

- 一、前言
- 二、操作过程
  - 1. 第一个版本的库
  - 2. 第二个版本的库
- 三 \_Pragma 其他用法
  - 1. 处理头文件重复包含
  - 2. 输出编译信息

## 一、前言

想象一下这个工作场景：你在为一个项目写一个**功能库**，别人调用库中提供的函数，后来你发现库里的函数A是**多余**的。

具有完美情节的你，就是想把这个函数A**废弃掉**，此时肯定是不能直接**删掉**，因为你不知道别人在多少个地方调用了这个函数。

这种情况如何处理比较好呢？

这篇小短文就来聊一聊这个问题。

## 二、操作过程

### 1. 第一个版本的库

测试文件只有**3**个：api.h, api.c 和 main.c

- 1. api.h 和 api.c: 库文件，编译得到 libapi.so;
- 2. main.c: 生成可执行程序，利用了上面生成的库 libapi.so;

api.h 文件内容：**声明**了 2 个函数。

```
_Pragma("once")

#define API_VERSION    1

extern void init();
extern void doWork();
```

api.c 文件内容：**定义**了 2 个函数。

```

#include <stdio.h>
#include "api.h"

void init()
{
    printf("init... \n");
}

void doWork()
{
    printf("doWork ... \n");
}

```

编译得到库文件 `libapi.so`。编译指令：

```
gcc -fPIC -shared api.c -o libapi.so
```

main.c 文件内容：

```

#include <stdio.h>
#include "api.h"

int main()
{
    init();
    doWork();
    return 0;
}

```

编译得到可执行文件：

```
gcc main.c -o main -L./ -Wl,-rpath=./ -lapl
```

以上代码的简单程度，等价于 helloworld 了。

## 2. 第二个版本的库

现在，你觉得 `init` 这个函数是多余的，想把它去掉，可以这么来修改。

`api.c` 文件中，把 `init()` 函数删除掉。

`api.h` 文件内容改为如下：

```

#pragma("once")

#define API_VERSION      2

//extern void init(); // 废弃不用了
extern void doWork();

#define STRINGIFY(contents)    #contents
#define API_DEPRECATED         _Pragma(STRINGIFY
(GCC warning "Do not use this function!"))

#if API_VERSION > 1

#define init()                 (1) API_DEPRECATED

#endif

```

关键代码是这一行：

```
#define init()                (1) API_DEPRECATED
```

既然 api.c 文件已经把这个函数删除了，但是 main.c 文件中又调用了这个函数，因此以宏定义的形式提供 init 这个符号。

也就是说：

在第一个版本中，main.c 文件中的 init 是一个函数，被编译器处理，在链接阶段从 libapi.so 库中找到这个函数的地址；

在第二个版本中，init 被定义成宏，在预处理阶段被替换成后面的 (1) API\_DEPRECATED。

(1) 是在宏替换时的表达式。因为这个函数可能被用在 if 条件判断中，因此需要返回一个值。API\_DEPRECATED 是另一个宏定义，展开开后就是让编译器在编译可执行程序时，打印出一段提示信息。

在编译可执行文件时，编译器输出下面的这段话：

```
gcc main.c -o main -L./ -wl,-rpath=./ -lapl
```

```

main.c: In function 'main':
main.c:6:13: warning: Do not use this function!

```

这样就达到了最初的目的！也就是提示使用者：这个函数已经被废弃了，最好别用它！

## 三 \_Prama 其他用法

\_Pragma 类似于 Microsoft 特定的 \_\_pragma 关键字，只不过它是标准的一部分。它是在 C99 中为 C 引入的。对于 c++，它是在 c++ 11 中引入的。它允许将指令放入宏定义中。

### 1. 处理头文件重复包含

在头文件中，为了防止被重复包含，一般有 3 种处理方式：

(1) 第一种处理方式：

```
#ifndef MY_API
#define MY_API

// 头文件内容

#endif
```

## (2) 第二种处理方式

```
#pragma once

// 头文件内容
```

以上这 2 种方式都可以防止同一个头文件被重复包含，但是还是有一些区别的。

第一种方式：预处理器还是需要去**搜寻**文件，然后**打开**文件，**读取**文件的内容之后，检查 MY\_API 是否已经被定义过。

第二种方式：能加快编译速度，因为这是一种高端的机制；编译器会自动**比对文件名**，而不需要在头文件去判断 #ifndef 和 #endif，这样就**省去了**中间的搜寻、打开和读取操作。

## (3) 第三种处理方式

```
_Pragma("once")
```

这种方式与第二种方式的**区别**是：

#pragma：是一条预处理的指令，用来向编译器传达语言标准以外的一些信息，不能使用在宏中；

\_Pragma：是一个操作符，属于语言的标准，因此可以嵌套在宏中，就像上面示例中那样；

#pragma 是编译器的扩展，也就是说它是由**编译器来决定的**，也许编译器A支持，但是编译器B就**不一定**支持了，虽然这种可能性比较小。

\_Pragma 操作符是语言层面的**标准**，既然是标准，那么编译器就**必须要遵循标准**，所以也推荐使用这种方式。

记得侯杰老师在 C++ 的视频课程中说到：我们写代码，不仅仅要保证**功能**上的正确，而且要把代码写的很**大气**！我感觉用 \_Pragma 可能比 #ifndef 更大气一些。

## 2. 输出编译信息

```
#pragma message("the #pragma way")
_Pragma ("message( \"the _Pragma way\")")
```

上面两行的内容输出信息是一样的，需要注意的是嵌套的双引号需要用反斜线去转义。

That's All! 周末愉快！

---

好文章，**要转发**；越分享，越幸运！

**星标公众号**，能更快找到我！

---



添加“道哥”个人微信，  
加入技术交流群。



公号：IOT物联网小镇，  
关注 + 星标。

## 推荐阅读

### 【C 语言】

- [1. C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻](#)
- [2. 原来gdb的底层调试原理这么简单](#)
- [3. 一步步分析-如何用C实现面向对象编程](#)
- [4. 提高代码逼格的利器：宏定义-从入门到放弃](#)
- [5. 利用C语言中的setjmp和longjmp，来实现异常捕获和协程](#)

### 【应用程序设计】

- [1. 都说软件架构要分层、分模块，具体应该怎么做\(一\)](#)
- [2. 都说软件架构要分层、分模块，具体应该怎么做\(二\)](#)
- [3. 物联网网关开发：基于MQTT消息总线的设计过程\(上\)](#)
- [4. 物联网网关开发：基于MQTT消息总线的设计过程\(下\)](#)
- [5. 我最喜欢的进程之间通信方式-消息总线](#)

### 【操作系统】

- [1. 为什么航天器、导弹喜欢用单片机，而不是嵌入式系统？](#)

### 【物联网】

- [1. 关于加密、证书的那些事](#)
- [2. 深入LUA脚本语言，让你彻底明白调试原理](#)

### 【胡说八道】

- [1. 以我失败的职业经历：给初入职场的技术人员几个小建议](#)

