

# *Fuck Your Brain*

## (一)

有时候程序员就特么一群疯子。

无论你能想到想不到，很多东西在他们的脑袋里面产生然后产出的时候，你就会被震惊了；

Brainfuck 也是这么奇葩的一个东西。

这是一个程序设计语言：没错；但是，只有八个关键字（符）的语言。读起来颇有机器语言的感觉。

`+ - [ ] > < , .`

当用到等宽体的时候才能够没有歧义的把他们全部显示出来，而我也不得不在文章里面用中文标点，以免弄混。

好吧，介绍一下吧：

他只需要一个可以索引元素的字节数组（堆栈？），然后一个能够指向该数组元素的指针（类似于 C 语言中数组中的下标）；一个输入流用来获取字节输入，一个输出流用来进行字节输出。然后通过判断当前指针位置的值是否非零来进行流程控制，所以需要个保存状态的位和一个保存指针位置的寄存器。好吧，这就是全部。

然后下面是各个操作符的含义：

`+ / -` 当前指针指向的值加（减）一；

`> / <` 当前指针位置向前（后）移动一；

`, / .` 对当前指针指向的值获取输入（输出）；

`[ / ]` 这个是最蛋疼的：当前指针指向的值不为零时，执行其内部的指令，否则，跳过执行。

看不懂？那就来试验下吧：

一个最简单的：获取输入的值并输出：

`, [ . , ]`

蛋疼死你.....这个真的只是简单的。首先，程序执行会获得一个输入，保存到当前指针指向的空间里面，然后进入循环；循环体中，首先要对当前值进行输出，然后再一次获得输入，然后循环：理论上，只要我们输入一个空字符把当前位置置为零，才有可能退出循环。所以，这个程序就产生这样一个结果：你输入什么，他就输出什么。具体示例，大家自己实现了以后看一下吧。

OK，第二个。

我们先不讨论输出 `Hello, world` 这种东西，要知道，这太复杂了。先打印出 0~9 吧：

`++++++ [->+++++++<]+++++++ [->. +<]`

第一次看到这个我想到了鱼骨图案。好吧，来看一下，他要输出 0~9，那么就要通过 ASCII 码来搞定，0 的 ASCII 为 48，而第一个循环的作用就是要把第二数据空间置为 48（第一数据空间减六次，每次减的过程中第二数据空间增加 8）；然后一个计数循环把 0~9 的 ASCII 码输出。

那么我们的第三个程序就来了嘛：

`+++++++ [->+++++++<]+++++ [->>+++++<<]>>+ [-<+.>]`

没错哦没错，你应该懂得这个程序是做什么的了吧。

## (二)

这个毕竟是叫做 Brainfuck 的，邪恶点想想，你的 Brain 被 fuck 的感觉是什么样子的。

想死？不可能，这个还没有这么大的威力；想驯服他？好吧，我希望你能做到。

那么，既然我们现在仍然纠结在这个地方，不如针对他来做点什么嘛。

我想说的就是，当你被这个标题吸引过来，并且又看完了第一章的时候，是不是有种想试一试的感觉？那么，不要着急的，我们马上就来了。

既然想亲手写一下代码并且要执行通过，那我们就要自己尝试下写一个解释器嘛。没错，解释器，只有这样才有可能理解他的内部结构和具体实现嘛。

OK, Let's do it.

首先，作为一个解释器，要有一个专门的变量来保存每一个指令（instruction）；然后嘛，还要专门的变量来存放数据（data）；还有指针（pointer）、当前指令、当前数据空间。那，以下就是我们需要的吧：

```
char is[65536],ir;
unsigned char ds[65536],dr;
int ip,dp;
```

这大概就是 we 整个运行时所需要的全部变量了，如果有其他的，我们会在后面给出定义；但是，有没有发现一个比较蛋疼的地方：当前变量和当前指针如果要这样定义，每次调用的时候都仍然要赋值的。所以嘛，要想方法解决了：

```
#define ir is[ip]
#define dr ds[ip]
```

有没有一种被 Brainfuck 了的感觉？没错，就这样嘛。

然后，当我们对其进行定义以后就要进行代码的解释工作了。假设有下面一行代码：

```
,[.,]
（即上一章的程序1）
switch(ir){
...          //对当前指令进行解析
case ',':dr=getchar();break;
case '.':putchar(dr);break;
...
}
```

没错哦，这就是 I/O 的设计，远比你想象的简单。

好嘛，考虑下面一个问题：

```
++++++[->+++++++<]+++++++[->.+<]
```

输出 0~9，输出我们已经实现了，那么接下来是移动数据指针和对数据进行增减操作。

```
case '+':dr++;break;
case '-':dr--;break;
case '>':dp++;break;
case '<':dp--;break;
```

尽情的笑吧孩纸，你距离完成不远了。

我们暂且先不去想流程控制怎么完成，因为这个要涉及到指令的执行顺序，好嘛，我们先看一看吧，在 switch 外用个 ip++，然后再加一个循环语句迫使他一个个的按顺序执行指令。

那么，现在的问题是，我们该怎么使用流程控制。

因为我们要实现指令跳转的。

好嘛，首先我们要想想嘛，要有一个变量来保存一个指针地址，然后通过这个地址让我们去过去找到原始的位置嘛。OK，我们暂时先叫他 ic 吧 (Instruction Cache)。

```
case '[':(dr?ic=ip:ic=ic);break;
```

```
case ']':(dr?ip=ic:ip=ip);break;
```

嗯，没错嘛，这应该就是你过得那样的吧。

OK,Let's run it.

但你会发现，很多时候你得不到你想要的那些结果。

你或许会问：why!!

那么我们就看下面一段程序嘛：

```
++++++[->++++[->++<]<]+++++++[->>.<<<]
```

嗯，没错，程序中有一个嵌套的循环。仔细想想应该就会知道问题出在哪儿了：一个变量用来保存地址是不够的，而是需要一个栈 (stack) 来存放——既符合 LIFO (后进先出) 的执行方式，又能够保证每一个循环/跳转语句都能够较为准确的执行。

OK，看吧，当你能够确定这一点的时候，整个系统将就快完成了。

#### 【注释】

1.关于 LIFO: Last In, First Out 的缩写，堆栈 (Stack) 的基本特性，使得某些嵌套式的结构能够比较简单的实现。

2.由于用词方式问题，有些内容可能表述不同，但其本质上所表示的其实仍然是同一个内容：或许和主流表述会有差异，理解方面造成的不便希望能够谅解。

3.关于示例代码和伴随其一起的实现代码的内容上的差异：一方面，前面有部分内容是早期编写，另外为了规范代码，实现中的命名等做了一定的修改，相对来说会容易理解些。示例代码中的 ir、is 等等之类简略命名全都使用了完整的 instruction\_register, instruction\_stack 等来代替。

4.示例代码和实现代码都在 GCC 3/4下编译通过。

License: CC by-nc-sa 2.0.

Code License: MIT License.

Author: KPSN.Leo.