# 泛函编程

一种关注类型的编程

# Scala的历史

2003年，Martin Odersky 因为感到Java源代码级别的兼容性太难以满足，在完成了Java 1.5 的泛型（Generic）开发工作之后，发布了基于JVM运行的Scala语言。

# 程序开发中，往往需要不断地开发一些底层功能

对一个已经分好组的数据进行分类求解，这一类要求在各行各业经常都会看到。比如对账户资金的流入流出进行分解，对不同队伍游戏角色进行分类等等。这些程序不但繁杂，易错，而且不断打断程序员对业务逻辑的思考。

因此我们才需要通过重用，把程序员从底层功能的开发中解放出来。

# 开闭原则（OCP）

数据是资产，所以要对资产进行保护，不能随意修改。Immutable是对数据的强保护。

行为是工具，工具要随时换新，以便能够及时欢迎设计更好，效率更高的工具。

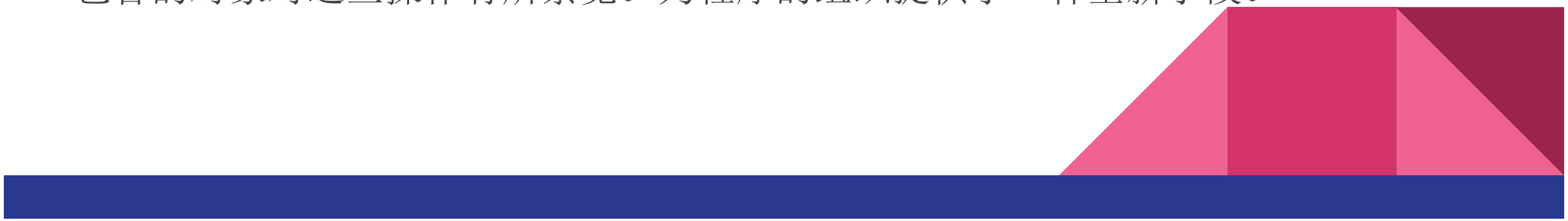
举例来讲就是List里面的数据，以及数据的类型都要得到很好的保护。而sort这样的方法，则要考虑允许引入更好的算法。可以随时扩充新的方法，随时更新旧的实现而不用去改动源代码。

# 容器与类的区别（Container vs Class)

OO Class着眼于对数据的封装，和对行为的绑定。

泛函编程的容器，在类的基础上进一步增加了容器元素的类型定义。容器所容纳的不仅是数据，也可以容纳函数（函数也是值的概念）。容器的设计概念，是为所存储的对象，提供一种解耦后的公共操作。

比如，ZIO这个容器，对它所包含的对象，提供了运行环境（单，多线程），网络，文件或者Console，错误重试，错误处理等等一系列的公共服务。同时不需要所包含的对象对这些操作有所察觉。为程序的组织提供了一种全新手段。



# 高内聚，低耦合的革命

本着获得最少知识原则（least known），对类型抽象程度越高的那些函数，其完成的功能越单一，对调用者的要求越少，可被重复使用的可能性越大。

对类型的抽象程度越低的函数，其获得的功能越多，可以完成的任务越复杂。但同时对调用者的要求越高，可被重复使用的可能性越低。

OO因为缺乏对类型的抽象，所以它制作出可被广泛复用的工具难度很高。

# 控制类型关系，使得泛函编程的关注点发生变化

Scala提供了很多对类型限定的方法。规定了类型之间的相互关系，使得对类型抽象的研究成为了泛函编程的一种重要考量。

[A <: B, A >: B, A <% B, A >% B, {def ...}, F[_], +A, -A, ({ type T[A] = Map[Int, A] })#T]

# 函数式编程

函数式编程中的概念，Monoid，Monad，Applicative，Functor，Arrow这些函数式编程的概念在泛函编程中的确有着重要地位。但是泛函编程并不拘泥于只有这些应用。

# 对类型的关注和保护是泛函编程与OO的最大区别

类型提供了编译期优化程序和发现逻辑缺陷的更强能力。

类型擦除是OO解决多态问题的重要手段，但这种手段破坏了泛函编程带来的优点。

隐式转换+类型抽象+Lambda函数的使用，在实现对接口概念的全面替代的基础上，提供了一个更好的保存对象类型的解决方案。

辟谣，Lambda绝不是函数式编程的精髓，它只是一只小工具。类型系统才是我们的主要着眼点。泛函的继承，不是为了使用父类所提供的工具，而是为了说明类型之间的关系，和OO的思路有着本质的区别。

# 其他新概念

Lazy的好处

Immutable的好处

容器的好处

# 发展远未结束

Scala的种种缺陷：

容器的类型擦除依然存在

泛型定义的不可枚举性

编译效率的进一步提高

# Q&A