

代码编程中的编程范式

陈皓

个人简介

- **18年工作经历，超大型分布式系统基础架构研发和设计**
- **擅长领域：金融、电子商务、云计算、大数据**
- **职业背景**
 - 阿里巴巴资深架构师（阿里云、天猫、淘宝）
 - 亚马逊高级研发经理（AWS、全球购、商品需求预测）
 - 汤森路透研发经理（实时金融数据处理基础架构）
- **目前在创业，致力于为企业提供技术架构管理产品**
 - 目标：用户不用改一行代码就可以提高系统的性能和稳定性



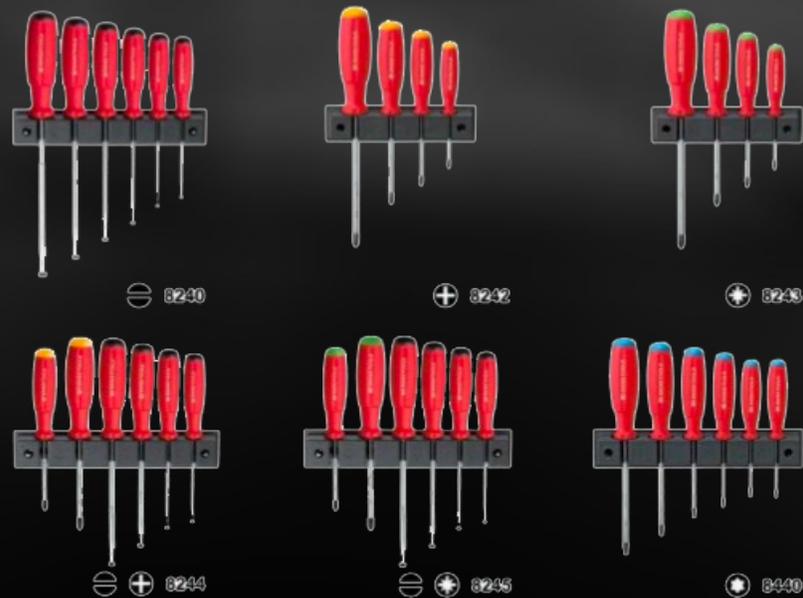
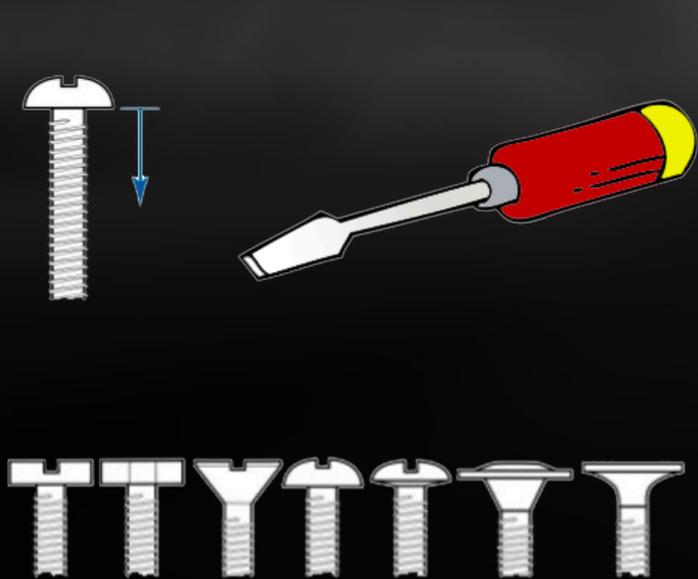


从 C 语言开始说起

C 语言的 swap 函数

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

现实世界的一个类比



是否可以做得更好？



C语言的泛型 – swap函数

```
void swap(void* x, void* y, size_t size)
{
    char temp[size]; //not ANSI C
    memcpy(temp, &y, sizeof(x));
    memcpy(&y, &x, sizeof(x));
    memcpy(&x, temp, sizeof(x));
}
```

```
#define swap(x,y) do \
{ char temp[sizeof(x) == sizeof(y) ? sizeof(x) : -1]; \
  memcpy(temp, &y, sizeof(x)); \
  memcpy(&y, &x, sizeof(x)); \
  memcpy(&x, temp, sizeof(x)); \
} while(0)
```

C语言泛型的问题

- 接口开始变得复杂，需要加入size
- 如果是字符串 - `char*`，那么 `swap`的参数是否要二级指针 - `void**`？
- 指针看不到类型，那么如果不同的类型会怎么样？ `swap (&double, &int) ??`
- C语言的宏只是做字符串替换，可能会造成很多问题
- 检查类型的长度 - `sizeof`可能会有问题
- 检查类型的长度 vs 类型转换 - 痛苦的二选一

C 语言的 search 函数

```
int search(int* a, size_t size, int target) {  
    for(int i=0; i<size; i++) {  
        if (a[i] == target) {  
            return i;  
        }  
    }  
    return -1;  
}
```

C语言 search 函数泛型化

```
int search(void* a, size_t size, void* target,
           size_t elem_size, int(*cmpFn)(void*, void*))
{
    for(int i=0; i<size; i++) {
        // why not use memcmp()
        if ( cmpFn (a + elem_size * i, target) == 0 ) {
            return i;
        }
    }
    return -1;
}
```

```
int int_cmp(int* x, int* y)
{
    return *x - *y;
}

int string_cmp(char* x, char* y){
    return strcmp(x, y);
}
```

C 语言泛型的问题

- 数据类型的自适应问题
- 随着算法越来越复杂，接口越来越复杂。
- 如果再继续进入数据结构中的泛型。如：vector, stack, map … 几乎很难了
- 太多的数据封装和基于此类数据类型的算法实现，几乎不可能完全照顾到
- 数据容器还需要解决两个问题：
 - 数据对象的内存是如何分配和释放的？
 - 数据对象的复制是如何复制的？深拷贝 还是 浅拷贝？
- 纠结 - 哪些工作应该是“用户处理”？哪些应该是“自己处理”？



泛型编程

程序抽象

- 程序的算法（或应用逻辑）应该是和数据类型甚至数据结构无关的。
- 各种特殊的数据类型（或数据结构）理应做好自己的份内的工作。
- 算法只关于一个完全标准和通用的实现。
- 对于泛型的抽象，我们需要回答一个问题：
如果让我的数据类型符合通用的算法，那么什么是数据类型最小的需求？

C++有效地解决了程序的泛型问题

- **类的出现**

- 构造函数、析构函数 — 定义了数据模型的内存分配和释放。
- 拷贝构造函数、赋值函数 — 定义了数据模型中数据的拷贝和赋值。
- 重载操作符 — 定义了数据模型中数据的操作是如何进行的

- **模板的出现**

- 根据不同的类型直接生成不同类型的函数，对不同的类型进行了有效的隔离。
- 具化的模板和特定的重载函数，可以为特定的类型指定特定的操作。

C++ 的模板泛型初探

```
long sum(int *a, size_t size) {  
    long result = 0;  
    for(int i=0; i<size; i++) {  
        result += a[i];  
    }  
    return result;  
}
```



```
template<typename T>  
T sum(T* pStart, size_t size, T target) {  
    T result = 0;  
    for(int i=0; i<size; i++) {  
        result += pStart[i];  
    }  
    return result;  
}
```

- 问题一：数组方式的迭代只适合顺序式的数据结构
- 问题二：result初始化的那个值还没有被泛型化
- 问题三：解决问题一，我们需要使用一个更通用的“迭代器”，那么，template的参数会变成“迭代器”的类型，那么，算法里面的result的类型怎么办？

泛型中需要解决的问题

1. 需要把数据类型抽象化掉。
2. 需要用一个更为通用的“迭代泛型”，而不只是基于数组的 for-loop
3. 需要一个 Value Type 的抽象。
4. 需要解决数据对象的创建、销毁、拷贝、复制、比较、+ - * /等算术操作。
5. 需要解决数据容器对内部对象的“取引用”，“”

一个糙快猛的“迭代器”

```
//Pseudocode, not C++
class Iterator {
    typedef value_type;
    Iterator operator++;
    value_type operator*();
};

bool operator!=(Iterator const&,
                Iterator const&);
```

```
template <class Iter, class T>
T sum(Iter start, Iter end, T init) {
    T s = init;
    while( start != end) {
        s = s + *start;
        start++;
    }
    return s;
}
```

- 在模板上扩展了一个参数，用于数据容器的迭代器 `Iter`，而 `T` 则变成了数据类型
- 在 `sum` 函数上也扩展了一个参数，用于做初始化值。
- 问题：我们可不可以把 `Iter` 的类型给映射到 `T` 上？()

泛型：容器、迭代器、算法

```
template <class T>
class container {
public:
    class iterator {
    public:
        typedef iterator self_type;
        typedef T value_type;
        typedef T* pointer;
        typedef T& reference;

        reference operator*();
        pointer operator->();
        bool operator==(const self_type& rhs);
        bool operator!=(const self_type& rhs);
        ...
        ...
    private:
        pointer _ptr;
    };

    iterator begin();
    iterator end();
    ...
    ...
};
```

```
template <class Iter>
typename Iter::value_type
sum(Iter start, Iter end, T init) {
    typename Iter::value_type result = init;
    while( start != end) {
        result = result + *start;
        start++;
    }
    return result;
}
```

- 泛型需要处理的三件事：

- 数据容器
- 数据容器的迭代器
- 泛型的算法

是否足够泛型了？

- **如果我们有这样的数据结构**
 - 如果我即想计算员工的总薪水，也想计算员工的总休假
 - 如果我想想计算员工中拿薪水最高了，和休假最少的？
 -
- **面对这么多的需求，我们是否还能再泛型一些？**

```
struct Employee {  
    string name;  
    string id;  
    int vacation;  
    double salary;  
};  
  
vector<Employee> staff;  
//total salary or total vacation days?  
sum(staff.begin(), staff.end(), 0);  
  
//what if I need max salary or min vacation?
```

更加泛型 - Reduce函数

```
template<class Iter, class T, class Op>
T reduce (Iter start, Iter end, T init, Op op) {
    T result = init;
    while ( start != end ) {
        result = op( result, *start );
        start++;
    }
    return result;
}
```

```
double sum_salries =
    reduce( staff.begin(), staff.end(), 0.0
           [](double s, Employee e)
             {return s + e.salary;} );

double max_slary =
    reduce( staff.begin(), staff.end(), 0.0
           [](double s, Employee e)
             {return s > e.salary? s: e.salary; } );
```

- 使用函数对象有两个好处：
 - 函数式编程
 - 不用维护状态

函数式可以组合出更多的东西

```
template<class T, class Cond>
struct counter {
    size_t operator()(size_t c, T t) const {
        return c + (Cond(t) ? 1 : 0);
    }
};

template<class Iter, class Cond>
size_t count_if(Iter begin, Iter end, Cond c){
    return reduce(begin, end, 0,
                 counter<Iter::value_type, Cond>(c));
}

size_t cnt = count_if(staff.begin(), staff.end(),
                    [](Employee e){ return e.salary > 10000});
```



函数式编程

Booleans, integers, (and other data structures) *can be entirely replaced by functions!*

"Church encodings"

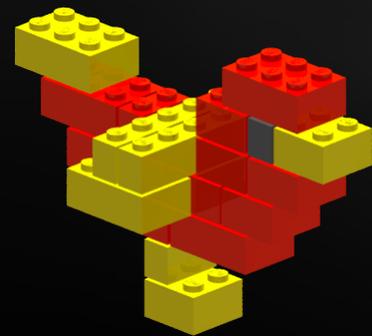
Early versions of the Glasgow Haskell compiler actually implemented data-structures this way!



Alonzo Church

函数式编程

- **借鉴于数学代数，函数是所有的一切**
 - 函数就是一个表达式，是一个单纯的运算过程
- **通过对比简单的函数的组合，完成复杂的功能**
- **函数只是定义“输入数据”到“输出数据”的相关关系（表达式）**
 - $f(x) = 5x^2 + 4x + 3$
 - $g(x) = 2f(x) + 5 = 10x^2 + 8x + 11$
 - $h(x, y) = f(x) + g(y) = 15x^2 + 12x + 14$
 - $f(x) = f(f(x - 1) + f(x - 2))$



函数式的核心精神 – Pure Function

- **特征**
 - Stateless - 函数不维护任何状态
 - Immutable - 也不修改输入数据, 返回新的数据集
- **好处**
 - 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
- **劣处**
 - 数据复制比较严重, 性能不是很好
- **完全纯函数式**
 - Haskell
- **容易写纯函数**
 - F#, Ocaml, Clojure, Scala
- **纯函数需要花点精力**
 - C#, Java, JavaScript

函数式编程 - 无状态的函数

```
// 非函数式, 不是pure function, 有状态
int cnt;
void increment(){
    cnt++;
}

// 函数式, pure function, 无状态
int increment(int cnt){
    return cnt+1;
}
```

不依赖也不改变外部数据的值

主要描述输入数据和输出数据的关系

```
def inc(x):
    def incx(y):
        return x+y
    return incx

inc2 = inc(2)
inc5 = inc(5)

print inc2(5) # 输出 7
print inc5(5) # 输出 10
```

函数就是表达式

关注于描述问题而不是怎么实现

State is like a box of chocolates.
You never know what you are gonna get.



函数式代码示例 – Scheme

```
(define (plus x y) (+ x y))
(define (times x y) (* x y))
(define (square x) (times x x))

(define (f1 x) ;;; f(x) = 5 * x^2 + 10
  (plus 10 (times 5 (square x))))

(define f2
  (lambda (x)
    (define plus
      (lambda (a b) (+ a b)))
    (define times
      (lambda (a b) (* a b)))
    (plus 10 (times 5 (times x x)))))
```

```
;;; recursion
(define factorial (lambda (x)
  (if (<= x 1) 1
      (* x (factorial (- x 1))))))

(newline)
(display(factorial 6))

;;; another version of recursion
(define (factorial_x n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))
  (iter 1 1))
```

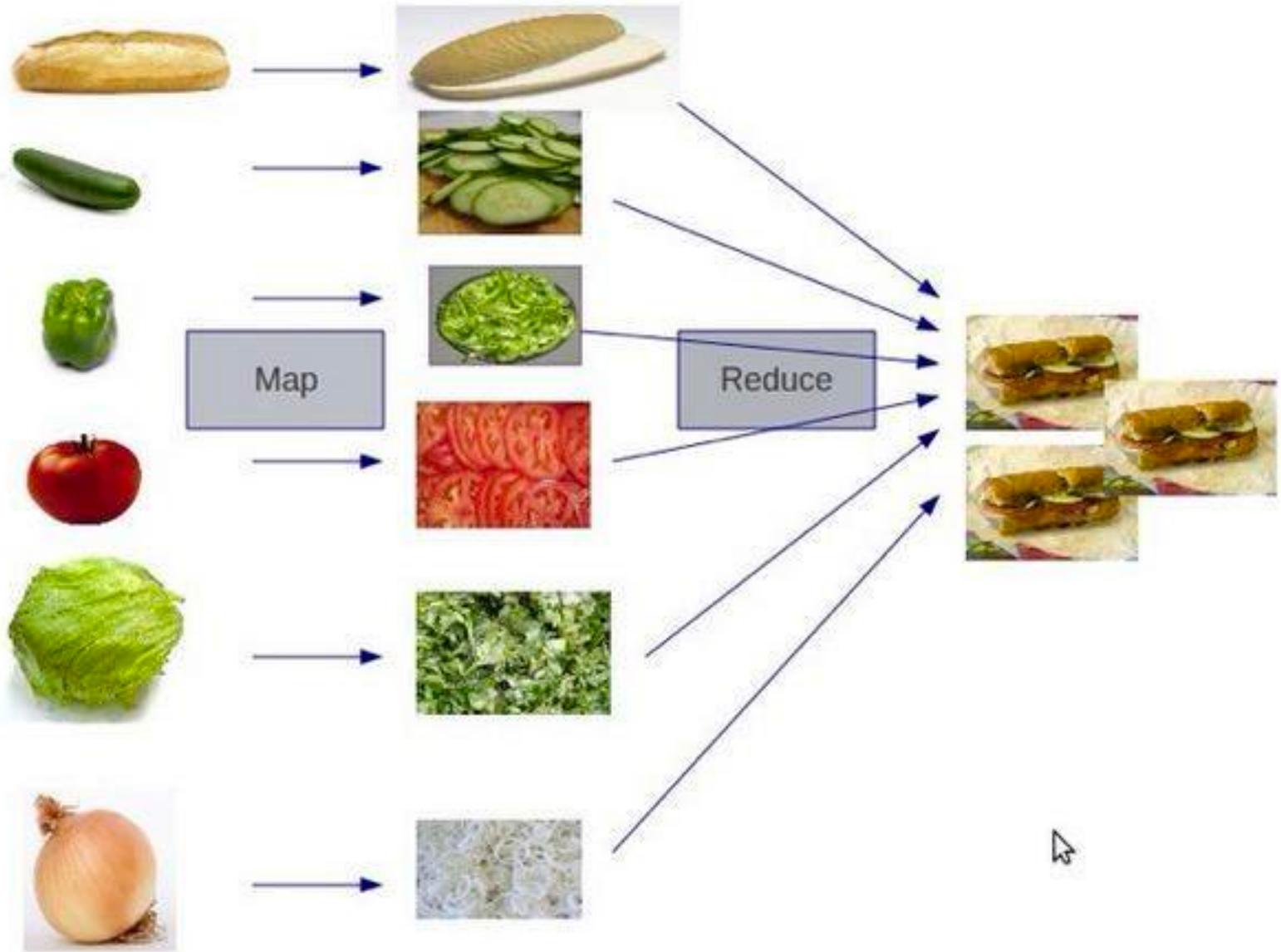
函数式的三套件 – map/reduce/filter

```
# 传统的非函数式
upname = ['HAO', 'CHEN', 'COOLSHELL']
lowname = []
for i in range(len(upname)):
    lowname.append( upname[i].lower() )

# 函数式
def toUpper(item):
    return item.upper()

upper_name = map(toUpper, ["hao", "chen", "coolshell"])
```

```
string s="hello";
transform(s.begin(), s.end(), back_inserter(out), ::toupper);
```



函数式编程的示例

- 代码变得更简洁和优雅了
- 数据集，操作，返回值都放到了一起。
- 没有了循环体，于是就可以少了些临时变量，以及变量倒来倒去逻辑
- 代码变成了在描述你要干什么，而不是怎么去干

```
# 计算数组中正数的平均值
num = [2, -5, 9, 7, -2, 5, 3, 1, 0, -3, 8]
positive_num_cnt = 0
positive_num_sum = 0
for i in range(len(num)):
    if num[i] > 0:
        positive_num_cnt += 1
        positive_num_sum += num[i]

if positive_num_cnt > 0:
    average = positive_num_sum / positive_num_cnt

print average

#计算数组中正数的平均值
positive_num = filter(lambda x: x>0, num)
average = reduce(lambda x,y: x+y, positive_num) / len( positive_num )

vector num {2, -5, 9, 7, -2, 5, 3, 1, 0, -3, 8};
vector p_num;
copy_if(num.begin(), num.end(), back_inserter(p_num), [](int i){ return (i>0);} );
int average = accumulate(p_num.begin(), p_num.end(), 0) / p_num.size();
```

函数式编程 vs Unix 管道

- **Unix Shell pipeline**

```
ps auwx | grep chen hao | awk '{print $2}' | xargs echo
```

- **抽象成函数式的样子**

```
# 抽象成函数式的语言
```

```
xargs(echo, awk('print $2', grep ("chenhao", ps(auwx))))
```

```
# 也可以如下所示
```

```
pids = for_each(result, [ps_auwx, grep_chen hao, awk_p2, xargs_echo])
```

面向过程 vs 函数式

```
def process(num):  
    # filter out non-evens  
    if num % 2 != 0:  
        return  
    num = num * 3  
    num = 'The Number: %s' % num  
    return num  
  
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
for num in nums:  
    print process(num)
```

```
def even_filter(nums):  
    for num in nums:  
        if num % 2 == 0:  
            yield num  
def multiply_by_three(nums):  
    for num in nums:  
        yield num * 3  
def convert_to_string(nums):  
    for num in nums:  
        yield 'The Number: %s' % num  
  
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
pipeline = convert_to_string(  
    multiply_by_three(  
        even_filter(nums) ) )
```

函数式编程示例

我们有3辆车比赛，简单起见，我们分别给这3辆车有70%的概率可以往前走一步，一共有5次机会，我们打出每一次这3辆车的前行状态。

```
from random import random

time = 5
car_positions = [1, 1, 1]

while time:
    # decrease time
    time -= 1

    print ''
    for i in range(len(car_positions)):
        # move car
        if random() > 0.3:
            car_positions[i] += 1

    # draw car
    print '-' * car_positions[i]
```

抽取出函数 - 但函数间强耦合

```
def move_cars():
    for i, _ in enumerate(car_positions):
        if random() > 0.3:
            car_positions[i] += 1

def draw_car(car_position):
    print '-' * car_position

def run_step_of_race():
    global time
    time -= 1
    move_cars()

def draw():
    print ''
    for car_position in car_positions:
        draw_car(car_position)
```

```
time = 5
car_positions = [1, 1, 1]
```

```
from random import random

while time:
    run_step_of_race()
    draw()
```

函数式编程

- 它们之间没有共享的变量
- 函数间通过参数和返回值来传递数据
- 在函数里没有状态。

```
from random import random

def move_cars(car_positions):
    return map(lambda x: x + 1 if random() > 0.3 else x,
              car_positions)

def output_car(car_position):
    return '-' * car_position

def run_step_of_race(state):
    return {'time': state['time'] - 1,
           'car_positions': move_cars(state['car_positions'])}

def draw(state):
    print ''
    print '\n'.join(map(output_car, state['car_positions']))

def race(state):
    draw(state)
    if state['time']:
        race(run_step_of_race(state))

race({'time': 5,
     'car_positions': [1, 1, 1]})
```



Python 的修饰器

Python 的修饰器 - 示例一

```
def hello(fn):  
    def wrapper():  
        print "hello, %s" % fn.__name__  
        fn()  
        print "goodby, %s" % fn.__name__  
    return wrapper
```

```
@hello  
def foo():  
    print "i am foo"
```

```
foo()
```

```
@decorator  
def func():  
    pass  
  
func = decorator(func)
```

```
foo = hello(foo)
```

```
$ python hello.py
```

```
hello, foo
```

```
i am foo
```

```
goodby, foo
```

Python 修饰器 - 示例二

```
def makeHtmlTag(tag, *args, **kws):
    def real_decorator(fn):
        css_class = " class='{0}'".format(kws["css_class"]) \
            if "css_class" in kws else ""
        def wrapped(*args, **kws):
            return "<"+tag+css_class+">" + fn(*args, **kws) + "</"+tag+">"
        return wrapped
    return real_decorator

@makeHtmlTag(tag="b", css_class="bold_css")
@makeHtmlTag(tag="i", css_class="italic_css")
def hello():
    return "hello world"

print hello()
```

```
@decorator_one
@decorator_two
def func():
    pass

func = decorator_one(decorator_two(func))

@decorator(arg1, arg2)
def func():
    pass

func = decorator(arg1, arg2)(func)
```

输出

```
<b class='bold_css'><i class='italic_css'>hello world</i></b>
```

Python 修饰器 + 管道

```
class Pipe(object):  
    def __init__(self, func):  
        self.func = func  
  
    def __ror__(self, other):  
        def generator():  
            for obj in other:  
                if obj is not None:  
                    yield self.func(obj)  
        return generator()
```

```
@Pipe  
def even_filter(num):  
    return num if num % 2 == 0 else None  
  
@Pipe  
def multiply_by_three(num):  
    return num*3  
  
@Pipe  
def convert_to_string(num):  
    return 'The Number: %s' % num  
  
@Pipe  
def echo(item):  
    print item  
    return item
```

```
def force(sqs):  
    for item in sqs: pass  
  
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
force(nums | even_filter | multiply_by_three | convert_to_string | echo)
```



面向对象编程

面向对象

- 把数据、属性、方法的封装或抽象成对象。
- 每个对象都可以接受/处理数据并将数据传达给其它对象，就像一个小型独立的“机器”。
- 通过独立对象的抽象（多态），提高软件的重用性、灵活性和扩展性
- 支持面向对象的语言：
 - Common Lisp、Python、C++、Objective-C、Smalltalk、Delphi、Java、Swift、C#、Perl、Ruby、PHP、Javascript等。

面向对象的核心理念

- **"Program to an 'interface', not an 'implementation'."**
 - 使用者不需要知道数据类型、结构、算法的细节。
 - 使用者不需要知道实现细节，只需要知道提供的接口。
 - 利于抽象、封装，动态绑定，多态。
 - 符合面向对象的特质和理念。
- **"Favor 'object composition' over 'class inheritance'."**
 - 继承需要给子类暴露一些父类的设计和实现细节。
 - 父类的实现的改变会造成子类也需要改变。
 - 我们以为继承主要是为了代码重用，但实际上在子类中需要重新实现很多父类的方法。
 - 继承更多的应该是为了多态。

面向对象算法拼装

```
interface BillingStrategy {
    public double getActPrice(double rawPrice);
}

// Normal billing strategy (unchanged price)
class NormalStrategy implements BillingStrategy {
    @Override
    public double getActPrice(double rawPrice) {
        return rawPrice;
    }
}

// Strategy for Happy hour (50% discount)
class HappyHourStrategy implements BillingStrategy {
    @Override
    public double getActPrice(double rawPrice) {
        return rawPrice * 0.5;
    }
}
```

```
class Order {
    private List<Double> orderItems = new ArrayList<Double>();
    private BillingStrategy strategy = new NormalStrategy();

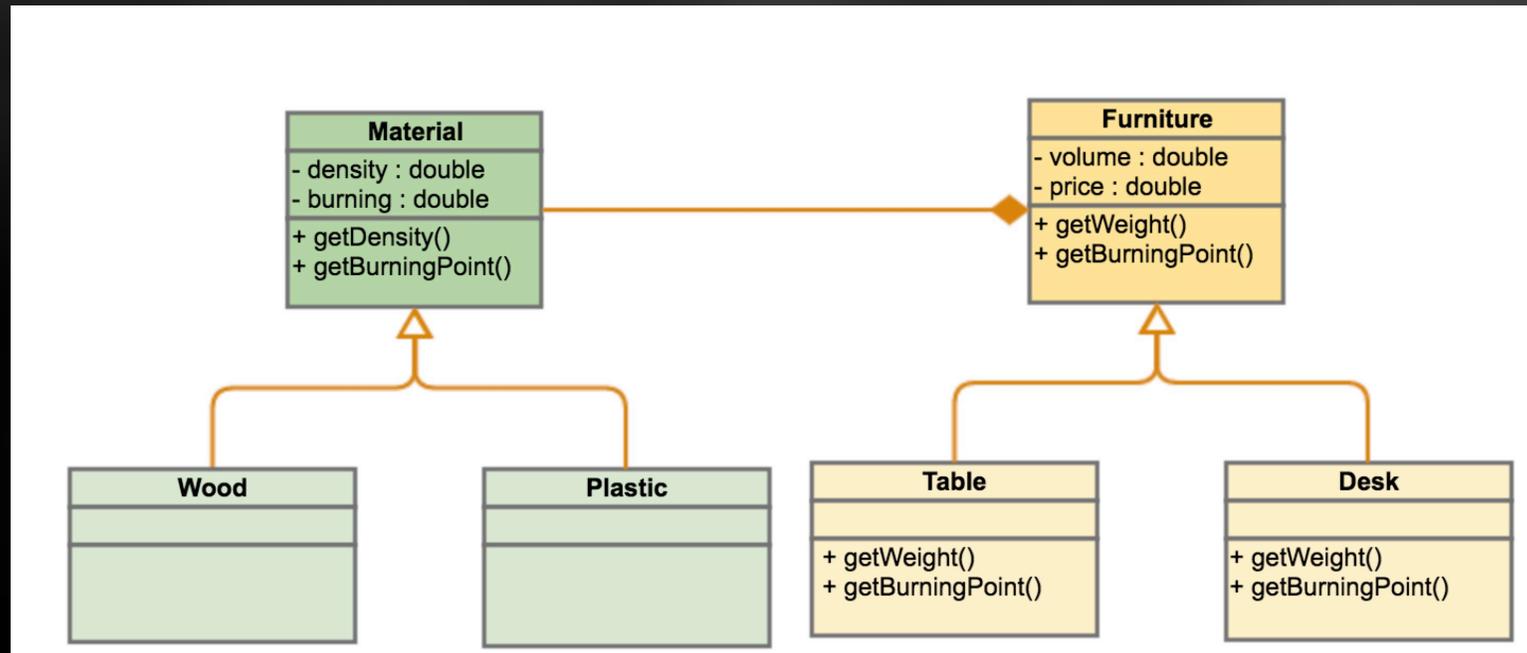
    public void add(double price, int quantity) {
        orderItems.add(strategy.getActPrice(price * quantity));
    }

    // Payment of bill
    public void payBill() {
        double sum = 0;
        for (Double item : orderItems) {
            sum += strategy.getActPrice(item);
        }
        System.out.println("Total due: " + sum);
    }

    // Set Strategy
    public void setStrategy(BillingStrategy strategy) {
        this.strategy = strategy;
    }
}
```

面向对象中的对象拼装

- 示例：
 - 四个物体：木头桌子、木头椅子、塑料桌子、塑料椅子
 - 四个属性：燃点、密度、价格、重量



面向对象的优缺点

- **优点**

- 能和真实的世界相辉映，符合人的直觉。
- 面向对象和数据库模型设计类型，更多的关注对象间的模型设计。
- 强调于“名词”而不是“动词”，更多的关注对象和对象间的接口。
- 根据业务的特征形成一个个高内聚的对象，有效地分离了抽象和具体实现，增强了重用性和扩展性。
- 拥有大量非常优秀的设计原则和设计模式 – 、S.O.L.I.D、IoC/DI……

- **批评**

- 代码都需要附着在一个类上，从一侧面说，其鼓励了类型。
- 代码需要通过对象来达到抽象的效果，导致了相当厚重的“代码粘合层”。
- 因为太多的封装以及对状态的鼓励，导致了大量不透明并在并发下出现很多问题。

Prototype-based 基于原型编程

- 是面向对象编程的一种方式。没有class化的，直接使用对象。又叫，基于实例的编程。
- 主流的语言是 Javascript
- 比较
 - 基于类的面向对象，关于于类和子类的模型。
 - 基于原型的面向对象，关注一系列对象实例的行为，而之后才关心如何将对象划分到最近的使用方式相似的原型对象，而不是分成类。
- 很多基于原型的系统提倡运行时原型的修改，而基于类的面向对象系统只有动态语言允许类在运行时被修改（Common Lisp, Dylan, Objective-C, Perl, Python, Ruby, or Smalltalk）。

一个简单的示例

```
//Define human class
var Person = function (fullName) {
  this.fullName = fullName;
  // Add a couple of methods to Person.prototype
  this.speak = function(){
    console.log("I speak English!");
  };
  this.introduction = function(){
    console.log("Hi, I am " + this.fullName);
  };
}

//Define Student class
var Student = function(fullName, school, courses) {

  Person.call(this, fullName);

  // Initialize our Student properties
  this.school = school;
  this.courses = courses;
  // override the "introduction" method
  this.introduction= function(){
    console.log("Hi, I am " + this.fullName +
      ". I am a student of " + this.school +
      ", I study "+ this.courses +".");
  };
  // Add a "exams" method
  this.takeExams = function(){
    console.log("This is my exams time!");
  };
};
```

```
// Create a Student.prototype object that inherits
// from Person.prototype.
Student.prototype = Object.create(Person.prototype);

// Set the "constructor" property to refer to Student
Student.prototype.constructor = Student;

var student = new Student("Hao Chen",
  "XYZ University",
  "Computer Science");

student.introduction();
student.speak();
student.takeExams();

// Check that instanceof works correctly
console.log(student instanceof Person); // true
console.log(student instanceof Student); // true
```

- 担心：正确性、安全性、可预测性以及效率
- 在 ECMAScript 标准的第四版开始寻求使 JavaScript 提供基于类的构造，且 ECMAScript 第六版有提供"class"(类)作为原有的原型架构之上的语法糖，提供建构对象与处理继承时的另一种语法



GO 语言的委托模式

Go语言的接口和委托

```
type Widget struct {
    X, Y int
}

type Label struct {
    Widget      // Embedding (delegation)
    Text string // Aggregation
    X int       // Override
}

func (label Label) Paint() {
    // [0xc4200141e0] - Label.Paint("State")
    fmt.Printf("[%p] - Label.Paint(%q)\n",
        &label, label.Text)
}
```

```
label := Label{Widget{10, 10}, "State", 100}

// X=100, Y=10, Text=State, Widget.X=10
fmt.Printf("X=%d, Y=%d, Text=%s Widget.X=%d\n",
    label.X, label.Y, label.Text,
    label.Widget.X)
fmt.Println()
// {Widget:{X:10 Y:10} Text:State X:100}
// {{10 10} State 100}
fmt.Printf("%+v\n%v\n", label, label)
```

- 通过直接写入类型达到委托的效果
- 如果有成员变量重名，则需要解决冲突

Go语言的接口和委托

```
type Button struct {
    Label // Embedding (delegation)
}

func NewButton(x, y int, text string) Button {
    return Button{Label{Widget{x, y}, text, x}}
}

func (button Button) Paint() { // Override
    fmt.Printf("[%p] - Button.Paint(%q)\n",
        &button, button.Text)
}

func (button Button) Click() {
    fmt.Printf("[%p] - Button.Click()\n", &button)
}
```

```
type ListBox struct {
    Widget // Embedding (delegation)
    Texts []string // Aggregation
    Index int // Aggregation
}

func (listBox ListBox) Paint() {
    fmt.Printf("[%p] - ListBox.Paint(%q)\n",
        &listBox, listBox.Texts)
}

func (listBox ListBox) Click() {
    fmt.Printf("[%p] - ListBox.Click()\n", &listBox)
}
```

```
type Painter interface {
    Paint()
}

type Clicker interface {
    Click()
}
```

```
for _, painter := range []Painter{label, listBox, button1, button2} {
    painter.Paint()
}
```

```
for _, widget := range []interface{}{label, listBox, button1, button2} {
    if clicker, ok := widget.(Clicker); ok {
        clicker.Click()
    }
}
```

```

type IntSet struct {
    data map[int]bool
}

func NewIntSet() IntSet {
    return IntSet{make(map[int]bool)}
}

func (set *IntSet) Add(x int) {
    set.data[x] = true
}

func (set *IntSet) Delete(x int) {
    delete(set.data, x)
}

func (set *IntSet) Contains(x int) bool {
    return set.data[x]
}

```

- 通过委托扩展原有的功能
- UndoableIntSet 增加了一个Undo的方法

```

type UndoableIntSet struct { // Poor style
    IntSet // Embedding (delegation)
    functions []func()
}

func NewUndoableIntSet() UndoableIntSet {
    return UndoableIntSet{NewIntSet(), nil}
}

func (set *UndoableIntSet) Add(x int) { // Override
    if !set.Contains(x) {
        set.data[x] = true
        set.functions = append(set.functions, func() { set.Delete(x) })
    } else {
        set.functions = append(set.functions, nil)
    }
}

func (set *UndoableIntSet) Delete(x int) { // Override
    if set.Contains(x) {
        delete(set.data, x)
        set.functions = append(set.functions, func() { set.Add(x) })
    } else {
        set.functions = append(set.functions, nil)
    }
}

```

```

func (set *UndoableIntSet) Undo() error {
    if len(set.functions) == 0 {
        return errors.New("No functions to undo")
    }
    index := len(set.functions) - 1
    if function := set.functions[index]; function != nil {
        function()
        set.functions[index] = nil // Free closure for garbage collection
    }
    set.functions = set.functions[:index]
    return nil
}

```

```

type Undo []func()

func (undo *Undo) Add(function func()) {
    *undo = append(*undo, function)
}

func (undo *Undo) Undo() error {
    functions := *undo
    if len(functions) == 0 {
        return errors.New("No functions to undo")
    }
    index := len(functions) - 1
    if function := functions[index]; function != nil {
        function()
        functions[index] = nil // Free closure for garbage collection
    }
    *undo = functions[:index]
    return nil
}

```

- 单独实现一个Undo 的泛型
- 这样可以直接委托给其它结构，从而完成这个功能
- 不足，每个结构都需要实现类似的Undo协议

```

type IntSet struct {
    data map[int]bool
    undo Undo
}

func NewIntSet() IntSet {
    return IntSet{data: make(map[int]bool)}
}

func (set *IntSet) Add(x int) {
    if !set.Contains(x) {
        set.data[x] = true
        set.undo.Add(func() { set.Delete(x) })
    } else {
        set.undo.Add(nil)
    }
}

func (set *IntSet) Delete(x int) {
    if set.Contains(x) {
        delete(set.data, x)
        set.undo.Add(func() { set.Add(x) })
    } else {
        set.undo.Add(nil)
    }
}

func (set *IntSet) Undo() error {
    return set.undo.Undo()
}

func (set *IntSet) Contains(x int) bool {
    return set.data[x]
}

```



编程的本质

程序的本质

- Pascal语言之父Niklaus Wirth在70年代提出：
 - Program = Data Structure + Algorithm
- 随后逻辑学家和计算机科学家R Kowalski进一步提出：
 - Algorithm = Logic + Control
- **程序的复杂性有两个：**
 - 一个是代码需要处理的逻辑，也就是我们说的业务逻辑
 - 另一个是是控制，控制或组织代码完成复杂的逻辑

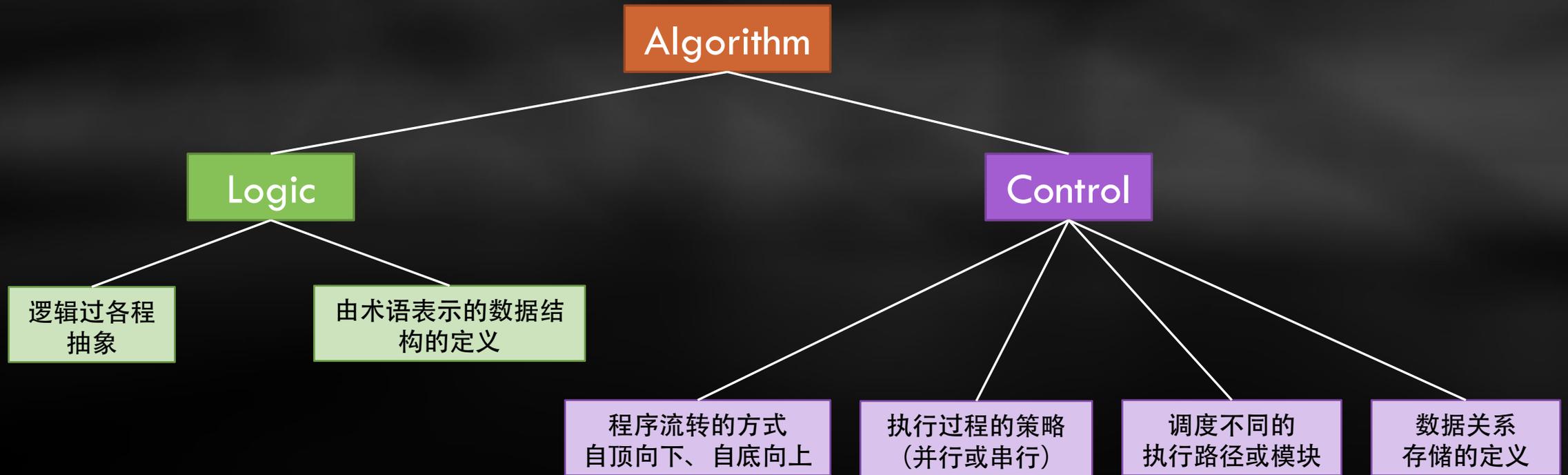
示例：Control Logic 混在一起

- 通配符匹配

- isMatch("aa","a") → false
- isMatch("aa","aa") → true
- isMatch("aaa","aa") → false
- isMatch("aa", "*") → true
- isMatch("aa", "a*") → true
- isMatch("ab", "?*") → true
- isMatch("aab", "c*a*b") → false

```
bool isMatch(const char *s, const char *p) {
    const char *last_s = NULL;
    const char *last_p = NULL;
    while( *s != '\0' ){
        if (*p=='*'){
            p++;
            if (*p=='\0') return true;
            last_s = s;
            last_p = p;
        }else if (*p=='?' || *s == *p){
            s++; p++;
        }else if (last_s != NULL){
            p = last_p;
            s = ++last_s;
        }else{
            return false;
        }
    }
    while (*p == '*') p++;
    return *p == '\0';
}
```

程序 的本质

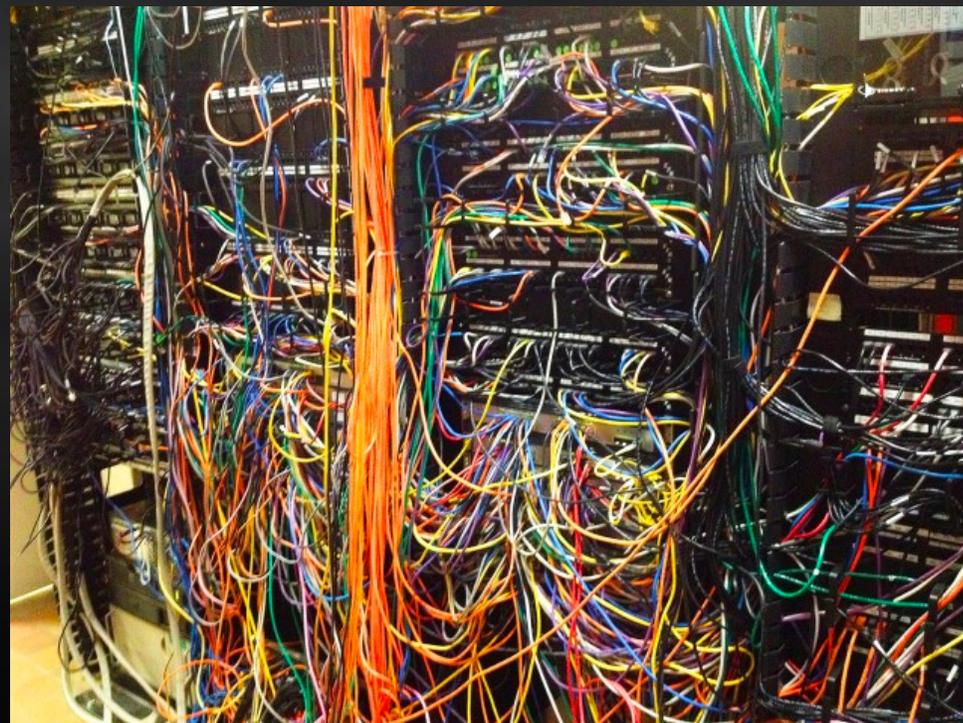


- **Logic 部分才是真正有意义的 - What**
- **Control部分只是影响Logic部分的效率 - How**

程序的复杂度

- 业务逻辑的复杂度决定了代码的复杂度
- 控制逻辑的复杂度 + 业务逻辑的复杂度 ==> 程序代码的混乱不堪
- 绝大多数程序复杂混乱的根本原因：

业务逻辑 与 控制逻辑 的 耦合



如何分离 Control 和 Logic

- **State Machine**
 - 状态定义,
 - 状态变迁条件
 - 状态的action
- **DSL – Domain Specific Language**
 - HTML, SQL, Unix Shell Script, AWK, 正则表达式……
- **编程范式**
 - 面向对象 - 委托、策略、桥接、修饰、IoC/DIP、MVC……
 - 函数式编程 - 修饰、管道、拼装
 - 逻辑推导式编程 - Prolog

一个简单的示例

```
function check_form() {
  var name = $('#name').val();
  if (null == name || name.length <= 3) {
    return { status : 1, message: 'Invalid name' };
  }
  var password1 = $('#password1').val();
  if (null == password1 || password1.length <= 8) {
    return { status : 2, message: 'Invalid password' };
  }
  var password2 = $('#password2').val();
  if (password2 != password1) {
    return { status : 3, message: 'Password mismatch' };
  }
  var email = $('#email').val();
  if (check_email_format(email)) {
    return { status : 4, message: 'Invalid email' };
  }
  return { status : 0, message: 'OK' };
}
```

```
var meta_create_user = {
  form_id : 'create_user',
  fields : [
    { id : 'name', type : 'text', min_length : 3 },
    { id : 'password1', type : 'password', min_length : 8 },
    { id : 'password2', type : 'password', min_length : 8 },
    { id : 'email', type : 'email' }
  ]
};

var r = check_form(meta_create_user);
```



逻辑编程范式

逻辑编程范式

- 逻辑编程的要点是将正规的逻辑风格带入电脑程序设计之中。
- 逻辑编程建立了描述一个问题里的世界的逻辑模型。
- 逻辑编程的目标是对它的模型建立新的陈述。
 - 通过陈述事实——因果关系
 - 程序自动推导出相关的逻辑

```
program mortal(X) :- person(X).  
  
person(Socrates).  
person(Plato).  
person(Aristotle).  
  
mortal_report:-  
write('Known mortals are:'),nl, mortal(X),  
write(X),nl,  
fail.
```

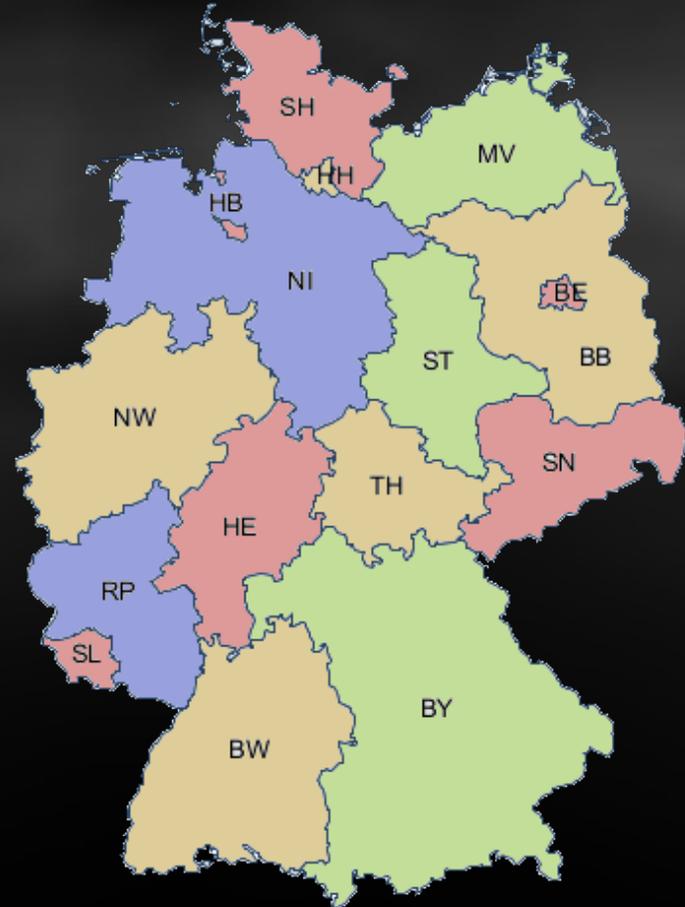
地图四色问题 - Prolog

```
/* 定义四种颜色 */
color(red).
color(green).
color(blue).
color(yellow).

/* 定义逻辑 */
neighbor(StateAColor, StateBColor) :-
    color(StateAColor),
    color(StateBColor),
    StateAColor \= StateBColor.
/* \= is the not equal operator */

/* 定义地图 BW 和 BY 相邻 */
germany(BW, BY) :- neighbor(BW, BY).

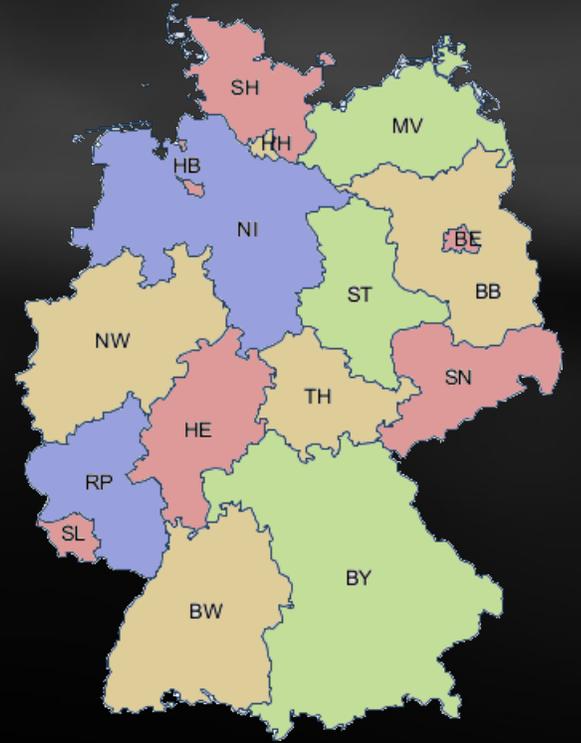
/* 定义地图 BW 和BY RP、HE相邻关系 */
germany(BW, BY, RP, HE) :-
    neighbor(BW, BY),
    neighbor(BW, RP),
    neighbor(BW, HE).
```



地图四色问题 - Prolog

```
/* 定义整个地图的相邻关系 */
germany(SH, MV, HH, HB, NI, ST, BE, BB, SN, NW, HE, TH, RP, SL, BW, BY) :-
  neighbor(SH, NI), neighbor(SH, HH), neighbor(SH, MV),
  neighbor(HH, NI),
  neighbor(MV, NI), neighbor(MV, BB),
  neighbor(NI, HB), neighbor(NI, BB), neighbor(NI, ST), neighbor(NI, TH),
  neighbor(NI, HE), neighbor(NI, NW),
  neighbor(ST, BB), neighbor(ST, SN), neighbor(ST, TH),
  neighbor(BB, BE), neighbor(BB, SN),
  neighbor(NW, HE), neighbor(NW, RP),
  neighbor(SN, TH), neighbor(SN, BY),
  neighbor(RP, SL), neighbor(RP, HE), neighbor(RP, BW),
  neighbor(HE, BW), neighbor(HE, TH), neighbor(HE, BY),
  neighbor(TH, BY),
  neighbor(BW, BY).

/* 推导求值 */
?- germany(SH, MV, HH, HB, NI, ST, BE, BB, SN, NW, HE, TH, RP, SL, BW, BY).
```





程序世界里的编程范式

```
utils:File;
FileStream;
void fill meta data;
objInfo:Object;
tv_file
strSource:String;
videoFileName:String;
od_pic:MovieClips;
```

Left brain

I am the left brain.
I am a scientist. A mathematician.
I love the familiar. I categorize. I am accurate. Linear.
Analytical. Strategic. I am practical.
Always in control. A master of words and language.
Realistic. I calculate equations and play with numbers.
I am order. I am logic.
I know exactly who I am.

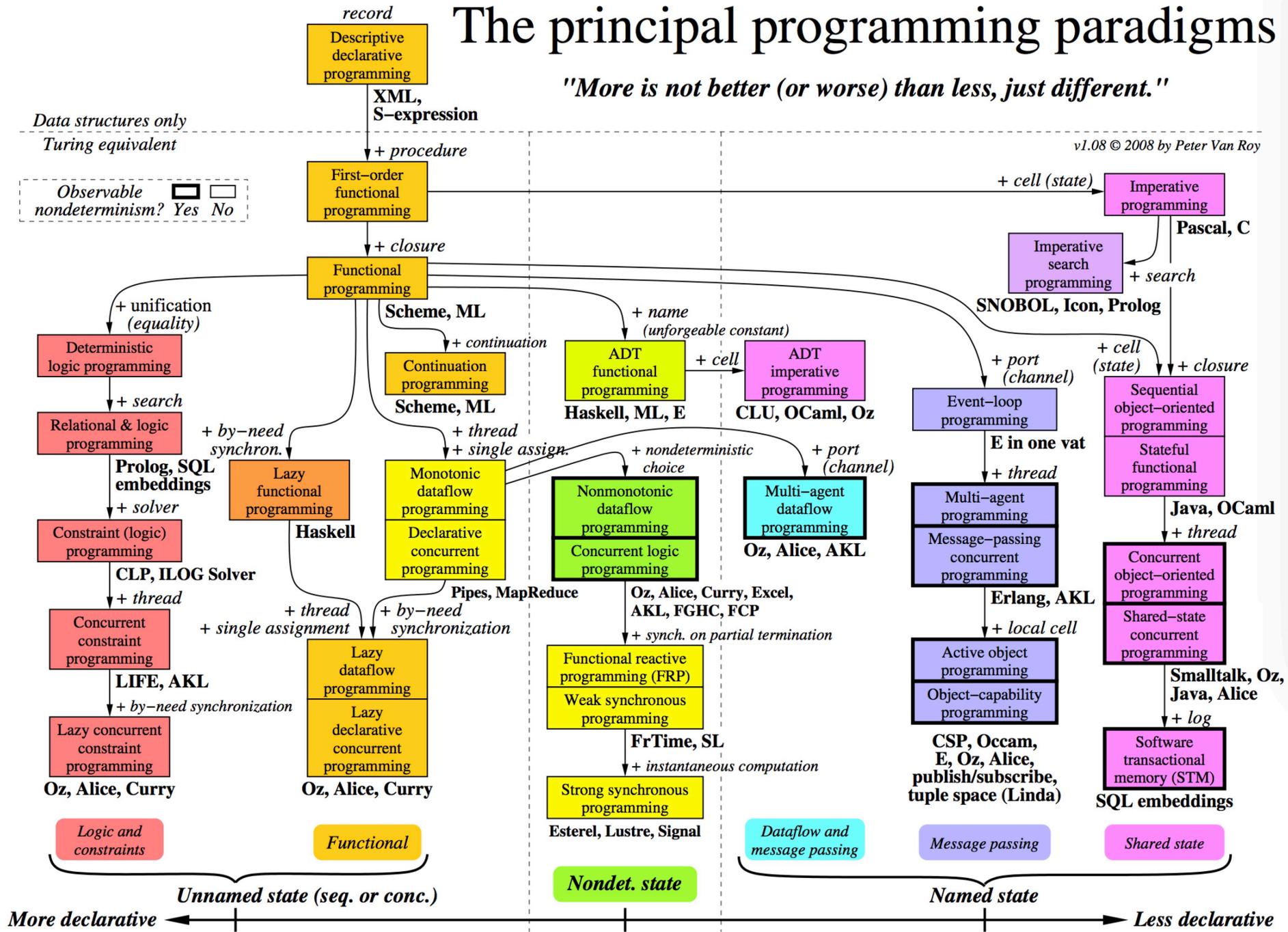
理性分析型
喜欢数据证据
线性思维
陷入细节
具体化的

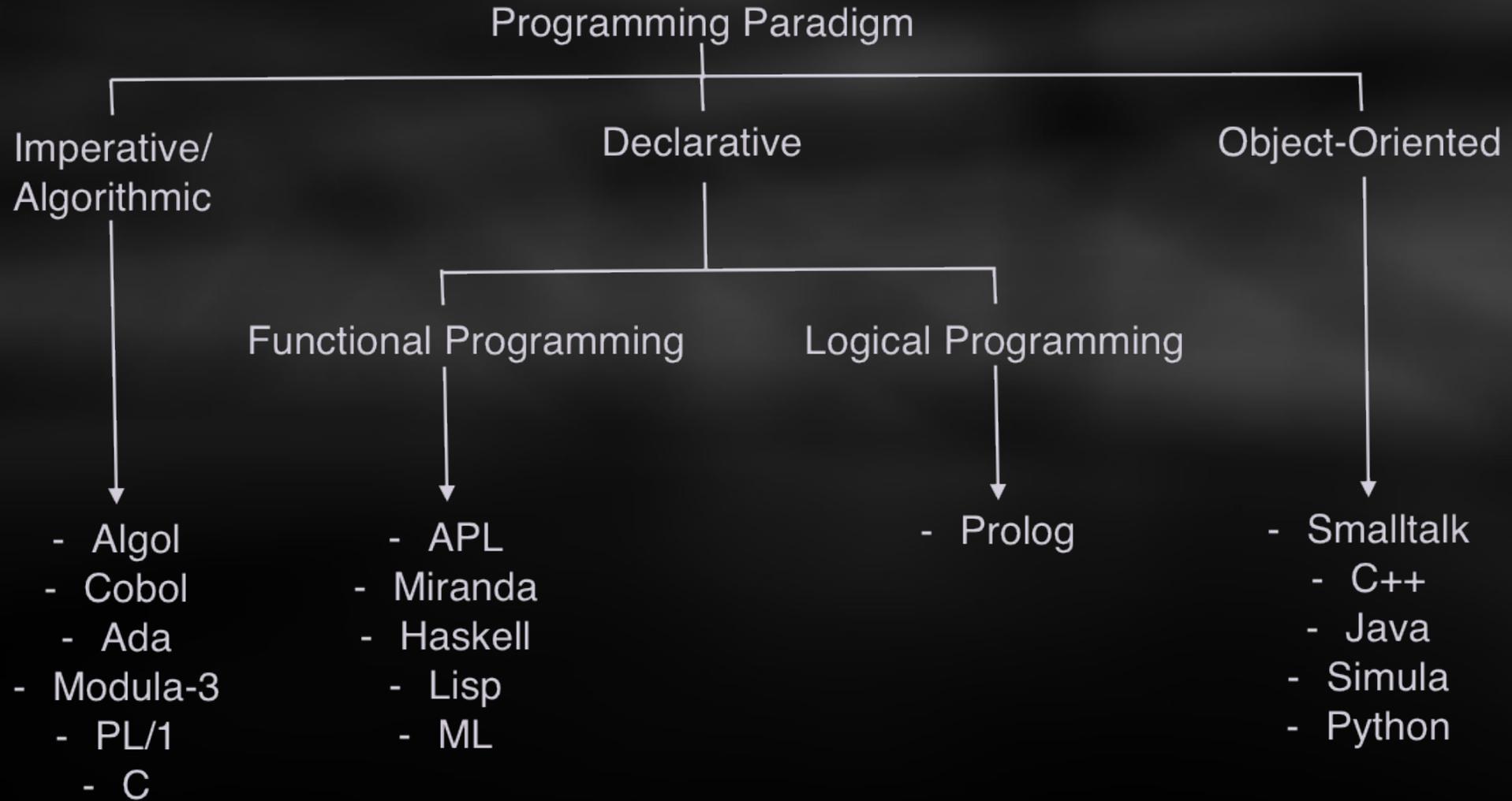
```
5 310011: begin // DAN 1144
// decimal adjust and...
// results in nibbles
if (word11 <= 0) {
    i_axxor, word11, rdb a
    ebad <= deus dca; // linear
    pc <<= pc+8 h; // Next instruction
end
6 3000:00, 6 3001:00, 6 3010:00, 6 3020:00, 6 3030:00, 6 3040:00, 6 3050:00, 6 3060:00, 6 3070:00, 6 3080:00, 6 3090:00, 6 3100:00, 6 3110:00, 6 3120:00, 6 3130:00, 6 3140:00, 6 3150:00, 6 3160:00, 6 3170:00, 6 3180:00, 6 3190:00, 6 3200:00, 6 3210:00, 6 3220:00, 6 3230:00, 6 3240:00, 6 3250:00, 6 3260:00, 6 3270:00, 6 3280:00, 6 3290:00, 6 3300:00, 6 3310:00, 6 3320:00, 6 3330:00, 6 3340:00, 6 3350:00, 6 3360:00, 6 3370:00, 6 3380:00, 6 3390:00, 6 3400:00, 6 3410:00, 6 3420:00, 6 3430:00, 6 3440:00, 6 3450:00, 6 3460:00, 6 3470:00, 6 3480:00, 6 3490:00, 6 3500:00, 6 3510:00, 6 3520:00, 6 3530:00, 6 3540:00, 6 3550:00, 6 3560:00, 6 3570:00, 6 3580:00, 6 3590:00, 6 3600:00, 6 3610:00, 6 3620:00, 6 3630:00, 6 3640:00, 6 3650:00, 6 3660:00, 6 3670:00, 6 3680:00, 6 3690:00, 6 3700:00, 6 3710:00, 6 3720:00, 6 3730:00, 6 3740:00, 6 3750:00, 6 3760:00, 6 3770:00, 6 3780:00, 6 3790:00, 6 3800:00, 6 3810:00, 6 3820:00, 6 3830:00, 6 3840:00, 6 3850:00, 6 3860:00, 6 3870:00, 6 3880:00, 6 3890:00, 6 3900:00, 6 3910:00, 6 3920:00, 6 3930:00, 6 3940:00, 6 3950:00, 6 3960:00, 6 3970:00, 6 3980:00, 6 3990:00, 6 4000:00, 6 4010:00, 6 4020:00, 6 4030:00, 6 4040:00, 6 4050:00, 6 4060:00, 6 4070:00, 6 4080:00, 6 4090:00, 6 4100:00, 6 4110:00, 6 4120:00, 6 4130:00, 6 4140:00, 6 4150:00, 6 4160:00, 6 4170:00, 6 4180:00, 6 4190:00, 6 4200:00, 6 4210:00, 6 4220:00, 6 4230:00, 6 4240:00, 6 4250:00, 6 4260:00, 6 4270:00, 6 4280:00, 6 4290:00, 6 4300:00, 6 4310:00, 6 4320:00, 6 4330:00, 6 4340:00, 6 4350:00, 6 4360:00, 6 4370:00, 6 4380:00, 6 4390:00, 6 4400:00, 6 4410:00, 6 4420:00, 6 4430:00, 6 4440:00, 6 4450:00, 6 4460:00, 6 4470:00, 6 4480:00, 6 4490:00, 6 4500:00, 6 4510:00, 6 4520:00, 6 4530:00, 6 4540:00, 6 4550:00, 6 4560:00, 6 4570:00, 6 4580:00, 6 4590:00, 6 4600:00, 6 4610:00, 6 4620:00, 6 4630:00, 6 4640:00, 6 4650:00, 6 4660:00, 6 4670:00, 6 4680:00, 6 4690:00, 6 4700:00, 6 4710:00, 6 4720:00, 6 4730:00, 6 4740:00, 6 4750:00, 6 4760:00, 6 4770:00, 6 4780:00, 6 4790:00, 6 4800:00, 6 4810:00, 6 4820:00, 6 4830:00, 6 4840:00, 6 4850:00, 6 4860:00, 6 4870:00, 6 4880:00, 6 4890:00, 6 4900:00, 6 4910:00, 6 4920:00, 6 4930:00, 6 4940:00, 6 4950:00, 6 4960:00, 6 4970:00, 6 4980:00, 6 4990:00, 6 5000:00, 6 5010:00, 6 5020:00, 6 5030:00, 6 5040:00, 6 5050:00, 6 5060:00, 6 5070:00, 6 5080:00, 6 5090:00, 6 5100:00, 6 5110:00, 6 5120:00, 6 5130:00, 6 5140:00, 6 5150:00, 6 5160:00, 6 5170:00, 6 5180:00, 6 5190:00, 6 5200:00, 6 5210:00, 6 5220:00, 6 5230:00, 6 5240:00, 6 5250:00, 6 5260:00, 6 5270:00, 6 5280:00, 6 5290:00, 6 5300:00, 6 5310:00, 6 5320:00, 6 5330:00, 6 5340:00, 6 5350:00, 6 5360:00, 6 5370:00, 6 5380:00, 6 5390:00, 6 5400:00, 6 5410:00, 6 5420:00, 6 5430:00, 6 5440:00, 6 5450:00, 6 5460:00, 6 5470:00, 6 5480:00, 6 5490:00, 6 5500:00, 6 5510:00, 6 5520:00, 6 5530:00, 6 5540:00, 6 5550:00, 6 5560:00, 6 5570:00, 6 5580:00, 6 5590:00, 6 5600:00, 6 5610:00, 6 5620:00, 6 5630:00, 6 5640:00, 6 5650:00, 6 5660:00, 6 5670:00, 6 5680:00, 6 5690:00, 6 5700:00, 6 5710:00, 6 5720:00, 6 5730:00, 6 5740:00, 6 5750:00, 6 5760:00, 6 5770:00, 6 5780:00, 6 5790:00, 6 5800:00, 6 5810:00, 6 5820:00, 6 5830:00, 6 5840:00, 6 5850:00, 6 5860:00, 6 5870:00, 6 5880:00, 6 5890:00, 6 5900:00, 6 5910:00, 6 5920:00, 6 5930:00, 6 5940:00, 6 5950:00, 6 5960:00, 6 5970:00, 6 5980:00, 6 5990:00, 6 6000:00, 6 6010:00, 6 6020:00, 6 6030:00, 6 6040:00, 6 6050:00, 6 6060:00, 6 6070:00, 6 6080:00, 6 6090:00, 6 6100:00, 6 6110:00, 6 6120:00, 6 6130:00, 6 6140:00, 6 6150:00, 6 6160:00, 6 6170:00, 6 6180:00, 6 6190:00, 6 6200:00, 6 6210:00, 6 6220:00, 6 6230:00, 6 6240:00, 6 6250:00, 6 6260:00, 6 6270:00, 6 6280:00, 6 6290:00, 6 6300:00, 6 6310:00, 6 6320:00, 6 6330:00, 6 6340:00, 6 6350:00, 6 6360:00, 6 6370:00, 6 6380:00, 6 6390:00, 6 6400:00, 6 6410:00, 6 6420:00, 6 6430:00, 6 6440:00, 6 6450:00, 6 6460:00, 6 6470:00, 6 6480:00, 6 6490:00, 6 6500:00, 6 6510:00, 6 6520:00, 6 6530:00, 6 6540:00, 6 6550:00, 6 6560:00, 6 6570:00, 6 6580:00, 6 6590:00, 6 6600:00, 6 6610:00, 6 6620:00, 6 6630:00, 6 6640:00, 6 6650:00, 6 6660:00, 6 6670:00, 6 6680:00, 6 6690:00, 6 6700:00, 6 6710:00, 6 6720:00, 6 6730:00, 6 6740:00, 6 6750:00, 6 6760:00, 6 6770:00, 6 6780:00, 6 6790:00, 6 6800:00, 6 6810:00, 6 6820:00, 6 6830:00, 6 6840:00, 6 6850:00, 6 6860:00, 6 6870:00, 6 6880:00, 6 6890:00, 6 6900:00, 6 6910:00, 6 6920:00, 6 6930:00, 6 6940:00, 6 6950:00, 6 6960:00, 6 6970:00, 6 6980:00, 6 6990:00, 6 7000:00, 6 7010:00, 6 7020:00, 6 7030:00, 6 7040:00, 6 7050:00, 6 7060:00, 6 7070:00, 6 7080:00, 6 7090:00, 6 7100:00, 6 7110:00, 6 7120:00, 6 7130:00, 6 7140:00, 6 7150:00, 6 7160:00, 6 7170:00, 6 7180:00, 6 7190:00, 6 7200:00, 6 7210:00, 6 7220:00, 6 7230:00, 6 7240:00, 6 7250:00, 6 7260:00, 6 7270:00, 6 7280:00, 6 7290:00, 6 7300:00, 6 7310:00, 6 7320:00, 6 7330:00, 6 7340:00, 6 7350:00, 6 7360:00, 6 7370:00, 6 7380:00, 6 7390:00, 6 7400:00, 6 7410:00, 6 7420:00, 6 7430:00, 6 7440:00, 6 7450:00, 6 7460:00, 6 7470:00, 6 7480:00, 6 7490:00, 6 7500:00, 6 7510:00, 6 7520:00, 6 7530:00, 6 7540:00, 6 7550:00, 6 7560:00, 6 7570:00, 6 7580:00, 6 7590:00, 6 7600:00, 6 7610:00, 6 7620:00, 6 7630:00, 6 7640:00, 6 7650:00, 6 7660:00, 6 7670:00, 6 7680:00, 6 7690:00, 6 7700:00, 6 7710:00, 6 7720:00, 6 7730:00, 6 7740:00, 6 7750:00, 6 7760:00, 6 7770:00, 6 7780:00, 6 7790:00, 6 7800:00, 6 7810:00, 6 7820:00, 6 7830:00, 6 7840:00, 6 7850:00, 6 7860:00, 6 7870:00, 6 7880:00, 6 7890:00, 6 7900:00, 6 7910:00, 6 7920:00, 6 7930:00, 6 7940:00, 6 7950:00, 6 7960:00, 6 7970:00, 6 7980:00, 6 7990:00, 6 8000:00, 6 8010:00, 6 8020:00, 6 8030:00, 6 8040:00, 6 8050:00, 6 8060:00, 6 8070:00, 6 8080:00, 6 8090:00, 6 8100:00, 6 8110:00, 6 8120:00, 6 8130:00, 6 8140:00, 6 8150:00, 6 8160:00, 6 8170:00, 6 8180:00, 6 8190:00, 6 8200:00, 6 8210:00, 6 8220:00, 6 8230:00, 6 8240:00, 6 8250:00, 6 8260:00, 6 8270:00, 6 8280:00, 6 8290:00, 6 8300:00, 6 8310:00, 6 8320:00, 6 8330:00, 6 8340:00, 6 8350:00, 6 8360:00, 6 8370:00, 6 8380:00, 6 8390:00, 6 8400:00, 6 8410:00, 6 8420:00, 6 8430:00, 6 8440:00, 6 8450:00, 6 8460:00, 6 8470:00, 6 8480:00, 6 8490:00, 6 8500:00, 6 8510:00, 6 8520:00, 6 8530:00, 6 8540:00, 6 8550:00, 6 8560:00, 6 8570:00, 6 8580:00, 6 8590:00, 6 8600:00, 6 8610:00, 6 8620:00, 6 8630:00, 6 8640:00, 6 8650:00, 6 8660:00, 6 8670:00, 6 8680:00, 6 8690:00, 6 8700:00, 6 8710:00, 6 8720:00, 6 8730:00, 6 8740:00, 6 8750:00, 6 8760:00, 6 8770:00, 6 8780:00, 6 8790:00, 6 8800:00, 6 8810:00, 6 8820:00, 6 8830:00, 6 8840:00, 6 8850:00, 6 8860:00, 6 8870:00, 6 8880:00, 6 8890:00, 6 8900:00, 6 8910:00, 6 8920:00, 6 8930:00, 6 8940:00, 6 8950:00, 6 8960:00, 6 8970:00, 6 8980:00, 6 8990:00, 6 9000:00, 6 9010:00, 6 9020:00, 6 9030:00, 6 9040:00, 6 9050:00, 6 9060:00, 6 9070:00, 6 9080:00, 6 9090:00, 6 9100:00, 6 9110:00, 6 9120:00, 6 9130:00, 6 9140:00, 6 9150:00, 6 9160:00, 6 9170:00, 6 9180:00, 6 9190:00, 6 9200:00, 6 9210:00, 6 9220:00, 6 9230:00, 6 9240:00, 6 9250:00, 6 9260:00, 6 9270:00, 6 9280:00, 6 9290:00, 6 9300:00, 6 9310:00, 6 9320:00, 6 9330:00, 6 9340:00, 6 9350:00, 6 9360:00, 6 9370:00, 6 9380:00, 6 9390:00, 6 9400:00, 6 9410:00, 6 9420:00, 6 9430:00, 6 9440:00, 6 9450:00, 6 9460:00, 6 9470:00, 6 9480:00, 6 9490:00, 6 9500:00, 6 9510:00, 6 9520:00, 6 9530:00, 6 9540:00, 6 9550:00, 6 9560:00, 6 9570:00, 6 9580:00, 6 9590:00, 6 9600:00, 6 9610:00, 6 9620:00, 6 9630:00, 6 9640:00, 6 9650:00, 6 9660:00, 6 9670:00, 6 9680:00, 6 9690:00, 6 9700:00, 6 9710:00, 6 9720:00, 6 9730:00, 6 9740:00, 6 9750:00, 6 9760:00, 6 9770:00, 6 9780:00, 6 9790:00, 6 9800:00, 6 9810:00, 6 9820:00, 6 9830:00, 6 9840:00, 6 9850:00, 6 9860:00, 6 9870:00, 6 9880:00, 6 9890:00, 6 9900:00, 6 9910:00, 6 9920:00, 6 9930:00, 6 9940:00, 6 9950:00, 6 9960:00, 6 9970:00, 6 9980:00, 6 9990:00, 6 1000:00, 6 1001:00, 6 1002:00, 6 1003:00, 6 1004:00, 6 1005:00, 6 1006:00, 6 1007:00, 6 1008:00, 6 1009:00, 6 1010:00, 6 1011:00, 6 1012:00, 6 1013:00, 6 1014:00, 6 1015:00, 6 1016:00, 6 1017:00, 6 1018:00, 6 1019:00, 6 1020:00, 6 1021:00, 6 1022:00, 6 1023:00, 6 1024:00, 6 1025:00, 6 1026:00, 6 1027:00, 6 1028:00, 6 1029:00, 6 1030:00, 6 1031:00, 6 1032:00, 6 1033:00, 6 1034:00, 6 1035:00, 6 1036:00, 6 1037:00, 6 1038:00, 6 1039:00, 6 1040:00, 6 1041:00, 6 1042:00, 6 1043:00, 6 1044:00, 6 1045:00, 6 1046:00, 6 1047:00, 6 1048:00, 6 1049:00, 6 1050:00, 6 1051:00, 6 1052:00, 6 1053:00, 6 1054:00, 6 1055:00, 6 1056:00, 6 1057:00, 6 1058:00, 6 1059:00, 6 1060:00, 6 1061:00, 6 1062:00, 6 1063:00, 6 1064:00, 6 1065:00, 6 1066:00, 6 1067:00, 6 1068:00, 6 1069:00, 6 1070:00, 6 1071:00, 6 1072:00, 6 1073:00, 6 1074:00, 6 1075:00, 6 1076:00, 6 1077:00, 6 1078:00, 6 1079:00, 6 1080:00, 6 1081:00, 6 1082:00, 6 1083:00, 6 1084:00, 6 1085:00, 6 1086:00, 6 1087:00, 6 1088:00, 6 1089:00, 6 1090:00, 6 1091:00, 6 1092:00, 6 1093:00, 6 1094:00, 6 1095:00, 6 1096:00, 6 1097:00, 6 1098:00, 6 1099:00, 6 1100:00, 6 1101:00, 6 1102:00, 6 1103:00, 6 1104:00, 6 1105:00, 6 1106:00, 6 1107:00, 6 1108:00, 6 1109:00, 6 1110:00, 6 1111:00, 6 1112:00, 6 1113:00, 6 1114:00, 6 1115:00, 6 1116:00, 6 1117:00, 6 1118:00, 6 1119:00, 6 1120:00, 6 1121:00, 6 1122:00, 6 1123:00, 6 1124:00, 6 1125:00, 6 1126:00, 6 1127:00, 6 1128:00, 6 1129:00, 6 1130:00, 6 1131:00, 6 1132:00, 6 1133:00, 6 1134:00, 6 1135:00, 6 1136:00, 6 1137:00, 6 1138:00, 6 1139:00, 6 1140:00, 6 1141:00, 6 1142:00, 6 1143:00, 6 1144:00, 6 1145:00, 6 1146:00, 6 1147:00, 6 1148:00, 6 1149:00, 6 1150:00, 6 1151:00, 6 1152:00, 6 1153:00, 6 1154:00, 6 1155:00, 6 1156:00, 6 1157:00, 6 1158:00, 6 1159:00, 6 1160:00, 6 1161:00, 6 1162:00, 6 1163:00, 6 1164:00, 6 1165:00, 6 1166:00, 6 1167:00, 6 1168:00, 6 1169:00, 6 1170:00, 6 1171:00, 6 1172:00, 6 1173:00, 6 1174:00, 6 1175:00, 6 1176:00, 6 1177:00, 6 1178:00, 6 1179:00, 6 1180:00, 6 1181:00, 6 1182:00, 6 1183:00, 6 1184:00, 6 1185:00, 6 1186:00, 6 1187:00, 6 1188:00, 6 1189:00, 6 1190:00, 6 1191:00, 6 1192:00, 6 1193:00, 6 1194:00, 6 1195:00, 6 1196:00, 6 1197:00, 6 1198:00, 6 1199:00, 6 1200:00, 6 1201:00, 6 1202:00, 6 1203:00, 6 1204:00, 6 1205:00, 6 1206:00, 6 1207:00, 6 1208:00, 6 1209:00, 6 1210:00, 6 1211:00, 6 1212:00, 6 1213:00, 6 1214:00, 6 1215:00, 6 1216:00, 6 1217:00, 6 1218:00, 6 1219:00, 6 1220:00, 6 1221:00, 6 1222:00, 6 1223:00, 6 1224:00, 6 1225:00, 6 1226:00, 6 1227:00, 6 1228:00, 6 1229:00, 6 1230:00, 6 1231:00, 6 1232:00, 6 1233:00, 6 1234:00, 6 1235:00, 6 1236:00, 6 1237:00, 6 1238:00, 6 1239:00, 6 1240:00, 6 1241:00, 6 1242:00, 6 1243:00, 6 1244:00, 6 1245:00, 6 1246:00, 6 1247:00, 6 1248:00, 6 1249:00, 6 1250:00, 6 1251:00, 6 1252:00, 6 1253:00, 6 1254:00, 6 1255:00, 6 1256:00, 6 1257:00, 6 1258:00, 6 1259:00, 6 1260:00, 6 1261:00, 6 1262:00, 6 1263:00, 6 1264:00, 6 1265:00, 6 1266:00, 6 1267:00, 6 1268:00, 6 1269:00, 6 1270:00, 6 1271:00, 6 1272:00, 6 1273:00, 6 1274:00, 6 1275:00, 6 1276:00, 6 1277:00, 6 1278:00, 6 1279:00, 6 1280:00, 6 1281:00, 6 1282:00, 6 1283:00, 6 1284:00, 6 1285:00, 6 1286:00, 6 1287:00, 6 1288:00, 6 1289:00, 6 1290:00, 6 1291:00, 6 1292:00, 6 1293:00, 6 1294:00, 6 1295:00, 6 1296:00, 6 1297:00, 6 1298:00, 6 1299:00, 6 1300:00, 6 1301:00, 6 1302:00, 6 1303:00, 6 1304:00, 6 1305:00, 6 1306:00, 6 1307:00, 6 1308:00, 6 1309:00, 6 1310:00, 6 1311:00, 6 1312:00, 6 1313:00, 6 1314:00, 6 1315:00, 6 1316:00, 6 1317:00, 6 1318:00, 6 1319:00, 6 1320:00, 6 1321:00, 6 1322:00, 6 1323:00, 6 1324:00, 6 1325:00, 6 1326:00, 6 1327:00, 6 1328:00, 6 1329:00, 6 1330:00, 6 1331:00, 6 1332:00, 6 1333:00, 6 1334:00, 6 1335:00, 6 1336:00, 6 1337:00, 6 1338:00, 6 1339:00, 6 1340:00, 6 1341:00, 6 1342:00, 6 1343:00, 6 1344:00, 6 1345:00, 6 1346:00, 6 1347:00, 6 1348:00, 6 1349:00, 6 1350:00, 6 1351:00, 6 1352:00, 6 1353:00, 6 1354:00, 6 1355:00, 6 1356:00, 6 1357:00, 6 1358:00, 6 1359:00, 6 1360:00, 6 1361:00, 6 1362:00, 6 1363:00, 6 1364:00, 6 1365:00, 6 1366:00, 6 1367:00, 6 1368:00, 6 1369:00, 6 1370:00, 6 1371:00, 6 1372:00, 6 1373:00, 6 1374:00, 6 1375:00, 6 1376:00, 6 1377:00, 6 1378:00, 6 1379:00, 6 1380:00, 6 1381:00, 6 1382:00, 6 1383:00, 6 1384:00, 6 1385:00, 6 1386:00, 6 1387
```

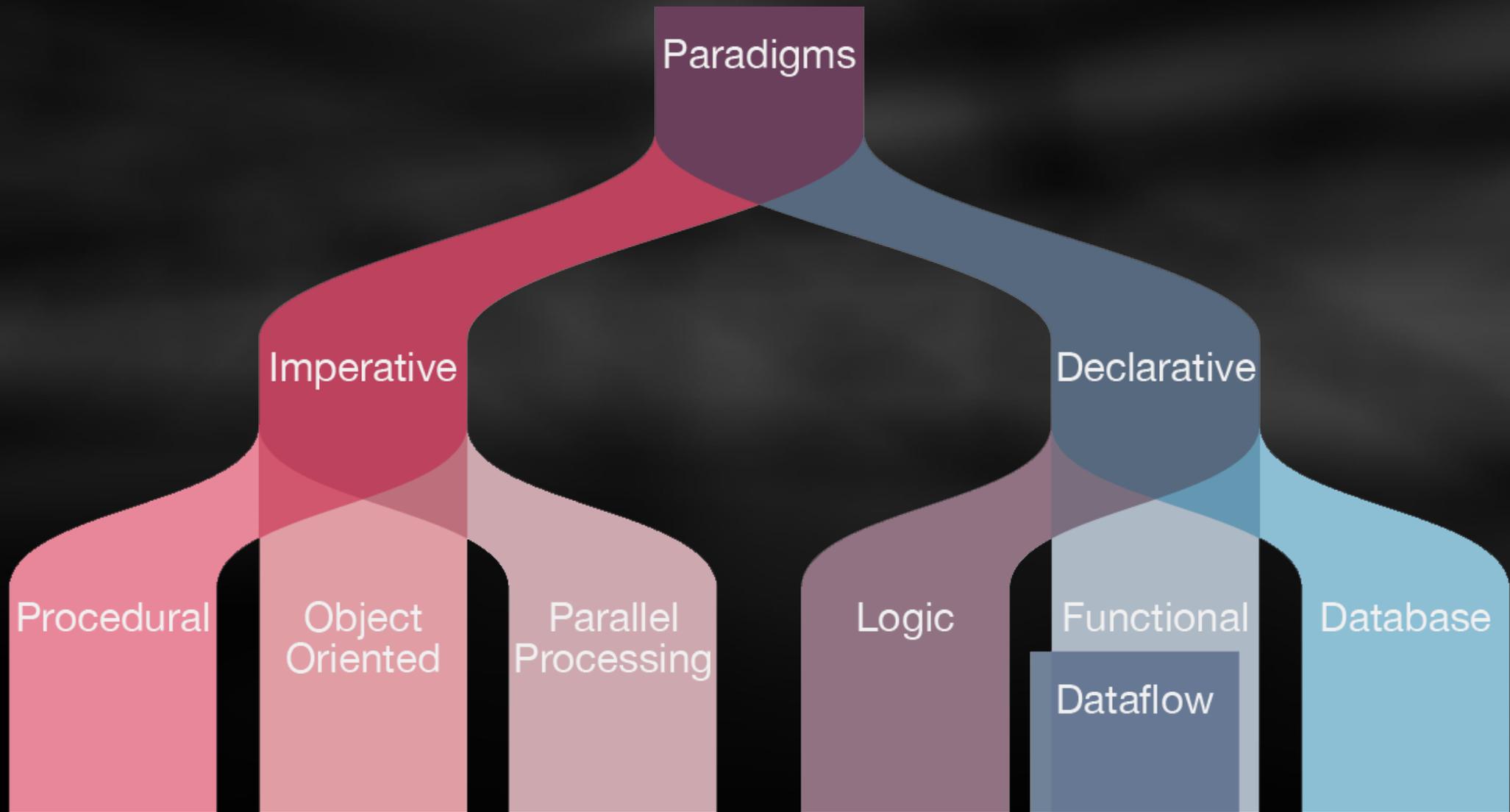
The principal programming paradigms

"More is not better (or worse) than less, just different."

v1.08 © 2008 by Peter Van Roy







编程范式	描述	主要的特性	相关的编程语言
Imperative 命令式	使用流程化的语句和过程直接控制程序的运行和数据状态。	直接赋值 常用的数据结构 全局变量、局部变量 Goto语句 顺序化数据的操作和迭代 以功能为主的模块化	C, C++, Java, PHP, Python, Ruby
Functional 函数式	通过数学函数表达式的方式来避免状态和可变的数据。	代码公式化 Lambda 表达式 函数的包装和嵌套（高阶函数、Pipeline、Currying、Map/Reduce/Filter） 递归（尾递归） 无数据共享或依赖 无副作用（并行、重构、组合）	C++, Clojure, Coffeescript, Elixir, Erlang, F#, Haskell, Lisp, Python, Ruby, Scala, SequenceL, SML
Object-Oriented 面向对象	把一组字段和作用在其上面的方法抽象成一个个对象。	对象封装 消息传递 隐藏细节 数据和接口抽象 多态 继承 对象的序列化和反序列	Common Lisp, C++, C#, Eiffel, Java, PHP, Python, Ruby, Scala
Declarative 声明式	定义计算的逻辑而不是定义具体的流程控制。	4GLs, spreadsheets, report program generators	SQL, 正规表达式, CSS, Prolog, OWL, SPARQL



谢谢