编程范式简介

陈艳宾 2019.1.18

目录

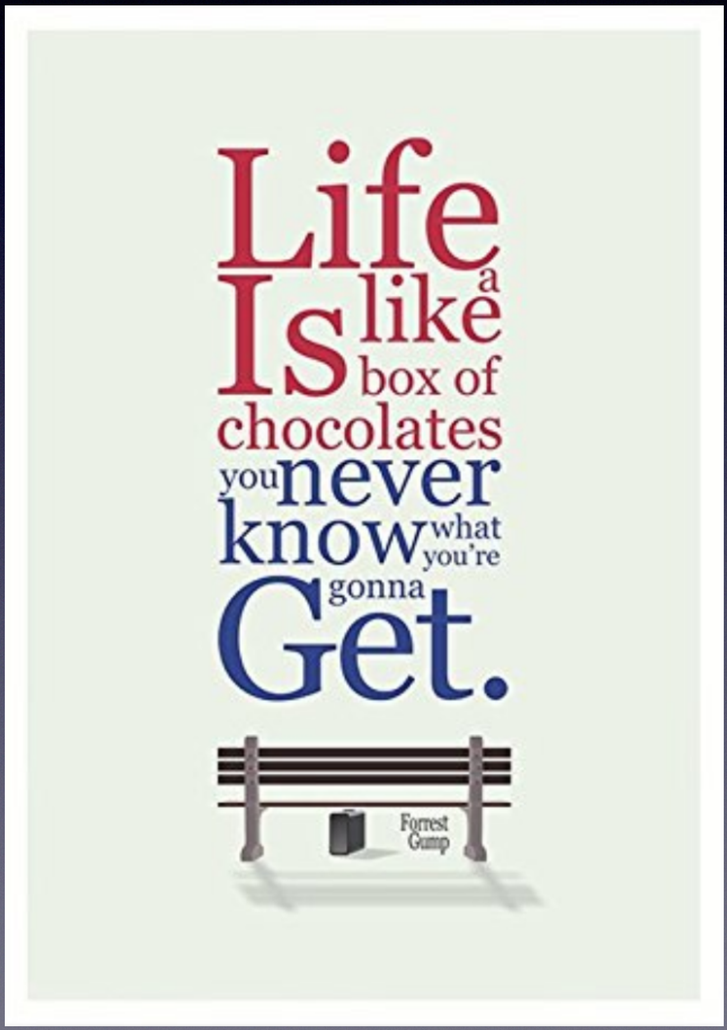
- 什么是编程范式?
- 编程语言的发展历史
- 主要范式介绍
- 延伸阅读

什么是编程范式?

- 范式是一种做某事的方式（比如编程），而不是具体的事情（比如语言）。
- 编程范式是一类典型的编程风格，是指从事软件工程的一类典型的风格。
- 是一种对编程语言进行分类的方式：如果编程语言 L 碰巧使特定的编程范式 P 易于表达，那么我们常说“ L 是 P 语言”。
- 范式并不相互排斥; 一个程序可以有多个范式，一种语言可以同时支持多种方式。

为什么要了解?

- 通过编程语言的范式，不但可以知道整个编程语言的发展史，而且还能提高自己的编程技能写出更好的代码。
- 编程本身是为了解决问题。
- 基于编程范式可以更好的理解问题以便找到解决方案。
- 不同的范式各有所长。



编程语言的发展历史- 机器语言

```
1  10110000000000000000000011
2  00000101000000000000110000
3  00101101000000000000000101
```

8086 机器上 s=768+12288-1280

太难写,太难读,太难改

汇编语言

- 为了解决机器语言编写、阅读、修改复杂的问题，汇编语言应运而生。汇编语言又叫“符号语言”，用助记符代替机器指令的操作码，用地址符号（Symbol）或标号（Label）代替指令或操作数的地址。
- 机器语言：1000100111011000 => 汇编语言：mov ax,bx
- 面向机器底层， CPU 指令，寄存器，内存地址。
- 不同 CPU 的汇编指令和结构是不同的。例如，Intel 的 CPU 和 Motorola 的 CPU 指令不同，同样一个程序，为 Intel 的 CPU 写一次，还要为 Motorola 的 CPU 再写一次，而且指令完全不同。

```
1  .section .data
2    a: .int 10
3    b: .int 20
4    format: .asciz "%d\n"
5  .section .text
6  .global _start
7  _start:
8    movl a, %edx
9    addl b, %edx
10   pushl %edx
11   pushl $format
12   call printf
13   movl $0, (%esp)
14   call exit
```

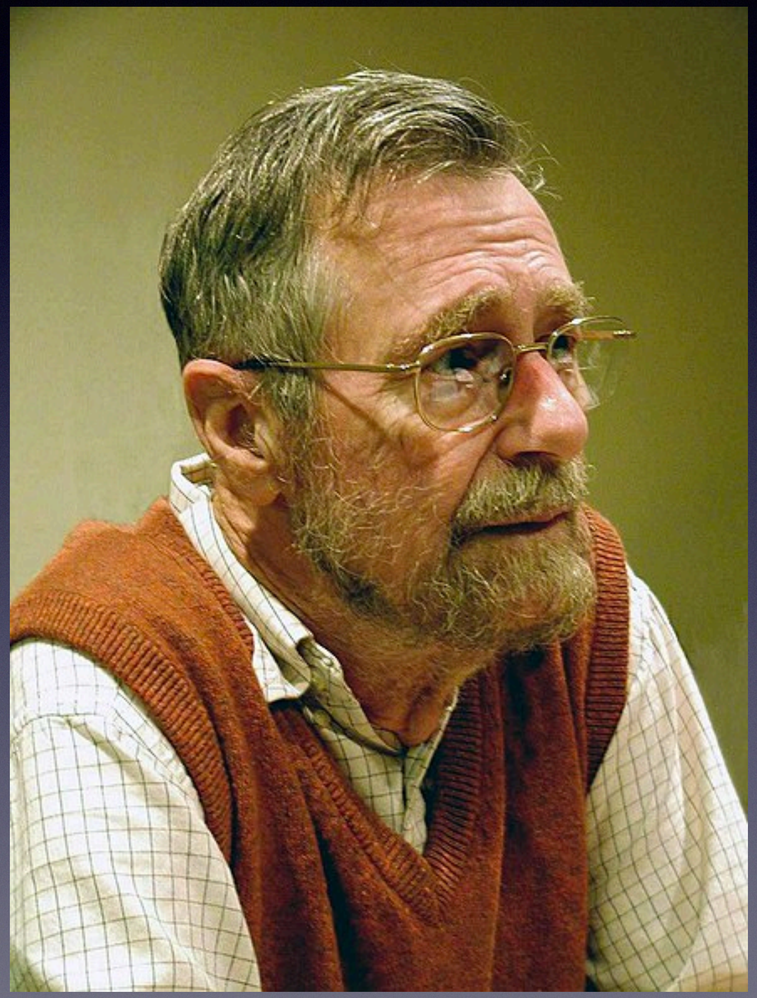
高级语言

- Fortran：1955 年，名称取自”FORmula TRANslator”，即公式翻译器，由约翰·巴科斯（John Backus）等人发明。
- LISP：1958 年，名称取自”LISt Processor”，即枚举处理器，由约翰·麦卡锡（John McCarthy）等人发明。
- Cobol：1959 年，名称取自”Common Business Oriented Language”，即通用商业导向语言，由葛丽丝·霍普（Grace Hopper）发明。
- 这些语言让程序员不需要关注机器底层的低级结构和逻辑，而只要关注具体的问题和业务即可。
- 通过编译程序的处理，高级语言可以被编译为适合不同 CPU 指令的机器语言。程序员只要写一次程序，就可以在多个不同的机器上编译运行，无须根据不同的机器指令重写整个程序。

```
    result = []
    i = 0
start:
    numPeople = length(people)
    if i >= numPeople goto finished
    p = people[i]
    nameLength = length(p.name)
    if nameLength <= 5 goto nextOne
    upperName = toUpper(p.name)
    addToList(result, upperName)
nextOne:
    i = i + 1
    goto start
finished:
    return sort(result)
```

结构化程序设计方法

- 高级语言的出现，解放了程序员，但随着软件的规模和复杂度的大大增加，20 世纪 60 年代中期开始爆发了第一次软件危机，典型表现有软件质量低下、项目无法如期完成、项目严重超支等，因为软件而导致的重大事故时有发生。
- IBM 的 System/360 的操作系统 / 5000 人一年 / 100 万行源码 / 5 亿美元 / 佛瑞德·布鲁克斯 / 人月神话
- 艾兹赫尔·戴克斯特拉（Edsger Dijkstra）于 1968 年发表了著名的《GOTO 有害论》
- 结构化程序设计的主要特点是抛弃 goto 语句，采取“自顶向下、逐步细化、模块化”的指导思想。结构化程序设计本质上还是一种面向过程的设计思想，但通过“自顶向下、逐步细化、模块化”的方法，将软件的复杂度控制在一定范围内，从而从整体上降低了软件开发的复杂度。结构化程序方法成为了 20 世纪 70 年代软件开发的潮流。
- structures: if/then/else and while loops.

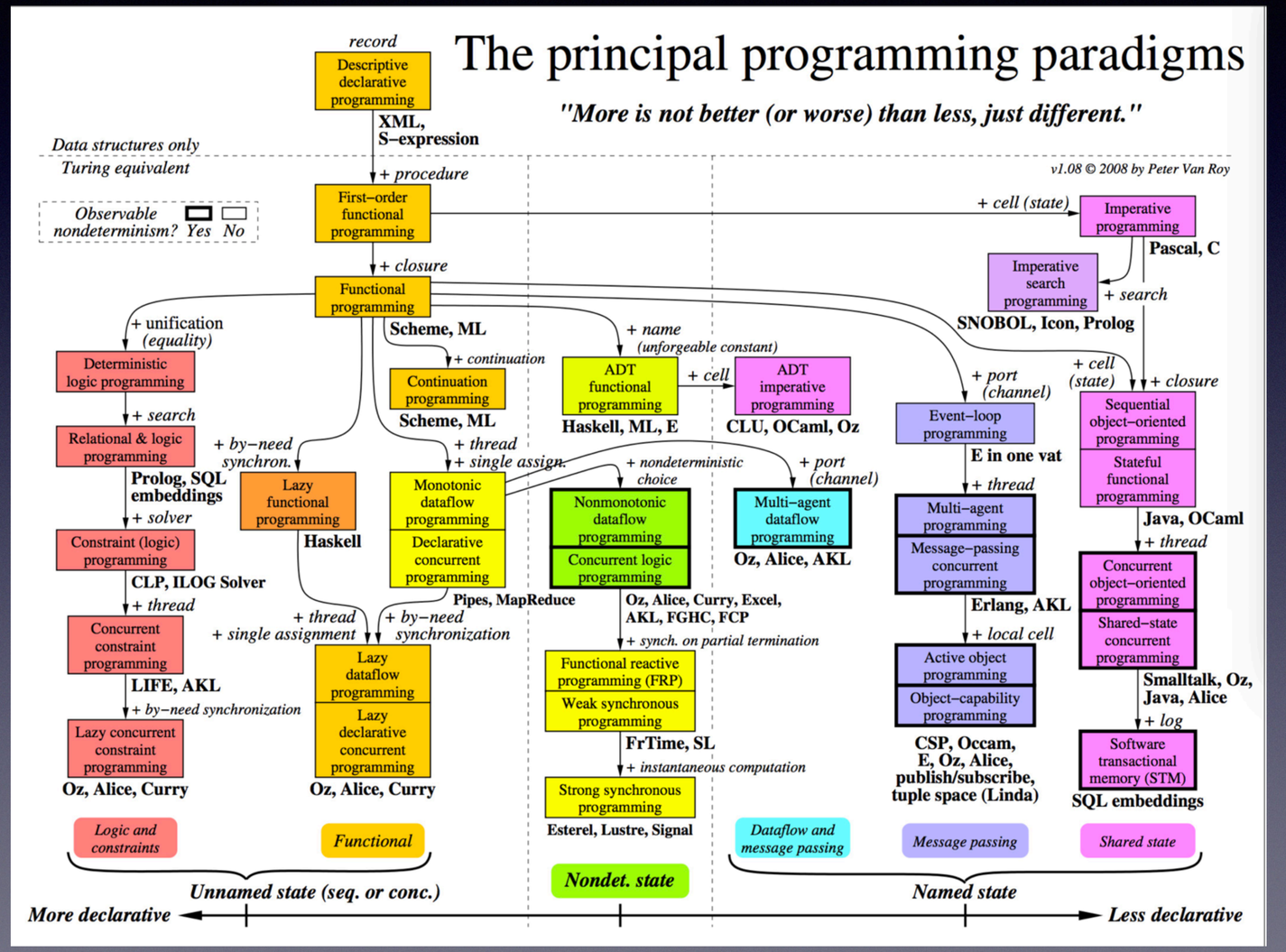


面向对象的设计方法

- 结构化编程的一定程度上缓解了软件危机。随着硬件的快速发展，业务需求越来越复杂，以及编程应用领域越来越广泛，第二次软件危机很快就到来了。
- 第二次软件危机的根本原因还是在于软件生产力远远跟不上硬件和业务的发展。第一次软件危机的根源在于软件的“逻辑”变得非常复杂，而第二次软件危机主要体现在软件的“扩展”变得非常复杂。结构化程序设计虽然能够解决（也许用“缓解”更合适）软件逻辑的复杂性，但是对于业务变化带来的软件扩展却无能为力，软件领域迫切希望找到新的银弹来解决软件危机。
- 得益于 C++ 的功劳，后来的 Java、C# 把面向对象推向了新的高峰。到现在为止，面向对象已经成为了主流的开发思想。

主要范式介绍

- 泛型
- 面向对象
- 函数式编程



泛型 - 类型

- 类型是对内存的一种抽象。不同的类型，会有不同的内存布局和内存分配的策略。
- 不同的类型，有不同的操作。所以，对于特定的类型，也有特定的一组操作。

类型系统

- 在计算机科学中，类型系统用于定义如何将编程语言中的数值和表达式归类为许多不同的类型，以及如何操作这些类型，还有这些类型如何互相作用。类型可以确认一个值或者一组值具有特定的意义和目的

- 编程语言会有两种类型，一种是内建类型，如 int、float 和 char 等，一种是抽象类型，如 struct、class 和 function 等。抽象类型在程序运行中，可能不表示为值。

类型功能

- 程序语言的安全性：使用类型可以让编译器侦测一些代码的错误。
- 利于编译器的优化：静态类型语言的类型声明，可以让编译器明确地知道程序员的意图。因些，编译器就可以利用这一信息做很多代码优化工作。
- 代码的可读性：有类型的编程语言，可以让代码更易读和更易维护。代码的语义也更清楚，代码模块的接口（如函数）也更丰富和更清楚。
- 抽象化：类型允许程序设计者对程序以较高层次的方式思考，而不是烦人的低层次实现。

类型分类

- 静态类型 vs 动态类型
- 强类型 vs 弱类型
- 编译 vs 解释

类型的问题 - 如何泛型

- “类型”有时候是一个有用的事，有时候又是一件很讨厌的事情。因为类型是对底层内存布局的一个抽象，会让我们的代码要关注于这些非业务逻辑上的东西。而且，我们的代码需要在不同类型的数据间做处理。但是如果程序语言类型检查得过于严格，那么，我们写出来的代码就不能那么随意。

- 类型带来的问题就是作用于不同类型的代码，虽然长得非常相似，但是由于类型的问题需要根据不同版本写出不同的算法。

C 如何实现泛型?

```
void swap(int* x, int* y)
{
        int temp = *x;
        *x = *y;
        *y = tmp;
}
```

```
void swap(void* x, void* y, size_t size)
{
     char tmp[size];
     memcpy(tmp, y, size);
     memcpy(y, x, size);
```

```
#define swap(x, y, size) {\
        char temp[size]; \
        memcpy(temp, &y, size); \
        memcpy(&y,   &x, size); \
        memcpy(&x, temp, size); \
}
```

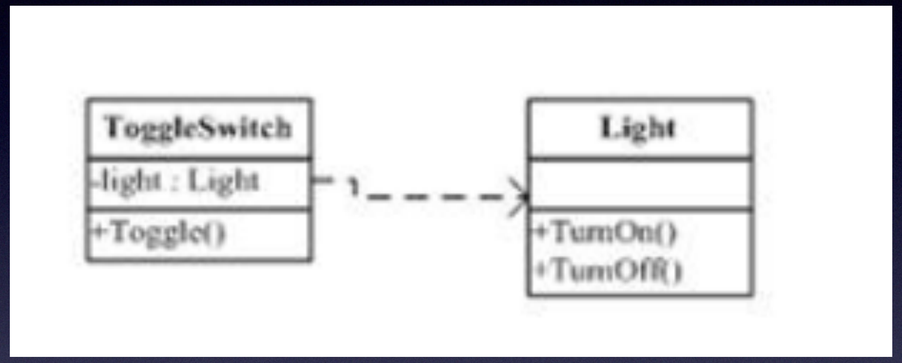
C++ 如何解决?

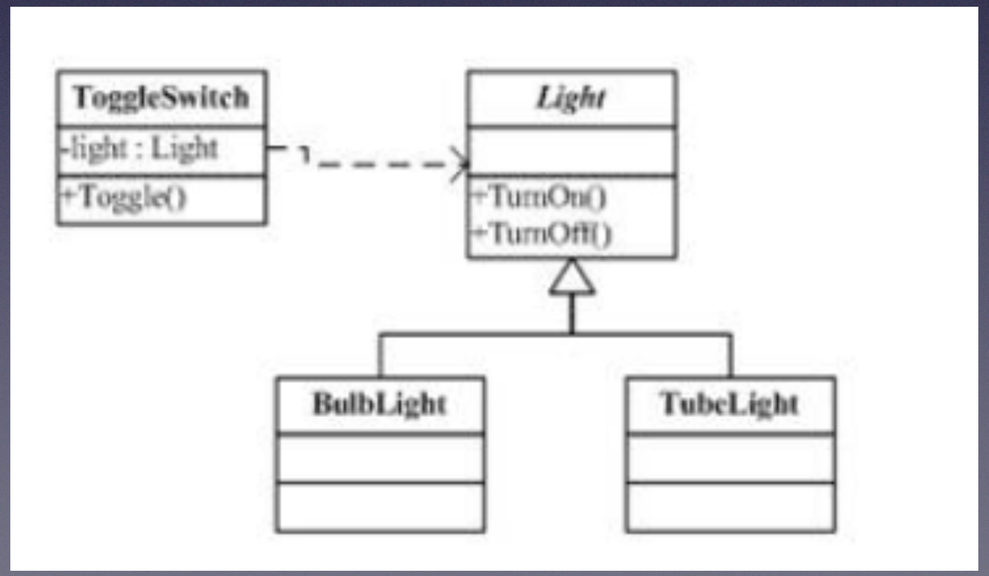
- 算法应是和数据结构以及类型无关的，各种特殊的数据类型理应做好自己分内的工作。算法只关心一个标准的实现。
- 类的方式：构造函数和析构函数以及操作符重载等特性，使得一个用户自定义的数据类型和内建的那些数据类型就很一致了。
- 通过模板达到类型和算法的妥协：模板的特化会根据使用者的类型在编译时期生成那个模板的代码。模板可以通过一个虚拟类型来做类型绑定，这样不会导致类型转换时的问题。
- 通过虚函数和运行时类型识别,这样一来，就可以写出基于抽象接口的泛型。

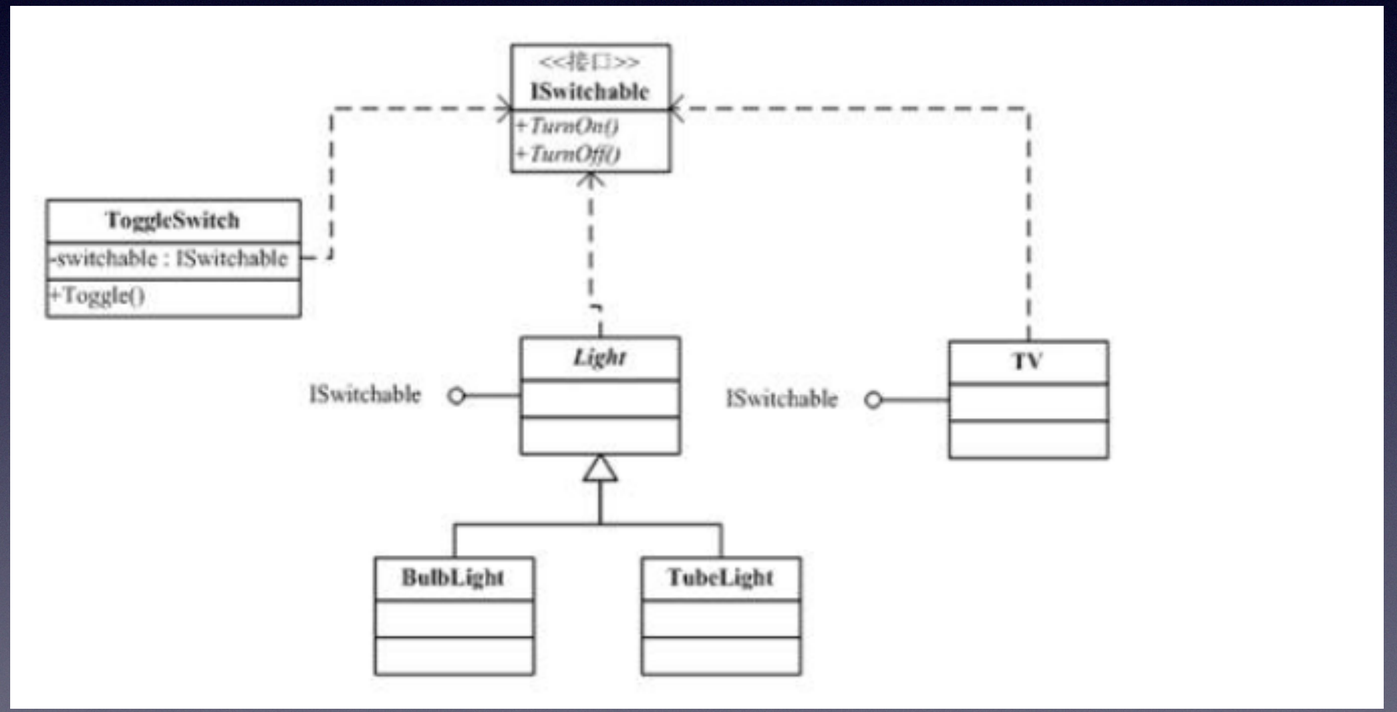
面向对象编程

- 对象是构成程序的基本单元，将程序和数据封装其中，以提高软件的可重用性、灵活性和可扩展性，对象里的程序可以访问及修改对象相关联的数据。计算机程序就是彼此相关联的对象的集合。
- 面向对象编程三大特性：封装、继承和多态。
- 面向接口编程，而不是依赖实现。
- 优先使用组合而不是继承来达到代码复用的目的。
- 继承有代码重用的功能，但更多的应该是为了多态使用。
- 面向对象编程除了 class 之后，还有其他方式，比如 javascript 的 prototype chain。

面向对象编程-例子


ToggleSwitch
-light : Light
+Toggle()
Light
+TurnOn()
+TurnOff()


ToggleSwitch
-light : Light
+Toggle()
Light
+TurnOn()
+TurnOff()
BulbLight
TubeLight


<<接口>>
ISwitchable
+TurnOn()
+TurnOff()
ToggleSwitch
-switchable : ISwitchable
+Toggle()
Light
TV
ISwitchable
ISwitchable
BulbLight
TubeLight

面向对象编程-prototype chain

```
// inherit() 返回了一个继承自原型对象p的属性的新对象
// 这里使用ECMAScript 5中的Object.create()函数（如果存在的话）
// 如果不存在Object.create()，则退化使用其他方法
function inherit(p) {
    if (p == null) throw TypeError();      // p是一个对象，但不能是null
    if (Object.create)                     // 如果Object.create()存在
        return Object.create(p);           // 直
    var t = typeof p;                      // 否
    if (t !== "object" && t !== "function") thr
    function f() {};                       // 定
    f.prototype = p;                       //将
    return new f();                        //使
}
```

```
var o = {}   // o 从 Object.prototype 继承对象的方法
o.x = 1;     // 给o定义一个属性x
var p = inherit(o); // p继承o和Object.prototype
p.y = 2;     // 给p定义一个属性y
var q = inherit(p); // q继承p、o和Object.prototype
q.z = 3;     // 给q定义一个属性z
var s = q.toString(); // toString继承自Object.prototype
q.x + q.y    // => 3: x和y分别继承自o和p
```

函数式编程

- f(a) == f(b) when a == b
- 定义输入数据和输出数据相关的关系，数学表达式里面其实是在做一种映射（mapping），输入的数据和输出的数据关系是什么样的，是用函数来定义的。

主要特征

- 函数不维护任何状态。函数式编程的核心精神是 stateless，简而言之就是它不能存在状态，打个比方，你给我数据我处理完扔出来。里面的数据是不变的。
- immutable：输入数据是不能动的，动了输入数据就有危险，所以要返回新的数据集。

优势

- 没有状态就没有伤害。
- 并行执行无伤害。
- Copy-Paste 重构代码无伤害。
- 函数的执行没有顺序上的问题。

劣势

- 数据复制比较严重 。这个劣势不见得会导致性能不好。因为没有状态，所以代码在并行上根本不需要锁（不需要对状态修改的锁），所以可以拼命地并发，反而可以让性能很不错。比如：Erlang 就是其中的代表。
- 函数式编程和过程式编程的思维方式完全不一样。过程式编程是在把具体的流程描述出来，而函数式编程的抽象度更大，在实现方式上，函数套函数，函数返回函数，函数里定义函数。

延伸阅读

- 编程范式游记 - 陈浩
- 从0开始学架构
- Programming Paradigms
- What Is Functional Programming?
- What pure functional programming is all about
- WIKI Programming paradigm
- 编程的宗派
- Three Paradigms by Robert C. Martin

Thank You!