作 者：道哥，10+年嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。

目录

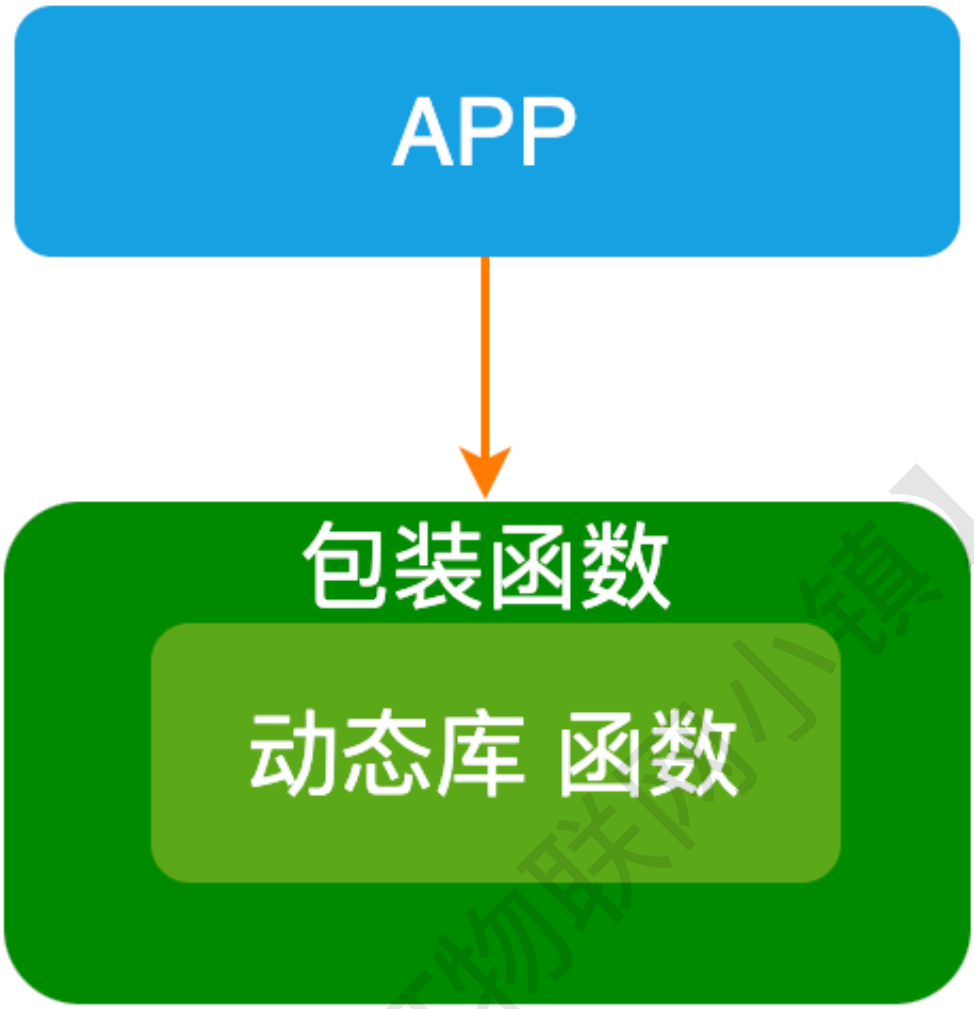
别人的经验，我们的阶梯！

什么是插桩?

在稍微具有一点规模的代码中(C 语言)，调用第三方动态库中的函数来完成一些功能，是很常见的工作场景。

假设现在有一项任务：需要在调用某个动态库中的某个函数的之前和之后，做一些额外的处理工作。

这样的需求一般称作：插桩，也就是对于一个指定的目标函数，我们新建一个包装函数，来完成一些额外的功能。



在包装函数中去调用真正的目标函数，但是在调用之前或者之后，可以做一些额外的事情。

比如：统计函数的调用次数、验证函数的输入参数是否合法等等。

关于程序插桩的官方定义，可以看一下【百度百科】中的描述：

> 1. 程序插桩，最早是由J.C. Huang 教授提出的。
> 2. 它是在保证被测程序原有逻辑完整性的基础上在程序中插入一些探针（又称为“探测仪”，本质上就是进行信息采集的代码段，可以是赋值语句或采集覆盖信息的函数调用）。
> 3. 通过探针的执行并抛出程序运行的特征数据，通过对这些数据的分析，可以获得程序的控制流和数据流信息，进而得到逻辑覆盖等动态信息，从而实现测试目的的方法。
> 4. 根据探针插入的时间可以分为目标代码插桩和源代码插桩。

这篇文章，我们就一起讨论一下：在 Linux 环境下的 C 语言开发中，可以通过哪些方法来实现插桩功能。

插桩示例代码分析

示例代码很简单：

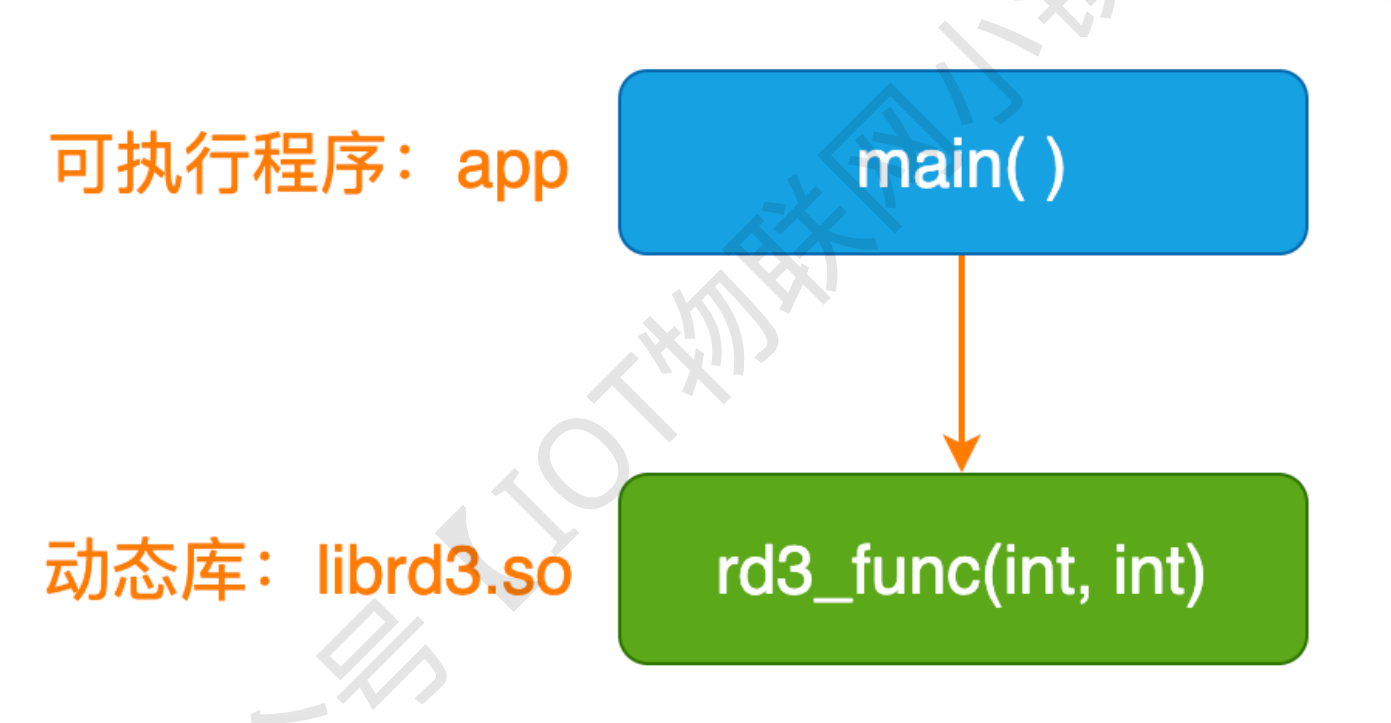
```
├── app.c
└── lib
    ├── rd3.h
    └── librd3.so
```

假设动态库librd3.so是由第三方提供的，里面有一个函数：int rd3_func(int, int);。

```
// lib/rd3.h

#ifndef _RD3_H_
#define _RD3_H_
extern int rd3_func(int, int);
#endif
```

在应用程序app.c中，调用了动态库中的这个函数:



app.c代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include "rd3.h"

int main(int argc, char *argv[])
{
    int result = rd3_func(1, 1);
    printf("result = %d \n", result);
    return 0;
}
```

编译：

```
$ gcc -o app app.c -I./lib -L./lib -lrd3 -Wl,--rpath=./lib
```

1. -L./lib: 指定编译时，在 lib 目录下搜寻库文件。
2. -Wl,--rpath=./lib: 指定执行时，在 lib 目录下搜寻库文件。

生成可执行程序：app，执行：

```
$ ./app
result = 3
```

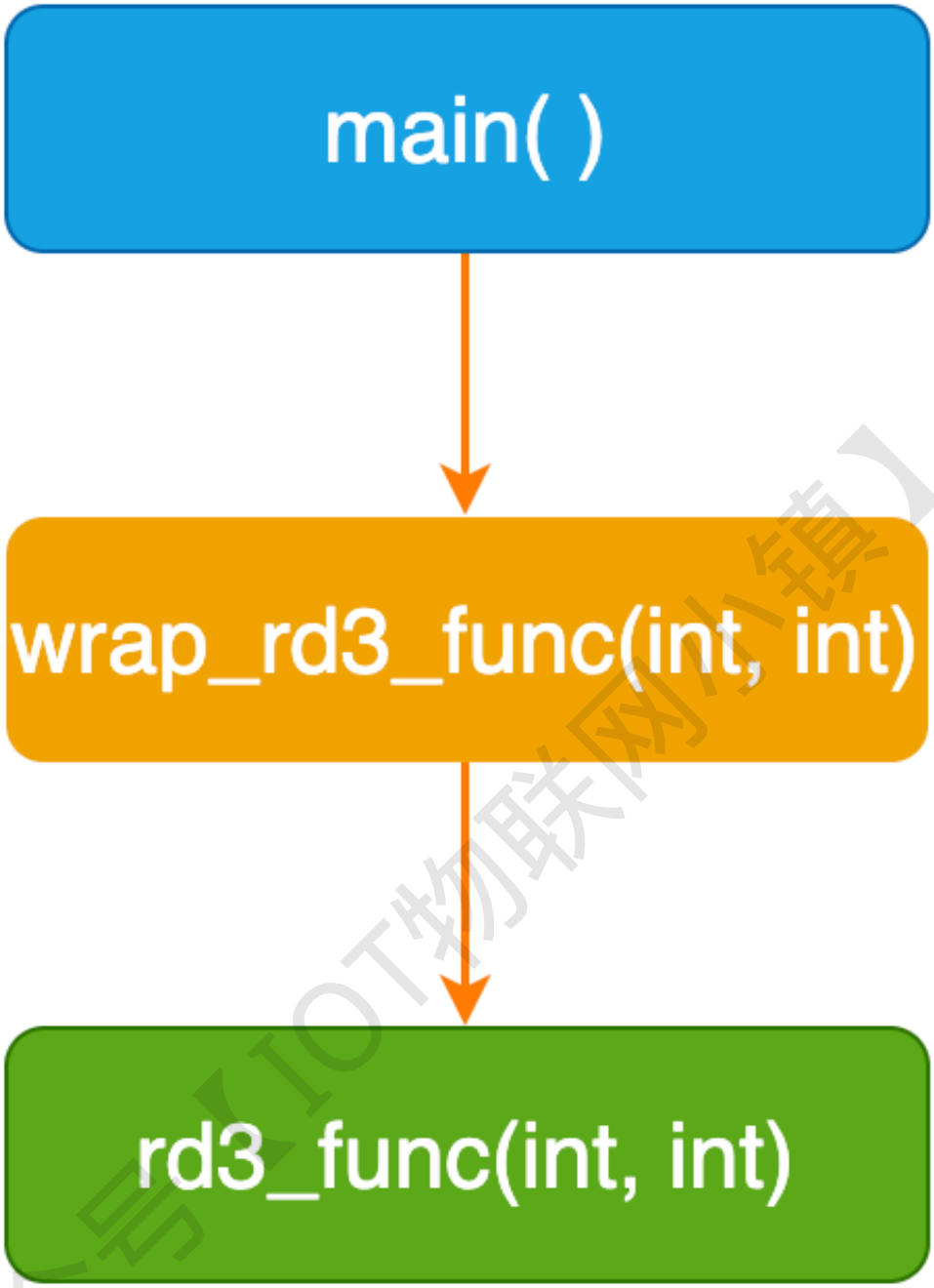
示例代码足够简单了，称得上是helloworld的兄弟版本！

在编译阶段插桩

对函数进行插桩，基本要求是：不应该对原来的文件(app.c)进行额外的修改。

由于app.c文件中，已经include "rd3.h"了，并且调用了其中的rd3_func(int, int)函数。

所以我们需要新建一个假的 "rd3.h" 提供给app.c，并且要把函数rd3_func(int, int)"重导向"到一个包装函数，然后在包装函数中去调用真正的目标函数，如下图所示：



"重导向"函数：可以使用宏来实现。

包装函数：新建一个C文件，在这个文件中，需要#include "lib/rd3.h"，然后调用真正的目标文件。

完整的文件结构如下：

```
├── app.c
├── lib
│   ├── librd3.so
│   └── rd3.h
├── rd3.h
└── rd3_wrap.c
```

最后两个文件是新建的：rd3.h, rd3_wrap.c，它们的内容如下：

```
// rd3.h

#ifndef _LIB_WRAP_H_
#define _LIB_WRAP_H_

// 函数“重导向”，这样的话 app.c 中才能调用 wrap_rd3_func
#define rd3_func(a, b)   wrap_rd3_func(a, b)

// 函数声明
extern int wrap_rd3_func(int, int);

#endif
```

```
// rd3_wrap.c

#include <stdio.h>
#include <stdlib.h>

// 真正的目标函数
#include "lib/rd3.h"

// 包装函数，被 app.c 调用
int wrap_rd3_func(int a, int b)
{
    // 在调用目标函数之前，做一些处理
    printf("before call rd3_func. do something... \n");

    // 调用目标函数
    int c = rd3_func(a, b);

    // 在调用目标函数之后，做一些处理
    printf("after call rd3_func. do something... \n");

    return c;
}
```

让app.c 和 rd3_wrap.c一起编译：

```
$ gcc -I./ -L./lib -Wl,--rpath=./lib -o app app.c rd3_wrap.c -lrd3
```

头文件的搜索路径不能错：必须在当前目录下搜索rd3.h，这样的话，app.c中的#include "rd3.h"找到的才是我们新增的那个头文件 rd3.h。

所以在编译指令中，第一个选项就是 -I./，表示在当前目录下搜寻头文件。

另外，由于在rd3_wrap.c文件中，使用#include "lib/rd3.h"来包含库中的头文件，因此在编译指令中，就不需要指定到lib目录下去查找头文件了。

编译得到可执行程序app，执行一下：

```
$ ./app
before call rd3_func. do something...
after call rd3_func. do something...
result = 3
```

完美！

链接阶段插桩

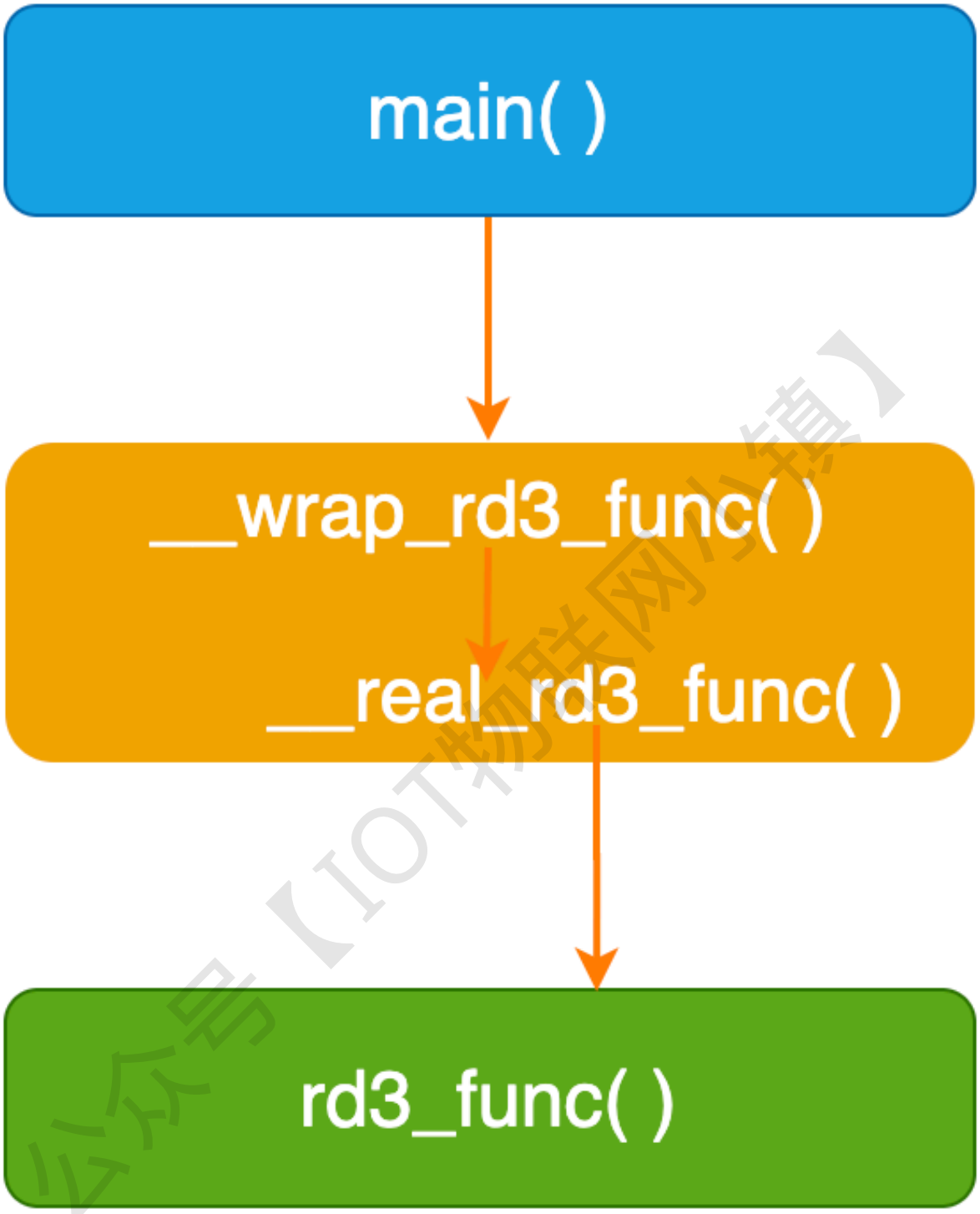
Linux系统中的链接器功能是非常强大的，它提供了一个选项：`--wrap f`，可以在链接阶段进行插桩。

这个选项的作用是：告诉链接器，遇到f符号时解析成__wrap_f，在遇到__real_f符号时解析成f，正好是一对！

我们就可以利用这个属性，新建一个文件rd3_wrap.c，并且定义一个函数__wrap_rd3_func(int, int)，在这个函数中去调用__real_rd3_func函数。

只要在编译选项中加上`-Wl,--wrap,rd3_func`, 编译器就会：

1. 把 app.c 中的 rd3_func 符号，解析成 __wrap_rd3_func，从而调用包装函数;
2. 把 rd3_wrap.c 中的 __real_rd3_func 符号，解析成 rd3_func，从而调用真正的函数。



这几个符号的转换，是由链接器自动完成的！

按照这个思路，一起来测试一下。

文件目录结构如下：

```
.
├── app.c
├── lib
│   ├── librd3.so
│   └── rd3.h
├── rd3_wrap.c
└── rd3_wrap.h
```

rd3_wrap.h是被app.c引用的，内容如下：

```
#ifndef _RD3_WRAP_H_
#define _RD3_WRAP_H_
extern int __wrap_rd3_func(int, int);
#endif
```

rd3_wrap.c的内容如下：

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

#include "rd3_wrap.h"

// 这里不能直接饮用 lib/rd3.h 中的函数了，而要由链接器来完成解析。
extern int __real_rd3_func(int, int);

// 包装函数
int __wrap_rd3_func(int a, int b)
{
    // 在调用目标函数之前，做一些处理
    printf("before call rd3_func. do something... \n");

    // 调用目标函数，链接器会解析成 rd3_func。
    int c = __real_rd3_func(a, b);

    // 在调用目标函数之后，做一些处理
    printf("after call rd3_func. do something... \n");

    return c;
}
```

rd3_wrap.c中，不能直接去include "rd3.h"，因为lib/rd3.h中的函数声明是int rd3_func(int, int);，没有__real前缀。

编译一下：

```
$ gcc -I./lib -L./lib -Wl,--rpath=./lib -Wl,--wrap,rd3_func -o app app.c rd3_wrap.c -lrd3
```

注意：这里的头文件搜索路径仍然设置为-I./lib，是因为app.c中include了这个头文件。

得到可执行程序app，执行：

```
$ ./app
before call rd3_func. do something...
before call rd3_func. do something...
result = 3
```

完美！

执行阶段插桩

在编译阶段插桩，新建的文件rd3_wrap.c是与app.c一起编译的，其中的包装函数名是wrap_rd3_func。

app.c中通过一个宏定义实现函数的"重导向"：rd3_func --> wrap_rd3_func。

我们还可以直接"霸王硬上弓"：在新建的文件rd3_wrap.c中，直接定义rd3_func函数。

然后在这个函数中通过dlopen, dlsym系列函数来动态的打开真正的动态库，查找其中的目标文件，然后调用真正的目标函数。

当然了，这样的话在编译app.c时，就不能连接lib/librd3.so文件了。

按照这个思路继续实践！

文件目录结构如下：

```
├── app.c
├── lib
│   ├── librd3.so
│   └── rd3.h
└── rd3_wrap.c
```

rd3_wrap.c文件的内容如下(一些错误检查就暂时忽略了)：

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

// 库的头文件
#include "rd3.h"

// 与目标函数签名一致的函数类型
typedef int (*pFunc)(int, int);

int rd3_func(int a, int b)
{
    printf("before call rd3_func. do something... \n");

    //打开动态链接库
```

```
    void *handle = dlopen("./lib/librd3.so", RTLD_NOW);

    // 查找库中的目标函数
    pFunc pf = dlsym(handle, "rd3_func");

    // 调用目标函数
    int c = pf(a, b);

    // 关闭动态库句柄
    dlclose(handle);

    printf("after call rd3_func. do something... \n");
    return c;
}
```

编译包装的动态库：

```
$ gcc -shared -fPIC -I./lib -o librd3_wrap.so rd3_wrap.c
```

得到包装的动态库： librd3_wrap.so。

编译可执行程序，需要链接包装库librd3_wrap.so：

```
$ gcc -I./lib -L./ -o app app.c -lrd3_wrap -ldl
```

得到可执行程序app，执行：

```
$ ./app
before call rd3_func. do something...
after call rd3_func. do something...
result = 3
```

完美！

------ End ------

文中的测试代码，已经放在网盘了。

在公众号【IOT物联网小镇】后台回复关键字：220109，即可获取下载地址。

原创不易，请支持一下道哥，把文章分享给更多的嵌入式小伙伴，谢谢！

推荐阅读

【1】《Linux 从头学》系列文章

【2】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

【3】原来gdb的底层调试原理这么简单

【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、应用程序设计、物联网、 C语言。

星标公众号，第一时间看文章！

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计

扫码关注公众号

道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。