**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](#_Toc19919408)

[2. 源码 2](#_Toc19919409)

[3. Hello, world! 2](#_Toc19919410)

[4. 原理 4](#_Toc19919411)

[5. API 4](#_Toc19919412)

[6. Object 4](#_Toc19919413)

[7. OOP 4](#_Toc19919414)

[6.1. 继承 4](#_Toc19919421)

[6.2. 封装 4](#_Toc19919422)

[6.3. 多态 4](#_Toc19919423)

[8. 设计模式实例 4](#_Toc19919424)

[9. 感谢….. 5](#_Toc19919425)

[10. 数据结构 5](#_Toc19919426)

[8.1. 方法 5](#_Toc19919429)

[8.2. 方法环 5](#_Toc19919430)

[8.3. 实例 6](#_Toc19919431)

[8.4. 实例链 7](#_Toc19919432)

[11. API 7](#_Toc19919433)

[12. Object 8](#_Toc19919434)

# 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，或者说最优化。

事实上不乏前辈追求或追求过同样的事情。我的另一个目标是实用，而不是完全空洞的优雅。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

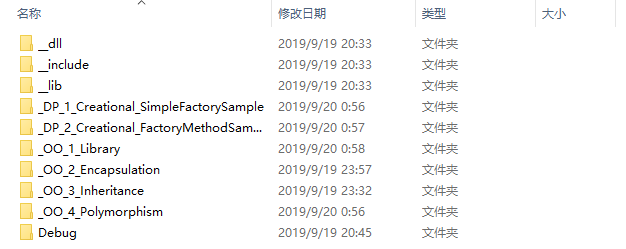
OopC的源代码托管在GitHub上，<https://github.com/GoodmanTao/OopC.git>，编译环境为Visual Studio 2017，采用MIT授权，也就是说，如果我的代码有幸能供您使用，还望您将软件著作权声明的证书附加到所使用的项目中，此外，没有其它的要求了。如果能够通知我一下，您正在使用这些代码，我会非常高兴的。

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 }

//////////////////////////////////////////////待续//////////////////////////////////////////////

# 原理

# API

# Object

# OOP



## 继承

## 封装

## 多态

# 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。