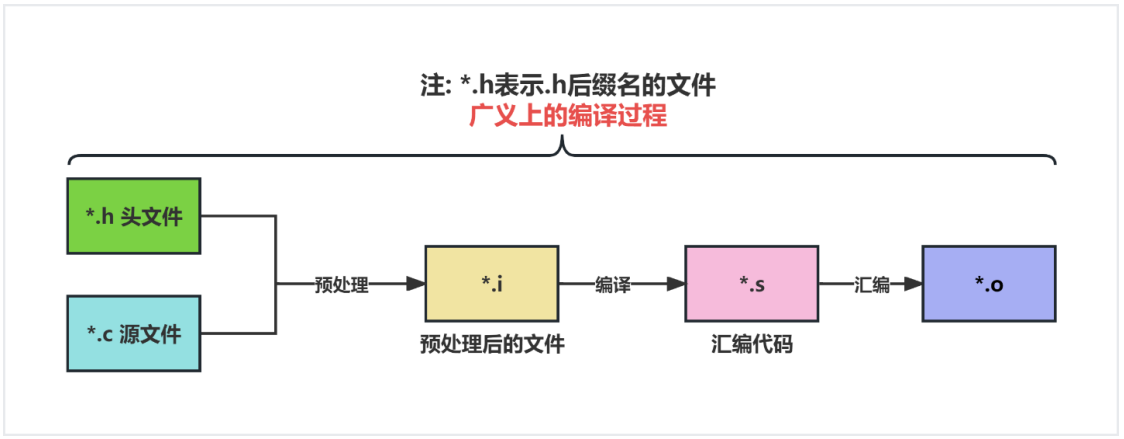
一个C语言的源文件.c文件，它在大的过程上，转换成可执行文件，需要两个步骤：
1.广义上的编译过程
2.链接过程



预处理过程：
将源文件以及用到的.h头文件经过预处理过程，生成了一个.i文件
在这个过程中，预处理器主要做两个事情：
1.执行预处理指令
2.丢弃代码中的注释，注释就不参与后续的任何过程了。

预处理指令是什么?
在C语言代码中，以"#"开头的指令就是预处理指令。
最常见的预处理指令有两种形式：
1.#include 包含头文件指令
包含头文件可以简单的认为是把头文件中的内容 复制到预处理指令的位置
但实际上会有一些区别，但总体上可以这么理解
头文件中大概有：函数的声明，类型的声明别名等等。

```
// 函数的声明 = 函数头 + ; 结尾 函数的声明只是告诉编译器有这个函数且这个函数长什么样子，如何调用它
// 函数的声明和函数的实现没有关系，声明是不包含实现的，一个函数只有声明显然是不能调用的
// 函数头 = 返回值类型 + 函数名(形参列表)

void fun(void);
int fun2(int a, int b);

// 函数的定义：实现某个函数。就是带着函数体的{}
int fun2(int a, int b) {
}
```

在C代码中，包含头文件时
使用尖括号 <>：
使用尖括号时，编译器只在标准库头文件的路径中搜索头文件。它不会在当前文件的目录中查找

使用双引号""
当使用双引号时，编译器首先在当前文件的目录中搜索头文件。
如果在这里没有找到，它会继续在标准库头文件的路径中搜索。

2.#define 表示宏定义
1.用于定义宏常量，符号常量

#define 用于定义宏常量
其实就是给字面值常量 起个名字
注意：
1.宏常量的名字必须是全部大写的，如果有多个单词，下划线隔开
2.宏定义是没有分号的，不要加分号

宏常量定义后在代码中使用，在预处理阶段 本质上就是文本替换。

2.用于定义宏函数，函数宏
宏函数的定义
定义一个宏函数，用于求两个数的平方差
注意事项：
1.宏函数的名字全部大写 下划线隔开 见名知意
2.在写宏函数定义的表达式时，要添加上必要的()，增强表达式的可读性（建议）
3.在宏函数定义的表达式中，每一个参数都必须用()括起来
4.在宏函数定义的表达式中，整个表达式必须用()括起来

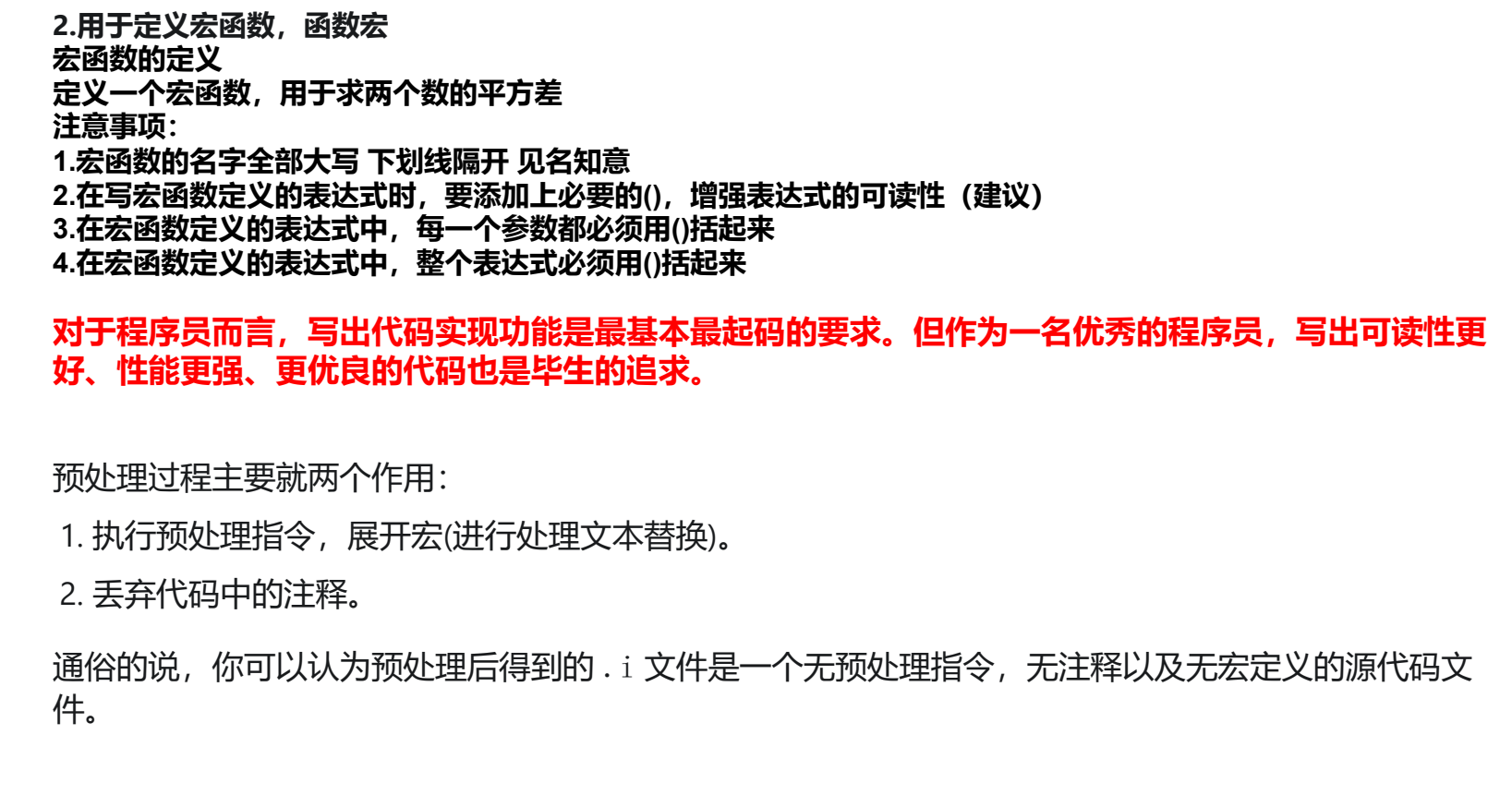
对于程序员而言，写出代码实现功能是最基本最起码的要求。但作为一名优秀的程序员，写出可读性更好、性能更强、更优良的代码也是毕生的追求。

预处理过程主要就两个作用：

1. 执行预处理指令，展开宏(进行处理文本替换)。
2. 丢弃代码中的注释。

通俗的说，你可以认为预处理后得到的 .i 文件是一个无预处理指令，无注释以及无宏定义的源代码文件。

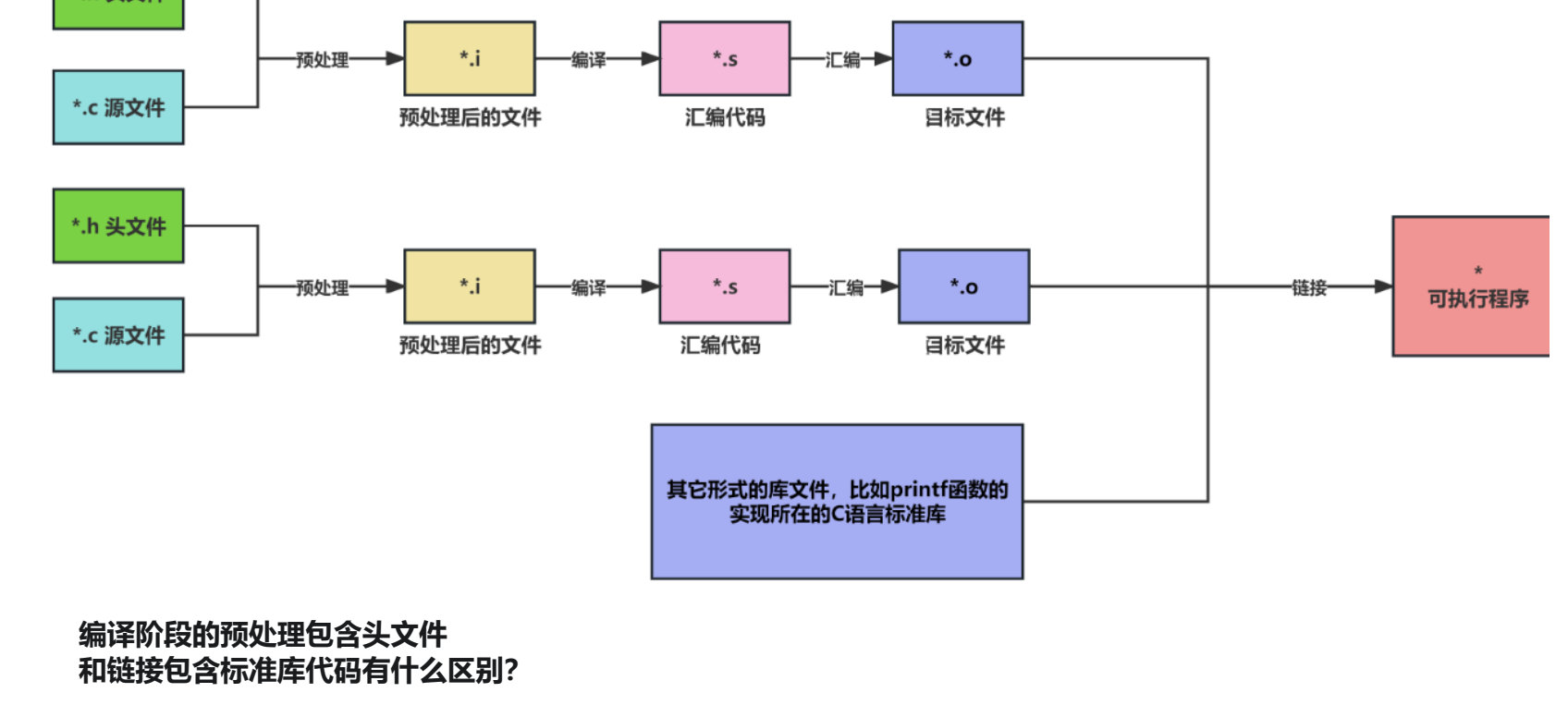
上面的过程都是广义上编译的过程，编译完成后得到一个.o目标文件
是机器语言指令
接下来需要进行链接，才能生成可执行程序



编译阶段的预处理包含头文件
和链接包含标准库代码有什么区别?

比如.c文件中调用了printf函数
于是需要包含头文件stdio.h
这个包含 你可以理解为包含函数的声明
包含了声明就意味着可以调用函数了，这样就不会编译出错了，或者不会警告

链接可以理解成包含函数的定义 使得函数真正可以调用
C语言的这种设定 使得C代码可以更方便灵活的调用外部函数



```
/*
* 在C代码中，包含头文件时
*    使用尖括号 <>:
*    使用尖括号时，编译器只在标准库头文件的路径中搜索头文件。它不会在当前文件的目录中查找。
*
*    使用双引号""
*    当使用双引号时，编译器首先在当前文件的目录中搜索头文件。
*    如果在这里没有找到，它会继续在标准库头文件的路径中搜索。
*/
```

编译错误：绝大多数都是由于疏忽大意导致的语法错误。

链接错误：绝大多数链接都和函数调用有关
如果链接出错要检查：
1.是否包含头文件
2.是不是函数名写错了

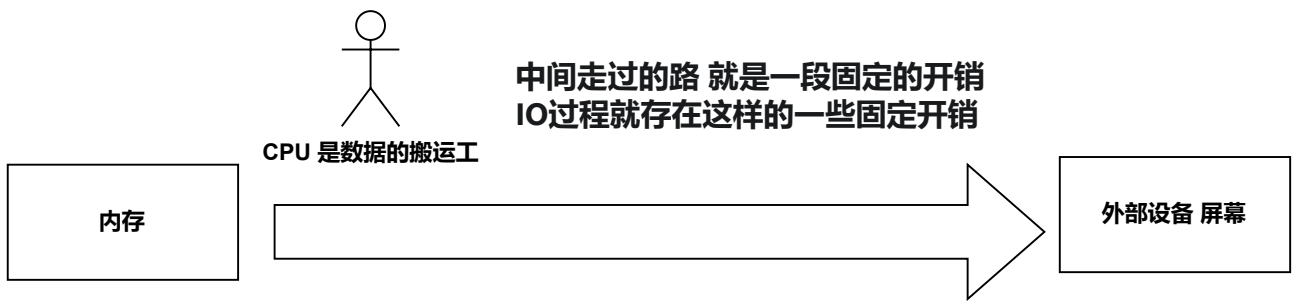
思考：
调用一个函数，这个函数需要2个参数，但我只给一个
这是什么错误?
这是不符合语法 是编译错误

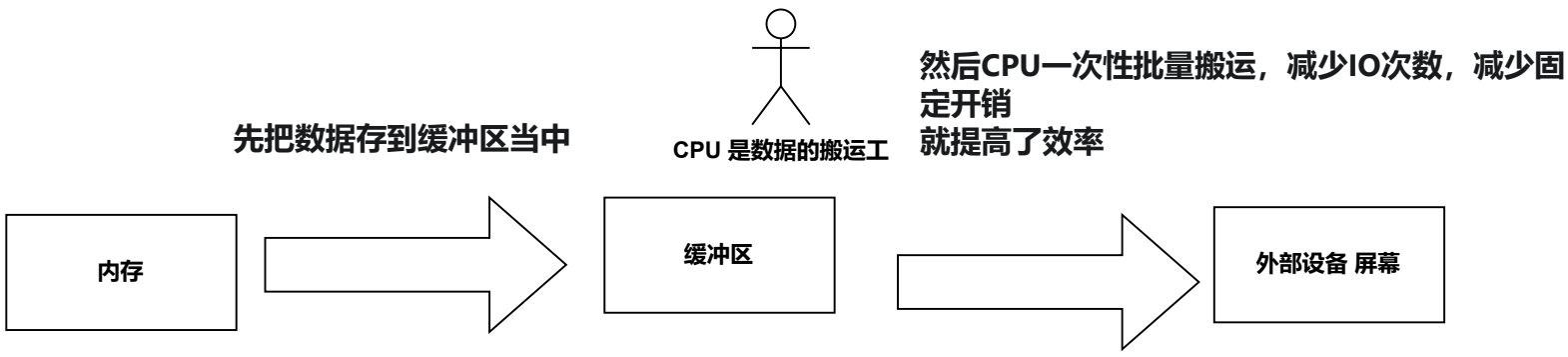
在C语言中调用printf函数，将内存中的数据输出到外部设备显示器上
它是一个IO操作的过程，效率是非常低的

这个效率低，从硬件的层面考虑，是由于内存以及外部设备的效率低下，尤其是外部设备
外部设备在IO的过程中就需要大量的准备初始化以及等待时间，耗费了很多时间

硬件上的性能若我们程序员是没办法的
但是我们要知道，在IO过程中，除了硬件上性能的损害，还会存在一些软件层面上的固定开销：
1.每次IO过程都需要进行系统调用，这往往涉及内核态，用户态的切换，造成了很大的性能孙秀
2.操作系统在IO的过程中可能会涉及一些处理，这也会损耗性能
....

很明显在软件层面上，IO的次数越少，这些固定的开销就越少
相当于搬运货物的过程 少搬运了几次





1. scanf函数。当从键盘输入时，输入的字符首先被保存在 stdin 的缓冲区(标准输入流)中。当满足某个触发条件后，程序才会从缓冲区读取并处理这些字符，从而减少了IO交互次数。
2. printf函数。输出到屏幕的内容会先被暂存到 stdout 的缓冲区。当满足某个触发条件后，这些内容会一次性写入并显示到屏幕，降低了与显示设备的交互频率。
3. 不管是stdin还是stdout缓冲区都是内存的一部分。

从上述内容中，我们可以明确地看到缓冲区的一个显著特点：当满足特定的条件时，程序会开始对缓冲区内的数据执行输入或输出操作。

这种"满足条件即触发数据传输"的行为，被我们称为"缓冲区的自动刷新"机制。当然缓冲区一般都有手动刷新的机制。

根据自动刷新机制的不同，缓冲区大致有以下分类：

1.全缓冲区，满缓冲区。只有当缓冲区满了以后，才会自动刷新缓冲区。实际上所有缓冲区满了后都会自动刷新。文件流缓冲区是非常典型的满缓冲区。

2.行缓冲区。只有碰到换行符才会自动刷新。stdin和stdout是最典型的行缓冲区。

3.不缓冲，无缓冲区。只要有数据就刷新。保证数据刷新的实时性时，就需要用到不缓冲。比如stderr标准错误缓冲区。

注意：

1.一般来说，C程序执行完毕，也就是main函数执行完毕。普遍也会自动刷新缓冲区。

2.除此之外,不同的平台编译器 也会有自己比较独特的刷新缓冲区的机制

printf函数在调用时 的第一个参数 是格式字符串
这个字符串由两部分组成:
1.普通字符, 除了转换说明都是普通字符串, 怎么写, 打印就是什么样子
2.转换说明, 转换说明以%开头
它的作用是:
a.占位符
b.控制输出的格式,比如长度,精度等
c.指示被转换成字符数据的对应参数的数据类型

scanf函数的第一个参数也是格式字符串,可能包含:

1. 普通字符，比如空格和其他字符，scanf函数会期望输入中有与之匹配的字符。一般来说，格式字符串不需要普通字符。所以普通字符怎么写,就意味着在匹配时,就意味着输入数据时 要怎么输入

格式字符串是"%d, %d"
当然普通字符中比较特殊的是空格,如果写了一个空格,他会跳过所有的空格 包括回车 换行 制表等

1. 转换说明，以字符"%" 开头，它告诉scanf函数应该如何解释输入中的数据并如何存储它。在上述示例中，"%d"和"%f"就是转换说明。转换说明中的说明符 两个函数基本上是一样的
2. 注意 在键盘录入整数和浮点数时 也就是说说明符是d和f时 自动跳过空格
3. 注意 匹配过程中一旦匹配失败 后面都不会继续匹配了 函数会直接返回
4. **转换说明scanf的说明符i和printf是不同的, i是可以自动匹配十进制 八进制 十六进制的整数, d只能匹配十进制**
5. **键盘录入char字符时的特殊性 它不会像录入整数和浮点数一样 跳过空格 如果需要跳过空格 需要自己在格式字符串中添加普通字符 空格 以跳过输入时所有的空格**

scanf函数本质上是一个"模式匹配"函数，试图把**"stdin缓冲区"**中的字符与格式字符串匹配。

1. 首先，C 标准规定了整数类型的最小字节长度：short(2)、int(2)、long(4)、long long(8)
2. 其次，C 标准规定了各个整数类型的字节长度满足下面的关系：short <= int <= long <= long long

有符号整数的取值范围
[-2^(位数-1), 2^(位数-1) -1]
无符号整数的取值范围
[0, 2^(位数)-1]

补码有以下两个重要的特性（假设x是一个n位整数）：

1. x + (-x) = 1, 000...$0_{2进制补码}$(其中0一共有n个，高位的1溢出被舍掉) = 0

2. x + (~x) = 111....$1_{2进制补码}$(其中1一共有n个) = -1

a = 10
补码: 0000 1010
-a = -10
补码: 1111 0110
+
1 0000 0000 溢出 结果就是0

给出一个整数: 1010 0011(补码)
求它相反数的补码
0101 1101

~按位取反,就是将二进制位的每一个数都取反
0000 1010
1111 0101
+
1111 1111
+
0000 0001
=
1 0000 0000
这个数就 = -1