scala函数式与闭包的字节码分析

scala中函数作为一等公民，在scala中可以理解函数同class一样具有类型，scala中默认定义了22种类型的函数，分别对应含有不同数量类型参数的函数。scala同java一样编译后的字节码都是跑在JVM中，最终都会生成class的文件。函数式范畴作为scala利器，其实现方式也受限JVM。scala中函数类型实现采用匿名内部类方式实现，在上下文中定义的函数都将转化为匿名类，下面会详细讲解在类和对象中两种实现方式。从定义上讲函数包含闭包，闭包是会产生副作用的函数或非纯函数，修改引用上下文中变量值或从引用上下文中获取数据，在分布式系统中，如果忽略引用上下文的可见性将导致闭包函数执行产生不可预期的结果。

在类中定义匿名内部类实现函数式和闭包

```
class UserHandler{
  val bean:String=""
  val testF1=(idx:Int)=>{
    idx+bean //使用UserHandler中的bean变量
  }
  val testF2=(idx1:String,f:Int=>String)=>{
    idx1+f(1)
  }
  val testF3=(idx:String)=>{
    //TODO
  }
}
```

编译后会生成4个class文件

```
UserHandler$$anonfun$1.class //testF1生成的class文件
UserHandler$$anonfun$2.class //testF2生成的class文件
UserHandler$$anonfun$3.class //testF3生成的class文件
UserHandler.class //UserHandler生成的文件
```

> 注：***UserHandler$$anonfun$_*** 后面的名字可能不一样,由编译器在编译的时候决定

使用***javap -private***命令参看4个文件的签名

```
public final class org.apache.spark.UserHandler$$anonfun$1 extends scala.r
untime.AbstractFunction1<java.lang.Object, java.lang.String> implements sc
ala.Serializable {
  public static final long serialVersionUID;
  private final org.apache.spark.UserHandler $outer;
  public final java.lang.String apply(int);
  public final java.lang.Object apply(java.lang.Object);
  public org.apache.spark.UserHandler$$anonfun$1(org.apache.spark.UserHand
ler);
}
public final class org.apache.spark.UserHandler$$anonfun$2 extends scala.r
untime.AbstractFunction2<java.lang.String, scala.Function1<java.lang.Object
, java.lang.String>, java.lang.String> implements scala.Serializable {
  public static final long serialVersionUID;
  public final java.lang.String apply(java.lang.String, scala.Function1<ja
va.lang.Object, java.lang.String>);
  public final java.lang.Object apply(java.lang.Object, java.lang.Object);
  public org.apache.spark.UserHandler$$anonfun$2(org.apache.spark.UserHand
ler);
}
public final class org.apache.spark.UserHandler$$anonfun$3 extends scala.r
untime.AbstractFunction1<java.lang.String, scala.runtime.BoxedUnit> implem
ents scala.Serializable {
  public static final long serialVersionUID;
  public final void apply(java.lang.String);
  public final java.lang.Object apply(java.lang.Object);
  public org.apache.spark.UserHandler$$anonfun$3(org.apache.spark.UserHand
ler);
}
public class org.apache.spark.UserHandler {
```

```
  private final java.lang.String bean;
  private final scala.Function1<java.lang.Object, java.lang.String> testF1;

  private final scala.Function2<java.lang.String, scala.Function1<java.lang
.Object, java.lang.String>, java.lang.String> testF2;
  private final scala.Function1<java.lang.String, scala.runtime.BoxedUnit>
testF3;
  public java.lang.String bean();
  public scala.Function1<java.lang.Object, java.lang.String> testF1();
  public scala.Function2<java.lang.String, scala.Function1<java.lang.Object
, java.lang.String>, java.lang.String> testF2();
  public scala.Function1<java.lang.String, scala.runtime.BoxedUnit> testF3
();
  public org.apache.spark.UserHandler();
}
```

从生成的三个匿名内部类中可以看出，testF1生成的内部类中添加了指向UserHandler的***$outer***指针，因为在testF1中引用了UserHandler的内部变量，其他两个匿名内部类中没有指向外部UserHandler的***$outer***指针。从闭包的定义分析（闭包是由引用上下文和函数组成分析），testF1是闭包，testF2和testF3都不是闭包，是线程安全的自闭性很好的纯函数。对于testF2和testF3做序列化存储不会出现问题，但是对于testF1就需要考虑引用类和外部类的序列化问题。

在对象中定义匿名内部类实现函数式和闭包

```
object Reflector {
  var bean="singleton"
  val testF=(idx1:String,idx2:String)=>{
    bean=idx1+idx2
    bean
  }
}
```

编译后会生成3个class文件

```
Reflector$$anonfun$4.class
Reflector$.class
Reflector.class
```

> 可以这样理解，在scala中如果只是定义了伴生对象没有定义伴生对象类，scala编译器会在编译时，为我们创建一个包含所有伴生对象(编译后的文件***Reflector.class***)中定义的变量和方法的伴生对象类（编译后的文件***Reflector$.class***），并在伴生对象类中初始化一个指向伴生对象类的静态变量(***MODULE$***)；伴生对象中所有变量的存储和方法的实现都在伴生对象类中。

使用***javap -private***命令参看3个文件的签名

```
public final class org.apache.spark.Reflector$$anonfun$4 extends scala.run
time.AbstractFunction2<java.lang.String, java.lang.String, scala.runtime.B
oxedUnit> implements scala.Serializable {
  public static final long serialVersionUID;
  public final void apply(java.lang.String, java.lang.String);
  public final java.lang.Object apply(java.lang.Object, java.lang.Object);
  public org.apache.spark.Reflector$$anonfun$4();
}
public final class org.apache.spark.Reflector$ {
  public static final org.apache.spark.Reflector$ MODULE$;
  private java.lang.String bean;
  private final scala.Function2<java.lang.String, java.lang.String, scala.
runtime.BoxedUnit> testF;
  public static {};
  public java.lang.String bean();
  public void bean_$eq(java.lang.String);
  public scala.Function2<java.lang.String, java.lang.String, scala.runtime.
BoxedUnit> testF();
  private org.apache.spark.Reflector$();
```

```
}
public final class org.apache.spark.Reflector {
  public static scala.Function2<java.lang.String, java.lang.String, scala.
runtime.BoxedUnit> testF();
  public static void bean_$eq(java.lang.String);
  public static java.lang.String bean();
}
```

按照上面的说法伴生对象中定义的***textF***函数类型的方法中，赋值操作引用伴生对象类中***bean***变量，按照闭包定影，testF应该属于闭包类型；但是为什么没有在***textF***生成的字节码文件***Reflector$$anonfun$4.class***中找到对伴生对象类的引用。对比testF和testF1的构造函数可以了解一些东西，对于在class中定义的闭包，编译器在生成闭包字节码的时候，会创建一个含有外部类参数的构造函数，在初始化闭包实体的时候会将外部类的应用传递到闭包实现中，使用***$outer***变量保存起来方便在方法中的操作使用。对于在Object中创建的闭包，编译器默认为生成一个不含参数的构造函数，函数操作中使用object生成的伴生对象类中的***MODULE$***来操作使用到伴生对象类中的变量和方法。并且生成的伴生对象类默认是实现了序列化接口。

testF闭包编译后***apply***和构造函数方法的代码段如下。

```
public final java.lang.String apply(java.lang.String, java.lang.String);
    Code:
       0: getstatic     #23                 // Field org/apache/spark/Refl
ector$.MODULE$:Lorg/apache/spark/Reflector$; //获取伴生对象类的引用
       3: new           #25                 // class scala/collection/muta
ble/StringBuilder //创建StringBuilder容器
       6: dup
       7: invokespecial #26                 // Method scala/collection/mut
able/StringBuilder."<init>":()V //初始化StringBuidler容器
      10: aload_1
      11: invokevirtual #30                 // Method scala/collection/mut
able/StringBuilder.append:(Ljava/lang/Object;)Lscala/collection/mutable/St
ringBuilder; //向StringBuilder中添加idx1
```

```
      14: aload_2
      15: invokevirtual #30                 // Method scala/collection/mut
able/StringBuilder.append:(Ljava/lang/Object;)Lscala/collection/mutable/St
ringBuilder;//向StringBuilder中添加idx2
      18: invokevirtual #34                 // Method scala/collection/mut
able/StringBuilder.toString:()Ljava/lang/String; //拼接字符串
      21: invokevirtual #38                 // Method org/apache/spark/Ref
lector$.bean_$eq:(Ljava/lang/String;)V //调用伴生对象类的bean_$eq方法对bean进
行赋值
      24: getstatic     #23                 // Field org/apache/spark/Refl
ector$.MODULE$:Lorg/apache/spark/Reflector$; //获取伴生对象类的引用
      27: invokevirtual #41                 // Method org/apache/spark/Ref
lector$.bean:()Ljava/lang/String; //获取bean的值
      30: areturn
  public org.apache.spark.Reflector$$anonfun$4();
    Code:
       0: aload_0
       1: invokespecial #55                 // Method scala/runtime/Abstra
ctFunction2."<init>":()V
       4: return
```

testF1闭包编译后***apply***和构造函数方法的代码段如下

```
public final java.lang.String apply(int);
    Code:
       0: new           #23                 // class scala/collection/muta
ble/StringBuilder
       3: dup
       4: invokespecial #24                 // Method scala/collection/mut
able/StringBuilder."<init>":()V
       7: iload_1
       8: invokevirtual #28                 // Method scala/collection/mut
able/StringBuilder.append:(I)Lscala/collection/mutable/StringBuilder;
```

```
      11: aload_0
      12: getfield      #30                 // Field $outer:Lorg/apache/spark/UserHandler;
      15: invokevirtual #34                 // Method org/apache/spark/UserHandler.bean:()Ljava/lang/String;
      18: invokevirtual #37                 // Method scala/collection/mutable/StringBuilder.append:(Ljava/lang/Object;)Lscala/collection/mutable/StringBuilder;
      21: invokevirtual #40                 // Method scala/collection/mutable/StringBuilder.toString:()Ljava/lang/String;
      24: areturn
public org.apache.spark.UserHandler$$anonfun$1(org.apache.spark.UserHandler);
    Code:
       0: aload_1
       1: ifnonnull     12
       4: new           #58                 // class java/lang/NullPointerException
       7: dup
       8: invokespecial #59                 // Method java/lang/NullPointerException."<init>":()V
      11: athrow
      12: aload_0
      13: aload_1
      14: putfield      #30                 // Field $outer:Lorg/apache/spark/UserHandler;
      17: aload_0
      18: invokespecial #60                 // Method scala/runtime/AbstractFunction1."<init>":()V
      21: return
```

> 注：不管是在class或object中定义闭包，当需要考虑对闭包做持久化存储或网络传递都应该考虑闭包函数中引用上下文中变量的序列化问题和可见性问题。