链滴

《Let'"'"'s Build a Simple Interpreter》系列文章翻译：Part 1

作者：StephenZhang

原文链接：https://ld246.com/article/1597760984099

来源网站：链滴

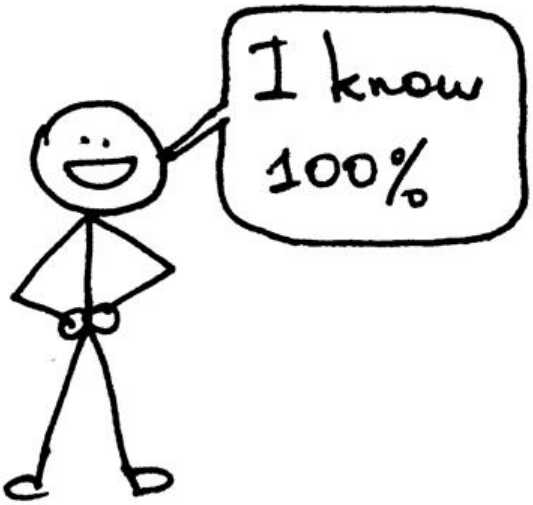
如果你不知道编译器如何工作，那你也不清楚计算机如何工作；如果你不能完全确定你是否知道编译如何工作，那你一定不清楚编译器是如何工作的 —— Steve Yegge

想一想这句话。无论你在软件开发方面是菜鸟还是经验丰富，这都是显而易见的：如果你不知道解释或者编译器如何工作，那你肯定不清楚计算机是如何工作的。

所以，你到底清楚吗？我的意思是说，你完全确定你知道它们的工作方式吗？如果你的回答是“不”那么这个系列的文章将对你有所帮助。

如果你不知道而且对此抱有好奇心的话，请看下面。

不必担忧。如果你持续跟进这个系列的文章而且理解文章的示例，并且和我一起完成一个解释器或者译器的构建，你将可以理解它们的工作原理。你也可以成为一个自信的IT从业者[1]，至少我希望如此。



为什么你应该学习一些关于解释器或者编译器的知识？我认为有如下三点原因。

1. 为了构建一个解释器或者编译器，你必须掌握许多技术性的技巧并在这个过程中应用它们。完成这一个工作，会帮助你提升这些技巧，让你成为一个更优秀的软件开发者。而且，你会发现这些技巧在写任何软件时都适用，而不仅仅适用于编译器或者解释器；

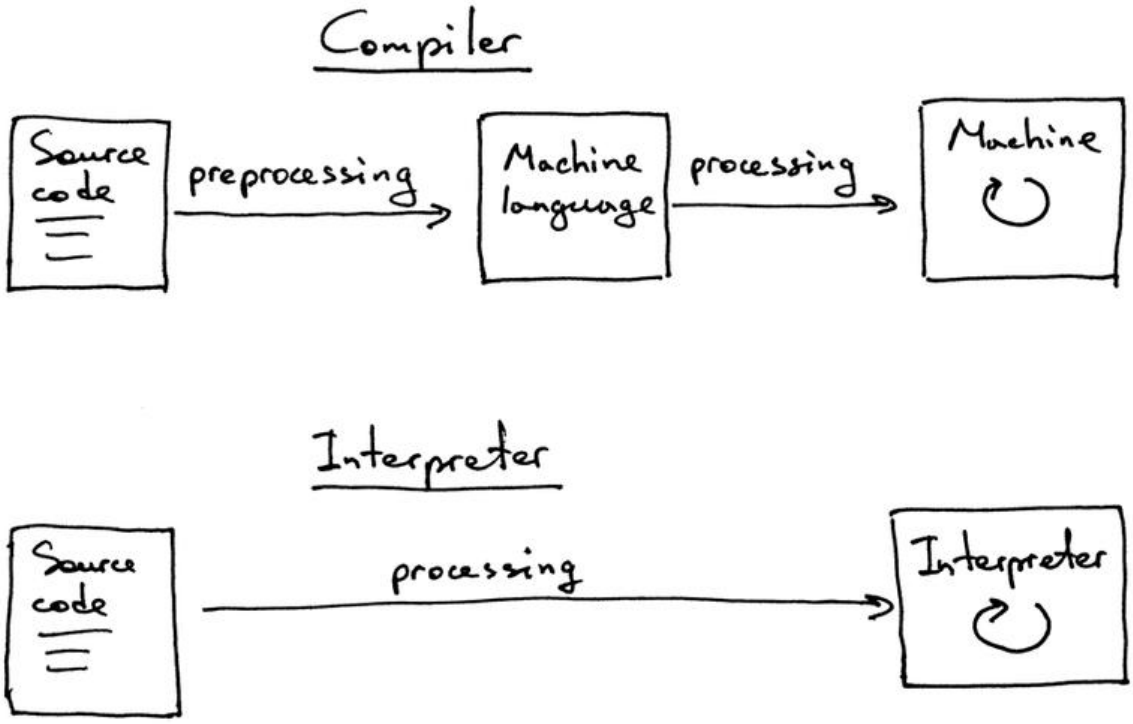
2. 你真的对计算机的工作原理感到好奇。解释器或编译器看起来就像魔法一样，而你可能对这些魔法到不舒服。你也许想揭开构建解释器或者编译器过程中的神秘面纱，理解它们的原理，并掌握一切；

3. 你也许想创建属于自己的编程语言或者某个特定领域的语言。如果你当真这么做了，那么你需要为创建一个配套的解释器或者编译器。最近人们对新的编程语言重新产生了兴趣。你可以从Elixir、GoLang和Rust这些语言中可以窥见这个趋势的一斑。

好吧，但是解释器或者编译器中到底有什么呢？

编译器或解释器的目标是将某些高级语言的源程序转译为其他的形式。这个描述相当模糊，对吧？再受一会，你可以在本系列后续的文章中切实了解到源程序到底被转译为了什么。

现在你也许在好奇编译器和解释器的区别究竟是什么。回答这个问题，也是本系列文章的目的之一：果转译器把源程序翻译为机器语言，那么它就是编译器；如果转译器直接处理和执行源程序，而不是先翻译为机器语言，那么它就是解释器。显然它们的处理过程如下图：



我想现在你应该相信自己是真的想学习如何构建解释器和编译器了。你能在本系列文章中获得什么？

这有一个约定[2]，你将和我一起构建一个简单的解释器，用于Pascal语言的一个子集。当这个系列结时，你将会得到一个可以正常工作的Pascal解释器和一个源码级别的调试器，就像Python中附带的pd一样。

你也许会问，为什么选择Pascal？其中一个原因是，它并不是我为了这个系列而编造的语言：Pascal一门真正的编程语言，具有很多重要的编程语言结构。而且很多旧而有用的CS书籍使用Pascal编写示（在我的理解中，这并不是选择它进行编译器构建的重要理由，但是我想学习一种非主流的语言是一很好的改变）。

这是一个使用Pascal编写的阶乘函数的示例，你应该能够使用自己的解释器去解释执行它并且使用你自创建的、交互式的源码级别的调试器调试它：

```
program factorial;

function factorial(n: integer): longint;
begin
   if n = 0 then
      factorial := 1
   else
      factorial := n * factorial(n - 1);
end;
```

```
var
   n: integer;

begin
   for n := 0 to 16 do
      writeln(n, '! = ', factorial(n));
end.
```

我们将用python实现这个解释器，但是你可以使用任意语言实现它，因为实现解释器或者编译器并依赖特定的语言。好了，让我们继续向下看。预备——走起！

你的第一次尝试可以从一个简单的算术表达式的解释器开始，换句话说，计算器。今天的小目标是这的：编写一个可以处理两个数字的加法，例如"3+5"这样算式的计算器。这是你的计算器，哦不，解器的源代码：

```
# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER, PLUS, EOF = 'INTEGER', 'PLUS', 'EOF'
class Token(object):
    def __init__(self, type, value):
        # token type: INTEGER, PLUS, or EOF
        self.type = type

        # token value: 0, 1, 2. 3, 4, 5, 6, 7, 8, 9, '+', or None
        self.value = value

    def __str__(self):
        """String representation of the class instance.
        Examples:
            Token(INTEGER, 3)
            Token(PLUS '+')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()

class Interpreter(object):
    def __init__(self, text):
        # client string input, e.g. "3+5"
        self.text = text

        # self.pos is an index into self.text
        self.pos = 0

        # current token instance
        self.current_token = None
```

```
def error(self):
    raise Exception('Error parsing input')


def get_next_token(self):
    """Lexical analyzer (also known as scanner or tokenizer)
    This method is responsible for breaking a sentence
    apart into tokens. One token at a time.
    """
    text = self.text

    # is self.pos index past the end of the self.text ?
    # if so, then return EOF token because there is no more
    # input left to convert into tokens
    if self.pos > len(text) - 1:
        return Token(EOF, None)

    # get a character at the position self.pos and decide
    # what token to create based on the single character
    current_char = text[self.pos]

    # if the character is a digit then convert it to
    # integer, create an INTEGER token, increment self.pos
    # index to point to the next character after the digit,

    # and return the INTEGER token
    if current_char.isdigit():
        token = Token(INTEGER, int(current_char))
        self.pos += 1
        return token

    if current_char == '+':
        token = Token(PLUS, current_char)
        self.pos += 1
        return token

    self.error()

def eat(self, token_type):
    # compare the current token type with the passed token
    # type and if they match then "eat" the current token
    # and assign the next token to the self.current_token,
    # otherwise raise an exception.
    if self.current_token.type == token_type:
        self.current_token = self.get_next_token()
    else:
        self.error()


def expr(self):
    """expr -> INTEGER PLUS INTEGER"""
    # set current token to the first token taken from the input
    self.current_token = self.get_next_token()
```

```
        # we expect the current token to be a single-digit integer
        left = self.current_token
        self.eat(INTEGER)

        # we expect the current token to be a '+' token
        op = self.current_token
        self.eat(PLUS)

        # we expect the current token to be a single-digit integer
        right = self.current_token
        self.eat(INTEGER)

        # after the above call the self.current_token is set to
        # EOF token
        # at this point INTEGER PLUS INTEGER sequence of tokens
        # has been successfully found and the method can just
        # return the result of adding two integers, thus
        # effectively interpreting client input
        result = left.value + right.value
        return result

def main():
    while True:
        try:
            # To run under Python3 replace 'raw_input' call
            # with 'input'
            text = raw_input('calc> ')
        except EOFError:
            break

        if not text:
            continue

        interpreter = Interpreter(text)
        result = interpreter.expr()
        print(result)



if __name__ == '__main__':
    main()
```

将上面的代码保存为 calc1.py或者从我的GitHub上 下载。在你对这份代码深入探究之前，先在命令中运行这个程序，观察到底会发生什么。享受它吧！下面是在我的笔记本电脑上执行该程序的一个样（如果你想使用python3运行它，将其中的 raw_input()换做 input()即可）：

```
$ python calc1.py
calc> 3+4
7
calc> 3+5
8
calc> 3+9
12
calc>
```

为使你的简单计算器正常工作而不至于抛出异常，你的输入需要遵循以下确定的规则：

- 仅能输入只有个位的整形数字
- 现在只支持进行加法运算
- 在输入中任何地方不能出现空格

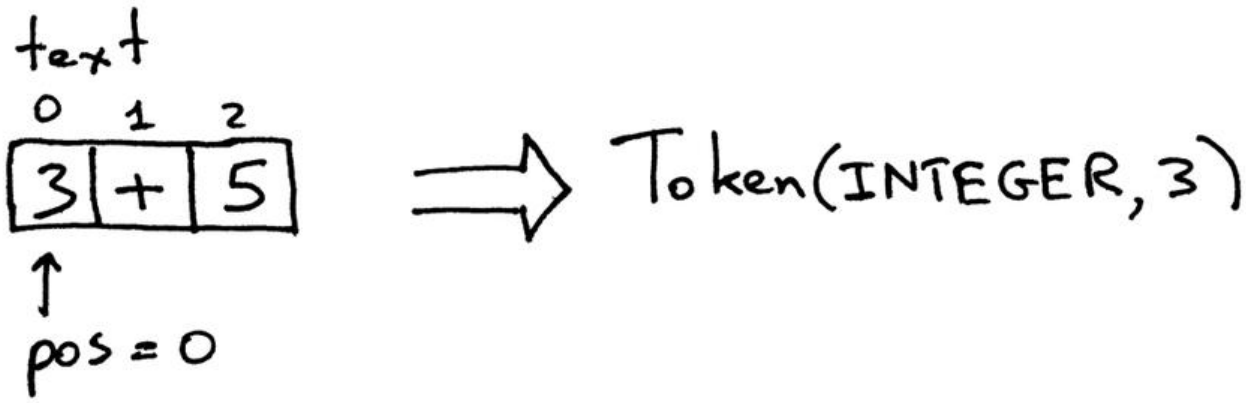
这些限制是为了使计算器更简单，不必担心，你马上可以使它处理一些更复杂的情况。

好，现在让我们深入了解一下你这个解释器是如何工作的，以及它为什么可以计算算术表达式。

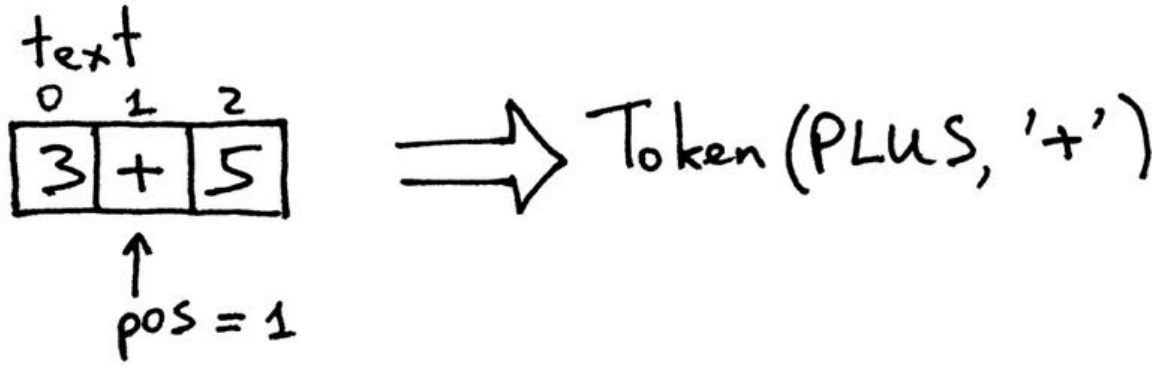
当你在命令行中输入"3+5"这个表达式时，解释器得到的是一个形如 '3+5'的字符串。为了使解释器正理解读取到该字符串后需要做什么，你首先需要将该字符串打散成一个个组件，这些组件被称为标。一个标记可以看做带有类型和值的对象。例如，对于字符串 '3'，其对应的标记的类型为 INTEGER 而它的值为整形 3。

将输入字符串打散成为标记（Token，下文仍旧使用英文）的处理过程称为词法分析。所以，你的解器需要首先读取输入字符串并将其转换为Token流。在解释器中，进行这一工作的部分被称为词法分器，或者简称为词法器（lexer）。你也许会见到词法器的其他名字，例如扫描器（scanner）或者标化器（tokenizer），它们具有相同的含义：解释器或者编译器中将输入字符串转换为Token流的部（或组件）。

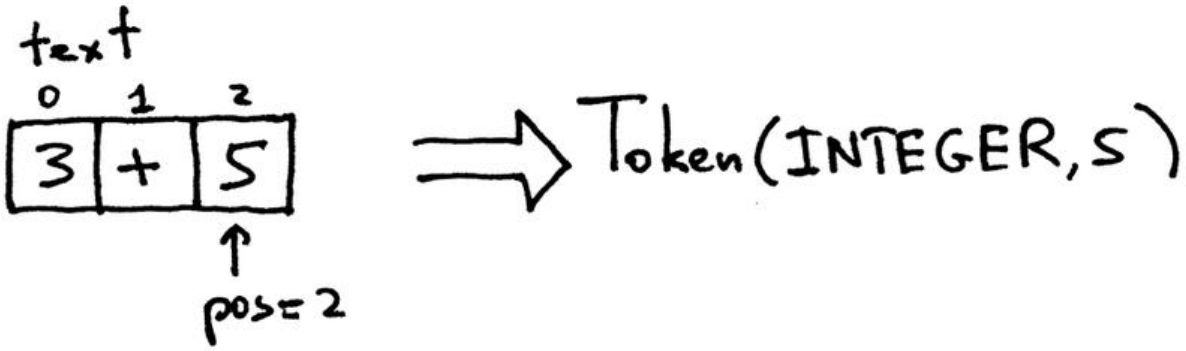
类 Interpreter中的方法 get_next_token()就是你的词法器。你每次调用它的时候，新的Token将会被传递给解释器的字符串中创建并返回，让我们更详细的观察一下方法本身，看看它到底是是如何进将字符串转换为Token的工作的。输入字符串被储存在变量 text中，而 pos保存了遍历字符串的索引（将字符串视作单个字符组成的数组）。pos被初始化为 0，此时它指向字符 '3'。于是 get_next_token()方法（根据 pos读取字符）首先检查这个字符是不是数字，如果是，pos将会加一，然后返回一个 token的实例——这个实例的类型是 INTEGER，而值被设置为字符 '3'的值，即3：



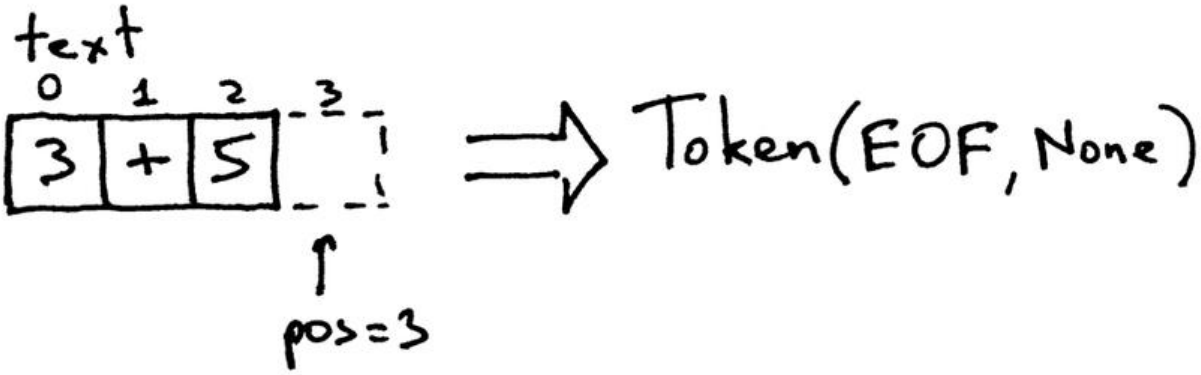
现在，pos指向字符串中的 '+'。当你下次调用那个方法时，它会判断 pos指向的字符是否为数字，如不是，继续判断这个字符是不是+。结果，该方法会使 pos加一，并返回一个新创建的 token实例，的类型是 PLUS，值为 '+'：



当 pos指向 '5'的时候，该方法做的事情和上面相同，返回的 toke实例的类型是 INTEGER，值是5:



当索引值 pos的值越过了字符串 '3+5'的结束符以后，每次调用方法 get_next_token()都将会返回类为 EOF，值为空的 token实例：



将词法器独立出来，你可以观察到它是如何工作的：

```
>>> from calc1 import Interpreter
>>>
>>> interpreter = Interpreter('3+5')
>>> interpreter.get_next_token()
Token(INTEGER, 3)
>>>
>>> interpreter.get_next_token()
Token(PLUS, '+')
```

```
>>>
>>> interpreter.get_next_token()
Token(INTEGER, 5)
>>>
>>> interpreter.get_next_token()
Token(EOF, None)
>>>
```

现在，你的解释器可以访问由输入字符串生成的Token流，并且要对Token流做些额外的操作：它应在 get_next_token()方法获取的Token流中搜索类似于 INTEGER -> PLUS -> INTEGER的语法结构也就是说，它尝试查找到一个“整数、加号、整数”的Token序列。

负责搜索和解释的方法是 expr()，它将会验证Token序列是否完全契合上文中的理想的Token序列。果验证通过，它会把左右两个操作数相加，生成计算结果，因此，对你传入的算术表达式的一次成功执行就完成了。

expr()方法本身使用了辅助方法 eat()来验证传入 eat()方法的Token类型是否匹配类 Interpreter的成员urrent_token的类型。如果二者相匹配，eat()方法将会获取下一个Token（通过调用 get_next_token)方法），并分配给 current_token变量，这样就能有效地“吃掉”当前的Token并在流中迅速推进逻上的指针的位置。如果在Token流中找不到期待的序列，那么 eat()方法就会抛出异常。

让我们回顾一下你的解释器在计算算术表达式时到底做了什么：

- 解释器接收一个字符串，这里是 '3+5'
- 解释器调用 expr()方法在词法器get_next_token()返回的Token流中寻找期待的语法结构。在确到该结构后，它把输入解释为将两个整形Token的值相加，因为它很清楚自己要做的事情就是把两个形（3和5）。

祝贺你自己吧！你刚刚学习了如何构建一个非常原始[3]的解释器！

现在是时候做一些训练了！

你肯定不会以为读完这篇文章就足够了，对吧？好，现在自己动手，做做下面的训练：

1. 修改代码，使之支持多位数字的输入，例如"12+3"
2. 添加一个可以跳过空白字符的方法，使你的计算器可以处理诸如 '12 + 3'这样带有空白字符的输入
3. 修改代码，替换 +为-，即可以计算类似"7-5"这样的减法表达式

检查一下你的理解：

1. 解释器是什么？
2. 编译器是什么？
3. 它们的区别何在？
4. Token是什么？
5. 把输入打散为Token的处理过程叫做什么？
6. 解释器的哪一部分执行了词法分析的工作？
7. 这一部分的其他通用名称是什么？

在我结束这篇文章之前，我真的希望你可以致力于学习解释器和编译器。而且我希望你现在就去学习而不是将它作为一句口号[4]，不要等待。如果你只是迅速浏览了本文，那么请重新仔细阅读它；如果阅读了本文但是没有进行文末的训练，那么现在去完成它；如果你完成了本文的一部分任务，那么就束掉余下的任务，明白了吧。而且，你知道吗？签署承诺书并开始你的学习之旅吧！

我，_____，拥有健全的思想和身体，今天在此承诺我将致力于学习解释器和编译器的知识，直到百分之百的了解它的工作原理。

签字：_____ 日期：_____

签上名字，写上日期，把它放在你每天都能看到的地方，以确保你坚持你的承诺。请牢记承诺的定义：

“承诺是你说要做的事，在你说它的心情离开你很久之后才去做。” —— Darren Hardy

好了，今天就是这些内容了，下一篇文章中你将看到如何扩展你的计算器，使它支持更多的算术表达。现在不懂也没关系。

如果你迫不及待地想看第二篇文章，迫不及待地想开始深入了解解释器和编译器，下面是我推荐给你一些书，它们可能会对你有所帮助:

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers)
2. Writing Compilers and Interpreters: A Software Engineering Approach
3. Modern Compiler Implementation in Java
4. Modern Compiler Design
5. Compilers: Principles, Techniques, and Tools (2nd Edition)

1. 此处原文为camper，联系上下文，如果译作“露营者”似乎不妥，此处采取意译（猜的）
2. 约定，原文为deal，此处翻译存疑
3. 非常原始，原文为very first，此处存疑
4. 一句口号，原文是put it back banner，即“放回到横幅上”，此处采取意译