

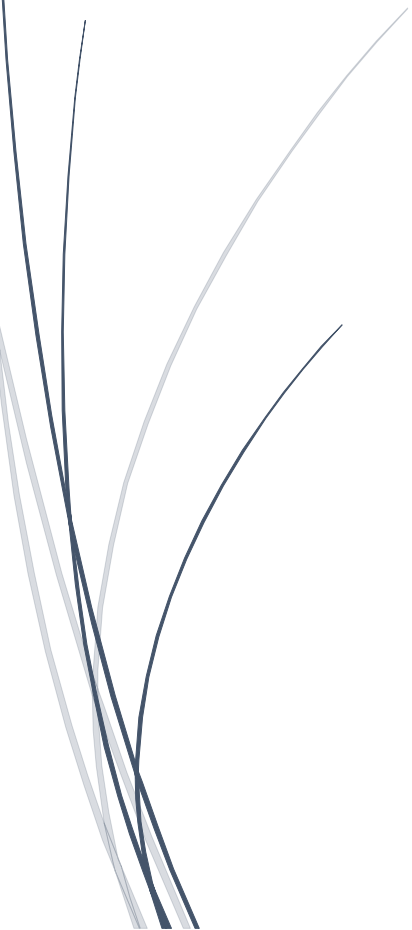


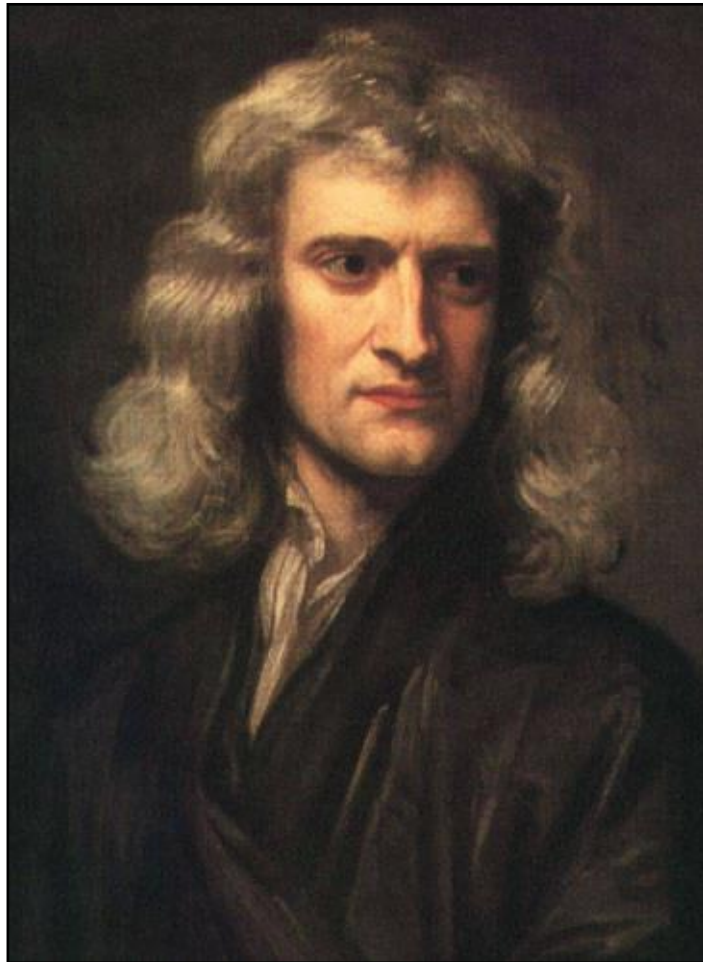
OopC

C 语言面向对象特性的支持库

Goodman Tao

`taoxing2113@foxmail.com`





无比崇敬的 *Isaac Newton* 爵士

目录

1. 起因.....	2
2. 源码.....	2
3. Hello, world!.....	3
4. 原理.....	6
5. API.....	7
6. Object.....	7
7. OOP.....	7
6.1. 封装.....	7
6.2. 继承.....	7
6.3. 多态.....	7
8. 设计模式实例.....	7
9. 感谢.....	7
10. 一起.....	7
11. 数据结构.....	7
8.1. 方法.....	7
8.2. 方法环.....	8
8.3. 实例.....	9
8.4. 实例链.....	9
12. API.....	10
13. Object.....	11

1. 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。

2. 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，

名称	修改日期	类型	大小
.git	2019/9/20 22:32	文件夹	
.vs	2019/9/17 18:42	文件夹	
doc	2019/9/20 23:33	文件夹	
OopC	2019/9/19 20:33	文件夹	
.gitignore	2019/9/17 18:42	文本文档	1 KB
LICENSE	2019/9/17 18:42	文件	2 KB
OopC.sln	2019/9/19 20:33	Microsoft Visual...	7 KB
README.md	2019/9/17 18:42	MD 文件	1 KB

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，

名称	修改日期	类型	大小
__dll	2019/9/19 20:33	文件夹	
__include	2019/9/19 20:33	文件夹	
__lib	2019/9/19 20:33	文件夹	
_DP_1_Creational_SimpleFactorySample	2019/9/20 0:56	文件夹	
_DP_2_Creational_FactoryMethodSam...	2019/9/20 0:57	文件夹	
_OO_1_Library	2019/9/20 0:58	文件夹	
_OO_2_Encapsulation	2019/9/19 23:57	文件夹	
_OO_3_Inheritance	2019/9/19 23:32	文件夹	
_OO_4_Polymorphism	2019/9/20 0:56	文件夹	
Debug	2019/9/19 20:45	文件夹	

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。

3. 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 *pH1lWrld = CREATE(HelloWorld) ();
6
7      INVOKE(HelloWorld) (pH1lWrld, "Print", NULL);
8
9      DELETE(HelloWorld) (&pH1lWrld);
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 的核心文件。第 5 行的 `CLASSDEF` 类函数宏入参为 `HelloWorld`，表示定义一个类 `HelloWorld`，后面将会具体说明这个宏的具体意义。第 7 行的类型定义含义为，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）。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，这个宏含义为实例链定义，它实际扩展为 `InstanceChain *pChain`。

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。这个问题我们后续会讨论，这里想要说明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。调用函数和扩展函数的含义将在以后说明，实际上，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 `SWITCH` 宏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。对于第 53~55 行，您现在只需要知道，通过这几行类 `HelloWorld` 继承了类 `Object` 即可。这几行代码现在没有理解也不要紧，后面我将陆续给出说明。

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!`。

4. 原理

//////////待续//////////

5. API

6. Object

7. OOP

6.1. 封装

6.2. 继承

6.3. 多态

8. 设计模式实例

9. 感谢

10. 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。

11. 数据结构

8.1. 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。

8.2. 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；

8.3. 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。

8.4. 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；

12.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。

## 13.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。