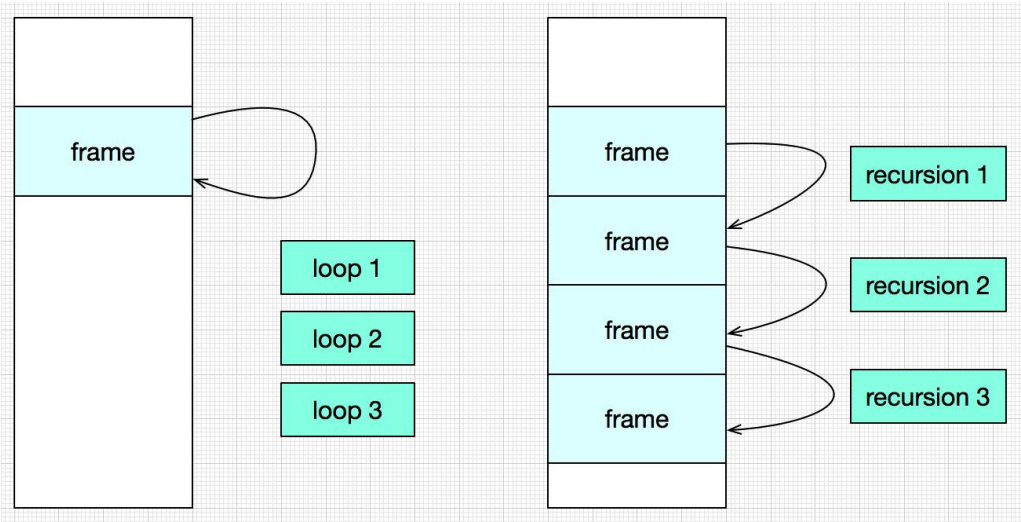
作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

循环与递归的区别

循环与递归是两种重要的代码流程，都能实现重复执行代码块。他们的核心区别为循环复用固定的栈帧，递归使用新的栈帧。循环在当前的栈帧中执行代码块，重复很多次也依然使用当前的栈帧；递归重复调用自身函数，调用函数一次则为函数分配一个栈帧，所以产生多个栈帧。
多角度对比：
内存角度，递归使用更多栈帧，占用很多内存，可能导致栈溢出。
指令角度，递归创建栈帧和销毁栈帧，使用栈操作指令，耗费更多时间。
性能角度，循环的内部外部变量的内存距离更近，使得时间局部性、空间局部性发挥作用，性能相对更好。



用 C 分析循环与递归的差异

编写代码： different.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

struct cat
{
    char desc[64];
};

int32_t count_loop = 0;      // 计数，循环
int32_t count_recursion = 0; // 计数，递归
int32_t count_max = 100;     // 最大计数
```

```
// 记录循环的变量的地址
uint64_t loop_begin_addr;
uint64_t loop_end_addr;

// 循环
void func_loop()
{
    for (; count_loop <= count_max; count_loop++)
    {
        // 栈上的变量
        struct cat tom;
        if (count_loop == 0)
        {
            loop_begin_addr = (uint64_t)&tom;
            printf("\n loop begin . at %#llX \n", loop_begin_addr);
        }
        if (count_loop == count_max)
        {
            loop_end_addr = (uint64_t)&tom;
            printf("\n loop end   . at %#llX  offset = %llu \n", loop_end_addr,
                   (loop_begin_addr - loop_end_addr));
        }
    }
}

// 记录递归的变量的地址
uint64_t recursion_begin_addr;
uint64_t recursion_end_addr;

// 递归
void func_recursion()
{
    // 退出递归
    if (count_recursion > count_max)
    {
        return;
    }
    // 栈上的变量
    struct cat tom;
    if (count_recursion == 0)
    {
        recursion_begin_addr = (uint64_t)&tom;
        printf("\n recursion begin . at %#llX \n", recursion_begin_addr);
    }
    else if (count_recursion == count_max)
    {
        recursion_end_addr = (uint64_t)&tom;
        printf("\n recursion end   . at %#llX  offset = %llu \n", recursion_end_addr,
               (recursion_begin_addr - recursion_end_addr));
```

```
    }
    // 调用自身
    ++count_recursion;
    func_recursion();
}

int main()
{
    printf(" struct cat size = %d \n", sizeof(struct cat));

    func_loop();
    func_recursion();
    return 0;
}
```

编译代码：

```
gcc different.c -o different
```

运行代码：

```
[root@local loop_recursion]# ./different
 struct cat size = 64

 loop begin . at 0X7FFE01C31A40

 loop end   . at 0X7FFE01C31A40  offset = 0

 recursion begin . at 0X7FFE01C31A40

 recursion end   . at 0X7FFE01C2FB00  offset = 8000
```

分析结果：
代码逻辑为，在栈上重复创建多次变量，查看变量的地址和地址差值。
循环很多次，复用 1 个栈帧，变量的地址一样，差值为 0。
递归很多次，创建了多个栈帧，变量的地址不一样，差值为 8000。

计算地址差值。
struct cat 大小为 64 字节。
callq 指令把 rip 写入栈帧，占用 8 字节。
pushq %rbp 指令把 rbp 写入栈帧，占用 8 字节。
差值计算 `( 64 + 8 + 8 ) * 100 = 8000` 。

递归的优点和适用场景

某些场景，递归的表达能力更强，更加符合开发人员的思维，使得开发成本、维护成本更低。在算法领域尤其明显，比如二叉树的深度遍历，用通配符匹配字符串，整数数组查找求和数字。
这里分析整数数组查找求和数字，给定 1 个整数数组，找到满足求和为 100 的 3 个整数，输出全部情况。

编写代码： digui_sum.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

// 原始数组
int nums[6] = {10, 30, 50, 90, 20, 0};

// 找到的 3 个数字
int pick[3];

// 递归查找满足求和条件的数字
void find_sum(int nums_index, int pick_index)
{
    // 如果已经找到 3 个元素了，就判断是否满足条件。
    if (pick_index == 3)
    {
        // 累加
        int sum = pick[0] + pick[1] + pick[2];
        if (sum == 100)
        {
            printf(" found : %d %d %d \n", pick[0], pick[1], pick[2]);
        }
        return;
    }
    // 判断边界条件
    if (pick_index > 3)
    {
        return;
    }
    // 判断边界条件
    if (nums_index >= 6)
    {
        return;
    }

    // 使用当前元素。读取元素，找下一个元素。
    pick[pick_index] = nums[nums_index];
    find_sum(nums_index + 1, pick_index + 1);

    // 不使用当前元素。继续看下一个元素。
    find_sum(nums_index + 1, pick_index);
}

int main()
{
```

```
    printf(" find 3 nums that can sum to 100 \n");
    find_sum(0, 0);
    return 0;
}
```

编译代码：

```
gcc digui_sum.c -o digui_sum
```

运行代码：

```
[root@local loop_recursion]# ./digui_sum
 find 3 nums that can sum to 100
 found : 10 90 0
 found : 30 50 20
```

分析结果：
递归函数的逻辑清晰，首先判断边界条件，打印满足求和条件的整数，然后枚举可能的情况执行递归，继续看下一个整数。
找到了满足条件的 2 种情况。10 90 0 的和为 100，30 50 20 的和为 100。

递归的一般结构为：
在递归函数的上部设置边界条件，退出递归函数。对应代码 `if (pick_index == 3)` 、 `if (pick_index > 3)` 、 `if (nums_index >= 6)` 。
在递归函数的下部枚举全部可能的情况，调用递归函数。对应代码 `find_sum(nums_index + 1, pick_index + 1)` 、`find_sum(nums_index + 1, pick_index)` 。