list`，可能需要 针对 `string`， `integer`， `float` 产生三份目标代码。

2. `Code sharing`。对每个泛型类只生成唯一的一份目标代码；该泛型类的所有实例都映射到这份目标代码上，在需要的时候执行类型检查和类型转换。

**C++**中的模板（`template`）是典型的 `Code specialization` 实现。**C++**编译器会为每一个泛型类实例生成一份执行代码。执行代码中 `integer list` 和 `string list` 是两种不同的类型。这样会导致**代码膨胀（code bloat）**。 **C#**里面泛型无论在程序源码中、编译后的 `IL` 中（Intermediate Language，中间语言，这时候泛型是一个占位符）或是运行期的CLR中都是切实存在的， `List<int>` 与 `List<String>` 就是两个不同的类型，它们在系统运行期生成，有自己的虚方法表和类型数据，这种实现称为类型膨胀，基于这种方法实现的泛型被称为 `真实泛型`。**Java**语言中的泛型则不一样，它只在程序源码中存在，在编译后的字节码文件中，就已经被替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此对于运行期的Java语言来说，`ArrayList<int>` 与 `ArrayList<String>` 就是同一个类。所以说泛型技术实际上是Java语言的一颗语法糖，Java语言中的泛型实现方法称为**类型擦除**，基于这种方法实现的泛型被称为 `伪泛型`。

C++ 和 C# 是使用 Code specialization 的处理机制，前面提到，他有一个缺点，那就是**会导致代码膨胀。**另外一个弊端是在引用类型系统中，浪费空间，因为引用类型集合中元素本质上都是一个指针。没必要为每个类型都产生一份执行代码。而这也是Java编译器中采用 Code sharing 方式处理泛型的主要原因。

Java 编译器通过 Code sharing 方式为每个泛型类型创建唯一的字节码表示，并且将该泛型类型的实例都映射到这个唯一的字节码表示上。将多种泛型类形实例映射到唯一的字节码表示是通过**类型擦除**（ type erasue ）实现的。

---

### 二、什么是类型擦除

前面我们多次提到这个词：**类型擦除**（ type erasue ）**，那么到底什么是类型擦除呢？（泛型java代码转换成普通字节码）

> 类型擦除指的是通过类型参数合并，将泛型类型实例关联到同一份字节码上。编译器只为泛型类型生成一份字节码，并将其实例关联到这份字节码上。类型擦除的关键在于从泛型类型中清除类型参数的相关信息，并且再必要的时候添加类型检查和类型转换的方法。 类型擦除可以简单的理解为将泛型java代码转换为普通java代码，只不过编译器更直接点，将泛型java代码直接转换成普通java字节码。 类型擦除的主要过程如下： 1.将所有的泛型参数用其最左边界（最顶级的父类型）类型替换。（这部分内容可以看：Java泛型中extends和super的理解 (/archives/255)） 2.移除所有的类型参数。

类型擦除时机：编译器对.java进行编译成.class字节码时候，进行泛型擦除，泛型擦除只在编译器层次javac

---

### 三、Java编译器处理泛型的过程

**code 1:**

```
public static void main(String[] args) {
    Map<String, String> map = new HashMap<String, String>();
    map.put("name", "hollis");
    map.put("age", "22");
    System.out.println(map.get("name"));
    System.out.println(map.get("age"));
}
```

**反编译后的code 1:**

```
public static void main(String[] args) {
    Map map = new HashMap();
      map.put("name", "hollis");
      map.put("age", "22");
     System.out.println((String) map.get("name"));这里字节码中有checkcase string强制类型转换
      System.out.println((String) map.get("age"));
}
```

我们发现泛型都不见了，程序又变回了Java泛型出现之前的写法，泛型类型都变回了原生类型，

---

**code 2:**

```
interface Comparable<A> {
    public int compareTo(A that);
}

public final class NumericValue implements Comparable<NumericValue> {
    private byte value;

    public NumericValue(byte value) {
        this.value = value;
    }

    public byte getValue() {
        return value;
    }

    public int compareTo(NumericValue that) {
        return this.value - that.value;
    }
}
```

### 反编译后的code 2:

```
 interface Comparable {
   public int compareTo( Object that);擦除后 A被替换为最左边界Object顶级父类
}

public final class NumericValue
    implements Comparable
{
    public NumericValue(byte value)
    {
this.value = value;
    }
    public byte getValue()
    {
return value;
    }
    public int compareTo(NumericValue that)
    {
return value - that.value;
} 桥接方法
    public volatile int compareTo(Object obj)
    {
return compareTo((NumericValue)obj);
    }
    private byte value;
}
```

---

### code 3:

```
public class Collections {
    public static <A extends Comparable<A>> A max(Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0)
                w = x;
        }
        return w;
    }
}
```

**反编译后的code 3:**

```
public class Collections
{
         public Collections()
     {
     }
     public static Comparable max(Collection xs)     { A泛型的顶级父类是Comparable，因此泛型参数替换
 为Comparable
       Iterator xi = xs.iterator();
       Comparable w = (Comparable)xi.next();
       while(xi.hasNext())
       {
           Comparable x = (Comparable)xi.next();
           if(w.compareTo(x) < 0)
               w = x;
       }
       return w;
   }
}
```

第2个泛型类 `Comparable <A>` 擦除后 A被替换为最左边界 `Object` 。 `Comparable<NumericValue>` 的类型参数 `NumericValue` 被擦除掉，但是这直 接导致 `NumericValue` 没有实现接口 `Comparable的compareTo(Object that)` 方法，于是编译器充当好人，添加了一个**桥接方法。** 第3个示例中限定了类型参数的边界 `<A extends Comparable<A>>A` ，A必须为 `Comparable<A>` 的子类，按照类型擦除的过程，先讲所有的类型参数 ti换为最左边界 `Comparable<A>` ，然后去掉参数类型 `A` ，得到最终的擦除后结果。

---

**四、泛型带来的问题**

**一、当泛型遇到重载:**

```
public class GenericTypes {

    public static void method(List<String> list) {
        System.out.println("invoke method(List<String> list)");
    }

    public static void method(List<Integer> list) {
        System.out.println("invoke method(List<Integer> list)");
    }
}
```

上面这段代码，有两个重载的函数，因为他们的参数类型不同，一个是 `List<String>` 另一个是 `List<Integer>` ，但是，这段代码是编译通不过的。因为我们前面讲过，参数 `List<Integer>` 和 `List<String>` 编译之后都被擦除了，变成了一样的原生类型List，擦除动作导致这两个方法的特征签名变得一模一样。

**二、当泛型遇到catch:**

如果我们自定义了一个泛型异常类GenericException，那么，不要尝试用多个catch取匹配不同的异常类型，例如你想要分别捕获GenericException、GenericException，这也是有问题的。

三、当泛型内包含静态变量

```
public class StaticTest{
    public static void main(String[] args){
        GT<Integer> gti = new GT<Integer>();
        gti.var=1;
        GT<String> gts = new GT<String>();
        gts.var=2;
        System.out.println(gti.var);
    }
}
class GT<T>{
    public static int var=0;
    public void nothing(T x){}
}
```

答案是——2！由于经过类型擦除，所有的泛型类实例都关联到同一份字节码上，泛型类的所有静态变量是共享的。

---

## 五、总结

1.虚拟机中没有泛型，只有普通类和普通方法,所有泛型类的类型参数在编译时都会被擦除,泛型类并没有自己独有的Class类对象。比如并不存在 `List<String>.class`或是 `List<Integer>.class` ，而只有 `List.class` 。 2.创建泛型对象时请指明类型，让编译器尽早的做参数检查（**Effective Java，第23条：请不要在新代码中使用原生态类型**） 3.不要忽略编译器的警告信息，那意味着潜在的 `ClassCastException` 等着你。 4.静态变量是被泛型类的所有实例所共享的。对于声明为 `MyClass<T>` 的类，访问其中的静态变量的方法仍然是 `MyClass.myStaticVar` 。不管是通过 `new MyClass<String>` 还是 `new MyClass<Integer>` 创建的对象，都是共享一个静态变量。 5.泛型的类型参数不能用在 `Java` 异常处理的 `catch` 语句中。因为异常处理是由JVM在运行时刻来进行的。由于类型信息被擦除， `JVM` 是无法区分两个异常类型 `MyException<String>` 和 `MyException<Integer>` 的。对于 `JVM` 来说，它们都是 `MyException` 类型的。也就无法执行与异常对应的 `catch` 语句。

泛型java代码经过类型擦除后得到的class字节码

```
class Point< T>{  // 此处可以随便写标识符号，T是type的简称
   private T var ; // var的类型由T指定，即：由外部指定
   public Point(T var) {
      this.var = var;
   }
   public T getVar(){ // 返回值的类型由外部决定
      return var ;
   }
   public void setVar(T var){ // 设置的类型也由外部决定
      this.var = var ;
   }
};
```

```
class se基础/demo/泛型/Point {

  // compiled from:
GenericsDemo06.java

  // access flags 0x2
  // signature TT;
  // declaration: var extends T
这里T在字节码中泛型擦除，使用Object代替
  private Ljava/lang/Object; var

  // access flags 0x1
  // signature (TT;)V
  // declaration: void <init>(T)
  public <init>(Ljava/lang/Object;)V
```

Signature

Signature是JDK1.5时发布的，出现于类、属性表和方法表结构的属性表中。任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量（Type Variables）或参数化类型（Parameterized Types），则Signature属性会为他记录泛型签名信息。Signature就是参数化类型的泛型。它的结构为：

```
Signature_attribute {
   u2 attribute_name_index; // 固定Signature
   u4 attribute_length; // 固定2
   /*
   如果该签名属性是类文件结构的属性，则该索引处的常量池项必须是表示类签名的常量信息结构）；   如果该签名属性是方法信息结构的属性，则必须是方法签名；否则，必须是字段签名。
   */
   u2 signature_index; // 常量池有效索引。泛型类型T和擦除后的类型Object保存的字符串。类似map映射
}
```

复制代码

之所以要专门使用这样一个属性去记录泛型类型，是因为Java语言的泛型采用的是擦除法实现的伪泛型，在字节码（Code属性）中，泛型信息编译（类型变量、参数化类型）之后都统统被擦除掉。使用擦除法的好处是实现简单（主要修改Javac编译器，虚拟机内部只做了很少的改动）、非常容易实现Backport，运行期也能够节省一些类型所占的内存空间。但坏处是运行期就无法像C#等有真泛型支持的语言那样，将泛型类型与用户定义的普通类型同等对待，例如运行期做反射时无法获得到泛型信息。Signature属性就是为了弥补这个缺陷而增设的，现在java的反射API能够获取泛型类型，最终的数据来源也就是这个属性。

反射通过class文件内部的signature获取泛型信息

```
Class clz = Point.class;
    Field f = clz.getField("var");
    Class cl = f.getType();//这个方法获取到了类型，但是不带泛型信息
    System.out.println(cl);//class java.lang.Object，泛型擦除后的真实类型
    //获取原始属性的泛型信息T
    Type type = f.getGenericType();
    System.out.println(type);//泛型T
```