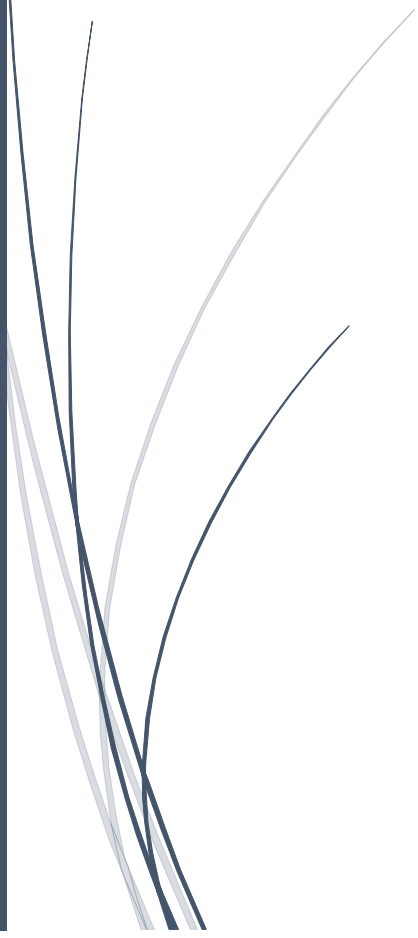
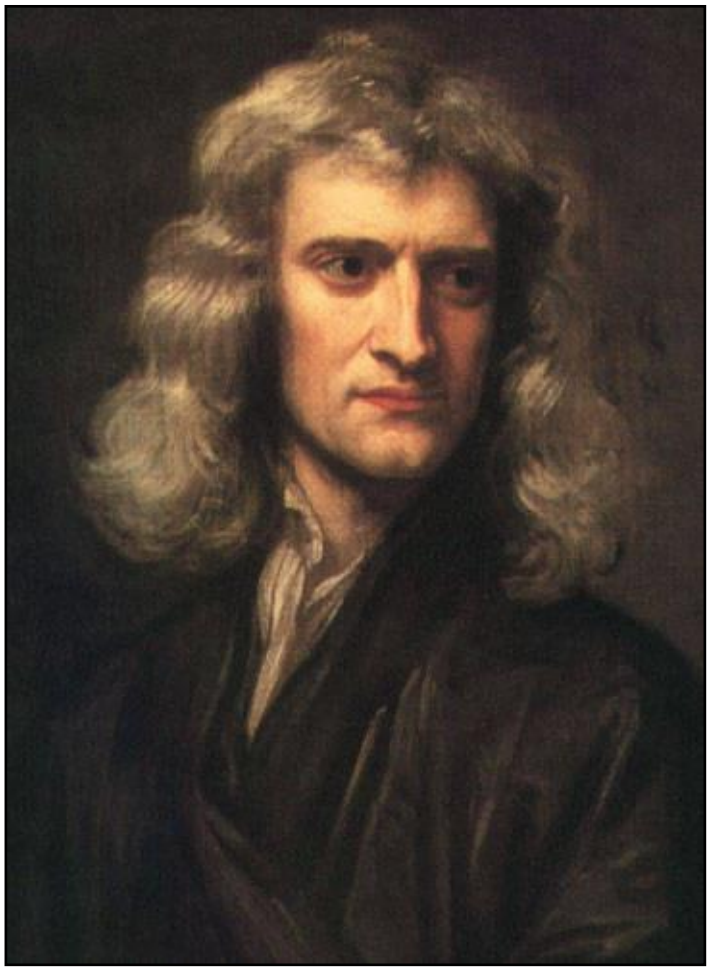
OopC

C 语言面向对象特性的支持

Goodman Tao

taoxing2113@foxmail.com





无比崇敬的 *Isaac Newton* 爵士

目录

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

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

```
#ifndef HELLOWORLD_H__
#define HELLOWORLD_H__
#include <OopBase.h>

CLASSDEF(HelloWorld)

typedef ParamNull HelloWorld_Print;

#endif // !HELLOWORLD_H__
```

- HelloWorld.c

```
#include "HelloWorld.h"

#include <malloc.h>
#include <stdio.h>

struct HelloWorld
{
    CHAINDEF;
```

```
};

////////////////////////////////////////////////////////////////
//

static void Print(void *pParams)
{
    HelloWorld *pThis = ((ParamIn *)pParams)->pInst;
    HelloWorld_Print *pIn = ((ParamIn*)pParams)->pIn;

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

////////////////////////////////////////////////////////////////
//

void INVOKE(HelloWorld)(HelloWorld *pInst, char *pFuncName, void
*pParams)
{
    DOINVOKE(pInst, pFuncName, pParams);
}

void *EXTEND(HelloWorld)(HelloWorld *pInst)
{
    return pInst->pChain;
}

void DELETE(HelloWorld)(HelloWorld **ppInst)
{
    Object *pSuper = SWITCH((*ppInst), HelloWorld, Object);
    DELETE(Object)(&pSuper);
    *ppInst = NULL;
}

HelloWorld *CREATE(HelloWorld)()
{
    HelloWorld *pCreate = malloc(sizeof(HelloWorld));
    if (!pCreate) { return NULL; }

    MethodRing *pMethods = GenerateMethodRing();
    if (!pMethods) { return NULL; }

    pMethods = InsertMethod(pMethods, 1,
        GenerateMethod(Print, "Print"));
    pCreate->pChain =
InsertInstance(EXTEND(Object)(CREATE(Object)()),
GenerateInstance(pCreate, "HelloWorld", NULL, pMethods));

    return pCreate;
```

```
}
```

- main.c

```
#include "HelloWorld.h"

int main(int argc, char **argv)
{
    HelloWorld *pHllWrld = CREATE(HelloWorld)();

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

    DELETE(HelloWorld)(&pHllWrld);

    return 0;
}
```

//待续//

4. 原理

5. API

6. Object

7. OOP

6.1. 继承

6.2. 封装

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。