目录

1.C 语言设计模式（开篇）

关于软件设计方面的书很多，比如《重构》，比如《设计模式》。至于软件开发方式，那就更多了，什么极限编程、精益方法、敏捷方法。随着时间的推移，很多的方法又会被重新提出来。

其实，就我个人看来，不管什么方法都离不开人。一个人写不出二叉树，你怎么让他写？敏捷吗？你写一行，我写一行。还是迭代？写三行，删掉两行，再写三行。项目的成功是偶然的，但是项目的失败却有很多原因，管理混乱、需求混乱、设计低劣、代码质量差、测试不到位等等。就软件企业而言，没有比优秀的文化和出色的企业人才更重要的了。

从软件设计层面来说，一般来说主要包括三个方面：

（1）软件的设计受众，是小孩子、老人、女性，还是专业人士等等；

（2）软件的基本设计原则，以人为本、模块分离、层次清晰、简约至上、适用为先、抽象基本业务等等；

（3）软件编写模式，比如装饰模式、责任链、单件模式等等。

从某种意义上说，设计思想构成了软件的主题。软件原则是我们在开发中的必须遵循的准绳。软件编写模式是开发过程中的重要经验总结。灵活运用设计模式，一方面利于我们编写高质量的代码，另一方面也方便我们对代码进行维护。毕竟对于广大的软件开发者来说，软件的维护时间要比软件编写的时间要多得多。编写过程中，难免要有新的需求，要和别的模块打交道，要对已有的代码进行复用，那么这时候设计模式就派上了用场。我们讨论的主题其实就是设计模式。

讲到设计模式，人们首先想到的语言就是 c#或者是 java，最不济也是 c++，一般来说没有人会考虑到 c 语言。其实，我认为设计模式就是一种基本思想，过度美化或者神化其实没有必要。其实阅读过 linux kernel 的朋友都知道，linux 虽然自身支持很多的文件系统，但是 linux 自身很好地把这些系统的基本操作都抽象出来了，成为了基本的虚拟文件系统。

举个例子来说，现在让你写一个音乐播放器，但是要支持的文件格式很多，什么 ogg，wav，mp3 啊，统统要支持。这时候，你会怎么编写呢？如果用 C++语言，你可能会这么写。

```
class music_file
{
    HANDLE hFile;

public:
    void music_file() {}
    virtual ~music_file() {}
    virtual void read_file() {}
    virtual void play() {}
```

```
        virtual void stop() {}
        virtual void back() {}
        virtual void front() {}
        virtual void up() {}
        virtual void down() {}
    };
```

其实，你想想看，如果用 C 语言能够完成相同的抽象操作，那不是效果一样的吗？

```
    typedef struct _music_file
    {
        HANDLE hFile;
        void (*read_file)(struct _music_file* pMusicFile);
        void (*play)(struct _music_file* pMusicFile);
        void (*stop)(struct _music_file* pMusicFile);
        void (*back)(struct _music_file* pMusicFile);
        void (*front)(struct _music_file* pMusicFile);
        void (*down)(struct _music_file* pMusicFile);
        void (*up)(struct _music_file* pMusicFile);
    }music_file;
```

当然，上面的例子比较简单，但是也能说明一些问题。写这篇文章的目的一是希望和朋友们共同学习模式的相关内容，另一方面也希望朋友们能够活学活用，既不要迷信权威，也不要妄自菲薄。只要付出努力，付出汗水，肯定会有收获的。有些大环境你改变不了，那就从改变自己开始。万丈高楼平地起，一步一个脚印才能真真实实学到东西。如果盲目崇拜，言必 google、微软、apple，那么除了带来几个唾沫星，还能有什么受用呢？无非白费了口舌而已。

希望和大家共勉。

2.C 语言和设计模式（继承、封装、多态）

记得还在我们大学 C++第一门课的时候，老师就告诉我们说，C++是一门面向对象的语言。C++有三个最重要的特点，即继承、封装、多态。等到后来随着编码的增多和工作经验的积累，我也慢慢明白了面向对象的含义。可是，等我工作以后，使用的编程语言更多的是 C 语言，这时候我又想能不能把 C 语言变成面向对象的语言呢？等到后来通过思考和实践，我发现其实 C 语言也是可以面向对象的，也是可以应用设计模式的，关键就在于如何实现面向对象语言的三个重要属性。

2.1 继承性

```
typedef struct _parent
{
    int data_parent;

}Parent;
```

```
typedef struct _Child
{
    struct _parent parent;
    int data_child;

}Child;
```

在设计 C 语言继承性的时候，我们需要做的就是把基础数据放在继承的结构的首位置即可。这样，不管是数据的访问、数据的强转、数据的访问都不会有什么问题。

2.2 封装性

```
struct _Data;

typedef    void (*process)(struct _Data* pData);

typedef struct _Data
{
    int value;
    process pProcess;

}Data;
```

封装性的意义在于，函数和数据是绑在一起的，数据和数据是绑在一起的。这样，我们就可以通过简单的一个结构指针访问到所有的数据，遍历所有的函数。封装性，这是类拥有的属性，当然也是数据结构体拥有的属性。

2.3 多态

3.单件模式

有过面试经验的朋友，或者对设计模式有点熟悉的朋友，都会对单件模式不陌生。对很多面试官而言，单件模式更是他们面试的保留项目。其实，我倒认为，单件模式算不上什么设计模式。最多也就是个技巧。

单件模式要是用 C++写，一般这么写。

```
#include <string.h>
#include <assert.h>

class object
{
public:
    static class object* pObject;

    static object* create_new_object()
    {
        if(NULL != pObject)
         return pObject;

     pObject = new object();
```

```
        assert(NULL != pObject);
        return pObject;
       }

    private:
        object() {}
        ~object() {}
    };

    class object* object::pObject = NULL;
```

单件模式的技巧就在于类的构造函数是一个私有的函数。但是类的构造函数又是必须创建的？怎么办呢？那就只有动用static函数了。我们看到static里面调用了构造函数，就是这么简单。

```
    int main(int argc, char* argv[])
    {
     object* pGlobal = object::create_new_object();
     return 1;
    }
```

4.工厂模式

工厂模式是比较简单，也是比较好用的一种方式。根本上说，工厂模式的目的就根据不同的要求输出不同的产品。比如说吧，有一个生产鞋子的工厂，它能生产皮鞋，也能生产胶鞋。如果用代码设计，应该怎么做呢？

```
    typedef struct _Shoe
    {
        int type;
        void (*print_shoe)(struct _Shoe*);
    }Shoe;
```

就像上面说的，现在有胶鞋，那也有皮鞋，我们该怎么做呢？

```
    void print_leather_shoe(struct _Shoe* pShoe)
    {
        assert(NULL != pShoe);
        printf("This is a leather show!\n");
    }

    void print_rubber_shoe(struct _Shoe* pShoe)
    {
        assert(NULL != pShoe);
        printf("This is a rubber shoe!\n");
    }
```

所以，对于一个工厂来说，创建什么样的鞋子，就看我们输入的参数是什么？至于结果，

那都是一样的。

```
#define LEATHER_TYPE 0x01
#define RUBBER_TYPE    0x02

Shoe* manufacture_new_shoe(int type)
{
    assert(LEATHER_TYPE == type || RUBBER_TYPE == type);

    Shoe* pShoe = (Shoe*)malloc(sizeof(Shoe));
    assert(NULL != pShoe);

    memset(pShoe, 0, sizeof(Shoe));
    if(LEATHER_TYPE == type)
    {
        pShoe->type == LEATHER_TYPE;
        pShoe->print_shoe = print_leather_shoe;
    }
    else
    {
        pShoe->type == RUBBER_TYPE;
        pShoe->print_shoe = print_rubber_shoe;
    }

    return pShoe;
}
```

5.抽象工厂模式

前面我们写过的工厂模式实际上是对产品的抽象。对于不同的用户需求，我们可以给予不同的产品，而且这些产品的接口都是一致的。而抽象工厂呢？顾名思义，就是说我们的工厂是不一定的。怎么理解呢，举个例子。

假设有两个水果店都在卖水果，都卖苹果和葡萄。其中一个水果店买白苹果和白葡萄，另外一个水果店卖红苹果和红葡萄。所以说，对于水果店而言，尽管都在卖水果，但是两个店卖的品种不一样。

既然水果不一样，那我们先定义水果。

```
typedef struct _Apple
{
    void (*print_apple)();
}Apple;

typedef struct _Grape
{
```

```
    void (*print_grape)();
}Grape;
```

上面分别对苹果和葡萄进行了抽象，当然它们的具体函数也是不一样的。

```
void print_white_apple()
{
    printf("white apple!\n");
}

void print_red_apple()
{
    printf("red apple!\n");
}

void print_white_grape()
{
    printf("white grape!\n");
}

void print_red_grape()
{
    printf("red grape!\n");
}
```

完成了水果函数的定义。下面就该定义工厂了，和水果一样，我们也需要对工厂进行抽象处理。

```
typedef struct _FruitShop
{
    Apple* (*sell_apple)();
    Grape * (*sell_grape)();
}FruitShop;
```

所以，对于卖白苹果、白葡萄的水果店就该这样设计了，红苹果、红葡萄的水果店亦是如此。

```
Apple* sell_white_apple()
{
    Apple* pApple = (Apple*) malloc(sizeof(Apple));
    assert(NULL != pApple);

    pApple->print_apple = print_white_apple;
    return pApple;
}

Grape* sell_white_grape()
{
    Grape* pGrape = (Grape*) malloc(sizeof(Grape));
    assert(NULL != pGrape);
```

```
    pGrape->print_grape = print_white_grape;
    return pGrape;
}
```

这样，基本的框架就算搭建完成的，以后创建工厂的时候，

```
FruitShop* create_fruit_shop(int color)
{
    FruitShop* pFruitShop = (FruitShop*) malloc(sizeof(FruitShop));
    assert(NULL != pFruitShop);

    if(WHITE == color)
    {
        pFruitShop->sell_apple = sell_white_apple;
        pFruitShop->sell_grape = sell_white_grape;
    }
    else
    {
        pFruitShop->sell_apple = sell_red_apple;
        pFruitShop->sell_grape = sell_red_grape;
    }

    return pFruitShop;
}
```

6.访问者模式

不知不觉当中，我们就到了最后一种设计模式，即访问者模式。访问者模式，听上去复杂一些。但是，这种模式用简单的一句话说，就是不同的人对不同的事物有不同的感觉。比如说吧，豆腐可以做成麻辣豆腐，也可以做成臭豆腐。可是，不同的地方的人未必都喜欢这两种豆腐。四川的朋友可能更喜欢辣豆腐，江浙的人就可能对臭豆腐更喜欢一些。那么，这种情况应该怎么用设计模式表达呢？

```
typedef struct _Tofu
{
    int type;
    void (*eat)      (struct _Visitor* pVisitor, struct _Tofu* pTofu);
}Tofu;

typedef struct _Visitor
{
    int region;
    void (*process)(struct _Tofu* pTofu, struct _Visitor* pVisitor);
}Visitor;
```

7.状态模式

状态模式是协议交互中使用得比较多的模式。比如说，在不同的协议中，都会存在启动、保持、中止等基本状态。那么怎么灵活地转变这些状态就是我们需要考虑的事情。假设现在有一个 state，

说明一下，这里定义了两个变量，分别 process 函数和 change_state 函数。其中 proces 函数就是普通的数据操作，

```
void normal_process()
{
    printf("normal process!\n");
}
```

change_state 函数本质上就是确定下一个状态是什么。

所以，在 context 中，应该有一个 state 变量，还应该有一个 state 变换函数。

```
typedef struct _Context
{
    State* pState;
    void (*change)(struct _Context* pContext);

}Context;

void context_change(struct _Context* pContext)
{
    State* pPre;
    assert(NULL != pContext);

    pPre = pContext->pState;
    pContext->pState = pPre->changeState();
    free(pPre);
    return;
}
```

8.命令模式

命令模式的目的主要是为了把命令者和执行者分开。老规矩，举个范例吧。假设李老板是一家公司的头儿，他现在让他的秘书王小姐去送一封信。王小姐当然不会自己亲自把信送到目的地，她会把信交给邮局来完成整个投递的全过程。现在，我们就对投递者、命令、发令者分别作出定义。

首先定义 post 的相关数据。

```
typedef struct _Post
```

```
{
    void (*do)(struct _Post* pPost);
}Post;
```

Post 完成了实际的投递工作，那么命令呢？

```
void post_exe(struct _Command* pCommand)
{
    assert(NULL != pCommand);

    (Post*)(pCommand->pData)->do((Post*)(pCommand->pData));
    return;
}
```

我们看到了 Post、Command 的操作，那么剩下的就是 boss 的定义了。

```
typedef struct _Boss
{
    Command* pCommand;
    void (*call)(struct _Boss* pBoss);
}Boss;

void boss_call(struct _Boss* pBoss)
{
    assert(NULL != pBoss);

    pBoss->pCommand->exe(pBoss->pCommand);
    return;
}
```

9.解释器模式

命令模式的目的主要是为了把命令者和执行者分开。老规矩，举个范例吧。假设李老板是一家公司的头儿，他现在让他的秘书王小姐去送一封信。王小姐当然不会自己亲自把信送到目的地，她会把信交给邮局来完成整个投递的全过程。现在，我们就对投递者、命令、发令者分别作出定义。

首先定义 post 的相关数据。

```
typedef struct _Post
{
    void (*do)(struct _Post* pPost);
}Post;
```

Post 完成了实际的投递工作，那么命令呢？

```
void post_exe(struct _Command* pCommand)
{
    assert(NULL != pCommand);
```

```
    (Post*)(pCommand->pData)->do((Post*)(pCommand->pData));
    return;
}
```

我们看到了 Post、Command 的操作，那么剩下的就是 boss 的定义了。

```
typedef struct _Boss
{
    Command* pCommand;
    void (*call)(struct _Boss* pBoss);
}Boss;

void boss_call(struct _Boss* pBoss)
{
    assert(NULL != pBoss);

    pBoss->pCommand->exe(pBoss->pCommand);
    return;
}
```

10.备忘录模式

备忘录模式的起源来自于撤销的基本操作。有过 word 软件操作经验的朋友，应该基本上都使用过撤销的功能。举个例子，假设你不小心删除了好几个段落的文字，这时候你应该怎么办呢？其实要做的很简单，单击一些【撤销】就可以全部搞定了。撤销按钮给我们提供了一次反悔的机会。

既然是撤销，那么我们在进行某种动作的时候，就应该创建一个相应的撤销操作？这个撤销操作的相关定义可以是这样的。

```
typedef struct _Action
{
    int type;
    struct _Action* next;

    void* pData;
    void (*process)(void* pData);

}Action;
```

数据结构中定义了两个部分：撤销的数据、恢复的操作。那么这个撤销函数应该有一个创建的函数，还有一个恢复的函数。所以，作为撤销动作的管理者应该包括，

既然数据在创建和修改的过程中都会有相应的恢复操作，那么要是真正恢复原来的数据也就变得非常简单了。

```
void restore(struct _Organizer* pOrganizer)
{
```

```
    Action* pHead;
    assert(NULL != pOrganizer);

    pHead = pOrganizer->pActionHead;
    pHead->process(pHead->pData);
    pOrganizer->pActionHead = pHead->next;
    pOrganizer->number --;
    free(pHead);
    return;
}
```

11.观察者模式

观察者模式可能是我们在软件开发中使用得比较多的一种设计模式。为什么这么说？大家可以听我一一到来。我们知道，在 windows 的软件中，所有的界都是由窗口构成的。对话框是窗口，菜单是窗口，工具栏也是窗口。那么这些窗口，在很多情况下要对一些共有的信息进行处理。比如说，窗口的放大，窗口的减小等等。面对这一情况，观察者模式就是不错的一个选择。

首先，我们可以对这些共有的 object 进行提炼。

既然 Observer 在创建的时候就把自己绑定在某一个具体的 Object 上面，那么 Object 发生改变的时候，统一更新操作就是一件很容易的事情了。

```
void notify(struct _Object* pObject)
{
    Obserer* pObserver;
    int index;

    assert(NULL != pObject);
    for(index = 0; index < pObject->number; index++)
    {
        pObserver = pObjecet->pObserverList[index];
        pObserver->update(pObserver);
    }
}
```

12.桥接模式

在以往的软件开发过程中，我们总是强调模块之间要低耦合，模块本身要高内聚。那么，可以通过哪些设计模式来实现呢？桥接模式就是不错的一个选择。我们知道，在现实的软件开发过程当中，用户的要求是多种多样的。比如说，有这么一个饺子店吧。假设饺子店原来

只卖肉馅的饺子，可是后来一些吃素的顾客说能不能做一些素的饺子。听到这些要求的老板自然不敢怠慢，所以也开始卖素饺子。之后，又有顾客提出，现在的肉馅饺子只有猪肉的，能不能做点牛肉、羊肉馅的饺子？一些只吃素的顾客也有意见了，他们建议能不能增加一些素馅饺子的品种，什么白菜馅的、韭菜馅的，都可以做一点。由此看来，顾客的要求是一层一层递增的。关键是我们如何把顾客的要求和我们的实现的接口进行有效地分离呢？

上面只是对饺子进行归类。第一类是对肉馅饺子的归类，第二类是对素馅饺子的归类，这些地方都没有什么特别之处。那么，关键是我们怎么把它和顾客的要求联系在一起呢？

```
typedef struct _DumplingReuqest
{
    int type;
    void* pDumpling;
}DumplingRequest;
```

这里定义了一个饺子买卖的接口。它的特别支持就在于两个地方，第一是我们定义了饺子的类型 type，这个 type 是可以随便扩充的；第二就是这里的 pDumpling 是一个 void*指针，只有把它和具体的 dumpling 绑定才会衍生出具体的含义。

```
void buy_dumpling(DumplingReuqest* pDumplingRequest)
{
    assert(NULL != pDumplingRequest);

    if(MEAT_TYPE == pDumplingRequest->type)
        return (MeatDumpling*)(pDumplingRequest->pDumpling)->make();
    else
        return (NormalDumpling*)(pDumplingRequest->pDumpling)->make();
}
```

13.建造者模式

如果说前面的工厂模式是对接口进行抽象化处理，那么建造者模式更像是对流程本身的一种抽象化处理。这话怎么理解呢？大家可以听我慢慢到来。以前买电脑的时候，大家都喜欢自己组装机器。一方面可以满足自己的个性化需求，另外一方面也可以在价格上得到很多实惠。但是电脑是由很多部分组成的，每个厂家都只负责其中的一部分，而且相同的组件也有很多的品牌可以从中选择。这对于我们消费者来说当然非常有利，那么应该怎么设计呢？

```
typedef struct _AssemblePersonalComputer
{
    void (*assemble_cpu)();
    void (*assemble_memory)();
    void (*assemble_harddisk)();

}AssemblePersonalComputer;
```

对于一个希望配置 intel cpu，samsung 内存、日立硬盘的朋友。他可以这么设计，

14.中介者模式

中介者模式，听上去有一点陌生。但是，只要我给朋友们打个比方就明白了。早先自由恋爱没有现在那么普遍的时候，男女之间的相识还是需要通过媒婆之间才能相互认识。男孩对女方有什么要求，可以通过媒婆向女方提出来；当然，女方有什么要求也可以通过媒婆向男方提出来。所以，中介者模式在我看来，就是媒婆模式。

```
typedef struct _Mediator
{
    People* man;
    People* woman;
}Mediator;
```

上面的数据结构是给媒婆的，那么当然还有一个数据结构是给男方、女方的。

```
typedef struct _People
{
    Mediator* pMediator;

    void (*request)(struct _People* pPeople);
    void (*process)(struct _Peoplle* pPeople);
}People;
```

所以，这里我们看到的如果是男方的要求，那么这个要求应该女方去处理啊，怎么处理呢？

```
void man_request(struct _People* pPeople)
{
    assert(NULL != pPeople);

    pPeople->pMediator->woman->process(pPeople->pMediator->woman);
}
```

上面做的是男方向女方提出的要求，所以女方也可以向男方提要求了。毕竟男女平等嘛。

```
void woman_request(struct _People* pPeople)
{
    assert(NULL != pPeople);

    pPeople->pMediator->man->process(pPeople->pMediator->man);
}
```

15.策略模式

策略模式就是用统一的方法接口分别对不同类型的数据进行访问。比如说，现在我们想用 pc 看一部电影，此时应该怎么做呢？看电影嘛，当然需要各种播放电影的方法。rmvb 要 rmvb 格式的方法，avi 要 avi 的方法,mpeg 要 mpeg 的方法。可是事实上，我们完全可以不去管是什么文件格式。因为播放器对所有的操作进行了抽象，不同的文件会自动调用相应的访问方法。

```
typedef struct _MoviePlay
{
    struct _CommMoviePlay* pCommMoviePlay;

}MoviePlay;
```

这个时候呢，对于用户来说，统一的文件接口就是 MoviePlay。接下来的一个工作，就是编写一个统一的访问接口。

```
void play_movie_file(struct MoviePlay* pMoviePlay)
{
    CommMoviePlay* pCommMoviePlay;
    assert(NULL != pMoviePlay);

    pCommMoviePlay = pMoviePlay->pCommMoviePlay;
    pCommMoviePlay->play(pCommMoviePlay->hFile);
}
```

最后的工作就是对不同的 hFile 进行 play 的实际操作，写简单一点就是，

```
void play_avi_file(HANDLE hFile)
{
    printf("play avi file!\n");
}

void play_rmvb_file(HANDLE hFile)
{
    printf("play rmvb file!\n");
}

void play_mpeg_file(HANDLE hFile)
{
    printf("play mpeg file!\n");
}
```

16.适配器模式

现在的生活当中，我们离不开各种电子工具。什么笔记本电脑、手机、mp4 啊，都离不开充电。既然是充电，那么就需要用到充电器。其实从根本上来说，充电器就是一个个普通的适配器。什么叫适配器呢，就是把 220v、50hz 的交流电压编程 5~12v 的直流电压。充电器就干了这么一件事情。

那么，这样的一个充电适配器，我们应该怎么用 c++描述呢？

```
class voltage_12v
{
public:
	voltage_12v() {}
	virtual ~voltage_12v() {}
	virtual void request() {}
};


class adapter: public voltage_12v
{
	v220_to_v12* pAdaptee;

public:
	adapter() {}
	~adapter() {}

	void request()
	{
		pAdaptee->voltage_transform_process();
	}
};
```

通过上面的代码，我们其实可以这样理解。类 voltage_12v 表示我们的最终目的就是为了获得一个 12v 的直流电压。当然获得 12v 可以有很多的方法，利用适配器转换仅仅是其中的一个方法。adapter 表示适配器，它自己不能实现 220v 到 12v 的转换工作，所以需要调用类 v220_to_v12 的转换函数。所以，我们利用 adapter 获得 12v 的过程，其实就是调用 v220_to_v12 函数的过程。

不过，既然我们的主题是用 c 语言来编写适配器模式，那么我们就要实现最初的目标。这其实也不难，关键一步就是定义一个 Adapter 的数据结构。然后把所有的 Adapter 工作都由 Adaptee 来做，就是这么简单。不知我说明白了没有？

```
typdef struct _Adaptee
{
	void (*real_process)(struct _Adaptee* pAdaptee);
}Adaptee;
```

17.装饰模式

装饰模式是比较好玩，也比较有意义。其实就我个人看来，它和责任链还是蛮像的。只不过一个是比较判断，一个是迭代处理。装饰模式就是那种迭代处理的模式，关键在哪呢？我们可以看看数据结构。

```
typedef struct _Object
{
    struct _Object* prev;

    void (*decorate)(struct _Object* pObject);
}Object;
```

装饰模式最经典的地方就是把 pObject 这个值放在了数据结构里面。当然，装饰模式的奥妙还不仅仅在这个地方，还有一个地方就是迭代处理。我们可以自己随便写一个 decorate 函数试试看，

```
void decorate(struct _Object* pObeject)
{
    assert(NULL != pObject);

    if(NULL != pObject->prev)
        pObject->prev->decorate(pObject->prev);

    printf("normal decorate!\n");
}
```

所以，装饰模式的最重要的两个方面就体现在：prev 参数和 decorate 迭代处理。

18.亨元模式

享元模式看上去有点玄乎，但是其实也没有那么复杂。我们还是用示例说话。比如说，大家在使用电脑的使用应该少不了使用 WORD 软件。使用 WORD 呢，那就少不了设置模板。什么模板呢，比如说标题的模板，正文的模板等等。这些模板呢，又包括很多的内容。哪些方面呢，比如说字体、标号、字距、行距、大小等等。

上面的 Font 表示了各种 Font 的模板形式。所以，下面的方法就是定制一个 FontFactory 的结构。

```
typedef struct _FontFactory
{
    Font** ppFont;
    int number;
```

```
    int size;

    Font* GetFont(struct _FontFactory* pFontFactory, int type, int sequence, int gap, int lineDistance);
}FontFactory;
```

这里的 GetFont 即使对当前的 Font 进行判断，如果 Font 存在，那么返回；否则创建一个新的 Font 模式。

```
Font* GetFont(struct _FontFactory* pFontFactory, int type, int sequence, int gap, int lineDistance)
{
    int index;
    Font* pFont;
    Font* ppFont;

    if(NULL == pFontFactory)
        return NULL;

    for(index = 0; index < pFontFactory->number; index++)
    {
        if(type != pFontFactory->ppFont[index]->type)
            continue;

        if(sequence != pFontFactory->ppFont[index]->sequence)
            continue;

        if(gap != pFontFactory->ppFont[index]->gap)
            continue;

        if(lineDistance != pFontFactory->ppFont[index]->lineDistance)
            continue;

        return pFontFactory->ppFont[index];
    }

    pFont = (Font*)malloc(sizeof(Font));
```

19.代理模式

代理模式是一种比较有意思的设计模式。它的基本思路也不复杂。举个例子来说，以前在学校上网的时候，并不是每一台 pc 都有上网的权限的。比如说，现在有 pc1、pc2、pc3，但是只有 pc1 有上网权限，但是 pc2、pc3 也想上网，此时应该怎么办呢？

此时，我们需要做的就是在 pc1 上开启代理软件，同时把 pc2、pc3 的 IE 代理指向 pc1 即可。这个时候，如果 pc2 或者 pc3 想上网，那么报文会先指向 pc1，然后 pc1 把 Internet 传回的报文再发给 pc2 或者 pc3。这样一个代理的过程就完成了整个的上网过程。

在说明完整的过程之后，我们可以考虑一下软件应该怎么编写呢？

```
void ftp_request()
{
	printf("request from ftp!\n");
}

void http_request()
{
	printf("request from http!\n");
}

void smtp_request()
{
	printf("request from smtp!\n");
}
```

这个时候，代理的操作应该怎么写呢？怎么处理来自各个协议的请求呢？

```
typedef struct _Proxy
{
	PC_Client* pClient;
}Proxy;

void process(Proxy* pProxy)
{
	assert(NULL != pProxy);

	pProxy->pClient->request();
}
```

20.外观模式

外观模式是比较简单的模式。它的目的也是为了简单。什么意思呢？举个例子吧。以前，我们逛街的时候吃要到小吃一条街，购物要到购物一条街，看书、看电影要到文化一条街。那么有没有这样的地方，既可以吃喝玩乐，同时相互又靠得比较近呢。其实，这就是悠闲广场，遍布全国的万达广场就是干了这么一件事。

首先，我们原来是怎么做的。

```
typedef struct _ShopStreet
{
```

```
	void (*buy)();
}ShopStreet;

void buy()
{
	printf("buy here!\n");
}

typedef struct _BookStreet
{
	void (*read)();
}BookStreet;

void read()
{
	printf("read here");
}
```

下面，我们就要在一个 plaza 里面完成所有的项目，怎么办呢？

```
typedef struct _Plaza
{
	FoodStreet* pFoodStreet;
	ShopStreet* pShopStreet;
	BookStreet* pBookStreet;

	void (*play)(struct _Plaza* pPlaza);
}Plaza;

void play(struct _Plaza* pPlaza)
{
	assert(NULL != pPlaza);

	pPlaza->pFoodStreet->eat();
	pPlaza->pShopStreet->buy();
	pPlaza->pBookStreet->read();
}
```

21.迭代器模式

使用过 C++的朋友大概对迭代器模式都不会太陌生。这主要是因为我们在编写代码的时候离不开迭代器，队列有迭代器，向量也有迭代器。那么，为什么要迭代器呢？这主要是为了提炼一种通用的数据访问方法。

比如说，现在有一个数据的容器，

我们看到，容器有 get_first，迭代器也有 get_first，这中间有什么区别？

```
int vector_get_first(struct _Container* pContainer)
{
	assert(NULL != pContainer);

	return pContainer->pData[0];
}

int vector_get_last(struct _Container* pContainer)
{
	assert(NULL != pContainer);

	return pContainer->pData[pContainer->size -1];
}

int vector_interator_get_first(struct _Interator* pInterator)
{
	Container* pContainer;
	assert(NULL != pInterator && NULL != pInterator->pVector);

	pContainer = (struct _Container*) (pInterator->pVector);
	return pContainer ->get_first(pContainer);
}

int vector_interator_get_last(struct _Interator* pInterator)
{
	Container* pContainer;
	assert(NULL != pInterator && NULL != pInterator->pVector);

	pContainer = (struct _Container*) (pInterator->pVector);
	return pContainer ->get_last(pContainer);
}
```

看到上面的代码之后，我们发现迭代器的操作实际上也是对容器的操作而已。

22.责任链模式

责任链模式是很实用的一种实际方法。举个例子来说，我们平常在公司里面难免不了报销流程。但是，我们知道公司里面每一级的领导的报批额度是不一样的。比如说，科长的额度是 1000 元，部长是 10000 元，总经理是 10 万元。

那么这个时候，我们应该怎么设计呢？其实可以这么理解。比如说，有人来找领导报销费用了，那么领导可以自己先看看自己能不能报。如果费用可以顺利报下来当然最好，可是万一报不下来呢？那就只能请示领导的领导了。

```
typedef struct _Leader
{
    struct _Leader* next;
    int account;

    int (*request)(strcut _Leader* pLeader, int num);
}Leader;

void set_next_leader(const struct _Leader* pLeader, struct _Leader* next)
{
    assert(NULL != pLeader && NULL != next);

    pLeader->next = next;
    return;
}
```

此时，如果有一个员工过来报销费用，那么应该怎么做呢？假设此时的 Leader 是经理，报销额度是 10 万元。所以此时，我们可以看看报销的费用是不是小于 10 万元？少于这个数就 OK，反之就得上报自己的领导了。

```
int request_for_manager(struct _Leader* pLeader, int num)
{
    assert(NULL != pLeader && 0 != num);

    if(num < 100000)
        return 1;
    else if(pLeader->next)
        return pLeader->next->request(pLeader->next, num);
    else
        return 0;
}
```

23．模版模式

模板对于学习 C++的同学，其实并不陌生。函数有模板函数，类也有模板类。那么这个模板模式是个什么情况？我们可以思考一下，模板的本质是什么。比如说，现在我们需要编写一个简单的比较模板函数。

```
template <typename type>
int compare (type a, type b)
{
```

```
    return a > b ? 1 : 0;
}
```

模板函数提示我们，只要比较的逻辑是确定的，那么不管是什么数据类型，都会得到一个相应的结果。固然，这个比较的流程比较简单，即使没有采用模板函数也没有关系。但是，要是需要拆分的步骤很多，那么又该怎么办呢？如果相通了这个问题，那么也就明白了什么是 template 模式。

比方说，现在我们需要设计一个流程。这个流程有很多小的步骤完成。然而，其中每一个步骤的方法是多种多样的，我们可以很多选择。但是，所有步骤构成的逻辑是唯一的，那么我们该怎么办呢？其实也简单。那就是在基类中除了流程函数外，其他的步骤函数全部设置为 virtual 函数即可。

basic 的类说明了基本的流程 process 是唯一的，所以我们要做的就是对 step1 和 step2 进行改写。

```
class data_A : public basic
{
public:
    data_A() {}
    ~data_A() {}
    void step1()
    {
        printf("step 1 in data_A!\n");
    }

    void step2()
    {
        printf("step 2 in data_A!\n");
    }
};
```

所以，按照我个人的理解，这里的 template 主要是一种流程上的统一，细节实现上的分离。明白了这个思想，那么用 C 语言来描述 template 模式就不是什么难事了。

```
typedef struct _Basic
{
    void* pData;
    void (*step1) (struct _Basic* pBasic);
    void (*step2) (struct _Basic* pBasic);
    void (*process) (struct _Basic* pBasic);
}Basic;
```

因为在 C++中 process 函数是直接继承的，C 语言下面没有这个机制。所以，对于每一个 process 来说，process 函数都是唯一的，但是我们每一次操作的时候还是要去复制一遍函数指针。而 step1 和 step2 是不同的，所以各种方法可以用来灵活修改自己的处理逻辑，没有问题。

```
void process(struct _Basic* pBasic)
{
	pBasic->step1(pBasic);
	pBasic->step2(pBasic);
}
```

24.组合模式

组合模式听说去很玄乎，其实也并不复杂。为什么?大家可以先想一下数据结构里面的二叉树是怎么回事。为什么就是这么一个简单的二叉树节点既可能是叶节点，也可能是父节点?

那什么时候是叶子节点，其实就是 left、right 为 NULL 的时候。那么如果它们不是 NULL 呢，那么很明显此时它们已经是父节点了。那么，我们的这个组合模式是怎么一个情况呢？

```
typedef struct _Object
{
	struct _Object** ppObject;
	int number;
	void (*operate)(struct _Object* pObject);

}Object;
```

就是这么一个简单的数据结构，是怎么实现子节点和父节点的差别呢。比如说，现在我们需要对一个父节点的 operate 进行操作，此时的 operate 函数应该怎么操作呢？

```
void operate_of_parent(struct _Object* pObject)
{
	int index;
	assert(NULL != pObject);
	assert(NULL != pObject->ppObject && 0 != pObject->number);

	for(index = 0; index < pObject->number; index ++)
	{
		pObject->ppObject[index]->operate(pObject->ppObject[index]);
	}
}
```

当然，有了 parent 的 operate，也有 child 的 operate。至于是什么操作，那就看自己是怎么操作的了。

```
void operate_of_child(struct _Object* pObject)
{
	assert(NULL != pObject);
```

```
        printf("child node!\n");
    }
```

父节点也好，子节点也罢，一切的一切都是最后的应用。其实，用户的调用也非常简单，就这么一个简单的函数。

```
    void process(struct Object* pObject)
    {
        assert(NULL != pObject);
        pObject->operate(pObject);
    }
```

25.原型模式

原型模式本质上说就是对当前数据进行复制。就像变戏法一样，一个鸽子变成了两个鸽子，两个鸽子变成了三个鸽子，就这么一直变下去。在变的过程中，我们不需要考虑具体的数据类型。为什么呢？因为不同的数据有自己的复制类型，而且每个复制函数都是虚函数。

用 C++怎么编写呢，那就是先写一个基类，再编写一个子类。就是这么简单。

那怎么使用呢？其实只要一个通用的调用接口就可以了。

```
    class data* clone(class data* pData)
    {
        return pData->copy();
    }
```

就这么简单的一个技巧，对 C 来说，当然也不是什么难事。

```
    typedef struct _DATA
    {
        struct _DATA* (*copy) (struct _DATA* pData);
    }DATA;
```

假设也有这么一个类型 data_A,

```
    DATA data_A = {data_copy_A};
```

既然上面用到了这个函数，所以我们也要定义啊。

```
    struct _DATA* data_copy_A(struct _DATA* pData)
    {
        DATA* pResult = (DATA*)malloc(sizeof(DATA));
        assert(NULL != pResult);
        memmove(pResult, pData, sizeof(DATA));
        return pResult;
    };
```

使用上呢，当然也不含糊。

```
struct _DATA* clone(struct _DATA* pData)
{
	return pData->copy(pData);
};
```