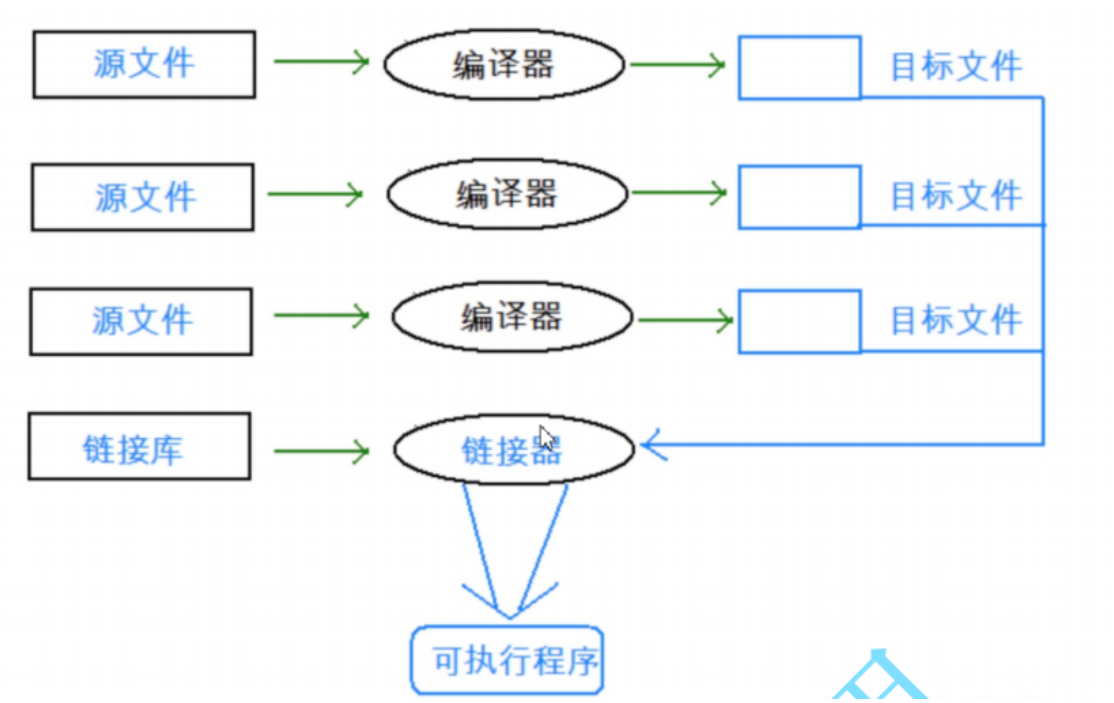
程序环境和预处理

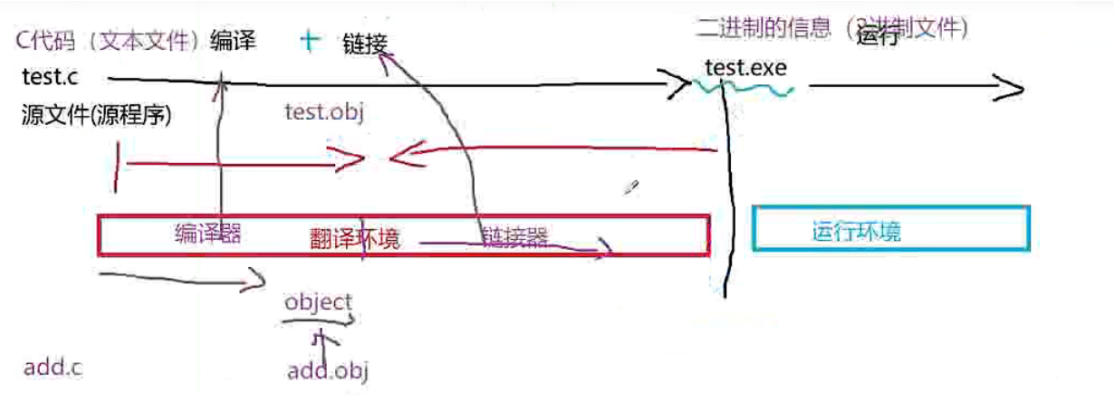
笔记本:	My Notebook		
创建时间:	2023/11/9 18:54	**更新时间:**	2023/11/14 21:46
作者:	lwb		
URL:	https://blog.csdn.net/lym940928/article/details/88368363		

在ANSI C的任何一种实现中，存在两个不同的环境
第1种是翻译环境，在这个环境中源代码被转换为可执行的机器指令 （把.c文件变为.exe文件依赖的环境）
第2种是执行环境，它用于实际执行代码

1.详解编译+链接
1.1翻译环境



- 组成一个程序的每个源文件通过编译过程分别转换成目标代码(.obj目标文件)
- 每个目标文件由链接器捆绑在一起，形成一个单一而完整的可执行程序
- 链接器同时也会引入标准C函数库中任何被该程序所用到的函数，而且它可以搜索程序员个人的程序库，将其需要的函数也链接到程序中



1.2编译本身也分为几个阶段

linux环境下

1.预处理/预编译 gcc -E test.c -o test.i (预处理做的事：#include 包含头文件，删除注释，处理#define-替换文本)

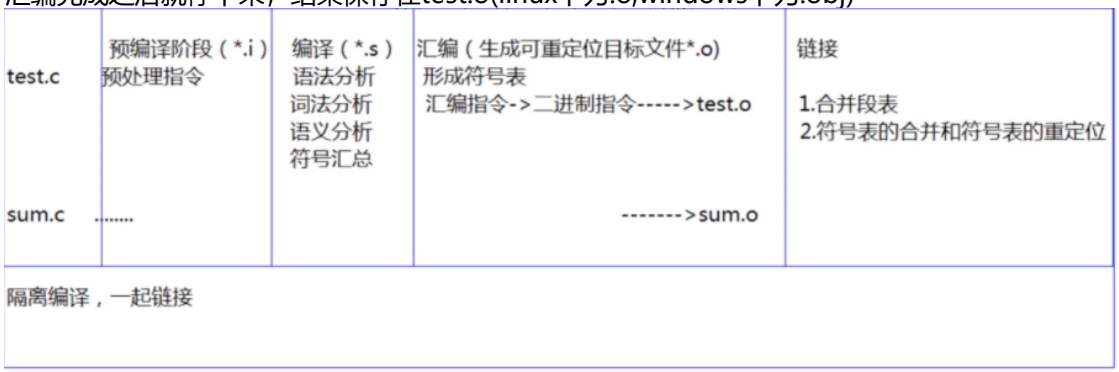
预处理完成之后就停下来，预处理之后产生的结果都放在test.i文件中

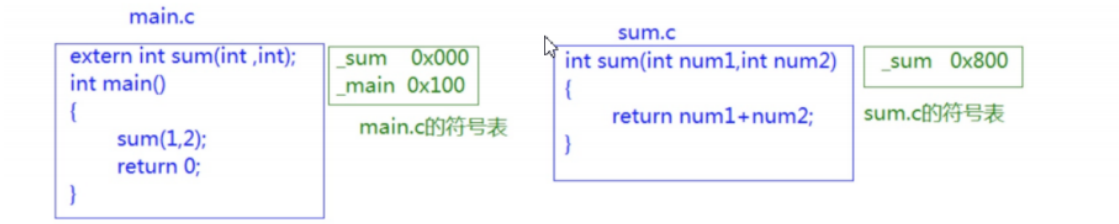
2.编译 gcc -S test.c (将c语言代码转换为汇编代码 - 1. 语法分析 2. 词法分析 3. 语义分析 4. 符号汇总)

编译完成之后就停下来，结果保存在test.s中

3.汇编 gcc -c test.c (把汇编代码转换成了二进制指令， 形成符号表)

汇编完成之后就停下来，结果保存在test.o(linux下为.o,windows下为.obj)





链接

1.合并段表（elf文件格式）

2.符号表的合并和重定位 （如上图，符号表重定位后_sum的地址就变为0x800了）

1.3运行环境

程序执行的过程：

1. 程序必须载入内存中。在有操作系统的环境中：一般这个由操作系统完成。在独立的环境中，程序的载入必须有手工安排，也可能是通过可执行代码置入只读内存来完成。
2. 程序的执行便开始。接着便调用main函数
3. 开始执行程序代码。这个时候程序将使用一个运行时堆栈，存储函数的局部变量和返回地址。程序同时也可以使用静态(static)内存，存储于静态内存中的变量在程序的整个执行过程一直保留他们的值。
4. 终止程序。正常终止main函数；也可能是意外终止

2.预处理详解

2.1预定义符号

```
__FILE__      //进行编译的源文件
__LINE__     //文件当前的行号
__DATE__    //文件被编译的日期
__TIME__    //文件被编译的时间
__STDC__    //如果编译器遵循ANSI C，其值为1，否则未定义
```

这些预定义符号都是语言内置的

2.2 #define

2.2.1 #define定义标识符

```
#define MAX 1000
#define reg register          //为 register这个关键字，创建一个简短的名字
#define do_forever for(;;)     //用更形象的符号来替换一种实现
#define CASE break;case        //在写case语句的时候自动把 break写上。
// 如果定义的 stuff过长，可以分成几行写，除了最后一行外，每行的后面都加一个反斜杠(续行符)。
#define DEBUG_PRINT printf("file:%s\tline:%d\t \
                           date:%s\ttime:%s\n" ,\
                           __FILE__,__LINE__ ,       \
                           __DATE__,__TIME__
```

2.2.2 #define定义宏

`#define name( parament-list ) stuff`
其中的 `parament-list` 是一个由逗号隔开的符号表，它们可能出现在stuff中。
注意：
参数列表的左括号必须与name紧邻。
如果两者之间有任何空白存在，参数列表就会被解释为stuff的一部分。

2.2.3 #define替换规则

在程序中扩展#define定义符号和宏时，需要涉及几个步骤。
1. 在调用宏时，首先对参数进行检查，看看是否包含任何由#define定义的符号。如果是，它们首先
被替换。
2. 替换文本随后被插入到程序中原来文本的位置。对于宏，参数名被他们的值所替换。
3. 最后，再次对结果文件进行扫描，看看它是否包含任何由#define定义的符号。如果是，就重复上
述处理过程。
注意：
1. 宏参数和#define 定义中可以出现其他#define定义的符号。但是对于宏，不能出现递归。
2. 当预处理器搜索#define定义的符号的时候，字符串常量的内容并不被搜索。

如何把参数插入到字符串中?
使用 # ，把一个宏参数变成对应的字符串

```
#define PRINT(X) printf("the value of "#X" is %d\n", X)
int main()
{
        // printf("hello " "world\n");
        int a = 10;
        int b = 20;
        PRINT(a);
        // printf("the value of ""a""is %d\n", a);
        PRINT(b);
        // printf("the value of ""b""is %d\n", b);
        return 0;
}
```

##的作用
##可以把位于它两边的符号合成一个符号
它允许宏定义从分离的文本片段创建标识符
注：这样的连接必须产生一个合法的标识符。否则其结果就是未定义的

2.2.5 带副作用的宏参数

当宏参数在宏的定义中出现超过一次的时候，如果参数带有副作用，那么你在使用这个宏的时候就可能出现危险，导致不可预测的后果。副作用就是表达式求职的时候出现的永久性效果。

```
#define MAX(X,Y) ((X)>(Y)?(X):(Y))
int main()
{
        int a = 10;
        int b = 11;
        int max = MAX(a++, b++);
```

```
        // int max = ((a++)>(b++)?(a++):(b++));
        printf("%d\n", max); // 12
        printf("%d\n", a); // 11
        printf("%d\n", b); // 13
        return 0;
}
```

2.2.6 宏和函数对比
宏通常被应用于执行简单的运算。

原因有二：
1.用于调用函数和从函数返回的代码可能比实际执行这个小型计算工作所需要的时间更多，所以宏比函数在程序的规模和速度方面更胜一筹（函数在调用的时候会有调用和返回的开销，宏在预处理阶段就完成了替换）
2.更为重要的是函数的参数必须声明为特定的类型，所以函数只能在类型合适的表达式上使用，反之这个宏可以适用于整形、长整型、浮点型等可以用于来比较的类型，宏是类型无关的

宏的缺点：
1.每次使用宏的时候，一份宏定义的代码将插入到程序中，除非宏比较短，否则可能大幅度增加程序的长度。
2.宏是无法调试的
3.宏由于类型无关，也就不够严谨
4.宏可能会带来运算符优先级的问题，导致程序容易出错

宏有时候可以做函数做不到的事情。比如：宏的参数可以出现**类型**，但是函数做不到。

```
#define MALLOC(num, type)\
(type *)malloc(num * sizeof(type))
...
//使用
MALLOC(10, int);//类型作为参数
//预处理器替换之后：
(int *)malloc(10 * sizeof(int));
```

inline - 内联函数，解决一些频繁调用的小函数大量消耗栈空间(栈内存)的问题，解决了函数调用开销的问题
栈空间就是指放置程序的局部数据的内存空间

2.2.7 命名约定
函数名不要全部大写
宏名要全部大写

2.3 #undef
这条指令用于移除一个宏定义

2.4 命令行定义
许多C的编译器提供了一种能力，允许在命令行中定义符号。用于启动编译过程。
例如：当我们根据同一个源文件要编译出一个程序的不同版本的时候，这个特性有点用处。（假定某个
程序中声明了一个某个长度的数组，如果机器内存有限，我们需要一个很小的数组，但是另外一个机器
内存大些，我们需要一个数组能够大些。）

```
#include <stdio.h>
int main()
{
   int array [ARRAY_SIZE];
   int i = 0;
   for(i = 0; i< ARRAY_SIZE; i ++)
  {
```

```
        array[i] = i;
  }
   for(i = 0; i< ARRAY_SIZE; i ++)
  {
        printf("%d " ,array[i]);
  }
   printf("\n" );
   return 0;
}
```

```
//linux 环境演示
gcc -D ARRAY_SIZE=10 programe.c
```

2.5 条件编译

在编译一个程序的时候我们如果要将一条语句（一组语句）编译或者放弃，可以使用条件编译

```
1.
#if 常量表达式
//...
#endif
//常量表达式由预处理器求值。
如:
#define __DEBUG__ 1
#if __DEBUG__
//..
#endif
2.多个分支的条件编译
#if 常量表达式
//...
#elif 常量表达式
//...
#else
//...
#endif
3.判断是否被定义
#if defined(symbol)
#ifdef symbol
#if !defined(symbol)
#ifndef symbol
4.嵌套指令
#if defined(OS_UNIX)
#ifdef OPTION1
unix_version_option1();
#endif
#ifdef OPTION2
unix_version_option2();
#endif
#elif defined(OS_MSDOS)
#ifdef OPTION2
msdos_version_option2();
#endif
#endif
```

2.6 文件包含

#include 指令可以使另外一个文件被编译，就像该文件实际出现在#include指令的地方一样
这种替换即：预处理器先删除这条指令，并用包含文件的内容替换，这样一个源文件被包含几次就编译几次

2.6.1 头文件被包含的方式

本地文件包含

```
#include "filename"
```

查找策略：先在源文件所在目录下查找，如果该头文件未找到，编译器就像查找库函数头文件一样在标准位置查找头文件，如果仍找不到就提示编译错误。

Linux环境的标准头文件的路径：

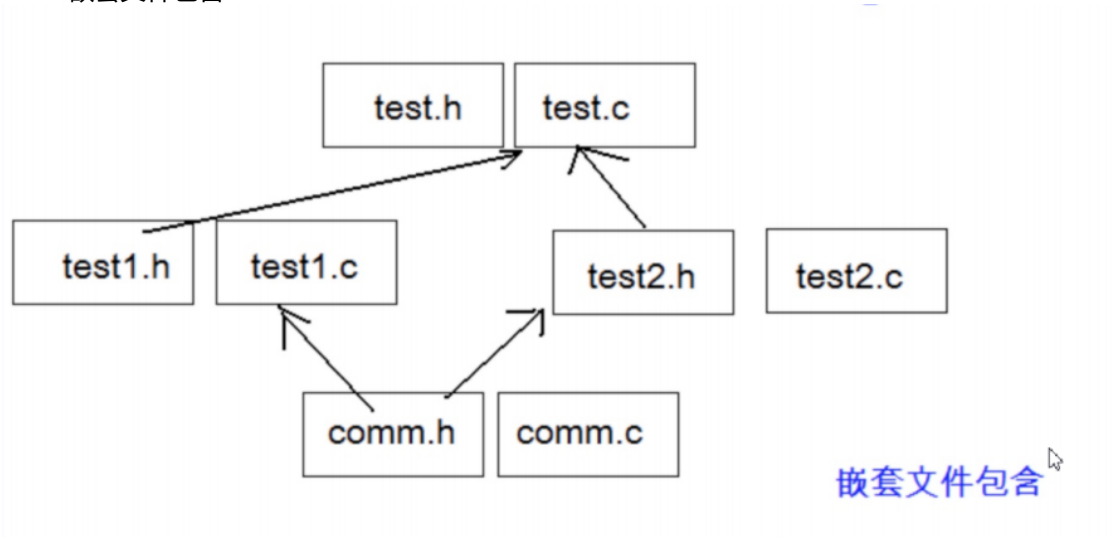
```
/usr/include
```

库文件包含

```
#include <filename.h>
```

查找头文件直接去标准路径下查找，如果找不到就提示编译错误

2.6.2 嵌套文件包含



comm.h和comm.c是公共模块。
test1.h和test1.c使用了公共模块。
test2.h和test2.c使用了公共模块。
test.h和test.c使用了test1模块和test2模块。
这样最终程序中就会出现两份comm.h的内容。这样就造成了文件内容的重复。
如何解决？ 条件编译
每个头文件的开头写：

```
#ifndef __TEST_H__
#define __TEST_H__
//头文件的内容
#endif   //__TEST_H__
```

或者

```
#pragma once
```