

# 第9讲：函数递归

## 目录

1. 什么是递归
2. 递归的限制条件
3. 递归的举例
4. 递归与迭代

## 正文开始

### 1. 递归是什么？

递归是学习C语言函数绕不开的一个话题，那什么是递归呢？

递归其实是一种解决问题的方法，在C语言中，递归就是**函数自己调用自己**。

写一个史上最简单的C语言递归代码：

代码块

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hehe\n");
6      main();//main函数中又调用了main函数
7      return 0;
8  }
```

上述就是一个简单的递归程序，只不过上面的递归只是为了演示递归的基本形式，不是为了解决问题，代码最终也会陷入死递归，导致栈溢出（Stack overflow）。

#### 未经处理的异常

0x7BF20907 (ucrtbased.dll) (test.exe 中)处有未经处理的异常：0xC00000FD: **Stack overflow** (参数：0x00000000, 0x00602000)。

[复制详细信息](#) | [启动 Live Share 会话...](#)

▷ 异常设置

## 1.1 递归的思想：

把一个大型复杂问题层层转化为一个与原问题相似，但规模较小的子问题来求解；直到子问题不能再被拆分，递归就结束了。所以递归的思考方式就是**把大事化小**的过程。

递归中的**递就是递推**的意思，**归就是回归**的意思，接下来慢慢来体会。

## 1.2 递归的限制条件

递归在书写的时候，有2个必要条件：

- 递归存在限制条件，当满足这个限制条件的时候，递归便不再继续。
- 每次递归调用之后越来越接近这个限制条件。

在下面的例子中，我们逐步体会这2个限制条件。

## 2. 递归举例

### 2.1 举例1：求n的阶乘

一个正整数的**阶乘**（**factorial**）是所有小于及等于该数的正整数的积，并且0的阶乘为1。

自然数n的阶乘写作  $n!$ 。

题目：计算n的阶乘（不考虑溢出），n的阶乘就是1~n的数字累积相乘。

#### 2.1.1 分析和代码实现

n的阶乘的公式： $n! = n * (n - 1)!$

代码块

1 举例：

2  $5! = 5 * 4 * 3 * 2 * 1$

3  $4! = 4 * 3 * 2 * 1$

4 所以： $5! = 5 * 4!$

从这个公式不难看出：如何把一个较大的问题，转换为一个与原问题相似，但规模较小的问题来求解的。

n的阶乘和n-1的阶乘是相似的问题，但是规模要少了n。有一种特殊情况是：当  $n=0$  的时候，n的阶乘是1，而其余n的阶乘都是可以通过上面的公式计算。

这样就能写出 n 的阶乘的递归公式如下：

$$\text{Fact}(n) = \begin{cases} 1, & n == 0 \\ n * \text{Fact}(n-1), & n > 0 \end{cases}$$

那我们就可以写出函数Fact求n的阶乘，假设Fact(n)就是求n的阶乘，那么Fact(n-1)就是求n-1的阶乘，函数如下：

代码块

```
1  int Fact(int n)
2  {
3      if(n == 0)
4          return 1;
5      else
6          return n * Fact(n-1);
7  }
```

测试：

代码块

```
1  #include <stdio.h>
2
3  int Fact(int n)
4  {
5      if(n == 0)
6          return 1;
7      else
8          return n * Fact(n-1);
9  }
10
11 int main()
12 {
13     int n = 0;
14     scanf("%d", &n);
15     int ret = Fact(n);
16     printf("%d\n", ret);
17     return 0;
18 }
```

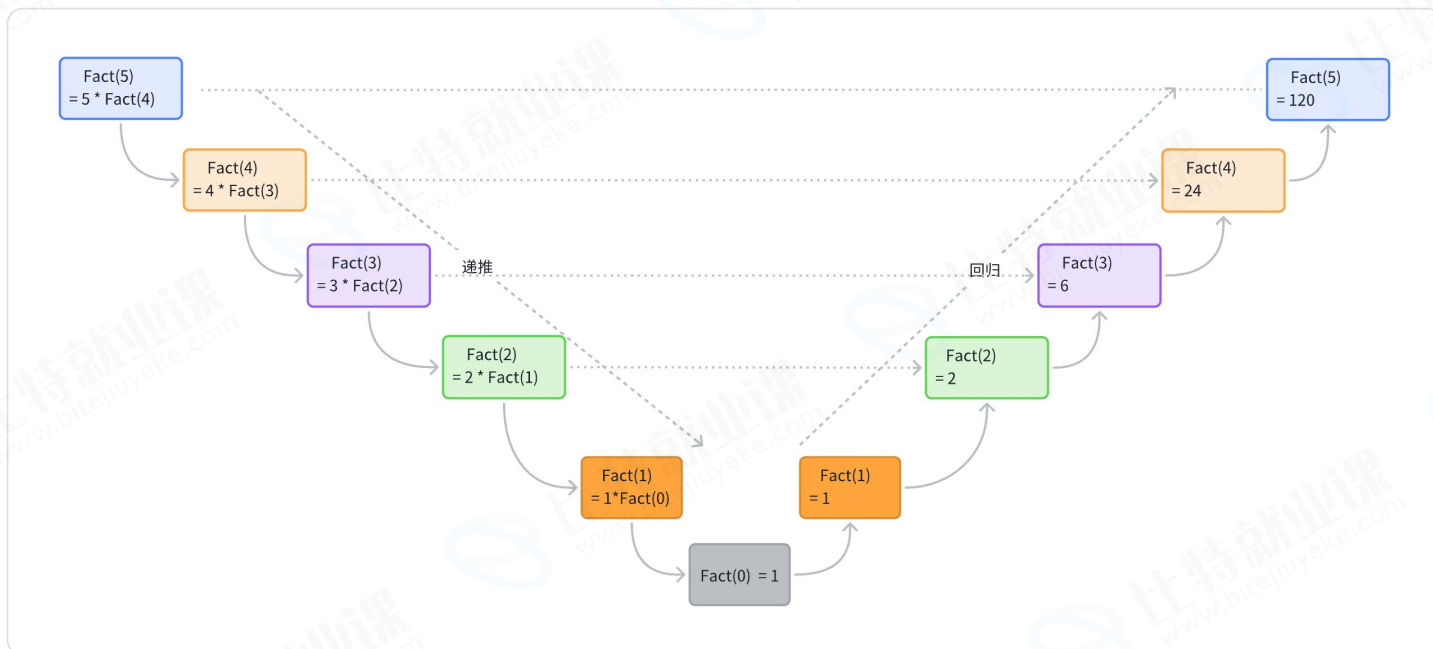
运行结果（这里不考虑n太大的情况，n太大存在溢出）：

5

120

D:\code\2023\test\课件代码测试\Debug\课件代码测试.exe (进程 52252) 已退出，

### 2.1.2 画图推演



## 2.2 举例2：顺序打印一个整数的每一位

输入一个整数m，按照顺序打印整数的每一位。

比如：

输入：1234 输出：1 2 3 4

输入：520 输出：5 2 0

### 2.2.1 分析和代码实现

这个题目，放在我们面前，首先想到的是，怎么得到这个数的每一位呢？

如果n是一位数，n的每一位就是n自己

n是超过1位数的话，就得拆分每一位

1234%10就能得到4，然后1234/10得到123，这就相当于去掉了4

然后继续对123%10，就得到了3，再除10去掉3，以此类推

不断的 `%10` 和 `/10` 操作，直到1234的每一位都得到；

但是这里有个问题就是得到的数字顺序是倒着的

但是我们有了灵感，我们发现其实一个数字的最低位是最容易得到的，通过%10就能得到  
那我们假设有想写一个函数Print来打印n的每一位，如下表示：

代码块

```
1 Print(n)
2 如果n是1234，那表示为
3 Print(1234) //打印1234的每一位
4
5 其中1234中的4可以通过%10得到，那么
6 Print(1234)就可以拆分为两步：
7 1. Print(1234/10) //打印123的每一位
8 2. printf(1234%10) //打印4
9 完成上述2步，那就完成了1234每一位的打印
10
11 那么Print(123)又可以拆分为Print(123/10) + printf(123%10)
```

以此类推下去，就有

代码块

```
1 Print(1234)
2 ==>Print(123) + printf(4)
3 ==>Print(12) + printf(3)
4 ==>Print(1) + printf(2)
5 ==>printf(1)
```

直到被打印的数字变成一位数的时候，就不需要再拆分，递归结束。

那么代码完成也就比较清楚：

代码块

```
1 void Print(int n)
2 {
3     if(n>9)
4     {
5         Print(n/10);
6     }
7     printf("%d ", n%10);
8 }
9
10 int main()
11 {
12     int m = 0;
13     scanf("%d", &m);
```

```
14     Print(m);
15     return 0;
16 }
```

输入和输出结果：



在这个解题的过程中，我们就是使用了大事化小的思路

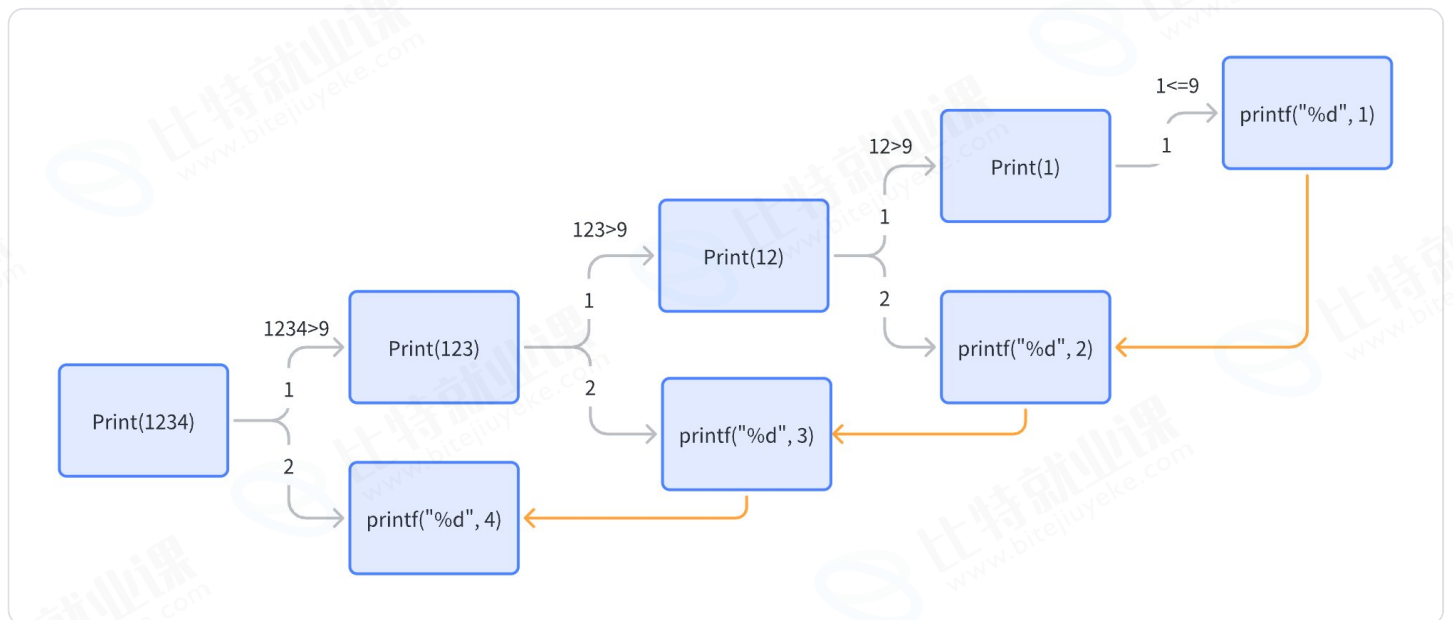
把Print(1234) 打印1234每一位，拆解为首先Print(123)打印123的每一位，再打印得到的4

把Print(123) 打印123每一位，拆解为首先Print(12)打印12的每一位，再打印得到的3

直到Print打印的是一位数，直接打印就行。

### 2.2.2 画图推演

以1234每一位的打印来推演一下



## 3. 递归与迭代

递归是一种很好的编程技巧，但是和很多技巧一样，也是可能被误用的，就像举例1一样，看到推导的公式，很容易就被写成递归的形式：

$$\text{Fact}(n) = \begin{cases} 1, & n == 0 \\ n * \text{Fact}(n-1), & n > 0 \end{cases}$$

代码块

```
1  int Fact(int n)
2  {
3      if(n == 0)
4          return 1;
5      else
6          return n * Fact(n-1);
7  }
```

Fact函数是可以产生正确的结果，但是在递归函数调用的过程中涉及一些运行时的开销。

在C语言中每一次函数调用，都需要为本次函数调用在内存的栈区，申请一块内存空间来保存函数调用期间的各种局部变量的值，这块空间被称为**运行时堆栈**，或者**函数栈帧**。

函数不返回，函数对应的栈帧空间就一直占用，所以如果函数调用中存在递归调用的话，每一次递归函数调用都会开辟属于自己的栈帧空间，直到函数递归不再继续，开始回归，才逐层释放栈帧空间。

所以如果采用函数递归的方式完成代码，递归层次太深，就会浪费太多的栈帧空间，也可能引起栈溢出（stack overflow）的问题。

**注：**关于**函数栈帧**的详细内容，鹏哥录制了视频专门讲解的，下课导入课程，自行学习。

所以如果不想使用递归，就得想其他的办法，通常就是迭代的方式（通常就是循环的方式）。

比如：计算 n 的阶乘，也是可以产生1~n的数字累计乘在一起的。

代码块

```
1  int Fact(int n)
2  {
3      int i = 0;
4      int ret = 1;
5      for(i = 1; i <= n; i++)
6      {
7          ret *= i;
8      }
9      return ret;
```



```
10 }
```

上述代码是能够完成任务，并且效率是比递归的方式更好的。

事实上，我们看到的许多问题是以递归的形式进行解释的，这只是因为它比非递归的形式更加清晰，但是这些问题的迭代实现往往比递归实现效率更高。

当一个问题非常复杂，难以使用迭代的方式实现时，此时递归实现的简洁性便可以补偿它所带来的运行时开销。

### 举例3：求第n个斐波那契数

我们也能举出更加极端的例子，就像计算第n个斐波那契数，是不适合使用递归求解的，但是斐波那契数的问题通过是使用递归的形式描述的，如下：

$$\text{Fib}(n) = \begin{cases} 1, & n \leq 2 \\ \text{Fib}(n-1) + \text{Fib}(n-2), & n > 2 \end{cases}$$

看到这公式，很容易诱导我们将代码写成递归的形式，如下所示：

代码块

```
1  int Fib(int n)
2  {
3      if(n <= 2)
4          return 1;
5      else
6          return Fib(n-1) + Fib(n-2);
7  }
```

测试代码：

代码块

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n = 0;
```

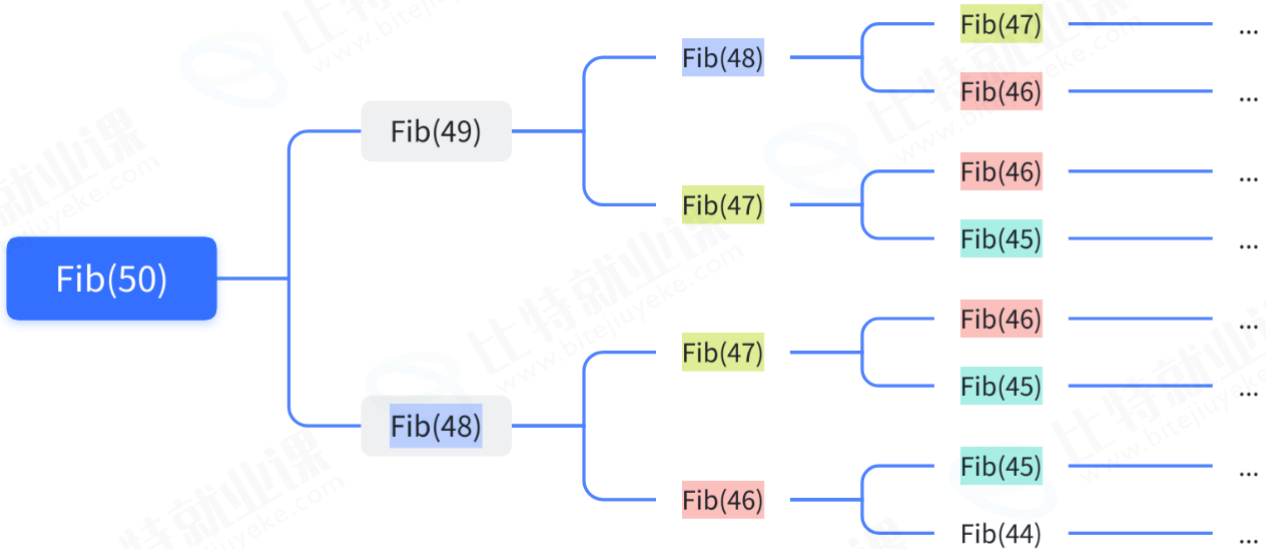


```

6     scanf("%d", &n);
7     int ret = Fib(n);
8     printf("%d\n", ret);
9     return 0;
10  }

```

当我们n输入为50的时候，需要很长时间才能算出结果，这个计算所花费的时间，是我们很难接受的，这也说明递归的写法是非常低效的，那是为什么呢？



其实递归程序会不断的展开，在展开的过程中，我们很容易就能发现，在递归的过程中会有重复计算，而且递归层次越深，冗余计算就会越多。我们可以作业测试：

代码块

```

1  #include <stdio.h>
2  int count = 0;
3
4  int Fib(int n)
5  {
6      if(n == 3)
7          count++; //统计第3个斐波那契数被计算的次数
8      if(n <= 2)
9          return 1;
10     else
11         return Fib(n-1) + Fib(n-2);
12 }
13


```

```

14
15  int main()
16  {
17      int n = 0;
18      scanf("%d", &n);
19      int ret = Fib(n);
20      printf("%d\n", ret);
21      printf("\ncount = %d\n", count);
22      return 0;
23  }

```

输出结果：

 Microsoft Visual Studio 调试控制台

```

40
102334155

count = 39088169

```

这里我们看到了，在计算第40个斐波那契数的时候，使用递归方式，第3个斐波那契数就被重复计算了39088169次，这些计算是非常冗余的。所以斐波那契数的计算，使用递归是非常不明智的，我们就得想迭代的方式解决。

我们知道斐波那契数的前2个数都1，然后前2个数相加就是第3个数，那么我们从前往后，从小到大计算就行了。

这样就有下面的代码：

代码块

```

1  int Fib(int n)
2  {
3      int a = 1;
4      int b = 1;
5      int c = 1;
6      while(n>2)
7      {
8          c = a+b;
9          a = b;
10         b = c;
11         n--;
12     }
13     return c;
14 }

```

迭代的方式去实现这个代码，效率就要高出很多了。

有时候，递归虽好，但是也会引入一些问题，所以我们一定不要迷恋递归，适可而止就好。

递归经常会使用到：树/图遍历、分治算法、回溯算法中，大家在后期学习数据结构和算法的知识时候，再逐步去体会学习。

#### 拓展学习：

- 青蛙跳台阶问题
- 汉诺塔问题

以上2个问题都可以使用递归很好的解决，有兴趣可以研究。

---

完