

Scala中的函数式特征

ADC 2013

关于我

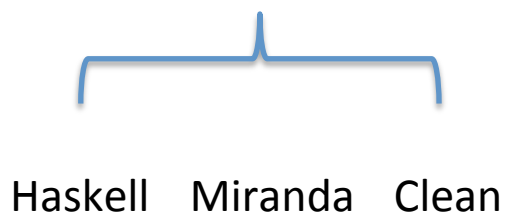
- 王宏江，花名：宏江
- Blog: <http://hongjiang.info>
- 经历：
 - Java : 10y+
 - Scala : 2y+
 - 十年工作经验，2009年加入阿里，曾在阿里巴巴中文站和laiwang.com担任架构师，现在中间件&稳定性平台部门
 - Scala布道者
 - 业余马拉松爱好者

说明

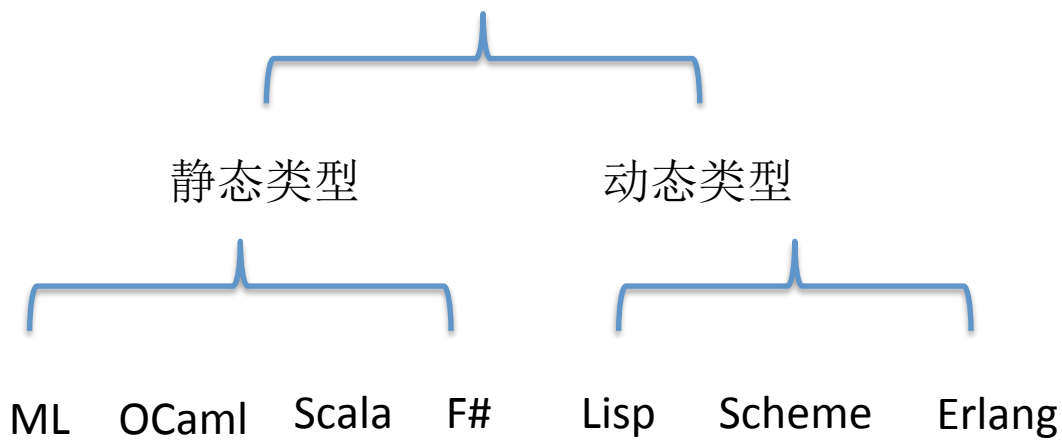
- 内容
 - 1) Scala中的函数式的特征
 - 2) Scala类型系统中的函数式特征
- 交流: <https://github.com/CSUG/csug/>
- 旺旺群:
Scala交流和答疑: 94329267/sugsug

函数式语言

纯函数式语言



非纯函数式语言



关于Scala语言



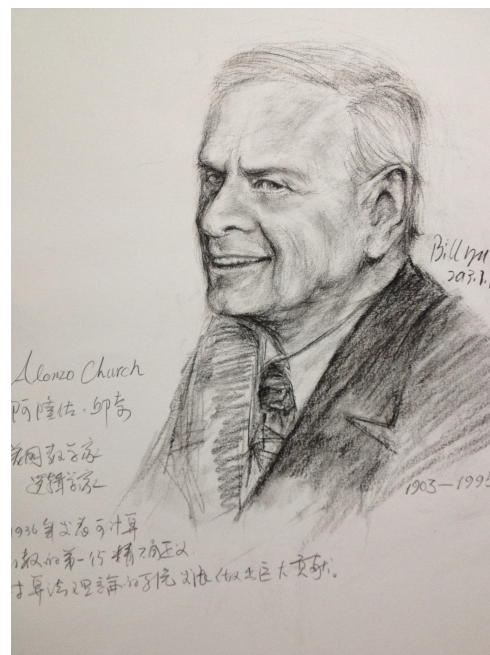
面向对象与函数式被看成一个硬币的两面，Scala试图将这两面融合起来

怎么理解这两种风格？

命令式与函数式

在《程序设计语言—实践之路》一书中
将“面向对象”与“面向过程”都归类为“命令式”语言

命令式与函数式



命令式与函数式

- 图灵机/冯诺依曼体系
 - 其运算可看做：
 - 通过修改内存来反映运算的结果。
 - (用命令修改状态)
- Lambda演算(λ calculus)形式
 - 连续运算(基于函数)得到结果

- 函数式程序是跟值打交道的，而不是跟状态打交道的。它们的工具是表达式，而不是命令。

— 《ML for the Working Programmer》

值 vs 状态

副作用(side effect)

Scala的做法

- 同时支持val和var体现了scala的“平衡”也可以说是“圆滑”
- 鼓励使用val，但并不排斥var；实际编码中val占大多数。

函数式是面向表达式的

Scala中的表达式

- 1) 所有的表达式都有值
- 2) 除了赋值和函数调用表达式，内置的几个表达式只有：if,while,for,try,match
- 3) 块表达式{...}是基本表达式的组合，它的值是最后一个表达式的值。

Scala中的表达式

- 一些表达式的值：
 - 1) `a=1;`
 - 2) `while(a<100){print(a)}`
 - 3) `if(a<2) 1;`
- 赋值表达式、`while`表达式的值是`Unit`类型,它的值是唯一的: `()`
- `if`表达式缺乏`else`的话, 不符合条件则返回`Unit`类型的`()`; 即上面相当于: `if(a<2) 1 else ()`

赋值语句的注意点：

不同于java：

1) `while((line = readLine()) != null)`

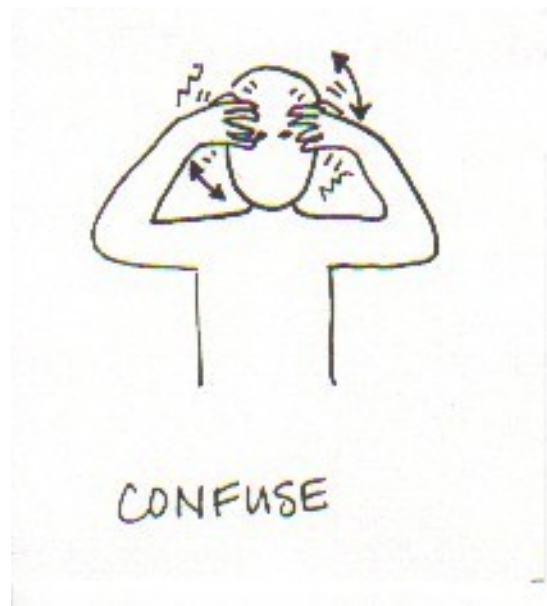
不起作用，前边的 `line = readLine()` 得到的是Unit类型值

2) `x=y=1; // y=1; x=()`

`y=1`表达式的结果是(), `x`被赋予了Unit类型的值

表达式主要是由函数调用组成的

Function vs Method



Scala中的Function/Method/Functor

1) 狭义地区分(从可传递性上):

方法(method): 指的是在trait/class/object中以def关键字声明的, 它不能被直接传递。

函数(function): 类型为ParamsType=>ResultType的变量, 这些变量背后是用FunctionN对象来封装的; 可以被传递。方法可以转换为函数。

2) 广义上, 抛开背后的实现, 方法就是函数; 编译器某些场景自动把方法封装为一个函数对象来传递。Scala社区并不特别区分这两个名词, 注意语境, 有时候函数就是指方法, 有时候则是指函数对象

函数作为一等公民

函数作为一等公民体现在哪儿？

- 1) 可传递/赋值
- 2) 嵌套函数和匿名函数
- 3) 高阶
- 4) 偏应用(partial application)
- 5) 闭包

参考: http://en.wikipedia.org/wiki/First-class_function

可传递

```
scala> def hf(f: ()=>String) = println(f)  
hf: (f: () => String)Unit
```

```
scala> hf(()=>"hi") // 传递一个匿名函数  
hi
```

```
scala> def foo() = "ok"  
foo: ()String
```

```
scala> hf(foo) // eta-conversion: ()=>foo()  
ok
```

可传递

```
scala> val fun = (x:Int) => print(x)
```

```
fun: Int => Unit = <function1>
```

```
scala> fun(2)
```

```
2
```


嵌套函数

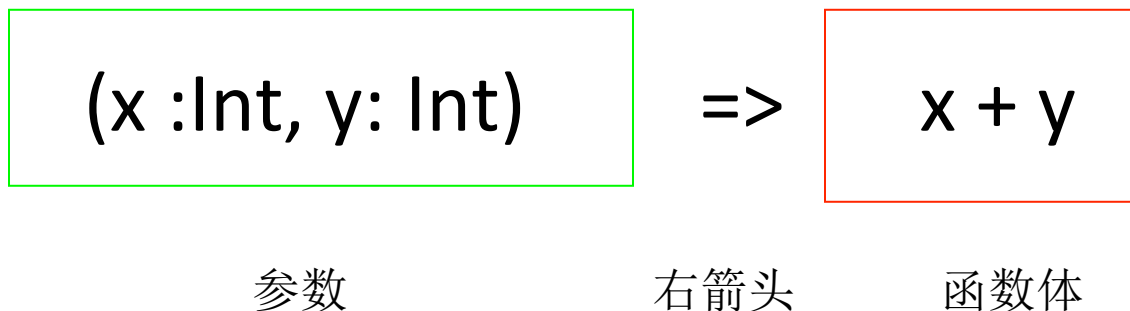
```
scala> def foo () {  
  |   def bar() {  
  |     println("hi")  
  |   }  
  |   bar  
  | }
```

嵌套函数

嵌套函数应用场景并不多，其中一个场景是
将递归函数的转为尾递归方式

匿名函数

lambda: 函数字面量(Function literal)



产生一段匿名函数，类型为 `(Int,Int)=>Int`
Scala中参数的个数为0到22个。

高阶函数



Scala里的高阶函数

第一种：用函数做参数的函数。eg:

```
scala> def f2(f: ()=>Unit) { f() }  
f2: (f: () => Unit)Unit
```

```
scala> def f1() {println(1)}  
f1: ()Unit
```

```
scala> f2(f1)  
1
```

```
scala> f2(()=>println( “hi” )) //传入匿名函数  
hi
```

Scala里的高阶函数

第二种：产生的结果是一个函数的函数。eg:

```
scala> def hf():Int=>Int = x=>x+1  
hf: ()Int => Int
```

```
scala> val fun = hf  
fun: Int => Int = <function1>
```

```
scala> fun(2)  
res0: Int = 3
```

在lambda演算中，每个表达式都代表一个只有**单独参数**的函数，这个函数的参数本身也是一个只有**单一参数**的函数，同时，函数的值是又一个只有**单一参数**的函数。

- 多个参数的函数怎么办？

柯里化(currying)



柯里化(currying)

```
scala> def sum(x:Int, y:Int) = x+y  
sum: (x: Int, y: Int)Int
```

//参数打散，两个参数分开

```
scala> def sum2(x:Int)(y:Int) = x+y  
sum2: (x: Int)(y: Int)Int
```

柯里化(currying)

```
scala> sum2(1)(2)  
res1: Int = 3
```

// 上面的调用相当于下面的几个步骤

```
scala> def first(x:Int) = (y:Int)=>x+y  
first: (x: Int)Int => Int
```

```
scala> first(1)  
res2: Int => Int = <function1>
```

```
scala> val second = first(1)  
second: Int => Int = <function1>
```

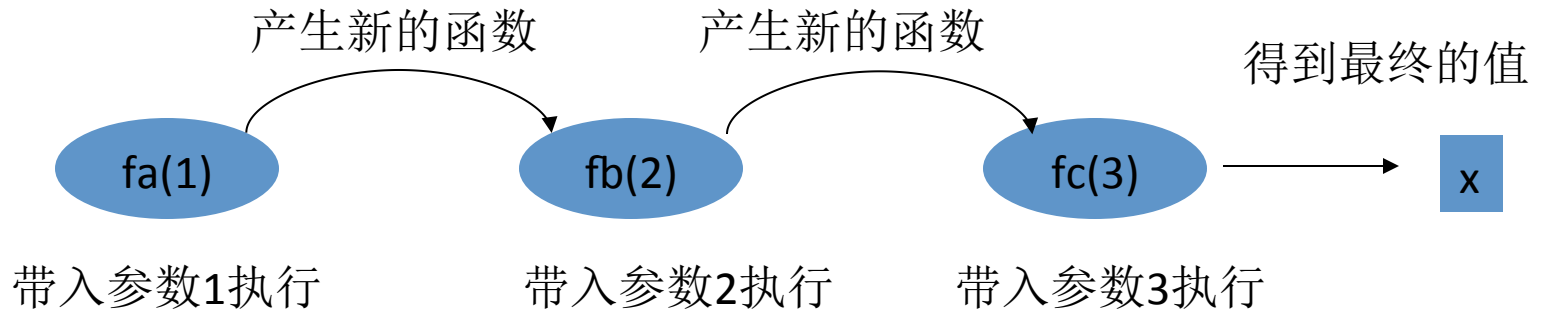
```
scala> second(2)  
res3: Int = 3
```

柯里化(currying)

- 函数链

把一个带有多个参数的函数，转换为多个只有一个参数的函数来执行

$f(1)(2)(3) \rightarrow ((f(1))(2))(3)$



柯里化(currying)

- 柯理化的实际用途？
 - 1) 控制抽象，可改变代码的书写风格。

```
foo(res, ()=>print( “test”))
```

```
foo(res)(()=>print( “test” ))
```

```
foo(res){  
  ()=>print( “test” )  
}
```

柯里化(currying)

- 柯理化的实际用途？
2) 实现部分应用函数。

部分应用函数(partial application function)

把一个函数适配为另一个函数



偏应用函数(partial application function)

占位符: _

```
scala> def pow(x:Int, y:Int) = Math.pow(x,y)  
pow: (x: Int, y: Int)Double
```

```
scala> pow(2,3)  
res4: Double = 8.0
```

```
scala> val square = pow(_:Int, 2)  
square: Int => Double = <function1>
```

```
scala> square(3)  
res5: Double = 9.0
```


部分应用函数(partial application function)

```
scala> def log(time:Date, msg:String) { println(time + ": " + msg) }  
log: (time: java.util.Date, msg: String)Unit
```

```
scala> val log2 = log(new Date, _:String)  
log2: String => Unit = <function1>
```

```
scala> log2("test1")  
scala> log2("test2")  
scala> log2("test3")
```

三次时间一样吗？

绑定的是表达式，还是表达式的结果？

部分应用函数(partial application function)

不绑定任何参数

```
scala> val pow2 = pow _  
pow2: (Int, Int) => Double = <function2>
```

闭包(closure)



Java中的匿名内部类如何访问局部变量

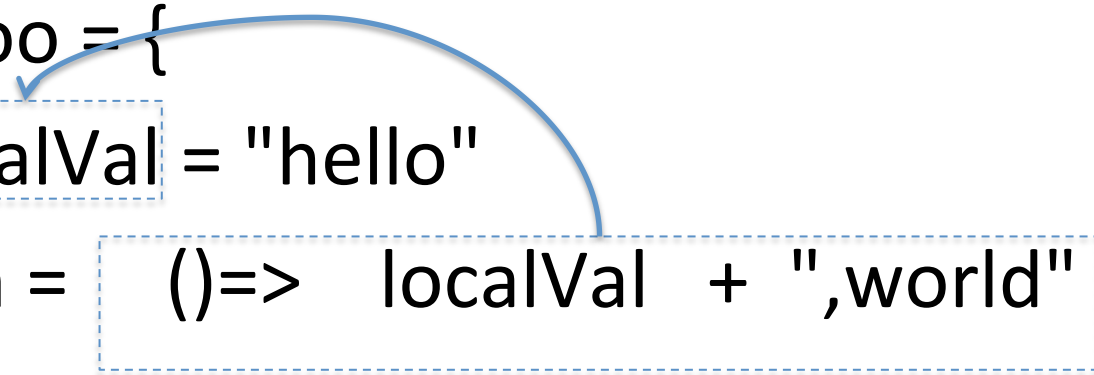
```
public Thread createThread(){  
    // 提升局部变量的生命周期  
    final int innerVar = 100;  
  
    return new Thread(){  
        public void run(){  
            System.out.println(innerVar);  
        }  
    };  
}
```

innerVar 还是分配在栈空间上么？

Java的匿名内部类，和闭包很像。但用匿名内部类来实现，前提是先要定义好该行为的接口。繁琐一些，不那么灵活

逻辑行为 + 上下文

```
scala> def foo = {  
  |   val localVal = "hello"  
  |   val fun = ()=> localVal + ",world"  
  |   fun  
  | }
```



The diagram illustrates a variable reference in the Scala code. A blue curved arrow originates from the `localVal` property access within the lambda expression `()=> localVal + ",world"` and points back to the `localVal` variable definition in the line above. Both the variable name `localVal` in the definition and the lambda expression are enclosed in dashed blue rectangular boxes.

Scala的类型系统中的函数式特征

generic types as first-class types

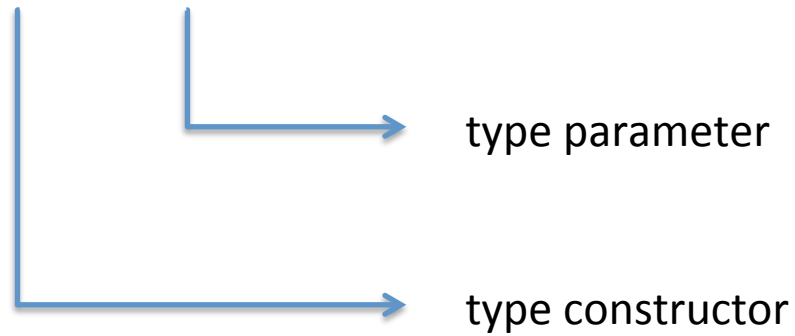
generic types as first-class types

怎么理解？

Java : `class List<T> {}`

Scala: `class List[T]`

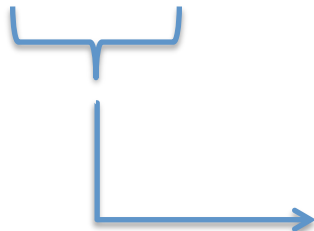
List[T]



Java : `class List2<C<T>> {}` //不支持

Scala: `class List2[C[T]]` 或
`class List2[C[_]]`

List2[C[_]]



类型参数也是一个类型构造器

generic types as first-class types

```
scala> new List2[List]
```



泛型也可以被当作类型
参数传递，与普通类型
没有区别

对类型归纳

type



Proper type(自身类型)

Int, String,
List[String]
List2[List]

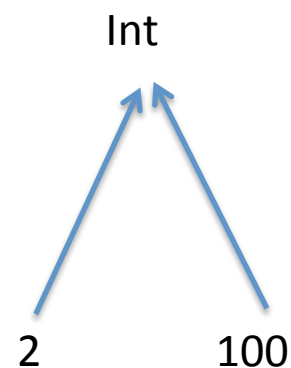
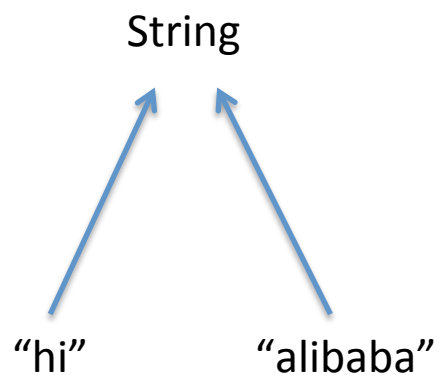
first-order/higher-order type
(一阶/高阶类型: 类型构造器)

List (一阶),
List2 (高阶)

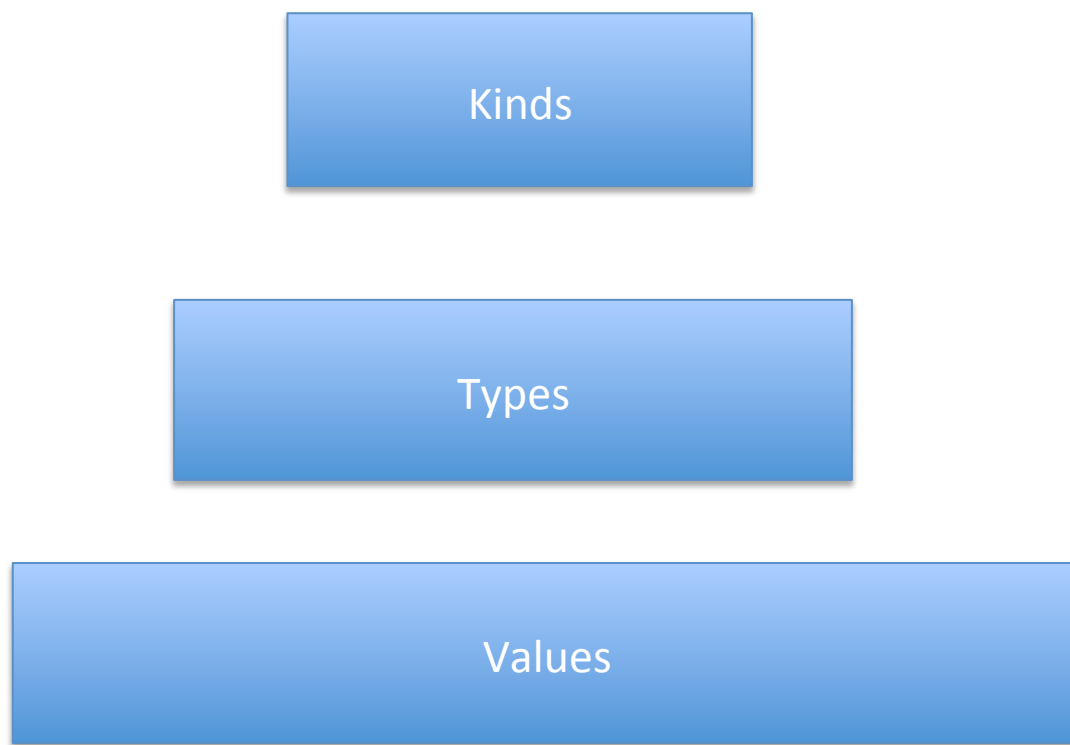
Higher-kinded type

Kind ?

类型是对数据的抽象

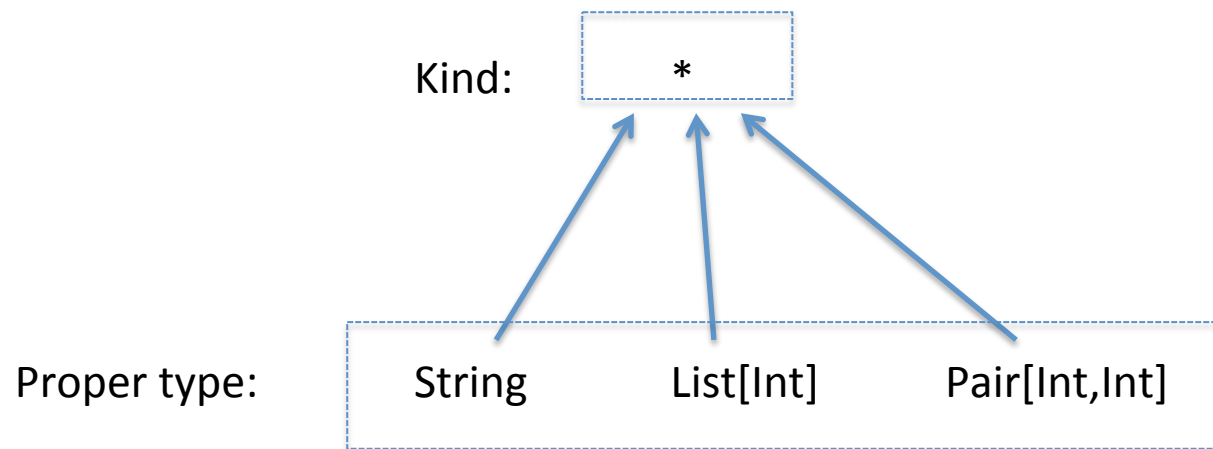


Kind: 类别，对类型的抽象



注：“类别”这个翻译不确定是不是标准

所有的proper type 被抽象为同一种 kind，用 * 表示



对一阶类型的抽象:

一阶Kind:

$* \rightarrow *$

$* \rightarrow * \rightarrow *$

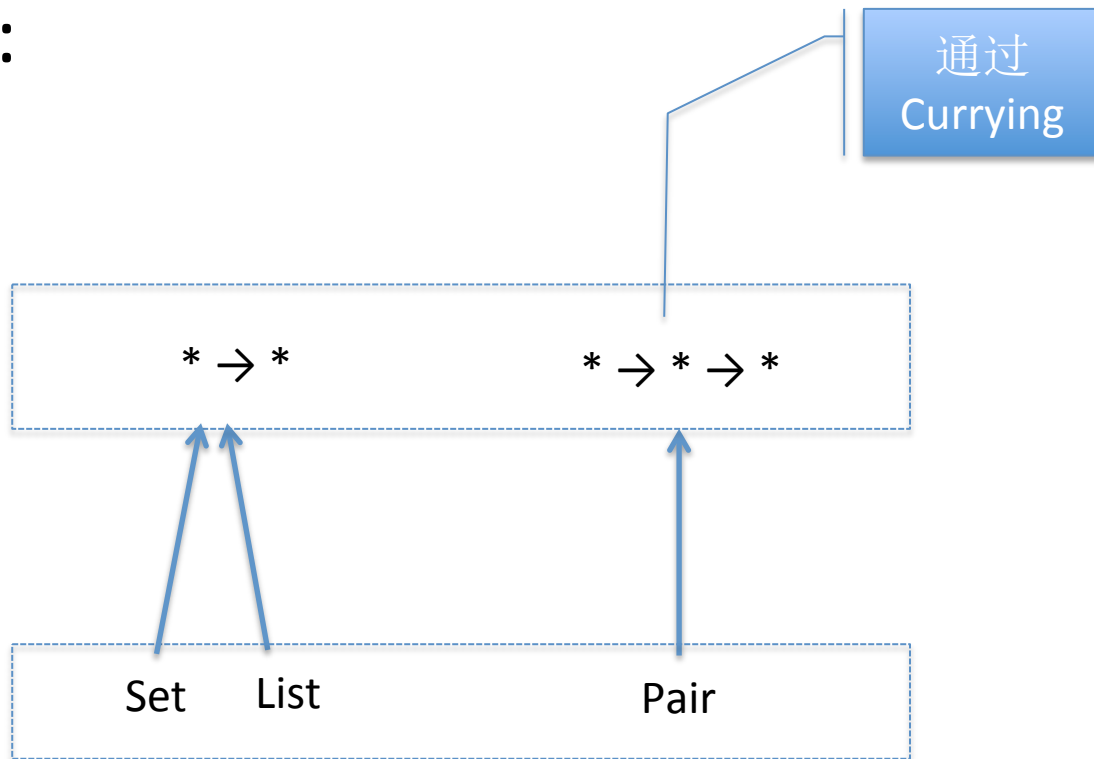
通过
Currying

一阶类型/ 类型构造器:

Set

List

Pair



对高阶类型的抽象:

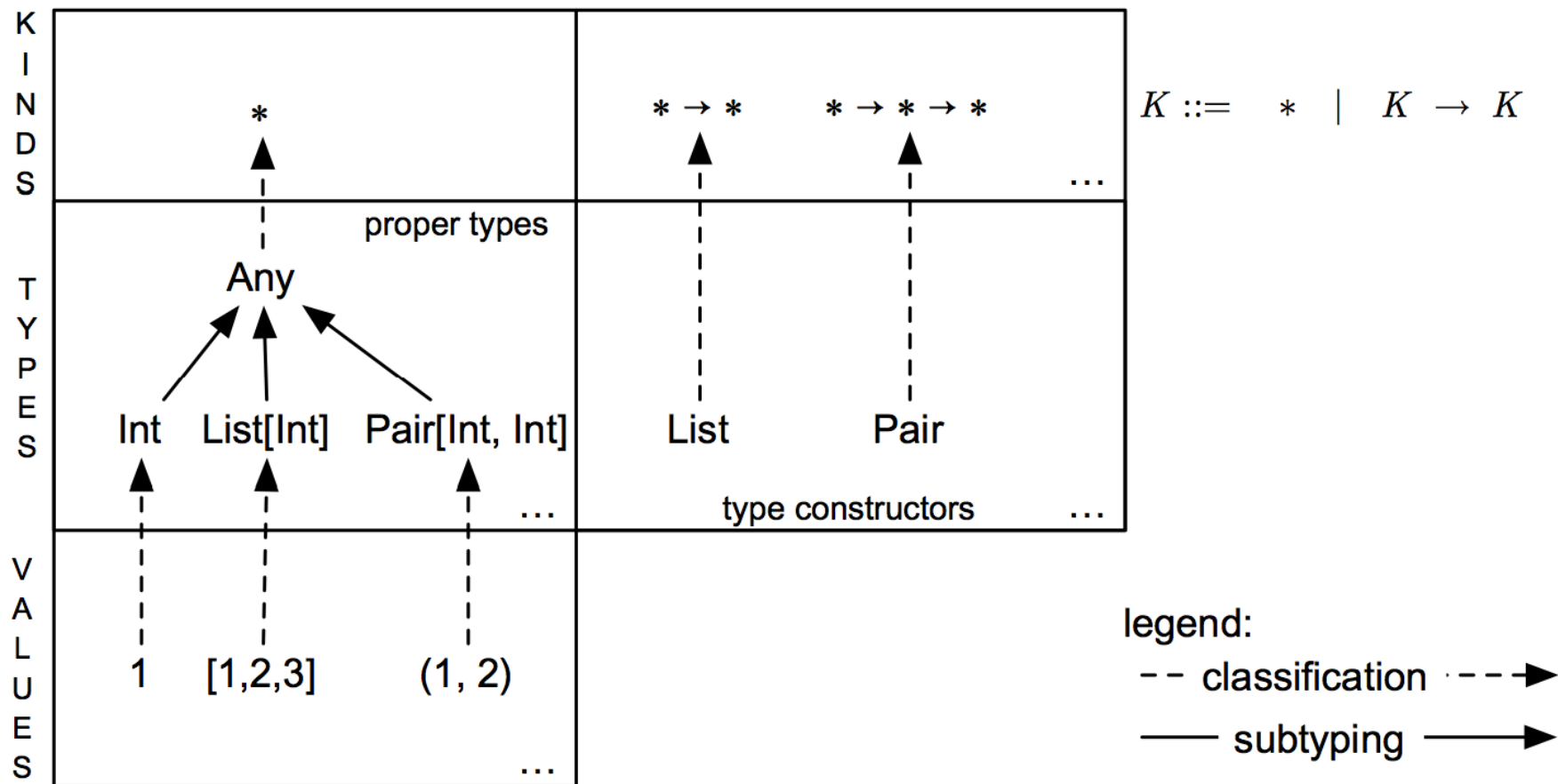
Higher-Kind:

$(* \rightarrow *) \rightarrow *$

高阶类型/ 类型构造器:

List2





图上少了higher-kind , <http://adriaanm.github.io/files/higher.pdf>

以函数的视角

```
class List[T]
```

List是一个类型构造器，类似一个一阶函数，接受一个proper type参数，并生成一个proper type.

$$\text{List} : (T) \Rightarrow \text{List}[T]$$
$$* \rightarrow *$$

```
class List2[C[_]]
```

List2类似高阶函数，接受一个类型构造器，生成一个proper type

$$\text{List2} : (T \Rightarrow C[T]) \Rightarrow \text{List2}[C]$$
$$(* \rightarrow *) \rightarrow *$$

type lambda

Scala supports a limited version of type lambda

type lambda

```
scala> def foo[M[_]] (f : M[Int]) = f
```

```
scala> foo[List] ( List(1,2,3) )
```

```
res9: List[Int] = List(1, 2, 3)
```

```
scala> foo ( List(1,2,3) )           // 类型参数可以省略，编译器会推断
```

```
res9: List[Int] = List(1, 2, 3)
```

```
scala> foo( (x:Int) => println(x) ) // 如何让这句能编译通过？
```


type lambda

`(x:Int) => println(x)` 背后是 `Function1[Int, Unit]`

```
scala> foo[ ({type X[Y] = Function1[Y, Unit]})#X ] ( (x:Int)=>print(x) )  
res5: Int => Unit = <function1>
```

Scala 的类型系统是图灵完备的，即利用类型系统本身就可以解决一些问题。

利用scala的类型系统解决汉诺塔：<https://gist.github.com/jrudolph/66925>

我们在招人

- 不限内部或外部
- 联系: hongjiang.wanghj@alibaba-inc.com

Q&A