代码编程中的编程范式

陈皓

个人简介

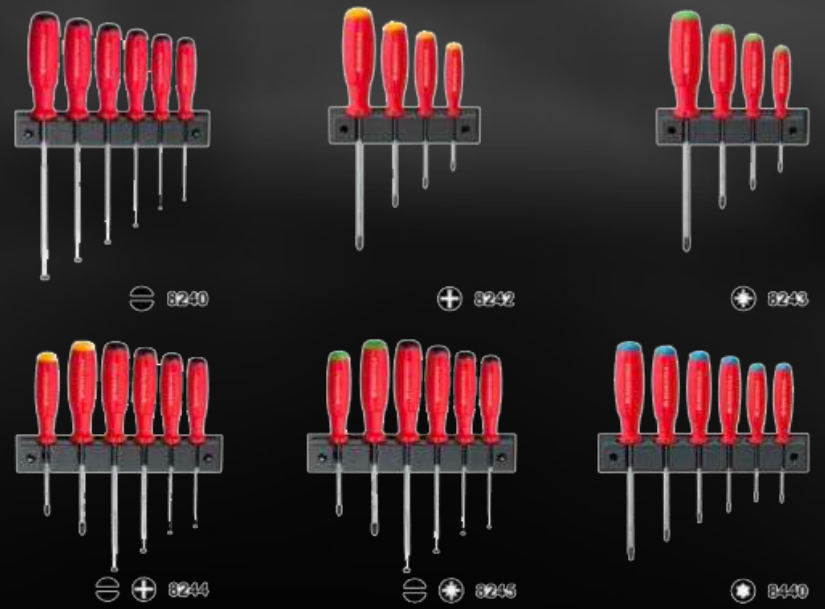
- **18年工作经历，超大型分布式系统基础架构研发和设计**
- **擅长领域：金融、电子商务、云计算、大数据**
- **职业背景**
  - 阿里巴巴资深架构师（阿里云、天猫、淘宝）
  - 亚马逊高级研发经理（AWS、全球购、商品需求预测）
  - 汤森路透研发经理（实时金融数据处理基础架构）
- **目前在创业，致力于为企业提供技术架构管理产品**
  - 目标：用户不用改一行代码就可以提高系统的性能和稳定性

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

```c
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;
}
```

```c
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

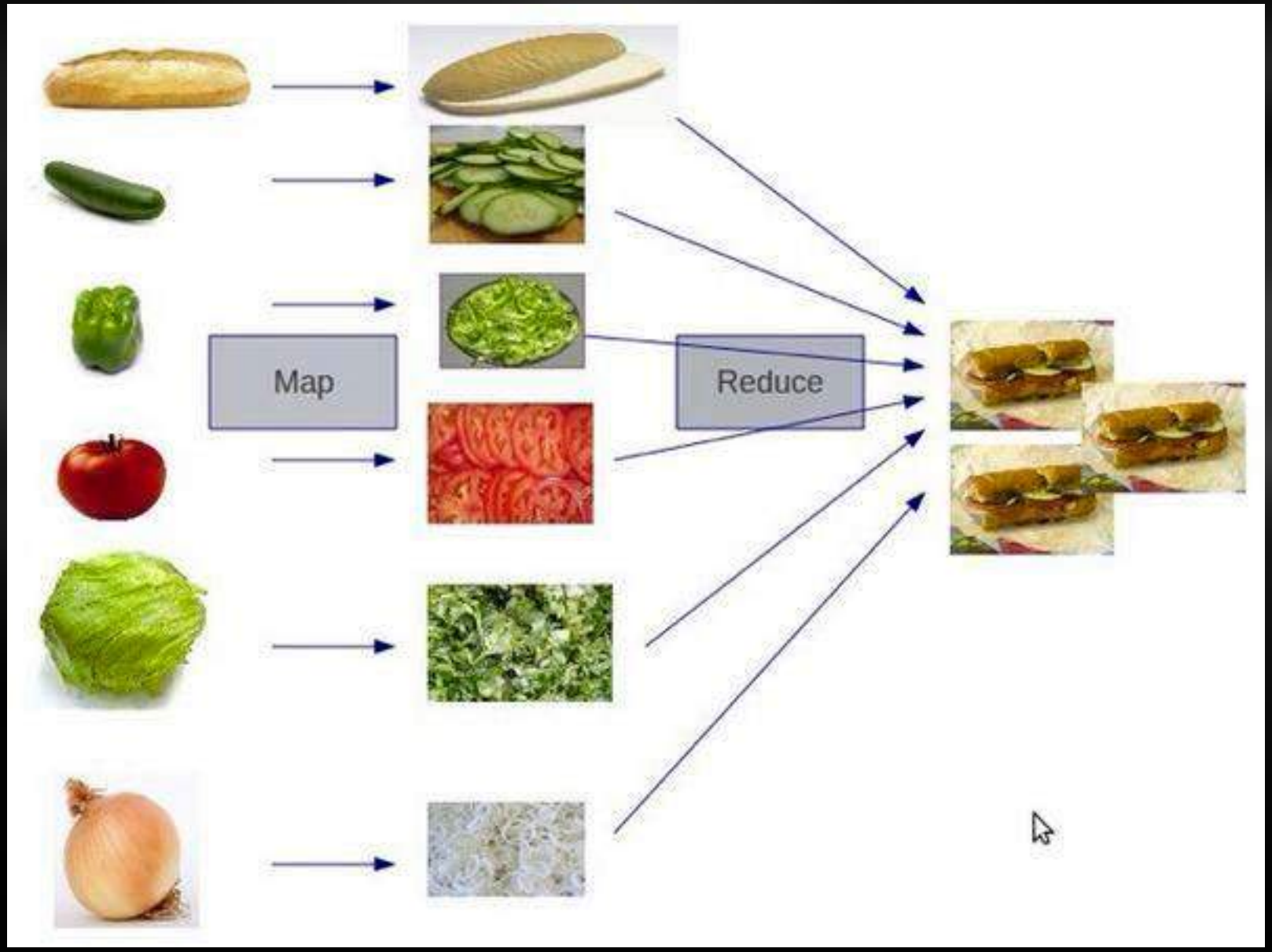
```scheme
(define (plus x y)  (+ x y))
(define (times x y) (* x y))
(define (square x)  (times x x))

(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

```python
# 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

```
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()
```

```
@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。

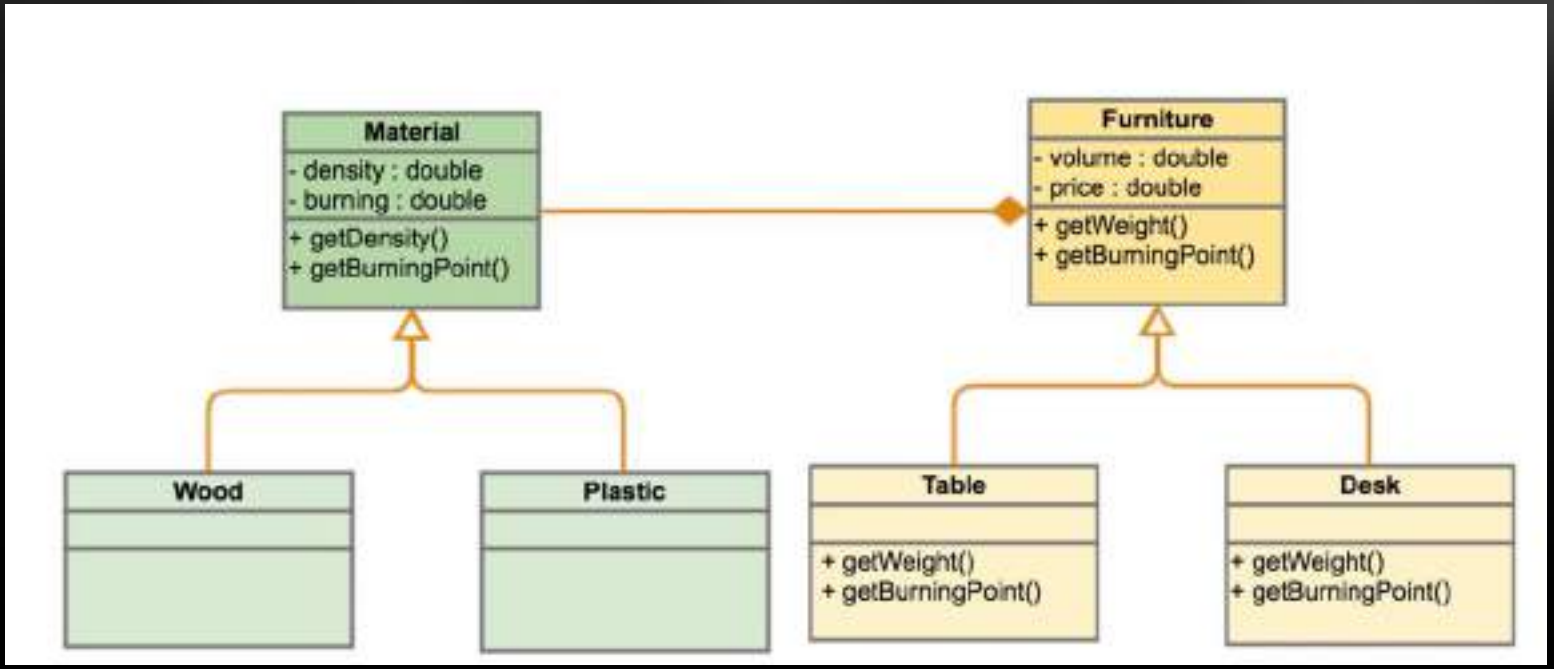
面向对象算法拼装

```java
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;
    }
}
```

```java
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。