

# 第23讲：编译和链接

## 目录

1. 翻译环境和运行环境
2. 翻译环境：预编译+编译+汇编+链接

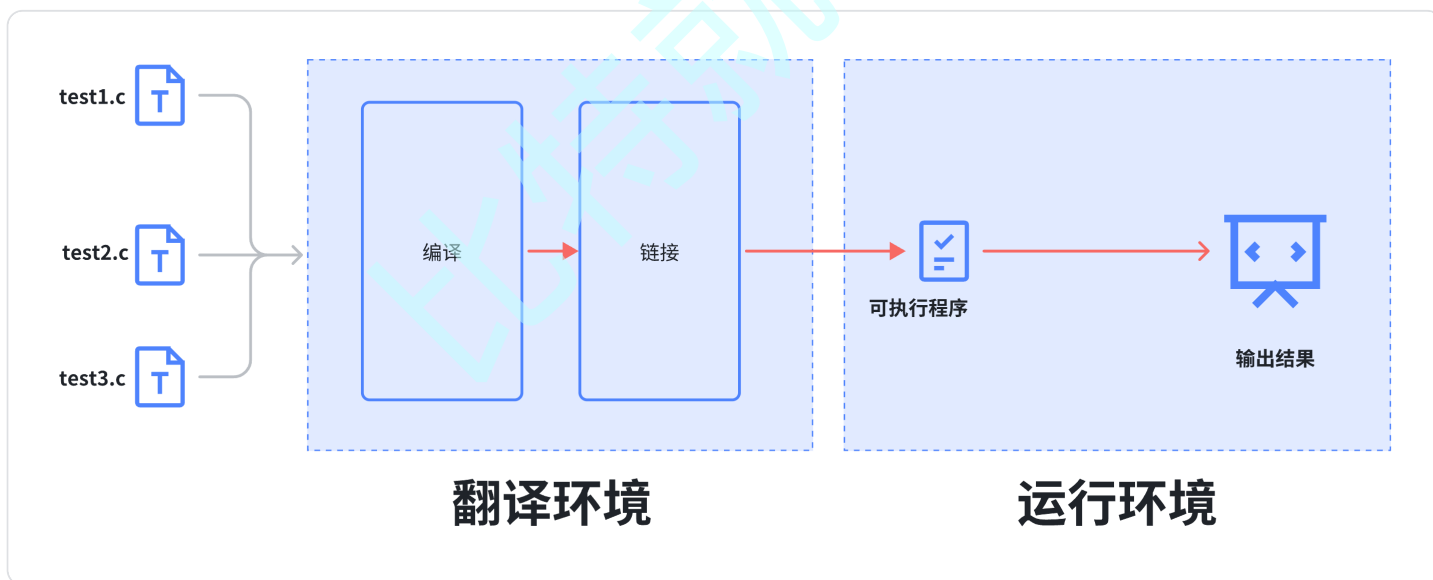
正文开始

## 1. 翻译环境和运行环境

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（二进制指令）。

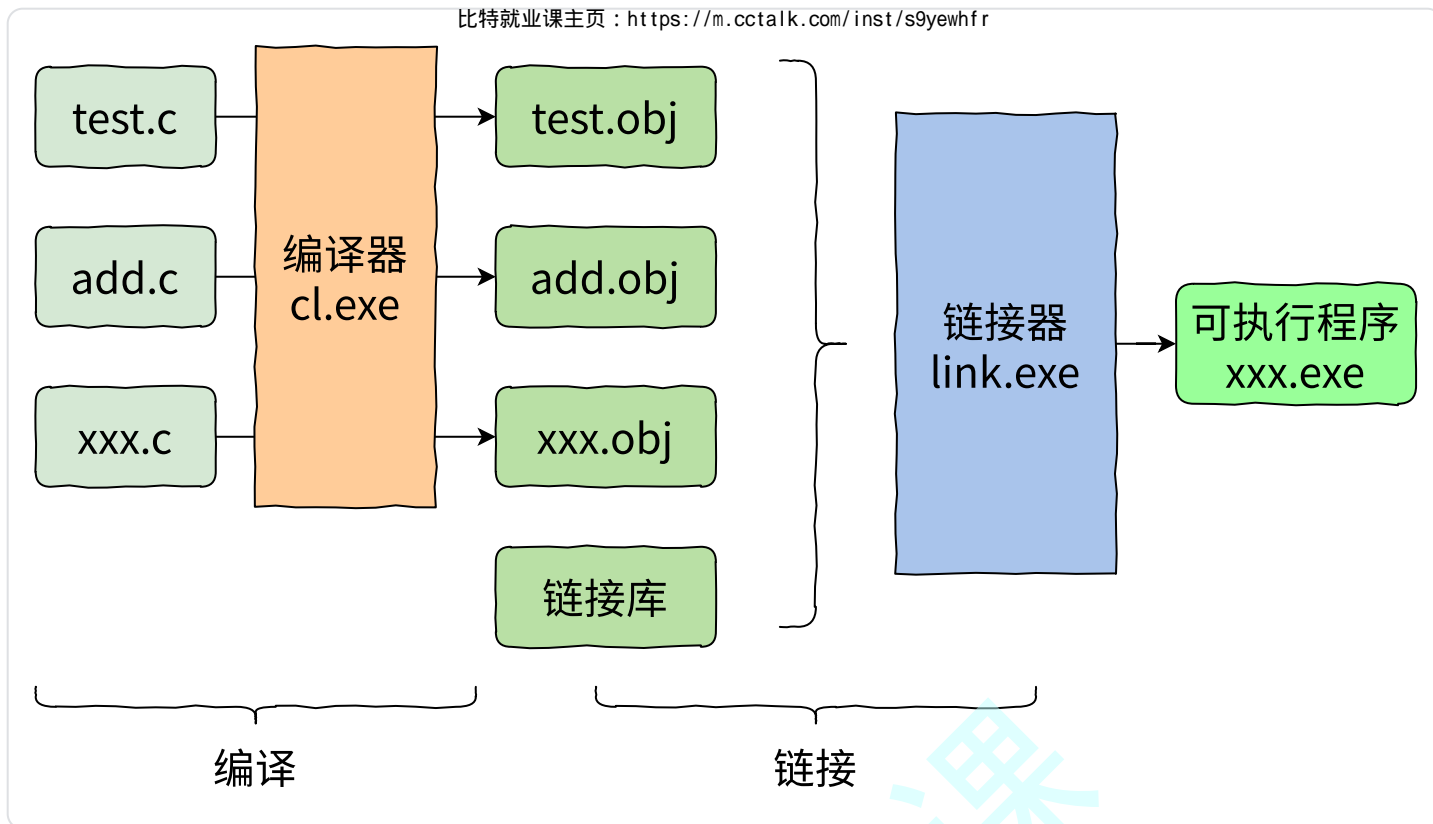
第2种是执行环境，它用于实际执行代码。



## 2. 翻译环境

那翻译环境是怎么将源代码转换为可执行的机器指令的呢？这里我们就得展开讲解一下翻译环境所做的事情。

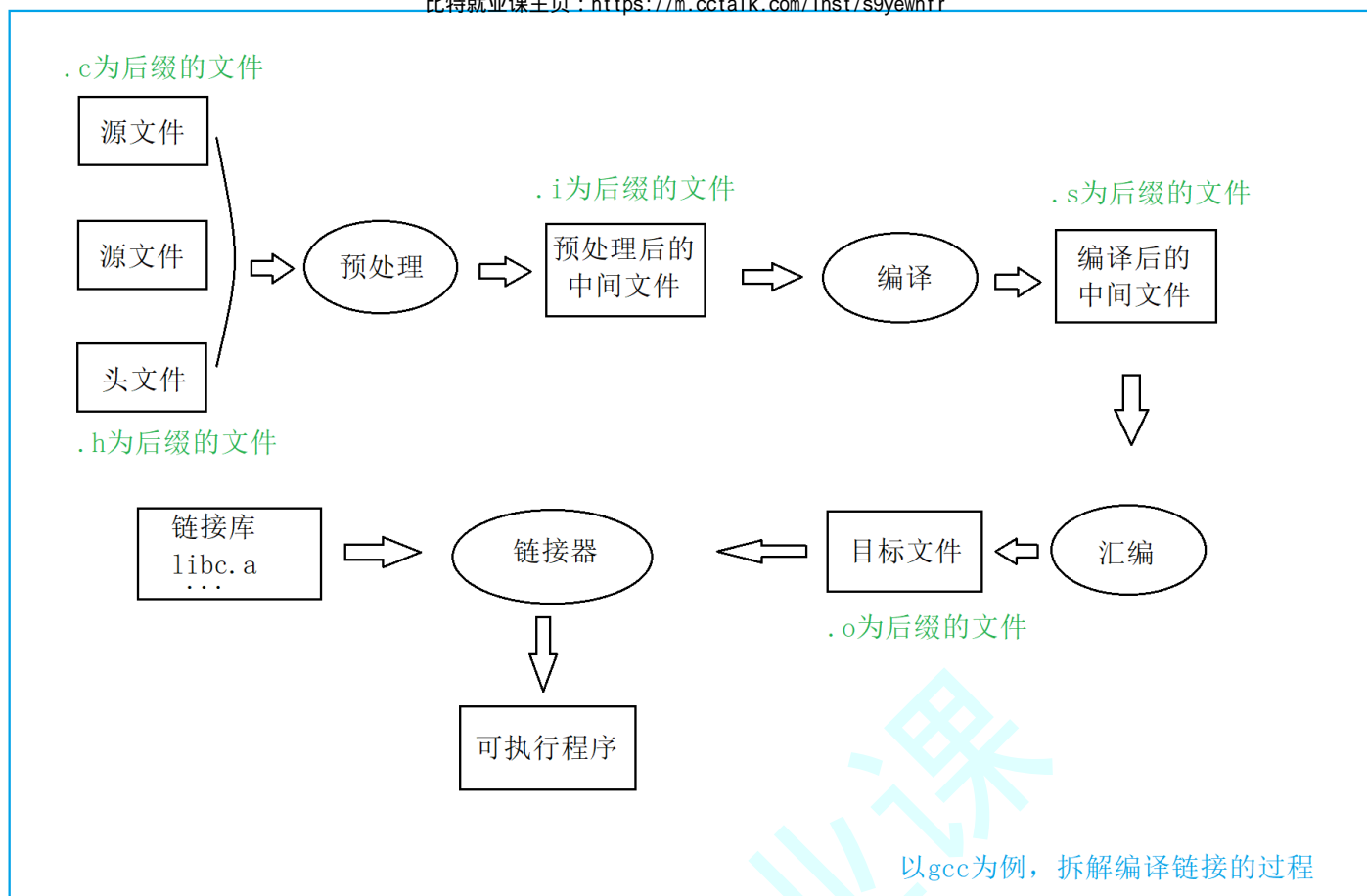
其实翻译环境是由**编译**和**链接**两个大的过程组成的，而**编译**又可以分解成：预处理（有些书也叫预编译）、编译、汇编三个过程。



一个C语言的项目中可能有多个 `.c` 文件一起构建，那多个 `.c` 文件如何生成可执行程序呢？

- 多个.c文件单独经过编译器，编译处理生成对应的目标文件。
- 注：在Windows环境下的目标文件的后缀是 `.obj`，Linux环境下目标文件的后缀是 `.o`
- 多个目标文件和链接库一起经过链接器处理生成最终的可执行程序。
- 链接库是指运行时库(它是支持程序运行的基本函数集合)或者第三方库。

如果再把编译器展开成3个过程，那就变成了下面的过程：



## 2.1 预处理（预编译）

在预处理阶段，源文件和头文件会被处理成为 `.i` 为后缀的文件。

在 `gcc` 环境下想观察一下，对 `test.c` 文件预处理后的 `i` 文件，命令如下：

```
1 gcc -E test.c -o test.i
```

预处理阶段主要处理那些源文件中#开始的预编译指令。比如：`#include`、`#define`，处理的规则如下：

- 将所有的 `#define` 删除，并展开所有的宏定义。
- 处理所有的条件编译指令，如：`#if`、`#ifdef`、`#elif`、`#else`、`#endif`。
- 处理 `#include` 预编译指令，将包含的头文件的内容插入到该预编译指令的位置。这个过程是递归进行的，也就是说被包含的头文件也可能包含其他文件。
- 删除所有的注释
- 添加行号和文件名标识，方便后续编译器生成调试信息等。
- 或保留所有的 `#pragma` 的编译器指令，编译器后续会使用。

经过预处理后的 `.i` 文件中不再包含宏定义，因为宏已经被展开。并且包含的头文件都被插入到 `.i` 文件中。所以当我们无法知道宏定义或者头文件是否包含正确的时候，可以查看预处理后的 `.i` 文件来确认。

## 2.2 编译

编译过程就是将预处理后的文件进行一系列的：**词法分析、语法分析、语义分析及优化**，生成相应的汇编代码文件。

编译过程的命令如下：

```
1 gcc -S test.i -o test.s
```

对下面代码进行编译的时候，会怎么做呢？假设有下面的代码

```
1 array[index] = (index+4)*(2+6);
```

### 2.2.1 词法分析

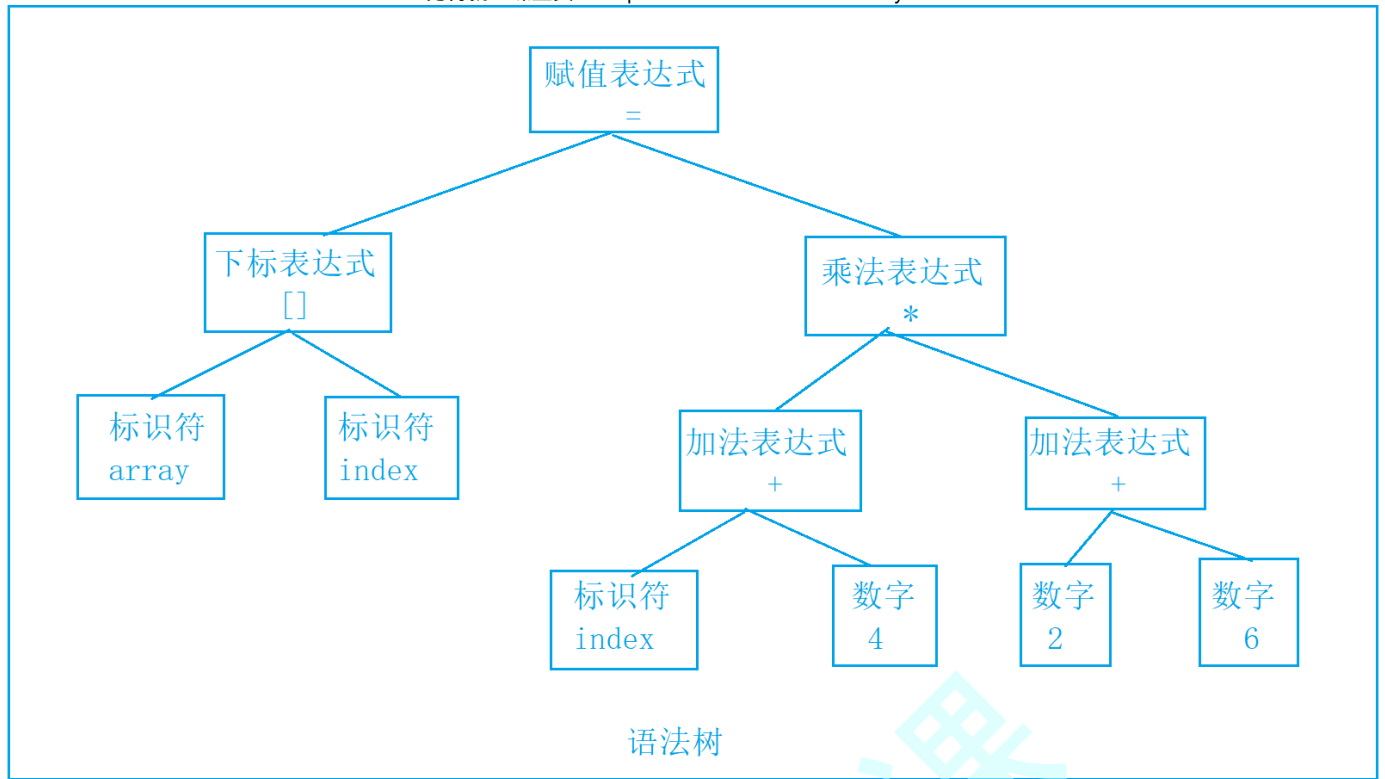
将源代码程序被输入**扫描器**，扫描器的任务就是简单的进行**词法分析**，把代码中的字符分割成一系列的记号（关键字、标识符、字面量、特殊字符等）。

上面程序进行词法分析后得到了16个记号：

记号	类型
array	标识符
[	左方括号
index	标识符
]	右方括号
=	赋值
(	左圆括号
index	标识符
+	加号
4	数字
)	右圆括号
*	乘号
(	左圆括号
2	数字
+	加号
6	数字
)	右圆括号

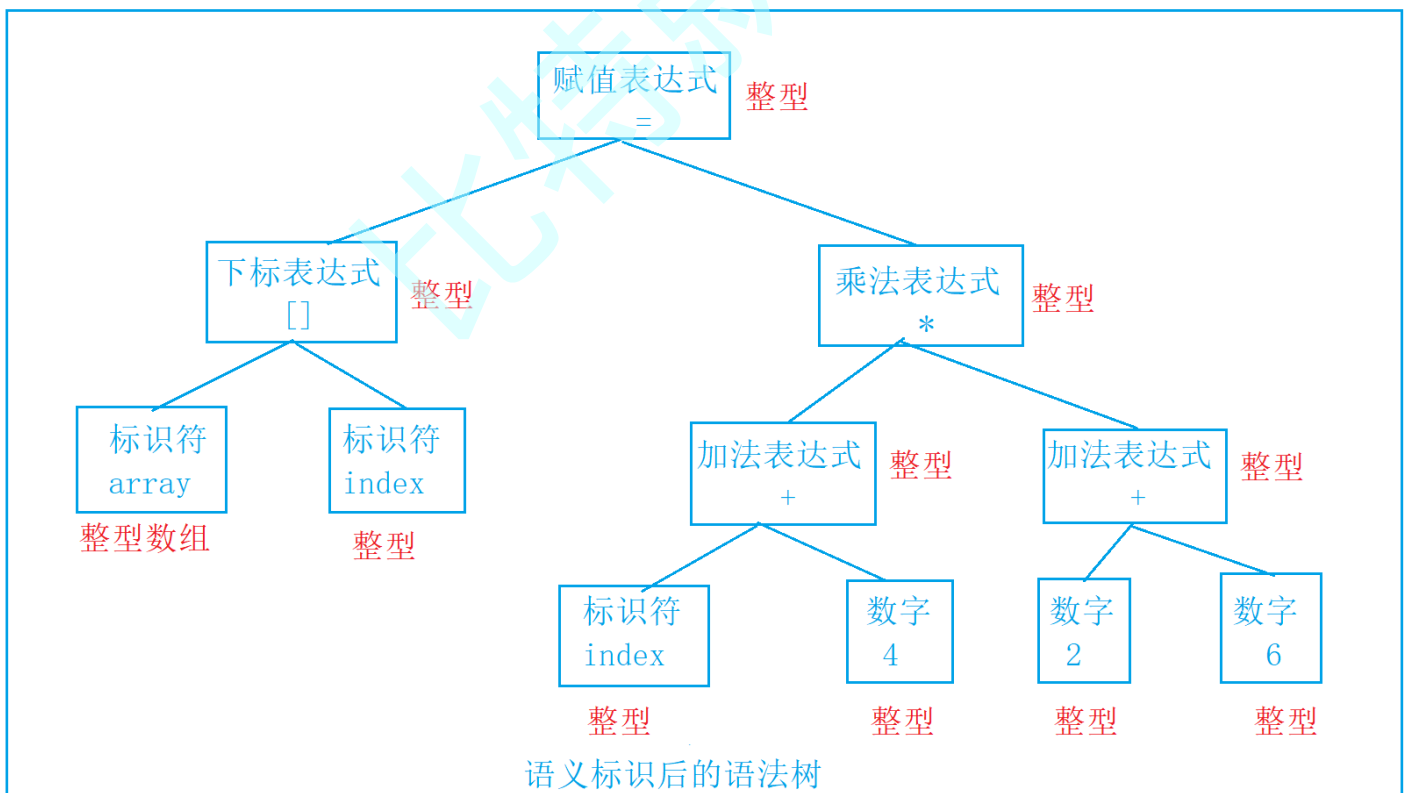
### 2.2.2 语法分析

接下来**语法分析器**，将对扫描产生的记号进行语法分析，从而产生**语法树**。这些语法树是以表达式为节点的树。



### 2.2.3 语义分析

由**语义分析器**来完成语义分析，即对表达式的语法层面分析。编译器所能做的分析是语义的静态分析。静态语义分析通常包括声明和类型的匹配，类型的转换等。这个阶段会报告错误的语法信息。



## 2.3 汇编

**编译器**是将汇编代码转变成机器可执行的指令，每一个汇编语句几乎都对应一条机器指令。就是根据汇编指令和机器指令的对照表一一的进行翻译，也不做指令优化。

汇编的命令如下：

```
1 gcc -c test.s -o test.o
```

## 2.4 链接

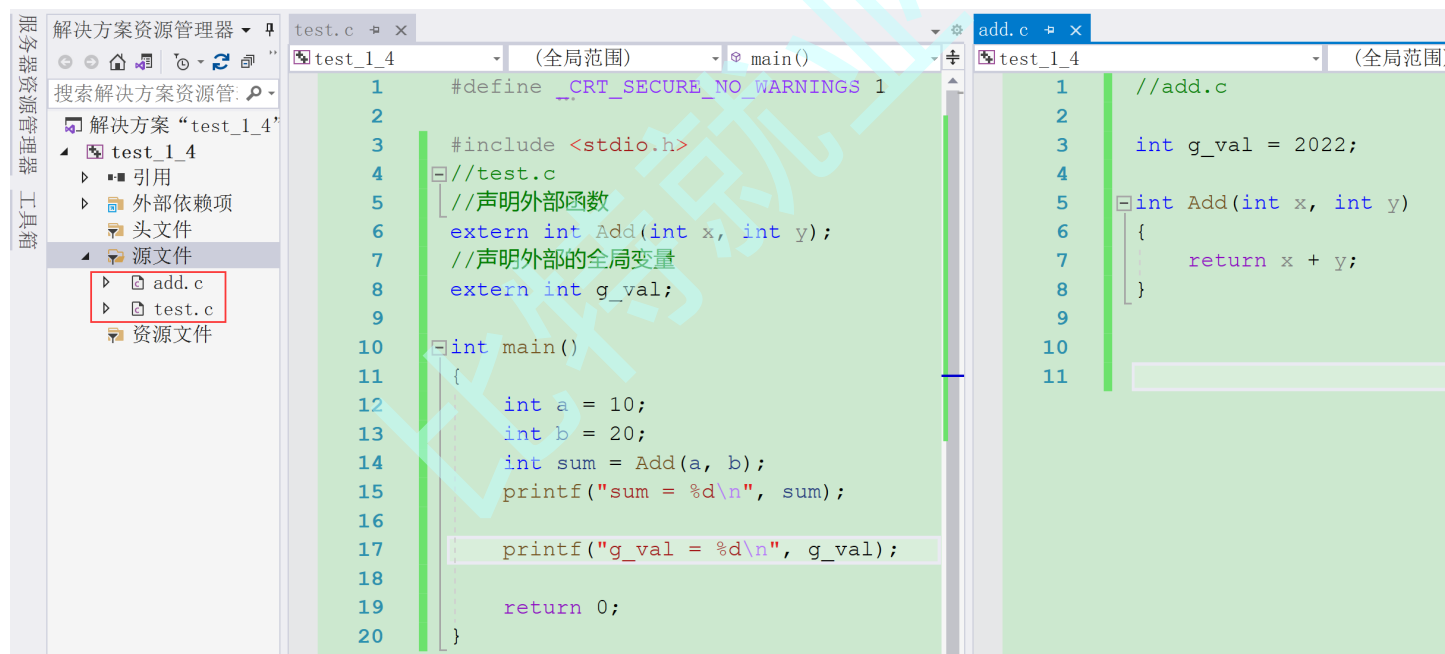
链接是一个复杂的过程，链接的时候需要把一堆文件链接在一起才生成可执行程序。

链接过程主要包括：地址和空间分配，符号决议和重定位等这些步骤。

**链接**解决的是一个项目中多文件、多模块之间互相调用的问题。

比如：

在一个C的项目中有2个.c文件（`test.c` 和 `add.c`），代码如下：



test.c

```
1 #include <stdio.h>
2 //test.c
3 //声明外部函数
4 extern int Add(int x, int y);
5 //声明外部的全局变量
6 extern int g_val;
7
8 int main()
```

add.c

```
1 int g_val = 2022;
2
3 int Add(int x, int y)
4 {
5     return x+y;
6 }
```

```
9 {  
10     int a = 10;  
11     int b = 20;  
12     int sum = Add(a, b);  
13     printf("%d\n", sum);  
14     return 0;  
15 }
```

我们已经知道，每个源文件都是单独经过编译器处理生成对应的目标文件。

test.c 经过编译器处理生成 test.o

add.c 经过编译器处理生成 add.o

我们在 test.c 的文件中使用了 add.c 文件中的 Add 函数和 g\_val 变量。

我们在 test.c 文件中每一次使用 Add 函数和 g\_val 的时候必须确切的知道 Add 和 g\_val 的地址，但是由于每个文件是单独编译的，在编译器编译 test.c 的时候并不知道 Add 函数和 g\_val 变量的地址，所以暂时把调用 Add 的指令的目标地址和 g\_val 的地址搁置。等待最后链接的时候由链接器根据引用的符号 Add 在其他模块中查找 Add 函数的地址，然后将 test.c 中所有引用到 Add 的指令重新修正，让他们的目标地址为真正的 Add 函数的地址，对于全局变量 g\_val 也是类似的方法来修正地址。这个地址修正的过程也被叫做：**重定位**。

前面我们非常简洁的讲解了一个C的程序是如何编译和链接，到最终生成可执行程序的过程，其实很多内部的细节无法展开讲解。比如：目标文件的格式elf，链接底层实现中的空间与地址分配，符号解析和重定位等，如果你有兴趣，可以看《程序员的自我修养》一书来详细了解。

### 3. 运行环境

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，程序的载入必须由手工安排，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（stack），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；也有可能是意外终止。



比特就业课