List<String>和List<Int>类型是不一样的，但是jvm运行时会采用泛型擦除。导致List<String>和List<Int>都是Class<List>.为了得到正确的类型，需要通过反射。

### **泛型擦除**

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，会在编译器在编译的时候去掉**。这个过程就称为类型擦除。泛型擦除是为了兼容jdk1.5之前的jvm，在这之前是不支持泛型的。

Manifest是scala2.8引入的一个特质，用于编译器在运行时也能获取泛型类型的信息。在JVM上，泛型参数类型T在运行时是被“擦拭”掉的，编译器把T当作Object来对待，所以T的具体信息是无法得到的；为了使得在运行时得到T的信息，scala需要额外通过Manifest来存储T的信息，并作为参数用在方法的运行时上下文。

def foo[T](x: List[T])(implicit m: Manifest[T]) = {

if (m <:< manifest[String])

println("Hey, this list is full of strings")

else

println("Non-stringy list")

}

foo(List("one", "two")) // Hey, this list is full of strings

foo(List(1, 2)) // Non-stringy list

foo(List("one", 2)) // Non-stringy list

**隐式参数m是由编译器根据上下文自动传入的，比如上面是编译器根据 "one","two" 推断出 T 的类型是 String**，从而隐式的传入了一个Manifest[String]类型的对象参数，使得运行时可以根据这个参数做更多的事情。

不过上面的foo 方法定义使用隐式参数的方式，仍显得啰嗦，于是scala里又引入了“上下文绑定”，使得foo方法

def foo[T](x: List[T]) (implicit m: Manifest[T])

可以简化为：

def foo[T:Manifest] (x: List[T])

scala在2.10里却用TypeTag替代了Manifest，用ClassTag替代了ClassManifest，原因是在路径依赖类型中，Manifest存在问题：

scala> class Foo{class Bar}

scala> def m(f: Foo)(b: f.Bar)(implicit ev: Manifest[f.Bar]) = ev

scala> val f1 = new Foo;val b1 = new f1.Bar

scala> val f2 = new Foo;val b2 = new f2.Bar

scala> val ev1 = m(f1)(b1)

ev1: Manifest[f1.Bar] = Foo@681e731c.type#Foo$Bar

scala> val ev2 = m(f2)(b2)

ev2: Manifest[f2.Bar] = Foo@3e50039c.type#Foo$Bar

scala> ev1 == ev2 // they should be different, thus the result is wrong

res28: Boolean = true

ev1 不应该等于 ev2 的，因为其依赖路径（外部实例）是不一样的。

还有其他因素（见下面的[引用](http://stackoverflow.com/questions/10513336/how-do-the-new-scala-typetags-improve-the-deprecated-manifests)），所以在2.10版本里，使用 TypeTag 替代了 Manifest

scala中用isInstanceOf判断一个对象实例是否是某种类型时，如果类型包含参数类型，会被擦拭掉(jvm的做法)。所以会导致例如下面的问题：

scala> (1,2).isInstanceOf[Tuple2[String,String]]

<console>:11: warning: fruitless type test: a value of type (Int, Int) cannot also be a (String, String) (but still might match its erasure)

(1,2).isInstanceOf[Tuple2[String,String]]

^

res33: Boolean = true

在给出了一段警告之后，结果返回true，如果数据对象支持模式匹配且元素较少，可以用模式匹配来精确判断：

scala> (1,2) match{ case(\_:Int, \_:Int) => println("ok"); case \_ => "no"}

ok

但元素比较多的话，或数据不支持模式匹配，要判断参数类型，就需要让编译器在运行时保持参数类型的信息了，在scala里是通过TypeTag来实现的

scala> def checkType[A: TypeTag](a: A, t: Type) = typeOf[A] <:< t

scala> checkType((1,2), typeOf[Tuple2[String,String]])

res37: Boolean = false

scala> checkType((1,2), typeOf[Tuple2[Int,Int]])

res38: Boolean = true

**classOf获取运行时的类型。classOf[T] 相当于 java中的T.class**

scala> class A

scala> val a = new A

scala> a.getClass

res2: Class[\_ <: A] = class A

scala> classOf[A]

res3: Class[A] = class A

scala> a.getClass == classOf[A]

res13: Boolean = true

上面显示了两者的不同，getClass 方法得到的是 Class[A]的某个子类，而 classOf[A] 得到是正确的 Class[A]，但是去比较的话，这两个类型是equals为true的

(在java里因为早期一直使用class表达type，并且现在也延续这样的习惯)；类型(type)比类(class)更”具体”，任何数据都有类型。类是面向对象系统里对同一类数据的抽象，在没有泛型之前，类型系统不存在高阶概念，直接与类一一映射，而泛型出现之后，就不在一一映射了。比如定义class List[T] {}, 可以有List[Int] 和 List[String]等具体类型，它们的类是同一个List，但类型则根据不同的构造参数类型而不同。

类型一致的对象它们的类也是一致的，反过来，类一致的，其类型不一定一致。

scala> **classOf[List[Int]] == classOf[List[String]]**

res16: **Boolean = true**

scala> **typeOf[List[Int]] == typeOf[List[String]]**

res17: **Boolean = false**

**ClassTag[T]保存了被泛型擦除后的原始类型T,提供给运行时的。**