译 scala中的

2018年02月01日 00:

Overcomi a

原文来自Overcomi

本文旨在展示一些 起的一些常见问题。

介绍

Scala有一个非常强 类型，结构类型，嵌套类型，路径依赖类型，抽象和具体类型成员，类型边界（(upper, l context），使用站 btype, parametric, F-bounded, ad-hoc），更高级的类型，广义类型的约束……而且这个 续。

但是，即使Scala的 类型相关的特性由于其运行时环境的限制而受到了削弱 - 这就是类型擦除。

什么是类型擦除？ 一个过程，它在编译后删除所有的泛型类型信息。这意味着我们无法在运行时区分 `List` `List [String]`。 拟机（运行Java和Scala的底层运行时环境）并不知道泛型。

类型擦除存在的历 当他们最终加入到Java 5中时，他们不得不保持向后兼容性。他们希望允许与旧的非通用 无缝接口（这就是 什么是通用类中的类型参数被替换为Object或其上限。例如：

```
1  class Foo
2    val foo
3  }
4  class Bar
5    val bar
6  }
7
8  //-----ty
9
10 class Foo
11   val foo
12 }
13 class Bar
14   val bar
15 }
```

所以，运行时我们 例子中，编译器只能看到原始的Foo和Bar。

不要认为类型擦除是 ，而是一种权衡。

我想谈的是我们如 有办法防止类型擦除本身，但是我们会看到一些方法来解决它。

它是如何工

这里有一个简单的

```
1  object Ex
2    def ext
3      case
4      case
5    }
6  }
7
8  val list
9  val resul
10 println(r
```

Map {

2")

tring2)

方法extract()获取 象，我们可以把数字、布尔值、字符串、其他对象放入其中。顺便说一句，在一段代码中 `[Any]` 应该是一个

所以，我们的愿望是有一个方法，只需要一个混合对象的列表，并只提取某种类型的对象。我们可以通过参数化方法extract()来选择这个类型。在给定选择的类型是String，这意味着我们将尝试从给定列表中提取所有字符串。

从严格的语言角度（没有进入运行时细节），这个代码是合理的。我们知道，模式匹配能够通过解构给定对象的类型而没有问题。但是，由于在JVM上所有通用类型在编译之后被擦除。因此模式匹配不能真正走得太远;类型的“第一级”之外的所有东西都被删除了。直接在Int或String（或任何非泛型类型MyNonGenericClass）上匹配我们的变量可以正常工作，但是在T上匹配它（T是泛型参数）则就不能通过编译。编译器会给我们一个警告，说“abstrac T is unchecked since it is eliminated by erasure”。

为了对这些情况提供一些帮助，Scala在2.7版本左右的地方引入了Manifests。 然而，他们有问题，不能代表某些类型，所以Scala 2.10中，他们放弃了更强大的TypeTag。

类型标签分为三种不同的类型：

- TypeTag
- ClassTag
- WeakTypeTag

即使这是文档中的官方分类，我认为更好的分类将是这样的：

- TypeTag：
  - “classic”
  - WeakTypeTag
- ClassTag

我的意思是，TypeTag和WeakTypeTag实际上是两个相同的事物，只有一个显着的差异（如我们稍后会显示），而ClassTag是一个完全不同的构造。

ClassTag

让我们回到我们的提取器例子，看看我们如何解决类型擦除问题。 我们现在要做的就是向extract()方法添加一个隐式参数：

```
1  import scala.reflect.ClassTag
2  object Extractor {
3    def extract[T](list: List[Any])(implicit tag: ClassTag[T]) =
4      list.flatMap {
5        case element: T => Some(element)
6        case _ => None
7      }
8  }
9  val list: List[Any] = List(1, "string1", List(), "string2")
10 val result = Extractor.extract[String](list)
11 println(result) // List(string1, string2)
```

打印语句显示 `List（string1，string2）` 。

请注意，我们也可以在这里使用上下文绑定语法：

```
1  // def extract[T](list: List[Any])(implicit tag: ClassTag[T]) =
2  def extract[T : ClassTag](list: List[Any]) =
```

我将使用标准语法来简化代码，不需要额外的语法糖。

那么它是怎样工作的？那么，当我们需要一个类型为ClassTag的隐式值时，编译器会为我们创建这个值。文档说：

> If an implicit value of type u.ClassTag[T] is required, the compiler will make one up on demand.
> 如果需要一个类型为 `u.ClassTag [T]` 的隐式值，编译器会根据需要创建一个。

所以，编译器很乐意为我们提供一个需要ClassTag的隐式实例。这种机制也将与 `TypeTag` 和 `WeakTypeTag` 一起使用。

我们在 `extract()` 方法中提供了隐式的 `ClassTag` 值。一旦我们进入方法体内部会发生什么？

再次看一下这个例子 - 编译器不仅自动为我们提供了隐式参数标记的值，而且我们也不需要使用参数本身。我们从来不需要对 `Tag` 值做任何事情。只是我们的模式匹配就能够成功匹配我们列表中的字符串元素。

我们可以检查文档以寻找解释。事实上，它隐藏在这里：

> Compiler tries to turn unchecked type tests in pattern matches into checked ones by wrapping a *(: T) type pattern as ct(*: T), where ct is the ClassTag[T] instance.
> 编译器试图通过包装一个 `(_: T)` 类型模式为 `ct(_: T)`，其中ct是 `ClassTag [T]` 实例，将模式匹配中未经检查的类型测试变成已检查的类型。

基本上，如果我们为编译器提供一个隐式的ClassTag，它会重写模式匹配中的条件，以使用给定的标签作为extractor。我们的条件：

```
1 {case element: T => Some(element)}
```

由编译器翻译（如果在范围内有一个隐含的标签）到这里：

```
1 {case (element @ tag(_: T)) => Some(element)}
```

如果你以前从未见过“@”构造，那只是给你匹配的类命名的一种方法，例如：

```
1 {
2     case Foo(p, q) =>
3         // we can only reference parameters via p and q
4     case f @ Foo(p, q) =>
5         // we can reference the whole object via f
6 }
```

如果没有可用的类型为T的隐式 `ClassTag`，则编译器将被削弱（由于缺少类型信息），并且会发出警告，表明我们的模式匹配将受到类型T上的类型擦
译不会中断，但是当我们进行模式匹配时，不要期望编译器知道什么是T（因为它将在运行时被JVM擦除）。如果我们为类型T提供了一个隐式的 `Class`
译器会很高兴在编译时提供一个合适的 `ClassTag`，就像我们在例子中看到的那样。标签将带来关于T是一个字符串的信息，类型删除不能触摸它。

但是有一个重要的弱点。如果我们想要在更高级别上区分我们的类型，并从我们的初始列表中获得 `List [Int]` 的值，而忽略例如列出 `[String]`，我们
做：

```
1 val list: List[List[Any]] = List(List(1, 2), List("a", "b"))
2 val result = Extractor.extract[List[Int]](list)
3 println(result) // List(List(1, 2), List(a, b))
```

我们只想提取 `List [Int]`，但是我们也得到了 `List [String]`。 `Class tags` 不能在更高层次上进行区分。

这意味着我们的提取器可以区分例如 `sets` 和 `lists`，但它不能将一个列表与另一个列表区分开来（例如 `List [Int]` 和 `List [String]`）。当然，这
列表，这适用于所有的通用trait/class。

TypeTag

`ClassTag` 失败的地方，开发人员用 `TypeTag` 来弥补。 它可以区分 `List [String]` 和 `List [Integer]`。 它也可以更深入一些，比如区分 `List [Set`
中的 `List [Set [Int]]`。因为TypeTag在运行时有更丰富的关于泛型类型的信息。

我们可以很容易地得到所讨论类型的完整路径以及所有嵌套类型（如果有的话）。 要得到这个信息，你只需要在给定的标签上调用tpe()。

这是一个例子。 隐式标签参数由编译器提供，就像ClassTag一样。 请注意“args”参数 - 它是包含ClassTag没有的其他类型信息的信息（有关由Int参数
息）。

```
1  import scala.reflect.runtime.universe._
2  object Recognizer {
3    def recognize[T](x: T)(implicit tag: TypeTag[T]): String =
4      tag.tpe match {
5        case TypeRef(utype, usymbol, args) =>
6          List(utype, usymbol, args).mkString("\n")
7      }
8  }
9
10 val list: List[Int] = List(1, 2)
11 val result = Recognizer.recognize(list)
12 println(result)
13 // prints:
14 //   scala.type
15 //   type List
16 //   List(Int)
```

我在这里介绍了一个新的对象 - 一个Recognizer。

不幸的是，我们无法使用TypeTags实现Extractor。但是我们可以获得更多关于类型的信息，比如了解更高类型（也就是说，能够区分 `List[X]` 和 `List` 它们的缺点是它们不能用于运行。

我们可以使用TypeTag在运行时获取某种类型的信息，但是我们不能用它来在运行时找出某个对象的类型。我们传入recognize()的是一个简单的 `List` 们的 `List(1,2)` 值的声明类型。但是，如果我们将 `List(1,2)` 声明为 `List [Any]` ，TypeTag会告诉我们我们已经通过一个 `List [Any]` 。

下面是ClassTags和TypeTag之间的两个主要区别：

1. ClassTag不知道“更高类型”;给定一个 `List [T]` ，一个ClassTag只知道这个值是一个 `List` ，对 `T` 一无所知。
2. TypeTag知道“更高类型”，并且有更丰富的类型信息，但不能用于在运行时获取有关值的类型信息。换句话说，TypeTag提供了关于类型的运行时信息，而ClassTag提运行时信息（更具体地说，是在运行时告诉我们所讨论的值的实际类型的信息）。

还有一点值得一提的是ClassTag和(Weak)TypeTag之间的区别：ClassTag是一个经典的老式类。它为每个类型捆绑了一个单独的实现，这使得它成为一型模式。另一方面，(Weak)TypeTag有点复杂，为了使用它，我们需要在代码中有一个特殊的导入，正如你在前面给出的代码片段中注意到的那样。我 universe：

> Universe provides a complete set of reflection operations which make it possible for one to reflectively inspect Scala type relations, such as membership or subtyping.
> Universe提供了一套完整的反射操作，使得人们可以反思性地检查Scala类型关系，例如成员资格或子类型。

不要担心，只需要导入正确的Universe，并且在(Weak)TypeTag（scala.reflect.runtime.universe._ (docs)）的情况下。

WeakTypeTag

您可能觉得TypeTag和WeakTypeTag是非常相似的，因为迄今为止所有的差异都是在ClassTag中解释的。 这是正确的; 他们确实是同一个工具的两个变有一个重要的区别。

我们看到TypeTag足够聪明，可以检查类型，类型参数，类型参数等等。但是，所有类型都是具体的。 如果一个类型是抽象的，TypeTag将无法解决它 WeakTypeTag进场的地方。 让我们来修改TypeTag示例一下：

```
1  val list: List[Int] = List(1, 2)
2  val result = Recognizer.recognize(list)
```

看那边的那个Int？它可以是任何其他具体类型，如 `String` ， `Set [Double]` 或 `MyCustomClass` 。但是如果你有一个抽象类型，你需要一个 `WeakTypeT`

这是一个例子。 请注意，我们需要对抽象类型的引用，所以我们只需将所有内容都包含在抽象类中。

```
1  import scala.reflect.runtime.universe._
2  abstract class SomeClass[T] {
3    object Recognizer {
4      def recognize[T](x: T)(implicit tag: WeakTypeTag[T]): String =
5        tag.tpe match {
6          case TypeRef(utype, usymbol, args) =>
7            List(utype, usymbol, args).mkString("\n")
8        }
9    }
10
11   val list: List[T]
12   val result = Recognizer.recognize(list)
13   println(result)
14 }
15
16 new SomeClass[Int] { val list = List(1) }
17 // prints:
18 //   scala.type
19 //   type List
20 //   List(T)
```

结果类型是一个 `List [T]` 。

如果我们使用TypeTag而不是WeakTypeTag，编译器会抱怨“no TypeTag available for List[T]”。 所以，你可以把WeakTypeTag看作TypeTag的一个超集

请注意，WeakTypeTag尽可能具体，所以如果有一个类型标签可用于某种抽象类型，WeakTypeTag将使用该类型标记，从而使类型具体而不是抽象的

结论

在我们完成之前，让我提一下，每个类型标签也可以使用可用的助手来显式实例化：

```scala
import scala.reflect.classTag
import scala.reflect.runtime.universe._

val ct = classTag[String]
val tt = typeTag[List[Int]]
val wtt = weakTypeTag[List[Int]]

val array = ct.newArray(3)
array.update(2, "Third")

println(array.mkString(","))
println(tt.tpe)
println(wtt.equals(tt))

//  prints:
//    null,null,Third
//    List[Int]
//    true
```

就这样。 我们看到了三个构造，ClassTag，TypeTag和WeakTypeTag，它们将帮助您在日常Scala生活中解决大部分类型的擦除问题。

请注意，使用标签（这基本上是反射下）可以减慢速度，使生成的代码显着变大，所以不要在你的库中添加隐式类型标签，以使编译器更加“智能” 没有保存它们，当你真的需要它们。

而当你需要它们的时候，它们将会提供一个强大的武器来对付JVM的类型擦除。

Scala数据类型 - 自由的云 46

scala数据类型 Any是所有类型的父类型，其两个子类型是AnyVal和AnyRef(java.lang.Object) AnyVal代... 来自： 自由的云

相关热词 go语言和scala bootstrap中的文本框问题 c++中 const引用问题 c#中银行存取款问题 c++中兔子繁殖问题

Scala基础总结（三） - 一次次尝试 2525

Scala总结抽象类和抽象成员 与java相似，scala中abstract声明的类是抽象类，抽象类不可以被实例化... 来自： 一次次尝试

Kotlin语法（十二）-泛型（Generics） - tangxl2008008的专栏 8209

参考原文：http://kotlinlang.org/docs/reference/generics.html 泛型类 跟Java一样，Kotlin也支... 来自： tangxl2008008的专栏

快学Scala第18章----高级类型 - 大冰的小屋 1374

本章要点 单例类型可用于方法串接和带对象参数的方法。 类型投影对所有外部类型的对象都包含了其... 来自： 大冰的小屋

Kotlin泛型类型参数 - spy_develop的博客 2304

Kotlin泛型类型参数 泛型允许你定义带类型参数的类型。当这种类型的实例被创建出来的时候，类型... 来自： spy_develop的博客

Java笔记 – 泛型 泛型方法 泛型接口 擦除 边界 通配符 - DennisRuan(米粒橙) 3006

Java笔记 – 泛型 泛型方法 泛型接口 擦除 边界 通配符 Java中的泛型参考了C++的模板，Java的界... 来自： DennisRuan(米粒橙)

java泛型 泛型的内部原理：类型擦除以及类型擦除带来的问题 - wisgood的专栏 9037

一、Java泛型的实现方法：类型擦除 前面已经说了，Java的泛型是伪泛型。为什么说Java的泛型是伪... 来自： wisgood的专栏

处理Scala的类型擦除问题 - lovec的专栏 6

在Scala中，你如果使用了泛型的话，那你在Pattern Matching的时候要注意了，因为会有类型擦除的问... 来自： lovec的专栏

Java的类型擦除 - Franco的博客 378

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... 来自： Franco的博客

使用Gson解析Json数组遇到的泛型类型擦除问题解决方法 - submorino的专栏 3820

谷歌Gson转换Json串有如下方法： public Object fromJson(String json, Type typeOfT); 可以使用它进行... 来自： submorino的专栏

深入理解Java虚拟机 - 泛型与类型擦除 - itmyhome的专栏 635

泛型是JDK 1.5的一项新增特性，它的本质是参数化类型（Parametersized Type）的应用，也就是说所... 来自： itmyhome的专栏

scala类型检查和转换 - hangscer的博客 439

Scala和Java中的类型检查和转换 来自： hangscer的博客

isInstanceOf与类型擦除 - ZERO 620

转载自：http://hongjiang.info/scala-pitfalls-11-type-erasure/ scala中用isInstanceOf判断一个对象... 来自： ZERO

scala 的模式匹配与类型系统 - Hipparchus的博客 3210

主要内容： 1. scala模式匹配 2. scala类型系统 来自： Hipparchus的博客

scala的类与类型 - edwardsbean的专栏 1.3万

scala的类与类型 类和类型 List和List类型是不一样的，但是jvm运行时会采用泛型擦除。导致List和List... 来自： edwardsbean的专栏

Scala超全详解 - lukabruce的博客 739

转自： 【https://blog.csdn.net/c391183914/article/details/78647533?locationNum=2&fps=... 来自： lukabruce的博客

Scala基础语法大全总结（一） - 不清不慎的博客 259

Scala语言是一门基于JVM的编程语言，具有强大的功能，它即具有类似Java的面向对象的特性，而且... 来自： 不清不慎的博客

scala类型系统：this别名&自身类型 - hellojoy的博客 23

看scala的源码的话很发现很多源码开头都有一句：self => 这句相当于给this起了一个别名为self... 来自： hellojoy的博客

从头认识java-13.6 类型擦除（type erasure） - raylee2007的专栏 2288

这一章节我们讨论一下类型擦除。1.什么是类型擦除？在java里面泛型其实是伪泛型，因为他都只是在... 来自： raylee2007的专栏

java泛型的内部原理、类型擦除以及类型擦除带来的问题 - blueskyliulan的专栏 499

转自 http://blog.csdn.net/lonelyroamer/article/details/7868820 一、Java泛型的实现方法：类型擦除 ... 来自： blueskyliulan的专栏

C++中的类型擦除（type erasure in c++） - pud_zha的专栏 866

关于类型擦除，在网上搜出来的中文资料比较少，而且一提到类型擦除，检索结果里就跑出很多 Java ... 来自： pud_zha的专栏

泛型(二)->擦除&擦除带来的问题 - 朱利源的博客 1804

泛型(二)->擦除&擦除带来的问题 本篇首先介绍泛型的擦除,然后围绕泛型擦除所带来的问题进行精确打... 来自： 朱利源的博客

C++类型擦除 - xcw_1987的博客 40

转自：http://www.cnblogs.com/liyiwen/archive/2009/12/10/1621451.html 关于类型擦除，在网上... 来自： xcw_1987的博客

泛型擦除问题带来的无法正常解析Json问题 - wjt_developer的博客 220

我们在做网络请求，并讲结果解析未Bean的时候，因为Bean类型的不统一，所以无法统一的解析Bean,... 来自： wjt_developer的博客

【java】--泛型-类型擦除与多态的冲突和解决方法 - qfzhangwei的专栏 277

类型擦除与多态的冲突和解决方法 现在有这样一个泛型类： [java] view plain copy print? class Pair { ... 来自： qfzhangwei的专栏

java分布式环境下，反序列化遇到类型擦除问题，解决思路 - a1523407的博客 909

前言 公司使用springcloud搭建了一个分布式框架。但是在框架之间调用的时候，如果返回结果的真实... 来自： a1523407的博客

Java泛型04：泛型类型擦除 - 韩超的博客 (hanchao5272) 554

超级通道： Java泛型学习系列-绪论 本章主要对Java泛型的类型擦除进行学习。 经过前面几个章节的学... 来自： 韩超的博客 (hanchao5...

java_web初学笔记之<泛型的类型擦除> - bgk083的专栏 317

java泛型类型擦除： 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... 来自： bgk083的专栏

Java泛型，你了解类型擦除吗？ 侵立删 - 心神沫沫的收藏 25

转自：blog.csdn.net/briblue/article/details/76736356泛型，一个孤独的守门者。大家可能会有疑问，... 来自： 心神沫沫的收藏

Java泛型--编译器类型擦除 - 借你一秒 1213

Java的泛型是伪泛型。在编译期间，所有的泛型信息都会被擦除掉。正确理解泛型概念的首要前提是理... 来自： 借你一秒

Scala之若干细小问题汇总 - Laurence的技术博客 2993

Scala中下划线的应用场景一 Scala中下划线的应用场景二 identity方法的应用场景Scala中下划线的应用... 来自： Laurence的技术博客

Java 共变数组和类型擦除 - 书弋江山的博客 1065

SE5之前还没有泛型，但很多代码迫切需要泛型来解决问题，在真正引入泛型之前，如果数组不能协变... 来自： 书弋江山的博客

[运行时获取模板类类型] Java 反射机制 + 类型擦除机制 - tbwork 7722

运行时获取模板类T的类型：Java 反射机制 + 类型擦除机制。 来自： tbwork

类型擦除 - liudongdong_jlu 27

转载于：https://blog.csdn.net/wisgood/article/details/11762427 一、Java泛型的实现方法：类型擦除... 来自： liudongdong_jlu

Java -- 泛型中的类型擦除机制介绍（二） - 加油熬过去，别输给这个操蛋的时代 540

Java -- 泛型中的类型擦除机制介绍（二） 上一篇博文中，我们主要介绍了泛型的一些基本使用方法；... 来自： 加油熬过去，别输给...

1.5.7--1.5.8类型擦除、泛型的限制和类变量、实例变量、局部变量的区别 - FromNowOnUntil... 57

1.5.7 类型擦除 泛型类可以由编译器通过所谓的类型擦除过程而转变成非泛型类。这样，编译器就... 来自： FromNowOnUntilThe...

java基础(28)--泛型与类型擦除、泛型与继承 - 上善若水，厚德载物 921

本文转载自：http://blog.sina.com.cn/s/blog_7ffb8dd501012ku9.html 尊重原创【泛型与类型擦除】泛... 来自： 上善若水，厚德载物

巧用Scala结合par方法 - 关新宇的博客 146

不说话，直接看图： 是很简单的执行24次count，左边是串行（节省时间我把第五个jobkill掉了），右... 来自： 关新宇的博客

关于泛型中的类型擦除问题 - kongshaohao的博客 16

在JAVA中，我们会经常用到List<..>,其中<>中的即为泛型，在这里我想... 来自： kongshaohao的博客

java编程思想 泛型擦除的补偿 - zhuojl的博客 387

15.8 擦除的补偿 正如我们看到的，擦除丢失了在泛型代码中执行某些操作的能力。任何在运行时需要... 来自： zhuojl的博客

JVM学习笔记-类型擦除机制 - WD的博客 22

java的泛型在编译阶段实现，，在运行期被删除。编译器生成的字节码在运行期间并不包含泛型的类型... 来自： WD的博客

java语法糖一些简要心得（含有类型擦除和拆箱和装箱） - opprash 15

什么叫语法糖？ 语法糖既是指在计算机语言中添加某种语法，这个语法本生对程序来说并没有什么坏... 来自： opprash