代码编程中的编程范式

陈皓

个人简介

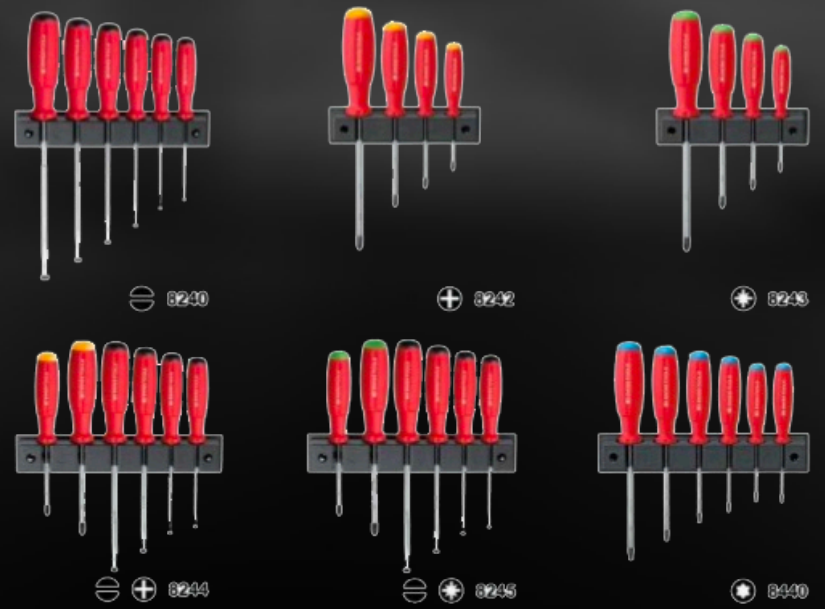
- **18年工作经历，超大型分布式系统基础架构研发和设计**
- **擅长领域：金融、电子商务、云计算、大数据**
- **职业背景**
  - 阿里巴巴资深架构师（阿里云、天猫、淘宝）
  - 亚马逊高级研发经理（AWS、全球购、商品需求预测）
  - 汤森路透研发经理（实时金融数据处理基础架构）
- **目前在创业，致力于为企业提供技术架构管理产品**
  - 目标：用户不用改一行代码就可以提高系统的性能和稳定性

从C语言开始说起

C 语言的 swap 函数

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

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

```c
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 \
   { chartemp[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，可能会造成很能多问题
- 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

```c
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。
  - 拷贝函数函数、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!*

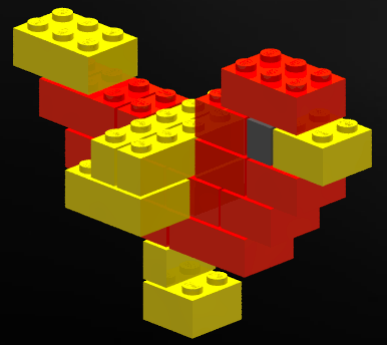
"Church encodings"

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 funciton，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

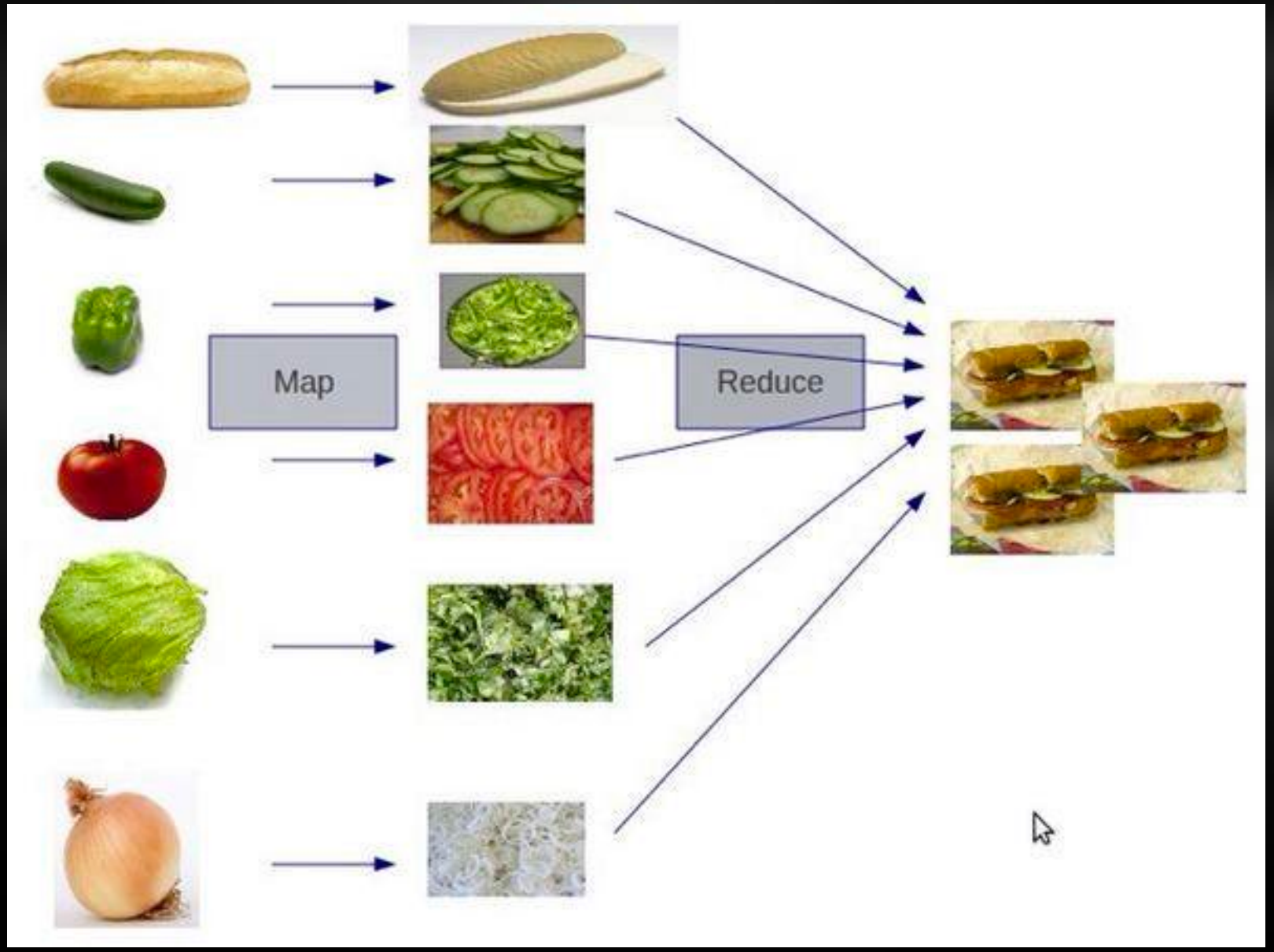
```scheme
(define (plus x y)  (+ x y))
(define (times x y) (* x y))
(define (square x)  (times x x))

(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)))))
```

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

(newline)
(display(factoral 6))


;;; another version of recursion
(define (factoral_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

```python
# 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"])
```

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


Map
Reduce

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 auwwx | grep chenhao | 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(auwwx))))

# 也可以如下所示
pids = for_each(result, [ps_auwwx, grep_chenhao, 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

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

```python
@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

hello world

Python 修饰器 + 管道

```python
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()
```

```python
@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
```

```python
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。

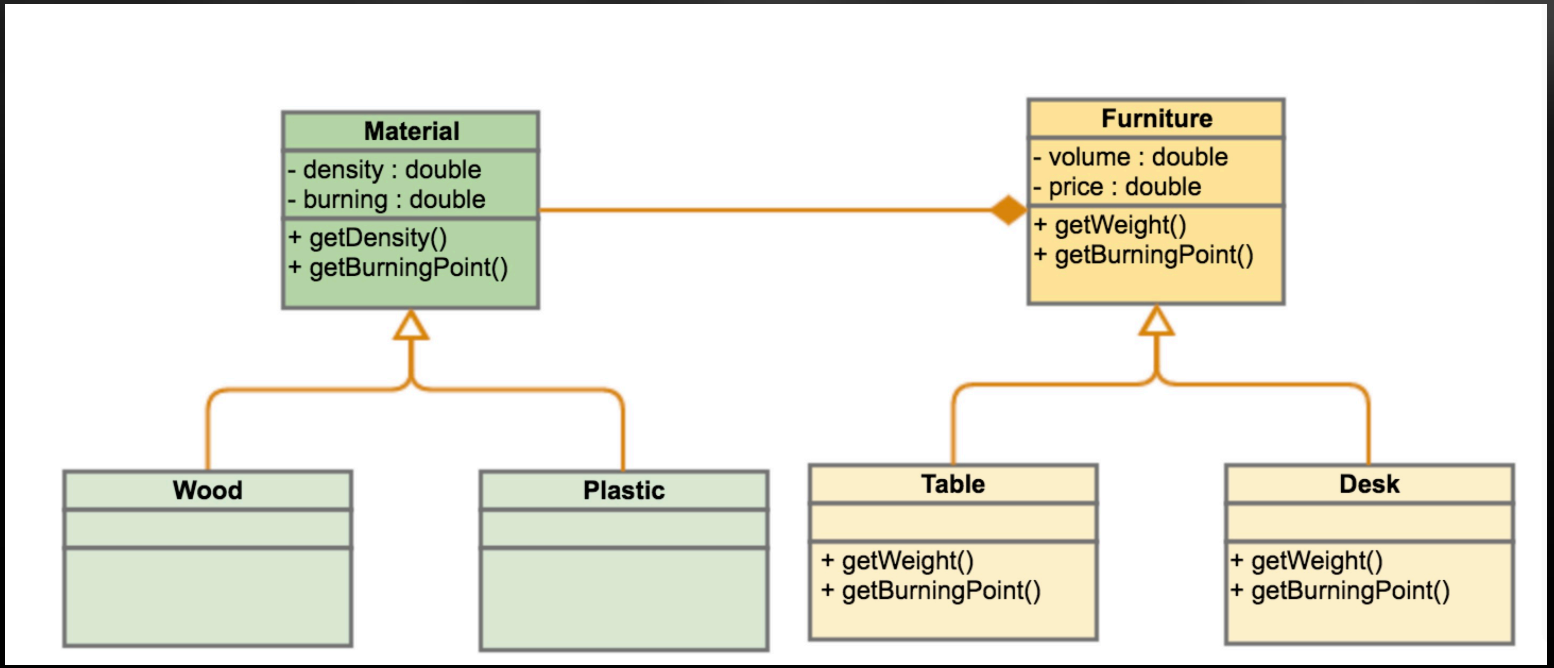
面向对象算法拼装

```
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，从一侧面上说，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这些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

```go
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)
}
```

```go
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

```go
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)
}
```

```go
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)
}
```

```go
type Painter interface {
    Paint()
}

type Clicker interface {
    Click()
}
```

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

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

```go
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]
}
```

```go
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)
    }
}
```

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

```go
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
}
```

```go
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

```go
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

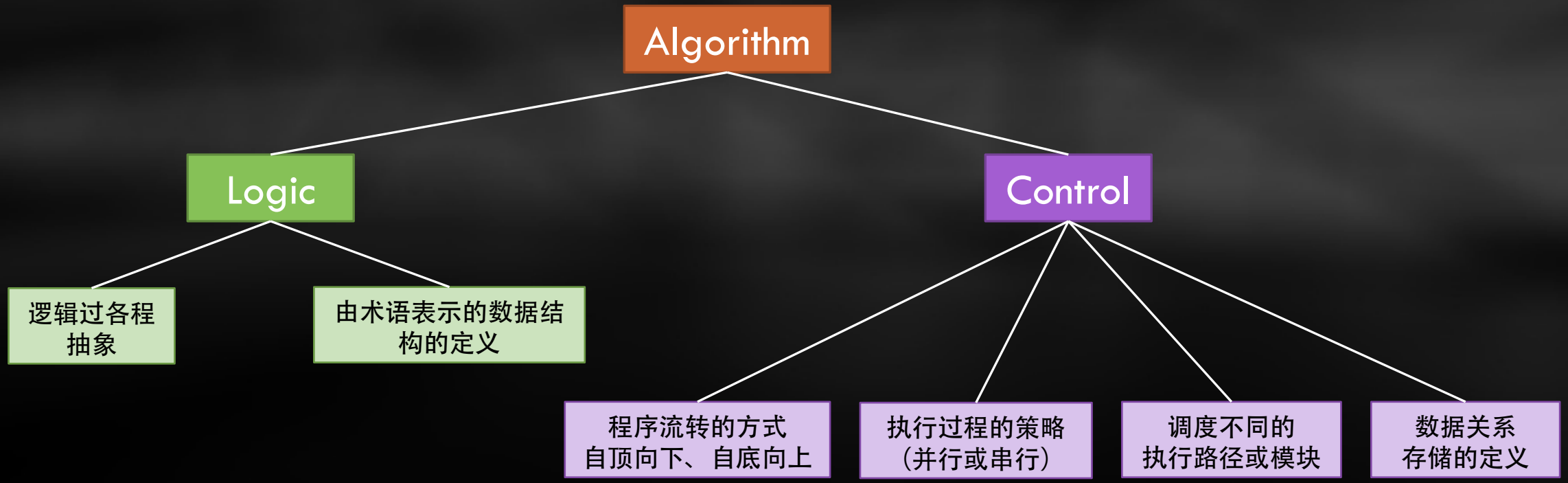
- Pascal语言之父Niklaus Wirth在70年代提出：
  - Program = Data Structure + Algorithm
- 随后逻辑学家和计算机科学家R Kowalski进一步提出：
  - Algorithm = Logic + Control
- **程序的复杂性有两个：**
  - 一个是代码需要处理的逻辑，也就是我们说的业务逻辑
  - 另一个是是控制，控制或组织代码完成复杂的逻辑

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

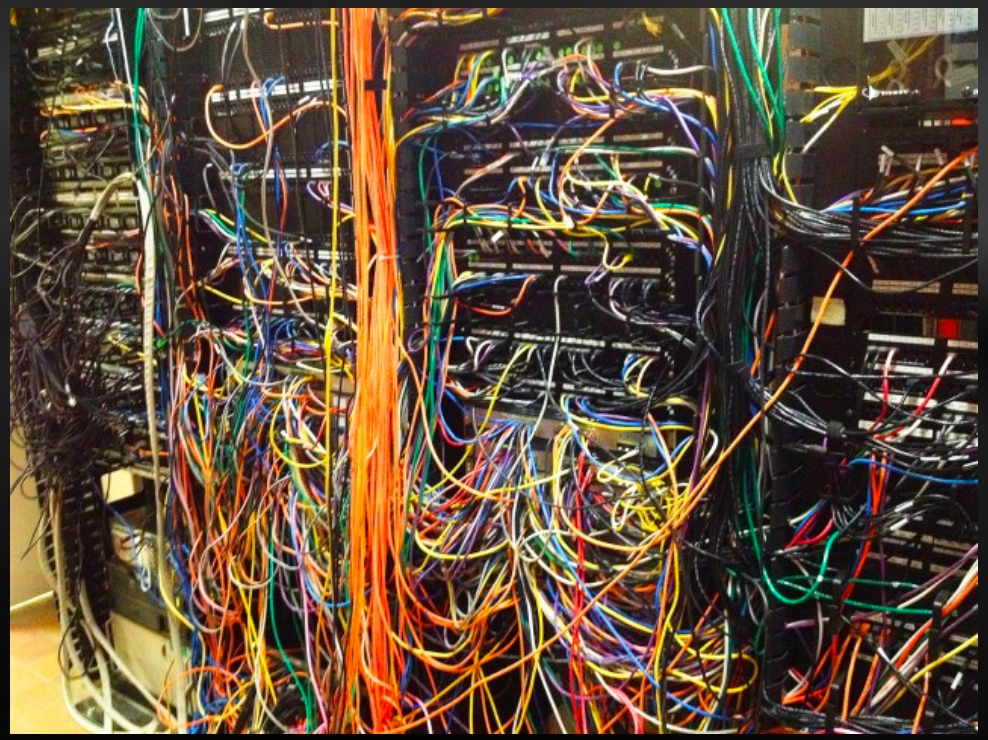


- Logic 部分才是真正有意义的 - What
- Control部分只是影响Logic部分的效率 - How

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.length) {
        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

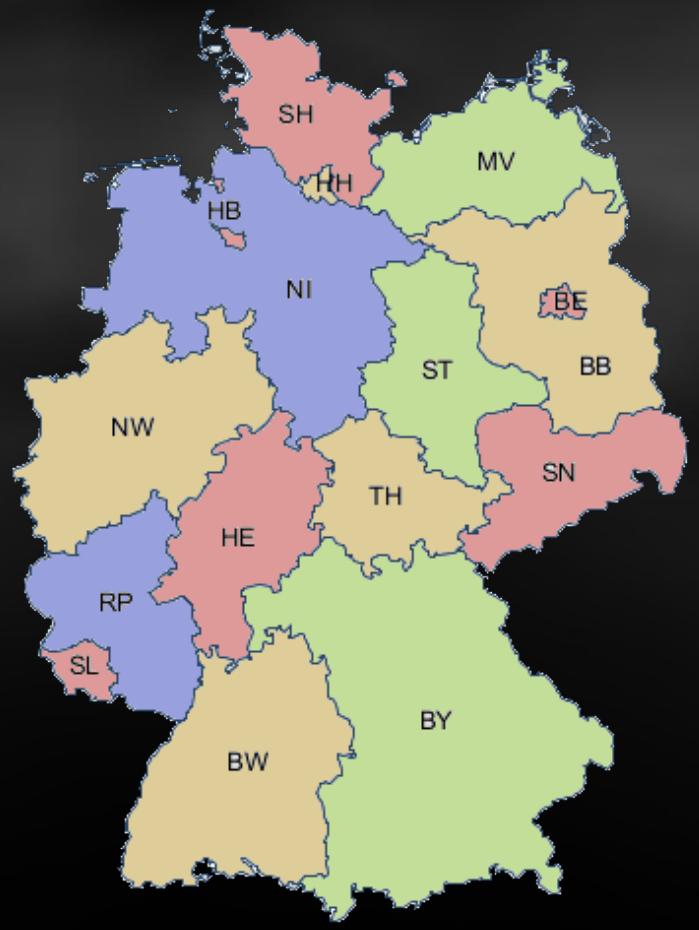
逻辑编程范式

- 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.
```

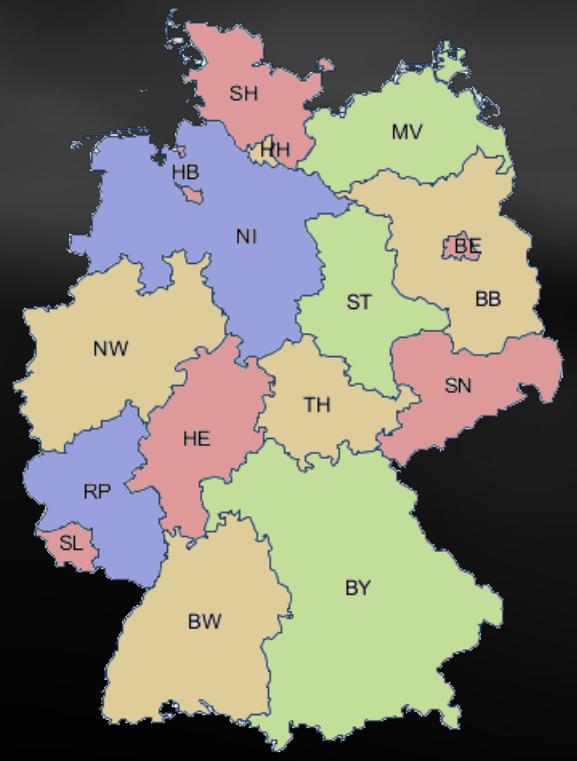
地图四色问题 - Prolog

```
/* 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).
```



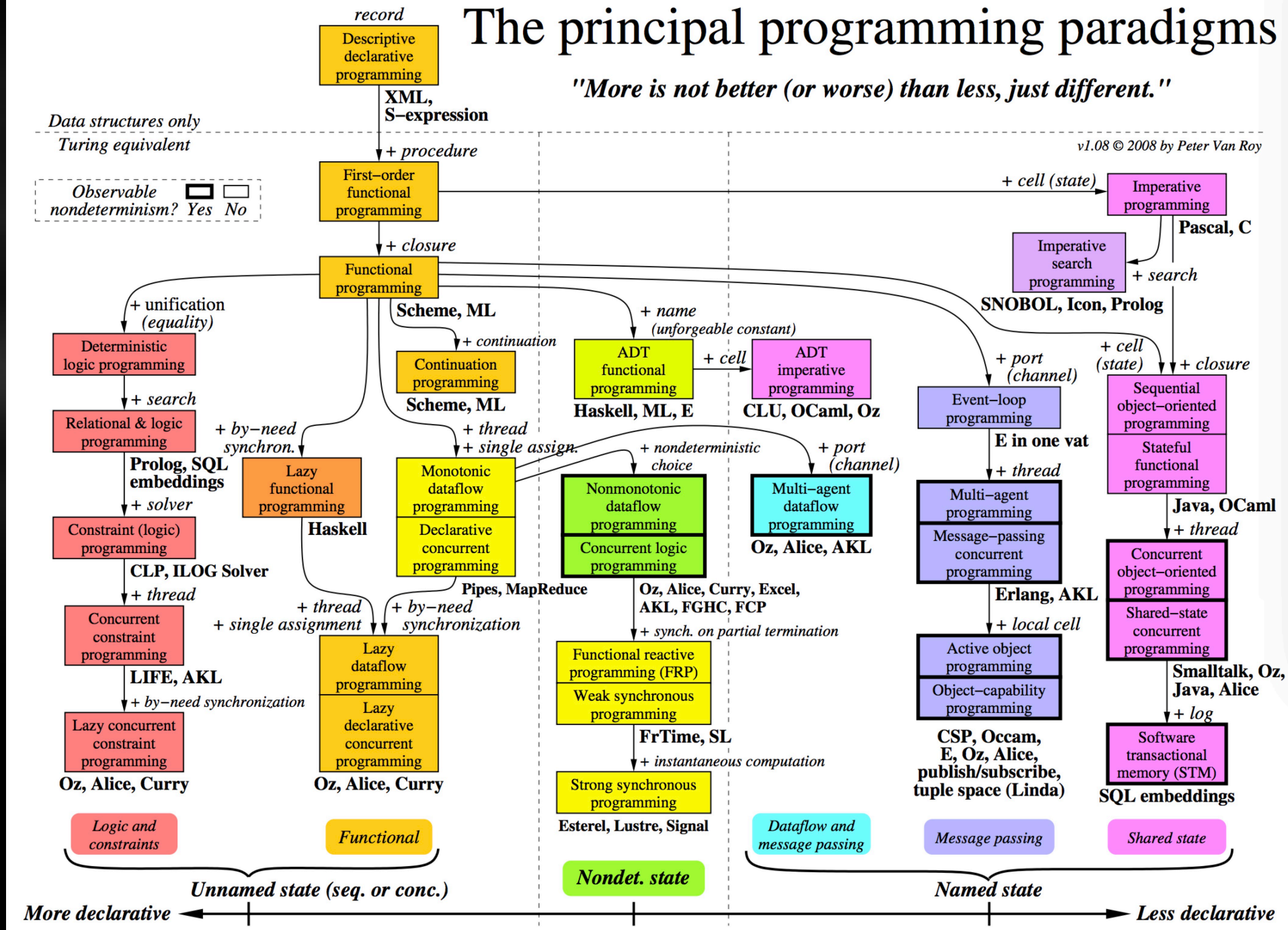
地图四色问题 - Prolog

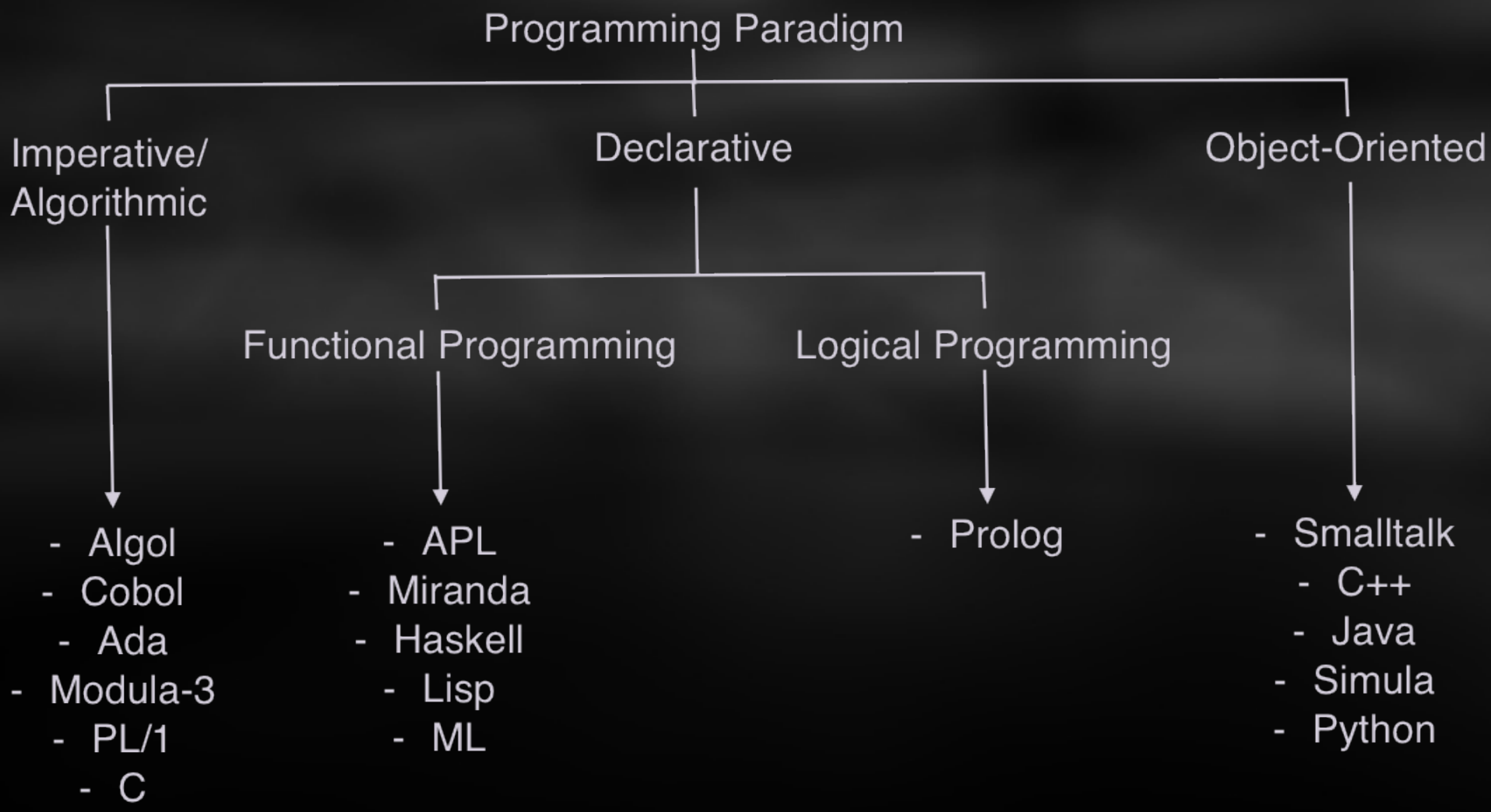
```
/* 定义整个地图的相邻关系 */
germany(SH, MV, HH, HB, NI, ST, BE, BB, SN, NW, HE, TH, RP, SL, BW, BY) :-
  neighbor(SH, NI), neighbor(SH, HH), neighbor(SH, MV),
  neighbor(HH, NI),
  neighbor(MV, NI), neighbor(MV, BB),
  neighbor(NI, HB), neighbor(NI, BB), neighbor(NI, ST), neighbor(NI, TH),
  neighbor(NI, HE),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

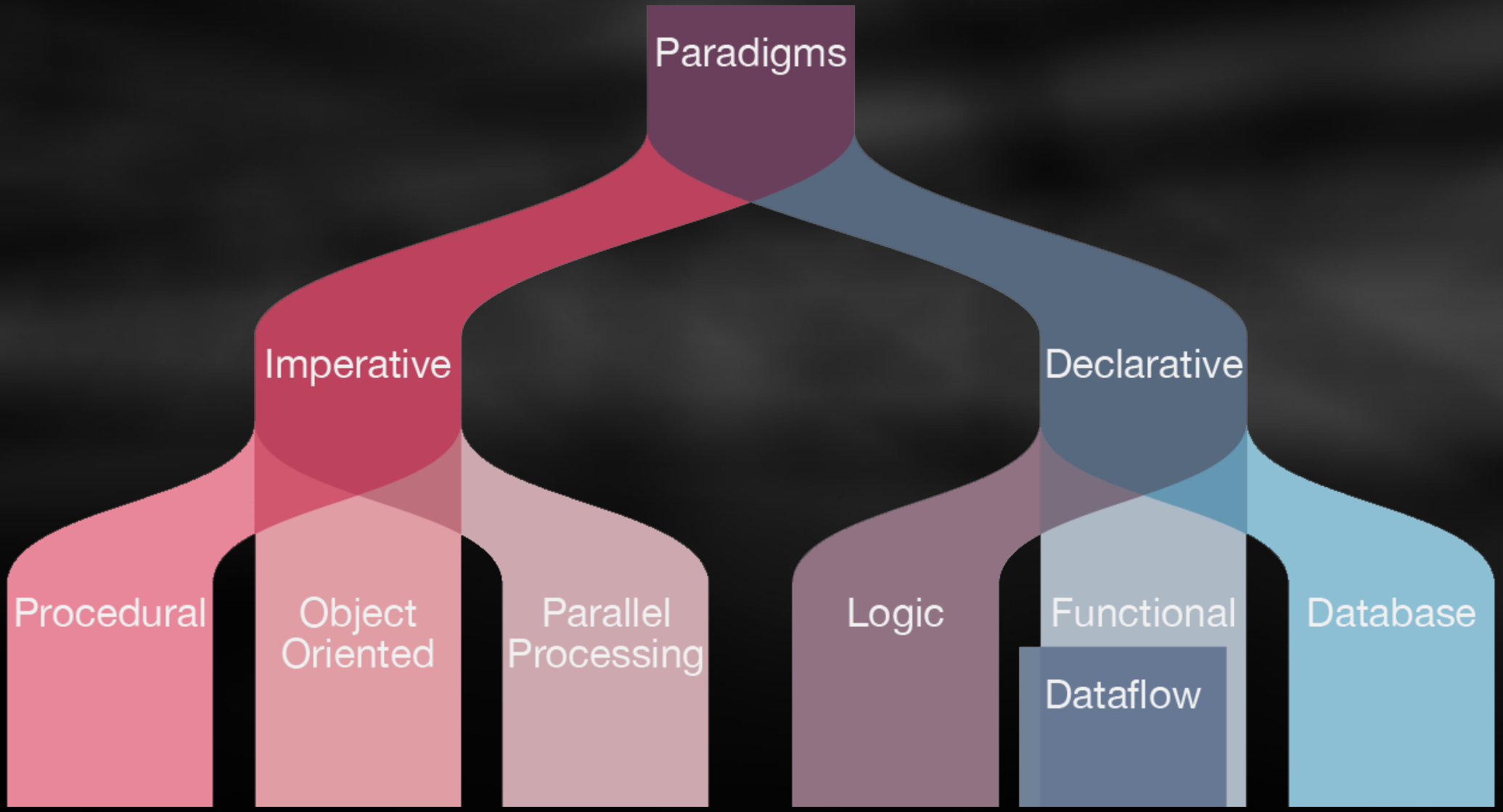


程序世界里的编程范式


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
Right
brain
I am the right brain.
I am creativity. A free spirit. I am passion.
Yearning. Sensuality. I am the sound of roaring laughter.
I am taste. The feeling of sand beneath bare feet.
I am movement. Vivid colors.
I am the urge to paint on an empty canvas.
I am boundless imagination. Art. Poetry. I sense. I feel.
I am everything I wanted to be.
直觉型
想像力
非线性
宏观思维
抽象化的


The principal programming paradigms
"More is not better (or worse) than less, just different."
v1.08 © 2008 by Peter Van Roy
Data structures only
Turing equivalent
Observable nondeterminism? Yes No
record
Descriptive declarative programming
XML, S–expression
+ procedure
First–order functional programming
+ closure
Functional programming
+ unification (equality)
Deterministic logic programming
+ search
Relational & logic programming
Prolog, SQL embeddings
+ solver
Constraint (logic) programming
CLP, ILOG Solver
+ thread
Concurrent constraint programming
LIFE, AKL
+ by–need synchronization
Lazy concurrent constraint programming
Oz, Alice, Curry
Scheme, ML
+ continuation
Continuation programming
Scheme, ML
+ by–need synchron.
Lazy functional programming
Haskell
+ thread
+ single assign.
Monotonic dataflow programming
Declarative concurrent programming
Pipes, MapReduce
+ thread
+ single assignment
+ by–need synchronization
Lazy dataflow programming
Lazy declarative concurrent programming
Oz, Alice, Curry
+ name (unforgeable constant)
ADT functional programming
Haskell, ML, E
+ cell
ADT imperative programming
CLU, OCaml, Oz
+ nondeterministic choice
Nonmonotonic dataflow programming
Concurrent logic programming
Oz, Alice, Curry, Excel, AKL, FGHC, FCP
+ synch. on partial termination
Functional reactive programming (FRP)
Weak synchronous programming
FrTime, SL
+ instantaneous computation
Strong synchronous programming
Esterel, Lustre, Signal
+ port (channel)
Multi–agent dataflow programming
Oz, Alice, AKL
+ cell (state)
Imperative programming
Pascal, C
+ search
Imperative search programming
SNOBOL, Icon, Prolog
+ port (channel)
Event–loop programming
E in one vat
+ thread
Multi–agent programming
Message–passing concurrent programming
Erlang, AKL
+ local cell
Active object programming
Object–capability programming
CSP, Occam, E, Oz, Alice, publish/subscribe, tuple space (Linda)
+ cell (state)
+ closure
Sequential object–oriented programming
Stateful functional programming
Java, OCaml
+ thread
Concurrent object–oriented programming
Shared–state concurrent programming
Smalltalk, Oz, Java, Alice
+ log
Software transactional memory (STM)
SQL embeddings
Logic and constraints
Functional
Dataflow and message passing
Message passing
Shared state
Unnamed state (seq. or conc.)
Nondet. state
Named state
More declarative
Less declarative


Programming Paradigm
Imperative/
Algorithmic
Declarative
Object-Oriented
Functional Programming
Logical Programming
- Algol
- Cobol
- Ada
- Modula-3
- PL/1
- C
- APL
- Miranda
- Haskell
- Lisp
- ML
- Prolog
- Smalltalk
- C++
- Java
- Simula
- Python


Paradigms
Imperative
Declarative
Procedural
Object Oriented
Parallel Processing
Logic
Functional
Dataflow
Database

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