对Java的修正和超越

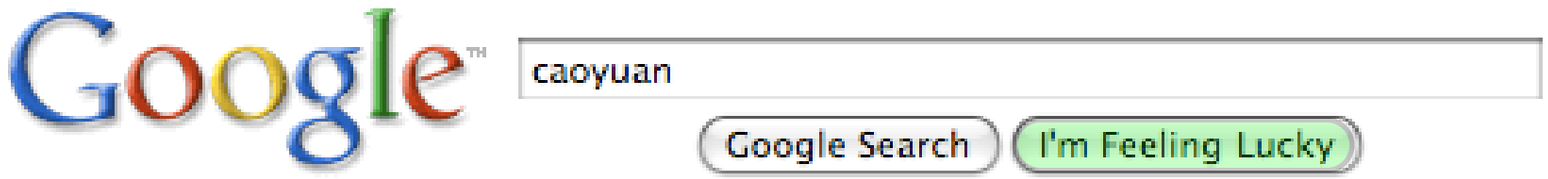
– Scala的一致性和简单性

2009.10 SD2China

Caoyuan (NetBeans Dream Team Memeber)

Regal Finance 宏爵财经资讯（北京）有限公司

http://blogtrader.net/dcaoyuan/



一切值都是对象的实例

- JVM中的原生类型是对象的实例

```
123.toByte
"1".toInt
true.toString
```

- 函数也是值，从而也是对象的实例

```
val compare = (x: Int, y: Int) => x > y
compare(1, 2) // result: Boolean = false
```

- Java的static方法和域再没有存在的理由，因为它们的所有者也必须是对象的实例(值)，所以有了Scala中的单例object

```
object Dog {
  val whatever = "dog" // static field in Java
}
class Dog {
  def callWhatever = Dog.whatever
}
```

函数作为值

- 可以当作参数传递

```
val compare = (x: Int, y: Int) => x > y
list sortWith compare
```

- 不管它是实例的方法

```
class AComparator {
  def compare(x: Int, y: Int) = x > y
}
list sortWith (new AComparator).compare
```

- 还是匿名子句

```
object annonymous extends scala.Function2[Int, Int, Boolean] {
  override def apply(x: Int, y: Int) = x > y
}
list sortWith annonymous
```

- 没有了static域，一切函数，包括object的方法调用现在都是值(实例)的方法

一切操作都是函数调用(1)

- 只有一个参数或零个参数的方法在调用时可以省略”.”和”()”

```
1.+(1)
1 + 1
1.>(0)
1 > 0
(1 > 0).&(2 > 1)
(1 > 0) & 2 > 1
stack.push(10)
stack push 10
stack.pop
stack pop
```

- 参数也可以是一系列操作{...}

```
stack push {
  val a = 1
  val b = 2
  a + b
}
```

- 更多的符号需要用作方法名

```
def !@#%^&*\-<=>?|~:/ = println("noop")
def √(x: Double) = Math.sqrt(x)
val π = Math.Pi
val r = √(9*π)
```

- ‘<’, ‘>’更适合作方法名，所以用’[’和‘]’来表示类型参数

一切操作都是函数调用(2)

- for语句是函数调用

```
for (i <- List(1, 2)) {
  println(i)
}
List(1, 2) foreach {i => println(i)}

for (i <- List(1, 2)) yield {
  i + 10
}
List(1, 2) map {i => i + 10}
```

- 更多的例子

```
// synchronized is function call instead of keyword
def check = synchronized {
  // isInstanceOf is function call instead of keyword
  100.isInstanceOf[String]
}
```

- 额外的好处：自左向右顺序书写语句

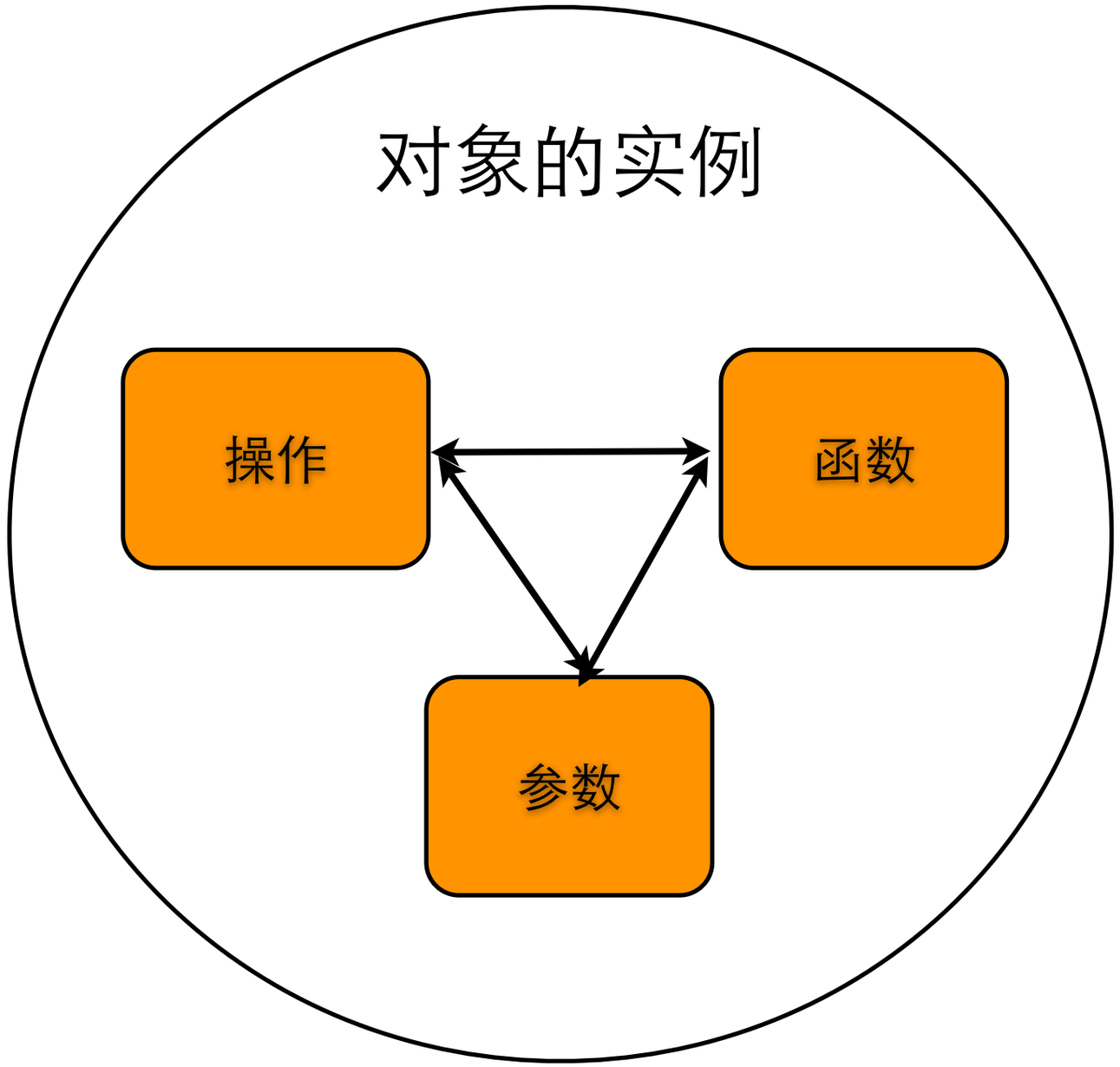
```
stack.pop.asInstanceOf[Int] // (Integer) stack.pop() in Java
```

一切操作都返回值

- 默认返回最后一条语句的值，也可以用return显式返回

```
val r1 = { // return 3
  val a = 1
  val b = 2
  a + b
}

val r2 = if (true) 1 else 2

val r3 = // return (): Unit
  for (i <- List(1, 2)) {
    println(i)
  }

val r4 = // return List(11, 12)
  for (i <- List(1, 2)) yield {
    i + 10
  }

val r5 = // return java.io.File
  try {
    val f = new File("afile")
    f
  } catch {
    case ex: IOException => null
  }
```

Scala的语句是怎么构成的



数据即操作
操作即数据

+

数据(值)是
对象的实例

=

OO + FP

接下来的问题是

怎样把对象的类型搞清楚

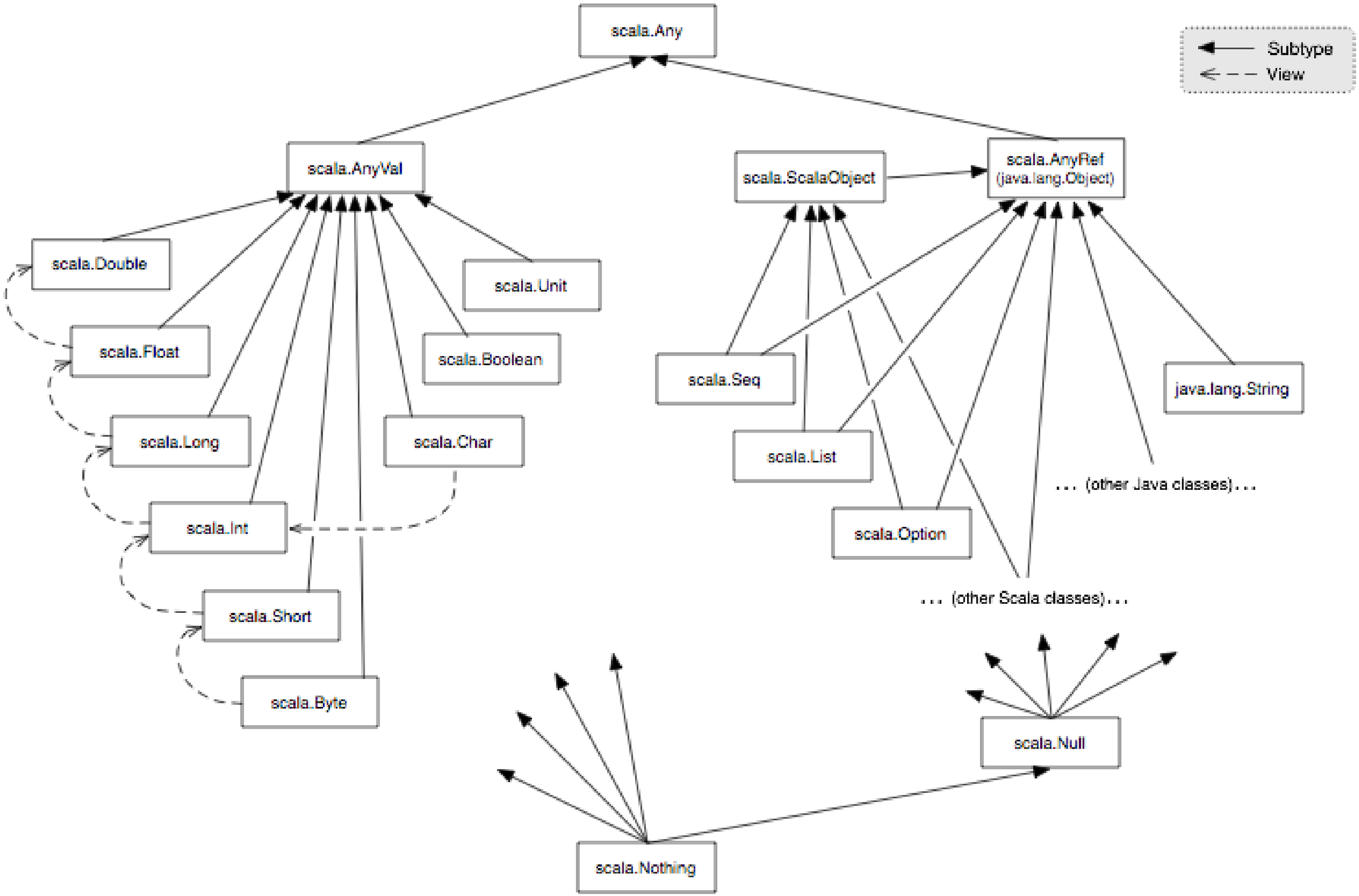
完整一致的类型体系(1)

- Scala上有一个共同的根类Any，Any统一了JVM上原本没有共同超类的原生类型和引用类型
  - JVM上的原生类型(byte, short, char, int, long, float, double, boolean)和void
  - 在Scala中对应(Byte, Short, Char, Int, Long, Float, Double, Boolean)和Unit，是AnyVal的子类，不再需要装箱类型
  - JVM上的引用类型是AnyRef (java.lang.Object)

```
// Scala, Array不再有8种写法
val arrayOfListOfString = new Array[List[String]](10)
val arrayOfAny: Array[_] = Array(1, 2)

// Java
List<String>[] arrayOfListOfString = new ArrayList<String>[10]
```

- Scala上有两个特殊的尾类：Null和Nothing
  - JVM上的null引用是Null类，它是所有AnyRef的子类
  - Nothing是Scala上的所有类型(包括AnyVal和AnyRef)的子类，它没有值，用于处理一些特殊情况，比如出错时的非正常返回类型

完整一致的类型体系(1)


scala.Any
Subtype
View
scala.AnyVal
scala.AnyRef
(java.lang.Object)
scala.ScalaObject
scala.Double
scala.Float
scala.Long
scala.Int
scala.Short
scala.Byte
scala.Char
scala.Boolean
scala.Unit
scala.Seq
scala.List
scala.Option
java.lang.String
... (other Java classes)...
... (other Scala classes)...
scala.Null
scala.Nothing

完整一致的类型体系(2)
-泛型与类型参数的随变

- 不变(Invariant): C[T], C is invariant on T
  如果Tsub或Tsup是T的子类或超类，C[Tsub]或C[Tsup]都不是C[T]类

- 协变(Covariant): C[+T], C is covariant on T
  如果Tsub是T的子类，则C[Tsub]也是C[T]的子类

- 逆变(Contravatiant): C[-T], C is contravariant on T
  如果Tsup是T的超类，则C[Tsup]也是C[T]的子类

Java让调用者在调用时再通过通配符?处理;
Scala鼓励设计者在定义时就通过+/-规定

完整一致的类型体系(2)

```
class Animal {}
class Dog extends Animal {}
class Fox extends Animal {}
```

- 如果Animal是神仙，Animal的子类都是吗?

```
class Xian<T> {} // Java

void isAnimalXian(Xian<? extends Animal> in)

isAnimalXian(new Xian<Dog>())
isAnimalXian(new Xian<Fox>())
```

```
class Xian[+T]  // 协变

def isAnimalXian(in: Xian[Animal])

isAnimalXian(new Xian[Dog])
isAnimalXian(new Xian[Fox])
```

- 如果Dog是神仙，Dog的超类也是吗?

```
class Xian<T> {} // Java

void isDogXian(Xian<? super Dog> in)

isDogXian(new Xian[Animal])
```

```
class Xian[-T]  // 逆变

def isDogXian(in: Xian[Dog])

isDogXian(new Xian[Animal])
```

类型参数vs抽象类型成员

- 类型参数

```
trait Pair[K, V] {
  def get(key: K): V
  def set(key: K, value: V)
}
```

```
class PairImpl extends Pair[Dog, Fox] {
  def get(key: Dog): Fox
  def put(key: Dog, value: Fox)
}
```

类型参数的签名会发散到所有的子类和引用的类 => 改变类型名时要修改许多地方

- 抽象类型成员：缓定类型+类型成员+类型别名

```
trait Pair {
  type K // deferred type
  type V // deferred type

  def get(key: K): V
  def set(key: K, value: V)
}
```

```
class PairImpl extends Pair {
  type K = Dog
  type V = Fox

  def get(key: K): V
  def set(key: K, value: V)
}
```

抽象类型使得子类的所有方法签名完全一致 => 改变类型名时只需改一处

- 类型成员 => 总体/部分模式下更自然的抽象
- 类型别名 => 更简洁的代码：type StrToList = HashMap[String, List[String]]

Trait和Structural Type

- Trait: 接口+可选的实现代码 => 通过混入实现代码重用
- Structural Type: 不要关心我叫什么名字，只要具有清单上的所有特征，我们就是同一类

构件Type，没有类名而只有抽象成员的类，又叫匿名类型

```
trait Subject {
  type Observer = {def update(subject: Subject)} // type member

  private var observers = List[Observer]()   // field
  def addObserver(observer: Observer) = observers ::= observer
  def notifyObservers = observers foreach (_.update(this))
}

class Price(var value: Double) {def setValue(v: Double) = value = v}
```

与Java的Interface不同
Trait可以随时混入

```
val observablePrice = new Price(10) with Subject {
  override def setValue(v: Double) = {
    super.setValue(v); notifyObservers
  }
}
```

实现了update(Subject)，就是Observer类

```
observablePrice.addObserver(new AnyRef {
    def update(subject: Subject) {
      println("price changed")
    }
  })
```

Trait和Self Type

- 多重继承并非洪水猛兽，关键是如何处处保证类型安全：
  - 只能扩展一个Class，但可以混入许多Trait: OilPrice extends Price with Subject with Market with ....
  - 线性顺序覆盖
  - 反身类型(Self Type)

```
trait Subject {self: Price =>
  type Observer = {def update(subject: Price)} // type member

  private var observers = List[Observer]()   // field
  def addObserver(observer: Observer) = observers ::= observer
  def notifyObservers = observers foreach (_.update(this))
}
```

this并非一定要是Subject

```
observablePrice.addObserver(new AnyRef { // only need a update(Any) method
    def update(subject: Price) {
      println("price changed to " + subject.value)
    }
  })
```

- Self Type:
  - 约定Trait的上接口（除了自己提供服务，还可以要求服务）
  - 可以看作包含了所有要求的服务（class或trait）的引用，从而使用它们的成员类型、域和方法
  - 指定self后，Trait的super保持原来的含义，this则指向self同一个引用。（self可以取成其它的任何名字）
  - 为什么不通过一个共同的Trait实现上接口?
    - 避免编译时的循环依赖
    - 更多的灵活性，比如，上接口可以是class，而非一定是trait (绕开只能扩展一个Class限制)

有一类方法不妨都叫apply

- 名字为apply的方法在调用时可以用<instance>(args)代替<instance>.apply(args)
- apply方法常用于实例取值，与Class的构造方法配合来生产实例（工厂模式），初始化实例等

```
class Price(var value: Int) {
  def apply() = value  // ‘def apply = value’, without ’()’ is bad idea
}

object Price {
  def apply(v: Int) = new Price(v)
}

val p = Price(10)
p           // Price@565f0e7d
p()         // 10
Price(10)() // 10
```

- apply方法在Scala标准库中的例子

```
val list = List(1, 2, 3)          // List.apply[A](xs: A*)
val array = Array("a", "b", "c") // Array.apply[T](xs: T*)
val map = Map("a" -> 1,
              "b" -> 2,
              "c" -> 3) // MapFactory.apply[A, B](elems: (A, B)*)

array(0) // array.apply(index): "a"
map("a") // map.apply(key): 1
```

类型匹配、派发的一致性

- Java的Switch只作用于整数和枚举(字符串 – Java7)，Scala中的模式匹配则扩展到了所有类型

```
val str = "abcdef"
str.toSeq match {
  case Seq('a', 'b', rest@_*) => println(rest) // cdef
  case _ => println("no match")
}

var any: Any = _
any match {
  case "scala" | "java" => "string"
  case i: Int if i > 10 => "int > 10"
  case `str` => "abcdef"
  case _: String => "string type"
  case hd :: tail => "List"
  case _ => "other"
}

case class Address(coutry: String, city: String, street: String, no: Int)
case class Person(name: String, age: Int, address: Address)

val p = Person("Wang wu", 12, Address("China", "Beijing", "Chang'an", 10))

p match {
  case Person(name, age, Address(coutry, city, street, no)) =>
    println(name + "|" + coutry + "|" + city + "|" + street + "|" + no)
}
```

Scala的其它特性

- Existential type (为了与Java兼容)
- Implicit type * (就算你没有这个方法，也可以从外部强加给你)
- val, var, lazy val (不变的东西总是更容易处理)
- Partial function * (有些情况到用时再判断，然后，还是可以处理)
- Primary constructor (构造器现在有了总入口，复杂情况不妨apply)
- Accessor和properties (Why getter, setter? 本来就不该那么复杂)
- XML Process (内建支持)

* 提供DSL能力的重要手段

通过类库扩展语言(DSL能力)

- Actors库

```
class Ping(count: Int, pong: Actor) extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! Ping
    while (true) {
     receive {
        case Pong =>
          if (pingsLeft % 1000 == 0)
            println("Pong")
          if (pingsLeft > 0) {
            pong ! Ping
            pingsLeft -= 1
          } else {
            pong ! Stop
            exit()
          }}}}}
```

操作就是方法调用
参数可以是{...}

- Bill Venners的scalatest

```
class StackSpec extends FlatSpec with ShouldMatchers {

  "A Stack" should "pop values in last-in-first-out order" in {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    stack.pop() should equal (2)
    stack.pop() should equal (1)
  }

  it should "throw NoSuchElementException if an empty stack is popped" in {
    val emptyStack = new Stack[String]
    evaluating { emptyStack.pop() } should produce [NoSuchElementException]
  }
}
```

隐式类型转换
所以可以有should方法

Scala会成为Java的继任者吗?

会!

Scala比Java更简单:

- Scala的语言规范比Java更简单，因为一致性意味着更少的特殊情况
  - Array、泛型
- Scala要解决的问题，Java同样面对，但Java解决的方式更复杂、更曲折 – 各种设计模式
  - 函数、单例object、模式匹配
- Scala的扩展性表现在通过类库来扩展语言能力，而不是加入更多的语法要素或改写编译器
  - Actors库、Packrat Parser库

更重要的是:

Scala的一致性 + 扩展性
会再次激发出程序员的创造力

Scala会成为Java的继任者吗?

但是:

- Scala本身的开发过程需要加强
- 更多的Java基本库被移植到Scala(比如nio库)
- 更多的Scala应用实例(比如liftweb)
- IDE的支持

IDE现状

- IntelliJ Idea
  - 5个开发人员
  - 自写的语法解析器和类型标注器等

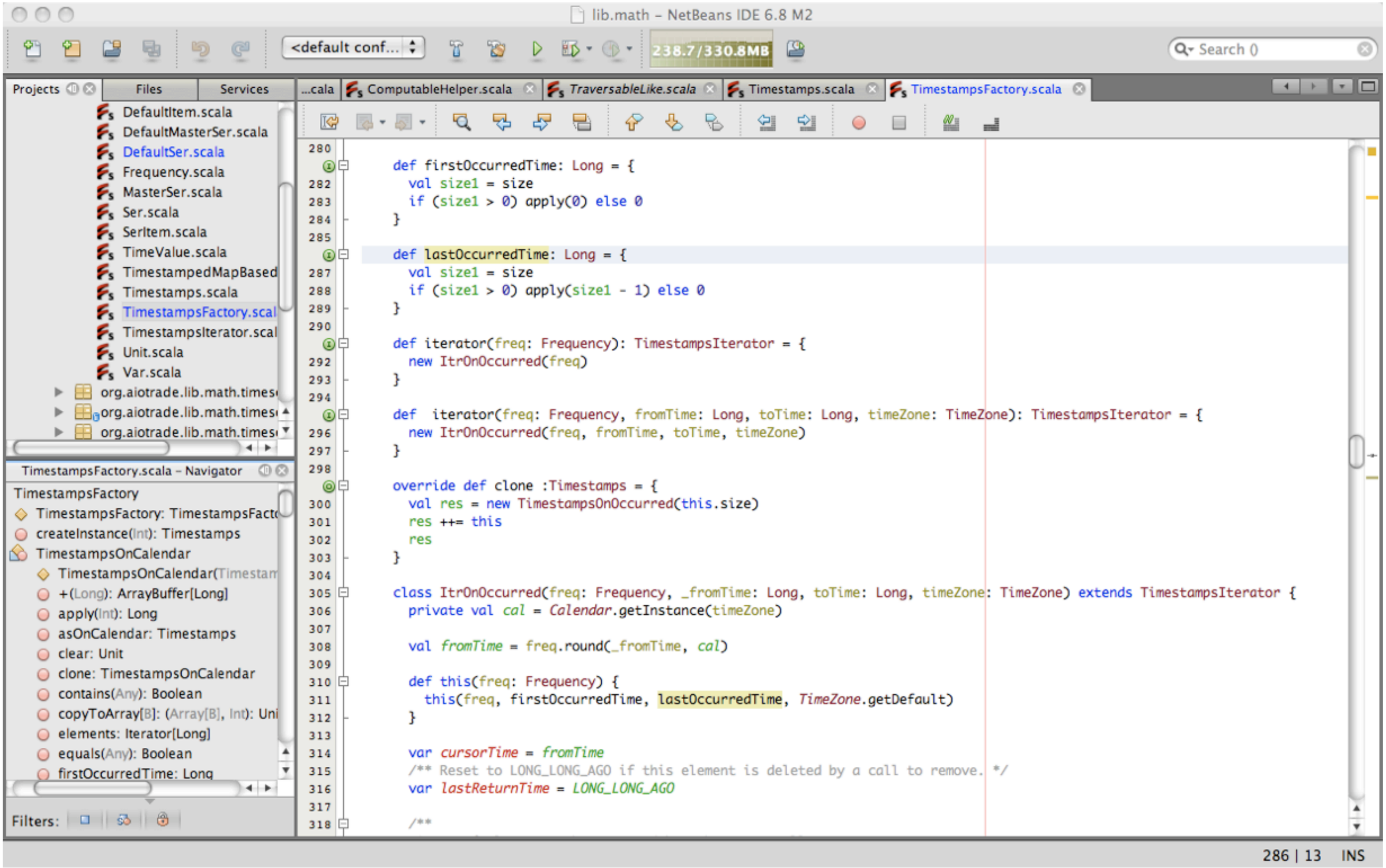
- Eclipse
  - 前期：Sean McDirmid
  - 现在：1个全职开发人员(Miles Sabin)
  - Scala自身的编译器 + Martin为IDE支持所做的改进

- NetBeans
  - 利用业余时间开发(dcaoyuan)
  - 2007年，LL(K)语法解析器 + 自写的类型标注器
  - 2008年上半年，PEGs语法解析器 + 自写的类型标注器
  - 2008年下半年，Scala自身的编译器 + Hacking，NetBeans Innovation Grant 金奖
  - 2009年8月，全部用Scala改写，Scala自身的编译器 + Martin为IDE支持所做的改进

IDE现状-NetBeans

- 功能一览
  - 代码高亮
  - 语法错误即时提示
  - 大纲导航
  - 代码折叠
  - 引用处处标注
  - 跳转到定义处
  - 即时重命名
  - 重构：重命名和引用查找（跨打开的项目）
  - 代码自动缩进和格式化
  - 输出到HTML文件，保持代码高亮
  - 自动代码提示、补齐
  - 文挡提示
  - Java/Scala混合项目
  - 重载标记和跳转到被重载的定义处
  - 自动修正import
  - 代码模版
  - 断点处处可设的调试器
  - 与Maven项目的结合
- 版本发布
  - NetBeans 6.7.x上的版本四个月下载约7000套
  - NetBeans 6.8m2上的版本针对Scala-2.8.0，已发布beta版本
    http://wiki.netbeans.org/Scala68v1

Scala for NetBeans



Q & A