编程范式游记（9）- 编程的本质

2018-02-08 陈皓

我们讲了各式各样的不同语言的编程范式，从C语言的泛型，讲到C++的泛型，再讲到函数式的 Map/Reduce/Filter，以及 Pipeline 和 Decorator，还有面向对象的多态通过依赖于接口而不是实现的桥接模式、策略模式和代理模式，以及面向对象的IoC，还有JavaScript的原型编程在运行时对对象原型进行修改，以及Go语言的委托模式.....

所有的这一切，不知道你是否看出一些端倪，或是其中的一些共性来了？

两篇论文

1976年，瑞士计算机科学家，Algol W，Modula，Oberon和Pascal语言的设计师 Niklaus Emil Wirth写了一本非常经典的书《Algorithms + Data Structures = Programs》（链接为1985年版），即算法 + 数据结构 = 程序。

这本书主要写了算法和数据结构的关系，这本书对计算机科学的影响非常深远，尤其在计算机科学的教育中。

1979年，英国逻辑学家和计算机科学家 Robert Kowalski 发表论文 Algorithm = Logic + Control，并且主要开发"逻辑编程"相关的工作。

Robert Kowalski是一位逻辑学家和计算机科学家，从20世纪70年代末到整个80年代致力于数据库的研究，并在用计算机证明数学定理等当年的重要应用上颇有建树，尤其是在逻辑、控制和算法等方面提出了革命性的理论，极大地影响了数据库、编程语言，直至今日的人工智能。

Robert Kowalski在这篇论文里提到：

> An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used. The logic component determines the meaning of the algorithm whereas the control component only affects its efficiency. The efficiency of an algorithm can often be improved by improving the control component without changing the logic of the algorithm. We argue that computer programs would be more often correct and more easily improved and modified if their logic and control aspects were identified and separated in the program text.

翻译过来的意思大概就是：

> 任何算法都会有两个部分， 一个是 Logic 部分，这是用来解决实际问题的。另一个是Control部分，这是用来决定用什么策略来解决问题。Logic部分是真正意义上的解决问题的算法，而Control部分只是影响解决这个问题的效率。程序运行的效率问题和程序的逻辑其实是没有关系的。我们认为，如果将 Logic 和 Control 部分有效地分开，那么代码就会变得更容易改进和维护。

注意，最后一句话是重点——如果将 Logic 和 Control 部分有效地分开，那么代码就会变得更容易改进和维护。

编程的本质

两位老先生的两个表达式：

- Programs = Algorithms + Data Structures
- Algorithm = Logic + Control

第一个表达式倾向于数据结构和算法，它是想把这两个拆分，早期都在走这条路。他们认为，如果数据结构设计得好，算法也会变得简单，而且一个好的通用的算法应该可以用在不同的数据结构上。

第二个表达式则想表达，数据结构不复杂，复杂的是算法，也就是我们的业务逻辑是复杂的。我们的算法由两个逻辑组成，一个是真正的业务逻辑，另外一种是控制逻辑。程序中有两种代码，一种是真正的业务逻辑代码，另一种代码是控制我们程序的代码，叫控制代码，这根本不是业务逻辑，业务逻辑不关心这个事情。

算法的效率往往可以通过提高控制部分的效率来实现，而无须改变逻辑部分，也就无无须改变算法的意义。举个阶乘的例子： X(n)！= X(n) X(n-1) X(n-2) X(n-3) ... 3 2 1。逻辑部分用来定义阶乘：1） 1是0的阶乘； 2）如果v是x的阶乘，且u=v(x+1)，那么u是x+1的阶乘。

用这个定义，既可以从上往下地将x+1的阶乘缩小为先计算x的阶乘，再将结果乘以1（recursive，递归），也可以由下而上逐个计算一系列阶乘的结果（iteration，遍历）。

控制部分用来描述如何使用逻辑。最粗略的看法可以认为"控制"是解决问题的策略，而不会改变算法的意义，因为算法的意义是由逻辑决定的。对同一个逻辑，使用不同控制，所得到的算法，本质是等价的，因为它们解决同样的问题，并得到同样的结果。

因此，我们可以通过逻辑分析，来提高算法的效率，保持它的逻辑，而更好地使用这一逻辑。比如，有时用自上而下的控制替代自下而上，能提高效率。而将自上而下的顺序执行改

为并行执行，也会提高效率。

总之，通过这两个表达式，我们可以得出：

Program = Logic + Control + Data Structure

前面讲了这么多的编程范式，或是程序设计的方法。其实，我们都是在围绕着这三件事来做的。比如：

- 就像函数式编程中的Map/Reduce/Filter，它们都是一种控制。而传给这些控制模块的那个lambda表达式才是我们要解决的问题的逻辑，它们共同组成了一个算法。最后，我再把数据放在数据结构里进行处理，最终就成为了我们的程序。
- 就像我们Go语言的委托模式的那个Undo示例一样。Undo这个事是我们想要解决的问题，是Logic，但是Undo的流程是控制。
- 就像我们面向对象中依赖于接口而不是实现一样，接口是对逻辑的抽象，真正的逻辑放在不同的具现类中，通过多态或是依赖注入这样的控制来完成对数据在不同情况下的不同处理。

如果你再仔细地结合我们之前讲的各式各样的编程范式来思考上述这些概念的话，你是否会觉得，所有的语言或编程范式都在解决上面的这些问题。也就下面的这几个事。

- Control是可以标准化的。比如：遍历数据、查找数据、多线程、并发、异步等，都是可以标准化的。
- 因为Control需要处理数据，所以标准化Control，需要标准化Data Structure，我们可以通过泛型编程来解决这个事。
- 而Control还要处理用户的业务逻辑，即Logic。所以，我们可以通过标准化接口/协议来实现，我们的Control模式可以适配于任何的Logic。

上述三点，就是编程范式的本质。

- 有效地分离Logic、Control和Data是写出好程序的关键所在！
- 有效地分离Logic、Control和Data是写出好程序的关键所在！
- 有效地分离Logic、Control和Data是写出好程序的关键所在！

我们在写代码当中，就会看到好多这种代码，会把控制逻辑和业务逻辑放在一块。里面有些变量和流程是跟业务相关的，有些是不相关的。业务逻辑决定了程序的复杂度，业务逻辑本身就复杂，你的代码就不可能写得简单。

Logic，它是程序复杂度的的下限，然后，我们为了控制程序，需要再搞出很多控制代码，于是Logic+Control的相互交织成为了最终的程序复杂度。

把逻辑和控制混淆的示例

我们来看一个示例，这是我在leetcode上做的一道题，这是通配符匹配，给两个字符串匹配。需求如下：

```
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
```

现在你再看看我写出来的代码：

```
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';
}
```

我也不知道我怎么写出来的，好像是为了要通过，我需要关注于性能，你看，上面这段代码有多乱。如果我不写注释你可能都看不懂了。就算我写了注释以后，你敢改吗？你可能连动都不敢动（哈哈）。上面这些代码里面很多都不是业务逻辑，是用来控制程序的逻辑。

业务逻辑是相对复杂的，但是控制逻辑跟业务逻辑交叉在一块，虽然代码写得不多，但是这个代码已经够复杂了。两三天以后，我回头看，我到底写的什么，我也不懂，为什么会写成这样？我当时脑子是怎么想的？我完全不知道。我现在就是这种感觉。

那么，怎么把上面那段代码写得更好一些呢？

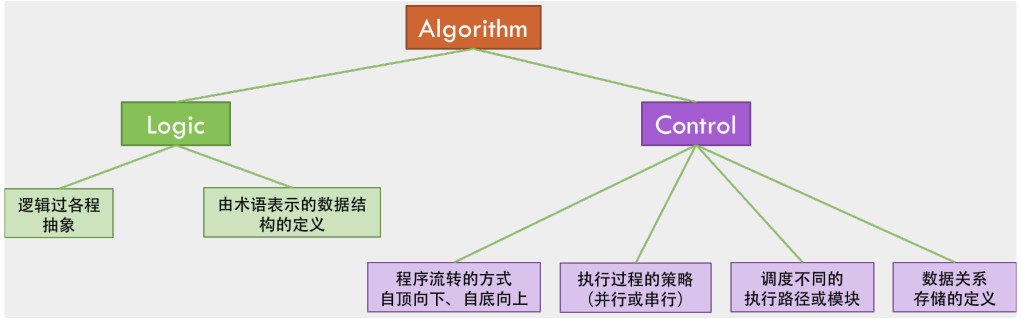
- 首先，我们需要一个比较通用的状态机（NFA，非确定有限自动机，或者DFA，确定性有限自动机），来维护匹配的开始和结束的状态。这属于Control。
- 如果我们做得好的话，还可以抽像出一个像程序的文法分析一样的东西。这也是Control。
- 然后，我们把匹配 * 和 ? 的算法形成不同的匹配策略。

这样，我们的代码就会变得漂亮一些了，而且也会快速一些。

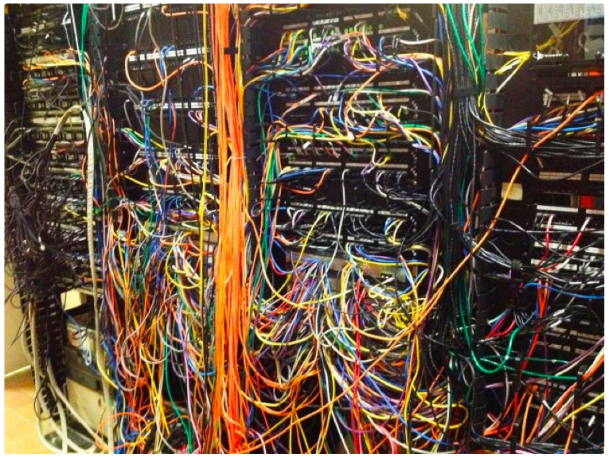
这里有篇正则表达式的高效算法的论文Regular Expression Matching Can Be Simple And Fast，推荐你读一读，里面有相关的实现，我在这里就不多说了。

这里，想说的程序的本质是Logic+Control+Data，而其中，Logic和Control是关键。注意，这个和系统架构也有相通的地方，逻辑是你的业务逻辑，逻辑过程的抽象，加上一个由术语表示的数据结构的定义，控制逻辑跟你的业务逻辑是没关系的，你控制它执行。

控制一个程序流转的方式，即程序执行的方式，并行还是串行，同步还是异步，以及调度不同执行路径或模块，数据之间的存储关系，这些和业务逻辑没有关系。



如果你看过那些混乱不堪的代码，你会发现其中最大的问题是我们把这Logic和Control纠缠在一起了，所以会导致代码很混乱，难以维护，Bug很多。绝大多数程序复杂的原因就是这个问题。就如同下面这幅图中表现的情况一样。



再来一个简单的示例

这里给一个简单的示例。

下面是一段检查用户表单信息常见的代码，我相信这样的代码你见得多了。

```
function check_form_x() {
    var name = $('#name').val();
    if (null == name || name.length <= 3) {
        return { status : 1, message: 'Invalid name' };
    }

    var password = $('#password').val();
    if (null == password || password.length <= 8) {
        return { status : 2, message: 'Invalid password' };
    }

    var repeat_password = $('#repeat_password').val();
    if (repeat_password != password.length) {
        return { status : 3, message: 'Password and repeat password mismatch' };
    }
```

```
    var email = $('#email').val();
    if (check_email_format(email)) {
        return { status : 4, message: 'Invalid email' };
    }

    ...

    return { status : 0, message: 'OK' };

}
```

但其实，我们可以做一个DSL+一个DSL的解析器，比如：

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

var r = check_form(meta_create_user);
```

这样，DSL的描述是“Logic”，而我们的 check_form 则成了“Control”，代码就非常好看了。

小结

代码复杂度的原因：

- 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呢？我们可以使用下面的这些技术来解耦。

- 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

这就是编程的本质：

- 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）

以下是《编程范式游记》系列文章的目录，方便你了解这一系列内容的全貌。这一系列文章中代码量很大，很难用音频体现出来，所以没有录制音频，还望谅解。

- 编程范式游记（1）- 起源
- 编程范式游记（2）- 泛型编程
- 编程范式游记（3）- 类型系统和泛型的本质
- 编程范式游记（4）- 函数式编程
- 编程范式游记（5）- 修饰器模式
- 编程范式游记（6）- 面向对象编程
- 编程范式游记（7）- 基于原型的编程范式
- 编程范式游记（8）- Go 语言的委托模式
- 编程范式游记（9）- 编程的本质
- 编程范式游记（10）- 逻辑编程范式
- 编程范式游记（11）- 程序世界里的编程范式

楊_宵夜

2018-02-13

Roger的留言，本人不才，想试着从"Control标准化"和"代码可重用"的角度来回答下:
仔细看那个检查表单信息的例子，叫做check_form_XXX()，是针对特定的一个form的，那么自然而然会有check_form_YYY()和check_form_ZZZ()等等...

所以说，这个form校验例子中，
1. 最简单的Control部分就是遍历表单fields.
2. 然后，虽然是不同的field，但相同的type是做相同逻辑的校验；
3. 如果还想提供"将整个form拆成不同的part，用并发来遍历"这种Control的话，抽象出来的check_form()函数还可以提供并发的版本；

那么单单是以上3点，全部都是"可标准化"的，并且"可重用"的，并不影响<业务的logic>；

那么，当有了check_from()这个Control之后，真正决定业务的<业务的logic>，有:
1. 每个field分别是什么type? 是text? 是password? 还是email?
2. 每个field的最低长度是多少? 类似的还有每个field的最大长度?
3. 等等...

以上的问题，决定了这个field通过校验的条件是什么?
而这个条件是无法"标准化"的，因为一个复杂系统的每个form的field不可能是一模一样的；
所以这些"条件"，就由陈老师写出的DSL来提供；

因此最后就变成了，check_form()提供一套"标准"来校验每个表单，而每个表单只需要告诉check_form()说："我有这些东西，你帮我校验一下"；而这样的说法，又有了些委托模式的味道了；

总而言之，个人愚见：Control和Logic部分的一个肉眼可见的界线就是：是否可以标准化??
因本人较熟悉Java，再扩展来说，全局的工具类就是一种全局Control，而一个类中的private方法大致可以认为是这个类中的Control.（仅为一种思路，未经推敲）；

SamZhou

2018-02-10

是处理什么（logic），怎么做（control），沟通方式（数据结构）？

nanquanmama

2018-02-08

焕然大悟

Roger

2018-02-09

还是不太明白 控制和逻辑的关系，检查表单的那个例子从我的理解来看已经是逻辑了。

帅广应s

2018-02-08

这段时间，正好在用python写一个从hive查数据，自动发邮件给运营产品的系统。借鉴了hadoop yarn的状态机后,整个逻辑结构清晰多了。但是也只是知道这样做可以解决问题，看了这篇文章后知道了为什么得这么做。感觉自己又上升了一个level .....

mingshun

2018-05-18

自从写业务的这几年来，做得最多的就是分离 Logic 和 Control。无论是编写新代码还是重构旧代码，都是从这个方向努力，目标是写出让团队里每个人都能轻松看得懂的代码。也用过许多语言，像 C、C++、Java、Golang、Lua、JS、Ruby、Elixir、Red.....虽然思维模式和习惯玩法各异，但编程的本质是一样的。毕竟代码写出来是给人看的。如果人都很难看懂，又谈何优化和改进代码？！

作者回复

2018-05-22

赞��

favorlm

2018-05-05

老师，您好，实际的编程工作中，我很想融合，但是发现处处都是方法，无法选择

Chris

2018-02-09

理解并认识到编程的本质，才真正可能跳出代码搬运工的圈子，感谢老师!

恩言

2018-02-08

言不由衷的喜欢啊，真的好。

joe2

2018-07-10

还是不明白，不知道如何区分好控制和逻辑。 如何区分逻辑和控制？有理解的朋友可以解说一下吗？

网虫子

2018-06-05

受益匪浅

saiyn

2018-06-03

我还以为会给出一开始代码的优化后的代码呢

WTF

2018-05-13

曾都梦想仗剑走天涯，哦不，是精通一门语言，然后一通百通吃遍所有语言。可以结合王垠的<如何掌握所有的程序语言>一起看看。

http://www.yinwang.org/blog-cn/2017/07/06/master-pl

Sammi

2018-05-13

很有用，以后工作中会提醒自己去思考哪些是logic，哪些是control，然后把更多精力放在control上，毕竟自己对代码的性能更感兴趣

Harry

2018-05-11

醍醐灌顶的一篇文章

候鸟归来的季节

2018-04-17

高手写的代码更加接近逻辑

不记年

2018-04-14

皓叔。。。怎么理解接口呢。。。。以前对接口自己感觉还挺了解的，现在感觉越来越不理解了

fsj

2018-04-06

逻辑和控制不太好理解，粗浅的觉得，控制是逻辑的实现，逻辑是唯一的，控制是多样的，以后再慢慢体会。
通过检查表单的例子，学会一个技能，以后写业务逻辑，先思考能不能通过某种方式（比如DSL，状态机等）完整的表达逻辑，然后在写控制，尽量避免逻辑被分散在多个控制之中

不记年

2018-04-05

通过接口将控制逻辑与业务逻辑分离，是这样吗

vvsuperman

2018-03-12

有没推荐阅读，逻辑和控制的意思实在不太懂

JK.Ryan

2018-02-16

确保是，放在大的架构里面其实也是这样的模式，开放闭合，单一权限原则等都可以作为control，而其他偏业务的逻辑都可以当作logic，其实都是相通的