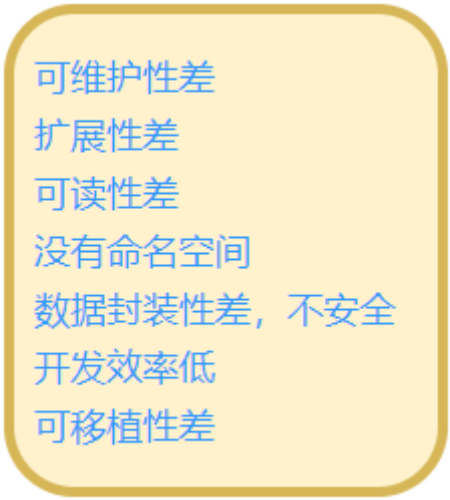
这是道哥的第009篇原创

一、前言

在嵌入式开发中，C/C++语言是使用最普及的，在C++11版本之前，它们的语法是比较相似的，只不过C++提供了面向对象的编程方式。

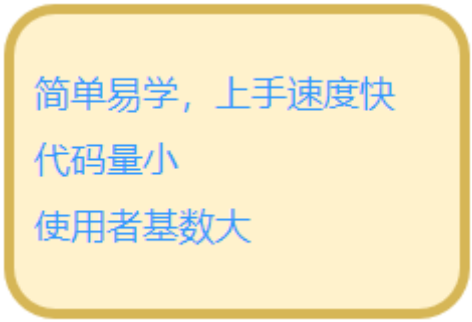
虽然C++语言是从C语言发展而来的，但是今天的C++已经不是当年的C语言的扩展了，从2011版本开始，更像是一门全新的语言。

那么没有想过，当初为什么要扩展出C++？C语言有什么样的缺点导致C++的产生？



C++在这几个问题上的解决的确很好，但是随着语言标准的逐步扩充，C++语言的学习难度也逐渐加大。没有开发过几个项目，都不好意思说自己学会了C++，那些左值、右值、模板、模板参数、可变模板参数等等一堆的概念，真的不是使用2，3年就可以熟练掌握的。

但是，C语言也有很多的优点：



其实最后一个优点是最重要的：使用的人越多，生命力就越强。就像现在的社会一样，不是优者生存，而是适者生存。

这篇文章，我们就来聊聊如何在C语言中利用面向对象的思想来编程。也许你在项目中用不到，但是也强烈建议你看一下，因为我之前在跳槽的时候就两次被问到这个问题。

二、什么是面向对象编程

有这么一个公式：程序=数据结构+算法。

C语言中一般使用面向过程编程，就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步调用，在函数中对数据结构进行处理(执行算法)，也就是说数据结构和算法是分开的。

C++语言把数据和算法封装在一起，形成一个整体，无论是对它的属性进行操作、还是对它的行为进行调用，都是通过一个对象来执行，这就是面向对象编程思想。

如果用C语言来模拟这样的编程方式，需要解决3个问题：

> 1. 数据的封装
> 2. 继承
> 3. 多态

第一个问题：封装

封装描述的是数据的组织形式，就是把属于一个对象的所有属性(数据)组织在一起，C语言中的结构体类型天生就支持这一点。

第二个问题：继承

继承描述的是对象之间的关系，子类通过继承父类，自动拥有父类中的属性和行为(也就是方法)。这个问题只要理解了C语言的内存模型，也不是问题，只要在子类结构体中的第一个成员变量的位置放置一个父类结构体变量，那么子类对象就继承了父类中的属性。

另外补充一点：学习任何一种语言，一定要理解内存模型！

第三个问题：多态

按字面理解，多态就是“多种状态”，描述的是一种动态的行为。在C++中，只有通过基类引用或者指针，去调用虚函数的时候才发生多态，也就是说多态是发生在运行期间的，C++内部通过一个虚表来实现多态。那么在C语言中，我们也可以按照这个思路来实现。

> 如果一门语言只支持类，而不支持多态，只能说它是基于对象的，而不是面向对象的。

既然思路上没有问题，那么我们就来简单的实现一个。

三、先实现一个父类，解决封装的问题

Animal.h

```
#ifndef _ANIMAL_H_
#define _ANIMAL_H_

// 定义父类结构
typedef struct {
    int age;
    int weight;
} Animal;

// 构造函数声明
void Animal_Ctor(Animal *this, int age, int weight);

// 获取父类属性声明
int Animal_GetAge(Animal *this);
int Animal_GetWeight(Animal *this);

#endif
```

Animal.c

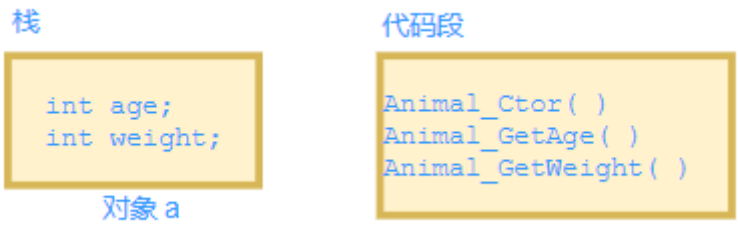
```
#include "Animal.h"

// 父类构造函数实现
void Animal_Ctor(Animal *this, int age, int weight)
{
    this->age = age;
    this->weight = weight;
}
```

```
int Animal_GetAge(Animal *this)
{
    return this->age;
}

int Animal_GetWeight(Animal *this)
{
    return this->weight;
}
```

测试一下：

```
#include <stdio.h>
#include "Animal.h"
#include "Dog.h"

int main()
{
    // 在栈上创建一个对象
    Animal a;
    // 构造对象
    Animal_Ctor(&a, 1, 3);
    printf("age = %d, weight = %d \n",
            Animal_GetAge(&a),
            Animal_GetWeight(&a));
    return 0;
}
```

可以简单的理解为：在代码段有一块空间，存储着可以处理Animal对象的函数；在栈中有一块空间，存储着a对象。



与C++对比：
在C++的方法中，隐含着第一个参数this指针。当调用一个对象的方法时，编译器会自动把对象的地址传递给这个指针。

所以，在Animal.h中函数我们就模拟一下，显示的定义这个this指针，在调用时主动把对象的地址传递给它，这样的话，函数就可以对任意一个Animal对象进行处理了。

四、 实现一个子类，解决继承的问题

Dog.h

```
#ifndef _DOG_H_
#define _DOG_H_

#include "Animal.h"
```

```
// 定义子类结构
typedef struct {
    Animal parent; // 第一个位置放置父类结构
    int legs;      // 添加子类自己的属性
}Dog;

// 子类构造函数声明
void Dog_Ctor(Dog *this, int age, int weight, int legs);

// 子类属性声明
int Dog_GetAge(Dog *this);
int Dog_GetWeight(Dog *this);
int Dog_GetLegs(Dog *this);

#endif
```

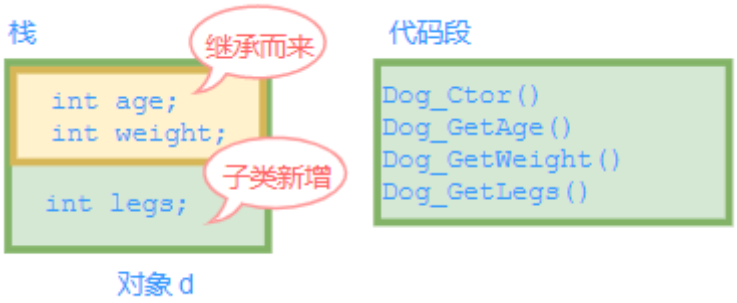
Dog.c

```
#include "Dog.h"

// 子类构造函数实现
void Dog_Ctor(Dog *this, int age, int weight, int legs)
{
    // 首先调用父类构造函数，来初始化从父类继承的数据
    Animal_Ctor(&this->parent, age, weight);
    // 然后初始化子类自己的数据
    this->legs = legs;
}

int Dog_GetAge(Dog *this)
{
    // age属性是继承而来，转发给父类中的获取属性函数
    return Animal_GetAge(&this->parent);
}

int Dog_GetWeight(Dog *this)
{
    return Animal_GetWeight(&this->parent);
}

int Dog_GetLegs(Dog *this)
{
    // 子类自己的属性，直接返回
    return this->legs;
}
```

测试一下：

```
int main()
{
    Dog d;
    Dog_Ctor(&d, 1, 3, 4);
    printf("age = %d, weight = %d, legs = %d \n",
            Dog_GetAge(&d),
            Dog_GetWeight(&d),
            Dog_GetLegs(&d));
    return 0;
}
```

在代码段有一块空间，存储着可以处理Dog对象的函数；在栈中有一块空间，存储着d对象。由于Dog结构体中的第一个参数是Animal对象，所以从内存模型上看，子类就包含了父类中定义的属性。



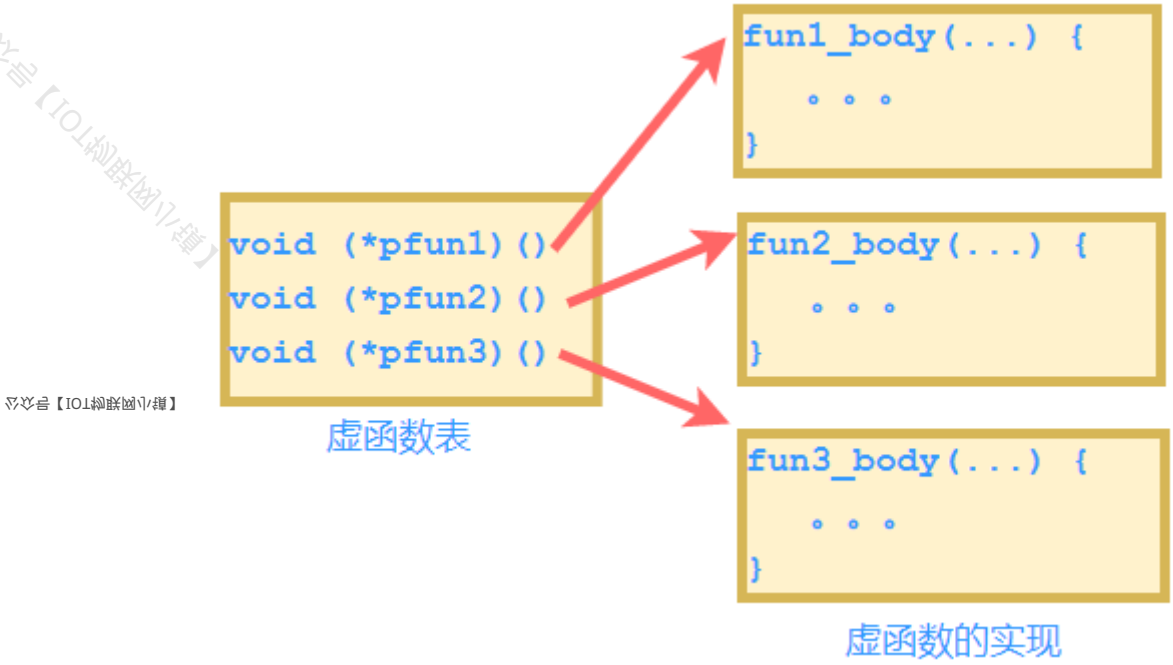
Dog的内存模型中开头部分就自动包括了Animal中的成员，也即是说Dog继承了Animal的属性。

五、利用虚函数，解决多态问题

在C++中，如果一个父类中定义了虚函数，那么编译器就会在这个内存中开辟一块空间放置虚表，这张表里的每一个item都是一个函数指针，然后在父类的内存模型中放一个虚表指针，指向上面这个虚表。

> 上面这段描述不是十分准确，主要看各家编译器的处理方式，不过大部分C++处理器都是这么干的，我们可以想这么理解。

子类在继承父类之后，在内存中又会开辟一块空间来放置子类自己的虚表，然后让继承而来的虚表指针指向子类自己的虚表。



既然C++是这么做的，那我们就用C来手动模拟这个行为：创建虚表和虚表指针。

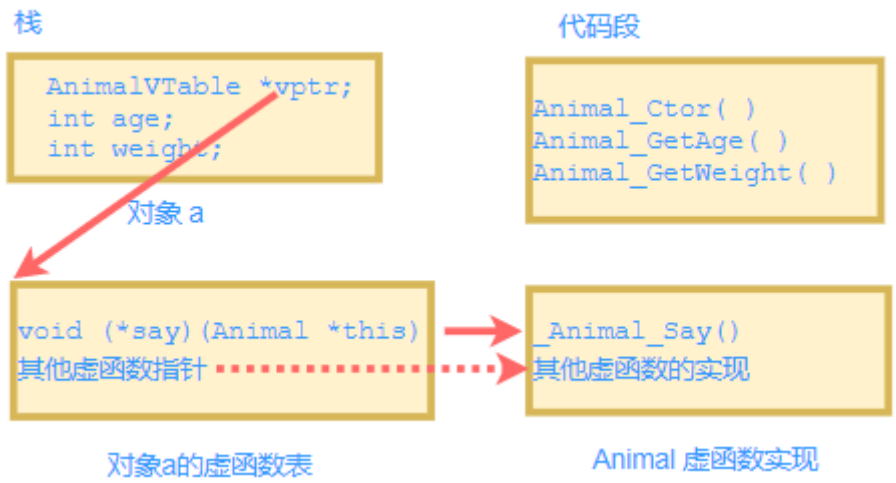
1. Animal.h为父类Animal中，添加虚表和虚表指针

```
#ifndef _ANIMAL_H_
#define _ANIMAL_H_

struct AnimalVTable;  // 父类虚表的前置声明

// 父类结构
typedef struct {
    struct AnimalVTable *vptr; // 虚表指针
    int age;
    int weight;
} Animal;

// 父类中的虚表
struct AnimalVTable{
    void (*say)(Animal *this); // 虚函数指针
};

// 父类中实现的虚函数
void Animal_Say(Animal *this);

#endif
```

2. Animal.c

```
#include <assert.h>
#include "Animal.h"

// 父类中虚函数的具体实现
static void _Animal_Say(Animal *this)
{
```

```
    // 因为父类Animal是一个抽象的东西，不应该被实例化。
    // 父类中的这个虚函数不应该被调用，也就是说子类必须实现这个虚函数。
    // 类似于C++中的纯虚函数。
    assert(0);
}

// 父类构造函数
void Animal_Ctor(Animal *this, int age, int weight)
{
    // 首先定义一个虚表
    static struct AnimalVTable animal_vtbl = {_Animal_Say};
    // 让虚表指针指向上面这个虚表
    this->vptr = &animal_vtbl;
    this->age = age;
    this->weight = weight;
}

// 测试多态：传入的参数类型是父类指针
void Animal_Say(Animal *this)
{
    // 如果this实际指向一个子类Dog对象，那么this->vptr这个虚表指针指向子类自己的虚表，
    // 因此，this->vptr->say将会调用子类虚表中的函数。
    this->vptr->say(this);
}
```



> 在栈空间定义了一个虚函数表animal_vtbl，这个表中的每一项都是一个函数指针，例如：函数指针say就指向了代码段中的函数_Animal_Say()。
> 对象a的第一个成员vptr是一个指针，指向了这个虚函数表animal_vtbl。

3. Dog.h不变

4. Dog.c中定义子类自己的虚表

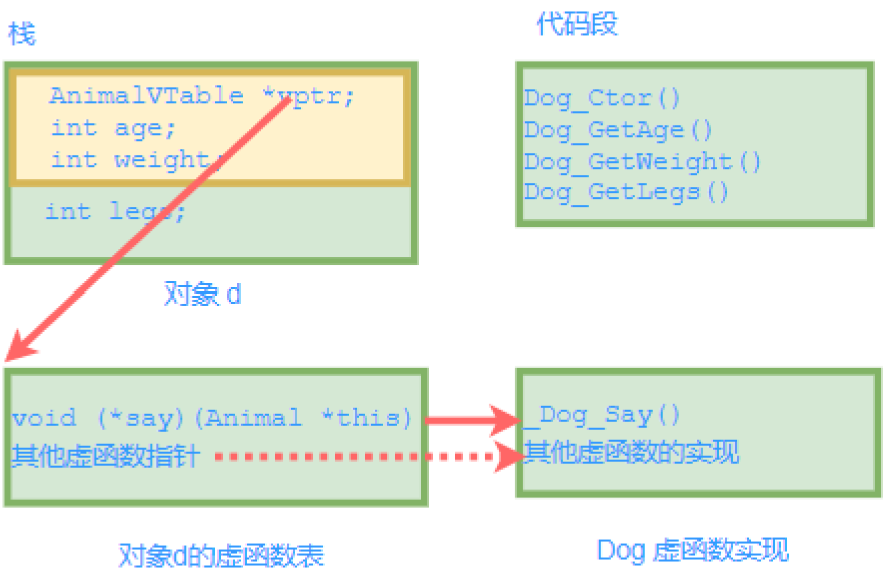
```
#include "Dog.h"

// 子类中虚函数的具体实现
static void _Dog_Say(Dog *this)
{
    printf("dag say \n");
}
```

```
// 子类构造函数
void Dog_Ctor(Dog *this, int age, int weight, int legs)
{
    // 首先调用父类构造函数。
    Animal_Ctor(&this->parent, age, weight);
    // 定义子类自己的虚函数表
    static struct AnimalVTable dog_vtbl = {_Dog_Say};
    // 把从父类中继承得到的虚表指针指向子类自己的虚表
    this->parent.vptr = &dog_vtbl;
    // 初始化子类自己的属性
    this->legs = legs;
}
```

5. 测试一下

```
int main()
{
    // 在栈中创建一个子类Dog对象
    Dog d;
    Dog_Ctor(&d, 1, 3, 4);

    // 把子类对象赋值给父类指针
    Animal *pa = &d;

    // 传递父类指针，将会调用子类中实现的虚函数。
    Animal_Say(pa);
}
```

内存模型如下:



> 对象d中，从父类继承而来的虚表指针vptr，所指向的虚表是dog_vtbl。

在执行**Animal_Say(pa)**的时候，虽然参数类型是指向父类Animal的指针，但是实际传入的pa是一个指向子类Dog的对象，这个对象中的虚表指针vptr指向的是子类中自己定义的虚表dog_vtbl，这个虚表中的函数指针say指向的是子类中重新定义的虚函数_Dog_Say，因此this->vptr->say(this)最终调用的函数就是_Dog_Say。

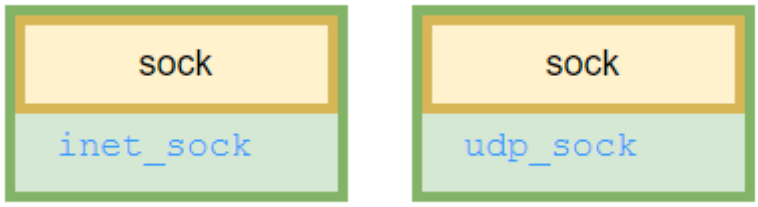
基本上，在C中面向对象的开发思想就是以上这样。
这个代码很简单，自己手敲一下就可以了。如果想偷懒，请在后台留言，我发给您。

六、C面向对象思想在项目中的使用

1. Linux内核

看一下关于socket的几个结构体：

```
struct sock {
    ...
}

struct inet_sock {
    struct sock sk;
    ...
};

struct udp_sock {
    struct sock sk;
    ...
};
```



sock可以看作是父类，inet_sock和udp_sock的第一个成员都是是sock类型，从内存模型上看相当于是继承了sock中的所有属性。

2. glib库

以最简单的字符串处理函数来举例：

> GString *g_string_truncate(GString *string, gint len)
> GString *g_string_append(GString *string, gchar *val)
> GString *g_string_prepend(GString *string, gchar *val)
> ...

API函数的第一个参数都是一个GString对象指针，指向需要处理的那个字符串对象。

```
GString *s1, *s2;
s1 = g_string_new("Hello");
s2 = g_string_new("Hello");

g_string_append(s1," World!");
g_string_append(s2," World!");
```

3. 其他项目

还有一些项目，虽然从函数的参数上来看，似乎不是面向对象的，但是在数据结构的设计上看来，也是面向对象的思想，比如：

> Modbus协议的开源库libmodbus
> 用于家庭自动化的无线通讯协议ZWave
> 很久之前的高通手机开发平台BREW

【原创声明】

> 作者：道哥(公众号: IOT物联网小镇)
>
> 知乎：道哥
>
> B站：道哥分享
>
> 掘金：道哥分享
>
> CSDN：道哥分享

公众号【IOT物联网小镇】

如果觉得文章不错，请转发、分享给您的朋友。

我会把十多年嵌入式开发中的项目实战经验进行总结、分享，相信不会让你失望的！

长按下图二维码关注，花20秒钟了解一下也没什么损失，万一是个适合你的宝藏公众号呢！

推荐阅读

[1] 原来gdb的底层调试原理这么简单
[2] 生产者和消费者模式中的双缓冲技术
[3] 深入LUA脚本语言，让你彻底明白调试原理