**Goodman Tao**

taoxing2113@foxmail.com

**OopC**

C语言面向对象特性的支持库

无比崇敬的Isaac Newton爵士

**目录**

[1. 起因 2](#_Toc20421105)

[2. 源码 2](#_Toc20421106)

[3. Hello, world! 3](#_Toc20421107)

[4. Object 6](#_Toc20421108)

[5. OOP 7](#_Toc20421109)

[5.1. 封装 7](#_Toc20421115)

[5.2. 继承 8](#_Toc20421116)

[5.3. 多态 8](#_Toc20421117)

[6. 特性 8](#_Toc20421118)

[7. 规范 8](#_Toc20421119)

[8. API汇总 8](#_Toc20421120)

[9. 原理 8](#_Toc20421121)

[10. 设计模式实例 8](#_Toc20421122)

[11. 感谢 8](#_Toc20421123)

[12. 一起 8](#_Toc20421124)

[13. 数据结构 8](#_Toc20421125)

[7.1. 方法 8](#_Toc20421128)

[7.2. 方法环 9](#_Toc20421129)

[7.3. 实例 10](#_Toc20421130)

[7.4. 实例链 10](#_Toc20421131)

[14. API 11](#_Toc20421132)

[15. Object 12](#_Toc20421133)

# 起因

“OopC”是“Object Oriented Programming in C”的缩写。应该归咎于我的偏执吧，我特别喜欢C语言简约的语法和高效的性能，但很遗憾的是它没有面向对象的特性。我曾试图深入理解和学习C++，但是它太过庞大，这显然和驻留在我头脑里的观点是不一致的。优雅的事物应该是简单而又强大的，或者说用相对简单的事物取得更大的效益，或者说我所认为的最优化。

事实上不乏前辈追求或追求过同样的事情。和他们的追求或许有些不同，我的另一个目标是实用，而不是完全空洞的优雅，在某些时候，我认为实用超过优雅。这也在OopC的代码中有所体现——提供面向对象特性的支持，而非达到“所有的都是对象”的目的。

# 源码

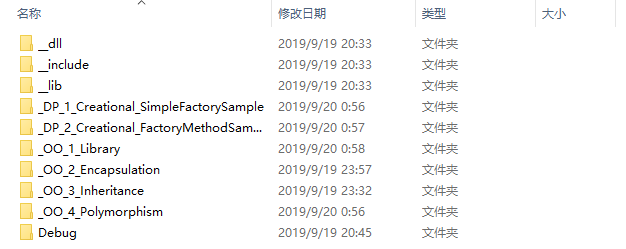
OopC的源代码托管在GitHub上，<https://github.com/GoodmanTao/OopC.git>，编译环境为Visual Studio 2017，使用标准C99实现。采用MIT授权，也就是说，如果我的代码有幸能供您使用，还望您将软件著作权声明的证书附加到所使用的项目中，此外，没有其它的要求了。如果能够通知我一下，您正在使用这些代码，我会非常高兴的。

代码的目录如图2.1，



图2.1

doc/ 包含项目的说明文档。OopC.sln为项目解决方案。OopC/ 包含与项目核心库代码和相关的示例工程，如下图2.2，

图2.2

文件夹名称前缀“OO”表示“Object Oriented”。\_OO\_1\_Library/ 包含OopC的核心代码，该工程在编译后输出OopBase.dll至\_\_dll/，输出OopBase.lib至\_\_lib/，复制OopBase.h至\_\_include/。其余三个以“OO”开头的目录，正如其名称后缀一样，为示例代码，分别演示OopC的封装、继承和多态特性。“DP”前缀表示“Design Pattern”，以此前缀开头的文件夹表示该文件夹下的工程演示一种设计模式的实现，比如上图中“\_DP\_1\_Creational\_SimpleFactorySample”，该工程演示设计模式中的创建型模式，具体为简单工厂模式。到文档撰写日止，只完成了两个模式的示例代码的编写，后面会持续更新。文件夹中的数字表示推荐阅读顺序，个人推荐先按序阅读“OO”部分，然后阅读“DP”部分。

# Hello, world!

照例，我们先体验一下OopC的“Hello, world!”，当然不是演示打印一条语句，这里通过该例子，展示OopC是如何支持C语言面向对象特性的。

* HelloWorld.h

1 #ifndef HELLOWORLD\_H\_\_

2 #define HELLOWORLD\_H\_\_

3 #include <OopBase.h>

4

5 CLASSDEF(HelloWorld)

6

7 typedef ParamNull HelloWorld\_Print;

8

9 #endif // !HELLOWORLD\_H\_\_

* HelloWorld.c

1 #include "HelloWorld.h"

2

3 #include <malloc.h>

4 #include <stdio.h>

5

6 struct HelloWorld

7 {

8 CHAINDEF;

9 };

10

11 /////////////////////////////////////////////////////////////

12 //

13

14 static void Print(void \*pParams)

15 {

16 HelloWorld \*pThis = ((ParamIn \*)pParams)->pInst;

17 HelloWorld\_Print \*pIn = ((ParamIn\*)pParams)->pIn;

18

19 printf("Hello, world!.\n");

20 }

21

22 /////////////////////////////////////////////////////////////

23 //

24

25 void INVOKE(HelloWorld)(HelloWorld \*pInst, char \*pFuncName,

26 void \*pParams)

27 {

28 DOINVOKE(pInst, pFuncName, pParams);

29 }

30

31 void \*EXTEND(HelloWorld)(HelloWorld \*pInst)

32 {

33 return pInst->pChain;

34 }

35

36 void DELETE(HelloWorld)(HelloWorld \*\*ppInst)

37 {

38 Object \*pSuper = SWITCH((\*ppInst), HelloWorld, Object);

39 DELETE(Object)(&pSuper);

40 \*ppInst = NULL;

41 }

42

43 HelloWorld \*CREATE(HelloWorld)()

44 {

45 HelloWorld \*pCreate = malloc(sizeof(HelloWorld));

46 if (!pCreate) { return NULL; }

47

48 MethodRing \*pMethods = GenerateMethodRing();

49 if (!pMethods) { return NULL; }

50

51 pMethods = InsertMethod(pMethods, 1,

52 GenerateMethod(Print, "Print"));

53 pCreate->pChain =

54 InsertInstance(EXTEND(Object)(CREATE(Object)()),

55 GenerateInstance(pCreate, "HelloWorld", NULL, pMethods));

56

57 return pCreate;

58 }

* main.c

1 #include "HelloWorld.h"

2

3 int main(int argc, char \*\*argv)

4 {

5 HelloWorld \*pHllWrld = CREATE(HelloWorld)();

6

7 INVOKE(HelloWorld)(pHllWrld, "Print", NULL);

8

9 DELETE(HelloWorld)(&pHllWrld);

10

11 return 0;

12 }

头文件HelloWorld.h中第1、2、9行的条件编译指令防止重复包含类。第3行的预编译包含指令包含OopBase.h，这是OopC的核心头文件。第5行的CLASSDEF类函数宏入参为HelloWorld，表示定义一个类HelloWorld。第7行typedef类型定义的含义为，类HelloWorld有一个名为Print的方法，方法的输入和输出参数为空（使用ParamNull示意，定义该结构体的目的也在于此）。至此，HelloWorld类的类定义就完成了。

类实现文件HelloWorld.c中，第1行包含类头文件。第3、4行包含类实现时所需的头文件。第6~9行包含类的数据域结构体，这个结构体的名称必须与类名相同。在OopC中，所有的类都应该有数据域结构体，每个数据域结构体的名称都应该与类名相同。每个类数据域结构体都应包含CHAINDEF，如代码第8行所示。

第14~20行定义类的成员方法Print，您可能会发现，在说明头文件HelloWorld.h时，曾提到Print方法输入输出参数为空，而这里的定义并非如此——Print的输入参数为指针类型。实际上，在类定义时指明Print方法的输入输出参数为空，意在说明pIn参数为NULL，如代码第17行所示。这里需要重点指出的是类的所有成员方法的存储类别都推荐声明为静态内部链接static，入参都必须声明为空指针类型void \*，返回参数都推荐声明为空类型void。第16行获取类的this指针，第17行获取输入输出参数，因为Print方法的输入输出参数为空，所以运行时pIn的值也就是NULL。

第25~58行，也就是实现文件中的剩下部分内容，为类的控制函数，在OopC中，每个类都有4个控制函数：调用函数（INVOKE）、扩展函数（EXTEND）、删除函数（DELETE）、创建函数（CREATE），删除函数和创建函数也就是析构函数和构造函数。调用函数用于帮助客户代码调用实例的方法，在main.c中有其使用示例。扩展函数有两个作用，其一是帮助子类继承该类，其二是子类、父类之间的类型转换，如何使用，在继承示例\_OO\_3\_Inheritance中有所体现。所有类的调用函数和扩展函数都是一致的，删除函数也是大同小异，创建函数除了不同的类的具体业务逻辑之外，大部分代码都相同。第38行，通过类函数宏SWITCH获取当前实例的父指针，第39行调用父类的删除方法进行内存清除，第40行将指针置为NULL。第43~58行中，第45行创建类实例。第48行创建方法环，您大概可以猜测出第51和52行将成员方法Print添加到方法环中，添加的目的在于指明该方法为public，可以被客户代码通过实例调用，第52行的1表示这里添加了一个方法。第53~55行的代码比较固定，第54行的Object说明类HelloWorld继承了它，第55行的null对应的形参，用于释放类相关的额外内存，具体怎么使用在后续章节将会有具体示例予以说明。

客户代码main.c中，第5行创建类实例，在OopC中，有创建则有删除，因此第9行删除类实例。第7行，与一般的函数调用形式相异，OopC中对成员方法的调用通过类的控制函数——调用函数实现，类函数宏INVOKE的输入参数为类名称HelloWorld，表示调用类HelloWorld的调用函数，以实现成员方法的调用，第一个参数为类实例，表示调用将调用该实例的成员方法，第二个参数为"Print"，表示希望调用名"Print"为的成员方法，第三个参数为NULL，表示调用该方法的传入传出参数为空。

好了，现在我们的“Hello, world!”的例子就完成了。注意在编译的时候需要添加头文件OopBase.h，库文件OopBase.lib，运行时需要添加动态链接库文件OopBase.dll。运行成功后，控制台将打印Hello, world!。

# Object

Object类是OopC核心库中唯一的一个类，包含在OopBase.h中，在OopC中，每个类都是Object类的子类，这并不意味着您设计的类会自动继承Object类，相反如果您设计的类没有继承自其它类，则应该显式继承该类。

下面是Object类的定义，

1 typedef struct Object Object;

2

3 Object\* CREATE(Object)();

4 void INVOKE(Object)(Object\* pInst, char\* pFuncName, void\* pParams);

5 void\* EXTEND(Object)(Object\* pInst);

6 void DELETE(Object)(Object\*\* ppInst);

7

8 typedef struct { bool\* pRet; void\* pToCmpr; } Object\_Equal;

9 typedef ParamNull Object\_ToString;

**OopC中，类本质是结构体和函数的一个功能组合。结构体变量作为类实例，而函数因为只能通过结构体变量来调用所以成为类的成员方法。**因此第1行将结构体声明为Object类型，也就声明了Object类。

在“Hello，world!”示例中，类头文件HelloWorld.h中并没有显示声明四个控制函数，这是因为在宏CLASSDEF中已经给出了声明，这里因为没有使用该宏，因此第3~6行显式声明四个控制函数：创建函数CREATE(Object)、调用函数INVOKE(Object)、扩展函数EXTEND(Object)和删除函数DELETE(Object)。第3行可以看出，创建函数没有输入参数，等价于无参构造函数。第4行调用函数包含三个输入参数：类实例、将要调用的方法名称字符串和该方法的输入输出参数，它是一个结构体指针，什么样的结构体指针呢？不急。第5行为扩展函数，入参为类实例，返回一个指针。第6行为删除函数，它的参数为类实例的地址。

我们先看第9行，从“Hello，world!”示例我们可以知道，Object类包含一个ToString方法，该方法的输入输出参数为NULL。从第8行可以看出，Object类还包含一个Equal方法，该方法和ToString不同，它有两个参数，bool指针pRet为返回值（当然，您可以将它作为输入参数，不过这好像不太优雅），表明比较结果，void\*指针pToCmpr则为需要与之比较的参数。我们刚才说到，调用函数的第三个参数为一个结构体指针，现在应该比较清晰了，这个结构体就是对应方法的出入参数结构体。下面用一个例子来说明，

bool bRet = false;

Object \*pToCmpr = CREATE(Object)();

Object \*pObj = CREATE(Object)();

INVOKE(Object)(pObj, "Equal", &(Object\_Equal){&bRet, pToCmpr});

# OOP



## 封装

## 继承

## 多态

# 特性

# 规范

# API汇总

# 原理

# 设计模式实例

# 感谢

# 一起

如果您喜欢OopC，并且想加入进来贡献自己的智慧，请记住，我一直在等待您这样的朋友。如果您在阅读代码后，有什么更好的建议，真诚的希望您能够向我提出来。

# 数据结构



## 方法

* 声明：

typedef struct Method Method;

* 实现：

struct Method

{

Method\* pPrev;

Method\* pNext;

Transit pAddr;

char\* pName;

};

* 说明：

Method：描述类的一个方法；

pPrev：指向前一个方法结构；

pNext：指向后一个方法结构；

Transit：函数指针类型——“typedef void (\*Transit)(void\*); ”，用于存储类成员方法的地址；

pName：用于存储类成员方法的名称。

## 方法环

* 声明：

typedef struct MethodRing MethodRing;

* 实现：

struct MethodRing

{

Method\* pHead;

Method\* pTail;

};

* 说明：

MethodRing：用于存储类的成员方法，存储的时候，使用方法结构的pPrev指针和pNext指针，相互连接，形成一个闭合的**环形结构**，注意，这里并没有指定一定要存储一个类的全部成员方法到该环中；

pHead：环形结构的头元素地址；

pTail：环形结构的尾元素地址；

## 实例

* 声明：

typedef struct Instance Instance;

* 实现：

struct Instance

{

Instance\* pPrev;

Instance\* pNext;

void\* pFields;

char\* pName;

MethodRing\* pMethods;

};

* 说明：

Instance：描述一个类实例；

pPrev：指向前一个类实例结构；

pNext：指向下一个类实例结构；

pFields：存储类的数据域结构体；

pName：存储类实例对应的类型的字符串名称；

pMethods：方法环，用于存储类的成员方法。

## 实例链

* 声明：

typedef struct InstanceChain InstanceChain;

* 实现：

struct InstanceChain

{

Instance\* pHead;

Instance\* pTail;

};

* 说明：

InstanceChain：用于存储有继承关系的一系列实例，各层级的实例从头(pHead)到尾(pTail)顺序排列，形成条**链式结构**；

# API

OopC库实现C语言面向对象的特性时，实例链(InstanceChain)的作用非常关键。而实例链的构造涉及实例结构体(Instance)构造，方法环(MethodRing)的构造和方法(Method)的构造，下面罗列相关的API。

* Method\* GenerateMethod(Transit pAddr, char\* pName);

使用成员方法的地址和名称生成一个成员方法。这里需要说明一点，按道理，不同的类的各个成员方法实现不同的功能，入参和出参不应一致，这里却认为成员方法全部为类型Transit，这个问题后面说明。

* MethodRing\* GenerateMethodRing();

生成一个方法环结构。

* typedef struct MethodUtil

{

MethodRing\* pRing;

struct MethodUtil\* (\*InsertMethod)(struct MethodUtil\*, Method\*);

} MethodUtil;

将方法插入环中时，使用的一个实用结构体。

* MethodUtil\* InsertMethod(MethodUtil\* pUtil, Method\* pMethod);

向环中插入一个方法，这个“环”指代pUtil中的pRing。具体如何使用参考下面的例子：

pMethods =

InsertMethod(&(MethodUtil){pMethods,InsertMethod},GenerateMethod(Input,"Input"))

->InsertMethod(&(MethodUtil){pMethods,InsertMethod},GenerateMethod(Add,"Add"))

->InsertMethod(&(MethodUtil){pMethods,InsertMethod},GenerateMethod(Output,"Output"))

->pRing;

* Instance\* GenerateInstance(void\* pFields, char\* pName, MethodRing\* pMethods);

使用类实例数据域、类名和类成员方法环构造一个实例结构体。

* InstanceChain\* GenerateInstanceChain();

生成一个实例链结构体。

* InstanceChain\* InsertInstance(InstanceChain\* pChain, Instance\* pInstance);

向实例链中插入一个实例。

# Object

OopC中每个类都应包括4个全域的控制函数，比如类名为MyClass，

构造函数：Create\_MyClass() 调用函数：Invoke\_MyClass()

扩展函数：Extend\_MyClass() 析构函数：Delete\_MyClass()

借助帮助宏CREATE、INVOKE、EXTEND和DELETE，控制函数可以改为

CREATE(MyClass)() INVOKE (MyClass)()

EXTEND(MyClass)() DELETE (MyClass)()

下面说明Object类，其定义如下，

typedef struct Object Object;

Object\* CREATE(Object)();

void INVOKE(Object)(Object\* pInst, char\* pFuncName, void\* pParams);

void\* EXTEND(Object)(Object\* pInst);

void DELETE(Object)(Object\*\* ppInst);

typedef struct { bool\* pRet; void\* pToCmpr; } Object\_Equal;

typedef ParamNull Object\_ToString;

第一行“typedef”声明Object为类类型；CREATE宏为无参构造函数；EXTEND用于子类扩展；DELETE用于析构，入参为二级指针，目的在于析构以后，将指针置空；第二个“typedef”的含义为：类Object的方法Equal的出入参数结构体为Object\_Equal；第三个“typedef”的含义为：类Object的方法ToString的出入参数结构体为Object\_ToString，可以看出，该结构体从ParamNull定义引出，而ParamNull用于表示空参数，ToString实际上没有出入参数；类Object并没有给出一般意义上的成员函数接口，而是给出了一个通用的成员函数调用入口INVOKE，从调用函数看入参可以看出，它需要类实例，需要指明所调用函数名称以及参数，您应该能想到，这个参数就是后面的结构体变量。