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

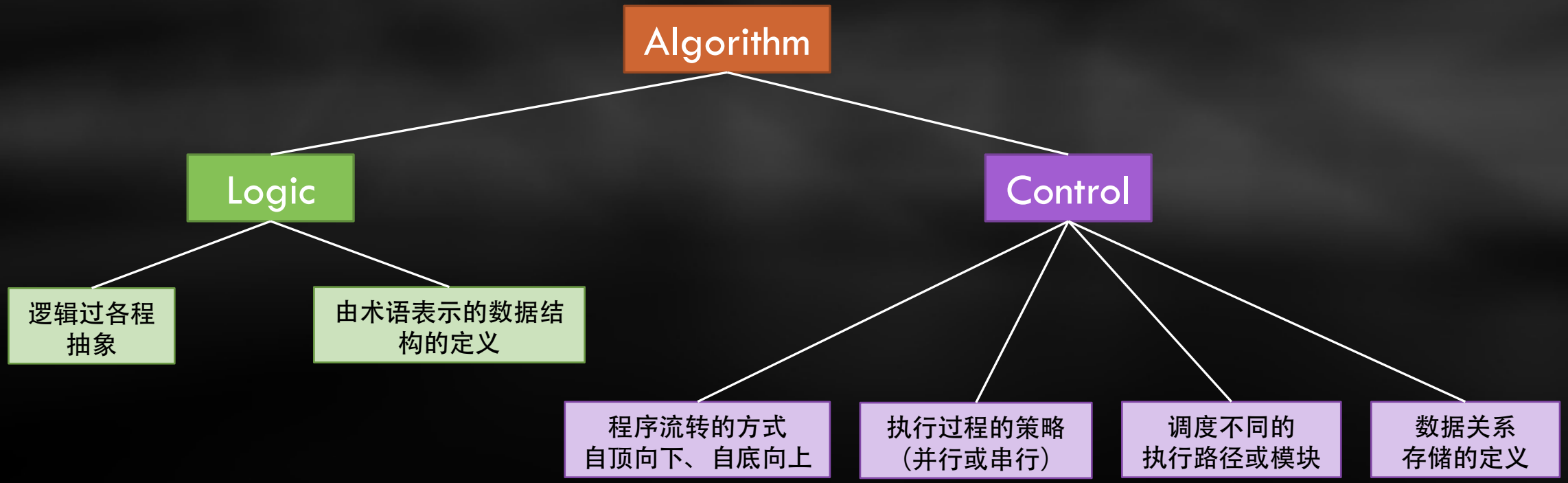
- Pascal语言之父Niklaus Wirth在70年代提出：
  - Program = Data Structure + Algorithm
- 随后逻辑学家和计算机科学家R Kowalski进一步提出：
  - Algorithm = Logic + Control
- **程序的复杂性有两个：**
  - 一个是代码需要处理的逻辑，也就是我们说的业务逻辑
  - 另一个是是控制，控制或组织代码完成复杂的逻辑

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

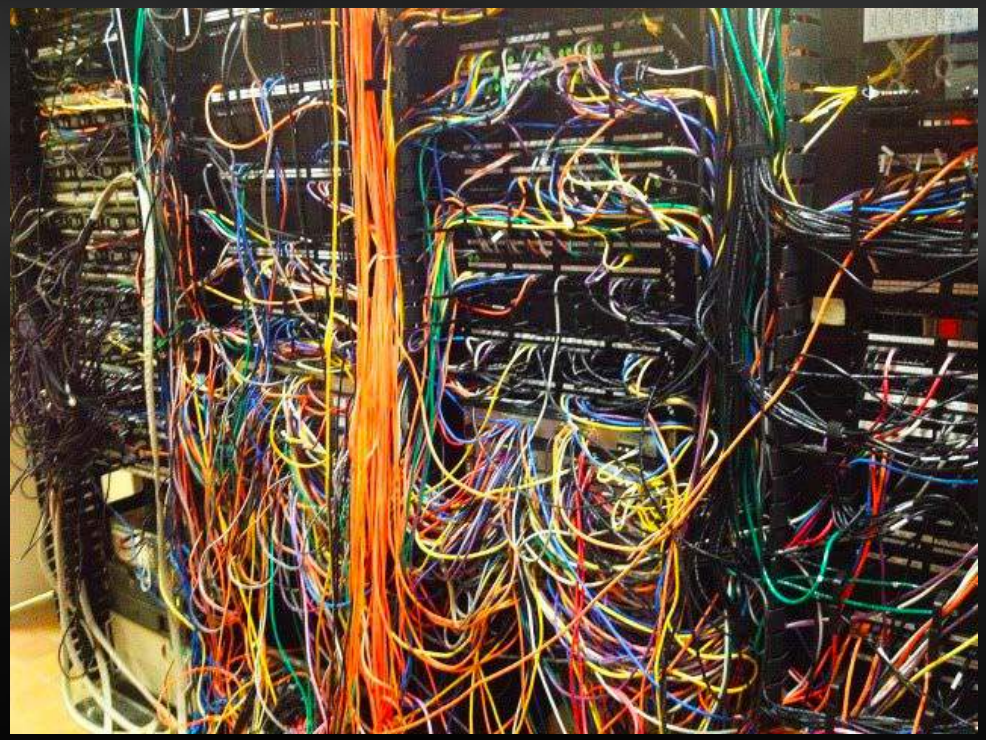


- Logic 部分才是真正有意义的 - What
- Control部分只是影响Logic部分的效率 - How

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.
```

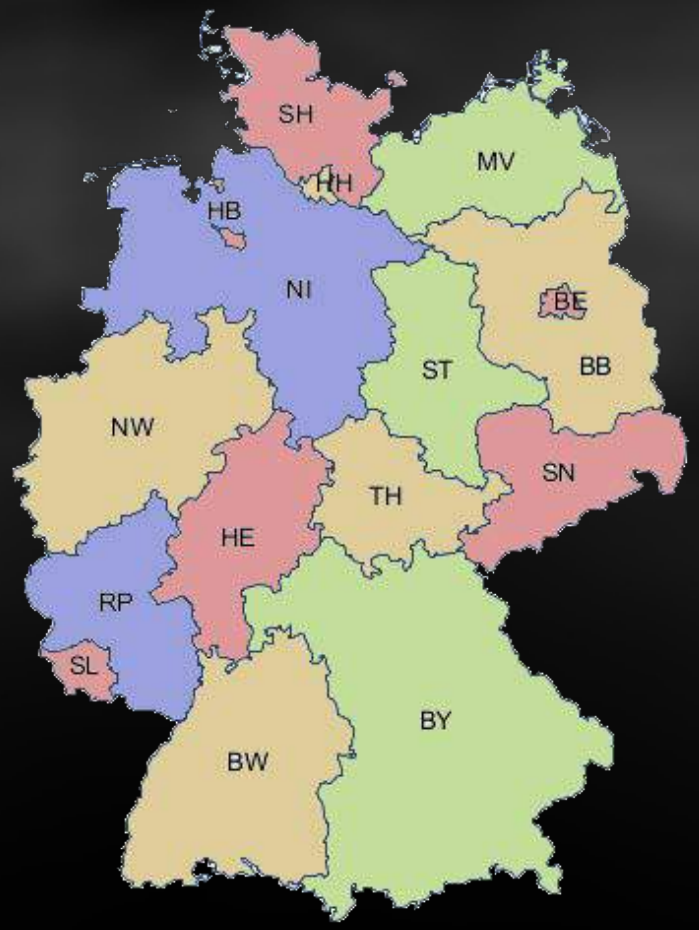
地图四色问题 - Prolog

```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).
```

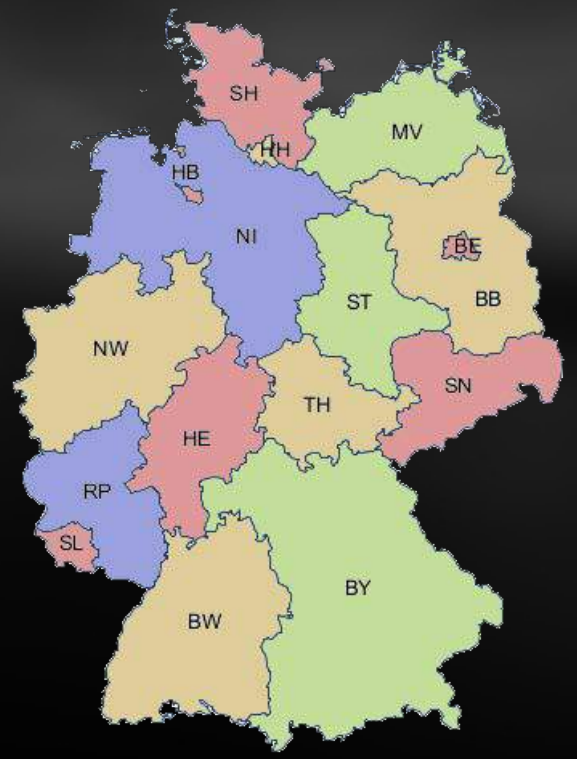


地图四色问题 - Prolog

```
/* 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).
```

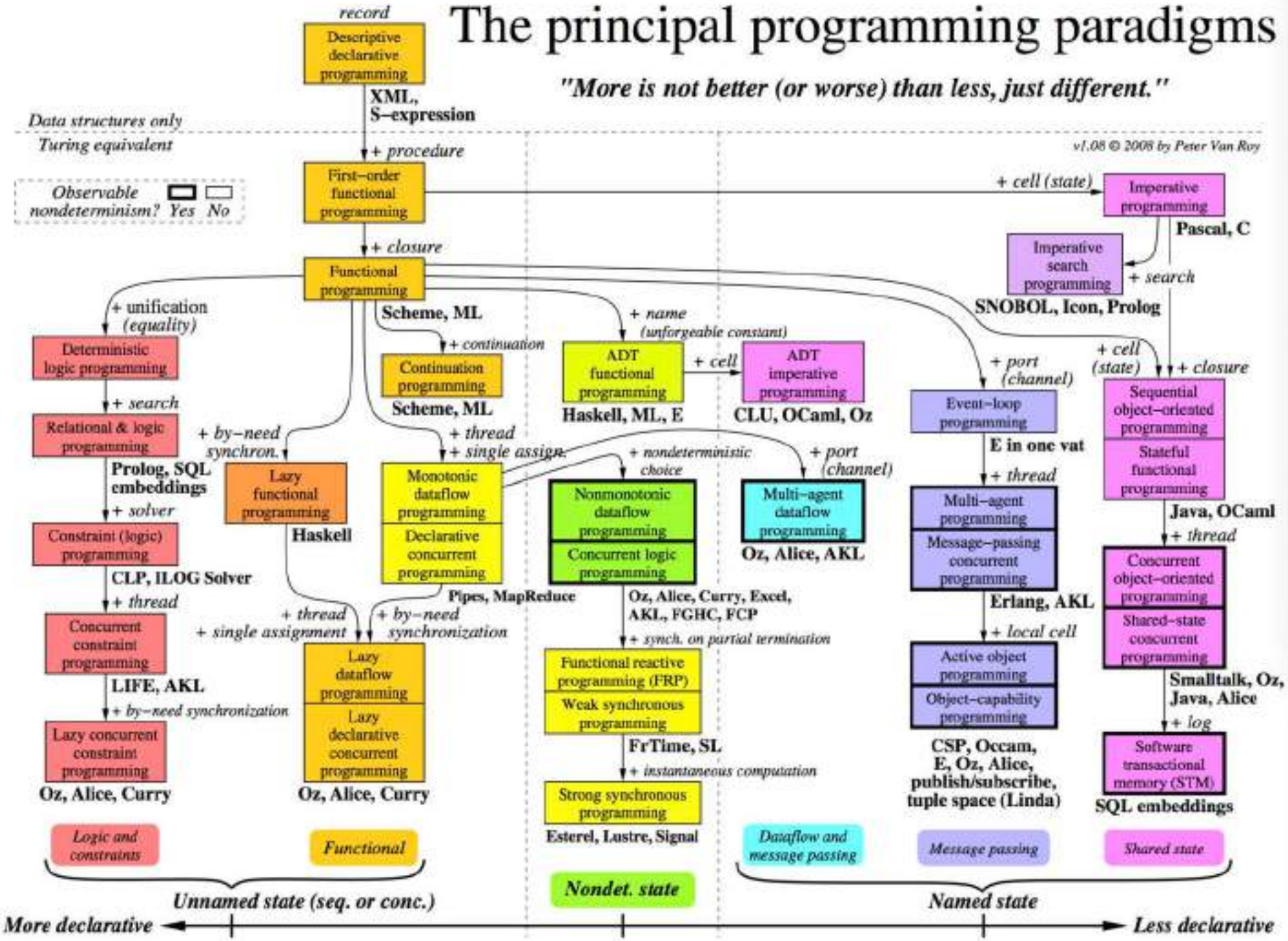


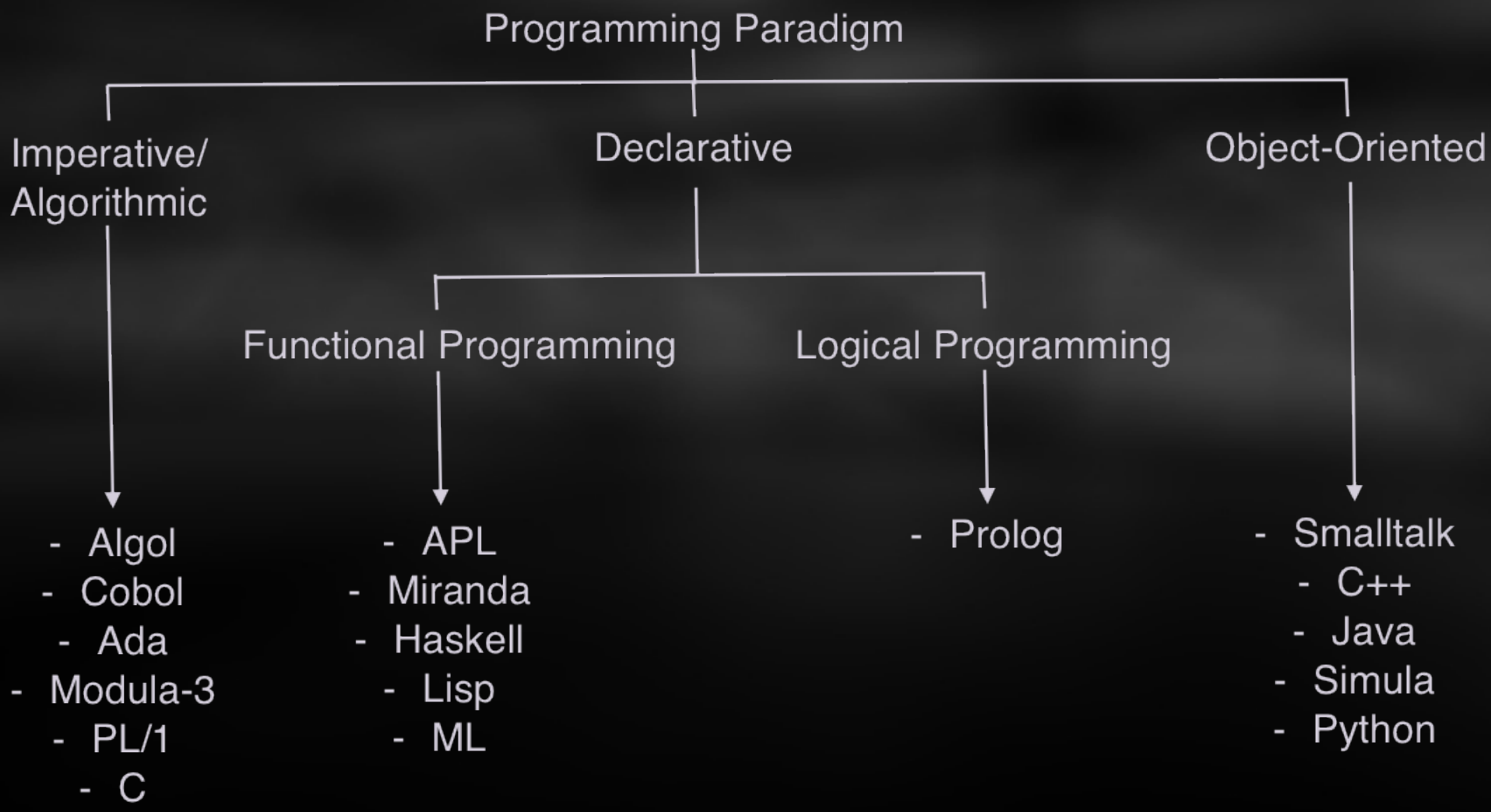
程序世界里的编程范式

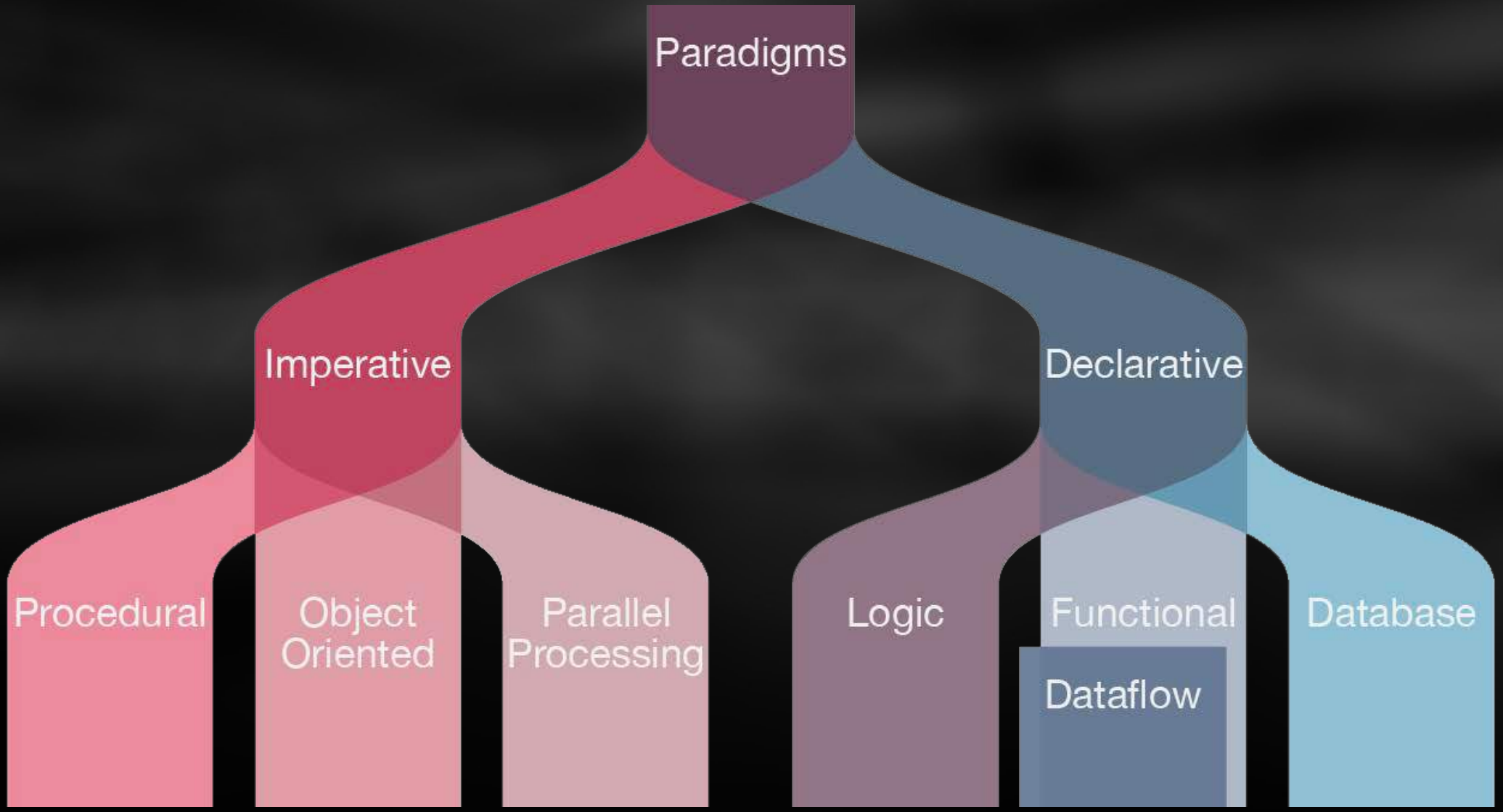

Left
brain
I am the left brain.
I am a scientist. A mathematician.
I love the familiar. I categorize. I am accurate. Linear.
Analytical. Strategic. I am practical.
Always in control. A master of words and language.
Realistic. I calculate equations and play with numbers.
I am order. I am logic.
I know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




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