

函数式编程原理

顾琳

微课堂签到规则



二维码有效期至: 2023-10-13

刷新

课堂名称: 函数式编程

课堂编号: LK861

1、扫码关注公众号: 微助教服务号。

2、点击系统通知: “[点击此处加入【函数式编程】课堂](#)”, 填写学生资料加入课堂。

*如未成功收到系统通知, 请点击公众号下方“学生” - “全部(A)” - “加入课堂” --- “输入课堂编号”手动加入课堂

- <https://www.teachermate.com.cn/classes/1316409>

头歌实验环境

顾琳老师邀请您加入头歌平台教学课堂-《函数式编程原理2023秋季》
您可以访问下方的链接，以学生身份加入该教学课堂

邀请码：ZCRW4

链接：<https://www.educoder.net/classrooms/21364?code=ZCRW4>

注意：按时提交，情况说明

计分包括：平时作业、课堂实验，随堂小测，课程报告

函数式编程原理

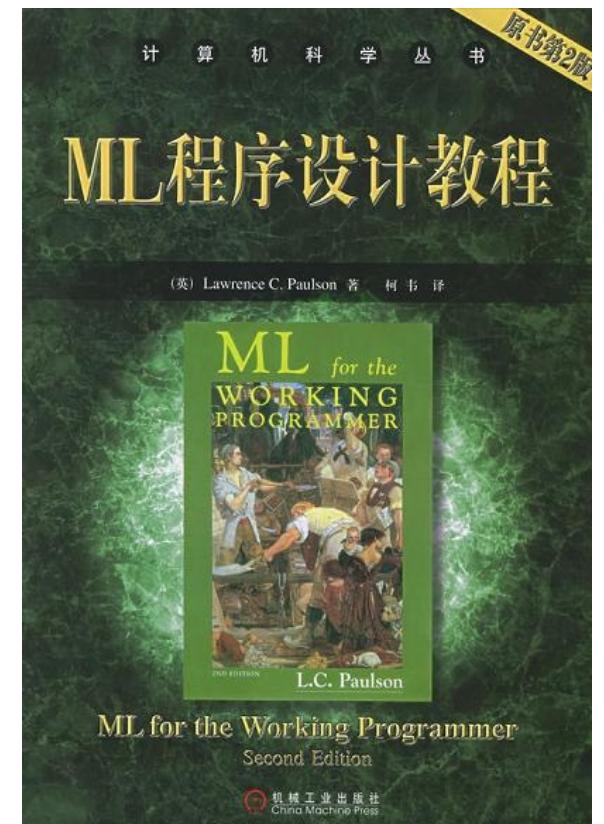
- 学时要求： 16+16
- 授课方式：讲授+上机
- 参考资料：CMU 15150
- 涉及内容：
 - 程序正确性/有效性证明
 - 类型, 声明, 表达式, 函数
 - 递归, 模式匹配, 多态类型检测
 - 高阶函数, 惰性求值
 -

- 课程目标：

- 掌握函数式编程方法
- 掌握程序书写规范，并采用严格的推导方法证明程序的正确性
- 掌握串/并行程序的性能分析方法
- 掌握各种数据结构的特点，学会选择合适的数据结构进行功能设计，提高程序效率

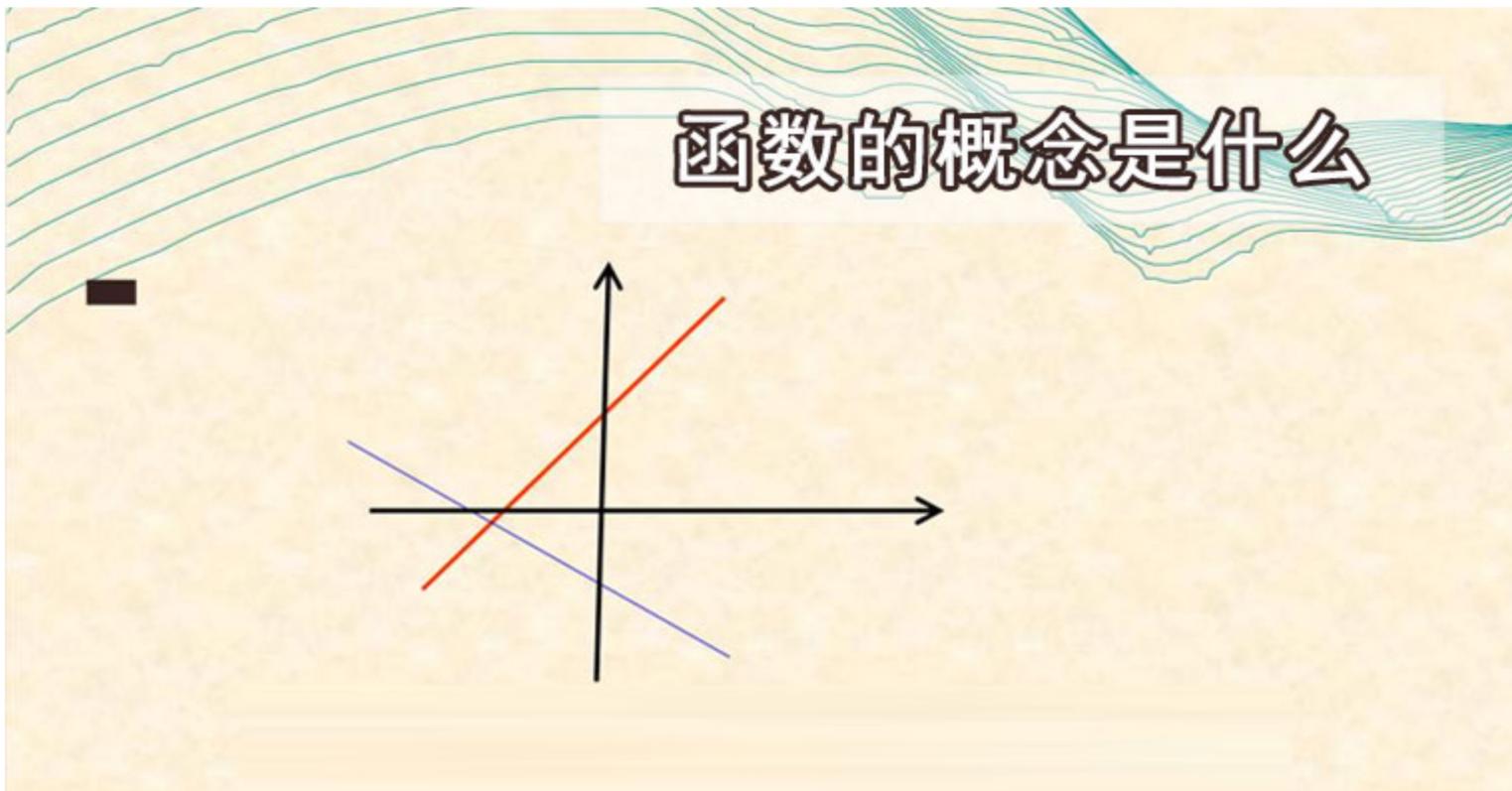
参考资料

- 以CMU 15-150课程相关文档为主
 - Lectures、Slice、Labs…
- ML程序设计教程(第2版)，机械工业出版社



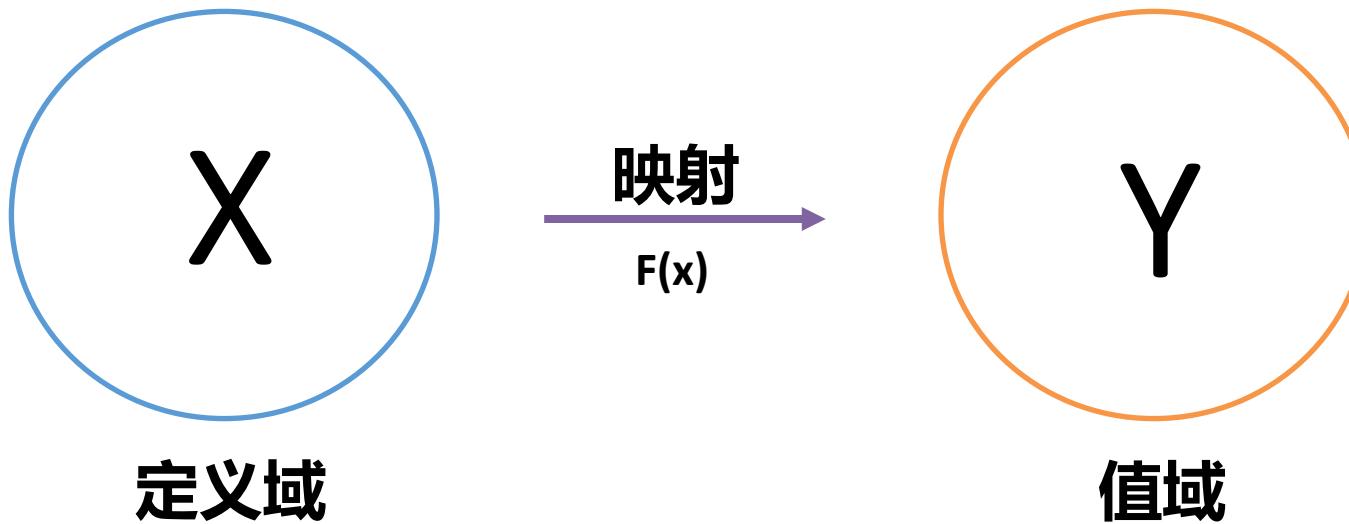
什么是函数？

函数的定义是给定一个数集A，假设其中的元素为x，对A中的元素x施加对应法则f，记作 $f(x)$ ，得到另一数集B，假设B中的元素为y，则y与x之间的等量关系可以用 $y=f(x)$ 表示。



函数概念含有三个要素：定义域A、值域C和对应法则f。其中核心是对应法则f，它是函数关系的本质特征。

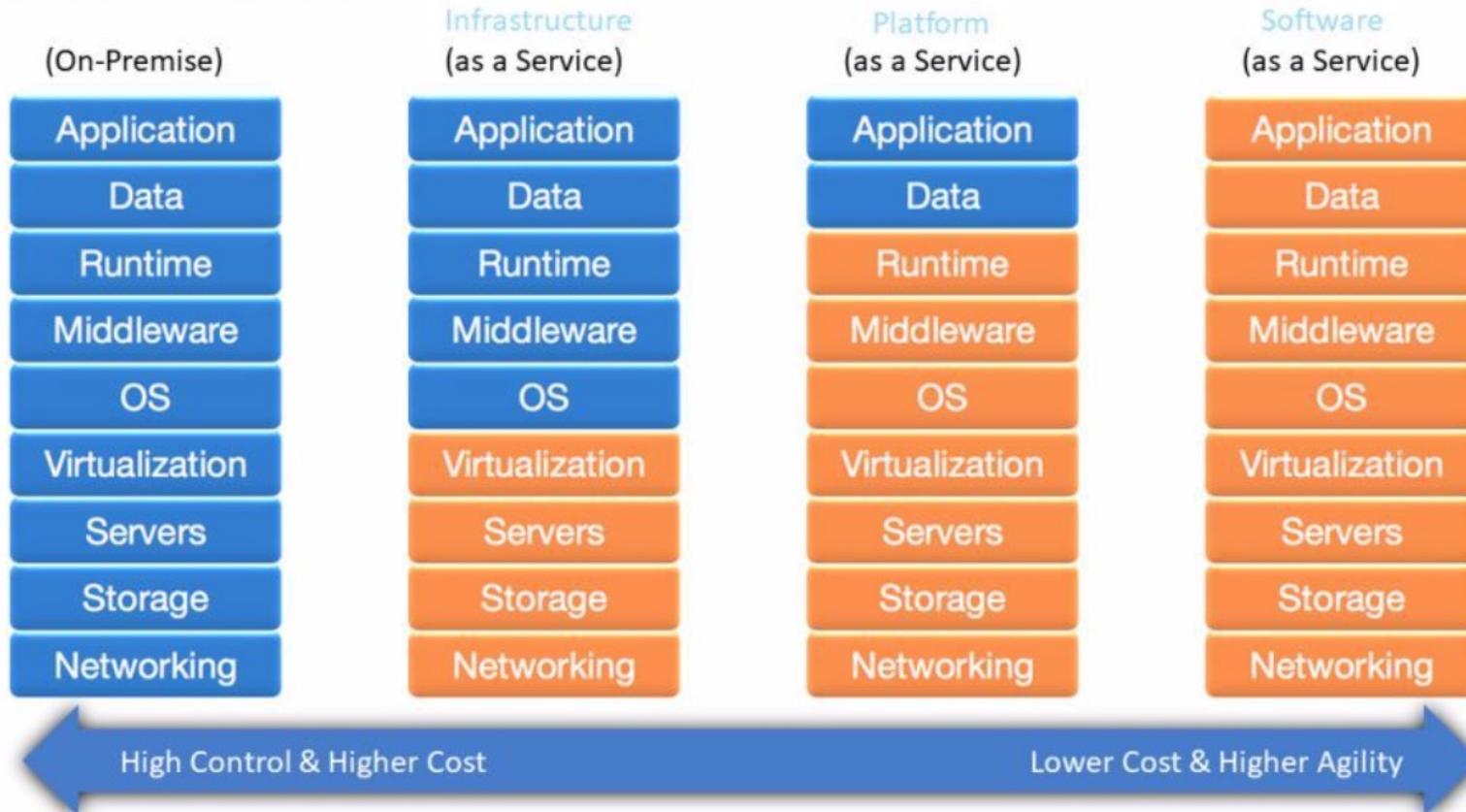
什么是函数？



生活和学习中有无数的函数

如果 $F(x)$ 太复杂怎么办？

函数有什么用？



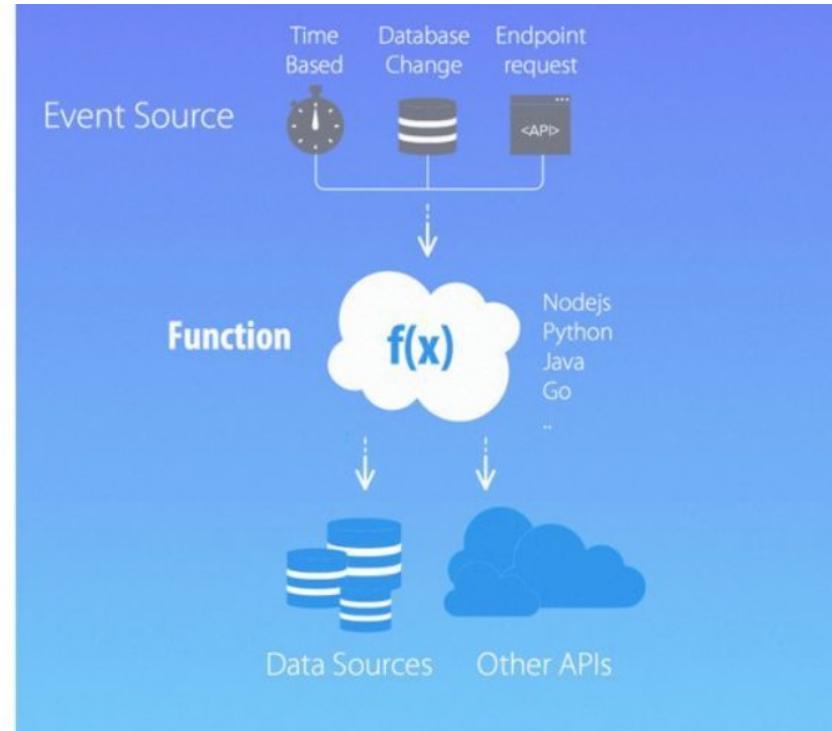
模块化，平台化，透明化 → 简单易用，功能固化

函数有什么用？

Functions as a Service

“Functions as a Service”即FaaS，指这样的应用，一部分服务逻辑由应用实现，但是跟传统架构不同在于，他们运行于无状态的容器中，可以由事件触发，短暂的，功能上FaaS就是不需要关心后台服务器或者应用服务，只需关心自己的代码即可。

其中AWS Lambda是目前最佳的FaaS实现之一。



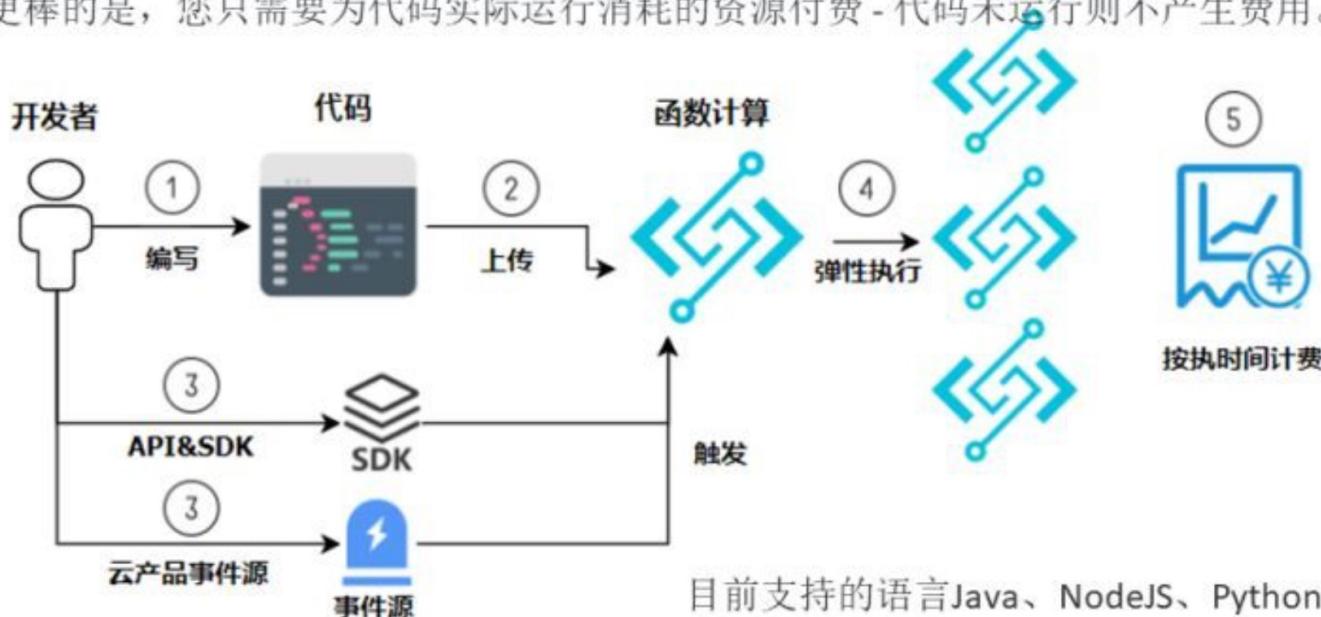
FaaS的出现也是服务发展的必然趋势，在微服务大行其道的现在，FaaS作为服务函数的出现，在服务细粒度实现和计费上给出了很好的解决方案。目前这块属AWS的lambda做的最好了。

下面可以看看FaaS一些常用的场景，目前这些场景大部分我在内部已经落地使用，效果非常不错，在之前看起来非常复杂的系统，现在有了FaaS，开发效率提升非常明显。另外FaaS在开发MVP中有非常大的优势。

函数有什么用？

aliyun 函数计算

阿里云函数计算，是一个事件驱动的全托管计算服务。通过函数计算，您无需管理服务器等基础设施，只需编写代码并上传。函数计算会为您准备好计算资源，以弹性、可靠的方式运行您的代码，并提供日志查询，性能监控，报警等功能。借助于函数计算，您可以快速构建任何类型的应用和服务，无需管理和运维。更棒的是，您只需要为代码实际运行消耗的资源付费 - 代码未运行则不产生费用。



目前支持的语言Java、NodeJS、Python等语言，[详见Node.js](#)。

函数有什么用？

腾讯云无服务器云函数

无服务器云函数（Serverless Cloud Function，SCF）是腾讯云为企业和开发者们提供的无服务器执行环境，帮助您在无需购买和管理服务器的情况下运行代码。您只需使用平台支持的语言编写核心代码并设置代码运行的条件，即可在腾讯云基础设施上弹性、安全地运行代码。SCF 是实时文件处理和数据处理等场景下理想的计算平台。

目前支持的语言NodeJS、Python等语言。

接口功能	Action ID	功能描述
获取函数列表	ListFunctions	用于查询获取当前地域下函数列表
获取函数详情	GetFunction	用于查询获取函数详细信息
创建函数	CreateFunction	用于创建函数
删除函数	DeleteFunction	用于删除指定函数
更新函数	UpdateFunction	用于更新指定函数配置信息，或更新指定函数代码
获取函数运行日志	GetFunctionLogs	用于查询获取指定函数的运行日志
获取函数监控数据	GetMonitorData	用于查询获取指定函数的监控信息
运行函数	InvokeFunction	用于运行指定函数

触发器相关接口

接口功能	Action ID	功能描述
设置函数触发器	SetTrigger	用于设置指定函数的触发器
删除函数触发器	DeleteTrigger	用于删除指定函数的指定触发器

感受一下...

程序1:

```
int a, b, r;
void add_abs() {
    scanf("%d %d", &a, &b);
    r = abs(a) + abs(b)
    printf("%d", r);
}
```

程序2:

```
int add_abs(int a, int b) {
    int r = 0;
    r += abs(a);
    r += abs(b);
    return r;
}
```

程序3:

```
int add_abs(int a, int b) {
    return abs(a) + abs(b);
}
```

程序1是用命令来表示程序, 用命令的顺序执行来表示程序的组合, 不算函数式

程序2是用函数来表示程序, 但在内部是用命令的顺序执行来实现, 不太函数式

程序3是用函数来表示程序, 用函数的组合来表达程序的组合, 是完全的函数式编程

函数式编程只是一种思维方式

注意我们的判断标准关注的是程序在人(也就是程序员)脑中的状态, 它关心的是某个人在思考问题的时候, 程序是以什么样的概念存在于脑中的 -- 是指令的序列呢, 还是计算呢 -- 而跟程序在机器中执行的状态无关, 毕竟函数式的代码和非函数式的代码最终编译得到的机器码完全有可能是一样的. 所以这里重要的区别在于人如何理解程序, 而非程序如何执行.

所以对于某段代码算不算函数式编程这个问题, 我觉得这跟它用的什么编程语言并没有什么直接的关系 (比如我在程序3中用的同样是非函数式语言), 它跟问题本身有关, 跟具体的实现思路有关, 甚至具体到一个程序员, 它还跟这个程序员如何去解读这段程序有关 (比如当我把程序2中的`add_abs`看作黑盒的时候, 我忽略它的实现, 而在其它地方把它当做纯函数去使用, 它也完全可能成为函数式代码的一部分).

几种编程语言

Pascal、C

命令式语言，基于动作的语言，以冯诺依曼计算机体系结构为背景，通过修改存储器的值产生副作用的方式去影响后续的计算

Fortran、ALGOL、COBOL

关注的焦点：如何进行计算

Java、C++

面向对象语言，以对象作为基本程序结构单位，提供类、继承等成分

Prolog

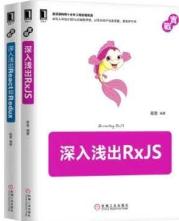
逻辑式语言，建立在逻辑学的理论基础之上，广泛应用在人工智能的研究中，可以用来建造专家系统、自然语言理解、智能知识库等

LISP、SML

函数式语言，数学函数是基础，没有内部状态，也没有副作用
程序的执行是表达式的计算

关注的焦点：计算什么

函数式编程可以用什么语言？



深入浅出RxJS+深入浅出React...

区域包邮C4
程墨

¥118.80 领券99减2 赠

放心购 8条评价 100%好评



[北京发货] Haskell函数式编程...

¥46.80

2条评价 100%好评

广告



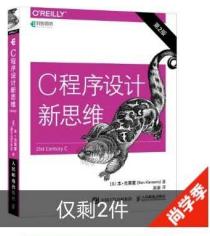
[按需印刷] Haskell函数式编程...

按需印刷 POD版 发货需要 4-7天

(美)Simon, Thompson

¥116.10 商家免邮

放心购 2条评价 100%好评

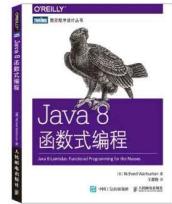


包邮 C程序设计新思维 第2版

[美]BenKlemens克莱蒙

¥60.20 商家免邮

1条评价 100%好评

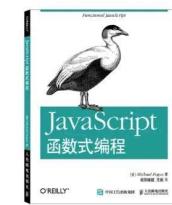


Java 8函数式编程

春日好读书 (此商品未参加优惠券...
[英]Richard Warburton

¥30.80

自营 放心购 3020条评价 98%好评
有电子版



JavaScript函数式编程 畅销书...

¥39.20

1条评价 100%好评

广告



Scala函数式编程

春日好读书 (此商品未参加优惠券...
[美]Paul Chiusano (保罗·基乌萨诺...

¥57.80

自营 放心购 13090条评价 98%好评



C#函数式编程 编写更优质的C#...

春日好读书 (此商品未参加优惠券...
[美]恩里科·博南诺 (Enrico,Buonan...

¥78.40

自营 放心购 84条评价 100%好评

函数式编程可以用什么语言？

C++ Java中都是语法等级的东西，换言之不加入一样也有高阶函数。同时，java也在努力的改革，进步，在像函数式“进化”，比如说java8提供的Stream流，lambda，努力将函数（方法）提升为一等公民

C#, Python的List Comprehension也是2010前的，Java Generic则是1998由Philip Walder（Haskell界大佬），Martin Odersky（Scala界大佬）提出来

Python, C#, Perl, JS, VB, 都是2010前加入Lambda的（Dart, Swift的确有2010后Lambda，不过他们俩就是2010后出的语言）

- FP一直都有影响各种各样的语言，不是近期才开始的。
- 一个编程范式从有一群人一起写一个语言，到这个编程范式被工业应用，要20年左右。

函数式编程是一种非常简单的风格

无类型的lambda 演算其实规则很简单 只有三条

1. 变量 (variable)
2. 函数 (lambda-abstraction)
3. 替代 (beta-reduction)

任何支持函数的语言都可以进行函数式风格的编程 注意到与命令式风格不同的是没有赋值，这意味着reason 程序的时候每个变量的值是不变的 不用考虑程序变量随着时间的变化 -- 大大降低了程序的复杂性。

判定丢番图方程的可解性问题

- “希尔伯特23个问题” (Hilbert 23 Problems) 的第10个问题 (1900年)
 - “给定一个系数均为有理整数，包含任意个未知数的丢番图方程：设计一个过程，通过有限次的计算，能够判定该方程在有理数整数上是否可解”
- 形式化描述：数理的完整性、一致性和可判定性 (1928年)
 - 通过有限次的步骤，对数学函数进行有效计算的方法——递归函数、可计算性理论



David Hilbert
(哥廷根大学)

计算机科学的理论基础

可判定性问题

- 可判定性问题：存不存在 确定的有效过程或程序（effective process or procedure），给定任何一个形式化了的关于算术的命题，这个程序将可以判定其真伪、或者判断它是否可证明。

首先，它涉及“有效过程”这个直观的概念，并没有也不可能有明确的形式化定义的，而是取决于人们如何具体安排和构造这样的程序，而这些安排和构造的可能性可以说是没有明确的边界的。

其次，它只要求针对一定的属性进行判定，而并不要求给出证明。就是说，这只是在要求按照程序就能给出“是”或者“非”的答案，**只要求答案正确，而不要求给出任何理由或者推理过程。**

- 比如哥德巴赫猜想：任何大于2的偶数都可以分解成两个素数之和。咱们对判定程序的要求就是，如果把这个命题作为输入，程序要输出“是”或者“非”，判定这个猜想的真伪。这样，即使这个猜想被正确地判定为“真”，咱们也依然不知道如何去证明这个猜想，但是（如果这个程序是可靠的话）至少可以知道它是“真”的。



David Hilbert
(哥廷根大学)

图灵机和 λ 演算

1936年，针对数理的可判定性问题的答案

- 以 λ 演算为基础：“An unsolvable problem of elementary number theory”
一切皆函数

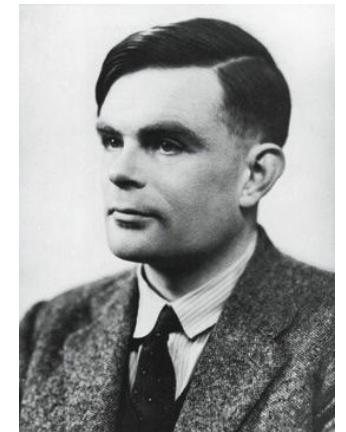


Alonzo Church(美国)

- 图灵机模型：“On computable numbers, with an application to the entscheidungs problem”

——计算机的基本理论模型

冯·诺依曼架构、历史上第一台电子计算机



Alan Turing(英国)

图灵机和 λ 演算

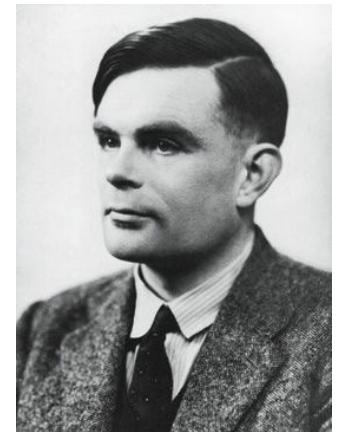
图灵的结果比丘奇的结果晚出来大概一年，但是由于跟丘奇的 λ 演算，以及同样是等价的哥德尔的递归函数相比，图灵机的构造要直观很多，可以说更直接地反映出了有效过程或者程序这个直观概念的关键部分。

更具体一点，图灵在这篇文章取得了一系列重要结果：

- 1) 提出了“图灵机”这种有效程序的模型，用于模拟一切算术和逻辑操作；
- 2) 开发了一套把任何图灵机加以编码的方式，并构造出以这些编码为输入的“通用图灵机”，能够执行任何其他图灵机的编码；
- 3) 说明了图灵机这种构造能够进行很多计算，包括圆周率 π 和自然对数的底 e 等等；
- 4) 说明了存在着可以准确定义，然而不能由任何图灵机完成的问题，比如判定任何给定的图灵机是不是只有有限多的输出，说明了存在着图灵机不能输出其二进制表示的数，从而指出了有效程序能完成的工作的限度；
- 5) 把希尔伯特的判定问题化约为数字的生成问题，证明了判定问题不可解：不存在图灵机意义上的般性的判定程序，可以就任何给定的关于自然数的一阶逻辑命题进行可证明性的判定。



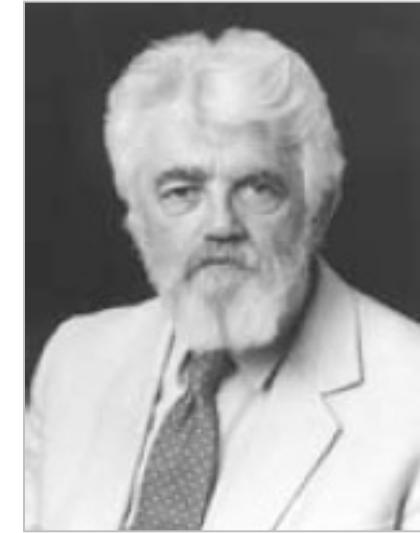
Alonzo Church(美国)



Alan Turing(英国)

那函数式编程？

- 不依赖于冯·诺依曼体系结构的计算机
- 设计的基础：**数学函数**
 - 程序的输出定义为其输入的一个数学函数
 - 没有内部状态，也没有副作用
- 第一个函数式语言：LISP
 - 初始动机：用于人工智能研究的表处理语言
 - 实现 λ 演算理论，采用符号表达式定义数据和函数，采用抽象数据列表与递归符号演算衍生人工智能



John McCarthy
(麻省理工)

函数式编程

与 **Lambda Calculus** 相对应的 **Lisp Machine** 也火过，不过那已经是上个世纪的事情了

Lisp Machine 的内部实现极其复杂精巧，至少包括硬件 GC 系统，能够直接运行 Lisp 元指令的特制处理器以及 Lisp 实现的 Lisp 本身，CLIM 图形环境以及完整的操作系统。可惜至少到目前为止，除了极少数当年的开发者以外，无人能一窥其全貌。

后来，由于 **AI** 方面研究的萎缩以及微机普及浪潮的推动，**Lisp Machine** 这种精巧复杂高贵冷艳的玩意就玩死了

最后实际普遍使用的computation model是Von-neumann模型，都带着可随机访问的内存。很少有真拿一条纸带左右移动算东西的。Von-neumann能流行，主要是因为这个能用电子元件做出来，而且速度和其他能做出来的model比要快不少。

Turing machine成为了最著名的model（而不是lambda calculus），恐怕是因为它不仅是个model，而且是个machine（可以搭出来），因此不仅有理论意义，而且有现实意义。

几个函数式编程语言

- ISWIM (If you See What I Mean) (1966年)
 - 以函数式为核心的指令式语言，函数由Sugared λ 演算组成
 - 没有完全实现，在一定程度上奠定了函数式语言设计的基础
- ML 语言 (1973年)
 - 具备命令式语言特点的函数式编程语言灵活的函数功能，允许副作用和指令式编程的使用
 - 有并行扩展，可以用来写并行系统
- Standard ML 语言 (20世纪90年代)
 - 高阶函数、I/O机制、参数化的模块系统和完善的类型系统
 - 交互式编译器，SML/NJ最著名



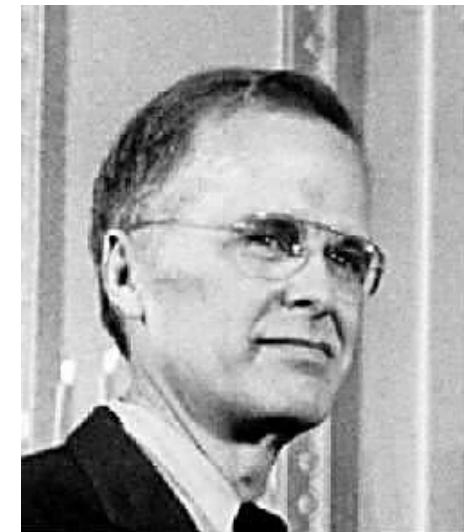
Peter Landin
(牛津大学)



Robin Milner
(爱丁堡大学)

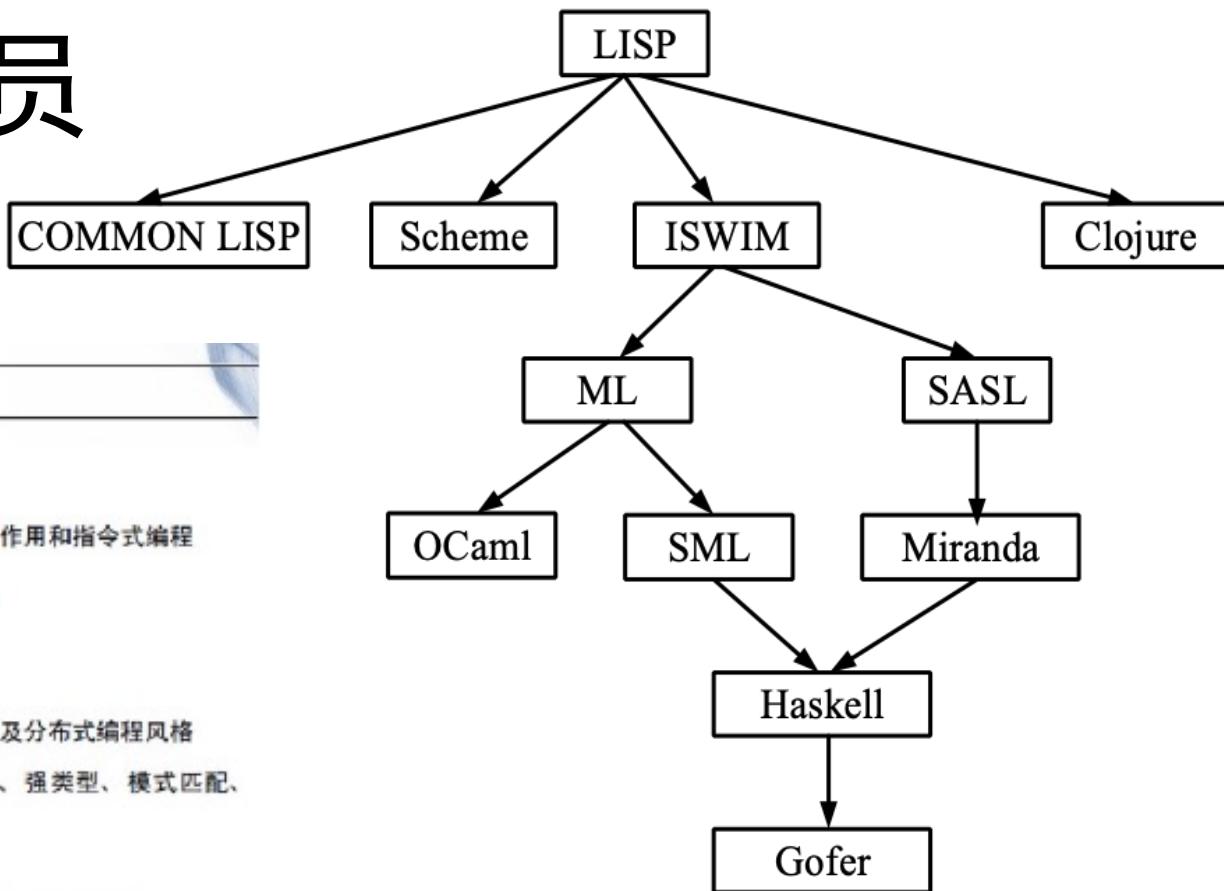
FP语言：开创纯函数语言研究之先河

- 1977年，ACM年会上获得计算机界最高奖：图灵奖 “Can Programming be Liberated from the von Neumann Style?: A Functional Style and Its Algebra of Programs”
- 取消了变量，用固定的泛函数结构和一些简单定义作为从现存函数构造新函数的唯一工具
- 不是最早的函数式编程语言，但发明了Functional programming这个概念



John Backus(美国)

函数式语言家族及主要成员



时间	名称	提出者/或提出机构	特征
1958	Lisp (包括 Common Lisp 和 Scheme)	McCarthy J	表处理语言, 适合于数学运算
1979	ML (包括 Standard ML 和 Caml)	Milner R	非纯函数式编程语言, 强类型, 允许副作用和指令式编程
1985	Miranda	Turner D	基于 ML 的惰性纯函数式语言, 强类型
1987	Clean	Radboud University Nijmegen	惰性高阶纯函数式语言, 强类型
1987	Erlang	Erlang A.K.	多范式编程语言, 支持函数式、并发式及分布式编程风格
1988	Haskell	Hudak P, Wadler P	纯函数式语言, 支持惰性求值、高阶、强类型、模式匹配、列表内包、类型类和类型多态
1996	OCaml	Leroy X, Vouillon J	非纯函数式编程语言, 强静态类型
2001	Scala	Odersky M	多范式编程语言, 支持对象式和函数式编程的典型特性
2002	F#	微软	基于 OCaml 的非纯函数式编程语言, 强静态类型, 适合程序核心数据多线程处理
2007	Clojure	Hickey R	针对 JVM 平台且基于 Lisp 的动态函数式编程语言, 支持高阶函数和惰性计算

函数式编程语言的兴衰

- 2000年以前的没落
 - 自身缺点：慢，消耗更多的资源
 - 外界因素：面向对象编程的崛起
 - 一直被学术界重视，很少应用到业界
- 2000年以后的兴盛
 - 摩尔定律失效，芯片工艺限制
 - 多核计算机的诞生，并发与数据共享需求
 - 命令式编程的天生缺陷 vs. 函数式编程的优势
 - 走出实验室，为业界所用
 - 工业和商业应用，如符号数学、统计、金融分析等

函数式编程语言的兴衰

由于命令式编程语言也可以通过类似函数指针的方式来实现高阶函数，函数式的最主要的好处主要是不可变性带来的。没有可变的状态，函数就是引用透明（Referential transparency）的和没有副作用（No Side Effect）。

- 1) 函数即不依赖外部的状态也不修改外部的状态，函数调用的结果不依赖调用的时间和位置，这样写的代码容易进行推理，不容易出错。这使得单元测试和调试都更容易。
- 2) 由于（多个线程之间）不共享状态，不会造成资源争用(Race condition)，也就不需要用锁来保护可变状态，也就不会出现死锁，这样可以更好地并发起来，尤其是在对称多处理器（SMP）架构下能够更好地利用多个处理器（核）提供的并行处理能力。

- cpu的性能提升将体现在核数增加，并行程序速度会越来越快。
- 出于对效率，安全，稳定以及运行速度的追求，在多核cpu上并行的编程是大势所在。
- 并行的程序的写法的核心就是找出不能并行的地方，其他地方都尽量并行。

图灵机 vs. λ 演算 命令式语言 vs. 函数式语言

- 图灵机和命令式语言
 - 遵循图灵机模型，核心概念是命令
 - 通过修改存储器的值而产生副作用的方式去影响后续的计算
 - 命令式程序的“函数”有副作用，如改变全局变量
- λ 演算和函数式语言
 - 一切皆函数，即用函数组合的方式描述计算过程
 - 避免繁琐的内存管理
 - 函数的结果仅仅依赖于提供给它的参数

函数或表达式计算时除了有返回值之外，还修改了某个状态或与调用它的函数或外部环境进行了明显的交互

什么是函数式编程

- In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

--From wikepedia

- 函数式编程是一种编程模型，它将计算机运算看作是数学中函数的计算，并且避免了**状态**以及**变量**的概念。

函数式编程的style

1. closure

按照上面的三条规则函数式是first class 的 是可以直接传递作为参数的,

2. 高阶类型推断

因为函数可以作为参数, 其类型可以非常复杂 比如下面的函数类型其实非常普遍:

```
val callCC : (('a -> 'b -> 'c) -> ('a -> 'c) -> 'd) -> ('a -> 'c) -> 'd
```

如果没有类型推断, 其实很难写对或者理解它的语义

3. tail-call

因为函数式风格没有赋值, 也就没有for循环, 要实现循环操作 只能通过递归调用, 比如下面简单的例子:

```
let rec even n = if n = 0 then true else if n = 1 then false else odd (n - 1)
and odd n = if n = 1 then true else if n = 0 then false else even (n - 1)
```

函数式编程的好处

$$h(g(f(x))) = (h * (g * f))(x) = ((h * g) * f)(x)$$

函数式的编程，在我看来，体验上主要就是上面这个表达式。在很多传统的命令式语言里面，你只能按左边这个形式编程。原因有很多，比如很多操作不是表达式，函数不是一等公民，不支持闭包等。

而一个语言如果对函数式支持良好，你就可以很方便的应用右边的形式。

对于函数式风格，你总是在构造一个巨大的表达式（通过组合已有的表达式），然后把数据灌进去，结果就出来了。

而对于命令式风格，你总是在操作数据，等你把数据操作完了，结果就出来了。

进一步的，你还可以操作持有数据的容器。这些容器提供map, filter, reduce, fold之类的高阶方法。它使得你可以在容器上组合函数逻辑。函数天然然是可以组合的，天然满足结合律。

函数式编程的特点

- 1) 数据是从哪里来不重要；
- 2) 每一个函数都是为了用小函数组织成更大的函数，函数的参数也是函数，函数返回的也是函数；
- 3) 最后得到一个超级牛逼的函数，就等着别人用**main**函数把数据灌进去了。

函数式编程语言的特点

- 函数是“first-class values”
 - 程序、函数和过程可以使用数学意义下的函数来表示

$y = f(x)$ //变量y就是值 $f(x)$ 的一个名字

- 程序、函数和过程之间没有区别，函数可以作为数据来使用
- 变量总是代表实际值，变量没有存储位置和左值的概念没有赋值操作，只有常量、参数和值

$x = x + 1$

函数式编程语言的特点 $x = x + 1$

没有意义

- x 等于 x 加一，此时我们大量依赖的是状态，可变的状态，或者说变量，它们的值可以随程序运行而改变。
- 在函数式编程的状态都是不可变的。你可以声明一个状态，但是不能改变这个状态。而且由于你无法改变它，所以在函数式编程中不需要变量。事实上对函数式编程的讨论更像是数学、公式，而不像是程序语句。如果你把 $x = x + 1$ 这句话交给一个程序员看，他会说“啊，你在增加 x 的值”，而如果你把它交给一个数学家看，他会说“嗯，我知道这不是true”。
- 在函数式编程语言中，函数接受一些参数，那么当你调用这个函数时，影响函数调用的只是你传进去的参数，而你得到的也只是计算结果。在一个纯函数式编程语言中，函数在计算时不会对进行一些神奇的改变，它只会使用你给它的参数，然后返回结果。在函数式编程语言中，一个void方法是没有意义的，它唯一的作用只是让你的CPU发热，而不能给你任何东西，也不会有副作用。

函数式编程语言的特点

- 引用透明性
 - 任何函数的值只取决于它的参数的值，而与求得参数值的先后或调用函数的路径无关
 - 表达式求值，避免繁琐的内存管理，没有内部状态，也没有副作用
 - 变量只是一个名称，而不是一个存储单元
 - 变量只能被定义一次，将反复拷贝大量数据
- ```
int x = 10;
int u = 0;

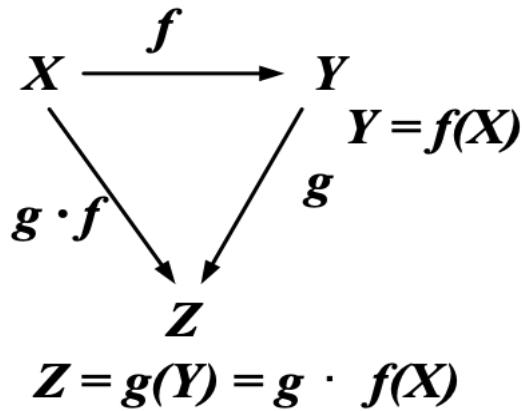
int f() {
 u++;
 return x+u;
}

int g() {
 return f() + u;
}
```

# 函数式编程语言的特点

- 高阶函数 (Higher-Order Function)

- 函数可以不依赖于任何其他的对象而独立存在，可以将函数作为参数传入另一个函数，也可以作为函数的返回值
- 两个基本运算：合成和柯里化



把接受多个参数的函数变换为接受一个单一参数（最初函数的第一个参数）的函数，并返回接受余下参数而且返回结果的新函数

```
real add(real x, real y, real z) {
 return x+y+z;
}
```

```
fun plus x y z: int = x + y + z
```

# 函数式编程语言的特点

- 惰性求值（延迟计算）与并行

- 当调用函数时，不是盲目的计算所有实参的值后再进入函数体，而是先进入函数体，只有当需要实参值时才计算所需的实参值（按需调用）

```
int a = f(x);
int b = g(y);
if x == 0 c = a else c = b;
```

```
int a = f(x);
int b = g(y);
int c = a + b;
```

- 用数学方法分析处理代码
    - 抵消相同项、避免执行无谓的代码
    - 重新调整代码执行顺序、效率更高
    - 重整代码以减少错误
  - 不能处理I/O, 不能和外界交互

# 函数式编程语言的特点

- 递归调用及其优化

【例】返回整数i和j之间的所有整数的和。

$$sum(i, j) = i + (i + 1) + \dots + (j - 1) + j$$

```
int sum(int i, int j) {
 int k, temp=0;
 for(k=i; k<=j; k++)
 temp+=k;
 return temp;
}
```

(a) 命令式循环版本

有什么问题 ?

```
int sum(int i, int j) {
 if (i>j) return 0;
 else return i+sum(i+1, j);
}
```

(b) 函数式递归版本

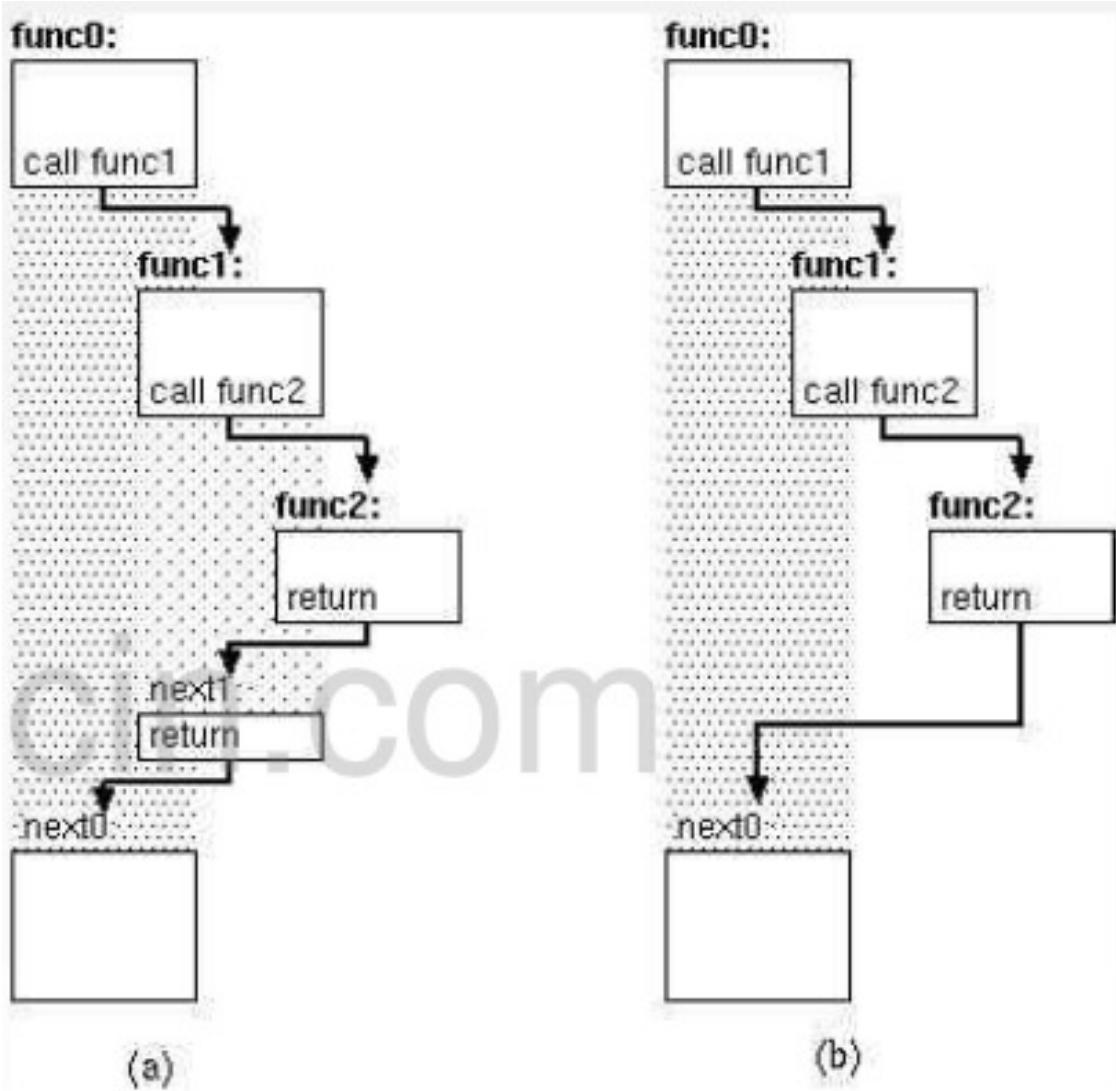
需要栈，空间浪费  
！可能堆栈溢出，  
很容易看出，普通的  
线性递归比尾递归  
更加消耗资源。  
在实现上说，每次重  
复的过程调用都使  
得调用链条不断加  
长。系统不得不使用  
栈进行数据保存和  
恢复。

# 函数式编程语言的特点

- 递归调用及其优化

```
int sum(int i, int j) {
 if (i>j) return 0;
 else return i+sum(i+1, j);
}
```

```
sum(3,9)
= 3+sum(4,9)
= 3+4+sum(5,9)
= 3+4+5+sum(6,9)
= 3+4+5+6+sum(7,9)
= 3+4+5+6+7+sum(8,9)
= 3+4+5+6+8+9+0
= ...
```



## • 尾递归

- 一个函数(调用者, caller)调用另一个函数(被调用者, callee)，而且callee产生的返回值被caller立刻返回出去，这种形式的调用称为尾调用(tail call)。
- 如果caller和callee是同一个函数，那么便将这个尾调用称为尾递归(tail recursion)。

- 问题：如何把函数转换成尾递归？

- 累积参数法

- 将递归调用之后要完成的操作预先计算好，并将结果传给递归调用。

```
int sum(int i, int j) {
 if (i>j) return 0;
 else return i+sum(i+1, j);
}
```



将sum函数转换  
成尾递归

尾调用并不需要为**callee**开辟一个新的栈帧（**stack frame**），但需要参数和局部数据（覆盖方式），使栈空间大大缩小，提高实际运行效率

```
int sum1(int i, int j, int sumSoFar) {
 if (i>j) return sumSoFar;
 else return sum1(i+1, j, sumSoFar+i);
}

int sum(int i, int j) {
 return sum1(i, j, 0);
}
```

- 问题：如何把函数转换成尾递归？
  - 累积参数法
    - 将递归调用之后要完成的操作预先计算好，并将结果传给递归调用。

```
int sum1(int i, int j, int sumSoFar) {
 if (i>j) return sumSoFar;
 else return sum1(i+1, j, sumSoFar+i);
}

int sum(int i, int j) {
 return sum1(i, j, 0);
}
```

```
sum(3,9)
= sum1(3,9,0)
= sum1(4,9,3)
= sum1(5,9,7)
= sum1(6,9,12)
= sum1(7,9,18)
= sum1(8,9,25)
= sum1(9,9,33)
= 42
```

而尾递归就不存在这样的问题，因为状态完全由i、j和sumSoFar保存。

# 函数式编程语言的特点

- 递归调用及其优化

ML语言的整数求和函数：

```
fun sum [] = 0
| sum (x::L) = x + sum(L);
```

```
fun sum' ([] , a) = a
| sum' (x::L, a) = sum' (L, x+a);
```

# 函数式编程语言的特点

- 递归调用及其优化

斐波那契数列  $F_n$ :  $F_0 = 0$ ,  $F_1 = 1$

$$F_n = F_{n-2} + F_{n-1} \quad (n \geq 2)$$

```
int Fib(int n) {
 int res=b=1, a=0;
 if (n==0) return 0;
 n = n - 1;
 while (n > 0) {
 res = a + b;
 a = b;
 b = res;
 n = n - 1;
 }
 return res;
}.
```

```
fun fib 0 = 0
| fib 1 = 1
| fib n = fib(n-2) + fib(n-1)
```

```
fun nextfib (prev, curr: int) = (curr, prev + curr)
fun fibpair 1 = (0, 1)
| fibpair n = nextfib(fibpair(n-1))
```

效率太低！用循环的思路编写的斐波那契数列计算代码描述了数列的求解思路，即应该先怎么做然后再怎么处理。

它会不断的重复计算同样的子问题，如在计算  $F_8$ 时： $F_8 = F_6 + F_7 = F_6 + (F_5 + F_6)$ 。这样， $F_6$ 计算了两遍。不断计算相同的子问题



# 函数式编程语言的特点

- 递归调用及其优化

斐波那契数列  $F_n$ :  $F_0 = 0$ ,  $F_1 = 1$

$$F_n = F_{n-2} + F_{n-1} \quad (n \geq 2)$$

```
int Fib(int n) {
 int temp, a = b = 1;
 n = n - 1;
 while (n > 0) {
 temp = a;
 a = a + b;
 b = temp;
 n = n - 1;
 }
 return b;
}
```

递归模式浪费了较多空间，  
构造了嵌套调用  $\text{nextfib}(\text{nextfib}(\dots\text{nextfib}(0,1)\dots))$ 。

在这个递归过程中，需要维持一个栈，用来保存迭代  
调用过程中产生的大量的中间结果，当数据量增大  
时，可能会出现堆栈溢出的情况。

```
fun nextfib (prev, curr: int) = (curr, prev + curr)
fun fibpair 1 = (0, 1)
| fibpair n = nextfib(fibpair(n-1))
```

```
fun itfib (1, prev, curr): int = curr
| itfib(n, prev, curr) = itfib(n-1, curr, prev + curr)
fun fib n = itfib(n, 0, 1)
```

累积参数

# 函数式编程语言的特点

- 递归调用及其优化

斐波那契数列  $F_n$ :  $F_0 = 0$ ,  $F_1 = 1$

$$F_n = F_{n-2} + F_{n-1} \quad (n \geq 2)$$

```
fun itfib (1, prev, curr): int = curr
 | itfib(n, prev, curr) = itfib(n-1, curr, prev + curr)
fun fib n = itfib(n, 0, 1)
```

计算过程反转变为：

itfib(7, 0, 1) →  
itfib(6, 1, 1) → ...  
itfib(1, 8, 13) →  
13

尾递归 ( Tail Recursion ) 方式