<https://blog.csdn.net/xiabodan/article/details/47999411>

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/xiabodan/article/details/47999411

本文主要讲解如果实现回调，特别是在封装接口的时候，回调显得特别重要，我们首先假设有两个程序员在写代码，A程序员写底层驱动接口，B程序员写上层应用程序，然而此时底层驱动接口A有一个数据d需要传输给B，此时有两种方式：   
　　　１、A将数据d存储好放在接口函数中，B自己想什么时候去读就什么时候去读，这就是我们经常使用的函数调用，此时主动权是B。   
　　　2、A实现回调机制，当数据变化的时候才将通知B，你可以来读取数据了，然后B在用户层的回调函数中读取速度d，完成OK。此时主动权是A。   
很明显第一种方法太低效了，B根本就不知道什么时候该去调用接口函数读取数据d。而第二种方式由于B的读取数据操作是依赖A的，只有A叫B读数据，那么B才能读数据。也即是实现了中断读取。   
那么回调是怎么实现的呢，其实回调函数就是一个通过函数指针调用的函数。如果用户层B把函数的指针（地址）作为参数传递给底层驱动A，当这个指针在A中被用为调用它所指向的函数时，我们就说这是回调函数。   
**注意**：是在A中被调用，这里看到尽管函数是在B中，但是B却不是自己调用这个函数，而是将这个函数的函数指针通过A的接口函数传自A中了，由A来操控执行，这就是回调的意义所在。   
下面就通过一个例子来演示   
首先写A程序员的代码

//-----------------------底层实现A-----------------------------

typedef void (\*pcb)(int a); //函数指针定义，后面可以直接使用pcb，方便

typedef struct parameter{

int a ;

pcb callback;

}parameter;

void\* callback\_thread(void \*p1)//此处用的是一个线程

{

//do something

parameter\* p = (parameter\*)p1 ;

while(1)

{

printf("GetCallBack print! \n");

sleep(3);//延时3秒执行callback函数

p->callback(p->a);//函数指针执行函数，这个函数来自于应用层B

}

}

//留给应用层B的接口函数

extern SetCallBackFun(int a, pcb callback)

{

printf("SetCallBackFun print! \n");

parameter \*p = malloc(sizeof(parameter)) ;

p->a = 10;

p->callback = callback;

//创建线程

pthread\_t thing1;

pthread\_create(&thing1,NULL,callback\_thread,(void \*) p);

pthread\_join(thing1,NULL);

}

上面的代码就是底层接口程序员A写的全部代码，留出接口函数SetCallBackFun即可

下面再实现应用者B的程序，B负责调用SetCallBackFun函数，以及增加一个函数，并将吃函数的函数指针通过SetCallBackFun(int a, pcb callback)的第二个参数pcb callback 传递下去。

//-----------------------应用者B-------------------------------

void fCallBack(int a) // 应用者增加的函数，此函数会在A中被执行

{

//do something

printf("a = %d\n",a);

printf("fCallBack print! \n");

}

int main(void)

{

SetCallBackFun(4,fCallBack);

return 0;

}

运行程序会看到

先会打印A程序的 printf("GetCallBack print! \n");

然后等待3秒钟才会打印应用者B的 printf("fCallBack print! \n");