你所需要知道的代码整洁之道

- 分享人：李忠键
- 分享时间：2020-01-17
- 联系方式：zhongjianlee@outlook.com

> 程序是写给人读的，只是偶尔让计算机执行一下。
> —— Donald Ervin Knuth（高德纳）

序

每次 review 过往写的代码，总有一种不忍直视的感觉。想提高编码能力，故阅读了一些相关书籍及博文，并有所感悟，今将一些读书笔记及个人心得感悟梳理出来。抛转引玉，希望这砖能抛得起来。

大纲

坏味道的代码

开始阅读之前，大家可以快速思考一下，大家脑海里的好代码和坏代码都是怎么样的“形象”呢?

如果看到这一段代码，如何评价呢?

```
if (a && d || b && c && !d || (!a || !b) && c) {
  // ...
} else {
  // ...
}
```

上面这段代码，尽管是特意为举例而写的，要是真实遇到这种代码，想必大家都“一言难尽”吧。大家多多少少都有一些坏味道的代码的“印象”，坏味道的代码总有一些共性：

- Duplicated Code（重复代码）
- Long Method（过长函数）
- Large Class （过大的类）
- Long Parameter List（过长参数列）
- Temporary Field（令人迷惑的暂时字段）
- Shotgun Surgery（霰弹式修改）：一种变化引发多个类相应修改
- ......

那坏味道的代码是怎样形成的呢?

- 上一个写这段代码的程序员经验、水平不足，或写代码时不够用心；
- 产品经理提出的奇葩需求导致写了很多hack代码；
- 某一个模块业务太复杂，需求变更的次数太多，经手的程序员太多。
- ......

对坏味道的代码有一个大概的了解后，或许读者心中有一个疑问：代码的好坏有没有一些量化的标准去评判呢？答案是肯定的。

接下来，通过了解圈复杂度去衡量我们写的代码。然而当代码的坏味道已经“弥漫”到处都是了，这时我们应该了解一下重构。代码到了我们手里，不能继续“发散”坏味道，这时应该了解如何编写 clean code。此外，我们还应该掌握一些编码原则及设计模式，这样才能做到有的放矢。

圈复杂度

> 圈复杂度（Cyclomatic complexity，简写CC）也称为条件复杂度，是一种代码复杂度的衡量标准。由托马斯·J·麦凯布（Thomas J. McCabe, Sr.）于1976年提出，用来表示程序的复杂度。

圈复杂度可以用来衡量一个模块判定结构的复杂程度，数量上表现为独立现行路径条数，也可理解为覆盖所有的可能情况最少使用的测试用例数。

判定方法

圈复杂度可以通过程序控制流图计算，公式为：

> V(G) = e + 2 - n

- e : 控制流图中边的数量
- n : 控制流图中节点的数量

有一个简单的计算方法：圈复杂度实际上就是等于判定节点的数量再加上1。

> 注： `if else` 、 `switch case` 、 `for循环` 、 `三元运算符` 、 `||` 、 `&&` 等，都属于一个判定节点。

衡量标准

> 代码复杂度低，代码不一定好，但代码复杂度高，代码一定不好。

圈复杂度	代码状况	可测性	维护成本
1 - 10	清晰、结构化	高	低
10 - 20	复杂	中	中
20 - 30	非常复杂	低	高
>30	不可读	不可测	非常高

圈复杂度检测方法

ESLint 规则

ESLint 提供了检测代码圈复杂度的 rules。开启 rules 中的 complexity 规则，并将圈复杂度大于 0 的代码的 rule severity 设置为 warn 或 error 。

```
rules: {
    complexity: [
        'warn',
        { max: 0 }
    ]
}
```

CLIEngine

借助 ESLint 的 CLIEngine ，在本地使用自定义的 ESLint 规则扫描代码，并获取扫描结果输出。

降低代码的圈复杂度

很多情况下，降低圈复杂度就能提高代码的可读性了。针对圈复杂度，结合例子给出一些改善的建议：

1. 抽象配置

通过抽象配置将复杂的逻辑判断进行简化。

before:

```
// ...
if (type === '扫描') {
    scan(args)
} else if (type === '删除') {
    delete(args)
} else if (type === '设置') {
    set(args)
} else {
    // ...
}
```

after:

```
const ACTION_TYPE = {
    扫描: scan,
    删除: delete,
    设置: set
}
ACTION_TYPE[type](args)
```

2. 提炼函数

将代码中的逻辑进行抽象提炼成单独的函数，有利于降低代码复杂度和降低维护成本。尤其是当一个函数的代码很长，读起来很费力的时候，就应该思考能否提炼成多个函数。

before:

```
function example(val) {
    if (val > MAX_VAL) {
        val = MAX_VAL
    }
    for (let i = 0; i < val; i++) {
        doSomething(i)
    }
    // ...
}
```

after:

```
function setMaxVal(val) {
    return val > MAX_VAL ? MAX_VAL : val
}

function getCircleArea(val) {
    for (let i = 0; i < val; i++) {
        doSomething(i);
    }
}
function example(val) {
    return getCircleArea(getCircleArea(val)
}
```

3. 逆向条件简化条件判断

某些复杂的条件判断可能逆向思考后会变的更简单，还能减少嵌套。

before:

```
function checkAuth(user){
    if (user.auth) {
        if (user.name === 'admin') {
            // ...
        } else if (user.name === 'root') {
            // ...
        }
    }
}
```

after:

```
function checkAuth(user){
    if (!user.auth) return
    if (user.name === 'admin') {
        // ...
    } else if (user.name === 'root') {
        // ...
     }
}
```

4. 合并条件简化条件判断

将冗余的条件合并，然后再进行判断。

before:

```
if (fruit === 'apple') {
    return true
} else if (fruit === 'cherry') {
    return true
} else if (fruit === 'peach') {
    return true
} else {
    return true
}
```

after:

```
const redFruits = ['apple', 'cherry', 'peach']
if (redFruits.includes(fruit) {
    return true
}
```

5. 提取条件简化条件判断

对复杂难懂的条件进行提取并语义化。

before:

```
if ((age < 20 && gender === '女') || (age > 60 && gender === '男')) {
    // ...
} else {
    // ...
}
```

after:

```
function isYoungGirl(age, gender) {
    return (age < 20 && gender === '女'
}
function isOldMan(age, gender) {
    return age > 60 && gender === '男'
}
if (isYoungGirl(age, gender) || isOldMan(age, gender)) {
    // ...
} else {
    // ...
}
```

后文有简化条件表达式更全面的总结。

重构

重构一词有名词和动词上的理解。名词：

> 对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本。

动词：

> 使用一系列重构手法，在不改变软件可观察行为的前提下，调整其结构。

为何重构

如果遇到以下的情况，可能就要思考是否需要重构了：

- 重复的代码太多
- 代码的结构混乱
- 程序没有拓展性
- 对象结构强耦合
- 部分模块性能低

为何重构，不外乎以下几点：

- 重构改进软件设计
- 重构使软件更容易理解
- 重构帮助找到 bug
- 重构提高编程速度

重构的类型：

- 对现有项目进行代码级别的重构；
- 对现有的业务进行软件架构的升级和系统的升级。

本文讨论的内容只涉及第一点，仅限代码级别的重构。

重构时机

> 第一次做某件事时只管去做；第二次做类似的事会产生反感，但无论如何还是可以去做；第三次再做类似的事，你就应该重构。

- 添加功能时：当添加新功能时，如果发现某段代码改起来特别困难，拓展功能特别不灵活，就要重构这部分代码使添加新特性和功能变得更容易；

- 修补错误时：在你改 bug 或查找定位问题时，发现自己以前写的代码或者别人的代码设计上有缺陷（如扩展性不灵活），或健壮性考虑得不够周全（如漏掉一些该处理的异常），导致程序频繁出现问题，那么此时就是一个比较好的重构时机；
- 复审代码时：团队进行 Code Review 的时候，也是一个进行重构的合适时机。

代码整洁之道

关键思想

- 代码应当易于理解；
- 代码的写法应当使别人理解它所需的时间最小化。

代码风格

> 关键思想：一致的风格比“正确”的风格更重要。

原则：

- 使用一致的布局
- 让相似的代码看上去相似
- 把相关的代码行分组，形成代码块

注释

注释的目的是尽量帮助读者了解得和作者一样多。因此注释应当有很高的 信息/空间 率。

不需要注释

- 不要为了注释而注释
- 不要给不好的名字加注释（应该把名字起好）
- 不要为那些从代码本身就能快速推断的事实写注释

需要写注释

- 对于为什么代码写成这样而不是那样的内在理由（指导性批注）
- 常量背后的故事，为什么是这个值
- 给读者意料之外的行为加上注释
- 在文件/类的级别上使用“全局观”注释来解释所有的部分是如何一起工作的
- 用注释来总结代码块，使读者不致迷失在细节中
- 尽量精确地描述函数的行为
- 声明代码的高层次意图，而非明显的细节

- 在注释中使用输入/输出例子进行说明
- 代码中的缺陷，使用标记

标记	通常的意义
TODO:	还没处理的事情
FIXME:	已知的无法运行的代码
HACK:	对一个问题不得不采用的比价粗糙的解决方案

命名

> 关键思想：把信息装入名字中。

良好的命名是一种以“低代价”取得代码高可读性的途径。

选择专业名词

“把信息装入名字中”包括要选择非常专业的词，并且避免使用“空洞”的词。

单词	更多选择
send	deliver, despatch, announce, distribute, route
find	search, extract, locate, recover
start	launch, create, begin, open
make	create, set up, build, generate, compose, add, new

避免像tmp和retval这样泛泛的名字

- retval这个名字没有包含很多信息
- tmp 只应用于短期存在且临时性为其主要存在因素的变量

用具体的名字代替抽象的名字

在给变量、函数或者其他元素命名时，要把它描述得更具体而不是更抽象。

为名字附带更多信息

如果关于一个变量有什么重要事情的读者必须知道，那么是值得把额外的“词”添加到名字中的。

名字的长度

- 在小的作用域里可以使用短的名字
- 为作用域大的名字采用更长的名字
- 丢掉没用的词

不会被误解的名字

- 用min和max来表示极限
- 用first和last来表示包含的范围
- 用begin和end来表示排除范围
- 给布尔值命名：is，has，can，should

语义相反的词汇要成对出现

正	反
add	remove
create	destory
insert	delete
get	set
increment	decrement
show	hide
start	stop

其他命名小建议

- 计算限定符作为前缀或后缀（Avg、Sum、Total、Min、Max）
- 变量名要能准确地表示事物的含义
- 用动名词命名函数名
- 变量名的缩写，尽量避免不常见的缩写

简化条件表达式

分解条件表达式

有一个复杂的条件（if-then-else）语句，从if、then、else三个段落中分别提炼出独立函数。根据每个小块代码的用途，为分解而得到的新函数命名，并将原函数中对应的代码改为调用新建函数，从而更清楚地表达自己的意图。对于条件逻辑，可以突出条件逻辑，更清楚地表明每个分支的作用，并且突出每个分支的原因。

合并条件表达式

有一系列条件测试，都得到相同结果。将这些测试合并为一个条件表达式，并将这个条件表达式提炼成为一个独立函数。

- 确定这些条件语句都没有副作用；
- 使用适当的逻辑操作符，将一系列相关条件表达式合并为一个；
- 对合并后的条件表达式实施提炼函数。

合并重复的条件片段

在条件表达式的每个分支上有着相同的一段代码，将这段重复代码搬移到条件表达式之外。

以卫语句取代嵌套条件表达式

函数中的条件逻辑使人难以看清正常的执行路径。使用卫语句表现所有特殊情况。

> 如果某个条件极其罕见，就应该单独检查该条件，并在该条件为真时立刻从函数中返回。这样的单独检查常常被称为“卫语句”（guard clauses）。

常常可以将条件表达式反转，从而实以卫语句取代嵌套条件表达式，写成更加“线性”的代码来避免深嵌套。

变量与可读性

变量存在的问题：

- 变量越多，就越难全部跟踪它们的动向
- 变量的作用域越大，就需要跟踪它的动向越久
- 变量改变得越频繁，就越难以跟踪它的当前值

内联临时变量

如果有一个临时变量，只是被简单表达式赋值一次，而将所有对该变量的引用动作，替换为对它赋值的那个表达式自身。

以查询取代临时变量

以一个临时变量保存某一表达式的运算结果，将这个表达式提炼到一个独立函数中。将这个临时变量的所有引用点替换为对新函数的调用。此后，新函数就可被其他函数使用。

总结变量

接上条，如果该表达式比较复杂，建议通过一个总结变量名来代替一大块代码，这个名字会更容易管理和思考。

引入解释性变量

将复杂表达式（或其中一部分）的结果放进一个临时变量，以此变量名称来解释表达式用途。

在条件逻辑中，引入解释性变量特别有价值：可以将每个条件子句提炼出来，以一个良好命名的临时变量来解释对应条件子句的意义。使用这项重构的另一种情况是，在较长算法中，可以运用临时变量来解释每一步运算的意义。

好处：

- 把巨大的表达式拆分成小段
- 通过用简单的名字描述子表达式来让代码文档化
- 帮助读者识别代码中的主要概念

分解临时变量

程序有某个临时变量被赋值超过一次，它既不是循环变量，也不是用于收集计算结果。针对每次赋值，创造一个独立、对应的临时变量。

临时变量有各种不同用途：

- 循环变量
- 结果收集变量（通过整个函数的运算，将构成的某个值收集起来）

如果临时变量承担多个责任，它就应该被替换（分解）为多个临时变量，每个变量只承担一个责任。

以字面常量取代 Magic Number

有一个字面值，带有特别含义。创造一个常量，根据其意义为它命名，并将上述的字面数值替换为这个常量。

减少控制流变量

```
let done = false;

while (condition && !done) {
    if (...) {
        done = true;
        continue;
    }
}
```

像done这样的变量，称为“控制流变量”。它们唯一的目的就是控制程序的执行，没有包含任何程序的数据。控制流变量通常可以通过更好地运用结构化编程而消除。

```
while (condition) {
    if (...) {
        break;
    }
}
```

如果有多个嵌套循环，一个简单的break不够用，通常解决方案包括把代码挪到一个新函数中。

重新组织函数

提炼函数

当一个过长的函数或者一段需要注释才能让人理解用途的代码，可以将这段代码放进一个独立函数中。

- 函数的粒度小，被复用的机会就很大；
- 函数的粒度小，覆写也会更容易些。

一个函数过长才合适？长度不是问题，关键在于函数名称和函数本体之间的语义距离。

将查询函数和修改函数分离

某个函数既返回对象状态值，又修改对象状态。建立两个不同的函数，其中一个负责查询，另一个负责修改。

以明确函数取代参数

有一个函数，其中完全取决于参数值而采取不同行为。针对该参数的每一个可能值，建立一个独立函数。

引入参数对象

某些参数总是很自然地同时出现，以一个对象取代这些参数。

从函数中提前返回

可以通过马上处理“特殊情况”，并从函数中提前返回。

拆分巨大的语句

如果有很难读的代码，尝试把它所做的所有任务列出来。其中一些任务可以很容易地变成单独的函数（或类）。其他的可以简单地成为一个函数中的逻辑“段落”。

其他建议及思路

减少循环内的嵌套

在循环中，与提早返回类似的技术是 `continue` 。与 `if...return;` 在函数中所扮演的保护语句一样， `if...continue;` 语句是循环中的保护语句。(注意：JavaScript 中 forEach 的特殊性)

德·摩根定律

对于一个布尔表达式，有两种等价写法：

- not (a or b or c) <=> (not a) and (not b) and (not c)
- not (a and b and c) <=> (not a) or (not b) or (not c)

可以使用这些法则让布尔表达式更具有可读性。例如：

```
// before
if (!(file_exitsts && !is_protected)) Error("sorry, could not read file.")

// after
if (!file_exists || is_protected) Error("sorry, could not read file.")
```

使用相关定律能优化开始举例的那段代码：

```
// before
if (a && d || b && c && !d || (!a || !b) && c) {
  // ...
} else {
  // ...
}

// after
if (a && d || c) {
  // ...
} else {
  // ...
}
```

具体简化过程及涉及相关定律可以参考这篇推文：你写这样的代码，不怕同事打你嘛?

重新组织代码

> 所谓工程学就是关于把大问题拆分成小问题再把这些问题的解决方案放回一起。

把这条原则应用于代码会使代码更健壮并且更容易读。

积极地发现并抽取不相关的子逻辑，是指：

- 看看某个函数或代码块，问问自己：这段代码高层次的目标是什么？
- 对于每一行代码，问一下：它是直接为了目标而工作吗？
- 如果足够的行数在解决不相关的子问题，抽取代码到独立的函数中。

把想法变成代码

> 如果你不能把一件事解释给你祖母听的话说明你还没真正理解它。 --阿尔伯特·爱因斯坦

步骤如下：

1. 像对着一个同事一样用自然语言描述代码要做什么
2. 注意描述中所用的关键词和短语
3. 写出与描述所匹配的代码

编码原则 & 设计模式

有必要熟知前人总结的一些经典的编码原则及涉及模式，以此来改善我们既有的编码习惯，所谓“站在巨人肩上编程”。

SOLID原则

> SOLID 是面向对象设计（OOD）的五大基本原则的首字母缩写组合，由俗称“鲍勃大叔”的Robert C.Martin在《敏捷软件开发：原则、模式与实践》一书中提出来。

- S(Single Responsibility Principle)：单一职责原则，简称SRP
- O(Open Close Principle)：开放封闭原则，简称OCP
- L(Liskov Substitution Principle)：里氏替换原则，简称LSP
- I(Interface SegregationPrinciple)：接口隔离原则，简称ISP
- D(Dependence Inversion Principle)：依赖倒置原则，简称DIP

单一职责原则

> A class should have only one reason to change.

一个类应该有且仅有一个原因引起它的变更。

通俗来讲：一个类只负责一项功能或一类相似的功能。当然这个“一”并不是绝对的，应该理解为一个类只负责尽可能独立的一项功能，尽可能少的职责。

优点：

- 功能单一，职责清晰。
- 增强可读性，方便维护。

缺点：

- 拆分得太详细，类的数量会急剧增加。
- 职责的度量没有统一的标准，需要根据项目实现情况而定。

这条定律同样适用于组织函数时的编码原则。

开放封闭原则

> Software entities （classes,modules,functions,etc.）should be openfor extension,but closed for modification.

软件实体（如类、模块、函数等）应该对拓展开放，对修改封闭。

在一个软件产品的生命周期内，不可避免会有一些业务和需求的变化，我们在设计代码的时候应该尽可能地考虑这些变化。在增加一个功能时，应当尽可能地不去改动已有的代码；当修改一个模块时不应该影响到其他模块。

```
const makeSound = function( animal ) {
    animal.sound();
};

const Duck = function(){};
Duck.prototype.sound = function(){
    console.log( '嘎嘎嘎' );
};

const Chicken = function(){};
Chicken.prototype.sound = function(){
    console.log( '咯咯咯' );
};

makeSound( new Duck() ); // 嘎嘎嘎
makeSound( new Chicken() ); // 咯咯咯
```

里氏替换原则

> Functions that use pointers to base classes must be able to useobjects of derived classes without knowing it.

所有能引用基类的地方必须能透明地使用其子类的对象。

只要父类能出现的地方子类就能出现（就可以用子类来替换它）。反之，子类能出现的地方父类不一定能出现（子类拥有父类的所有属性和行为，但子类拓展了更多的功能）。

接口隔离原则

> Clients should not be forced to depend upon interfaces that they don ' t use.Instead of one fat interface many small interfaces arepreferred based on groups of methods,each one serving onesubmodule.

客户端不应该依赖它不需要的接口。用多个细粒度的接口来替代由多个方法组成的复杂接口，每一个接口服务于一个子模块。

接口尽量小，但是要有限度。当发现一个接口过于臃肿时，就要对这个接口进行适当的拆分。但是如果接口过小，则会造成接口数量过多，使设计复杂化。

依赖倒置原则

> High level modules should not depend on low level modules; bothshould depend on abstractions.Abstractions should not depend ondetails.Details should depend upon abstractions.

高层模块不应该依赖低层模块，二者都该依赖其抽象。抽象不应该依赖细节，细节应该依赖抽象。

把具有相同特征或相似功能的类，抽象成接口或抽象类，让具体的实现类继承这个抽象类（或实现对应的接口）。抽象类（接口）负责定义统一的方法，实现类负责具体功能的实现。

是否一定要遵循这些设计原则

- 软件设计是一个逐步优化的过程
- 不是一定要遵循这些设计原则

没有这么充足的时间遵循这些原则去设计，或遵循这些原则设计的实现成本太大。在受现实条件所限不能遵循五大原则来设计时，我们还可以遵循下面这些更为简单、实用的原则：

LoD原则（Law of Demeter）

> Each unit should have only limited knowledge about other units: onlyunits "closely"related to the current unit.Only talk to your immediatefriends,don ' t talk to strangers.

每一个逻辑单元应该对其他逻辑单元有最少的了解：也就是说只亲近当前的对象。只和直接（亲近）的朋友说话，不和陌生人说话。

这一原则又称为迪米特法则，简单地说就是：一个类对自己依赖的类知道的越少越好，这个类只需要和直接的对象进行交互，而不用在乎这个对象的内部组成结构。

例如，类A中有类B的对象，类B中有类C的对象，调用方有一个类A的对象a，这时如果要访问C对象的属性，不要采用类似下面的写法：

```
a.getB().getC().getProperties()
```

而应该是：

```
a.getProperties()
```

KISS原则（Keep It Simple and Stupid）

> Keep It Simple and Stupid.

保持简单和愚蠢。

- “简单”就是要让你的程序能简单、快速地被实现；
- “愚蠢”是说你的设计要简单到任何人都能理解，即简单就是美！

DRY原则（Don't Repeat Yourself）

> DRY原则（Don't Repeat Yourself）

不要重复自己。

不要重复你的代码，即多次遇到同样的问题，应该抽象出一个共同的解决方法，不要重复开发同样的功能。也就是要尽可能地提高代码的复用率。

要遵循DRY原则，实现的方式非常多：

- 函数级别的封装：把一些经常使用的、重复出现的功能封装成一个通用的函数。
- 类级别的抽象：把具有相似功能或行为的类进行抽象，抽象出一个基类，并把这几个类都有的方法提到基类去实现。
- 泛型设计：Java 中可使用泛型，以实现通用功能类对多种数据类型的支持；C++中可以使用类模板的方式，或宏定义的方式；Python中可以使用装饰器来消除冗余的代码。

DRY原则在单人开发时比较容易遵守和实现，但在团队开发时不太容易做好，特别是对于大团队的项目，关键还是团队内的沟通。

YAGNI原则（You Aren ' t Gonna Need It）

> You aren ' t gonna need it,don ' t implement something until it isnecessary.

你没必要那么着急，不要给你的类实现过多的功能，直到你需要它的时候再去实现。

- 只考虑和设计必需的功能，避免过度设计。
- 只实现目前需要的功能，在以后需要更多功能时，可以再进行添加。
- 如无必要，勿增加复杂性。

Rule Of Three原则

Rule of three 称为“三次法则”，指的是当某个功能第三次出现时，再进行抽象化，即事不过三，三则重构。

- 第一次实现一个功能时，就尽管大胆去做；
- 第二次做类似的功能设计时会产生反感，但是还得去做；
- 第三次还要实现类似的功能做同样的事情时，就应该去审视是否有必要做这些重复劳动了，这个时候就应该重构你的代码了，即把重复或相似功能的代码进行抽象，封装成一个通用的模块或接口。

CQS原则（Command-Query Separation）

- 查询（Query）：当一个方法返回一个值来回应一个问题的时候，它就具有查询的性质；
- 命令（Command）：当一个方法要改变对象的状态的时候，它就具有命令的性质。

保证方法的行为严格的是命令或者查询，这样查询方法不会改变对象的状态，没有副作用；而会改变对象的状态的方法不可能有返回值。

设计模式

设计模式的开山鼻祖 GoF 在《设计模式：可复用面向对象软件的基础》一书中提出的23种经典设计模式被分成了三类，分别是创建型模式、结构型模式和行为型模式。

> 在面向对象软件设计过程中针对特定问题的简洁而优雅的解决方案。

常用的设计模式有：策略模式、发布—订阅模式、职责链模式等。

比如策略模式使用的场景：

> 策略模式：定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。

```
if (account === null || account === '') {
    alert('手机号不能为空');
    return false;
}
if (pwd === null || pwd === '') {
    alert('密码不能为空');
    return false;
}
if (!/(^1[3|4|5|7|8][0-9]{9}$)/.test(account)) {
    alert('手机号格式错误');
    return false;
}
if(pwd.length<6){
    alert('密码不能小于六位');
    return false;
}
```

使用策略模式：

```
const strategies = {
    isNonEmpty: function (value, errorMsg) {
        if (value === '' || value === null) {
            return errorMsg;
        }
    },
    isMobile: function (value, errorMsg) { // 手机号码格式
        if (!/(^1[3|4|5|7|8][0-9]{9}$)/.test(value)) {
            return errorMsg;
        }
    },
    minLength: function (value, length, errorMsg) {
        if (value.length < length) {
            return errorMsg;
        }
    }
};

const accountIsMobile = strategies.isMobile(account,'手机号格式错误');
const pwdMinLength = strategies.minLength(pwd,8,'密码不能小于8位');
const errorMsg = accountIsMobile || pwdMinLength;
if (errorMsg) {
    alert(errorMsg);
    return false;
}
```

又比如，发布—订阅模式具有的特点：

- 时间上的解耦

- 对象之间的解耦

既可以用在异步编程中，也可以帮助我们完成更松耦合的代码编写。

如果大家需要了解设计模式更多知识，建议另外找资料学习。

总结

宋代禅宗大师青原行思提出参禅的三重境界：

> 参禅之初，看山是山，看水是水；禅有悟时，看山不是山，看水不是水；禅中彻悟，看山仍是山，看水仍是水。

同理，编程同样存在境界：编程的一重境界是照葫芦画瓢，二重境界是可以灵活运用，三重境界则是心中无模式。唯有多实践，多感悟，方能突破一重又一重的境界。

最后，愿大家终将能写出自己不再讨厌的代码。

参考

- 《代码整洁之道》
- 《编写可读代码的艺术》
- 《重构-改善既有代码的设计》
- 《JavaScript设计模式与开发实践》
- 《人人都懂设计模式：从生活中领悟设计模式：Python实现》