目录

C/C++ 设计模式

对于一个开发者而言，能够胜任系统中任意一个模块的开发是其核心价值的体现。

对于一个架构师而言，掌握各种语言的优势并可以运用到系统中，由此简化系统的开发，是其架构生涯的第一步。

对于一个开发团队而言，能在短期内开发出用户满意的软件系统是起核心竞争力的体现。

每一个程序员都不能固步自封，要多接触新的行业，新的技术领域，突破自我。

设计模式系列

- Java设计模式
- Java和Android设计模式
- C/C++设计模式

目录

关注我

- Email：815712739@qq.com
- CSDN博客：Allen Iverson
- 新浪微博：AndroidDeveloper
- GitHub：JackChan1999
- GitBook：alleniverson
- 个人博客：JackChan

第1章 面向对象编程

C语言面向对象编程

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。

C++的结构体可以包含函数，但是C语言的结构体不可以包含函数，只有成员变量，可以用函数指针映射一个函数，实现结构体的封装。

继承

```
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。

封装

```
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。

多态

```
typedef struct _Play
{
    void* pData;
    void (*start_play)(struct _Play* pPlay);
}Play;
```

多态，就是说用同一的接口代码处理不同的数据。比如说，这里的Play结构就是一个通用的数据结构，我们也不清楚pData是什么数据，start_play是什么处理函数？但是，我们处理的时候只要调用pPlay->start_play(pPlay)就可以了。剩下来的事情我们不需要管，因为不同的接口会有不同的函数去处理，我们只要学会调用就可以了。

C++中的面向对象编程

```
class cmd
{
public:
    char *p;

public:
    void run()
    {
        system(p);
    }

    void print()
    {
        std::cout << p << std::endl;
    }
};

class newcmd :public cmd
{
public:
    int getlength()
    {
        return strlen(this->p);
    }
};

void main()
{
    newcmd cmd1;
    cmd1.p = "calc";
    cmd1.run();
    cmd1.print();
    std::cout << cmd1.getlength() << std::endl;

    system("pause");
}
```

C语言中的面向对象编程

```c
#include<stdio.h>
#include<stdlib.h>

struct cmd
{
    char *p;
    void (*prun)(struct cmd *pcmd); // 用函数指针表示一个函数
    void (*pprint)(struct cmd *pcmd);
};

typedef  struct cmd  CMD;

void run(CMD *pcmd )
{
    system(pcmd->p);
}

void print(CMD *pcmd)
{
    printf("%s", pcmd->p);
}

struct newcmd
{
    struct cmd cmd1;  // 实现继承
    int(*plength)(struct newcmd *pnewcmd); // 增加新的功能
};

int getlength(struct newcmd *pnewcmd)
{
    return strlen(pnewcmd->cmd1.p);//返回长度
}

void main1()
{
    CMD cmd1 = { "notepad", run, print };
    cmd1.pprint(&cmd1);
    cmd1.prun(&cmd1);
```

```
    system("pause");
}

void main()
{
    struct newcmd newcmd1;
    newcmd1.cmd1.p = "notepad";
    newcmd1.cmd1.pprint = print;
    newcmd1.cmd1.prun = run;
    newcmd1.plength = getlength;//初始化

    newcmd1.cmd1.pprint(&newcmd1.cmd1);
    newcmd1.cmd1.prun(&newcmd1.cmd1);
    printf("%d",newcmd1.plength(&newcmd1));
    system("pause");
}
```

C++中的多态

```
#include<iostream>
#include<stdlib.h>

class Person
{
public:
    virtual void gettooth()
    {
        printf("牙齿");
    }
};

class Woman :public Person
{
public:
    void gettooth()
    {
        printf("女人的白牙\n");
        system("notepad");
    }
```

```
};

class Man :public Person
{
public:
    void gettooth()
    {
        printf("男人的虎牙\n");
        system("calc");
    }
};

int main()
{
    Woman  woman;
    woman.gettooth();
    woman.Person::gettooth();

    Man   man;
    man.gettooth();
    man.Person::gettooth();

    return 0;
}
```

C语言中的多态

```
#include<stdio.h>
#include<stdlib.h>

struct Person
{
    void (*fun)(struct Person *p);
};

struct Man
{
    struct Person person;
    void (*fun)(struct Man *pm);
```

```
};

void work(struct Man *pm)
{
    printf("工作\n");
}

struct Woman
{
    struct Person person;
    void(*fun)(struct Woman *pw);
};

void shopping(struct Woman *pw)
{
    printf("购物\n");
}

void eat(struct Person *person)
{
    printf("吃饭\n");
}

int main()
{
    struct Man  man;
    man.person.fun = eat;
    man.person.fun(&man.person);

    man.fun = work;
    man.fun(&man);

    struct Woman   woman;
    woman.person.fun = eat;
    woman.person.fun(&woman.person);
    woman.fun = shopping;
    woman.fun(&woman);

    return 0;
}
```

C语言面向对象编程（一）：封装与继承

最近在用 C 做项目，之前用惯了 C++ ，转回头来用C 还真有点不适应。 C++ 语言中自带面向对象支持，如封装、继承、多态等面向对象的基本特征。 C 原本是面向过程的语言，自身没有内建这些特性，但我们还是可以利用 C 语言本身已有的特性来实现面向对象的一些基本特征。接下来我们就一一来细说封装、继承、多态、纯虚类等面向对象特性在 C 语言中如何实现，并且给出实例。

这篇文章中我们先说封装和继承。

先来看封装。

所谓封装，通俗地说，就是一个姑娘化了妆，只给你看她想让你看的那一面，至于里面是否刮了骨、垫了东西，不给你看。说到封装就得说隐藏，这是对兄弟概念；其实我理解隐藏是更深的封装，完全不给你看见，而封装可能是犹抱琵琶半遮面。封装在 C++ 语言中有 protected 、 private 关键字在语言层面上支持，而 C 语言中没有这些。 C 有结构体（ struct ），其实可以实现封装和隐藏。

在 QT 中，为了更好的隐藏一个类的具体实现，一般是一个公开头文件、一个私有头文件，私有头文件中定义实现的内部细节，公开头文件中定义开放给客户程序员的接口和公共数据。看看 QObject （qobject.h ），对应有一个 QObjectPrivate （qobject_p.h ），其他的也类似。而代码框架如下：

```
QObject{
public:
    xxx
    xxx
private:
    QObjectPrivate * priv;
};
```

我们在 C 语言中完全可以用同样的方法来实现封装和隐藏，只不过是放在结构体中而已。代码框架如下：

```
struct st_abc_private;
struct st_abc {
    int a;
    xxx;
    void (*xyz_func)(struct st_abc*);

    struct st_abc_private * priv;
};
```

上面的代码，我们只前向声明结构体 struct st_abc_private ，没人知道它里面具体是什么东西。假如 struct st_abc 对应的头文件是 abc.h ，那么把 st_abc_private 的声明放在 abc_p.h 中，abc.c 文件包含 abc_p.h ，那么在实现 struct st_abc 的函数指针 xyz_func 时如何使用 struct st_abc_private ，客户程序员根本无须知道。

这样做的好处是显而易见的，除了预定义好的接口，客户程序员完全不需要知道实现细节，即便实现经过重构完全重来，客户程序员也不需要关注，甚至相应的模块连重新编译都不要——因为 abc.h 自始至终都没变过。

上面代码有个问题，客户程序员如何得到 struct st_abc 的一个实例，他不知道 struct st_abc_private 如何实现的呀。 C 中没有构造函数，只好我们自己提供了：我们可以在 abc.h 中声明一个类似构造函数的函数来生成 struct st_abc 的实例，名字就叫作 new_abc() ，函数原型如下：

```
struct st_abc * new_abc();
```

至于实现，我们放在 abc.c 中，客户程序员不需要知道。相应的，还有个类似析构函数的函数，原型如下：

```
void delete_abc(struct st_abc *);
```

到现在为止，封装和隐藏就实现了，而且很彻底。接下来看继承。

什么是继承？在面向对象层面上不讲了，只说语法层面。语法层面上讲，继承就是派生类拥有父类的数据、方法，又添了点自己的东西，所谓子承父业，发扬光大。在 C 语言中可以用结构体的包含来实现继承关系。代码框架如下：

```
struct st_base{
    xxx;
};

struct st_derived{
    struct sb_base base;
    yyy;
};
```

代码上就是这么简单，不过有一点要注意：第一点就是派生类（结构体）中一定要把父类类型的成员放在第一个。

继承在语法层面上看，有数据成员、函数，数据成员通过上面的方法自动就“继承”了，至于函数，在结构体表示为函数指针，其实也是一个数据成员，是个指针而已，也会自动“继承”。之所以还要在这里列出来说明，是因为 C++ 中有一个很重要的概念：重载。要在 C 中完整实现有点儿麻烦。

重载，我们常说的重载大概有三种含义：

其一，函数重载，指函数名字一样，参数个数、类型不一样的函数声明和实现。由于 C 编译器的缘故，不支持。不过这个影响不大。

其二，重定义或者说覆盖，指派生类中定义与基类签名一样（名字、返回值、参数完全一样）的非虚函数，这样派生类的中的函数会覆盖基类的同签名函数，通过成员操作符访问时无法访问基类的同签名函数。

其三，虚函数重写，指在派生类中实现基类定义的虚函数或纯虚函数。虚函数是实现多态的关键，可以在结构体中使用函数指针来表达，但要完全实现，也很麻烦。

我们平常在交流时通常不明确区分上面三种类型的重载，这里出于习惯，也不作区分。 好了，第一篇就到这里，有时间会往下续。

C语言面向对象编程（二）：继承详解

在 C语言面向对象编程（一） 里说到继承，这里再详细说一下。

C++ 中的继承，从派生类与基类的关系来看（出于对比 C 与 C++，只说公有继承）：

- 派生类内部可以直接使用基类的 public 、protected 成员（包括变量和函数）
- 使用派生类的对象，可以像访问派生类自己的成员一样访问基类的成员
- 对于被派生类覆盖的基类的非虚函数，在派生类中可以通过基类名和域作用符（::）来访问
- 当使用基类指针调用虚函数时，会调用指针指向的实际对象实现的函数，如果该对象未重载该虚函数，则沿继承层次，逐级回溯，直到找到一个实现

上面的几个特点，我们在 C 语言中能否全部实现呢？我觉得可以实现类似的特性，但在使用方法上会有些区别。后面我们一个一个来说，在此之前呢，先说继承的基本实现。

先看 C 语言中通过“包含”模拟实现继承的简单代码框架：

```
struct base{
    int a;
};

struct derived{
    struct base parent;
    int b;
};

struct derived_2{
    struct derived parent;
    int b;
};
```

上面的示例只有数据成员，函数成员其实是个指针，可以看作数据成员。C 中的 struct 没有访问控制，默认都是公有访问（与 java 不同）。

下面是带成员函数的结构体：

```
struct base {
    int a;
    void (*func1)(struct base *_this);
};

struct derived {
    struct base parent;
    int b;
    void (*func2)(struct derived* _this);
};
```

为了像 C++ 中一样通过类实例来访问成员函数，必须将结构体内的函数指针的第一个参数定义为自身的指针，在调用时传入函数指针所属的结构体实例。这是因为 C 语言中不存在像 C++ 中那样的 this 指针，如果我们不显式地通过参数提供，那么在函数内部就无法访问结构体实例的其它成员。

下面是在 c 文件中实现的函数：

```
static void base_func1(struct base *_this)
{
    printf("this is base::func1\n");
}

static void derived_func2(struct derived *_this)
{
    printf("this is derived::func2\n");
}
```

C++ 的 new 操作符会调用构造函数，对类实例进行初始化。C 语言中只有 malloc 函数族来分配内存块，我们没有机会来自动初始化结构体的成员，只能自己增加一个函数。如下面这样（略去头文件中的声明语句）：

```
struct base * new_base()
{
    struct base * b = malloc(sizeof(struct base));
    b->a = 0;
    b->func1 = base_func1;
    return b;
}
```

好的，构造函数有了。通过 new_base() 调用返回的结构体指针，已经可以像类实例一样使用了：

```
struct base * b1 = new_base();
b1->func1(b1);
```

到这里我们已经知道如何在 C 语言中实现一个基本的“类”了。接下来一一来看前面提到的几点。

第一点，派生类内部可以直接使用基类的 public 、protected 成员（包括变量和函数）。具体到上面的例子，我们可以在 derived_func2 中访问基类 base 的成员 a 和 func1 ，没有任何问题，只不过是显式通过 derived 的第一个成员 parent 来访问：

```
static void derived_func2(struct derived *_this)
{
    printf("this is derived::func2, base::a = %d\n", _this->pare
nt.a);
    _this->parent.func1(&_this->parent);
}
```

第二点，使用派生类的对象，可以像访问派生类自己的成员一样访问基类的成员。这个有点变化，还是只能通过派生类实例的第一个成员 parent 来访问基类的成员（通过指针强制转换的话可以直接访问）。代码如下：

```
struct derived d;
printf("base::a = %d\n",d.parent.a);

struct derived *p = new_derived();
((struct base *)p)->func1(p);
```

第三点，对于被派生类覆盖的基类的非虚函数，在派生类中可以通过基类名和域作用符（::）来访问。其实通过前两点，我们已经熟悉了在 C 中访问“基类”成员的方法，总是要通过“派生类”包含的放在结构体第一个位置的基类类型的成员变量来访问。所以在 C 中，严格来讲，实际上不存在覆盖这种情况。即便定义了完全一样的函数指针，也没有关系，因为“包含”这种方式，已经从根本上分割了“基类”和“派生类”的成员，它们不在一个街区，不会冲突。

下面是一个所谓覆盖的例子：

```
struct base{
    int a;
    int (*func)(struct base * b);
};

struct derived {
    struct base b;
    int (*func)(struct derived *d);
};

// usage
struct derived * d = new_derived();
d->func(d);
d->b.func((struct base*)d);
```

如上面的代码所示，不存在名字覆盖问题。

第四点，虚函数。虚函数是 C++ 里面最有意义的一个特性，是多态的基础，要想讲明白比较困难，我们接下来专门写一篇文章讲述如何在 C 中实现类似虚函数的效果，实现多态。

C语言面向对象编程（三）：虚函数与多态

在《C++ 编程思想》一书中对虚函数的实现机制有详细的描述，一般的编译器通过虚函数表，在编译时插入一段隐藏的代码，保存类型信息和虚函数地址，而在调用时，这段隐藏的代码可以找到和实际对象一致的虚函数实现。

我们在这里提供一个 C 中的实现，模仿 VTABLE 这种机制，但一切都需要我们自己在代码中装配。

之前在网上看到一篇描述 C 语言实现虚函数和多态的文章，谈到在基类中保存派生类的指针、在派生类中保存基类的指针来实现相互调用，保障基类、派生类在使用虚函数时的行为和 C++ 类似。我觉得这种方法有很大的局限性，不说继承层次的问题，单单是在基类中保存派生类指针这一做法，就已经违反了虚函数和多态的本意——多态就是要通过基类接口来使用派生类，如果基类还需要知道派生类的信息……。

我的基本思路是：

- 在“基类”中显式声明一个 `void**` 成员，作为数组保存基类定义的所有函数指针，同时声明一个 int 类型的成员，指明 `void*` 数组的长度。
- “基类”定义的每个函数指针在数组中的位置、顺序是固定的，这是约定，必须的
- 每个“派生类”都必须填充基类的函数指针数组（可能要动态增长），没有重写虚函数时，对应位置置 0
- “基类”的函数实现中，遍历函数指针数组，找到继承层次中的最后一个非 0 的函数指针，就是实际应该调用的和对象相对应的函数实现

好了，先来看一点代码：

```
struct base {
    void ** vtable;
    int vt_size;

    void (*func_1)(struct base *b);
    int (*func_2)(struct base *b, int x);
};

struct derived {
    struct base b;
    int i;
};

struct derived_2{
    struct derived d;
    char *name;
};
```

上面的代码是我们接下来要讨论的，先说一点，在 C 中，用结构体内的函数指针和 C++ 的成员函数对应， C 的这种方式，所有函数都天生是虚函数（指针可以随时修改哦）。

注意，derived 和 derived_2 并没有定义 func_1 和 func_2 。在 C 的虚函数实现中，如果派生类要重写虚函数，不需要在派生类中显式声明。要做的是，在实现文件中实现你要重写的函数，在构造函数中把重写的函数填入虚函数表。

我们面临一个问题，派生类不知道基类的函数实现在什么地方（从高内聚、低耦合的原则来看），在构造派生类实例时，如何初始化虚函数表？在 C++ 中编译器会自动调用继承层次上所有父（祖先）类的构造函数，也可以显式在派生类的构造函数的初始化列表中调用基类的构造函数。怎么办？

我们提供一个不那么优雅的解决办法：

每个类在实现时，都提供两个函数，一个构造函数，一个初始化函数，前者用于生成一个类，后者用于继承层次紧接自己的类来调用以便正确初始化虚函数表。依据这样的原则，一个派生类，只需要调用直接基类的初始化函数即可，每个派生类都保证这一点，一切都可以进行下去。

下面是要实现的两个函数：

```
struct derived *new_derived();
void initialize_derived(struct derived *d);
```

new 开头的函数作为构造函数， initialize 开头的函数作为 初始化函数。我们看一下 new_derived 这个构造函数的实现框架：

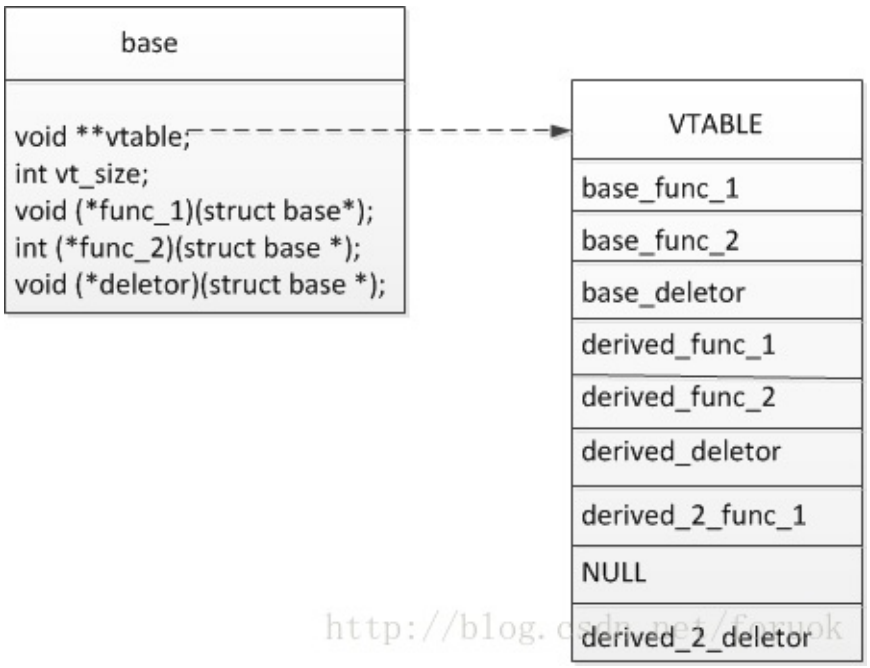
```
struct derived *new_derived()
{
    struct derived * d = malloc(sizeof(struct derived));
    initialize_base((struct base*)d);
    initialize_derived(d);/* setup or modify VTABLE */
    return d;
}
```

如果是 derived_2 的构造函数 new_derived_2，那么只需要调用 initialize_derived 即可。

说完了构造函数，对应的要说析构函数，而且析构函数要是虚函数。在删除一个对象时，需要从派生类的析构函数依次调用到继承层次最顶层的基类的析构函数。这点在 C 中也是可以保障的。做法是：给基类显式声明一个析构函数，基类的实现中查找虚函数表，从后往前调用即可。函数声明如下：

```
struct base {
    void ** vtable;
    int vt_size;

    void (*func_1)(struct base *b);
    int (*func_2)(struct base *b, int x);
    void (*deletor)(struct base *b);
};
```

说完构造、析构，该说这里的虚函数表到底是怎么回事了。我们先画个图，还是以刚才的 base 、 derived 、derived_2 为例来说明，一看图就明白了：



我们假定 derived 类实现了三个虚函数， derived_2 类实现了两个，func_2 没有实现，上图就是 derived_2 的实例所拥有的最终的虚函数表，表的长度（ vt_size ）是 9 。如果是 derived 的实例，就没有表中的最后三项，表的长度（ vt_size ）是 6 。

必须限制的是：基类必须实现所有的虚函数，只有这样，这套实现机制才可以运转下去。因为一切的发生是从基类的实现函数进入，通过遍历虚函数表来找到派生类的实现函数的。

当我们通过 base 类型的指针（实际指向 derived_2 的实例）来访问 func_1 时，基类实现的 func_1 会找到 VTABLE 中的 derived_2_func_1 进行调用。

好啦，到现在为止，基本说明白了实现原理，至于 初始化函数如何装配虚函数表、基类的虚函数实现，可以根据上面的思路写出代码来。按照我的这种方法实现的虚函数，通过基类指针访问，行为基本和 C++ 一致。

示例代码

多态，面向接口编程等设计方法并没有绑定到任何特定的语言上，使用纯C也可以实现简单的多态概念。下面给出一个非常简单粗糙的例子，只为说明概念。

父类Animal定义，文件：animal.h

```
#ifndef ANIMAL_H
#define ANIMAL_H

// 方法表, 类似于C++的虚函数表
typedef struct vtable vtable;
struct vtable
{
    void (*eat)();
    void (*bite)();
};

typedef struct Animal Animal;
struct Animal
{
    const vtable* _vptr;  // 每一个对象都有一个指向虚表的指针
};

/*
 如果不用虚表的话，每个对象都要包含所有的接口函数指针, 而实际上所有同类型对象的这些指针的值是相同的,造成内存浪费。
 接口函数是和类型一对一的，而不是和对象一对一。

struct Animal
{
    void (*eat)();
    void (*bite)();
};
*/

#endif
```

子类Dog，文件dog.h

```
#ifndef DOG_H
#define DOG_H

#include "animal.h"

typedef struct Dog Dog;
struct Dog
{
    Animal base_obj;
    int x;
};

Animal* create_dog();

#endif
```

dog.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "dog.h"

static void dog_eat()
{
    printf("dog_eat()\n");
};

static void dog_bite()
{
    printf("dog_bite()\n");
};

/* 虚表是在编译时确定的 */
static const vtable dog_vtable = {
    dog_eat,
    dog_bite
};

Animal* create_dog()
{
    Dog * pDog = malloc(sizeof(Dog));
    if(pDog){
        pDog->base_obj._vptr = &dog_vtable; /*运行时，绑定虚表指针*/

        pDog->x = 0;
    }
    return (Animal*)pDog;
}
```

另一个子类Cat, 文件cat.h

```
#ifndef CAT_H
#define CAT_H

#include "animal.h"

typedef struct Cat Cat;
struct Cat
{
    Animal base_obj;
    float y;
};

Animal* create_cat();

#endif
```

cat.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "animal.h"
#include "cat.h"

static void cat_eat()
{
    printf("cat_eat()\n");
};

static void cat_bite()
{
    printf("cat_bite()\n");
};

static const vtable cat_vtable = {
    cat_eat,
    cat_bite
};

Animal* create_cat()
{
    Cat * pCat = malloc(sizeof(Cat));
    if(pCat){
        pCat->base_obj._vptr = &cat_vtable;
        pCat->y = 0.0;
    }
    return (Animal*)pCat;
}
```

主文件 main.c

```
#include <stdio.h>
#include "animal.h"
#include "cat.h"
#include "dog.h"

void ShowBite(Animal* pAnimal)
{
    pAnimal->_vptr->bite();
}
void main()
{
    ShowBite(create_dog());
    ShowBite(create_cat());
}
```

用C语言完全模拟C++虚函数表的实现与运作方式

如果对前面两大节的描述仔细了解了的话，想用C语言来模拟C++的虚函数以及多态，想必是轻而易举的事情了!

前提

但是，话得说在前面，C++的编译器在生成类及对象的时候，帮助我们完成了很多事件，比如生成虚函数表! 但是，C语言编译器却没有，因此，很多事件我们必须手动来完成，包括但不限于:

1. 手动构造父子关系
2. 手动创建虚函数表
3. 手动设置__vfptr并指向虚函数表
4. 手动填充虚函数表
5. 若有虚函数覆盖，还需手动修改函数指针
6. 若要取得基类指针，还需手动强制转换
7. ......

总之，要想用C语言来实现，要写的代码绝对有点复杂。

C++原版调用

接下来，我们都将以最后那个，最繁杂的那个3个基类的实例来讲解，但作了一些简化与改动:

1. 用构造函数初始化成员变量
2. 减少成员变量的个数
3. 减少虚函数的个数
4. 调用函数时产生相关输出
5. Derive1增加一个基类虚函数覆盖

以下是对类的改动，很少：

```
class Base1
{
public:
```

```
    Base1() : base1_1(11) {}
    int base1_1;
    virtual void base1_fun1() {
        std::cout << "Base1::base1_fun1()" << std::endl;
    }
};

class Base2
{
public:
    Base2() : base2_1(21) {}
    int base2_1;
};

class Base3
{
public:
    Base3() : base3_1(31) {}
    int base3_1;
    virtual void base3_fun1() {
        std::cout << "Base3::base3_fun1()" << std::endl;
    }
};

class Derive1 : public Base1, public Base2, public Base3
{
public:
    Derive1() : derive1_1(11) {}
    int derive1_1;

    virtual void base3_fun1() {
        std::cout << "Derive1::base3_fun1()" << std::endl;
    }
    virtual void derive1_fun1() {
            std::cout << "Derive1::derive1_fun1()" << std::endl;
    }
};
```

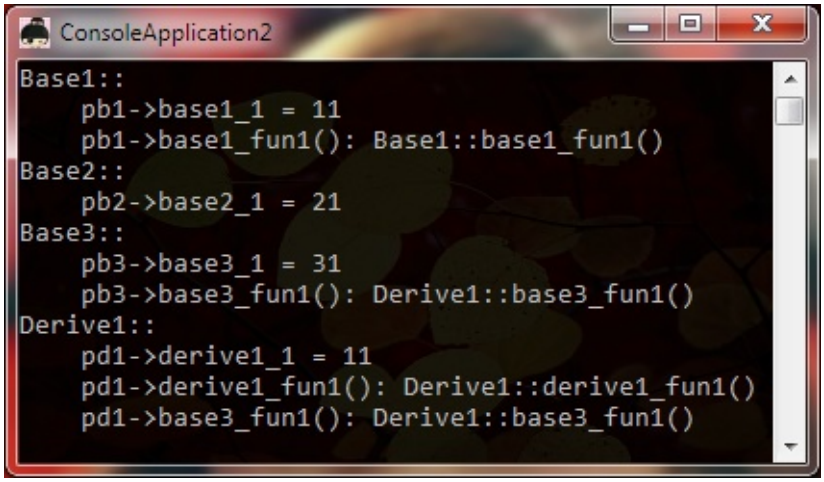
为了看到多态的效果，我们还需要定义一个函数来看效果:

```
void foo(Base1* pb1, Base2* pb2, Base3* pb3, Derive1* pd1)
{
    std::cout << "Base1::\n"
        << "    pb1->base1_1 = " << pb1->base1_1 << "\n"
        << "    pb1->base1_fun1(): ";
    pb1->base1_fun1();

    std::cout << "Base2::\n"
        << "    pb2->base2_1 = " << pb2->base2_1
        << std::endl;

    std::cout << "Base3::\n"
        << "    pb3->base3_1 = " << pb3->base3_1 << "\n"
        <<"    pb3->base3_fun1(): ";
    pb3->base3_fun1();

    std::cout << "Derive1::\n"
        << "    pd1->derive1_1 = " << pd1->derive1_1<< "\n"
        <<"    pd1->derive1_fun1(): ";
    pd1->derive1_fun1();
    std::cout<< "    pd1->base3_fun1(): ";
    pd1->base3_fun1();

    std::cout << std::endl;
}
```

调用方式如下:

```
Derive1 d1;
foo(&d1, &d1, &d1, &d1);
```

输出结果:



可以看到输出结果全部正确(当然了!)，哈哈

同时注意到 pb3->base3_fun1() 的多态效果哦!

用C语言来模拟

必须要把前面的理解了，才能看懂下面的代码!

为了有别于已经完成的C++的类，我们分别在类前面加一个大写的C以示区分(平常大家都是习惯在C++写的类前面加C，今天恰好反过来，哈哈)。

C语言无法实现的部分

C/C++是两个语言，有些语言特性是C++专有的，我们无法实现! 不过，这里我是指调用约定，我们应该把她排除在外。

对于类的成员函数，C++默认使用__thiscall，也即this指针通过ecx传递，这在C语言无法实现，所以我们必须手动声明调用约定为:

- `__stdcall` ，就像微软的组件对象模型那样
- `__cdecl` ，本身就C语言的调用约定，当然能使用了。

上面那种调用约定，使用哪一种无关紧要，反正不能使用 `__thiscall` 就行了。

因为使用了非__thiscall调用约定，我们就必须手动传入this指针，通过成员函数的第1个参数!

从最简单的开始: 实现 **Base2**

由于没有虚函数，仅有成员变量，这个当然是最好模拟的咯!

```
struct CBase2
{
    int base2_1;
};
```

有了虚函数表的**Base1**，但没被覆盖

下面是Base1的定义，要复杂一点了，多一个__vfptr：

```
struct CBase1
{
    void** __vfptr;
    int base1_1;
};
```

因为有虚函数表，所以还得单独为虚函数表创建一个结构体的哦!

但是，为了更能清楚起见，我并未定义前面所说的指针数组，而是用一个包含一个或多个函数指针的结构体来表示!

因为数组能保存的是同一类的函数指针，不太很友好!

但他们的效果是完全一样的，希望读者能够理解明白!

```
struct CBase1_VFTable
{
    void(__stdcall* base1_fun1)(CBase1* that);
};
```

注意: base1_fun1 在这里是一个指针变量!

注意: base1_fun1 有一个CBase1的指针，因为我们不再使用__thiscall，我们必须手动传入! Got it?

Base1的成员函数base1_fun1()我们也需要自己定义，而且是定义成全局的:

```
void __stdcall base1_fun1(CBase1* that)
{
    std::cout << "base1_fun1()" << std::endl;
}
```

有虚函数覆盖的**Base3**

虚函数覆盖在这里并不能体现出来，要在构造对象初始化的时候才会体现，所以:base3其实和Base1是一样的。

```
struct CBase3
{
    void** __vfptr;
    int base3_1;
};

struct CBase3_VFTable
{
    void(__stdcall* base3_fun1)(CBase3* that);
};
```

Base3的成员函数:

```
void __stdcall base3_fun1(CBase3* that)
{
    std::cout << "base3_fun1()" << std::endl;
}
```

定义继承类**CDerive1**

相对前面几个类来说，这个类要显得稍微复杂一些了，因为包含了前面几个类的内容:

```
struct CDerive1
{
    CBase1 base1;
    CBase3 base3;
    CBase2 base2;

    int derive1_1;
};
```

特别注意: CBase123的顺序不能错!

另外: 由于Derive1本身还有虚函数表，而且所以项是加到第一个虚函数表(CBase1)的后面的，所以此时的 `CBase1::__vfptr` 不应该单单指向 `CBase1_VFTable` ，而应该指向下面这个包含Derive1类虚函数表的结构体才行:

```
struct CBase1_CDerive1_VFTable
{
    void (__stdcall* base1_fun1)(CBase1* that);
    void(__stdcall* derive1_fun1)(CDerive1* that);
};
```

因为CDerive1覆盖了CBase3的base3_fun1()函数，所以不能直接用Base3的那个表:

```
struct CBase3_CDerive1_VFTable
{
    void(__stdcall* base3_fun1)(CDerive1* that);
};
```

Derive1覆盖 `Base3::base3_fun1()` 的函数以及自身定义的 `derive1_fun1()` 函数:

```
void __stdcall base3_derive1_fun1(CDerive1* that)
{
    std::cout << "base3_derive1_fun1()" << std::endl;
}

void __stdcall derive1_fun1(CDerive1* that)
{
    std::cout << "derive1_fun1()" << std::endl;
}
```

构造各类的全局虚函数表

由于没有了编译器的帮忙，在定义一个类对象时，所有的初始化工作都只能由我们自己来完成了!

首先构造全局的，被同一个类共同使用的虚函数表!

```
// CBase1 的虚函数表
CBase1_VFTable __vftable_base1;
__vftable_base1.base1_fun1 = base1_fun1;

// CBase3 的虚函数表
CBase3_VFTable __vftable_base3;
__vftable_base3.base3_fun1 = base3_fun1;
```

然后构造CDerive1和CBase1共同使用的虚函数表:

```
// CDerive1 和 CBase1 共用的虚函数表
CBase1_CDerive1_VFTable __vftable_base1_derive1;
__vftable_base1_derive1.base1_fun1 = base1_fun1;
__vftable_base1_derive1.derive1_fun1 = derive1_fun1;
```

再构造CDerive1覆盖CBase3后的虚函数表: 注意: 数覆盖会替换原来的函数指针

```
CBase3_CDerive1_VFTable __vftable_base3_derive1;
__vftable_base3_derive1.base3_fun1 = base3_derive1_fun1;
```

开始! 从**CDerive1**构造一个完整的**Derive1**类

先初始化成员变量与__vfptr的指向: 注意不是指错了!

```
CDerive1 d1;
d1.derive1 = 1;

d1.base1.base1_1 = 11;
d1.base1.__vfptr = reinterpret_cast<void**>(&__vftable_base1_der
ive1);

d1.base2.base2_1 = 21;

d1.base3.base3_1 = 31;
d1.base3.__vfptr = reinterpret_cast<void**>(&__vftable_base3_der
ive1);
```

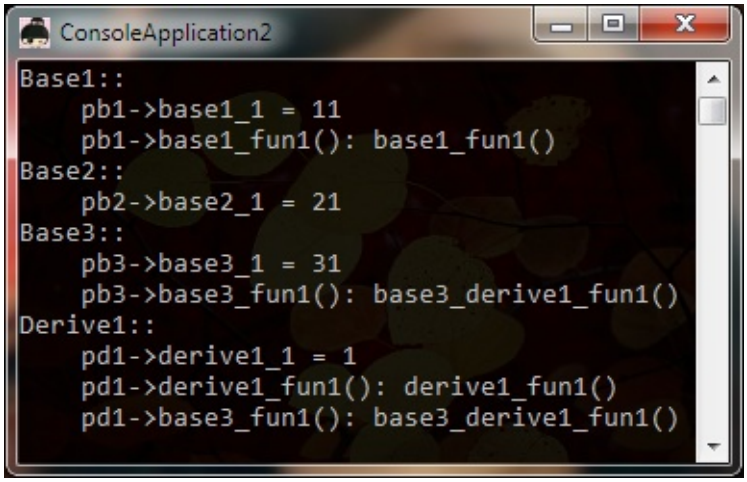
由于目前的CDerive1是我们手动构造的，不存在真正语法上的继承关系，如要得到各基类指针，我们就不能直接来取，必须手动根据偏移计算:

```
char* p = reinterpret_cast<char*>(&d1);
Base1* pb1 = reinterpret_cast<Base1*>(p + 0);
Base2* pb2 = reinterpret_cast<Base2*>(p + sizeof(CBase1) + sizeof
(CBase3));
Base3* pb3 = reinterpret_cast<Base3*>(p + sizeof(CBase1));
Derive1* pd1 = reinterpret_cast<Derive1*>(p);
```

真正调用:

```
foo(pb1, pb2, pb3, pd1);
```

调用结果:



结果相当正确!

源代码

我以为我把源代码搞丢了，结果过了一年多发现其实并没有。2015-12-24（每个圣诞我都在写代码）

有两个，忘了区别了：Source1.cpp，Source2.cpp

Source1.cpp

```
#include <iostream>

class Base1
{
public:
    int base1_1;
    int base1_2;

    virtual void __cdecl base1_fun(int x)
    {
        std::cout << "Base1::base1_fun(" << x <<")\n";
    }
};

class Base2
{
public:
```

```
    int base2_1;

    virtual void __cdecl base2_fun(int x)
    {
        std::cout << "Base2::base2_fun(" << x << ")\n";
    }

    int base2_2;
};

class Derive : public Base1, public Base2
{
public:
    Derive()
    {
        base1_1 = 11;
        base1_2 = 12;
        base2_1 = 21;
        base2_2 = 22;

        derive1 = 1;
        derive2 = 2;
    }

    int derive1;
    int derive2;

    virtual void __cdecl derive_fun1() {}
    virtual void __cdecl derive_fun2() {}
};

class Derive_Derive : public Derive
{
public:
    virtual void __cdecl derive_fun2() {}
    int ddd;
};

void foo2(Base1* pb1, Base2* pb2, Derive* pd, Derive_Derive* pdd)
```

```
{

}

void foo(Base1* pb1, Base2* pb2, Derive* pd)
{
    /*
    std::cout << "--------------------------------------" << std
::endl;
    std::cout << "pb1:\n"
        << "\t&pb1->base1_1  : " << &pb1->base1_1 << "\n"
        << "\t&pb1->base1_2  : " << &pb1->base1_2 << "\n"
        << "\t&pb1->base1_fun: " << &pb1->base1_fun << "\n"
        << std::endl;
    std::cout << "--------------------------------------" << std:
:endl;*/

    std::cout << "Base1:\n"
        << "\tbase1_1 = " << pb1->base1_1 << "\n"
        << "\tbase1_2 = " << pb1->base1_2 << "\n"
        << std::endl;

    std::cout << "Base2:\n"
        << "\tbase2_1 = " << pb2->base2_1 << "\n"
        << "\tbase2_2 = " << pb2->base2_2 << "\n"
        << std::endl;

    std::cout << "Derive:\n"
        << "\tderive1 = " << pd->derive1 << "\n"
        << "\tderive2 = " << pd->derive2 << "\n"
        << std::endl;

    pb1->base1_fun(11);
    pb2->base2_fun(22);

    pd->derive_fun1();
    pd->derive_fun2();
}

struct Base1_VPTR_VPTR{
```

```
    void (__cdecl* base1_fun)(Base1* that, int x);
};

struct Base1_VPTR{
    Base1_VPTR_VPTR* pvptr;
    int base1_1;
    int base1_2;
};

struct Base2_VPTR_VPTR{
    void (__cdecl* base2_fun)(Base2* that, int x);
};

struct Base2_VPTR{
    Base2_VPTR_VPTR* pvptr;
    int base2_1;
    int base2_2;
};

void __cdecl base1_fun(Base1* that, int x)
{
    std::cout << x << std::endl;
}

void __cdecl base2_fun(Base2* that, int x)
{
    std::cout << x << std::endl;
}

struct Derive_C
{
    Base1_VPTR base1;
    Base2_VPTR base2;

    int derive1;
    int derive2;
};

class Test1{
public:
```

```
    int a;
    int b;
    virtual void f1(){}
    virtual void f2(){}
};

int main()
{
    Base1 b1;
    std::cout << "offsetof Base1::base1_1 = " << (int)&b1.base1_
1 - (int)&b1 << std::endl;

    Derive d1;
    std::cout << "offsetof Derive::derive2 = " << (int)&d1.deriv
e2 - (int)&d1 << std::endl;

    Derive_Derive ddd;
    std::cout << "offsetof DD::ddd = " << (int)&ddd.ddd - (int)&
ddd << std::endl;
    foo2(&ddd, &ddd, &ddd, &ddd);

    Test1 test1;
    Test1 test2;

    // 同一个类型的不同对象的虚函数表地址相同
    //std::cout << *(int*)&test1 << std::endl;
    //std::cout << *(int*)&test2 << std::endl;

    std::cout << "sizeof(Base1  ) = " << sizeof(Base1   ) << std
::endl;
    std::cout << "sizeof(Base2  ) = " << sizeof(Base2   ) << std
::endl;
    std::cout << "sizeof(Test1   ) = " << sizeof(Test1   ) << std
::endl;
    std::cout << "sizeof(Derive  ) = " << sizeof(Derive  ) << std
::endl;
    std::cout << "sizeof(Derive_C) = " << sizeof(Derive_C) << std
::endl;

    Derive dd;
```

```
    foo(&dd, &dd, &dd);
    dd.derive_fun1();

    Derive_C d;
    Base1_VPTR_VPTR base1_vptr_vptr;
    Base2_VPTR_VPTR base2_vptr_vptr;

    base1_vptr_vptr.base1_fun = base1_fun;
    base2_vptr_vptr.base2_fun = base2_fun;

    d.base1.base1_1 = 11;
    d.base1.base1_2 = 12;
    d.base1.pvptr = &base1_vptr_vptr;

    d.base2.base2_1 = 21;
    d.base2.base2_2 = 22;
    d.base2.pvptr = &base2_vptr_vptr;

    d.derive1 = 1;
    d.derive2 = 2;

    foo((Base1*)&d.base1, (Base2*)&d.base2, (Derive*)&d);

    return 0;
}
```

Source2.cpp

```
#include <iostream>

class Base1
{
public:
    Base1() : base1_1(11) {}
    int base1_1;
    virtual void __stdcall base1_fun1() {
        std::cout << "Base1::base1_fun1()" << std::endl;
    }
```

```
};

class Base2
{
public:
    Base2() : base2_1(21) {}
    int base2_1;
};

class Base3
{
public:
    Base3() : base3_1(31) {}
    int base3_1;
    virtual void __stdcall base3_fun1() {
        std::cout << "Base3::base3_fun1()" << std::endl;
    }
};

class Derive1 : public Base1, public Base2, public Base3
{
public:
    Derive1() : derive1_1(11) {}
    int derive1_1;

    virtual void __stdcall base3_fun1() {
        std::cout << "Derive1::base3_fun1()" << std::endl;
    }
    virtual void __stdcall derive1_fun1() {
        std::cout << "Derive1::derive1_fun1()" << std::endl;
    }
};

struct CBase2
{
    int base2_1;
};

struct CBase1
{
```

```
    void** __vfptr;
    int base1_1;
};

struct CBase1_VFTable
{
    void (__stdcall* base1_fun1)(CBase1* that);
};

void __stdcall base1_fun1(CBase1* that)
{
    std::cout << "base1_fun1()" << std::endl;
}

struct CBase3
{
    void** __vfptr;
    int base3_1;
};

struct CBase3_VFTable
{
    void(__stdcall* base3_fun1)(CBase3* that);
};

void __stdcall base3_fun1(CBase3* that)
{
    std::cout << "base3_fun1()" << std::endl;
}

struct CDerive1
{
    CBase1 base1;
    CBase3 base3;
    CBase2 base2;

    int derive1_1;
};

struct CBase1_CDerive1_VFTable
```

```
{
    void (__stdcall* base1_fun1)(CBase1* that);
    void (__stdcall* derive1_fun1)(CDerive1* that);
};

struct CBase3_CDerive1_VFTable
{
    void(__stdcall* base3_fun1)(CDerive1* that);
};

void __stdcall base3_derive1_fun1(CDerive1* that)
{
    std::cout << "base3_derive1_fun1()" << std::endl;
}

void __stdcall derive1_fun1(CDerive1* that)
{
    std::cout << "derive1_fun1()" << std::endl;
}

void foo(Base1* pb1, Base2* pb2, Base3* pb3, Derive1* pd1)
{
    std::cout << "Base1::\n"
        << "    pb1->base1_1 = " << pb1->base1_1 << "\n"
        << "    pb1->base1_fun1(): ";
    pb1->base1_fun1();

    std::cout << "Base2::\n"
        << "    pb2->base2_1 = " << pb2->base2_1
        << std::endl;

    std::cout << "Base3::\n"
        << "    pb3->base3_1 = " << pb3->base3_1 << "\n"
        << "    pb3->base3_fun1(): ";
    pb3->base3_fun1();

    std::cout << "Derive1::\n"
        << "    pd1->derive1_1 = " << pd1->derive1_1 << "\n"
        << "    pd1->derive1_fun1(): ";
    pd1->derive1_fun1();
```

```
    std::cout << "    pd1->base3_fun1(): ";
    pd1->base3_fun1();

    std::cout << std::endl;
}

int main()
{
    // CBase1 的虚函数表
    CBase1_VFTable __vftable_base1;
    __vftable_base1.base1_fun1 = base1_fun1;

    // CBase3 的虚函数表
    CBase3_VFTable __vftable_base3;
    __vftable_base3.base3_fun1 = base3_fun1;

    // CDerive1 和 CBase1 共用的虚函数表
    CBase1_CDerive1_VFTable __vftable_base1_derive1;
    __vftable_base1_derive1.base1_fun1 = base1_fun1;
    __vftable_base1_derive1.derive1_fun1 = derive1_fun1;

    CBase3_CDerive1_VFTable __vftable_base3_derive1;
    __vftable_base3_derive1.base3_fun1 = base3_derive1_fun1;

    CDerive1 d1;
    d1.derive1_1 = 1;

    d1.base1.base1_1 = 11;
    d1.base1.__vfptr = reinterpret_cast<void**>(&__vftable_base1
_derive1);

    d1.base2.base2_1 = 21;

    d1.base3.base3_1 = 31;
    d1.base3.__vfptr = reinterpret_cast<void**>(&__vftable_base3
_derive1);

    char* p = reinterpret_cast<char*>(&d1);
    Base1* pb1 = reinterpret_cast<Base1*>(p + 0);
    Base2* pb2 = reinterpret_cast<Base2*>(p + sizeof(CBase1) + s
```

```
izeof(CBase3));
    Base3* pb3 = reinterpret_cast<Base3*>(p + sizeof(CBase1));
    Derive1* pd1 = reinterpret_cast<Derive1*>(p);

    foo(pb1, pb2, pb3, pd1);

    return 0;
}
```

C语言面向对象编程（四）：面向接口编程

Java 中有 interface 关键字，C++ 中有抽象类或纯虚类可以与 interface 比拟，C 语言中也可以实现类似的特性。

在面试 Java 程序员时我经常问的一个问题是：接口和抽象类有什么区别。

很多编程书籍也经常说要面向接口编程，我的理解是，接口强制派生类必须实现基类（接口）定义的契约，而抽象类则允许实现继承从而导致派生类可以不实现（重写）基类（接口）定义的契约。通常这不是问题，但在有一些特定的情况，看起来不那么合适。

比如定义一个 Shape 基类，其中定义一个 draw() 方法，给一个什么都不做的默认实现（通常是空函数体），这实际没有任何意义。

再比如基类改变某个方法的实现，而派生类采用实现继承并没有重写这个方法，此时可能会导致一些奇怪的问题。以鸟为例，基类为 Bird ，我们可能会定义一个 fly() 方法，一个 walk() 方法，因为有的人认为鸟既可以走又可以飞。开始时我们在 walk() 的实现里作了假定，认为双脚交叉前进才是 walk ，可是后来发现有些鸟是双脚一齐蹦的，不会交叉前进。这个时候怎么办？基类 Bird 的 walk() 方法是否要修改、如何修改？

在 C++ 中，没有接口关键字 interface ，同时为了代码复用，经常采用实现继承。在 C 语言中，我们前面几篇文章讨论了封装、隐藏、继承、虚函数、多态等概念，虽然都可以实现，但使用起来总不如自带这些特性的语言（如 C++ 、Java ）等得心应手。一旦你采用我们前面描述的方法来进行面向对象编程，就会发现，在 C 语言中正确的维护类层次是一件非常繁琐、容易出错的事情，而且要比面向对象的语言多写很多代码（这很容易理解，面向对象语言自带轮子，而 C 要自己造轮子，每实现一个类都要造一遍）。但有一点，当我们使用 C 语言作面向对象编程时，比 C++ 有明显的优势，那就是接口。

接口强制派生类实现，这点在 C 中很容易做到。而且我们在编程中，实际上多数时候也不需要那么多的继承层次，一个接口类作为基类，一个实现类继承接口类，这基本就够了。在 C 语言中采用这种方式，可以不考虑析构函数、超过 3 层继承的上下类型转换、虚函数调用回溯、虚函数表装配等等问题，我们所要做的，就是实现基类接口，通过基类指针，就只能操作继承层次中最底层的那个类的对象；而基类接口，天生就是不能实例化的（其实是实例化了没办法使用，因为结构体的函数指针没人给它赋值）。

一个示例如下：

```
struct base_interface {
    void (*func1)(struct base_interface* b);
    void (*func2)(struct base_interface* b);
    int (*func_3)(struct base_interface* b, char * arg);
};

struct derived {
    struct base_interface bi;
    int x;
    char ch;
    char *name;
};
```

上面是头文件，derived 结构体通过包含 base_interface 类型的成员 bi 来达到继承的效果；而 base_interface 无法实例化，我们没有提供相应的构造函数，也没有提供与 func_1 ， func_2 等函数指针对应的实现，即便有人 malloc 了一个 base_interface ，也无法使用。

derived 类可以提供一个构造函数 new_derived ，同时在实现文件中提供 func_1 ，func_2 ，func_3 的实现并将函数地址赋值给 bi 的成员，从而完成 derived 类的装配，实现 base_interface 定义的契约。

示例实现如下：

```
static void _derived_func_1(struct base_interface *bi)
{
    struct derived * d = (struct derived*)bi;
    d->x *= 2;
    printf("d->name = %s\n", d->name);
}

/* _derived_func_2 impl */
/* _derived_func_3 impl */

struct derived *new_derived()
{
    struct derived *d = malloc(sizeof(struct derived));
    d->bi.func_1 = _derived_func_1;
    d->bi.func_2 = _derived_func_2;
    d->bi.func_3 = _derived_func_3;
    d->x = 0;
    d->ch = 'a';
    d->name = NULL;

    return d;
}
```

我们可以这么使用 base_interface 接口：

```
void do_something(struct base_interface *bi)
{
    bi->func_1(bi);
}

int main(int argc, char **argv)
{
    struct derived * d = new_derived();
    do_something((struct base_interface*)d);

    return 0;
}
```

上面的代码中 do_something 函数完全按照接口编程，而 bi 可以实际指向任意一个实现了 base_interface 接口的类的实例，在一定程序上达到多态的效果，花费的代价相当小，却可以让我们的程序提高可扩展性，降低耦合。

这种简单的方法也是我在自己的项目中使用的方法，效果不错。

C语言面向对象编程（五）：单链表实现

前面我们介绍了如何在 C 语言中引入面向对象语言的一些特性来进行面向对象编程，从本篇开始，我们使用前面提到的技巧，陆续实现几个例子，最后呢，会提供一个基本的 http server 实现（使用 libevent ）。在这篇文章里，我们实现一个通用的数据结构：单链表。

这里实现的单链表，可以存储任意数据类型，支持增、删、改、查找、插入等基本操作。（本文提供的是完整代码，可能有些长。）

下面是头文件：

```
#ifndef SLIST_H
#define SLIST_H

#ifdef __cplusplus
extern "C" {
#endif

#define NODE_T(ptr, type) ((type*)ptr)

struct slist_node {
    struct slist_node * next;
};

typedef void (*list_op_free_node)(struct slist_node *node);
/*
 * return 0 on hit key, else return none zero
 */
typedef int (*list_op_key_hit_test)(struct slist_node *node, void
 *key);

struct single_list {
    /* all the members must not be changed manually by callee */

    struct slist_node * head;
    struct slist_node * tail;
    int size; /* length of the list, do not change it manually*/
```

```
    /* free method to delete the node
     */
    void (*free_node)(struct slist_node *node);
    /*
     * should be set by callee, used to locate node by key(*_by_
key() method)
     * return 0 on hit key, else return none zero
     */
    int (*key_hit_test)(struct slist_node *node, void *key);

    struct single_list *(*add)(struct single_list * list, struct
 slist_node * node);
    struct single_list *(*insert)(struct single_list * list, int
 pos, struct slist_node *node);
    /* NOTE: the original node at the pos will be freed by free_
node */
    struct single_list *(*replace)(struct single_list *list, int
 pos, struct slist_node *node);
    struct slist_node *(*find_by_key)(struct single_list *, void
 * key);
    struct slist_node *(*first)(struct single_list* list);
    struct slist_node *(*last)(struct single_list* list);
    struct slist_node *(*at)(struct single_list * list, int pos)
;
    struct slist_node *(*take_at)(struct single_list * list, int
 pos);
    struct slist_node *(*take_by_key)(struct single_list * list,
void *key);
    struct single_list *(*remove)(struct single_list * list, str
uct slist_node * node);
    struct single_list *(*remove_at)(struct single_list *list, i
nt pos);
    struct single_list *(*remove_by_key)(struct single_list *list
, void *key);
    int (*length)(struct single_list * list);
    void (*clear)(struct single_list * list);
    void (*deletor)(struct single_list *list);
};
```

```
struct single_list * new_single_list(list_op_free_node op_free,
list_op_key_hit_test op_cmp);

#ifdef __cplusplus
}
#endif

#endif // SLIST_H
```

struct single_list 这个类，遵循我们前面介绍的基本原则，不再一一细说。有几点需要提一下：

- 我们定义了 slist_node 作为链表节点的基类，用户自定义的节点，都必须从 slist_node 继承
- 为了支持节点（ node ）的释放，我们引入一个回调函数 list_op_free_node ，这个回调需要在创建链表时传入
- 为了支持查找，引入另外一个回调函数 list_op_key_hit_test

好了，下面看实现文件：

```
#include "slist.h"
#include <malloc.h>

static struct single_list * _add_node(struct single_list *list,
struct slist_node *node)
{

    if(list->tail)
    {
        list->tail->next = node;
        node->next = 0;
        list->tail = node;
        list->size++;
    }
    else
    {
        list->head = node;
        list->tail = node;
        node->next = 0;
```

```
        list->size = 1;
    }

    return list;
}

static struct single_list * _insert_node(struct single_list * li
st, int pos, struct slist_node *node)
{
    if(pos < list->size)
    {
        int i = 0;
        struct slist_node * p = list->head;
        struct slist_node * prev = list->head;
        for(; i < pos; i++)
        {
            prev = p;
            p = p->next;
        }
        if(p == list->head)
        {
            /* insert at head */
            node->next = list->head;
            list->head = node;
        }
        else
        {
            prev->next = node;
            node->next = p;
        }

        if(node->next == 0) list->tail = node;
        list->size++;
    }
    else
    {
        list->add(list, node);
    }

    return list;
```

```
}

static struct single_list * _replace(struct single_list * list,
int pos, struct slist_node *node)
{
    if(pos < list->size)
    {
        int i = 0;
        struct slist_node * p = list->head;
        struct slist_node * prev = list->head;
        for(; i < pos; i++)
        {
            prev = p;
            p = p->next;
        }
        if(p == list->head)
        {
            /* replace at head */
            node->next = list->head->next;
            list->head = node;
        }
        else
        {
            prev->next = node;
            node->next = p->next;
        }

        if(node->next == 0) list->tail = node;

        if(list->free_node) list->free_node(p);
        else free(p);
    }

    return list;
}

static struct slist_node * _find_by_key(struct single_list *list
, void * key)
{
    if(list->key_hit_test)
```

```
    {
        struct slist_node * p = list->head;
        while(p)
        {
            if(list->key_hit_test(p, key) == 0) return p;
            p = p->next;
        }
    }
    return 0;
}

static struct slist_node *_first_of(struct single_list* list)
{
    return list->head;
}

static struct slist_node *_last_of(struct single_list* list)
{
    return list->tail;
}

static struct slist_node *_node_at(struct single_list * list, int
 pos)
{
    if(pos < list->size)
    {
        int i = 0;
        struct slist_node * p = list->head;
        for(; i < pos; i++)
        {
            p = p->next;
        }
        return p;
    }

    return 0;
}

static struct slist_node * _take_at(struct single_list * list, i
nt pos)
```

```
{
    if(pos < list->size)
    {
        int i = 0;
        struct slist_node * p = list->head;
        struct slist_node * prev = p;
        for(; i < pos ; i++)
        {
            prev = p;
            p = p->next;
        }
        if(p == list->head)
        {
            list->head = p->next;
            if(list->head == 0) list->tail = 0;
        }
        else if(p == list->tail)
        {
            list->tail = prev;
            prev->next = 0;
        }
        else
        {
            prev->next = p->next;
        }

        list->size--;

        p->next = 0;
        return p;
    }

    return 0;
}

static struct slist_node * _take_by_key(struct single_list * list
, void *key)
{
    if(list->key_hit_test)
    {
```

```
        struct slist_node * p = list->head;
        struct slist_node * prev = p;
        while(p)
        {
            if(list->key_hit_test(p, key) == 0) break;
            prev = p;
            p = p->next;
        }

        if(p)
        {
            if(p == list->head)
            {
                list->head = p->next;
                if(list->head == 0) list->tail = 0;
            }
            else if(p == list->tail)
            {
                list->tail = prev;
                prev->next = 0;
            }
            else
            {
                prev->next = p->next;
            }

            list->size--;

            p->next = 0;
            return p;
        }
    }
    return 0;
}

static struct single_list *_remove_node(struct single_list * list
, struct slist_node * node)
{
    struct slist_node * p = list->head;
    struct slist_node * prev = p;
```

```
    while(p)
    {
        if(p == node) break;
        prev = p;
        p = p->next;
    }

    if(p)
    {
        if(p == list->head)
        {
            list->head = list->head->next;
            if(list->head == 0) list->tail = 0;
        }
        else if(p == list->tail)
        {
            prev->next = 0;
            list->tail = prev;
        }
        else
        {
            prev->next = p->next;
        }

        if(list->free_node) list->free_node(p);
        else free(p);

        list->size--;
    }
    return list;
}

static struct single_list *_remove_at(struct single_list *list,
int pos)
{
    if(pos < list->size)
    {
        int i = 0;
        struct slist_node * p = list->head;
        struct slist_node * prev = p;
```

```
        for(; i < pos ; i++)
        {
            prev = p;
            p = p->next;
        }
        if(p == list->head)
        {
            list->head = p->next;
            if(list->head == 0) list->tail = 0;
        }
        else if(p == list->tail)
        {
            list->tail = prev;
            prev->next = 0;
        }
        else
        {
            prev->next = p->next;
        }

        if(list->free_node) list->free_node(p);
        else free(p);

        list->size--;
    }

    return list;
}

static struct single_list *_remove_by_key(struct single_list *li
st, void *key)
{
    if(list->key_hit_test)
    {
        struct slist_node * p = list->head;
        struct slist_node * prev = p;
        while(p)
        {
            if(list->key_hit_test(p, key) == 0) break;
            prev = p;
```

```
            p = p->next;
        }

        if(p)
        {
            if(p == list->head)
            {
                list->head = list->head->next;
                if(list->head == 0) list->tail = 0;
            }
            else if(p == list->tail)
            {
                prev->next = 0;
                list->tail = prev;
            }
            else
            {
                prev->next = p->next;
            }

            if(list->free_node) list->free_node(p);
            else free(p);

            list->size--;
        }
    }

    return list;
}

static int _length_of(struct single_list * list)
{
    return list->size;
}

static void _clear_list(struct single_list * list)
{
    struct slist_node * p = list->head;
    struct slist_node * p2;
    while(p)
```

```
    {
        p2 = p;
        p = p->next;

        if(list->free_node) list->free_node(p2);
        else free(p2);
    }

    list->head = 0;
    list->tail = 0;
    list->size = 0;
}

static void _delete_single_list(struct single_list *list)
{
    list->clear(list);
    free(list);
}

struct single_list * new_single_list(list_op_free_node op_free,
list_op_key_hit_test op_cmp)
{
    struct single_list *list = (struct single_list *)malloc(size
of(struct single_list));
    list->head = 0;
    list->tail = 0;
    list->size = 0;
    list->free_node = op_free;
    list->key_hit_test = op_cmp;

    list->add = _add_node;
    list->insert = _insert_node;
    list->replace = _replace;
    list->find_by_key = _find_by_key;
    list->first = _first_of;
    list->last = _last_of;
    list->at = _node_at;
    list->take_at = _take_at;
    list->take_by_key = _take_by_key;
    list->remove = _remove_node;
```

```
    list->remove_at = _remove_at;
    list->remove_by_key = _remove_by_key;
    list->length = _length_of;
    list->clear = _clear_list;
    list->deletor = _delete_single_list;

    return list;
}
```

上面的代码就不一一细说了，下面是测试代码：

```c
/* call 1 or N arguments function of struct */
#define ST_CALL(THIS,func,args...) ((THIS)->func(THIS,args))

/* call none-arguments function of struct */
#define ST_CALL_0(THIS,func) ((THIS)->func(THIS))

struct int_node {
    struct slist_node node;
    int id;
};

struct string_node {
    struct slist_node node;
    char name[16];
};


static int int_free_flag = 0;
static void _int_child_free(struct slist_node *node)
{
    free(node);
    if(!int_free_flag)
    {
        int_free_flag = 1;
        printf("int node free\n");
    }
}

static int _int_slist_hittest(struct slist_node * node, void *ke
```

```
y)
{
    struct int_node * inode = NODE_T(node, struct int_node);
    int ikey = (int)key;
    return (inode->id == ikey ? 0 : 1);
}

static int string_free_flag = 0;
static void _string_child_free(struct slist_node *node)
{
    free(node);
    if(!string_free_flag)
    {
        string_free_flag = 1;
        printf("string node free\n");
    }
}

static int _string_slist_hittest(struct slist_node * node, void
*key)
{
    struct string_node * sn = (struct string_node*)node;
    return strcmp(sn->name, (char*)key);
}

void int_slist_test()
{
    struct single_list * list = new_single_list(_int_child_free,
 _int_slist_hittest);
    struct int_node * node = 0;
    struct slist_node * bn = 0;
    int i = 0;

    printf("create list && nodes:\n");
    for(; i < 100; i++)
    {
        node = (struct int_node*)malloc(sizeof(struct int_node))
;
        node->id = i;
        if(i%10)
```

```
        {
            list->add(list, node);
        }
        else
        {
            list->insert(list, 1, node);
        }
    }
    printf("create 100 nodes end\n----\n");
    printf("first is : %d, last is: %d\n----\n",
           NODE_T( ST_CALL_0(list, first), struct int_node )->id
,
           NODE_T( ST_CALL_0(list, last ), struct int_node )->id
);

    assert(list->size == 100);

    printf("list traverse:\n");
    for(i = 0; i < 100; i++)
    {
        if(i%10 == 0) printf("\n");
        bn = list->at(list, i);
        node = NODE_T(bn, struct int_node);
        printf(" %d", node->id);
    }
    printf("\n-----\n");

    printf("find by key test, key=42:\n");
    bn = list->find_by_key(list, (void*)42);
    assert(bn != 0);
    node = NODE_T(bn, struct int_node);
    printf("find node(key=42), %d\n------\n", node->id);

    printf("remove node test, remove the 10th node:\n");
    bn = list->at(list, 10);
    node = NODE_T(bn, struct int_node);
    printf("  node 10 is: %d\n", node->id);
    printf("  now remove node 10\n");
    list->remove_at(list, 10);
    printf(" node 10 was removed, check node 10 again:\n");
```

```
    bn = list->at(list, 10);
    node = NODE_T(bn, struct int_node);
    printf("  now node 10 is: %d\n------\n", node->id);

    printf("replace test, replace node 12 with id 1200:\n");
    bn = list->at(list, 12);
    node = NODE_T(bn, struct int_node);
    printf("  now node 12 is : %d\n", node->id);
    node = (struct int_node*)malloc(sizeof(struct int_node));
    node->id = 1200;
    list->replace(list, 12, node);
    bn = list->at(list, 12);
    node = NODE_T(bn, struct int_node);
    printf("  replaced, now node 12 is : %d\n----\n", node->id);


    printf("test remove:\n");
    ST_CALL(list, remove, bn);
    bn = ST_CALL(list, find_by_key, (void*)1200);
    assert(bn == 0);
    printf("test remove ok\n----\n");
    printf("test remove_by_key(90):\n");
    ST_CALL(list, remove_by_key, (void*)90);
    bn = ST_CALL(list, find_by_key, (void*)90);
    assert(bn == 0);
    printf("test remove_by_key(90) end\n----\n");
    printf("test take_at(80):\n");
    bn = ST_CALL(list, take_at, 80);
    printf("  node 80 is: %d\n", NODE_T(bn, struct int_node)->id
);
    free(bn);
    printf("test take_at(80) end\n");

    int_free_flag = 0;
    printf("delete list && nodes:\n");
    list->deletor(list);
    printf("delete list && nodes end\n");
    printf("\n test add/insert/remove/delete/find_by_key/replace
...\n");
}
```

```
void string_slist_test()
{
    struct single_list * list = new_single_list(_string_child_fr
ee, _string_slist_hittest);
}

void slist_test()
{
    int_slist_test();
    string_slist_test();
}
```

测试代码里主要演示了：

- 自定义链表节点类型
- 定义释放回调
- 定义用于查找的 hit test 回调
- 如何创建链表
- 如何使用（ add 、remove 、 take 、find 、 insert 等）

相信到这里，单链表的使用已经不成问题了。

以单链表为基础，可以进一步实现很多数据结构，比如树（兄弟孩子表示法），比如 key-value 链表等等。接下来根据例子的需要，会择机进行展示。

C语言面向对象编程（六）：配置文件解析

在实际项目中，经常会把软件的某些选项写入配置文件。Windows 平台上的 INI 文件格式简单易用，本篇文章利用《C语言面向对象编程（五）：单链表实现》中实现的单链表，设计了一个“类” ini_parser 来读写 INI 格式的配置文件。

struct ini_parser 可以解析 INI 格式的字符串、文件，也可以将内存中的符合 INI 格式的数据写入文件，能够支持 Windows 、 Linux 、 Android 等多平台。目前暂不支持选项分组功能。

功能相对简单，直接看源码吧。

下面是头文件：

```
struct single_list;

struct ini_parser {
    struct single_list * keyvalues;
    int (*parse_file)(struct ini_parser *, const char * file);
    int (*parse_string)(struct ini_parser *, const char *text);

    char * (*value)(struct ini_parser *, const char * key);
    void (*set_value)(struct ini_parser *, const char * key, con
st char * value);
    void (*remove)(struct ini_parser *, const char *key);
    int (*save_to_file)(struct ini_parser *, const char * file);

    void (*deletor)(struct ini_parser *ini);
};

struct ini_parser * new_ini_parser();
```

struct init_parser 的声明符合我们在本系列文章中提到的面向对象框架，需要说明的是，一旦 deletor 方法被调用， ini_parser 的实例将不再允许访问。

下面是源文件：

```
#include "ini_parser.h"
```

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

struct tag_value_pair{
    struct slist_node node;
    char * szTag;
    char * szValue;
};
typedef struct tag_value_pair tag_value;

static void _tag_value_free(struct slist_node *node)
{
    if(node) delete_tag_value_pair(node);
}

static int _tag_value_hittest(struct slist_node * node, void *ke
y)
{
    return strcmp((char*)tag, ((struct tag_value_pair*)node)->sz
Tag);
}

static struct single_list * new_tag_value_list()
{
    return new_single_list(_tag_value_free, _tag_value_hittest);

}

static struct tag_value_pair *new_tag_value_pair()
{
    struct tag_value_pair * pair = (struct tag_value_pair *)mall
oc(sizeof(struct tag_value_pair));
    pair->node.next = 0;
    pair->szTag = 0;
    pair->szValue = 0;
    return pair;
}

static struct tag_value_pair * make_tag_value_pair(char * tag, c
har * value)
```

```
{
    struct tag_value_pair *pair = 0;
    if(!tag || !value)return 0;

    pair = (struct tag_value_pair*)malloc(sizeof(struct tag_valu
e_pair));
    pair->szTag = strdup(tag);
    pair->szValue = strdup(value);
    pair->node.next = 0;
    return pair;
}


static struct tag_value_pair * parse_line(char *line, int len)
{
    struct tag_value_pair * pair = 0;
    int count = 0;
    char * p = line;
    char * end = 0;m
    char * start = line;
    if(!p) return 0;
    while(*p == ' ') ++p;


    /*blank line*/
    if(p - line == len ||
            *p == '\r' ||
            *p == '\n' ||
            *p == '\0') return 0;

    /*do not support group*/
    if(*p == '[') return 0;
    /*comments*/
    if(*p == '#') return 0;

    /* extract key */
    start = p;
    end = line + len;
    while(*p != '=' && p!= end) ++p;
    if(p == end)
```

```
    {
        /* none '=' , invalid line */
        return 0;
    }
    end = p - 1;
    while(*end == ' ') --end; /* skip blank at the end */
    count = end - start + 1;

    pair = new_tag_value_pair();
    pair->szTag = malloc(count + 1);
    strncpy(pair->szTag, start, count);
    pair->szTag[count] = 0;

    /* extract value */
    ++p;
    end = line + len; /* next pos of the last char */
    while( *p == ' ' && p != end) ++p;
    if(p == end)
    {
        delete_tag_value_pair(pair);
        return 0;
    }
    start = p;
    --end; /* to the last char */
    if(*end == '\n') { *end = 0; --end; }
    if(*end == '\r') { *end = 0; --end; }
    count = end - start + 1;
    if(count > 0)
    {
        pair->szValue = malloc(count + 1);
        strncpy(pair->szValue, start, count);
        pair->szValue[count] = 0;
    }

    /* release empty key-value pair */
    if(!pair->szValue)
    {
        delete_tag_value_pair(pair);
        return 0;
    }
```

```
    return pair;
}

static int _parse_file(struct ini_parser * ini, const char *file
){
    FILE * fp = fopen(file, "r");
    if(fp)
    {
        struct tag_value_pair * pair = 0;
        char buf[1024] = {0};
        while(fgets(buf, 1024, fp))
        {
            pair = parse_line(buf, strlen(buf));
            if(pair)
            {
                ini->keyvalues->add(ini->keyvalues, pair);
            }
        }
        fclose(fp);
        return ini->keyvalues->size;
    }
    return -1;
}

static int _parse_text(struct ini_parser * ini, const char * tex
t){
    char *p = text;
    char * start = 0;
    struct tag_value_pair * pair = 0;
    if(!text) return -1;

    while(1)
    {
        start = p;
        while(*p != '\n' && *p != '\0' )++p;
        if(*p == '\0') break;

        pair = parse_line(start, p - start);
        if(pair) ini->keyvalues->add(ini->keyvalues, pair);
```

```
        ++p;
    }

    return ini->keyvalues->size;
}

static char * _value(struct ini_parser * ini, const char * key){

    struct tag_value_pair * pair = NODE_T(ini->keyvalues->find_b
y_key(ini->keyvalues, key), struct tag_value_pair);
    if(pair) return pair->szValue;
    return 0;
}

static void _set_value(struct ini_parser * ini, const char * key
, const char *value){
    struct tag_value_pair * pair = NODE_T(ini->keyvalues->find_b
y_key(ini->keyvalues, key), struct tag_value_pair);
    if(pair)
    {
        if(pair->szValue) free(pair->szValue);
        pair->szValue = strdup(value);
    }
    else
    {
        ini->keyvalues->add(ini->keyvalues, make_tag_value_pair(
key, value));
    }
}

static void _remove(struct ini_parser * ini, const char * key){

    struct tag_value_pair * pair = NODE_T(ini->keyvalues->find_b
y_key(ini->keyvalues, key), struct tag_value_pair);
    if(pair)ini->keyvalues->remove(ini->keyvalues, pair);
}

static void write_keyvalue(struct tag_value_pair * pair, FILE *f
p)
```

```
{
    fputs(pair->szTag, fp);
    fputc('=', fp);
    fputs(pair->szValue, fp);
    fputc('\n', fp);
}

static int _save_to_file(struct ini_parser * ini, const char * f
ile){
    if(ini->keyvalues->size > 0)
    {
        FILE * fp = fopen(file, "w");
        if(fp)
        {
            struct tag_value_pair * pair = NODE_T(ini->keyvalues
->head,struct tag_value_pair);
            while(pair != 0)
            {
                write_keyvalue(pair, fp);
                pair = NODE_T(pair->node.next, struct tag_value_
pair);
            }

            fclose(fp);
            return 0;
        }
    }
    return -1;
}

static void _delete_ini_parser(struct ini_parser *ini){
    if(ini)
    {
        ini->keyvalues->deletor(ini->keyvalues);
        free(ini);
    }
}

struct ini_parser * new_ini_parser(){
    struct ini_parser * ini = (struct ini_parser*)malloc(sizeof(
```

```
struct ini_parser));
    ini->keyvalues = new_tag_value_list();
    ini->parse_file = _parse_file;
    ini->parse_string = _parse_text;
    ini->value = _value;
    ini->set_value = _set_value;
    ini->remove = _remove;
    ini->save_to_file = _save_to_file;
    ini->deletor = _delete_ini_parser;
    return ini;
}
```

下面是简单的测试代码：

```
static char * g_szIniString = "#abc\nfirst=2\nsecond\nname=charl
i  zhang \n";

static void ini_parser_test_string()
{
    struct ini_parser * ini = new_ini_parser();
    int size = ini->parse_string(ini, g_szIniString);

    assert( size > 0);
    assert( ini->value(ini, "second") == 0 );
    assert( ini->value(ini, "abc") == 0);
    assert( ini->value(ini, "name") != NULL );
    assert( ini->value(ini, "first") != NULL);

    printf("ini string: %s\n", g_szIniString);
    printf("key-value pairs count = %d\n", size);
    printf("key \'name\'', value = %s\n", ini->value(ini, "name"
));
    printf("key \'first\'', value = %s\n", ini->value(ini, "firs
t"));

    ini->set_value(ini, "baidu", "hahaha");
    ini->save_to_file(ini, "write.conf");

    ini->remove(ini, "first");
```

```
    ini->save_to_file(ini, "write2.conf");

    ini->deletor(ini);
}

static void ini_parser_test_file()
{
    struct ini_parser * ini = new_ini_parser();
    int size = ini->parse_file(ini, "test.conf");

    assert( size > 0);
    assert( ini->value(ini, "second") == 0 );
    assert( ini->value(ini, "abc") == 0);
    assert( ini->value(ini, "name") != NULL );
    assert( ini->value(ini, "first") != NULL);

    printf("ini string: %s\n", g_szIniString);
    printf("key-value pairs count = %d\n", size);
    printf("key \'name\'', value = %s\n", ini->value(ini, "name"
));
    printf("key \'first\'', value = %s\n", ini->value(ini, "firs
t"));
    printf("key \'baidu\'', value = %s\n", ini->value(ini, "baid
u"));

    ini->deletor(ini);
}

void ini_parser_test()
{
    ini_parser_test_string();
    ini_parser_test_file();
}
```

struct ini_parser 已经运用在实际的项目中，目前为止没发现什么问题。

第2章 设计模式

C语言设计模式

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;
```

单例模式

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

单例模式要是用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;
```

单例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;
}
```

上面说了C++语言的编写方法，那C语言怎么写？其实也简单。大家也可以试一试。

```
typedef struct _DATA
{
    void* pData;
}DATA;

void* get_data()
{
    static DATA* pData = NULL;

   if(NULL != pData)
       return pData;

   pData = (DATA*)malloc(sizeof(DATA));
   assert(NULL != pData);
   return (void*)pData;
}
```

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。

```
class data
{
public:
    data () {}
    virtual ~data() {}
    virtual class data* copy() = 0;
};

class data_A : public data
{
public:
    data_A() {}
    ~data_A() {}
    class data* copy()
    {
        return new data_A();
    }
};

class data_B : public data
{
public:
    data_B() {}
    ~data_B() {}
    class data* copy()
    {
        return new data_B();
    }
};
```

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);
};
```

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?

```
typedef struct _NODE
{
    void* pData;
    struct _NODE* left;
    struct _NODE* right;
}NODE;
```

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[inde
x]);
   }
}
```

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);
}
```

模板模式

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。

```
class basic
{
public:
    void basic() {}
    virtual ~basic() {}
    virtual void step1() {}
    virtual void step2() {}
    void process()
    {
       step1();
       step2();
   }
};
```

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);
}
```

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;
}
```

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)();
    Apple* (*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;
}
```

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;
```

所以这个时候，我们首先需要设置额度和领导。

```
void set_account(struct _Leader* pLeader, int account)
{
    assert(NULL != pLeader);
    pLeader->account = account;
    return;
}

void set_next_leader(const struct _Leader* pLeader, struct _Lead
er* 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;
}
```

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，

```
typedef struct _Container
{
    int* pData;
    int size;
    int length;

    Interator* (*create_new_interator)(struct _Container* pConta
iner);
    int (*get_first)(struct _Container* pContainer);
    int (*get_last)(struct _Container* pContainer);

}Container;
```

我们看到，容器可以创建迭代器。那什么是迭代器呢？

```
typedef struct _Interator
{
    void* pVector;
    int index;

    int(* get_first)(struct _Interator* pInterator);
    int(* get_last)(struct _Interator* pInterator);
}Interator;
```

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。

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 _FoodSteet
{
    void (*eat)();
}FoodStreet;

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

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();
}
```

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？

```
typedef struct _PC_Client
{
    void (*request)();
}PC_Client;

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();
}
```

享元模式

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。

```
typedef struct _Font
{
    int type;
    int sequence;
    int gap;
    int lineDistance;

    void (*operate)(struct _Font* pFont);

}Font;
```

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, i
nt 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 s
equence, 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]->lineDista
nce)
            continue;

       return pFontFactory->ppFont[index];
   }

   pFont = (Font*)malloc(sizeof(Font));
   assert(NULL != pFont);
   pFont->type = type;
   pFont->sequence = sequence;
   pFont->gap = gap;
   pFont->lineDistance = lineDistance;

   if(pFontFactory-> number < pFontFactory->size)
   {
       pFontFactory->ppFont[index] = pFont;
       pFontFactory->number ++;
       return pFont;
   }
```

```
    ppFont = (Font**)malloc(sizeof(Font*) * pFontFactory->size * 2
);
    assert(NULL != ppFont);
    memmove(ppFont, pFontFacoty->ppFont, pFontFactory->size);
    free(pFontFactory->ppFont);
    pFontFactory->size *= 2;
    pFontFactory->number ++;
    ppFontFactory->ppFont = ppFont;
    return pFont;
}
```

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。

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 v220_to_v12
{
public:
    v220_to_v12() {}
    ~v220_to_v12() {}
    void voltage_transform_process() {}
};

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;

typedef struct _Adapter
{
    void* pAdaptee;
    void (*transform_process)(struct _Adapter* pAdapter);

}Adapter;
```

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;

typedef struct _CommMoviePlay
{
    HANDLE hFile;
   void (*play)(HANDLE hFile);

}CommMoviePlay;
```

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");
}
```

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，

```
void assemble_intel_cpu()
{
    printf("intel cpu!\n");
}

void assemble_samsung_memory()
{
    printf("samsung memory!\n");
}

void assemble_hitachi_harddisk()
{
    printf("hitachi harddisk!\n");
}
```

而对于一个希望配置AMD cpu, kingston内存、西部数据硬盘的朋友。他又该怎么做呢？

```
void assemble_amd_cpu()
{
    printf("amd cpu!\n");
}

void assemble_kingston_memory()
{
    printf("kingston memory!\n");
}

void assmeble_western_digital_harddisk()
{
    printf("western digital harddisk!\n");
}
```

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？

其实我们可以这么做，通常的产品还是按照共同的属性进行归类。

```
typedef struct _MeatDumpling
{
    void (*make)();
}MeatDumpling;

typedef struct _NormalDumpling
{
    void (*make)();
}NormalDumpling;
```

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)->mak
e();
    else
        return (NormalDumpling*)(pDumplingRequest->pDumpling)->m
ake();
}
```

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。

```
typedef struct _Object
{
    observer* pObserverList[MAX_BINDING_NUMBER];
    int number;

    void (*notify)(struct _Object* pObject);
    void (*add_observer)(observer* pObserver);
    void (*del_observer)(observer* pObserver);

}Object;
```

其实，我们需要定义的就是观察者本身了。就像我们前面说的一样，观察者可以是菜单、工具栏或者是子窗口等等。

```
typedef struct _Observer
{
    Object* pObject;

    void (*update)(struct _Observer* pObserver);
}Observer;
```

紧接着，我们要做的就是在Observer创建的时候，把observer自身绑定到Object上面。

```
void bind_observer_to_object(Observer* pObserver, Object* pObjec
t)
{
    assert(NULL != pObserver && NULL != pObject);

    pObserver->pObject = pObject;
    pObject->add_observer(pObserver);
}

void unbind_observer_from_object(Observer* pObserver, Object* pO
bject)
{
    assert(NULL != pObserver && NULL != pObject);

    pObject->del_observer(observer* pObserver);
    memset(pObserver, 0, sizeof(Observer));
}
```

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);
   }
}
```

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，

```
typedef struct _Organizer
{
    int number;
    Action* pActionHead;

    Action* (*create)();
    void (*restore)(struct _Organizer* pOrganizer);
}Organizer;
```

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;
}
```

解释器模式

解释器模式虽然听上去有些费解，但是如果用示例说明一下就不难理解了。我们知道在C语言中，关于变量的定义是这样的：一个不以数字开始的由字母、数字和下划线构成的字符串。这种形式的表达式可以用状态自动机解决，当然也可以用解释器的方式解决。

```
typedef struct _Interpret
{
    int type;
    void* (*process)(void* pData, int* type, int* result);

}Interpret;
```

上面的数据结构比较简单，但是很能说明问题。就拿变量来说吧，这里就可以定义成字母的解释器、数字解释器、下划线解释器三种形式。所以，我们可以进一步定义一下process的相关函数。

```
#define DIGITAL_TYPE 1
#define LETTER_TYPE  2
#define BOTTOM_LINE  3

void* digital_process(void* pData, int* type, int* result)
{
    UINT8* str;
    assert(NULL != pData && NULL != type && NULL != result);

   str = (UNT8*)pData;
   while (*str >= '0' && *str <= '9')
   {
       str ++;
   }

   if(*str == '\0')
   {
       *result = TRUE;
       return NULL;
```

```
    }

    if(*str == '_')
    {
        *result = TRUE;
        *type = BOTTOM_TYPE;
        return str;
    }

    if(*str >= 'a' && *str <= 'z' || *str >= 'A' && *str <= 'Z')

    {
        *result = TRUE;
        *type = LETTER_TYPE;
        return str;
    }

    *result = FALSE;
    return NULL;
}

void* letter_process(void* pData, int* type, int* result)
{
    UINT8* str;
    assert(NULL != pData && NULL != type && NULL != result);

    str = (UNT8*)pData;
    while (*str >= 'a' && *str <= 'z' || *str >= 'A' && *str <=
'Z')
    {
        str ++;
    }

    if(*str == '\0')
    {
        *result = TRUE;
        return NULL;
    }

    if(*str == '_')
```

```
    {
        *result = TRUE;
        *type = BOTTOM_TYPE;
        return str;
    }

    if(*str >= '0' && *str <= '9')
    {
        *result = TRUE;
        *type = DIGITAL_TYPE;
        return str;
    }

    *result = FALSE;
    return NULL;
}

void* bottom_process(void* pData, int* type, int* result)
{
    UINT8* str;
    assert(NULL != pData && NULL != type && NULL != result);

    str = (UNT8*)pData;
    while ('_' == *str )
    {
        str ++;
    }

    if(*str == '\0')
    {
        *result = TRUE;
        return NULL;
    }

    if(*str >= 'a' && *str <= 'z' || *str >= 'A' && *str <= 'Z')

    {
        *result = TRUE;
        *type = LETTER_TYPE;
        return str;
```

```
    }

    if(*str >= '0' && *str <= '9')
       {
           *result = TRUE;
           *type = DIGITAL_TYPE;
           return str;
       }

       *result = FALSE;
       return NULL;
   }
```

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？

```
typedef struct _Command
{
    void* pData;
    void (*exe)(struct _Command* pCommand);

}Command;

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;
}
```

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，

```
typedef struct _State
{
    void (*process)();
    struct _State* (*change_state)();

}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。

```
struct _State* change_state()
{
    State* pNextState = NULL;

    pNextState = (struct _State*)malloc(sizeof(struct _State));

    assert(NULL != pNextState);

    pNextState ->process = next_process;
    pNextState ->change_state = next_change_state;
   return pNextState;
}
```

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;
}
```

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* pVisito
r);
}Visitor;
```

就是这样一个豆腐，eat的时候就要做不同的判断了。

```
void eat(struct _Visitor* pVisitor, struct _Tofu* pTofu)
{
    assert(NULL != pVisitor && NULL != pTofu);

    pVisitor->process(pTofu, pVisitor);
}
```

既然eat的操作最后还是靠不同的visitor来处理了，那么下面就该定义process函数了。

```
void process(struct _Tofu* pTofu, struct _Visitor* pVisitor)
{
    assert(NULL != pTofu && NULL != pVisitor);

    if(pTofu->type == SPICY_FOOD && pVisitor->region == WEST ||

        pTofu->type == STRONG_SMELL_FOOD && pVisitor->region ==
EAST)
    {
        printf("I like this food!\n");
        return;
   }

   printf("I hate this food!\n");
}
```

面试的时候，设计模式会经常被问到。其实我们在写代码中或多或少会用到一些模式，面试官问你设计模式的问题，更多是看你有没有总结过。如果一直都是在那垒代码，你当然会认为这是个很难的问题。所以我们需要总结一下设计模式。

1. SINGLETON 单例模式

单例模式：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例单例模式。单例模式只应在有眞正的“单一实例”的需求时才可使用。

俺有6个漂亮的老婆，她们的老公都是我，我就是我们家里的老公Sigleton，她们只要说道“老公”，都是指的同一个人，那就是我(刚才做了个梦啦，哪有这么好的事)。

2. FACTORY METHOD 工厂方法模式

工厂方法模式：核心工厂类不再负责所有产品的创建，而是将具体创建的工作交给子类去做，成为一个抽象工厂角色，仅负责给出具体工厂类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

请MM去麦当劳吃汉堡，不同的MM有不同的口味，要每个都记住是一件烦人的事情，我一般采用Factory Method模式，带着MM到服务员那儿，说“要一个汉堡”，具体要什么样的汉堡呢，让MM直接跟服务员说就行了。

3. FACTORY 工厂模式

工厂模式：客户类和工厂类分开。消费者任何时候需要某种产品，只需向工厂请求即可。消费者无须修改就可以接纳新产品。缺点是当产品修改时，工厂类也要做相应的修改。如：如何创建及如何向客户端提供。

追MM少不了请吃饭了，麦当劳的鸡翅和肯德基的鸡翅都是MM爱吃的东西，虽然口味有所不同，但不管你带MM去麦当劳或肯德基，只管向服务员说“来四个鸡翅”就行了。麦当劳和肯德基就是生产鸡翅的Factory。

4. BUILDER 建造模式

建造模式：将产品的内部表象和产品的生成过程分割开来，从而使一个建造过程生成具有不同的内部表象的产品对象。建造模式使得产品内部表象可以独立的变化，客户不必知道产品内部组成的细节。建造模式可以强制实行一种分步骤进行的建造过程。

MM最爱听的就是“我爱你”这句话了，见到不同地方的MM，要能够用她们的方言跟她说这句话哦，我有一个多种语言翻译机，上面每种语言都有一个按键，见到MM我只要按对应的键，它就能够用相应的语言说出“我爱你”这句话了，国外的MM也可以轻松搞掂，这就是我的“我爱你”builder。（这一定比美军在伊拉克用的翻译机好卖）

5. PROTOTYPE 原型模式

原型模式允许动态的增加或减少产品类，产品类不需要非得有任何事先确定的等级结构，原始模型模式适用于任何的等级结构。缺点是每一个类都必须配备一个克隆方法。

跟MM用QQ聊天，一定要说些深情的话语了，我搜集了好多肉麻的情话，需要时只要copy出来放到QQ里面就行了，这就是我的情话prototype了。 原型模式：通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的方法创建出更多同类型的对象。

6. ADAPTER 适配器模式

适配器（变压器）模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起工作。适配类可以根据参数返还一个合适的实例给客户端。

在朋友聚会上碰到了一个美女Sarah，从香港来的，可我不会说粤语，她不会说普通话，只好求助于我的朋友kent了，他作为我和Sarah之间的Adapter，让我和Sarah可以相互交谈了(也不知道他会不会耍我)。

7. BRIDGE 桥梁模式

桥梁模式：将抽象化与实现化脱耦，使得二者可以独立的变化，也就是说将他们之间的强关联变成弱关联，也就是指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使两者可以独立的变化。

早上碰到MM，要说早上好，晚上碰到MM，要说晚上好；碰到MM穿了件新衣服，要说你的衣服好漂亮哦，碰到MM新做的发型，要说你的头发好漂亮哦。不要问我“早上碰到MM新做了个发型怎么说”这种问题，自己用BRIDGE组合一下不就行了。

8. COMPOSITE 合成模式

合成模式：合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式就是一个处理对象的树结构的模式。合成模式把部分与整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由他们复合而成的合成对象同等看待。

Mary今天过生日。“我过生日，你要送我一件礼物。”“嗯，好吧，去商店，你自己挑。”“这件T恤挺漂亮，买，这条裙子好看，买，这个包也不错，买。”“喂，买了三件了呀，我只答应送一件礼物的哦。”“什么呀，T恤加裙子加包包，正好配成一套呀，小姐，麻烦你包起来。”“……”，MM都会用Composite模式了，你会了没有？

9. DECORATOR 装饰模式

装饰模式：装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案，提供比继承更多的灵活性。动态给一个对象增加功能，这些功能可以再动态的撤消。增加由一些基本功能的排列组合而产生的非常大量的功能。

Mary过完轮到Sarly过生日，还是不要叫她自己挑了，不然这个月伙食费肯定玩完，拿出我去年在华山顶上照的照片，在背面写上“最好的的礼物，就是爱你的Fita”，再到街上礼品店买了个像框（卖礼品的MM也很漂亮哦），再找隔壁搞美术设计的Mike设计了一个漂亮的盒子装起来……，我们都是Decorator，最终都在修饰我这个人呀，怎么样，看懂了吗？

10. FACADE 门面（外观）模式

门面模式：外部与一个子系统的通信必须通过一个统一的门面对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。每一个子系统只有一个门面类，而且此门面类只有一个实例，也就是说它是一个单例模式。但整个系统可以有多个门面类。

我有一个专业的Nikon相机，我就喜欢自己手动调光圈、快门，这样照出来的照片才专业，但MM可不懂这些，教了半天也不会。幸好相机有Facade设计模式，把相机调整到自动档，只要对准目标按快门就行了，一切由相机自动调整，这样MM也可以用这个相机给我拍张照片了。

11. FLYWEIGHT 享元模式

享元模式：FLYWEIGHT在拳击比赛中指最轻量级。享元模式以共享的方式高效的支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态，它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来，将不可以共享的状态从类里剔除出去。客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。享元模式大幅度的降低内存中对象的数量。

每天跟MM发短信，手指都累死了，最近买了个新手机，可以把一些常用的句子存在手机里，要用的时候，直接拿出来，在前面加上MM的名字就可以发送了，再不用一个字一个字敲了。共享的句子就是Flyweight，MM的名字就是提取出来的外部特征，根据上下文情况使用。

12. PROXY 代理模式

代理模式：代理模式给某一个对象提供一个代理对象，并由代理对象控制对源对象的引用。代理就是一个人或一个机构代表另一个人或者一个机构采取行动。某些情况下，客户不想或者不能够直接引用一个对象，代理对象可以在客户和目标对象直接起到中介的作用。客户端分辨不出代理主题对象与真实主题对象。代理模式可以并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口，这时候代理对象不能够创建被代理对象，被代理对象必须有系统的其他角色代为创建并传入。

跟MM在网上聊天，一开头总是“hi,你好”,“你从哪儿来呀？”“你多大了？”“身高多少呀？”这些话，真烦人，写个程序做为我的Proxy吧，凡是接收到这些话都设置好了自己的回答，接收到其他的话时再通知我回答，怎么样，酷吧。

13. CHAIN OF RESPONSIBLEITY 责任链模式

责任链模式：在责任链模式中，很多对象由每一个对象对其下家的引用而接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。客户并不知道链上的哪一个对象最终处理这个请求，系统可以在不影响客户端的情况下动态的重新组织链和分配责任。处理者有两个选择：承担责任或者把责任推给下家。一个请求可以最终不被任何接收端对象所接受。

晚上去上英语课，为了好开溜坐到了最后一排，哇，前面坐了好几个漂亮的MM哎，找张纸条，写上“Hi,可以做我的女朋友吗？如果不愿意请向前传”，纸条就一个接一个的传上去了，糟糕，传到第一排的MM把纸条传给老师了，听说是个老处女呀，快跑!

14. COMMAND 命令模式

命令模式：命令模式把一个请求或者操作封装到一个对象中。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。命令模式允许请求的一方和发送的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否执行，何时被执行以及是怎么被执行的。系统支持命令的撤消。

俺有一个MM家里管得特别严，没法见面，只好借助于她弟弟在我们俩之间传送信息，她对我有什么指示，就写一张纸条让她弟弟带给我。这不，她弟弟又传送过来一个COMMAND，为了感谢他，我请他吃了碗杂酱面，哪知道他说：“我同时给我姐姐三个男朋友送COMMAND，就数你最小气，才请我吃面。”

15. INTERPRETER 解释器模式

解释器模式：给定一个语言后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。解释器模式将描述怎样在有了一个简单的文法后，使用模式设计解释这些语句。在解释器模式里面提到的语言是指任何解释器对象能够解释的任何组合。在解释器模式中需要定义一个代表文法的命令类的等级结构，也就是一系列的组合规则。每一个命令对象都有一个解释方法，代表对命令对象的解释。命令对象的等级结构中的对象的任何排列组合都是一个语言。

俺有一个《泡MM真经》，上面有各种泡MM的攻略，比如说去吃西餐的步骤、去看电影的方法等等，跟MM约会时，只要做一个Interpreter，照着上面的脚本执行就可以了。

16. ITERATOR 迭代子模式

迭代子模式：迭代子模式可以顺序访问一个聚集中的元素而不必暴露聚集的内部表象。多个对象聚在一起形成的总体称之为聚集，聚集对象是能够包容一组对象的容器对象。迭代子模式将迭代逻辑封装到一个独立的子对象中，从而与聚集本身隔开。迭代子模式简化了聚集的界面。每一个聚集对象都可以有一个或一个以上的迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。迭代算法可以独立于聚集角色变化。

我爱上了Mary，不顾一切的向她求婚。Mary：“想要我跟你结婚，得答应我的条件”我：“什么条件我都答应，你说吧” Mary：“我看上了那个一克拉的钻石” 我：“我买，我买，还有吗？” Mary：“我看上了湖边的那栋别墅” 我：“我买，我买，还有吗？” Mary：“我看上那辆法拉利跑车” 我脑袋嗡的一声，坐在椅子上，一咬牙：“我买，我买，还有吗？” ……

17. MEDIATOR 调停者模式

调停者模式：调停者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使他们可以松散偶合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立的变化。调停者模式将多对多的相互作用转化为一对多的相互作用。调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

四个MM打麻将，相互之间谁应该给谁多少钱算不清楚了，幸亏当时我在旁边，按照各自的筹码数算钱，赚了钱的从我这里拿，赔了钱的也付给我，一切就OK啦，俺得到了四个MM的电话。

18. MEMENTO 备忘录模式

备忘录模式：备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捉住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

同时跟几个MM聊天时，一定要记清楚刚才跟MM说了些什么话，不然MM发现了会不高兴的哦，幸亏我有个备忘录，刚才与哪个MM说了什么话我都拷贝一份放到备忘录里面保存，这样可以随时察看以前的记录啦。

19. OBSERVER 观察者模式

观察者模式：观察者模式定义了一种一队多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使他们能够自动更新自己。

想知道咱们公司最新MM情报吗？加入公司的MM情报邮件组就行了，tom负责搜集情报，他发现的新情报不用一个一个通知我们，直接发布给邮件组，我们作为订阅者（观察者）就可以及时收到情报啦。

20. STATE 状态模式

状态模式：状态模式允许一个对象在其内部状态改变的时候改变行为。这个对象看上去象是改变了它的类一样。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式需要对每一个系统可能取得的状态创立一个状态类的子类。当系统的状态变化时，系统便改变所选的子类。

跟MM交往时，一定要注意她的状态哦，在不同的状态时她的行为会有不同，比如你约她今天晚上去看电影，对你没兴趣的MM就会说“有事情啦”，对你不讨厌但还没喜欢上的MM就会说“好啊，不过可以带上我同事么？”，已经喜欢上你的MM就会说“几点钟？看完电影再去泡吧怎么样？”，当然你看电影过程中表现良好的话，也可以把MM的状态从不讨厌不喜欢变成喜欢哦。

21. STRATEGY 策略模式

策略模式：策略模式针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。策略模把行为和环境分开。环境类负责维持和查询行为类，各种算法在具体的策略类中提供。由于算法和环境独立开来，算法的增减，修改都不会影响到环境和客户端。

跟不同类型的MM约会，要用不同的策略，有的请电影比较好，有的则去吃小吃效果不错，有的去海边浪漫最合适，单目的都是为了得到MM的芳心，我的追MM锦囊中有好多Strategy哦。

22. TEMPLATE METHOD 模板模式

模板方法模式：模板方法模式准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。先制定一个顶级逻辑框架，而将逻辑的细节留给具体的子类去实现。

看过《如何说服女生上床》这部经典文章吗？女生从认识到上床的不变的步骤分为巧遇、打破僵局、展开追求、接吻、前戏、动手、爱抚、进去八大步骤(Template method)，但每个步骤针对不同的情况，都有不一样的做法，这就要看你随机应变啦(具体实现)。

23. VISITOR 访问者模式

访问者模式：访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由的演化。访问者模式使得增加新的操作变的很容易，就是增加一个新的访问者类。访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。当使用访问者模式时，要将尽可能多的对象浏览逻辑放在访问者类中，而不是放到它的子类中。访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。

情人节到了，要给每个MM送一束鲜花和一张卡片，可是每个MM送的花都要针对她个人的特点，每张卡片也要根据个人的特点来挑，我一个人哪搞得清楚，还是找花店老板和礼品店老板做一下Visitor，让花店老板根据MM的特点选一束花，让礼品店老板也根据每个人特点选一张卡，这样就轻松多了。

第3章 C++语言设计模式

什么是设计模式？

设计模式是某类软件设计问题的经典解决方案，它可以提高代码的可重用性，增强系统的可维护性。将设计模式引入软件设计和开发过程，可以充分利用已有的开发经验指导新的软件设计工作。

设计模式的分类

创建型模式：是类实例化时使用的模式，规定了创建对象的规则和方法，工厂模式、单例模式都属于本模式。

结构型模式：从程序结构上解决模块之间的耦合性问题，代理模式属于本模式。

行为型模式：描述了算法和对象间的职责分配，迭代器模式属于本模式。

- 观察者模式
- 备忘录模式
- 解析器模式
- 命令模式
- 状态模式
- 访问者模式

1. 设计模式

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。

模式的经典定义：每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心，通过这种方式，我们可以无数次地重用那些已有的解决方案，无需再重复相同的工作。即模式是在特定环境中解决问题的一种方案

2. 设计模式 目的

其目的就是一方面教你如何利用真实可靠的设计来组织代码的模板。 简单地说，就是从前辈们在程序设计过程中总结、抽象出来的通用优秀经验。主要目的一方面是为了增加程序的灵活性、可重用性。 另一方面也有助于程序设计的标准化和提高系统开发进度。

也有人忠告：不要过于注重程序的“设计模式”。 有时候，写一个简单的算法，要比引入某种模式更容易。在多数情况下，程序代码应是简单易懂，甚至清洁工也能看懂。不过呢在大项目或者框架中，没有设计模式来组织代码，别人是不易理解的。

一个软件设计模型也仅仅只是一个引导。它必须根据程序设计语言和你的应用程序的特点和要求而特别的设计。

3. 设计模式历史

设计模式”这个术语最初被设计用于建筑学领域。Christopher Alexander 在他1977的著作“A Pattern Language :Towns/Building/Construction”里面描述了一些常见的建筑学设计问题，并解释了如何用这些已有的，著名的模式集合来开始全新 的有效的设计。Alexander的观点被很好的转化到软件开发上来，并且长期的合意的用原有的组件来构造新的解决方案。

4. 设计模式的四个基本要素

设计模式使人们可以更加简单方便地复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。

所有的设计模式都有一些常用的特性：一个标识（a pattern name），一个问题陈述（a problem statement）和一个解决方案(a solution)，效果(consequences)

模式名称（**pattern name**）：描述模式的问题、解决方案和效果

一个设计模式的标识（模式名称）是重要的，因为它会让其他的程序员不用进行太深入的学习就能立刻理解你的代码的目的（至少通过这个标识程序员会很熟悉这个模式）。没有这个模式名，我们便无法与其他人交流设计思想及设计结果。

问题**(problem)** ：描述是用来说明这个模式的应用的领域。

描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

解决方案**(solution)** ： 描述了这个模型的执行。

描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

效果**(consequences)**

描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。一个好的设计模式的论述应该覆盖使用这个模型的优点和缺点。

一个模式是解决特定问题的有效方法。一个设计模式不是一个库（能在你的项目中直接包含和使用的代码库）而是一个用来组织你的代码的模板（Java bean）。事实上，一个代码库和一个设计模式在应用上是有很多不同的。

比如，你从店铺里面买的一件衬衫是一个代码库，它的颜色，样式和大小都由设计师和厂商决定，但它满足了你的需求。 然而，如果店里面没有什么衣服适合你，那你就能自己创建自己的衬衫（设计它的形状，选择布料，然后裁缝在一起）。但是

如果你不是一个裁缝，你可能会发现自 己很容易的去找一个合适的模式然后按着这个模式去设计自己的衬衫。使用一个模型，你可以在更少的时间内得到一个熟练设计的衬衫。

回到讨论软件上来，一个数据提取层或者一个CMS（content management system）就是一个库——它是先前设计好而且已经编码好了的，如果它能准确的满足你的需要那它就是一个好的选择。但如果你正在读这本书《设计模式》，可能你会发现 库存的（原有的）解决方案并不是总是对你有效。至今你知道什么是你所要的，而且你能够实现它，你仅仅需要一个模型来引导你。

最后一个想法：就象一个裁缝模型，一个设计本身而言是没有什么用处的。毕竟，你不可能穿一个服装模型——它仅仅是由很薄的纸拼凑起来的。类似的，一个软件设计模型也仅仅只是一个引导。它必须根据程序设计语言和你的应用程序的特点和要求而特别的设计。

3. 设计模式分类

1）根据其目的（模式是用来做什么的）可分为创建型(Creational)，结构型(Structural)和行为型(Behavioral)三种：

- 创建型模式主要用于创建对象。
- 结构型模式主要用于处理类或对象的组合。
- 行为型模式主要用于描述对类或对象怎样交互和怎样分配职责。

2）根据范围，即模式主要是用于处理类之间关系还是处理对象之间的关系，可分为类模式和对象模式两种：

- 类模式：处理类和子类之间的关系，这些关系通过继承建立，在编译时刻就被确定下来，是属于静态的。
- 对象模式：处理对象间的关系，这些关系在运行时刻变化，更具动态性。

范围\目的	创建型模式	结构型模式	行为型模式
类模式	工厂方法模式	（类）适配器模式	解释器模式 模板方法模式
对象模式	抽象工厂模式 建造者模式 原型模式 单例模式	（对象）适配器模式 桥接模式 组合模式 装饰模式 外观模式 享元模式 代理模式	职责链模式 命令模式 迭代器模式 中介者模式 备忘录模式 观察者模式 状态模式 策略模式 访问者模式

4. 一些基本的设计模式

- Abstract Factory 抽象工厂模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
- Adapter适配器模式：将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
- Bridge桥接模式：将抽象部分与它的实现部分分离，使它们都可以独立地变化。
- Builder建造者模式：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。
- Chain of Responsibility 职责链：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它。
- Command命令模式：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作。
- Composite 组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构。它使得客户对单个对象和复合对象的使用具有一致性。

- Decorator 装饰器：动态地给一个对象添加一些额外的职责。就扩展功能而言，它比生成子类方式更为灵活。
- Facade外观模式：为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
- Factory Method 工厂方法：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。Factory Method使一个类的实例化延迟到其子类。
- Flyweight享元模式：运用共享技术有效地支持大量细粒度的对象。
- Interpreter模式：给定一个语言, 定义它的文法的一种表示，并定义一个解释器, 该解释器使用该表示来解释语言中的句子。
- Iterator 迭代器：提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。
- Mediator 中介者：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。
- Memento备忘录模式：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到保存的状态。
- Observer观察者模式：定义对象间的一种一对多的依赖关系,以便当一个对象的状态发生改变时,所有依赖于它的对象都得到通知并自动刷新。
- Prototype原型模式：用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。
- Proxy 代理模式：为其他对象提供一个代理以控制对这个对象的访问。
- Singleton单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
- State 状态：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它所属的类。
- Strategy 策略模式：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法的变化可独立于使用它的客户。

- Template Method 模板方法：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。
- Visitor 访问者模式：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

5. 设计模式六大原则

1）设计模式的核心原则是**:**"开-闭"原则**(Open - ClosedPrinciple** 缩写**:OCP)**：对扩展开放，对修改关闭

意思是,在一个系统中,对于扩展是开放的,对于修改是关闭的,一个好的系统是在不修改源代码的情况下,可以扩展你的功能..而实现开闭原则的关键就是抽象化.

通过扩展已有软件系统，可以提供新的行为，以满足对软件的新的需求，使变化中的软件有一定的适应性和灵活性。已有软件模块，特别是最重要的抽象层模块不能再修改，这使变化中的软件系统有一定的稳定性和延续性。

在"开-闭"原则中,不允许修改的是抽象的类或者接口,允许扩展的是具体的实现类,抽象类和接口在"开-闭"原则中扮演着极其重要的角色..即要预知可能变化的需求.又预见所有可能已知的扩展..所以在这里"抽象化"是关键!!!

可变性的封闭原则:找到系统的可变因素,将它封装起来..这是对"开-闭"原则最好的实现..不要把你的可变因素放在多个类中,或者散落在程序的各个角落..你应该将可变的因素,封套起来..并且切忌不要把所用的可变因素封套在一起..最好的解决办法是,分块封套你的可变因素!!避免超大类,超长类,超长方法的出现!!给你的程序增加艺术气息,将程序艺术化是我们的目标!!

2）里氏代换原则**:**任何基类可以出现的地方**,**子类也可以出现

Liskov Substitution Principle（里氏代换原则）：子类能够必须能够替换基类能够从出现的地方。子类也能在基类 的基础上新增行为。这yi讲的是基类和子类的关系，只有这种关系存在时，里氏代换原则才存在。正方形是长方形是理解里氏代换原则的经典例子。

3）依赖倒转原则**:**：要依赖抽象**,**而不要依赖具体的实现**.**

依赖倒置（Dependence Inversion Principle）原则讲的是：要依赖于抽象，不要依赖于具体。简单的说，依赖倒置原则要求客户端依赖于抽象耦合。原则表述：

（1）抽象不应当依赖于细节；细节应当依赖于抽象；（2）要针对接口编程，不针对实现编程。

如果说开闭原则是目标,依赖倒转原则是到达"开闭"原则的手段..如果要达到最好的"开闭"原则,就要尽量的遵守依赖倒转原则..可以说依赖倒转原则是对"抽象化"的最好规范!!我个人感觉,依赖倒转原则也是里氏代换原则的补充..你理解了里氏代换原则,再来理解依赖倒转原则应该是很容易的..

4）合成/聚合复用原则（**CARP**）**:**要尽量使用合成/聚合原则**,**而不是继承关系达到软件复用的目的

合成/聚合复用原则（Composite/Aggregate ReusePrinciple或CARP）经常又叫做合成复用原则（Composite ReusePrinciple或CRP），就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；新对象通过向这些对象的委派达到复用已有功能的目的。简而言之，要尽量使用合成/聚合，尽量不要使用继承。

什么是合成**?**

合成:是指一个整体对依托他而存在的关系,例如:一个人对他的房子和家具,其中他的房子和家具是不能被共享的,因为那些东西都是他自己的..并且人没了,这个也关系就没了..这个例子就好像,乌鸡百凤丸这个产品,它是有乌鸡和上等药材合成而来的一样..也比如网络游戏中的武器装备合成一样,多种东西合并为一种超强的东西一样

什么是聚合**?**

聚合:聚合是比合成关系的一种更强的依赖关系,聚合是一个整体对个体的部分,例如,一个奔驰S360汽车,对奔驰S360引擎,奔驰S360轮胎的关系..这些关系就是带有聚合性质的..因为奔驰S360引擎和奔驰S360轮胎他们只能被奔驰S360汽车所用,离开了奔驰S360汽车,它们就失去了存在的意义..在我们的设计中,这样的关系不应该频繁出现..这样会增大设计的耦合度..

明白了合成和聚合关系,再来理解合成/聚合原则应该就清楚了..要避免在系统设计中出现,一个类的继承层次超过3次..如果这样的话,可以考虑重构你的代码,或者重新设计结构..当然最好的办法就是考虑使用合成/聚合原则

5）迪米特法则**:**系统中的类**,**尽量不要与其他类互相作用**,**减少类之间的耦合度

迪米特法则（Law of Demeter或简写LoD）又叫最少知识原则（Least Knowledge Principle或简写为LKP），也就是说，一个对象应当对其它对象有尽可能少的了解。

其它表述：只与你直接的朋友们通信，不要跟"陌生人"说话。一个类应该对自己需要耦合或调用的类知道得最少，你（被耦合或调用的类）的内部是如何复杂都和我没关系，那是你的事情，我就知道你提供的public方法，我就调用这么多，其他的一概不关心。

迪米特法则与设计模式Facade模式、Mediator模式使民无知

系统中的类,尽量不要与其他类互相作用,减少类之间的耦合度,因为在你的系统中,扩展的时候,你可能需要修改这些类,而类与类之间的关系,决定了修改的复杂度,相互作用越多,则修改难度就越大,反之,如果相互作用的越小,则修改起来的难度就越小..例如A类依赖B类,则B类依赖C类,当你在修改A类的时候,你要考虑B类是否会受到影响,而B类的影响是否又会影响到C类..如果此时C类再依赖D类的话,呵呵,我想这样的修改有的受了..

6）接口隔离法则**:**这个法则与迪米特法则是相通的

接口隔离原则（Interface Segregation Principle）讲的是：使用多个专门的接口比使用单一的总接口总要好。换而言之，从一个客户类的角度来讲：一个类对另外一个类的依赖性应当是建立在最小接口上的。

过于臃肿的接口是对接口的污染。不应该强迫客户依赖于它们不用的方法。

迪米特法则是目的,而接口隔离法则是对迪米特法则的规范..为了做到尽可能小的耦合性,我们需要使用接口来规范类,用接口来约束类.要达到迪米特法则的要求,最好就是实现接口隔离法则,实现接口隔离法则,你也就满足了迪米特法则...

6. 总结

设计模式是从许多优秀的软件系统中总结出的成功的、能够实现可维护性复用的设计方案，使用这些方案将避免我们做一些重复性的工作，而且可以设计出高质量的软件系统。设计模式的主要优点如下：

1）设计模式融合了众多专家的经验，并以一种标准的形式供广大开发人员所用，它提供了一套通用的设计词汇和一种通用的语言以方便开发人员之间沟通和交流，使得设计方案更加通俗易懂。对于使用不同编程语言的开发和设计人员可以通过设计模式来交流系统设计方案，每一个模式都对应一个标准的解决方案，设计模式可以降低开发人员理解系统的复杂度。

2）设计模式使人们可以更加简单方便地复用成功的设计和体系结构，将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式使得重用成功的设计更加容易，并避免那些导致不可重用的设计方案。3）设计模式使得设计方案更加灵活，且易于修改。

4）设计模式的使用将提高软件系统的开发效率和软件质量，且在一定程度上节约设计成本。

5）设计模式有助于初学者更深入地理解面向对象思想，一方面可以帮助初学者更加方便地阅读和学习现有类库与其他系统中的源代码，另一方面还可以提高软件的设计水平和代码质量。设计模式不是学出来的，是用出来的。为了学习设计模式而学习，效果可能不是很好。一般框架都会使用设计模式。如PHP 的ZF用来很多设计模式，框架里面的类名或者目录名，都以某种设计模式的名称命名，这样大家一看到这个类名或者文件名，就知道它的代码组织结构了。如果精通了语言,剩下的编码自然是很简单,随着编码经验积累,对设计模式和原则的理解也就越透彻，其过程就是山穷水复疑无路，而结果柳暗花明又一村。另外需要注意，熟练模式后，切勿因模式二去模式。如果像著名数学家华罗庚谈到读书的三个境界所说，“读书是由薄到厚，再由厚到薄的过程”。说明你练到家了。

优秀程序设计的**18**大原则

良好的编程原则与良好的设计工程原则密切相关。本文总结的这些设计原则，帮助开发者更有效率的编写代码，并帮助成为一名优秀的程序员。作者Diggins是加拿大一位有25年编程经验的资深技术人员，曾效力于Microsoft和Autodesk，并创办过两家赢利的互联网公司。

1.避免重复原则(DRY - Don’t repeat yourself)

编程的最基本原则是避免重复。在程序代码中总会有很多结构体，如循环、函数、类等等。一旦你重复某个语句或概念，就会很容易形成一个抽象体。

2.抽象原则(Abstraction Principle)

与DRY原则相关。要记住，程序代码中每一个重要的功能，只能出现在源代码的一个位置。

3.简单原则(Keep It Simple and Stupid)

简单是软件设计的目标，简单的代码占用时间少，漏洞少，并且易于修改。

4.避免创建你不要的代码 Avoid Creating a YAGNI (You aren’t going to need it)

除非你需要它，否则别创建新功能。

5.尽可能做可运行的最简单的事(Do the simplest thing that could possibly work)

尽可能做可运行的最简单的事。在编程中，一定要保持简单原则。作为一名程序员不断的反思“如何在工作中做到简化呢?”这将有助于在设计中保持简单的路径。

6.别让我思考(Don’t make me think)

这是Steve Krug一本书的标题，同时也和编程有关。所编写的代码一定要易于读易于理解，这样别人才会欣赏，也能够给你提出合理化的建议。相反，若是繁杂难解的程序，其他人总是会避而远之的。

7.开闭原则(Open/Closed Principle)

你所编写的软件实体(类、模块、函数等)最好是开源的，这样别人可以拓展开发。不过，对于你的代码，得限定别人不得修改。换句话说，别人可以基于你的代码进行拓展编写，但却不能修改你的代码。

8.代码维护(Write Code for the Maintainer)

一个优秀的代码，应当使本人或是他人在将来都能够对它继续编写或维护。代码维护时，或许本人会比较容易，但对他人却比较麻烦。因此你写的代码要尽可能保证他人能够容易维护。用书中原话说“如果一个维护者不再继续维护你的代码，很可能他就有想杀了你的冲动。”

9.最小惊讶原则(Principle of least astonishment)

最小惊讶原则通常是在用户界面方面引用，但同样适用于编写的代码。代码应该尽可能减少让读者惊喜。也就是说，你编写的代码只需按照项目的要求来编写。其他华丽的功能就不必了，以免弄巧成拙。

10.单一责任原则(Single Responsibility Principle)

某个代码的功能，应该保证只有单一的明确的执行任务。

11.低耦合原则(Minimize Coupling)

代码的任何一个部分应该减少对其他区域代码的依赖关系。尽量不要使用共享参数。低耦合往往是完美结构系统和优秀设计的标志。

12.最大限度凝聚原则(Maximize Cohesion)

相似的功能代码应尽量放在一个部分。

13.隐藏实现细节(Hide Implementation Details)

隐藏实现细节原则，当其他功能部分发生变化时，能够尽可能降低对其他组件的影响。

14.迪米特法则又叫作最少知识原则(Law of Demeter)

该代码只和与其有直接关系的部分连接。(比如：该部分继承的类，包含的对象，参数传递的对象等)。

15.避免过早优化(Avoid Premature Optimization)

除非你的代码运行的比你想像中的要慢，否则别去优化。假如你真的想优化，就必须先想好如何用数据证明，它的速度变快了。

“过早的优化是一切罪恶的根源”——Donald Knuth

16.代码重用原则(Code Reuse is Good)

重用代码能提高代码的可读性，缩短开发时间。

17.关注点分离(Separation of Concerns)

不同领域的功能，应该由不同的代码和最小重迭的模块组成。

18.拥抱改变(Embrace Change)

这是Kent Beck一本书的标题，同时也被认为是极限编程和敏捷方法的宗旨。

许多其他原则都是基于这个概念的，即你应该积极面对变化。事实上，一些较老的编程原则如最小化耦合原则都是为了使代码能够容易变化。无论你是否是个极限编程者，基于这个原则去编写代码会让你的工作变得更有意义。

单例模式

单例模式也称为单件模式，其意图是保证一个类仅有一个实例并提供一个访问它的全局访问点，该实例被所有程序模块共享。

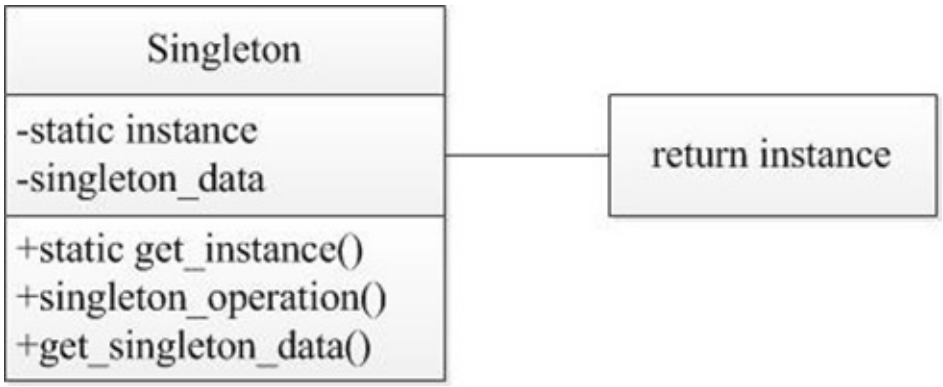
单例模式是一种对象创建型模式，使用单例模式，可以保证为一个类只生成唯一的实例对象。也就是说，在整个程序空间中，该类只存在一个实例对象。

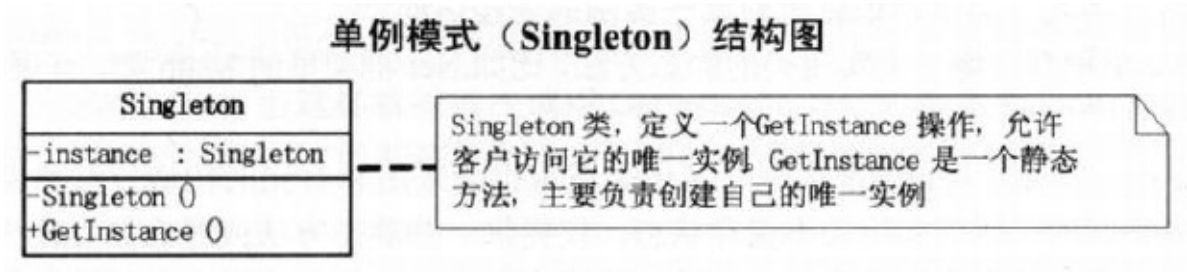
GoF对单例模式的定义是：保证一个类、只有一个实例存在，同时提供能对该实例加以访问的全局访问方法。

单例模式：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例单例模式 。单例模式只应在有眞正的“单一实例”的需求时才可使用。

俺有6个漂亮的老婆，她们的老公都是我，我就是我们家里的老公Sigleton，她们只要说道“老公”，都是指的同一个人，那就是我(刚才做了个梦啦，哪有这么好的事)。

类图角色和职责





为什么使用单例模式

在应用系统开发中，我们常常有以下需求：

- 单例模式有很多的应用场景，比如Windows中的任务管理器、网站的计数器等。
- 在多个线程之间，比如初始化一次socket资源；比如servlet环境，共享同一个资源或者操作同一个对象
- 在整个程序空间使用全局变量，共享资源
- 大规模系统中，为了性能的考虑，需要节省对象的创建时间等等。

因为Singleton模式可以保证为一个类只生成唯一的实例对象，所以这些情况，Singleton模式就派上用场了。

饿汉式

```
#include <iostream>
using namespace std;

class Singelton
{
private:
    Singelton()
    {
        cout << "Singelton 构造函数执行" << endl;
    }
public:
    static Singelton *getInstance()
    {
        return m_psl;
    }

    static void FreeInstance()
    {
        if (m_psl != NULL)
        {
            delete m_psl;
            m_psl = NULL;
        }
    }
private:
```

```
    static Singelton *m_psl;
};

Singelton *Singelton::m_psl = new Singelton;

void main2()
{
    printf("\n");
    Singelton *p1 = Singelton::getInstance();
    Singelton *p2 = Singelton::getInstance();

    if (p1 == p2)
    {
        cout << "是同一个对象" << endl;
    }
    else
    {
        cout << "不是同一个对象" << endl;
    }
    Singelton::FreeInstance();

    return ;
}

void main()
{
    main2();
    system("pause");
}
```

懒汉式

```
#include <iostream>
using namespace std;

class Singelton
{
private:
```

```
    Singelton()
    {
        cout << "Singelton 构造函数执行" << endl;
    }
public:
    static Singelton *getInstance()
    {
        if (m_psl == NULL)
        {
            m_psl = new Singelton;
        }
        return m_psl;
    }

    static void FreeInstance()
    {
        if (m_psl != NULL)
        {
            delete m_psl;
            m_psl = NULL;
        }
    }

private:
    static Singelton *m_psl;
};

Singelton *Singelton::m_psl = NULL;

void main2()
{
    Singelton *p1 = Singelton::getInstance();
    Singelton *p2 = Singelton::getInstance();

    if (p1 == p2)
    {
        cout << "是同一个对象" << endl;
    }
    else
    {
```

```
        cout << "不是同一个对象" << endl;
    }
    Singelton::FreeInstance();

    return ;
}

void main()
{
    main2();
    system("pause");
}
```

懒汉式多线程问题

- 懒汉"模式虽然有优点，但是每次调用GetInstance()静态方法时，必须判断NULL == m_instance，使程序相对开销增大。
- 多线程中会导致多个实例的产生，从而导致运行代码不正确以及内存的泄露。
- 提供释放资源的函数

这是因为C++中构造函数并不是线程安全的。C++中的构造函数简单来说分两步：

- 第一步：内存分配
- 第二步：初始化成员变量

由于多线程的关系，可能当我们在分配内存好了以后，还没来得急初始化成员变量，就进行线程切换，另外一个线程拿到所有权后，由于内存已经分配了，但是变量初始化还 没进行，因此打印成员变量的相关值会发生不一致现象。

```
#include "stdafx.h"
#include "windows.h"
#include "winbase.h"
#include <process.h>
#include "iostream"
using namespace std;

// 构造函数不是线程安全函数
class Singelton
{
```

```
private:
    Singelton()
    {
        cout<<"Singelton构造函数begin"<<endl;
        Sleep(1000);
        cout<<"Singelton构造函数end"<<endl;
    }

public:
    static Singelton *getSingelton()
    {
        if (single == NULL) //需要判断
        {
            count ++;
            single = new Singelton();
        }
        return single;
    }

    static Singelton *releaseSingelton()
    {
        if (single != NULL) //需要判断
        {
            delete single;
            single = NULL;
        }
        return single;
    }
    static void pirntS() //测试函数
    {
        cout<<"Singelton printS test"<<endl;
    }

private:
    static Singelton *single;
    static int count ;
};

//note 静态变量类外初始化  //懒汉式
Singelton *Singelton::single = NULL;
```

```
int Singelton::count = 0;

void main2()
{
    Singelton *s1 = Singelton::getSingelton();
    Singelton *s2 = Singelton::getSingelton();
    if (s1 == s2)
    {
        cout<<"ok....equal"<<endl;
    }
    else
    {
        cout<<"not.equal"<<endl;
    }
    cout <<"hello...."<<endl;
    system("pause");
}

void MyThreadFunc (void *)
{
    //cout << "我是线程体 ...." << endl;
    cout << "我是线程体 ....\n";
    Singelton::getSingelton()->pirntS();
}

int _tmain(int argc, _TCHAR* argv[])
{
    //main2();
    HANDLE hThread[10];

    for (int i=0; i<3; i++)
    {
        hThread[i] = (HANDLE)_beginthread (MyThreadFunc, 0, NULL
);
    }

    for (int i=0; i<3; i++)
    {
        WaitForSingleObject( hThread[i], INFINITE );
    }
```

```
    cout << "hello" << endl;
    system("pause");
    return 0;
}
```

懒汉式线程同步

示例代码

```
#include <iostream>
using namespace std;

class Singleton{                        //定义Singleton类
public:
    static Singleton *get_instance();//定义静态成员函数
    int get_var();
    void set_var(int);
private:
    Singleton();                    //声明构造函数为私有成员
    virtual ~Singleton();           //声明虚析构函数
    static Singleton *instance; //定义静态数据成员
    int var;
};
Singleton::Singleton()  //定义构造函数
{
    var = 20;
    cout << "Singleton constructor!" << endl;
}
Singleton::~Singleton() //定义析构函数
{
    delete instance;
}

Singleton *Singleton::instance = NULL; //对静态数据成员进行初始化
//定义静态成员函数get_instance()，该函数操作静态数据成员instance
//第一次调用本函数时，为instance提供有效值并一直保留到程序运行完毕
```

```
Singleton *Singleton::get_instance()
{
    if (NULL == instance){
        //本函数第一次调用时，创建一个Singleton对象，对象地址由instance
记录
        //该对象将一直存在直到程序结束
        instance = new Singleton();
    }
    return instance;
}
int Singleton::get_var()        //定义get_var()成员函数
{
    return var;
}
void Singleton::set_var(int n) //定义set_var()成员函数
{
    var = n;
}
int main()
{
    //定义两个Singleton*变量，并通过get_instance()函数为其初始化
    //第一次调用get_instance()时创建对象，对象地址记录在p1中
    Singleton *p1 = Singleton::get_instance();
    //第二次调用get_instance()时，不再创建对象，p2与p1记录同一个对象地址
    Singleton *p2 = Singleton::get_instance();

    cout << "p1 var = " << p1->get_var() << endl;
    cout << "p2 var = " << p2->get_var() << endl;
    p1->set_var(5);
    cout << "p1 var = " << p1->get_var() << endl;
    cout << "p2 var = " << p2->get_var() << endl;

    system("pause");
    return 0;
}
```

```
#include <iostream>
#include <string>
using namespace std;
```

```
// #define  public private

class
{
public:
protected:
private:
}a1;

class Singleton
{
private:
    int i;
    static Singleton *instance;
    Singleton(int i)
    {
        this->i = i;
    }
public:
    static Singleton *getInstance()
    {
        return instance;
    }
    void show()
    {
        cout << i << endl;
    }
};
Singleton* Singleton::instance = new Singleton(8899);

class A :public Singleton
{

};

int mainJ()
{
    Singleton *s = Singleton::getInstance();
    Singleton *s2 = A::getInstance();
    cout << (s == s2) << endl;
```

```
    cin.get();
    return 0;
}
```

总结

在很多人印象中，单例模式可能是23个设计模式中最简单的一个。如果不考虑多线程，的确如此，但是一旦要在多线程中运用，那么从我们的教程中可以了解到，它涉及到很多编译器，多线程，C++语言标准等方面的内容。本专题参考的资料如下：

- C++ Primer (Stanley B.Lippman),主要参考的是模板静态变量的初始化以及实例化。
- MSDN,有关线程同步interlocked相关的知识。
- Effective C++ 04条款(Scott Meyers) Non-Local-Static对象初始化顺序以及Meyers单例模式的实现。
- Double-Checked Locking,Threads,Compiler Optimizations,and More（Scott Meyers），解释了由于编译器的优化，导致auto_ptr.reset函数不安全，shared_ptr有类似情况。我们避免使用reset函数。
- C++全局和静态变量初始化顺序的研究(CSDN)。
- 四人帮的经典之作：设计模式
- windows 核心编程(Jeffrey Richter)

原型模式

Prototype模式是一种对象创建型模式，它采取复制原型对象的方法来创建对象的实例。使用Prototype模式创建的实例，具有与原型一样的数据。

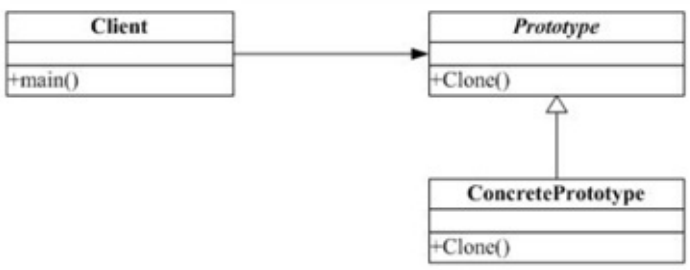
- 由原型对象自身创建目标对象。也就是说，对象创建这一动作发自原型对象本身。
- 目标对象是原型对象的一个克隆。也就是说，通过Prototype模式创建的对象，不仅仅与原型对象具有相同的结构，还与原型对象具有相同的值。
- 根据对象克隆深度层次的不同，有浅度克隆与深度克隆。

原型模式允许动态的增加或减少产品类，产品类不需要非得有任何事先确定的等级结构，原始模型模式适用于任何的等级结构。缺点是每一个类都必须配备一个克隆方法。

跟MM用QQ聊天，一定要说些深情的话语了，我搜集了好多肉麻的情话，需要时只要copy出来放到QQ里面就行了，这就是我的情话prototype了。

原型模式：通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的方法创建出更多同类型的对象。

角色和职责



Prototype 模式提供了一个通过已存在对象创建新对象的接口（clone），clone()的实现和具体的编程语言相关，在C++中我们通过拷贝构造函数实现之。

原型模式主要面对的问题是：“某些结构复杂的对象”的创建工作；由于需求的变化，这些对象经常面临着剧烈的变化，但是他们却拥有比较稳定一致的接口。

适用情况：一个复杂对象，具有自我复制功能，统一一套接口。

示例代码

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include "string"
using namespace std;

class Person
{
public:
    virtual Person* clone() = 0;
    virtual void printT() = 0;
};

class  CPlusPlusProgrammer : public Person
{
public:
    CPlusPlusProgrammer()
    {
        m_name = "";
        m_age = 0;
        m_resume = NULL;
        setResume("aaaa");
    }

    CPlusPlusProgrammer(string name, int age)
    {
        m_name = name;
        m_age = age;
        m_resume = NULL;
        setResume("aaaa");
    }

    void setResume(char *p)
    {
        if (m_resume != NULL)
        {
            delete m_resume;
        }
```

```
        m_resume = new char[strlen(p) + 1];
        strcpy(m_resume, p);
    }

    virtual void printT()
    {
        cout << "m_name" << m_name << " m_age" << m_age << "m_re
sume:" << m_resume <<  endl;
    }

    virtual Person* clone()
    {
        CPlusPlusProgrammer *tmp = new CPlusPlusProgrammer;
        //tmp->m_name = this->m_name;
        *tmp = *this; // = 浅拷贝
        return tmp;
    }

private:
    string    m_name;
    int        m_age ;
    char    *m_resume;
};

void main()
{
    Person *c1 = new CPlusPlusProgrammer("张三", 32);
    c1->printT();

    Person *c2 = c1->clone();
    c2->printT();

    system("pause");
    return ;
}
```

```
#include <iostream>
#include <string>
using namespace std;
```

```
class Resume
{
private:
    string name, sex, age, timeArea, company;
public:
    Resume(string s)
    {
        name = s;
    }
    void setPersonalInfo(string s, string a)
    {
        sex = s;
        age = a;
    }
    void setWorkExperience(string t, string c)
    {
        timeArea = t;
        company = c;
    }
    void display()
    {
        cout << name << "  " << sex << "  " << age << endl;
        cout << "工作经历:  " << timeArea << "  " << company << e
ndl << endl;

    }
    Resume *clone()
    {
        Resume *b;
        b = new Resume(name);
        b->setPersonalInfo(sex, age);
        b->setWorkExperience(timeArea, company);
        return b;
    }
};


int main()
{
```

```
    Resume *r = new Resume("李彦宏");
    r->setPersonalInfo("男", "30");
    r->setWorkExperience("2007-2010", "读研究生");
    r->display();

    Resume *r2 = r->clone();
    r2->setWorkExperience("2003-2007", "读本科");

    r->display();
    r2->display();

    cin.get();
    return 0;
}
```

组合模式

Composite模式也叫组合模式，是构造型的设计模式之一。通过递归手段来构造树形的对象结构，并可以通过一个对象来访问整个对象树。

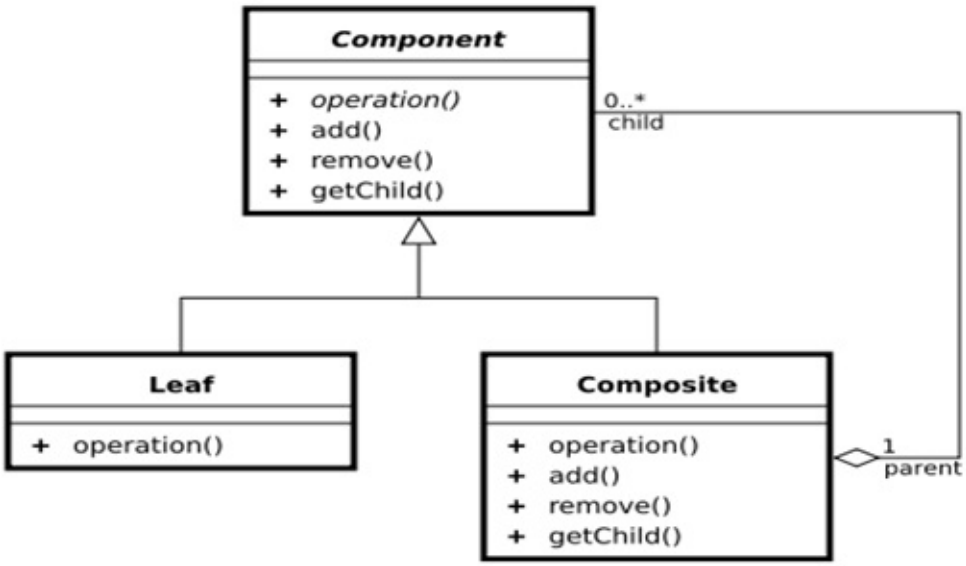
合成模式：合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。

合成模式就是一个处理对象的树结构的模式。合成模式把部分与整体的关系用树结构表示出来。

合成模式使得客户端把一个个单独的成分对象和由他们复合而成的合成对象同等看待。

- Mary今天过生日。“我过生日，你要送我一件礼物。”
- 嗯，好吧，去商店，你自己挑。
- “这件T恤挺漂亮，买，这条裙子好看，买，这个包也不错，买 。”
- “喂，买了三件了呀，我只答应送一件礼物的。”
- 什么呀，T恤加裙子加包包，正好配成一套呀，小姐，麻烦你包起来。
- “……”，MM都会用Composite模式了，你会了没有？

类图角色和职责



Component （树形结构的节点抽象）

- 为所有的对象定义统一的接口（公共属性，行为等的定义）
- 提供管理子节点对象的接口方法
- [可选]提供管理父节点对象的接口方法

Leaf（树形结构的叶节点）：Component的实现子类

Composite（树形结构的枝节点）：Component的实现子类

适用于：单个对象和组合对象的使用具有一致性。将对象组合成树形结构以表示“部分--整体”

示例代码

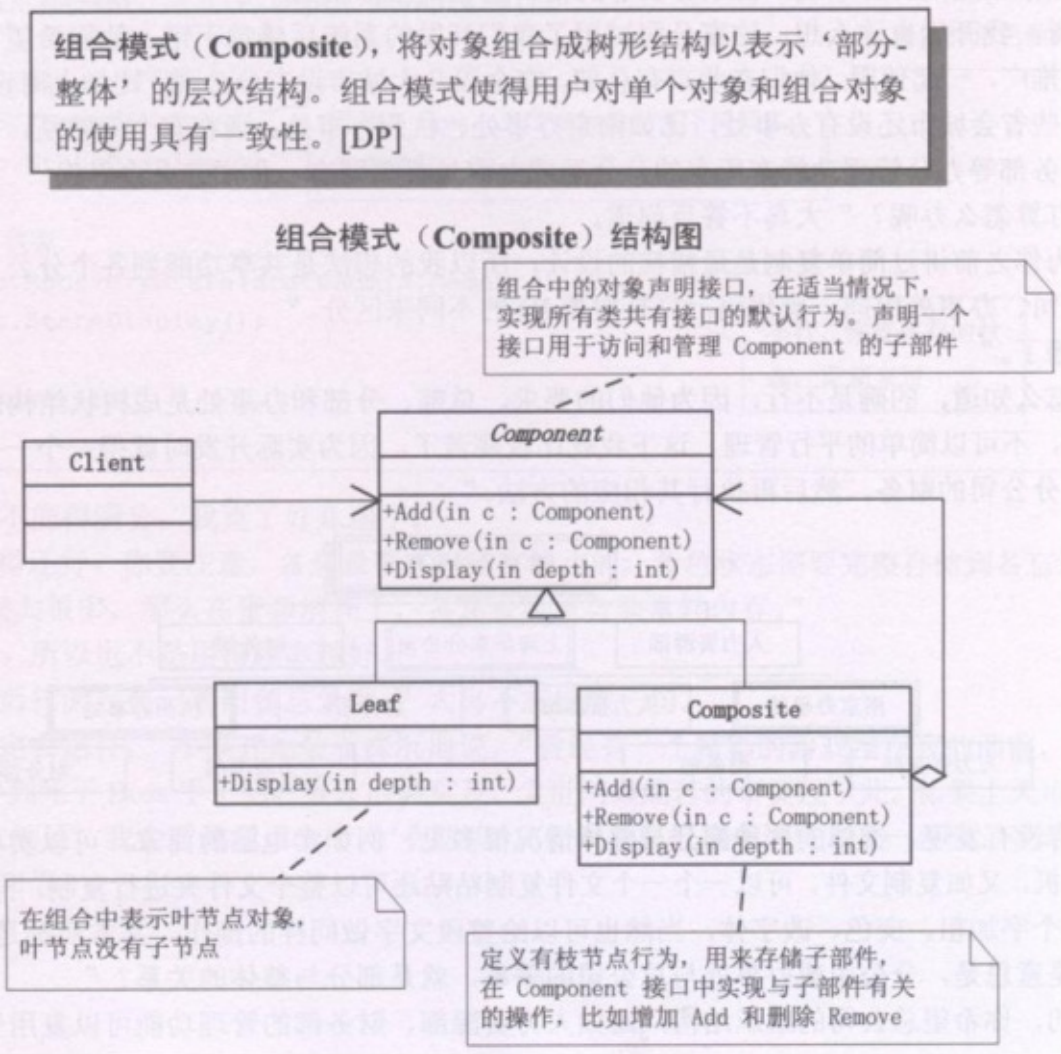
```
#include <iostream>
#include "string"
#include "list"
using namespace std;

class IFile
{
public:
    virtual void display() = 0;
    virtual int add(IFile *ifile) = 0;
    virtual int remove(IFile *ifile) = 0;
    virtual list<IFile *>* getChild() = 0;
};

//文件结点
class File : public IFile
{
public:
    File(string name)
    {
        m_name = name;
    }

    virtual void display()
    {
        cout << m_name << endl;
```

```
    }

    virtual int add(IFile *ifile)
    {
        return -1;
    }

    virtual int remove(IFile *ifile)
    {
        return -1;
    }

    virtual list<IFile *>*  getChild()
    {
        return NULL;
    }
private:
    string m_name;
};

//目录结点
class Dir : public IFile
{
public:
    Dir(string name)
    {
        m_name = name;
        m_list = new list<IFile *>;
        m_list->clear();
    }

    virtual void display()
    {
        cout << m_name << endl;
    }

    virtual int add(IFile *ifile)
    {
        m_list->push_back(ifile);
        return 0;
```

```
    }

    virtual int remove(IFile *ifile)
    {
        m_list->remove(ifile);
        return 0;
    }

    virtual list<IFile *>*  getChild()
    {
        return m_list;
    }
private:
    string m_name;
    list<IFile *>  *m_list;
};

// 递归的显示树
void showTree(IFile *root, int level)
{
    int i = 0;
    if (root == NULL)
    {
        return ;
    }
    for (i=0; i<level; i++)
    {
        printf("\t");
    }
    //1 显示根 结点
    root->display();

    //2 若根结点有孩子
    // 判读孩子是文件,显示名字
    // 判断孩子是目录,showTree(子目录)

    list<IFile *>  *mylist = root->getChild();
    if (mylist != NULL) //说明是一个目录
    {
        for (list<IFile *>::iterator it=mylist->begin(); it!=myl
```

```
ist->end(); it++)
        {
            if ( (*it)->getChild() == NULL )
            {
                for (i=0; i<=level; i++) //注意 <=
                {
                    printf("\t");
                }
                (*it)->display();
            }
            else
            {
                showTree(*it, level+1);
            }
        }
    }
}

void main()
{
    Dir *root = new Dir("C");
    //root->display();

    Dir *dir1 = new Dir("111dir");
    File *aaafile = new File("aaa.txt");

    //获取root结点下的 孩子集合
    list<IFile *>  *mylist =  root->getChild();

    root->add(dir1);
    root->add(aaafile);

    for ( list<IFile *>::iterator it=mylist->begin(); it!=mylist
->end(); it++ )
    {
        (*it)->display();
    }

    Dir *dir222 = new Dir("222dir");
    File *bbbfile = new File("bbb.txt");
```

```
    dir1->add(dir222);
    dir1->add(bbbfile);

    cout << "通过 showTree 方式显示 root 结点下的所有子结点" << endl;

    showTree(root, 0);

    system("pause");
    return ;
}
```

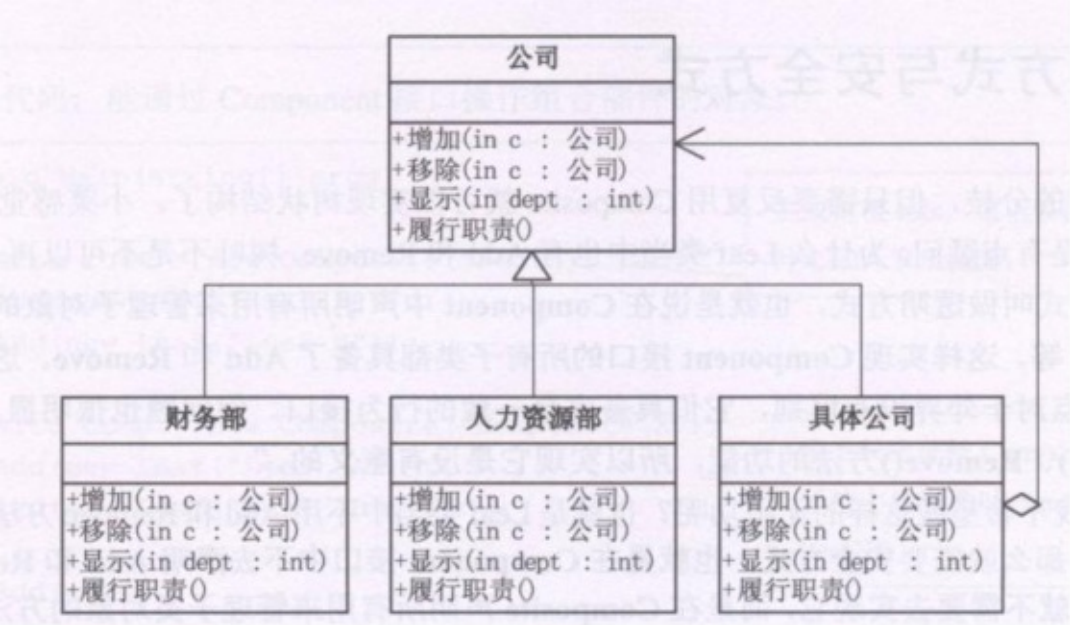



```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
```

```
class Component
{
public:
    string name;
    Component(string name)
    {
        this->name = name;
    }
    virtual void add(Component *) = 0;
    virtual void remove(Component *) = 0;
    virtual void display(int) = 0;
};

class Leaf :public Component
{
public:
    Leaf(string name) :Component(name)
    {}
    void add(Component *c)
    {
        cout << "leaf cannot add" << endl;
    }
    void remove(Component *c)
    {
        cout << "leaf cannot remove" << endl;
    }
    void display(int depth)
    {
        string str(depth, '-');
        str += name;
        cout << str << endl;
    }
};

class Composite :public Component
{
private:
    vector<Component*> component;
public:
    Composite(string name) :Component(name)
```

```
    {}
    void add(Component *c)
    {
        component.push_back(c);
    }
    void remove(Component *c)
    {
        vector<Component*>::iterator iter = component.begin();
        while (iter != component.end())
        {
            if (*iter == c)
            {
                component.erase(iter);
            }
            iter++;
        }
    }
    void display(int depth)
    {
        string str(depth, '-');
        str += name;
        cout << str << endl;

        vector<Component*>::iterator iter = component.begin();
        while (iter != component.end())
        {
            (*iter)->display(depth + 2);
            iter++;
        }
    }
};


int main()
{
    Component *p = new Composite("小李");
    p->add(new Leaf("小王"));
    p->add(new Leaf("小强"));

    Component *sub = new Composite("小虎");
```

```
    sub->add(new Leaf("小王"));
    sub->add(new Leaf("小明"));
    sub->add(new Leaf("小柳"));

    p->add(sub);
    p->display(0);

    cout << "*******" << endl;
    sub->display(2);

    cin.get();

    return 0;
}
```




```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Company
{
protected:
    string m_strName;
public:
```

```
    Company(string strName)
    {
        m_strName = strName;
    }

    virtual void Add(Company* c)=0;
    virtual void Display(int nDepth)=0;
    virtual void LineOfDuty()=0;
};

class ConcreteCompany: public Company
{
private:
    vector<Company*> m_company;
public:
    ConcreteCompany(string strName):Company(strName){}

    virtual void Add(Company* c)
    {
        m_company.push_back(c);
    }
    virtual void Display(int nDepth)
    {
        string strtemp;
        for(int i=0; i < nDepth; i++)
        {
            strtemp += "-";
        }
        strtemp +=m_strName;
        cout<<strtemp<<endl;

        vector<Company*>::iterator p=m_company.begin();
        while (p!=m_company.end())
        {
            (*p)->Display(nDepth+2);
            p++;
        }
    }
    virtual void LineOfDuty()
    {
```

```
        vector<Company*>::iterator p=m_company.begin();
        while (p!=m_company.end())
        {
            (*p)->LineOfDuty();
            p++;
        }
    }
};

class HrDepartment : public Company
{
public:

    HrDepartment(string strname) : Company(strname){}

    virtual void Display(int nDepth)
    {
        string strtemp;
        for(int i = 0; i < nDepth; i++)
        {
            strtemp += "-";
        }

        strtemp += m_strName;
        cout<<strtemp<<endl;
    }
    virtual void Add(Company* c)
    {
        cout<<"error"<<endl;
    }

    virtual void LineOfDuty()
    {
        cout<<m_strName<<":招聘人才"<<endl;
    }
};

//客户端：
int main()
{
```

```
    ConcreteCompany *p = new ConcreteCompany("清华大学");
    p->Add(new HrDepartment("清华大学人才部"));

    ConcreteCompany *p1 = new ConcreteCompany("数学系");
    p1->Add(new HrDepartment("数学系人才部"));

    ConcreteCompany *p2 = new ConcreteCompany("物理系");
    p2->Add(new HrDepartment("物理系人才部"));

    p->Add(p1);
    p->Add(p2);

    p->Display(1);
    p->LineOfDuty();
    return 0;
}
```

模板模式

Template Method模式也叫模板方法模式，是行为模式之一，它把具有特定步骤算法中的某些必要的处理委让给抽象方法，通过子类继承对抽象方法的不同实现改变整个算法的行为。

模板方法模式：模板方法模式准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。

不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。先制定一个顶级逻辑框架，而将逻辑的细节留给具体的子类去实现。

女生从认识到得手的不变的步骤分为巧遇、打破僵局、展开追求、接吻、得手

但每个步骤针对不同的情况，都有不一样的做法，这就要看你随机应变啦(具体实现)

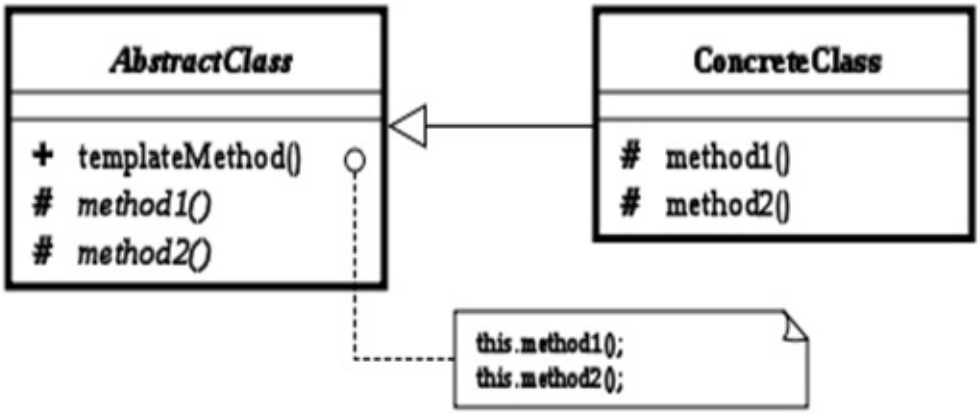
应用场景

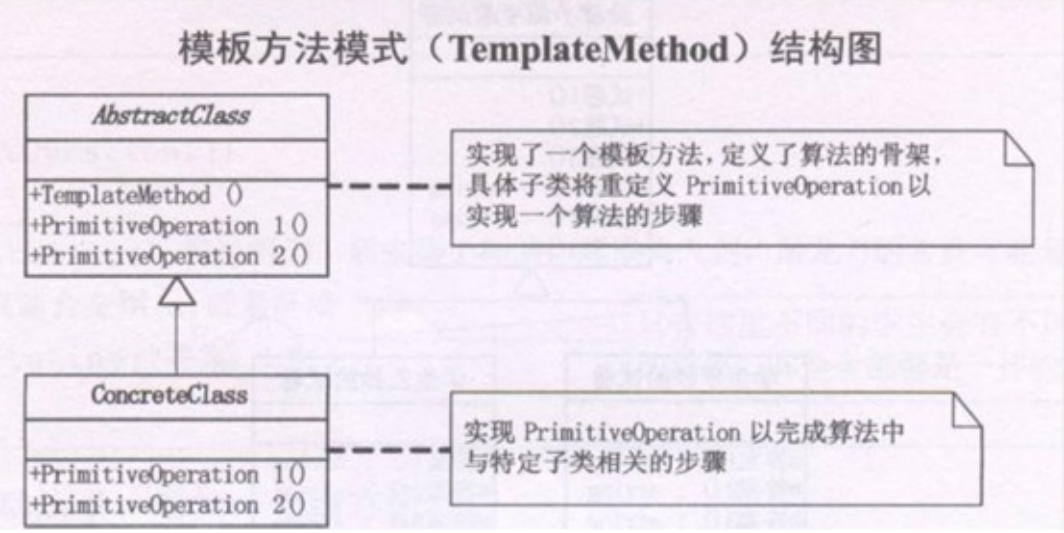
Template Method模式一般应用在具有以下条件的应用中：

- 具有统一的操作步骤或操作过程
- 具有不同的操作细节
- 存在多个具有同样操作步骤的应用场景，但某些具体的操作细节却各不相同

总结：在抽象类中统一操作步骤，并规定好接口；让子类实现接口。这样可以把各个具体的子类和操作步骤接耦合

类图角色和职责





- AbstractClass：抽象类的父类
- ConcreteClass：具体的实现子类
- templateMethod()：模板方法
- method1()与method2()：具体步骤方法

示例代码

```
#include <iostream>
using namespace std;

class MakeCar
{
public:
    virtual void MakeHead() = 0;
    virtual void MakeBody() = 0;
    virtual void MakeTail() = 0;
```

```
public:
    void Make() //模板函数 把业务逻辑给做好
    {
        MakeTail();
        MakeBody();
        MakeHead();
    }
};

class Jeep : public MakeCar
{
public:
    virtual void MakeHead()
    {
        cout << "jeep head" << endl;
    }

    virtual void MakeBody()
    {
        cout << "jeep body" << endl;
    }

    virtual void MakeTail()
    {
        cout << "jeep tail" << endl;
    }
};

class Bus : public MakeCar
{
public:
    virtual void MakeHead()
    {
        cout << "Bus head" << endl;
    }

    virtual void MakeBody()
    {
        cout << "Bus body" << endl;
    }
```

```
    virtual void MakeTail()
    {
        cout << "Bus tail" << endl;
    }
};

void main()
{
    MakeCar *car = new Bus;
    car->Make();
    delete car;

    MakeCar *car2 = new Jeep;
    car2->Make();
    delete car2;

    system("pause");
    return ;
}
```

```
#include<iostream>
#include <vector>
#include <string>
using namespace std;

class AbstractClass
{
public:
    void Show()
    {
        cout << "我是" << GetName() << endl;
    }
protected:
    virtual string GetName() = 0;
};

class Naruto : public AbstractClass
{
```

```
protected:
    virtual string GetName()
    {
        return "火影史上最帅的六代目---一鸣惊人naruto";
    }
};

class OnePice : public AbstractClass
{
protected:
    virtual string GetName()
    {
        return "我是无恶不做的大海贼---路飞";
    }
};

//客户端
int main()
{
    Naruto* man = new Naruto();
    man->Show();

    OnePice* man2 = new OnePice();
    man2->Show();

    cin.get();
    return 0;
}
```

简单工厂模式

简单工厂模式定义了创建对象的接口，将创建对象的细节隐藏，产生具体对象时只需要向工厂提供产品类型，而无需根据每一个具体类依次调用各自的构造函数。

一个工厂生产多种产品，采购商只需要提供所需的产品名称，而无需了解产品是哪个车间生产的。

简单工厂模式属于类的创建型模式,又叫做静态工厂方法模式。通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

核心思想是用一个工厂来根据输入的条件产生不同的类，然后根据不同类的virtual函数得到不同的结果。

工厂模式：客户类和工厂类分开。

消费者任何时候需要某种产品，只需向工厂请求即可。

消费者无须修改就可以接纳新产品。缺点是当产品修改时，工厂类也要做相应的修改。如：如何创建及如何向客户端提供。

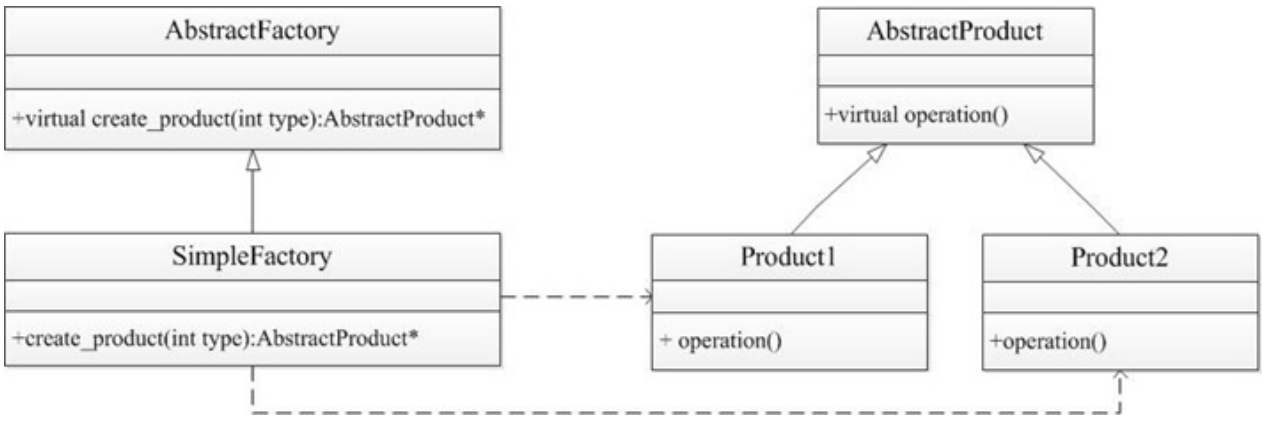
追MM少不了请吃饭了，麦当劳的鸡翅和肯德基的鸡翅都是MM爱吃的东西，虽然口味有所不同，但不管你带MM去麦当劳或肯德基，只管向服务员说“来四个鸡翅”就行了。麦当劳和肯德基就是生产鸡翅的Factory。

第一，基类存放数据

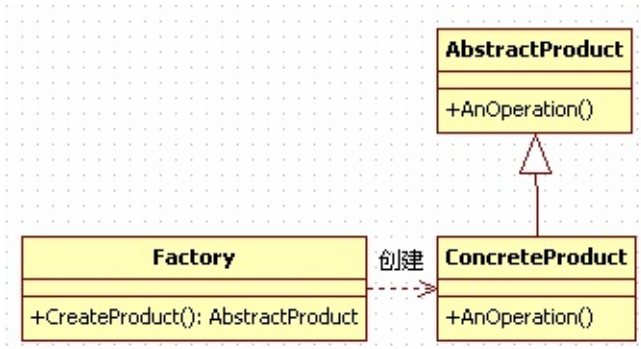
第二，派生类有很多，派生类存放数据的操作

第三实现接口类，用静态函数实现调用各种派生类

模式中包含的角色及其职责



- 抽象工厂：工厂类的基类，提供工厂的公共操作接口
- 具体工厂：简单工厂模式的核心，用来生产具体产品
- 抽象产品：具体产品类的基类，负责提供产品操作的公共接口
- 具体产品：是抽象产品的具体实现，描述不同产品的具体信息



- 工厂（Creator）角色

简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类可以被外界直接调用，创建所需的产品对象。

- 抽象（Product）角色

简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

- 具体产品（Concrete Product）角色

简单工厂模式所创建的具体实例对象

依赖: 一个类的对象当另外一个类的函数参数或者是返回值

简单工厂模式的优缺点

在这个模式中，工厂类是整个模式的关键所在。它包含必要的判断逻辑，能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。用户在使用时可以直接根据工厂类去创建所需的实例，而无需了解这些对象是如何创建以及如何组织的。有利于整个软件体系结构的优化。不难发现，简单工厂模式的缺点也正体现在其工厂类上，由于工厂类集中了所有实例的创建逻辑，所以“高内聚”方面做的并不好。另外，当系统中的具体产品类不断增多时，可能会出现要求工厂类也要做相应的修改，扩展性并不很好。

GOOD：适用于不同情况创建不同的类时

BUG：客户端必须要知道基类和工厂类，耦合性差

核心思想是用一个工厂来根据输入的条件产生不同的类，然后根据不同类的virtual函数得到不同的结果。

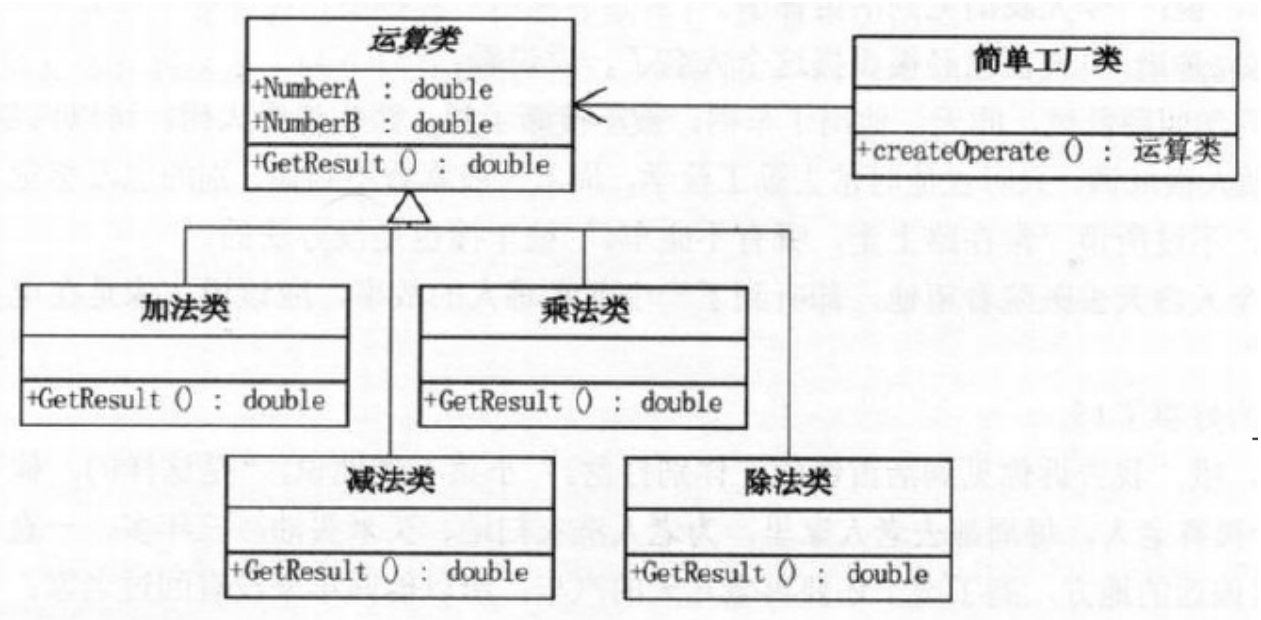
```
#include <iostream>
using namespace std;

class Fruit
{
public:
    virtual void getFruit() = 0;
};

class Banana : public Fruit
{
public:
    virtual void getFruit()
    {
        cout << "我是香蕉..." << endl;
    }
};

class Apple : public Fruit
{
public:
    virtual void getFruit()
    {
        cout << "我是苹果..." << endl;
    }
```

```
};

class Factory
{
public:
    Fruit *create(char *p)
    {
        if (strcmp(p, "banana") == 0)
        {
            return new Banana;
        }
        else if (strcmp(p, "apple") == 0)
        {
            return new Apple;
        }
        else
        {
            printf("不支持\n" ) ;
            return NULL;
        }
    }
};

void main()
{
    Factory *f = new Factory;
    Fruit *fruit = NULL;

    //工厂生产 香蕉
    fruit = f->create("banana");
    fruit->getFruit();
    delete fruit;

    fruit = f->create("apple");
    fruit->getFruit();
    delete fruit;

    delete f;

    system("pause");
```

```
    return ;
}
```

示例代码

模拟四则运算




```
#include <iostream>
#include <string>
using namespace std;

class Operation    //基类存放数据
{
public:
    double numberA, numberB;//两个数
    virtual double  getResult()//获取结果
    {
        return 0;
    }
};

class addOperation :public Operation//派生类存放操作
{
    double getResult()
    {
```

```
        return numberA + numberB;
    }
};

class subOperation :public Operation
{
    double getResult()
    {
        return numberA - numberB;
    }
};

class mulOperation :public Operation
{
    double getResult()
    {
        return numberA*numberB;
    }
};

class divOperation :public Operation
{
    double getResult()
    {
        return numberA / numberB;
    }
};

class operFactory
{
public:
    static Operation *createOperation(char c)
    {
        switch (c)
        {
        case '+':
            return new addOperation;
            break;

        case '-':
```

```
            return new subOperation;
            break;

        case '*':
            return new mulOperation;
            break;

        case '/':
            return new divOperation;
            break;
        }
    }
};

int main()
{
    Operation *oper = operFactory::createOperation('-');
    oper->numberA = 9;
    oper->numberB = 99;
    cout << oper->getResult() << endl;

    cin.get();
    return 0;
}
```

示例代码2

- 定义抽象工厂类

```
//factory.h
#ifndef FACTORY_H
#define FACTORY_H
class AbstractProduct;

class AbstractFactory{ //定义抽象工厂类
public:
    virtual ~AbstractFactory();
    virtual AbstractProduct *create_product(int) = 0;
protected:
    AbstractFactory();
};
//定义简单工厂类，派生自抽象工厂类，主要功能是生产不同产品
class Factory :public AbstractFactory{
public:
    ~Factory();
    Factory();
    AbstractProduct *create_product(int);//根据参数，产生不同产品对象

};
#endif
```

```
//factory.cpp
#include <iostream>
#include "factory.h"
#include "product.h"
using namespace std;

AbstractFactory::AbstractFactory()  //定义抽象工厂类构造函数
{
}
AbstractFactory::~AbstractFactory() //定义抽象工厂类析构函数
{
}
Factory::Factory()                  //定义简单工厂类构造函数
{
    cout << "factory constructor!" << endl;
}
Factory::~Factory()                  //定义简单工厂类析构函数
{
}
//定义create_product()函数，该函数根据参数创建不同产品对象
AbstractProduct *Factory::create_product(int type)
{
    switch (type){
    case 1:    return new Product1();
    case 2: return new Product2();
    }
}
```

- 定义抽象产品类，提供产品操作的接口

```
//product.h
#ifndef PRODUCT_H
#define PRODUCT_H

class AbstractProduct{//定义抽象产品类，提供产品操作的接口
public:
    virtual ~AbstractProduct();
    virtual void operation() = 0;
protected:
    AbstractProduct();
};
//定义类Product1，派生自抽象产品类
class Product1 :public AbstractProduct{
public:
    ~Product1();
    Product1();
    void operation(); //定义属于本产品的操作函数
};
//定义Product2，派生自抽象产品类
class Product2 :public AbstractProduct{
public:
    ~Product2();
    Product2();
    void operation();
};
#endif
```

```
//product.cpp
#include "product.h"
#include <iostream>
using namespace std;

//定义AbstractProduct构造函数
AbstractProduct::AbstractProduct(){}

//定义AbstractProduct析构函数
AbstractProduct::~AbstractProduct(){}

Product1::Product1()//定义Product1构造函数
{
    cout << "product1 constructor!" << endl;
}

//定义Product1析构函数
Product1::~Product1(){}

void Product1::operation()//定义Product1操作函数
{
    cout << "product1 operation!" << endl;
}

Product2::Product2()//定义Product2构造函数
{
    cout << "product2 constructor!" << endl;
}

//定义Product2析构函数
Product2::~Product2(){}

void Product2::operation()//定义Product2操作函数
{
    cout << "product2 operation!" << endl;
}
```

- main.cpp

```
#include <iostream>
#include "product.h"
#include "factory.h"
using namespace std;

int main()
{
    AbstractFactory *fac = new Factory();           //创建简单工厂类

    AbstractProduct *p1 = fac->create_product(1);  //通过简单工厂生产产品1
    AbstractProduct *p2 = fac->create_product(2);  //通过简单工厂生产产品2
    p1->operation();                                //产品1执行相应操作
    p2->operation();                                //产品2执行相应操作
    delete fac;
    delete p1;
    delete p2;
    system("pause");
    return 0;
}
```

工厂方法模式

简单工厂模式为用户隐藏了生产产品的细节，但若想生产新产品需要修改工厂类，违背了类设计原则中的开放-封闭原则，即在模块在扩展性方面应该是开放的，在更改性方面应该是封闭的，工厂方法模式通过添加工厂子类的方式改进了简单工厂模式，遵循了开放-封闭原则。

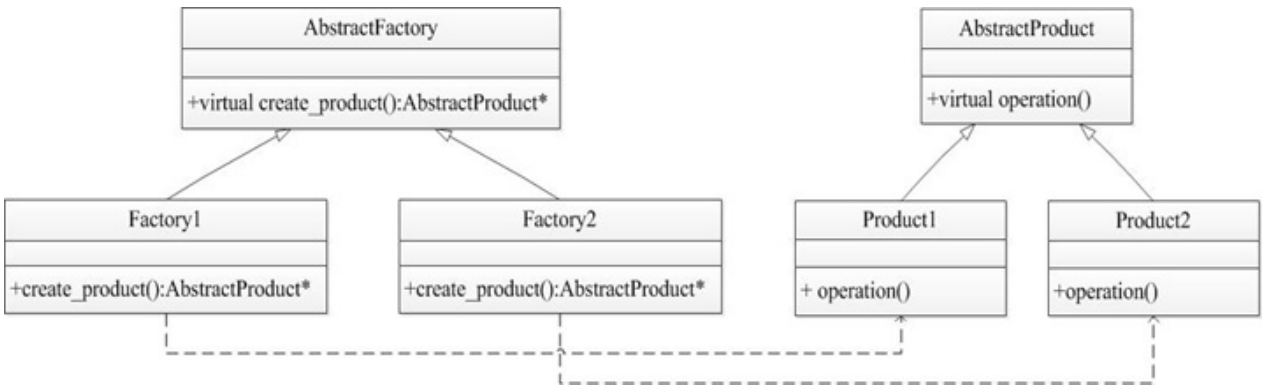
工厂方法模式同样属于类的创建型模式又被称为多态工厂模式 。工厂方法模式的意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。

工厂方法模式：核心工厂类不再负责所有产品的创建，而是将具体创建的工作交给子类去做，成为一个抽象工厂角色，仅负责给出具体工厂类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。

请MM去麦当劳吃汉堡，不同的MM有不同的口味，要每个都记住是一件烦人的事情，我一般采用Factory Method模式，带着MM到服务员那儿，说“要一个汉堡”，具体要什么样的汉堡呢，让MM直接跟服务员说就行了。

类图角色和职责



- 抽象工厂（Creator）角色

工厂方法模式的核心，任何工厂类都必须实现这个接口。

- 具体工厂（ Concrete Creator）角色

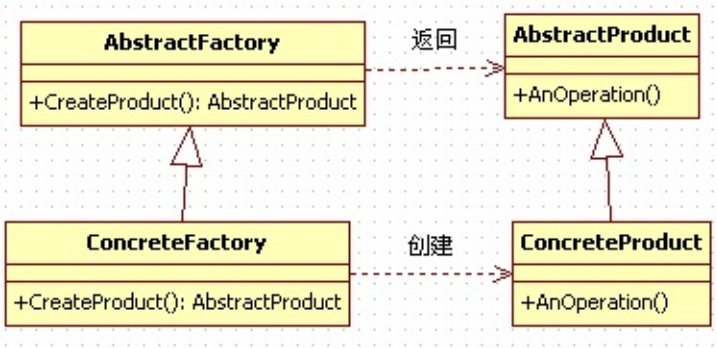
具体工厂类是抽象工厂的一个实现，负责实例化产品对象。

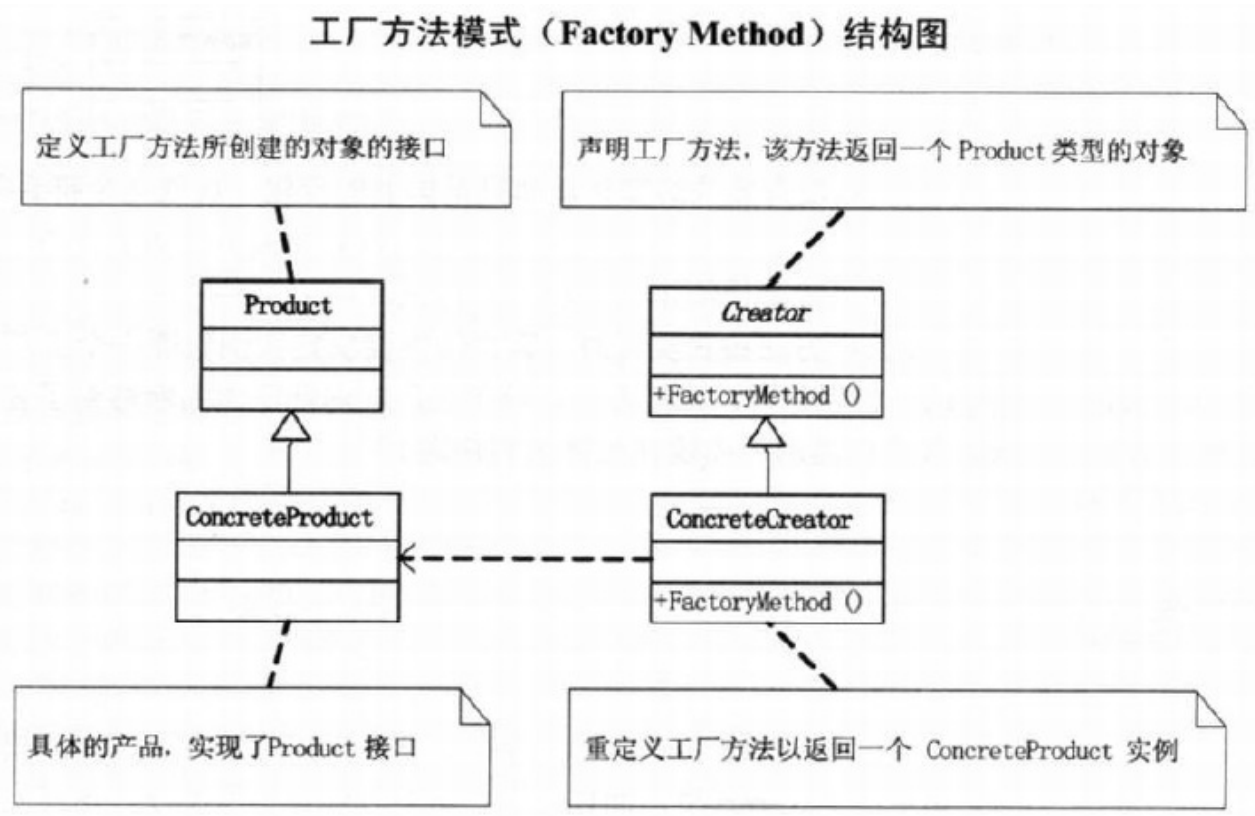
- 抽象（Product）角色

工厂方法模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

- 具体产品（Concrete Product）角色

工厂方法模式所创建的具体实例对象





工厂方法模式和简单工厂模式比较

工厂方法模式与简单工厂模式在结构上的不同不是很明显。工厂方法类的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上。

工厂方法模式之所以有一个别名叫多态性工厂模式是因为具体工厂类都有共同的接口，或者有共同的抽象父类。

当系统扩展需要添加新的产品对象时，仅仅需要添加一个具体对象以及一个具体工厂对象，原有工厂对象不需要进行任何修改，也不需要修改客户端，很好的符合了“开放－封闭”原则。而简单工厂模式在添加新产品对象后不得不修改工厂方法，扩展性不好。工厂方法模式退化后可以演变成简单工厂模式。

“开放－封闭”通过添加代码的方式，不是通过修改代码的方式完成功能的增强。

示例代码

```
#include <iostream>
using namespace std;

class Fruit
{
public:
    virtual void  sayname()  = 0;
};

class  Banana : public Fruit
{
public:
    void  sayname()
    {
        cout << "我是香蕉" << endl;
    }
};

class  Apple : public Fruit
{
public:
    void  sayname()
    {
        cout << "我是苹果" << endl;
    }
};
```

```
class  AbFactory
{
public:
    virtual Fruit *CreateProduct() = 0;
};

class BananaFactory :public AbFactory
{
public:
    virtual Fruit *CreateProduct()
    {
        return new Banana;
    }
};

class AppleFactory :public AbFactory
{
public:
    virtual Fruit *CreateProduct()
    {
        return new Apple;
    }
};

//添加新的产品
class Pear : public Fruit
{
public:
    virtual void sayname()
    {
        cout << "我是梨" << endl;
    }
};

class PearFactory : public AbFactory
{
public:
    virtual Fruit *CreateProduct()
    {
        return new Pear;
```

```
    }
};

void main()
{
    AbFactory    *factory = NULL;
    Fruit        *fruit = NULL;

    //吃香蕉
    factory = new BananaFactory;
    fruit = factory->CreateProduct();
    fruit->sayname();

    delete fruit;
    delete factory;

    //Pear
    factory = new PearFactory;
    fruit = factory->CreateProduct();
    fruit->sayname();

    delete fruit;
    delete factory;

    system("pause");
    return ;
}
```

```
#include <iostream>
#include <string>
using namespace std;

class Operation
{
public:
    double numberA, numberB;
    virtual double  getResult()
    {
        return 0;
```

```
    }
};

class addOperation :public Operation
{
    double getResult()
    {
        return numberA + numberB;
    }
};

class subOperation :public Operation
{
    double getResult()
    {
        return numberA - numberB;
    }
};

class mulOperation :public Operation
{
    double getResult()
    {
        return numberA*numberB;
    }
};

class divOperation :public Operation
{
    double getResult()
    {
        return numberA / numberB;
    }
};

class IFactory
{
public:
    virtual Operation *createOperation() = 0;
};
```

```
class AddFactory :public IFactory
{
public:
    static Operation *createOperation()
    {
        return new addOperation();
    }
};

class SubFactory :public IFactory
{
public:
    static Operation *createOperation()
    {
        return new subOperation();
    }
};

class MulFactory :public IFactory
{
public:
    static Operation *createOperation()
    {
        return new mulOperation();
    }
};

class DivFactory :public IFactory
{
public:
    static Operation *createOperation()
    {
        return new divOperation();
    }
};

int main()
{
```

```
    Operation *oper = MulFactory::createOperation();
    oper->numberA = 9;
    oper->numberB = 99;
    cout << oper->getResult() << endl;
    cin.get();
    return 0;
}
```

抽象工厂模式

除了简单工厂模式和工厂方法模式外，还有一种抽象工厂模式，它是对工厂方法模式的改进，增加了工厂生产产品的种类，减少了代码冗余，提高了程序效率。

抽象工厂模式是所有形态的工厂模式中最为抽象和最其一般性的。抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，能够创建多个产品族的产品对象。

工厂模式：客户类和工厂类分开。

消费者任何时候需要某种产品，只需向工厂请求即可。

消费者无须修改就可以接纳新产品。缺点是当产品修改时，工厂类也要做相应的修改。如：如何创建及如何向客户端提供。

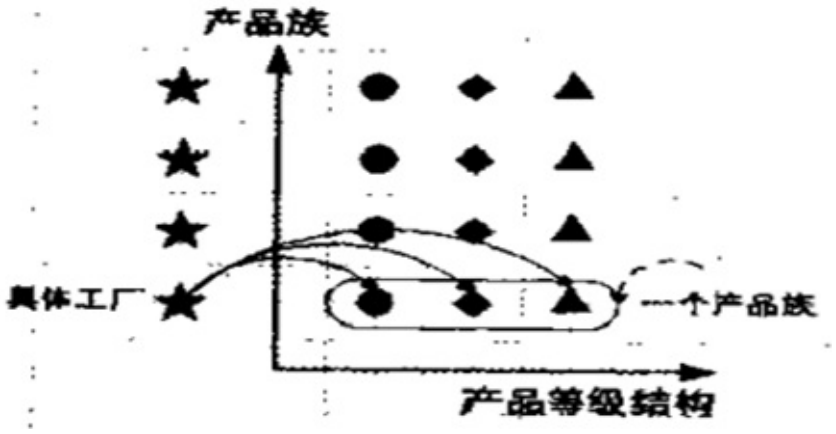
追MM少不了请吃饭了，麦当劳的鸡翅和肯德基的鸡翅都是MM爱吃的东西，虽然口味有所不同，但不管你带MM去麦当劳或肯德基，只管向服务员说“来四个鸡翅”就行了。麦当劳和肯德基就是生产鸡翅的Factory。

消费者不固定，工厂者不固定，（工厂根据消费者的行为进行动作）

实现消费者抽象基类，消费者派生类的实现，实例化就是消费者

- 操作的抽象基类，实现派生类各种操作，实例化的操作
- 工厂的抽象类，抽象类包含了两个抽象类的接口（消费者，操作）
- 抽象类实现了工厂类的抽象，实例化的派生类，实现工厂，
- 根据用户设置用户，根据操作设置操作

产品族和产品等级结构



工厂模式：要么生产香蕉、要么生产苹果、要么生产西红柿；但是不能同时生产一个产品组。

抽象工厂：能同时生产一个产品族。

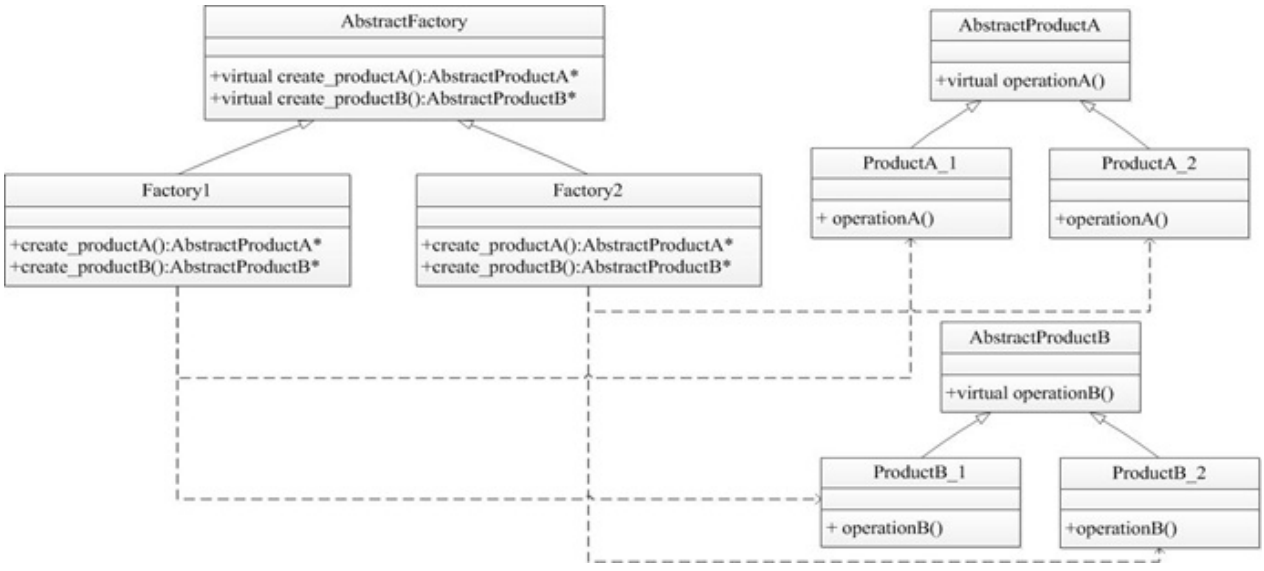
具体工厂在开闭原则下，能生产香蕉/苹果/梨子 (产品等级结构)

抽象工厂:在开闭原则下，能生产：南方香蕉/苹果/梨子，北方香蕉/苹果/梨子 (产品族)

工厂模式和抽象工厂的重要区别：

- 工厂模式只能生产一个产品。（要么香蕉、要么苹果）
- 抽象工厂可以一下生产一个产品族（里面有很多产品组成）

类图及职责



- 抽象工厂（Creator）角色

抽象工厂模式的核心，包含对多个产品结构的声明，任何工厂类都必须实现这个接口。

- 具体工厂（ Concrete Creator）角色

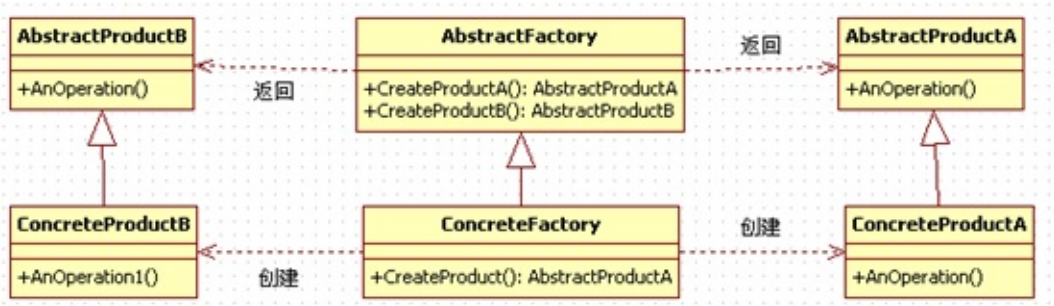
具体工厂类是抽象工厂的一个实现，负责实例化某个产品族中的产品对象。

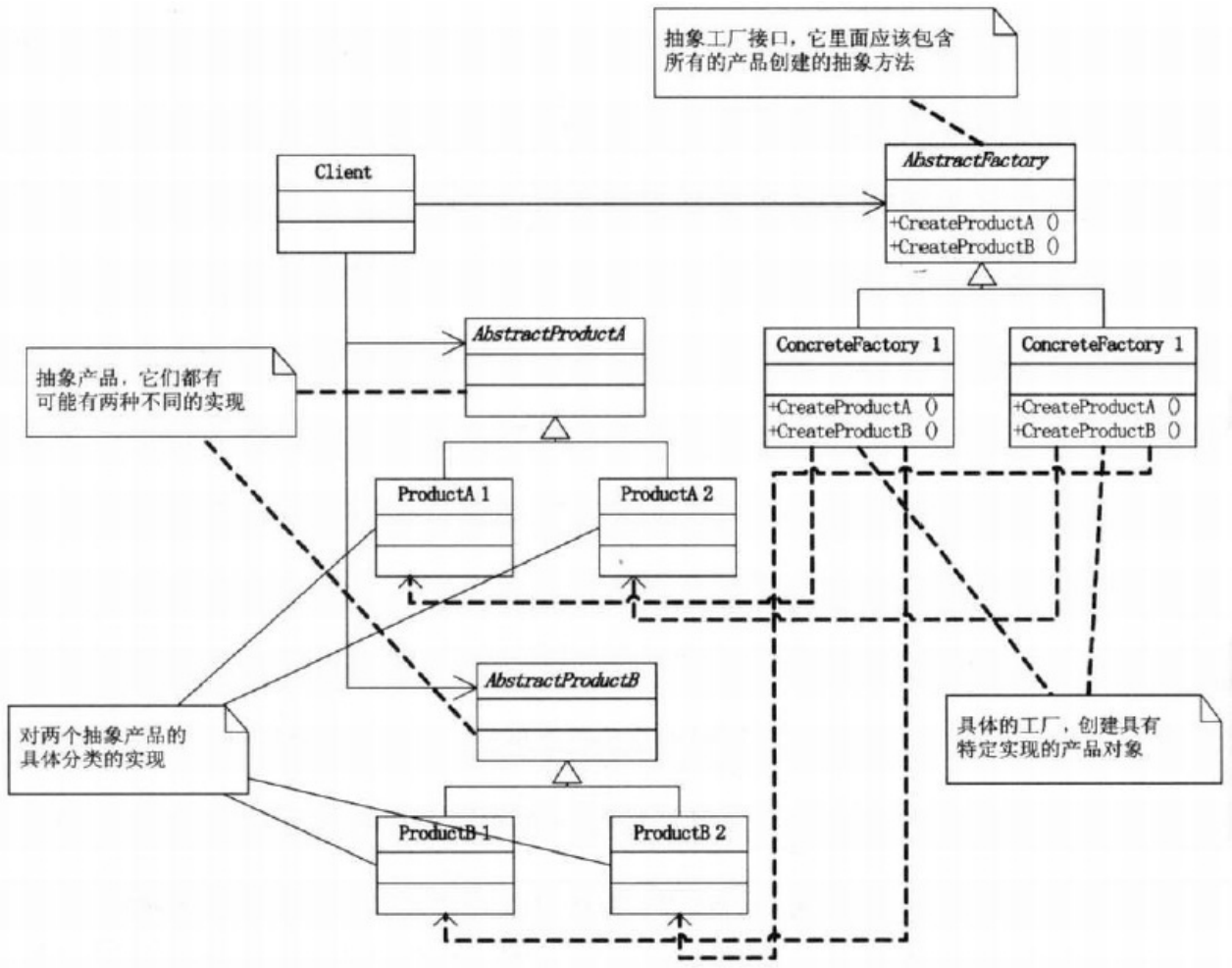
- 抽象（Product）角色

抽象模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

- 具体产品（Concrete Product）角色

抽象模式所创建的具体实例对象





示例代码

```
#include <iostream>
using namespace std;

class Fruit
{
public:
    virtual void SayName() = 0;
};

class AbstractFactory
{
public:
    virtual Fruit* CreateBanana() = 0;
    virtual Fruit* CreateApple() = 0;
};

class NorthBanana : public Fruit
{
public:
    virtual void SayName()
    {
        cout << "我是北方香蕉" << endl;
    }
};

class NorthApple : public Fruit
{
public:
    virtual void SayName()
    {
        cout << "我是北方苹果" << endl;
    }
};

class SourthBanana : public Fruit
{
public:
```

```
    virtual void SayName()
    {
        cout << "我是南方香蕉" << endl;
    }
};

class SourthApple : public Fruit
{
public:
    virtual void SayName()
    {
        cout << "我是南方苹果" << endl;
    }
};

class NorthFacorty : public AbstractFactory
{
    virtual Fruit * CreateBanana()
    {
        return new NorthBanana;
    }
    virtual Fruit * CreateApple()
    {
        return new NorthApple;
    }
};

class SourthFacorty : public AbstractFactory
{
    virtual Fruit * CreateBanana()
    {
        return new SourthBanana;
    }
    virtual Fruit * CreateApple()
    {
        return new SourthApple;
    }
};

void main()
```

```
{
    Fruit            *fruit = NULL;
    AbstractFactory *af = NULL;

    af = new SourthFacorty;
    fruit = af->CreateApple();
    fruit->SayName();
    delete fruit;
    fruit = af->CreateBanana();
    fruit->SayName();
    delete fruit;

    af = new NorthFacorty;
    fruit = af->CreateApple();
    fruit->SayName();
    delete fruit;
    fruit = af->CreateBanana();
    fruit->SayName();
    delete fruit;
    delete af;

    system("pause");
    return ;
}
```

```
#include <iostream>
#include <string>
using namespace std;

class IUser
{
public:
    virtual void getUser() = 0;  //纯虚接口类，抽象类
    virtual void setUser() = 0;
};

class SqlUser :public IUser   //继承抽象实现sql数据库使用者的实例化
{
public:
```

```
    void getUser()
    {
        cout << "在sql中返回user" << endl;
    }
    void setUser()
    {
        cout << "在sql中设置user" << endl;
    }
};

class AccessUser :public IUser //继承抽象实现access数据库使用者的实例化
{
public:
    void getUser()
    {
        cout << "在Access中返回user" << endl;
    }
    void setUser()
    {
        cout << "在Access中设置user" << endl;
    }
};

class IDepartment  //抽象类，提供接口
{
public:
    virtual void getDepartment() = 0;
    virtual void setDepartment() = 0;
};

class SqlDepartment :public IDepartment  //SQL操作的实现
{
public:
    void getDepartment()
    {
        cout << "在sql中返回Department" << endl;
    }
    void setDepartment()
    {
```

```
        cout << "在sql中设置Department" << endl;
    }
};

class AccessDepartment :public IDepartment //access操作的实现
{
public:
    void getDepartment()
    {
        cout << "在Access中返回Department" << endl;
    }
    void setDepartment()
    {
        cout << "在Access中设置Department" << endl;
    }
};

class IFactory     //抽象工厂
{
public:
    virtual IUser *createUser() = 0;
    virtual IDepartment *createDepartment() = 0;
};

class SqlFactory :public IFactory  //抽象工厂一个实现
{
public:
    IUser *createUser()
    {
        return new SqlUser();
    }
    IDepartment *createDepartment()
    {
        return new SqlDepartment();
    }
};

class AccessFactory :public IFactory // 抽象工厂一个实现
{
public:
```

```
    IUser *createUser()
    {
        return new AccessUser();
    }
    IDepartment *createDepartment()
    {
        return new AccessDepartment();
    }
};

//变相的实现了静态类
class DataAccess
{
private:
    static string db;
    //string db="access";
public:
    static IUser *createUser()
    {
        if (db == "access")
        {
            return new AccessUser();
        }
        else if (db == "sql")
        {
            return new SqlUser();
        }
    }
    static IDepartment *createDepartment()
    {
        if (db == "access")
        {
            return new AccessDepartment();
        }
        else if (db == "sql")
        {
            return new SqlDepartment();
        }
    }
};
```

```
string DataAccess::db = "sql";

int main()
{
    //IFactory *factory=new SqlFactory();
    IFactory *factory;//抽象工厂
    IUser *user;//抽象消费者
    IDepartment *department;//提供的操作

    factory = new AccessFactory();//基类的指针指指向派生类的对象
    user = factory->createUser();//基类的指针指向派生类的对象
    department = factory->createDepartment();//基类的指针指向派生类
的对象

    user->getUser();
    user->setUser();//访问acesss接口

    department->getDepartment();
    department->setDepartment();//接口

    user = DataAccess::createUser();
    department = DataAccess::createDepartment();

    user->getUser();
    user->setUser();
    department->getDepartment();
    department->setDepartment();

    cin.get();
    return 0;
}
```

责任链模式

Chain of Responsibility（CoR）模式也叫职责链模式或者职责连锁模式，是行为模式之一，该模式构造一系列分别担当不同的职责的类的对象来共同完成一个任务，这些类的对象之间像链条一样紧密相连，所以被称作职责链模式。

例1：比如客户Client要完成一个任务，这个任务包括a,b,c,d四个部分。

首先客户Client把任务交给A，A完成a部分之后，把任务交给B，B完成b部分，...，直到D完成d部分。

例2：比如政府部分的某项工作，县政府先完成自己能处理的部分，不能处理的部分交给省政府，省政府再完成自己职责范围内的部分，不能处理的部分交给中央政府，中央政府最后完成该项工作。

例3：软件窗口的消息传播。

例4：SERVLET容器的过滤器（Filter）框架实现。

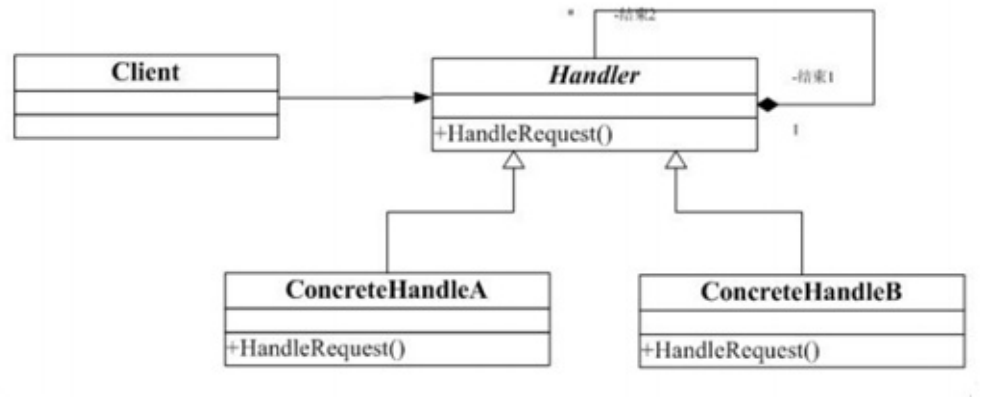
责任链模式：在责任链模式中，很多对象由每一个对象对其下家的引用而接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。

客户并不知道链上的哪一个对象最终处理这个请求，系统可以在不影响客户端的情况下动态的重新组织链和分配责任。处理者有两个选择：承担责任或者把责任推给下家。一个请求可以最终不被任何接收端对象所接受。

晚上去上英语课，为了好开溜坐到了最后一排，哇，前面坐了好几个漂亮的MM哎，找张纸条，写上“Hi, 可以做我的女朋友吗？如果不愿意请向前传”，纸条就一个接一个的传上去了，糟糕，传到第一排的MM把纸条传给老师了，听说是个老处女呀，快跑!

类图角色和职责

Handler ：处理类的抽象父类
concreteHandler ：具体的处理类



优缺点

优点：

- 责任的分担。每个类只需要处理自己该处理的工作（不该处理的传递给下一个对象完成），明确各类的责任范围，符合类的最小封装原则。
- 可以根据需要自由组合工作流程。如工作流程发生变化，可以通过重新分配对象链便可适应新的工作流程。
- 类与类之间可以以松耦合的形式加以组织。

缺点：因为处理时以链的形式在对象间传递消息，根据实现方式不同，有可能会影响处理的速度。

适用于：链条式处理事情。工作流程化、消息处理流程化、事物流程化。

示例代码

问题抛出

```
#include <iostream>
using namespace std;

class CarHandle
{
public:
    virtual void HandleCar() = 0;
};

class HeadCarHandle : public CarHandle
```

```
{
public:
    virtual void HandleCar()
    {
        cout << "造车头" << endl;
    }
};

class BodyCarHandle : public CarHandle
{
public:
    virtual void HandleCar()
    {
        cout << "造车身" << endl;
    }
};

class TailCarHandle : public CarHandle
{
public:
    virtual void HandleCar()
    {
        cout << "造 车 尾" << endl;
    }
};

void main()
{
    CarHandle *headHandle = new HeadCarHandle;
    CarHandle *bodyHandle = new BodyCarHandle;
    CarHandle *tailHandle = new TailCarHandle;

    //业务逻辑写死在客户端了
    headHandle->HandleCar();
    bodyHandle->HandleCar();
    tailHandle->HandleCar();

    delete headHandle;
    delete bodyHandle;
    delete tailHandle;
```

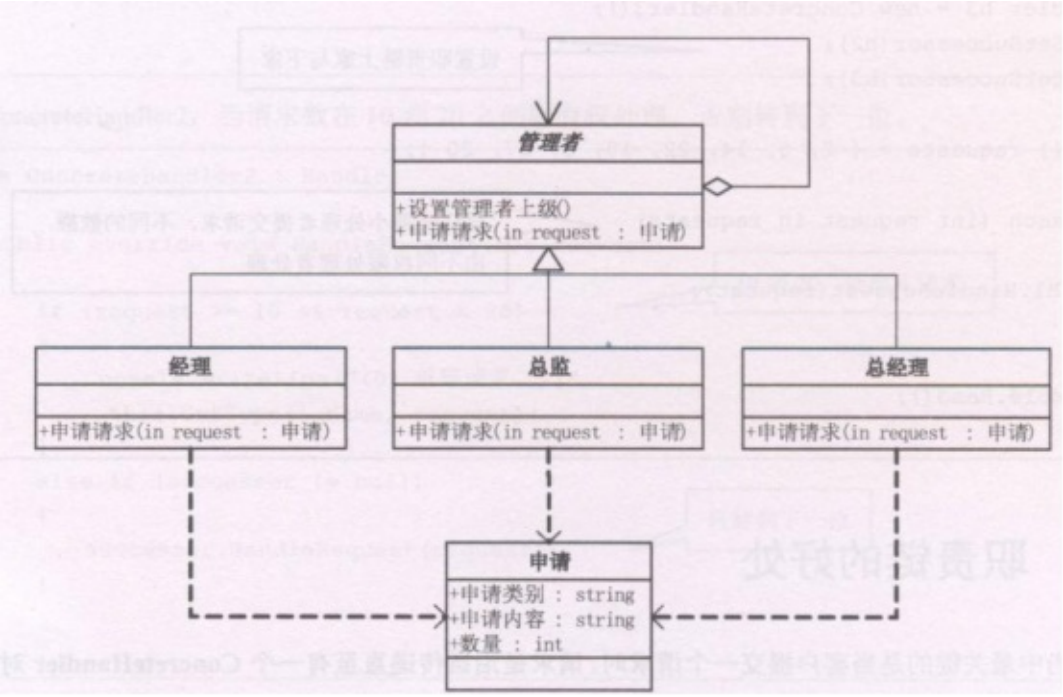
```
    system("pause");
    return ;
}
```

责任链模式

```
#include <iostream>
using namespace std;

//造完车以后,需要把任务传递下去
class CarHandle
{
public:
    virtual void HandleCar() = 0;

    CarHandle *setNextHandle(CarHandle *handle)
    {
        m_handle = handle;
        return m_handle;
    }

protected:
    CarHandle *m_handle; //下一个处理单元
};

class HeadCarHandle : public CarHandle
{
public:
    virtual void HandleCar()
    {
        cout << "造车头" << endl;
        //造玩车头以后把任务递交给下一个处理者
        if (m_handle != NULL)
        {
            m_handle->HandleCar();
        }
    }
};
```

```
class BodyCarHandle : public CarHandle
{
public:
    virtual void HandleCar()
    {
        cout << "造 车身" << endl;
        //造造车身以后把任务递交给下一个处理者
        if (m_handle != NULL)
        {
            m_handle->HandleCar();
        }
    }
};

class TailCarHandle : public CarHandle
{
public:
    virtual void HandleCar()
    {
        cout << "造车尾" << endl;
        //造造车尾以后把任务递交给下一个处理者
        if (m_handle != NULL)
        {
            m_handle->HandleCar();
        }
    }
};


void main()
{
    CarHandle *headHandle = new HeadCarHandle;
    CarHandle *bodyHandle = new BodyCarHandle;
    CarHandle *tailHandle = new TailCarHandle;

    //任务的处理关系
    /*
    headHandle->setNextHandle(bodyHandle);
    bodyHandle->setNextHandle(tailHandle);
```

```
    tailHandle->setNextHandle(NULL);
    */

    headHandle->setNextHandle(tailHandle);
    tailHandle->setNextHandle(bodyHandle);
    bodyHandle->setNextHandle(NULL);

    headHandle->HandleCar();

    /*
    //业务逻辑 写死在客户端了
    headHandle->HandleCar();
    bodyHandle->HandleCar();
    tailHandle->HandleCar();
    */

    delete headHandle;
    delete bodyHandle;
    delete tailHandle;

    system("pause");
    return ;
}
```



```
#include<iostream>
#include <string>
using namespace std;

class Request  //请求
{
public:
    string requestType;
    string requestContent;
    int number;
};

class Manager  ///管理者
{
protected:
    string name;
    Manager *superior;
public:
    Manager(string name)
    {
        this->name = name;
    }
    void setSuperior(Manager *superior)
    {
        this->superior = superior;
    }
    virtual void requestApplications(Request *request) = 0;
};

class CommonManager :public Manager  //经理
{
public:
    CommonManager(string name) :Manager(name)
    {}
    void requestApplications(Request *request)
    {
        if (request->requestType == "请假" && request->number <= 2
)
        {
```

```
            cout << name << " " << request->requestContent << "
数量: "
              << request->number << "被批准" << endl;
        }
        else
        {
            if (superior != NULL)
            {
                superior->requestApplications(request);
            }
        }
    }
};

class Majordomo :public Manager  //总监
{
public:
    Majordomo(string name) :Manager(name)
    {}
    void requestApplications(Request *request)
    {
        if (request->requestType == "请假" && request->number <= 5
)
        {
            cout << name << " " << request->requestContent << "
数量: "
              << request->number << "被批准" << endl;
        }
        else
        {
            if (superior != NULL)
            {
                superior->requestApplications(request);
            }
        }
    }
};


class GeneralManager :public Manager //总经理
```

```
{
public:
    GeneralManager(string name) :Manager(name)
    {}
    void requestApplications(Request *request)
    {
        if (request->requestType == "请假")
        {
            cout << name << " " << request->requestContent << "
数量: "
              << request->number << "被批准" << endl;
        }
    }
};


int main123213123213()
{
    CommonManager *jinli = new CommonManager("经理");
    Majordomo *zongjian = new Majordomo("总监");
    GeneralManager *zhongjingli = new GeneralManager("总经理");

    jinli->setSuperior(zongjian);
    zongjian->setSuperior(zhongjingli);

    Request *request = new Request();

    request->requestType = "请假";
    request->requestContent = "李俊宏请假";
    request->number = 1;
    jinli->requestApplications(request);


    request->requestType = "请假";
    request->requestContent = "李俊宏请假";
    request->number = 4;
    jinli->requestApplications(request);


    request->requestType = "请假";
```

```
    request->requestContent = "李俊宏请假";
    request->number = 10;
    jinli->requestApplications(request);

    cin.get();
    return 0;
}
```

迭代器模式

迭代器模式提供了一种顺序访问聚合对象中各个元素的方法，隐藏了对象的内部结构，为遍历不同聚合结构提供了如“开始”、“下一个”、“是否遍历结束”、“当前是哪个元素”等统一的操作接口。

Iterator模式也叫迭代模式，是行为模式之一，它把对容器中包含的内部对象的访问委让给外部类，使用Iterator（遍历）按顺序进行遍历访问的设计模式。

在应用Iterator模式之前，首先应该明白Iterator模式用来解决什么问题。或者说，如果不使用Iterator模式，会存在什么问题。

- 由容器自己实现顺序遍历。直接在容器类里直接添加顺序遍历方法
- 让调用者自己实现遍历。直接暴露数据细节给外部。

以上方法1与方法2都可以实现对遍历，这样有问题呢？

- 容器类承担了太多功能：一方面需要提供添加删除等本身应有的功能；一方面还需要提供遍历访问功能。
- 往往容器在实现遍历的过程中，需要保存遍历状态，当跟元素的添加删除等功能夹杂在一起，很容易引起混乱和程序运行错误等。

Iterator模式就是为了有效地处理按顺序进行遍历访问的一种设计模式，简单地说，Iterator模式提供一种有效的方法，可以屏蔽聚集对象集合的容器类的实现细节，而能对容器内包含的对象元素按顺序进行有效的遍历访问。所以，Iterator模式的应用场景可以归纳为满足以下几个条件：

- 访问容器中包含的内部对象
- 按顺序访问

迭代子模式：迭代子模式可以顺序访问一个聚集中的元素而不必暴露聚集的内部表象。

多个对象聚在一起形成的总体称之为聚集，聚集对象是能够包容一组对象的容器对象。

迭代子模式将迭代逻辑封装到一个独立的子对象中，从而与聚集本身隔开。

迭代子模式简化了聚集的界面。

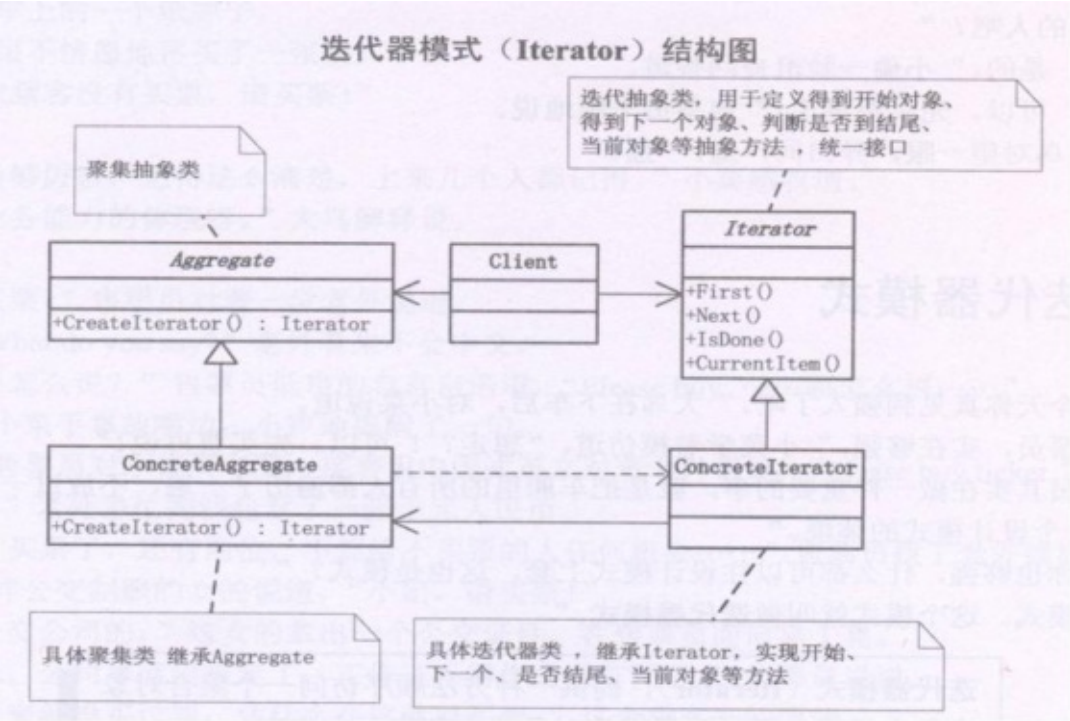
每一个聚集对象都可以有一个或一个以上的迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。迭代算法可以独立于聚集角色变化。

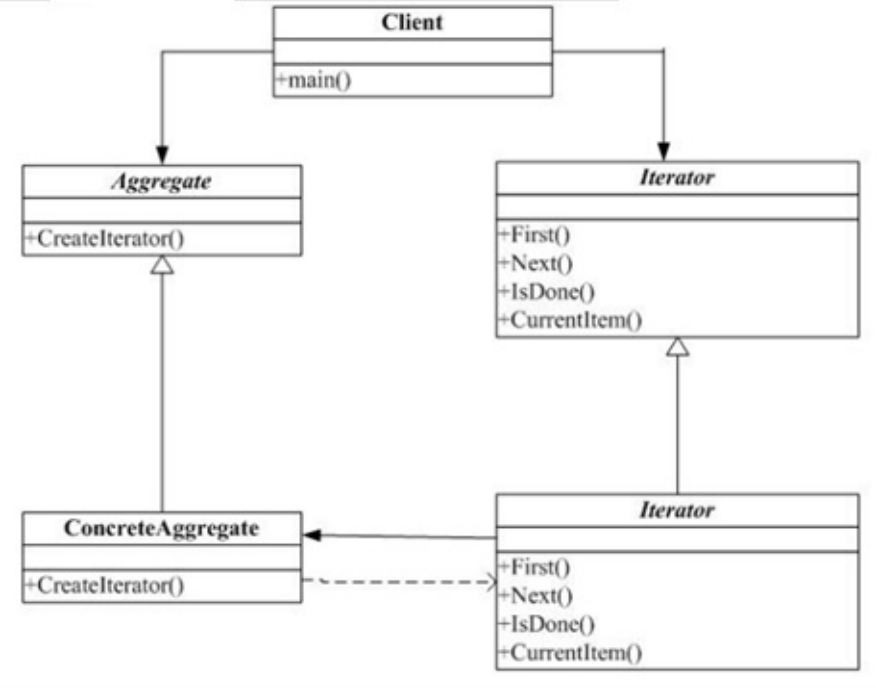
- 我爱上了Mary，不顾一切的向她求婚。
- Mary：“想要我跟你结婚，得答应我的条件”
- 我：“什么条件我都答应，你说吧”
- Mary：“我看上了那个一克拉的钻石”我：“我买，我买，还有吗？”
- Mary：“我看上了湖边的那栋别墅”我：“我买，我买，还有吗？”
- Mary：“我看上那辆法拉利跑车”
- 我脑袋嗡的一声，坐在椅子上，一咬牙：“我买，我买，还有吗？”.....

类图角色和职责

GOOD：提供一种方法顺序访问一个聚敛对象的各个元素，而又不暴露该对象的内部表示。

为遍历不同的聚集结构提供如开始，下一个，是否结束，当前一项等统一接口。





Iterator（迭代器接口）：该接口必须定义实现迭代功能的最小定义方法集，比如提供hasNext()和next()方法。

ConcreteIterator（迭代器实现类）：迭代器接口Iterator的实现类。可以根据具体情况加以实现。

Aggregate（容器接口）：定义基本功能以及提供类似Iterator iterator()的方法。

concreteAggregate（容器实现类）：容器接口的实现类。必须实现Iterator iterator()方法。

在迭代器中 持有 一个集合的 引用；所以通过迭代器，就可以访问集合

示例代码

```
#include <iostream>
using namespace std;
typedef int Object ;
#define SIZE 5

class MyIterator
{
public:
    virtual void First() = 0;
    virtual void Next() = 0;
```

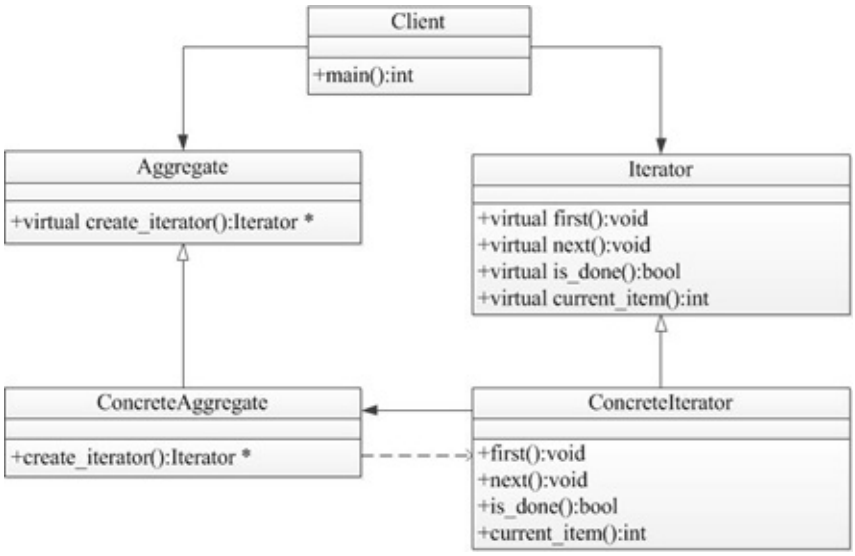
```
    virtual bool IsDone() = 0;
    virtual Object CurrentItem() = 0;
};

class Aggregate
{
public:
    virtual MyIterator *CreateIterator() = 0;
    virtual Object getItem(int index) = 0;
    virtual int getSize() = 0;
};

class ContreteIterator : public MyIterator
{
public:
    ContreteIterator(Aggregate *ag)
    {
        _ag = ag;
        _current_index = 0;
    }

    virtual void First()
    {
        _current_index = 0;  //让当前 游标 回到位置0
    }

    virtual void Next()
    {
        if (_current_index < _ag->getSize())
        {
            _current_index ++;
        }
    }

    virtual bool IsDone()
    {
        return  (_current_index ==  _ag->getSize());
    }

    virtual Object CurrentItem()
```

```
    {
        return _ag->getItem(_current_index);
    }
private:
    int            _current_index;
    Aggregate    *_ag;
};

class ConcreteAggregate : public Aggregate
{
public:
    ConcreteAggregate()
    {
        for (int i=0; i<SIZE; i++)
        {
            object[i] = i + 100;
        }
    }

    virtual MyIterator *CreateIterator()
    {
        return new ContreteIterator(this); //让迭代器 持有一个 集合的 引用
    }

    virtual Object getItem(int index)
    {
        return object[index];
    }

    virtual int getSize()
    {
        return SIZE;
    }
private:
    Object object[SIZE];
};

void main()
{
```

```
    Aggregate * ag = new ConcreteAggregate;

    MyIterator *it = ag->CreateIterator();

    for (; !(it->IsDone()); it->Next() )
    {
        cout << it->CurrentItem() << " ";
    }

    delete it;
    delete ag;

    system("pause");
    return ;
}
```

```
#include <iostream>
#include <string>
using namespace std;

class Iterator;

class Aggregate
{
public:
    virtual Iterator *createIterator() = 0;
};

class Iterator
{
public:
    virtual void first() = 0;
    virtual void next() = 0;
    virtual bool isDone() = 0;
    virtual bool isDoneA() = 0;
    //virtual bool isDoneA() = 0;
};

class ConcreteAggregate :public Iterator
```

```
{
public:
    void first(){}

    void next(){}

    bool isDone(){}

    virtual bool isDoneA(){}

};

int main()
{
    cin.get();
    return 0;
}
```



- 抽象聚合类型：负责提供创建具体迭代器角色的接口
- 具体聚合类型：实现创建具体迭代器
- 抽象迭代器：负责定义访问和遍历元素的接口
- 具体迭代器角色：实现了迭代器接口

```
//iterator.h
#ifndef ITERATOR_H
#define ITERATOR_H
class Aggregate;
typedef int Object;
class Iterator{//定义抽象迭代器类
public:
    virtual ~Iterator();
    virtual void first() = 0;
    virtual void next() = 0;
    virtual bool is_done() = 0;
    virtual Object current_item() = 0;
protected:
    Iterator();
};
//定义抽象迭代器的派生类，表示一个实际可操作的迭代器
class ConcreteIterator :public Iterator{
public:
    //声明迭代器的构造函数，首参数为要进行迭代的聚合类型指针
    ConcreteIterator(Aggregate *ag, int idx = 0);
    ~ConcreteIterator();
    void first();
    void next();
    bool is_done();
    Object current_item();
private:
    Aggregate *m_ag;
    int m_idx;
};
#endif
```

```
//iterator.cpp
#include <iostream>
#include "aggregate.h"
#include "iterator.h"
using namespace std;

Iterator::Iterator() //定义抽象迭代器的构造函数
```

```
{
}
Iterator::~Iterator() //定义抽象迭代器的析构函数
{
}
//定义具体迭代器类的构造函数，首参数表示对哪个聚合类型数据进行迭代，第二个参数表示从哪里开始迭代
ConcreteIterator::ConcreteIterator(Aggregate *ag, int idx)
{
    m_ag = ag;
    m_idx = idx;
}
ConcreteIterator::~ConcreteIterator() //定义具体迭代器的析构函数
{
}
Object ConcreteIterator::current_item() //定义获取聚合数据中当前对象的函数
{
    return m_ag->get_item(m_idx);
}
//定义设置聚合数据首位置的函数
void ConcreteIterator::first()
{
    m_idx = 0;
}
//定义设置聚合数据中下一个位置的函数
void ConcreteIterator::next()
{
    if (m_idx < m_ag->get_size())
        m_idx++;
}
//定义判断是否迭代完毕的函数
bool ConcreteIterator::is_done()
{
    return (m_idx == m_ag->get_size());
}
```

```
//aggregate.h
#ifndef AGGREGATE_H
#define AGGREGATE_H
//把int类型更名为Object，本聚合类型中的对象为int类型
typedef int Object;
class Iterator;  //声明迭代器类型
class Aggregate{ //定义抽象聚合类型
public:
    virtual ~Aggregate();
    virtual Iterator *create_iterator() = 0; //声明生成迭代器的接口
    virtual Object get_item(int idx) = 0;
    virtual int get_size() = 0;
protected:
    Aggregate();
};
//定义抽象聚合类的派生类，表示具体的某个聚合类型
class ConcreteAggregate :public Aggregate{
public:
    //定义枚举数据，表示聚合中有效数据的个数
    enum{ SIZE = 5 };
    ConcreteAggregate();
    ~ConcreteAggregate();
    Iterator *create_iterator();
    Object get_item(int idx);
    int get_size();
private:
    Object m_objs[SIZE]; //聚合类型中的具体数据
};
#endif
```

```
//aggregate.cpp
#include <iostream>
#include "aggregate.h"
#include "iterator.h"
using namespace std;

Aggregate::Aggregate()                    //定义抽象聚合类型的构造函数
{
}
Aggregate::~Aggregate()                   //定义抽象聚合类型的析构函数
{
}
ConcreteAggregate::ConcreteAggregate() //定义具体聚合类型的构造函数
{
    //向聚合类型中填充具体数据，数量为SIZE（5）个，保存在int类型的数组中
    for (int i = 0; i < SIZE; i++)
        m_objs[i] = i;
}
ConcreteAggregate::~ConcreteAggregate()//定义具体聚合类型的析构函数
{
}
//定义生成迭代器的函数
Iterator *ConcreteAggregate::create_iterator()
{
    return new ConcreteIterator(this);
}
//定义从聚合类型中获取具体元素的函数
Object ConcreteAggregate::get_item(int idx)
{
    if (idx < this->get_size())
        return m_objs[idx];
    return -1;
}
int ConcreteAggregate::get_size()//定义获取聚合中有效数据个数的函数
{
    return SIZE;
}
```

```
#include <iostream>
#include "aggregate.h"
#include "iterator.h"
using namespace std;

int main()
{
    //定义一个具体的聚合类型，由ag指向
    Aggregate *ag = new ConcreteAggregate();
    //定义迭代器it，操作ag指向的聚合类型数据
    Iterator *it = new ConcreteIterator(ag);
    //使用迭代器依次操作聚合类型中的对象
    for (; !(it->is_done()); it->next()){
        cout << it->current_item() << endl;
    }
    system("pause");
    return 0;
}
```

外观模式

Facade模式也叫外观模式，是由GoF提出的23种设计模式中的一种。Facade模式为一组具有类似功能的类群，比如类库，子系统等等，提供一个一致的简单的界面。这个一致的简单的界面被称作facade。

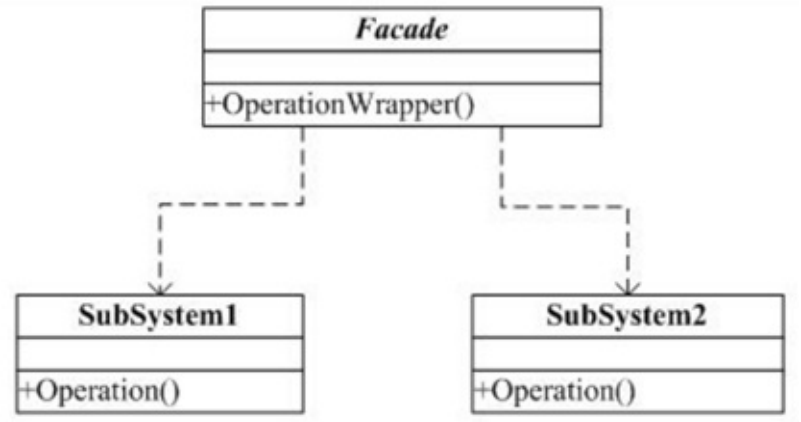
门面模式：外部与一个子系统的通信必须通过一个统一的门面对象进行。

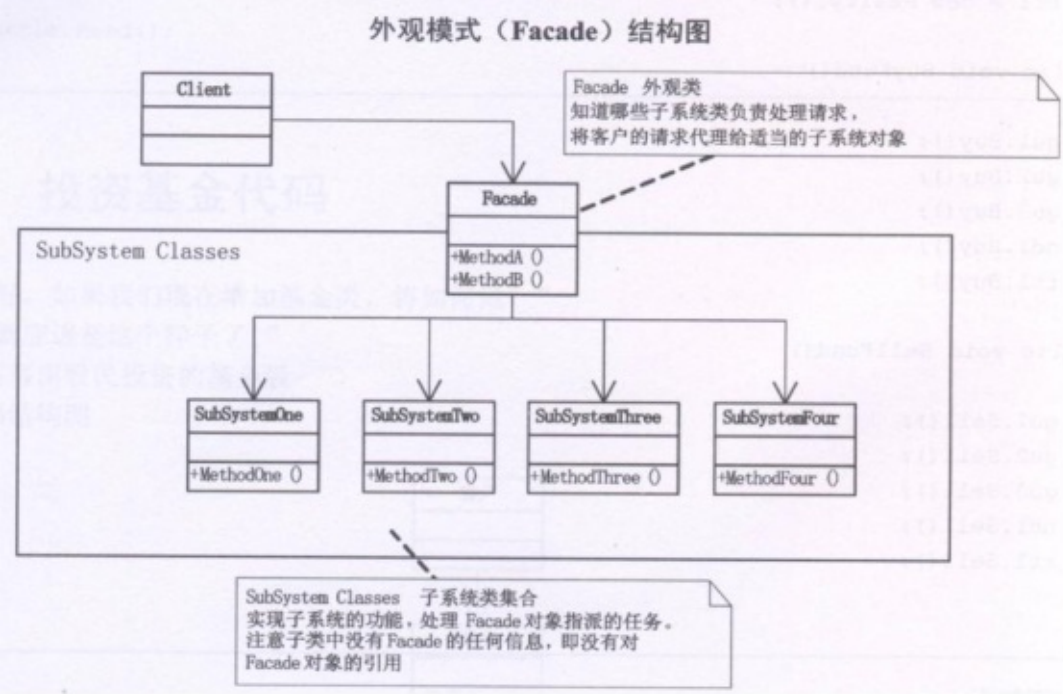
门面模式提供一个高层次的接口，使得子系统更易于使用。

每一个子系统只有一个门面类，而且此门面类只有一个实例，也就是说它是一个单例模式。但整个系统可以有多个门面类。

我有一个专业的Nikon相机，我就喜欢自己手动调光圈、快门，这样照出来的照片才专业，但MM可不懂这些，教了半天也不会。幸好相机有Facade设计模式，把相机调整到自动档，只要对准目标按快门就行了，一切由相机自动调整，这样MM也可以用这个相机给我拍张照片了。

类图角色和职责





- Façade：为调用方, 定义简单的调用接口。
- Clients：调用者。通过Facade接口调用提供某功能的内部类群。
- Packages：功能提供者。指提供功能的类群（模块或子系统）

适用于：为子系统中统一一套接口，让子系统更加容易使用。

示例代码

```
#include <iostream>
using namespace std;

class SubSystemA
{
public:
    void doThing()
    {
        cout << "SubSystemA run" << endl;
    }
};

class SubSystemB
{
public:
    void doThing()
```

```
    {
        cout << "SubSystemB run" << endl;
    }
};

class SubSystemC
{
public:
    void doThing()
    {
        cout << "SubSystemC run" << endl;
    }
};

class Facade
{
public:
    Facade()
    {
        sysA = new SubSystemA;
        sysB = new SubSystemB;
        sysC = new SubSystemC;
    }
    ~Facade()
    {
        delete sysA;
        delete sysB;
        delete sysC;
    }
public:
    void doThing()
    {
        sysA->doThing();
        sysB->doThing();
        sysC->doThing();
    }
private:
    SubSystemA *sysA;
    SubSystemB *sysB;
    SubSystemC *sysC;
```

```
};

void main1()
{
    SubSystemA *sysA = new SubSystemA;
    SubSystemB *sysB = new SubSystemB;
    SubSystemC *sysC = new SubSystemC;

    sysA->doThing();
    sysB->doThing();
    sysC->doThing();
    delete sysA;
    delete sysB;
    delete sysC;

    return ;
}

void main2()
{
    Facade *f = new Facade;
    f->doThing();
    delete f;
}

void main()
{
    //main1();
    main2();
    system("pause");
}
```

```
#include <iostream>
#include <string>
using namespace std;

class Sub1
{
public:
```

```
    void f1()
    {
        cout << "子系统的方法 1" << endl;
    }
};

class Sub2
{
public:
    void f2()
    {
        cout << "子系统的方法 2" << endl;
    }
};

class Sub3
{
public:
    void f3()
    {
        cout << "子系统的方法 3" << endl;
    }
};

class Facade
{
private:
    Sub1 *s1;
    Sub2 *s2;
    Sub3 *s3;
public:
    Facade()
    {
        s1 = new Sub1();
        s2 = new Sub2();
        s3 = new Sub3();
    }
    void method()
    {
        s1->f1();
```

```
        s2->f2();
        s3->f3();
    }
};

int main()
{
    Facade *f = new Facade();
    f->method();
    cin.get();
    return 0;
}
```

代理模式

Proxy模式又叫做代理模式，是构造型的设计模式之一，它可以为其他对象提供一种代理（Proxy）以控制对这个对象的访问。

所谓代理，是指具有与代理元（被代理的对象）具有相同的接口的类，客户端必须通过代理与被代理的目标类交互，而代理一般在交互的过程中（交互前后），进行某些特别的处理。

让类与类进行组合，获取更大的结构

代理模式：代理模式给某一个对象提供一个代理对象，并由代理对象控制对源对象的引用。

代理就是一个人或一个机构代表另一个人或者一个机构采取行动。

某些情况下，客户不想或者不能够直接引用一个对象，代理对象可以在客户和目标对象直接起到中介的作用。

客户端分辨不出代理主题对象与眞实主题对象。

代理模式可以并不知道眞正的被代理对象，而仅仅持有一个被代理对象的接口，这时候代理对象不能够创建被代理对象，
被代理对象必须有系统的其他角色代为创建并传入。

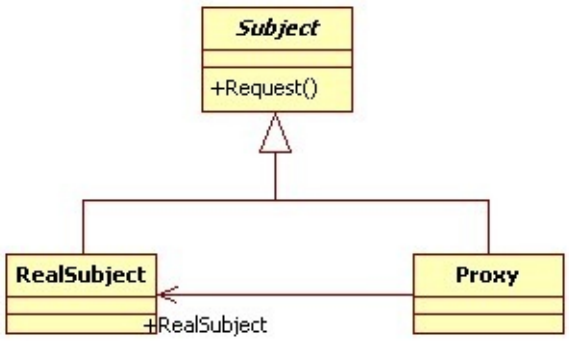
跟MM在网上聊天，一开头总是“hi, 你好”, “你从哪儿来呀？”“你多大了？”“身高多少呀？”这些话，眞烦人，写个程序做为我的Proxy吧，凡是接收到这些话都设置好了自己的回答，接收到其他的话时再通知我回答，怎么样，酷吧。

类图角色和职责

- subject（抽象主题角色）：眞实主题与代理主题的共同接口。
- RealSubject（眞实主题角色）：定义了代理角色所代表的眞实对象。
- Proxy（代理主题角色）：含有对眞实主题角色的引用，代理角色通常在将客户端调用传递给眞是主题对象之前或者之后执行某些操作，而不是单纯返回眞实的对象。

适合于：为其他对象提供一种代理以控制对这个对象的访问。

提示：a中包含b类；a、b类实现协议类protocol



示例代码

理论模型

```
#include <string>
#include <iostream>
using namespace std;

//定义接口
class Interface
{
public:
    virtual void Request()=0;
```

```
};

//真实类
class RealClass : public Interface
{
public:
    virtual void Request()
    {
        cout<<"真实的请求"<<endl;
    }
};

//代理类
class ProxyClass : public Interface
{
private:
    RealClass* m_realClass;
public:
    virtual void Request()
    {
        m_realClass= new RealClass();
        m_realClass->Request();
        delete m_realClass;
    }
};

// 客户端
int main()
{
    ProxyClass* test=new ProxyClass();
    test->Request();
    return 0;
}
```

cocos2d-x中应用程序代理类

```
#include "iostream"
using namespace std;
```

```
//a包含了一个类b，类b实现了某一个协议（一套接口）
class AppProtocol
{
public:
    virtual int ApplicationDidFinsh() = 0;
};

//协议实现类
class AppDelegate : public AppProtocol
{
public:
    AppDelegate() { }
    virtual int ApplicationDidFinsh()  //cocos2dx函数的入口点
    {
        cout<<"ApplicationDidFinsh do...\n";
        return 0;
    }
};

//Application是代理类，在代理类中包含一个真正的实体类
class Application
{
public:
    Application()
    {
        ap = NULL;
    }
public:
    void run()
    {
        ap = new AppDelegate();
        ap->ApplicationDidFinsh();
        delete ap;
    }
private:
    AppDelegate *ap;
};

//好处：main函数不需要修改了。只需要修改协议实现类
void main31()
```

```
{
    Application *app = new Application();
    app->run();

    if (app == NULL)
    {
        free(app);
    }

    system("pause");
}
```

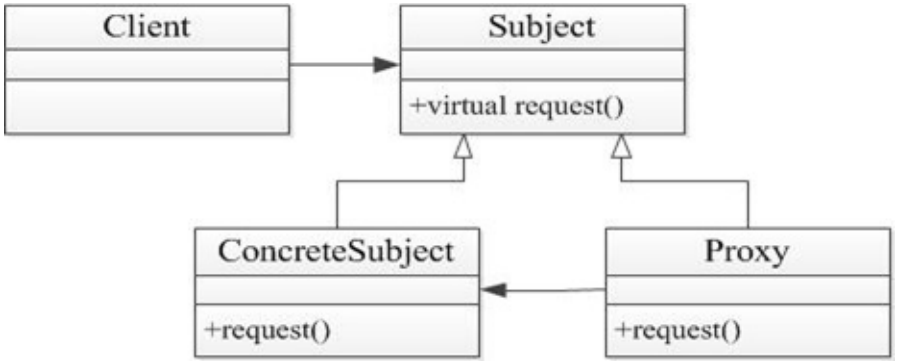
```
#include <iostream>
using namespace std;

class Subject
{
public:
    virtual void sailbook() = 0;
};

class RealSubjectBook : public Subject
{
public:
    virtual void sailbook()
    {
        cout << "卖书" << endl;
    }
};

//a中包含b类；a、b类实现协议类protocol
class dangdangProxy : public Subject
{
public:
    virtual void sailbook()
    {
        RealSubjectBook *rsb = new RealSubjectBook;
        dazhe();
        rsb->sailbook();
```

```
        dazhe();
    }
public:
    void dazhe()
    {
        cout << "双十一打折" << endl;
    }
private:
    Subject *m_subject;
};

void main()
{
    Subject *s = new dangdangProxy;
    s->sailbook();
    delete s;
    system("pause");
    return ;
}
```

```
#include <iostream>
#include <string>
using namespace std;

class SchoolGirl
{
public:
    string name;
};

class IGiveGift
{
public:
    virtual void giveDolls() = 0;
    virtual void giveFlowers() = 0;
};

class Pursuit :public IGiveGift
{
```

```
private:
    SchoolGirl mm;
public:
    Pursuit(SchoolGirl m)
    {
        mm = m;
    }
    void giveDolls()
    {
        cout << mm.name << " 送你娃娃" << endl;
    }
    void giveFlowers()
    {
        cout << mm.name << " 送你鲜花" << endl;
    }
};

class Proxy :public IGiveGift
{
private:
    Pursuit gg;
public:
    Proxy(SchoolGirl mm) :gg(mm)
    {
        //gg=g;
    }
    void giveDolls()
    {
        gg.giveDolls();
    }
    void giveFlowers()
    {
        gg.giveFlowers();
    }
};

int main()
{
    SchoolGirl lijiaojiao;
    lijiaojiao.name = "李娇娇";
```

```
    Pursuit zhuojiayi(lijiaojiao);
    Proxy daili(lijiaojiao);

    daili.giveDolls();
    cin.get();
    return 0;
}
```



- 抽象主体：声明了真实主体和代理主体的共同接口
- 真实主体：定义了代理所代表的真实对象
- 代理主体：通过在其中定义真实主体的引用，实现对真实主体的操作

```
//proxy.h
#ifndef PROXY_H
#define PROXY_H

class Subject{ //定义抽象主体类
public:
    virtual ~Subject();
    virtual void request() = 0;
protected:
    Subject();
};

//定义抽象主体类的派生类，描述一个具体的主体
class ConcreteSubject :public Subject{
public:
    ConcreteSubject();
    ~ConcreteSubject();
    void request();
};
class Proxy :public Subject{ //定义代理类
public:
    Proxy();
    Proxy(Subject *sub);
    ~Proxy();
    void request();
private:
    Subject *m_sub; //定义一个指向主体的指针
};
#endif
```

```cpp
//proxy.cpp
#include <iostream>
#include "proxy.h"
using namespace std;

Subject::Subject() //定义抽象主体类的构造函数
{
}
Subject::~Subject() //定义抽象主体类的析构函数
{
}
ConcreteSubject::ConcreteSubject() //定义真实主体类的构造函数
{
}
ConcreteSubject::~ConcreteSubject() //定义真实主体类的析构函数
{
}
void ConcreteSubject::request() //真实主体类要完成的操作
{
    cout << "ConcreteSubject request!" << endl;
}

Proxy::Proxy() //定义代理类构造函数
{
}
//代理类构造函数，参数为某个主体的指针
Proxy::Proxy(Subject *sub)
{
    m_sub = sub;
}
Proxy::~Proxy() //定义代理类析构函数
{
}
void Proxy::request() //通过代理实现主体需要完成的操作
{
    cout << "Proxy request!" << endl;
    m_sub->request();
}
```

```
#include <iostream>
#include "proxy.h"
using namespace std;

int main()
{
    //sub指针指向一个真实主体对象
    Subject *sub = new ConcreteSubject();
    Proxy *p = new Proxy(sub); //定义代理对象
    p->request();//通过代理完成主体需要的操作
    system("pause");
    return 0;
}
```

享元模式

Flyweight模式也叫享元模式，是构造型模式之一，它通过与其他类似对象共享数据来减小内存占用。

在面向对象系统的设计何实现中，创建对象是最为常见的操作。这里面就有一个问题：如果一个应用程序使用了太多的对象，就会造成很大的存储开销。特别是对于大量轻量级（细粒度）的对象，比如在文档编辑器的设计过程中，我们如果为没有字母创建一个对象的话，系统可能会因为大量的对象而造成存储开销的浪费。例如一个字母“a”在文档中出现了100000次，而实际上我们可以让这一万个字母“a”共享一个对象，当然因为在不同的位置可能字母“a”有不同的显示效果（例如字体和大小等设置不同），在这种情况我们可以为将对象的状态分为“外部状态”和“内部状态”，将可以被共享（不会变化）的状态作为内部状态存储在对象中，而外部对象（例如上面提到的字体、大小等）我们可以在适当的时候将外部对象最为参数传递给对象（例如在显示的时候，将字体、大小等信息传递给对象）。

享元模式：FLYWEIGHT在拳击比赛中指最轻量级。

享元模式以共享的方式高效的支持大量的细粒度对象。

享元模式能做到共享的关键是区分内蕴状态和外蕴状态。

内蕴状态存储在享元内部，不会随环境的改变而有所不同。

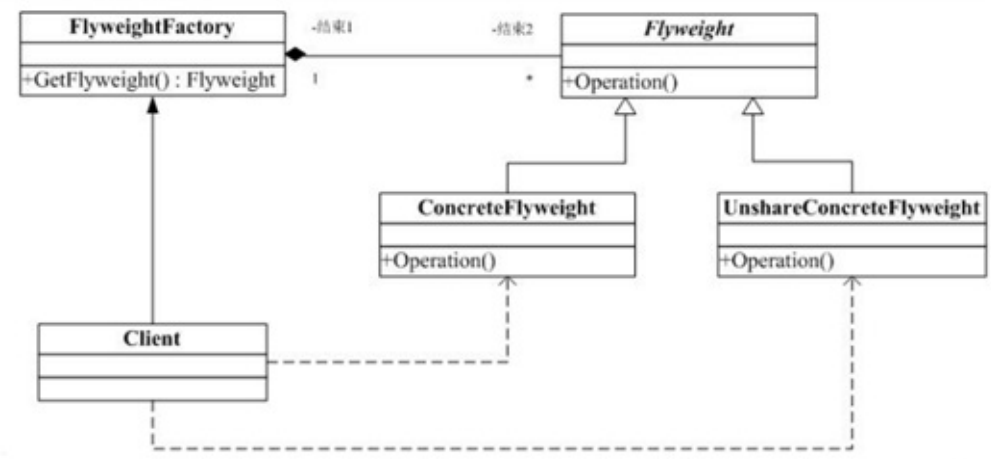
外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态，它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来，将不可以共享的状态从类里剔除出去。客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。

享元模式大幅度的降低内存中对象的数量。

每天跟MM发短信，手指都累死了，最近买了个新手机，可以把一些常用的句子存在手机里，要用的时候，直接拿出来，在前面加上MM的名字就可以发送了，再不用一个字一个字敲了。

共享的句子就是Flyweight，MM的名字就是提取出来的外部特征，根据上下文情况使用。

类图角色和职责



- 抽象享元角色：所有具体享元类的父类，规定一些需要实现的公共接口。
- 具体享元角色：抽象享元角色的具体实现类，并实现了抽象享元角色规定的方法。
- 享元工厂角色：负责创建和管理享元角色。
- 使用场景：是以共享的方式，高效的支持大量的细粒度的对象。

GOOD：运用共享技术有效地支持大量细粒度的对象（对于C++来说就是共用一个内存块啦，对象指针指向同一个地方）。

如果一个应用程序使用了大量的对象，而这些对象造成了很大的存储开销就应该考虑使用。

还有就是对象的大多数状态可以外部状态，如果删除对象的外部状态，那么可以用较少的共享对象取代多组对象，此时可以考虑使用享元。

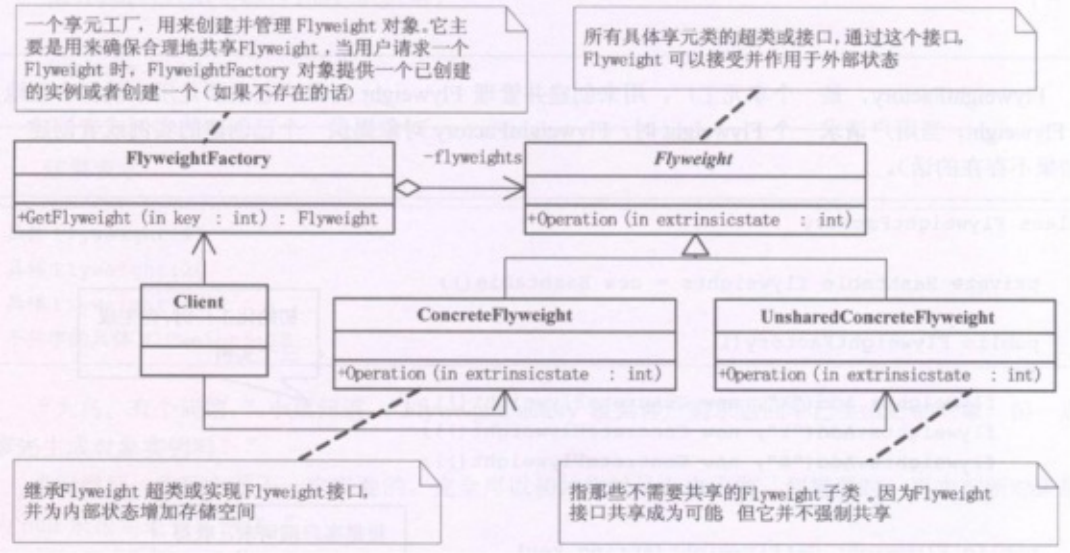
示例代码

```
#include <iostream>
#include "string"
#include "map"
using namespace std;

class Person
{
public:
    Person(string name, int age)
```

```
    {
        this->m_name = name;
        this->age = age;
    }
    virtual void printT() = 0;

protected:
    string    m_name;
    int        age;
};

class Teacher : public Person
{
public:
    Teacher(string name, int age, string id) : Person(name, age)
    {
        this->m_id = id;
    }

    void printT()
    {
        cout << "name:" << m_name << " age:" << age << " m_id:"
<< m_id << endl;
     }
private:
    string    m_id;
};

//完成老师结点存储
class FlyWeightTeacherFactory
{
public:
    FlyWeightTeacherFactory()
    {
        map1.clear();
    }

    ~FlyWeightTeacherFactory()
    {
        while ( !map1.empty())
```

```
        {
            Person *tmp = NULL;
            map<string, Person *>::iterator it = map1.begin();
            tmp = it->second;
            map1.erase(it); //把第一个结点从容器中删除
            delete tmp;
        }
    }

    Person * GetTeacher(string id)
    {
        Person *tmp = NULL;
        map<string, Person *>::iterator it ;
        it = map1.find(id);
        if (it == map1.end()) //没有找到
        {
            string    tmpname;
            int        tmpage;
            cout << "\n请输入老师name:";
            cin >> tmpname;

            cout << "\n请输入老师age:";
            cin >> tmpage;

            tmp = new Teacher(tmpname, tmpage, id);
            map1.insert(pair<string, Person*>(id, tmp) );
        }
        else
        {
            tmp = it->second;
        }
        return tmp;
    }
private:
    map<string, Person *> map1;
};

void main()
{
    Person *p1 = NULL;
```

```
    Person *p2 = NULL;
    FlyWeightTeacherFactory *fwtf = new FlyWeightTeacherFactory;
    p1 = fwtf->GetTeacher("001");
    p1->printT();

    p2 = fwtf->GetTeacher("001");
    p2->printT();

    delete fwtf;

    system("pause");
    return ;
}
```




```
#include <iostream>
#include <list>
#include <string>
#include <map>
using namespace std;

class WebSite  //抽象的网站
{
public:
    virtual void use() = 0;//预留接口实现功能
};

class ConcreteWebSite :public WebSite  //具体的共享网站
```

```
{
private:
    string name;
public:
    ConcreteWebSite(string name)//实例化
    {
        this->name = name;
    }
    void use()
    {
        cout << "网站分类: " << name << endl;
    }
};

//不共享的网站
class UnShareWebSite : public WebSite
{
private:
    string name;
public:
    UnShareWebSite(string strName)
    {
        name = strName;
    }
    virtual void Use()
    {
        cout<<"不共享的网站："<<name<<endl;
    }
};

class WebSiteFactory  //网站工厂类，用于存放共享的WebSite对象
{
private:
    map<string, WebSite*> wf;
public:

    WebSite *getWebSiteCategory(string key)
    {

        if (wf.find(key) == wf.end())
```

```
        {
            wf[key] = new ConcreteWebSite(key);
        }

        return wf[key];
    }

    int getWebSiteCount()
    {
        return wf.size();
    }
};

int main()
{
    WebSiteFactory *wf = new WebSiteFactory();

    WebSite *fx = wf->getWebSiteCategory("good");
    fx->use();

    WebSite *fy = wf->getWebSiteCategory("产品展示");
    fy->use();

    WebSite *fz = wf->getWebSiteCategory("产品展示");
    fz->use();


    WebSite *f1 = wf->getWebSiteCategory("博客");
    f1->use();

    WebSite *f2 = wf->getWebSiteCategory("博客");
    f2->use();

    cout << wf->getWebSiteCount() << endl;

      //不共享的类
    WebSite* ws3 = new UnShareWebSite("测试");
    ws3->Use();

    cin.get();
```

```
    return 0;
}
```

装饰者模式

装饰（ Decorator ）模式又叫做包装模式。通过一种对客户端透明的方式来扩展对象的功能，是继承关系的一个替换方案。

装饰模式就是把要添加的附加功能分别放在单独的类中，并让这个类包含它要装饰的对象，当需要执行时，客户端就可以有选择地、按顺序地使用装饰功能包装对象。

装饰模式：装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案，提供比继承更多的灵活性。

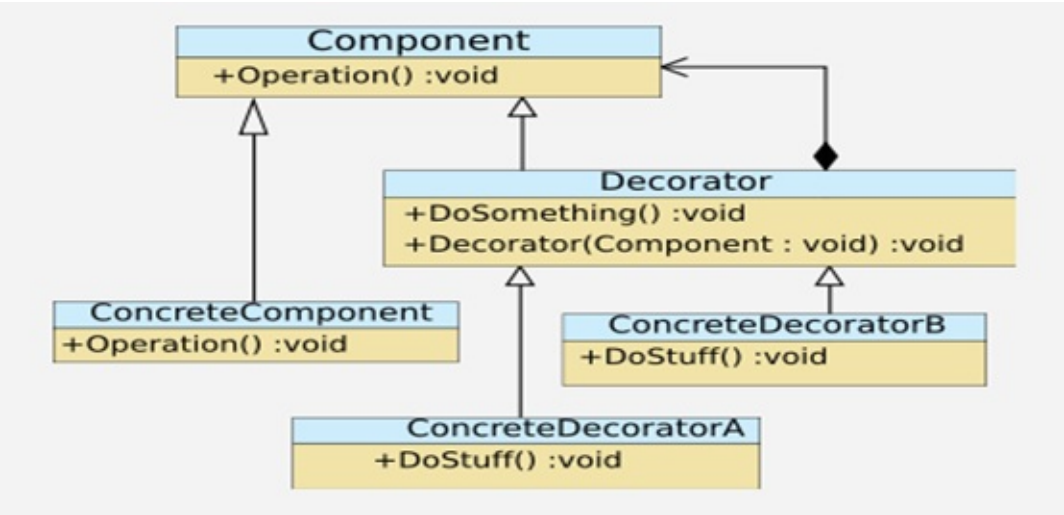
动态给一个对象增加功能，这些功能可以再动态的撤消。

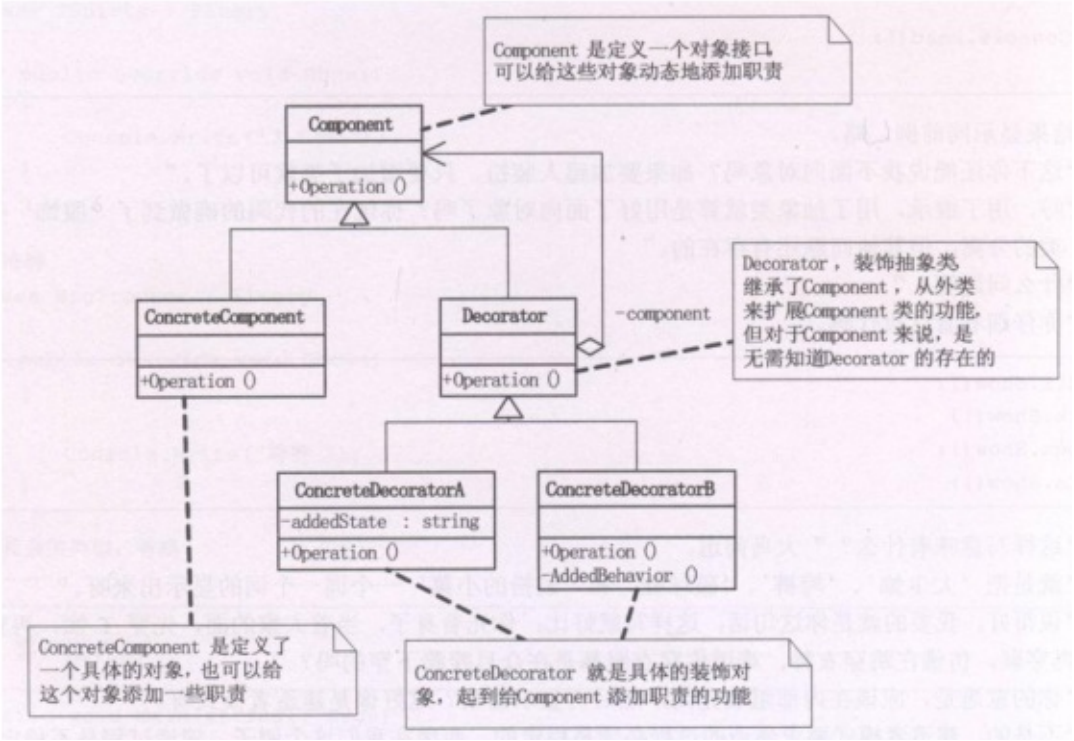
增加由一些基本功能的排列组合而产生的非常大量的功能。

Mary过完轮到Sarly过生日，还是不要叫她自己挑了，不然这个月伙食费肯定玩完，拿出我去年在华山顶上照的照片，在背面写上“最好的的礼物，就是爱你的Fita”，再到街上礼品店买了个像框（卖礼品的MM也很漂亮哦），再找隔壁搞美术设计的Mike设计了一个漂亮的盒子装起来……，我们都是Decorator，最终都在修饰我这个人呀，怎么样，看懂了吗？

类图角色和职责

适用于：装饰者模式（Decorator Pattern）动态的给一个对象添加一些额外的职责。就增加功能来说，此模式比生成子类更为灵活。





GOOD:当你向旧的类中添加新代码时，一般是为了添加核心职责或主要行为。而当需要加入的仅仅是一些特定情况下才会执行的特定的功能时（简单点就是不是核心应用的功能），就会增加类的复杂度。装饰模式就是把要添加的附加功能分别放在单独的类中，并让这个类包含它要装饰的对象，当需要执行时，客户端就可以有选择地、按顺序地使用装饰功能包装对象。

示例代码

```
#include <iostream>
using namespace std;

class Car
{
public:
    virtual void show() = 0;
};

class RunCar : public Car
{
public:
    virtual void show()
    {
```

```
        cout << "可以跑" << endl;
    }
};

class SwimCarDirector : public Car
{
public:
    SwimCarDirector(Car *car)
    {
        m_car = car;
    }

    void swim()
    {
        cout << "可以游" << endl;
    }

    virtual void show()
    {
        m_car->show();
        swim();
    }
private:
    Car *m_car;
};

class  FlyCarDirector : public Car
{
public:
    FlyCarDirector(Car *car)
    {
        m_car = car;
    }

    void fly()
    {
        cout << "可以飞" << endl;
    }

    virtual void show()
```

```
    {
        m_car->show();
        fly();
    }

private:
    Car *m_car;
};

void main()
{
    Car * mycar = NULL;
    mycar = new RunCar;
    printf("-----\n");
    mycar->show();

    printf("-----\n");

    FlyCarDirector *flycar = new FlyCarDirector(mycar);
    flycar->show();

    printf("-----\n");
    SwimCarDirector *swimcar = new SwimCarDirector(flycar);
    swimcar->show();

    delete swimcar;
    delete flycar;
    delete mycar;

    system("pause");
    return ;
}
```

```
#include <string>
#include <iostream>
using namespace std;

class Person
{
```

```
private:
    string m_strName;
public:
    Person(string strName)
    {
        m_strName = strName;
    }
    Person(){}
    virtual void show()
    {
        cout << "装扮的是:" << m_strName << endl;
    }
};

class Finery :public Person
{
protected:
    Person *m_component;
public:
    void decorate(Person* component)
    {
        m_component = component;
    }
    virtual void show()
    {
        m_component->show();
    }
};

class TShirts :public Finery
{
public:
    virtual void show()
    {
        m_component->show();
        cout << "T shirts" << endl;
    }
};

class BigTrouser :public Finery
```

```
{
public:
    virtual void show()
    {
        m_component->show();
        cout << "Big Trouser" << endl;
    }
};

int main()
{
    Person *p = new Person("小李");
    BigTrouser *bt = new BigTrouser();
    TShirts *ts = new TShirts();

    bt->decorate(p);
    ts->decorate(bt);
    ts->show();
    cin.get();
    return 0;
}
```

适配器模式

Adapter模式也叫适配器模式，是构造型模式之一，通过Adapter模式可以改变已有类（或外部类）的接口形式。

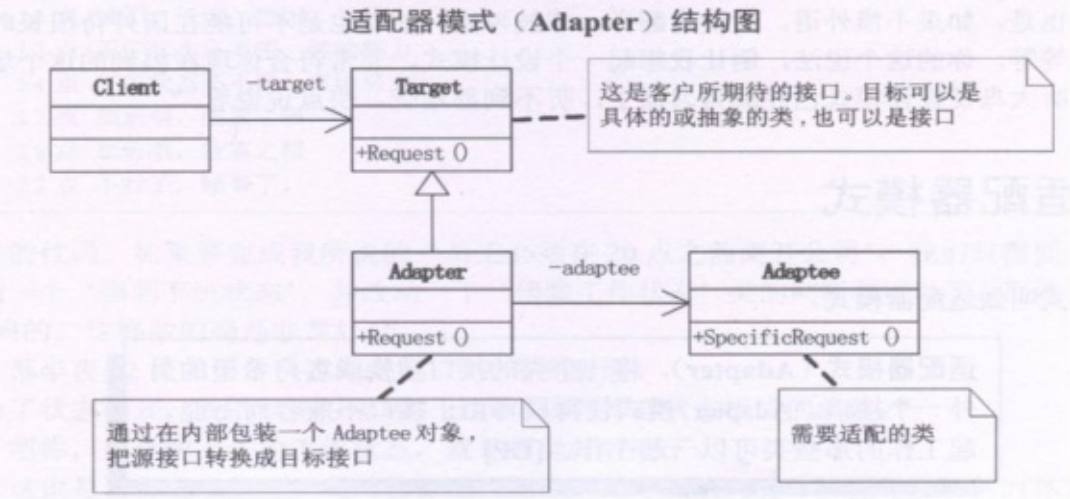
适配器（变压器）模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起工作。

适配类可以根据参数返还一个合适的实例给客户端。

在朋友聚会上碰到了一个美女Sarah，从香港来的，可我不会说粤语，她不会说普通话，只好求助于我的朋友kent了，他作为我和Sarah之间的Adapter，让我和Sarah可以相互交谈了(也不知道他会不会耍我)。

类图角色和职责

适用于：是将一个类的接口转换成客户希望的另外一个接口。使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。



示例代码

```
#include <iostream>
using namespace std;

// Current18
```

```
// Current220
// Adapter

class  Current18v
{
public:
    virtual void useCurrent18v() = 0;
};

class  Current220v
{
public:
    void useCurrent220v()
    {
        cout << "我是220v 欢迎使用" << endl;
    }
};

class Adapter : public Current18v
{
public:
    Adapter(Current220v *current)
    {
        m_current = current;
    }

    virtual void useCurrent18v()
    {
        cout << "适配器 适配 220v " ;
        m_current->useCurrent220v();
    }
private:
    Current220v *m_current;
};

void main()
{
    Current220v        *current220v = NULL;
    Adapter            *adapter = NULL;
```

```
    current220v = new Current220v;
    adapter = new Adapter(current220v);
    adapter->useCurrent18v();

    delete current220v ;
    delete adapter;

    system("pause");
    return ;
}
```

足球比赛中，中场球员可进攻和防守，教练通过翻译告诉中场球员，要进攻。

- Player：抽象的球员 （Attack、 Defense）
- class TransLater: public Player：适配器
- class Center : public Player：被适配的对象

```
#include <iostream>
#include <string>
using namespace std;

class Player
{
public:
    string name;
    Player(string name)
    {
        this->name = name;
    }
    virtual void attack() = 0;
    virtual void defence() = 0;
};

class Forwards :public Player
{
public:
    Forwards(string name) :Player(name){}
    void attack()
    {
```

```
        cout << name << " 前锋进攻" << endl;
    }
    void defence()
    {
        cout << name << " 前锋防守" << endl;
    }
};

class Center :public Player
{
public:
    Center(string name) :Player(name){}
    void attack()
    {
        cout << name << " 中锋进攻" << endl;
    }
    void defence()
    {
        cout << name << " 中锋防守" << endl;
    }
};

class Backwards :public Player
{
public:
    Backwards(string name) :Player(name){}
    void attack()
    {
        cout << name << " 后卫进攻" << endl;
    }
    void defence()
    {
        cout << name << " 后卫防守" << endl;
    }
};

class ForeignCenter
{
public:
    string name;
```

```
    ForeignCenter(string name)
    {
        this->name = name;
    }
    void myAttack()
    {
        cout << name << " 外籍中锋进攻" << endl;
    }
    void myDefence()
    {
        cout << name << " 外籍后卫防守" << endl;
    }
};

class Translator :public Player
{
private:
    ForeignCenter *fc;
public:
    Translator(string name) :Player(name)
    {
        fc = new ForeignCenter(name);
    }
    void attack()
    {
        fc->myAttack();
    }
    void defence()
    {
        fc->myDefence();
    }
};

int main()
{
    Player *p1 = new Center("李俊宏");
    p1->attack();
    p1->defence();

    Translator *tl = new Translator("姚明");
```

```
    tl->attack();
    tl->defence();
    cin.get();
    return 0;
}
```

策略模式

Strategy模式也叫策略模式是行为模式之一，它对一系列的算法加以封装，为所有算法定义一个抽象的算法接口，并通过继承该抽象算法接口对所有的算法加以封装和实现，具体的算法选择交由客户端决定（策略）。Strategy模式主要用来平滑地处理算法的切换 。

策略模式依赖于多态。

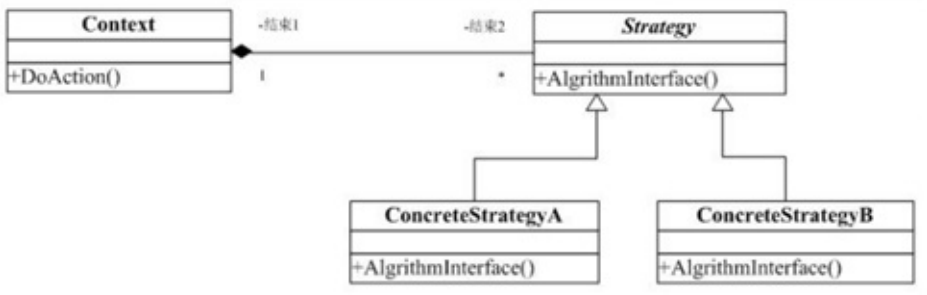
360服务端更新杀毒脚本进行客户端杀毒的操作。逻辑脚本存储在服务器，接口在客户端进行实现。

策略模式：策略模式针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。策略模把行为和环境分开。环境类负责维持和查询行为类，各种算法在具体的策略类中提供。由于算法和环境独立开来，算法的增减，修改都不会影响到环境和客户端。

跟不同类型的MM约会，要用不同的策略，有的请电影比较好，有的则去吃小吃效果不错，有的去海边浪漫最合适，单目的都是为了得到MM的芳心，我的追MM锦囊中有好多Strategy哦。

- 策略的抽象类，接口，抽象类的指针可以访问所有子类对象，（纯虚函数）
- 实现的各种策略，各种策略的实现类，都必须继承抽象类
- 策略的设置接口类，设置不同策略

类图角色和职责



这里的关键就是将算法的逻辑接口（DoAction）封装到一个类中（Context），再通过委托的方式将具体的算法委托给算法的具体的Strategy类来实现（ConcreteStrategyA类）。

- Strategy：策略（算法）抽象。
- ConcreteStrategy：各种策略（算法）的具体实现。
- Context：策略的外部封装类，或者说策略的容器类。根据不同策略执行不同的行为。策略由外部环境决定。

适用于：准备一组算法，并将每一个算法封装起来，使得它们可以互换。

策略模式优缺点

它的优点有：

- 策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免重复的代码。
- 策略模式提供了可以替换继承关系的办法。继承可以处理多种算法或行为。如果不是用策略模式，那么使用算法或行为的环境类就可能会有一些子类，每一个子类提供一个不同的算法或行为。但是，这样一来算法或行为的使用者就和算法或行为本身混在一起。决定使用哪一种算法或采取哪一种行为的逻辑就和算法或行为的逻辑混合在一起，从而不可能再独立演化。继承使得动态改变算法或行为变得不可能。
- 使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重转移语句里面，比使用继承的办法还要原始和落后。

策略模式的缺点有：

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道所有的算法或行为的情况。
- 策略模式造成很多的策略类。有时候可以通过把依赖于环境的状态保存到客户端里面，而将策略类设计成可共享的，这样策略类实例可以被不同客户端使用。换言之，可以使用享元模式来减少对象的数量。

示例代码

```
#include <iostream>
using namespace std;

class Strategy
{
public:
    virtual void crypt() = 0;
};

//对称加密：速度快，加密大数据块文件。特点:加密密钥和解密密钥是一样的
//非对称加密：加密速度慢，加密强度高，高安全性高;特点: 加密密钥和解密密钥不一样密钥对(公钥和私钥)

class AES :  public Strategy
{
public:
    virtual void crypt()
    {
        cout << "AES加密算法" << endl;
     }
};

class DES :  public Strategy
{
public:
    virtual void crypt()
    {
        cout << "DES 加密算法" << endl;
    }
};

class Context
{
public:
    void setStrategy(Strategy *strategy)
    {
        this->strategy = strategy;
```

```
    }

    void myoperator()
    {
        strategy->crypt();
    }

private:
    Strategy *strategy;
};

// 算法的实现和客户端的使用解耦合
// 使得算法变化，不会影响客户端
void main()
{
    /*
    DES *des = new DES;
    des->crypt();
    delete des;
    */

    Strategy *strategy = NULL;

    //strategy = new DES;
    strategy = new AES;
    Context *context = new Context;
    context->setStrategy(strategy);
    context->myoperator();

    delete  strategy;
    delete  context;

    system("pause");
    return ;
}
```

```
#include <iostream>
#include <cmath>
#include <string>
```

```
using namespace std;

class CashSuper
{
public:
    virtual double acceptMoney(double money) = 0;//抽象类，收钱的纯虚函数
};

class CashNormal :public CashSuper
{
public:
    double acceptMoney(double money)//正常收钱
    {
        return money;
    }
};

class CashRebate :public CashSuper //打折
{
private:
    double discount;
public:
    CashRebate(double dis) //折扣
    {
        discount = dis;
    }
    double acceptMoney(double money)//收钱
    {
        return money*discount;//折扣
    }
};

class CashReturn :public CashSuper
{
private:
    double moneyCondition;
    double moneyReturn;
public:
    CashReturn(double mc, double mr)//花多少钱，返回多少钱
```

```
    {
        moneyCondition = mc;
        moneyReturn = mr;
    }
    double acceptMoney(double money)//收钱，返款
    {
        double result = money;
        if (money >= moneyCondition)
        {
            result = money - floor(money / moneyCondition)*money
Return;
        }
        return result;
    }
};

class  CashContext
{
private:
    CashSuper *cs;
public:
    CashContext(string str)//设置策略
    {
        if (str == "正常收费")
        {
            cs = new CashNormal();
        }
        else if (str == "打9折")
        {
            cs = new CashRebate(0.9);
        }
        else if (str == "满1000送200")
        {
            cs = new CashReturn(1000, 200);
        }
    }
    double getResult(double money)
    {
        return cs->acceptMoney(money);
    }
```

```
};

int main()
{
    double money = 1000;
    CashContext *cc = new CashContext("正常收费");
    cout << cc->getResult(money);
    cin.get();
    return 0;
}
```

中介者模式

Mediator模式也叫中介者模式，是由GoF提出的23种软件设计模式的一种。Mediator模式是行为模式之一，在Mediator模式中，类之间的交互行为被统一放在Mediator的对象中，对象通过Mediator对象同其他对象交互，Mediator对象起着控制器的作用。

Mediator 调停者模式

调停者模式：调停者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使他们可以松散偶合。

当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。

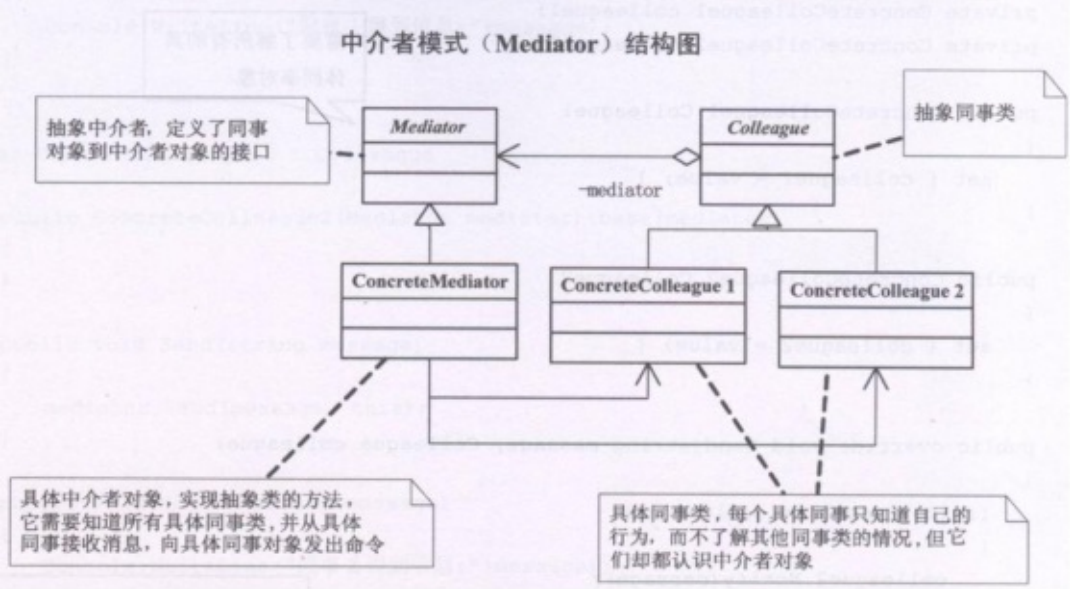
保证这些作用可以彼此独立的变化。调停者模式将多对多的相互作用转化为一对多的相互作用。调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

四个MM打麻将，相互之间谁应该给谁多少钱算不清楚了，幸亏当时我在旁边，按照各自的筹码数算钱，赚了钱的从我这里拿，赔了钱的也付给我，一切就OK啦，俺得到了四个MM的电话。

中介者模式，找不到老婆可以相亲靠婚介

类图角色和职责

GOOD：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显示的相互引用，从而降低耦合；而且可以独立地改变它们之间的交互。



- Mediator抽象中介者：中介者类的抽象父类
- concreteMediator：具体的中介者类
- Colleague：关联类的抽象父类
- concreteColleague：具体的关联类

适用于：用一个中介对象，封装一些列对象（同事）的交换，中介者是各个对象不需要显示的相互作用，从而实现了耦合松散，而且可以独立的改变他们之间的交换。

优点

- 将系统按功能分割成更小的对象，符合类的最小设计原则
- 对关联对象的集中控制
- 减小类的耦合程度，明确类之间的相互关系：当类之间的关系过于复杂时，其中任何一个类的修改都会影响到其他类，不符合类的设计的开闭原则 ，而Mediator模式将原来相互依存的多对多的类之间的关系简化为Mediator控制类与其他关联类的一对多的关系，当其中一个类修改时，可以对其他关联类不产生影响（即使有修改，也集中在Mediator控制类）。
- 有利于提高类的重用性

示例代码

```
#include <iostream>
#include "string"
```

```
using namespace std;

class Person
{
public:
    Person(string name, int sex, int condi)
    {
        m_name = name;
        m_sex = sex;
        m_condi = condi;
    }

    string getName()
    {
        return m_name;
    }

    int getSex()
    {
        return m_sex;
    }

    int getCondi()
    {
        return m_condi;
    }

    virtual void getParter(Person *p) = 0;

protected:
    string    m_name;
    int        m_sex;
    int        m_condi;
};

class Woman : public Person
{
public:
    Woman(string name, int sex, int condi) : Person(name, sex, c
ondi)
```

```
    {

    }

    virtual void getParter(Person *p)
    {
        if (this->m_sex == p->getSex())
        {
            cout << "我不是同性恋..." << endl;
        }

        if (this->getCondi() == p->getCondi() )
        {
            cout << this->getName() << " 和 " << p->getName() <<
"绝配 " <<endl;
        }
        else
        {
            cout << this->getName() << " 和 " << p->getName() <<
"不配 " <<endl;
        }
    }
};

class Man : public Person
{
public:
    Man(string name, int sex, int condi) : Person(name, sex, con
di)
    {

    }
    virtual void getParter(Person *p)
    {
        if (this->m_sex == p->getSex())
        {
            cout << "我不是同性恋..." << endl;
        }

        if (this->getCondi() == p->getCondi() )
```

```
        {
            cout << this->getName() << " 和 " << p->getName() <<
"绝配 " <<endl;
        }
        else
        {
            cout << this->getName() << " 和 " << p->getName() <<
"不配 " <<endl;
        }
    }
};

void main()
{
    //string name, int sex, int condi
    Person *xiaofang = new Woman("小芳", 2, 5);

    Person *zhangsan = new Man("张三", 1, 4);

    Person *lisi = new Man("李四", 1, 5);
    xiaofang->getParter(zhangsan);

    xiaofang->getParter(lisi);

    system("pause");
    return ;
}
```

以上 Woman Man 类的太紧密，需要解耦合

```
#include <iostream>
#include "string"
using namespace std;

class Mediator;

class Person
{
public:
```

```
    Person(string name, int sex, int condi, Mediator *m)
    {
        m_name = name;
        m_sex = sex;
        m_condi = condi;
        mediator = m;
    }

    string getName()
    {
        return m_name;
    }

    int getSex()
    {
        return m_sex;
    }

    int getCondi()
    {
        return m_condi;
    }

    virtual void getParter(Person *p) = 0;

protected:
    string    m_name;
    int        m_sex;
    int        m_condi;
     Mediator *mediator;
};

class Mediator
{
public:
    void setMan(Person *man)
    {
        pMan = man;
    }
```

```
    void setWoman(Person *woman)
    {
        pWoman = woman;
    }

public:
    virtual void getParter()
    {
        if (pWoman->getSex() == pMan->getSex())
        {
            cout << "同性恋 之间 不能找对象 " << endl;
        }

        if (pWoman->getCondi() == pMan->getCondi() )
        {
            cout << pWoman->getName() << " 和 " << pMan->getName
() << "绝配 " <<endl;
        }
        else
        {
            cout << pWoman->getName() << " 和 " << pMan->getName
() << "不配 " <<endl;
        }
    }

private:
    Person    *pMan;
    //list<Person *> m_list;
    Person    *pWoman;
};

class Woman : public Person
{
public:
    Woman(string name, int sex, int condi, Mediator *m) : Person
(name, sex, condi, m)
    {

    }
```

```
    virtual void getParter(Person *p)
    {
        mediator->setMan(p);
        mediator->setWoman(this);
        mediator->getParter(); //找对象
    }
};

class Man : public Person
{
public:
    Man(string name, int sex, int condi, Mediator *m) : Person(n
ame, sex, condi, m)
    {

    }

    virtual void getParter(Person *p)
    {
        mediator->setMan(this);
        mediator->setWoman(p);
        mediator->getParter(); //找对象
    }
};

void main()
{
    //string name, int sex, int condi
    Mediator *m = new Mediator;
    Person *xiaofang = new Woman("小芳", 2, 5, m);

    Person *zhangsan = new Man("张三", 1, 4, m);

    Person *lisi = new Man("李四", 1, 5, m);
    xiaofang->getParter(zhangsan);

    xiaofang->getParter(lisi);

    system("pause");
    return ;
```

```
}
```

```
#include<iostream>
#include <string>
using namespace std;

class Country;

class UniteNations
{
public:
    virtual void declare(string message, Country *colleague) = 0
;
};

class Country
{
protected:
    UniteNations *mediator;
public:
    Country(UniteNations *mediator)
    {
        this->mediator = mediator;
    }
};

class USA :public Country
{
public:
    USA(UniteNations *mediator) :Country(mediator)
    {}
    void declare(string message)
    {
        cout << "美发布信息: " << message << endl;
        mediator->declare(message, this);
    }
    void getMessage(string message)
    {
        cout << "美国获得对方信息: " << message << endl;
```

```
    }
};

class Iraq :public Country
{
public:
    Iraq(UniteNations *mediator) :Country(mediator)
    {}
    void declare(string message)
    {
        cout << "伊拉克发布信息: " << message << endl;
        mediator->declare(message, this);
    }
    void getMessage(string message)
    {
        cout << "伊拉克获得对方信息: " << message << endl;
    }
};

class UnitedNationsSecurityCouncil :public UniteNations
{
public:
    USA *usa;
    Iraq *iraq;
    void declare(string message, Country *colleague)
    {
        if (colleague == usa)
        {
            iraq->getMessage(message);
        }
        else
        {
            usa->getMessage(message);
        }
    }
};

int main()
{
    UnitedNationsSecurityCouncil *unsc = new UnitedNationsSecuri
```

```
tyCouncil();

    USA *c1 = new USA(unsc);
    Iraq *c2 = new Iraq(unsc);

    unsc->usa = c1;
    unsc->iraq = c2;

    c1->declare("不准开发核武器，否则打你！");
    c2->declare("他妈的美国去死！");

    cin.get();
    return 0;
}
```

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Colleague;
//中介者类
class Mediator
{
public:
    virtual void Send(string message,Colleague* col) = 0;
};
//抽象同事类
class Colleague
{
protected:
    Mediator* mediator;
public:
    Colleague(Mediator* temp)
    {
        mediator = temp;
    }
};
//同事一
```

```
class Colleague1 : public Colleague
{
public:
    Colleague1(Mediator* media) : Colleague(media){}

    void Send(string strMessage)
    {
        mediator->Send(strMessage,this);
    }

    void Notify(string strMessage)
    {
        cout<<"同事一获得了消息"<<strMessage<<endl;
    }
};

//同事二
class Colleague2 : public Colleague
{
public:
    Colleague2(Mediator* media) : Colleague(media){}

    void Send(string strMessage)
    {
        mediator->Send(strMessage,this);
    }

    void Notify(string strMessage)
    {
        cout<<"同事二获得了消息"<<strMessage<<endl;
    }
};

//具体中介者类
class ConcreteMediator : public Mediator
{
public:
    Colleague1 * col1;
    Colleague2 * col2;
    virtual void Send(string message,Colleague* col)
```

```
    {
        if(col == col1)
            col2->Notify(message);
        else
            col1->Notify(message);
    }
};

//客户端：
int main()
{
    ConcreteMediator * m = new ConcreteMediator();

    //让同事认识中介
    Colleague1* col1 = new Colleague1(m);
    Colleague2* col2 = new Colleague2(m);

    //让中介认识具体的同事类
    m->col1 = col1;
    m->col2 = col2;

    col1->Send("吃饭了吗？");
    col2->Send("还没吃，你请吗？");
    return 0;
}
```

建造者模式

Builder模式也叫建造者模式或者生成器模式，是由GoF提出的23种设计模式中的一种。Builder模式是一种对象创建型模式之一，用来隐藏复合对象的创建过程，它把复合对象的创建过程加以抽象，通过子类继承和重载的方式，动态地创建具有复合属性的对象。

对象的创建：Builder模式是为对象的创建而设计的模式- 创建的是一个复合对象：被创建的对象为一个具有复合属性的复合对象- 关注对象创建的各部分的创建过程：不同的工厂（这里指builder生成器）对产品属性有不同的创建方法

建造模式：将产品的内部表象和产品的生成过程分割开来，从而使一个建造过程生成具有不同的内部表象的产品对象。

建造模式使得产品内部表象可以独立的变化，客户不必知道产品内部组成的细节。

建造模式可以强制实行一种分步骤进行的建造过程。

MM最爱听的就是“我爱你”这句话了，见到不同地方的MM，要能够用她们的方言跟她说这句话哦，我有一个多种语言翻译机，上面每种语言都有一个按键，见到MM我只要按对应的键，它就能够用相应的语言说出“我爱你”这句话了，国外的MM也可以轻松搞掂，这就是我的“我爱你”builder。（这一定比美军在伊拉克用的翻译机好卖）

角色和职责

- Builder：为创建产品各个部分，统一抽象接口。
- ConcreteBuilder：具体的创建产品的各个部分，部分A， 部分B，部分C。
- Director：构造一个使用Builder接口的对象。
- Product：表示被构造的复杂对象。

ConcreteBuilder创建该产品的内部表示并定义它的装配过程，包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

适用情况：一个对象的构建比较复杂，将一个对象的构建和对象的表示进行分离。

Builder模式和工厂模式的区别

Factory模式中

- 有一个抽象的工厂。
- 实现一个具体的工厂---汽车工厂。
- 工厂生产汽车A，得到汽车产品A。
- 工厂生产汽车B，得到汽车产品B。

这样做，实现了购买者和生产线的隔离。强调的是结果。

Builder模式

- 引擎工厂生产引擎产品，得到汽车部件A。
- 轮胎工厂生产轮子产品，得到汽车部件B。
- 底盘工厂生产车身产品，得到汽车部件C。
- 将这些部件放到一起，形成刚好能够组装成一辆汽车的整体。
- 将这个整体送到汽车组装工厂，得到一个汽车产品。

这样做，目的是为了实现复杂对象生产线和其部件的解耦。强调的是过程

两者的区别在于

- Factory模式不考虑对象的组装过程，而直接生成一个我想要的对象。
- Builder模式先一个个的创建对象的每一个部件，再统一组装成一个对象。
- Factory模式所解决的问题是，工厂生产产品。
- 而Builder模式所解决的问题是工厂控制产品生成器组装各个部件的过程，然后从产品生成器中得到产品。
- Builder模式不是很常用。模式本身就是一种思想。知道了就可以了。

设计模式就是一种思想。学习一个模式，花上一两个小时把此模式的意思理解了，就够了。其精华的所在会在以后工作的设计中逐渐体现出来。

示例代码

问题抛出

```
#include <iostream>
#include "string"
using namespace std;

class House
```

```
{
public:
    void setDoor(string door)
    {
        this->m_door = door;
    }

    void setWall(string wall)
    {
        this->m_wall = wall;
    }
    void setWindow(string window)
    {
        this->m_window = window;
    }

    string getDoor( )
    {
        cout << m_door << endl;
        return this->m_door ;
    }

    string getWall()
    {
        cout << m_wall << endl;
        return this->m_wall;
    }

    string getWindow()
    {
        cout << m_window << endl;
        return m_window;
    }

private:
    string    m_door;
    string    m_wall;
    string    m_window;
};
```

```
class Build
{
public:
    Build()
    {
        m_house = new House;
    }

    void makeBuild()
    {
        buildDoor(m_house);
        buildWall(m_house);
        buildWindow(m_house);
    }

    void buildDoor(House *h)
    {
        h->setDoor("门");
    }

    void buildWall(House *h)
    {
        h->setWall("墙");
    }

    void buildWindow(House *h)
    {
        h->setWindow("窗");
    }

    House *getHouse()
    {
        return m_house;
    }

private:
    House *m_house;
};

void main()
```

```
{
    /* 客户直接造房子
    House *house = new House;
    house->setDoor("门");
    house->setWall("墙面");
    house->setWindow("窗口");
    delete house;
    */

    //请工程队建造房子
    House *house = NULL;
    Build * build = new Build;
    build->makeBuild();
    house = build->getHouse();
    house->getDoor();
    house->getWall();
    house->getWindow();

    system("pause");
    return ;
}
```

Builder模式

```
#include <iostream>
#include "string"
using namespace std;

class House
{
public:
    void setDoor(string door)
    {
        this->m_door = door;
    }

    void setWall(string wall)
    {
        this->m_wall = wall;
```

```
    }

    void setWindow(string window)
    {
        this->m_window = window;
    }

    string getDoor( )
    {
        cout << m_door << endl;
        return this->m_door ;
    }

    string getWall()
    {
        cout << m_wall << endl;
        return this->m_wall;
    }

    string getWindow()
    {
        cout << m_window << endl;
        return m_window;
    }

private:
    string    m_door;
    string    m_wall;
    string    m_window;
};

class  Builder
{
public:
    virtual void buildWall() = 0;
    virtual void buildDoor() = 0;
    virtual void buildWindow() = 0;
    virtual House* getHouse() = 0;
};
```

```
//公寓工程队
class  FlatBuilder : public Builder
{
public:
    FlatBuilder()
    {
        m_house = new House;
    }

    virtual void buildWall()
    {
        m_house->setWall(" flat wall");
    }

    virtual void buildDoor()
    {
        m_house->setDoor("flat door");
    }

    virtual void buildWindow()
    {
        m_house->setWindow("flat window");
    }

    virtual House* getHouse()
    {
        return m_house;
    }
private:
    House *m_house;
};

//别墅villa工程队
class  VillaBuilder : public Builder
{
public:
    VillaBuilder()
    {
        m_house = new House;
    }
```

```
    virtual void buildWall()
    {
        m_house->setWall(" villa wall");
    }

    virtual void buildDoor()
    {
        m_house->setDoor("villa door");
    }

    virtual void buildWindow()
    {
        m_house->setWindow("villa window");
    }

    virtual House* getHouse()
    {
        return m_house;
    }
private:
    House *m_house;
};

//设计师(指挥者)负责建造逻辑
//建筑队干具体的活
class Director
{
public:
    Director( Builder * build)
    {
        m_build = build;
    }
    void Construct()
    {
        m_build->buildWall();
        m_build->buildWindow();
        m_build->buildDoor();
    }
private:
```

```
     Builder * m_build;
};

void main()
{
    House        *house  = NULL;
    Builder        *builder = NULL;
    Director    *director = NULL;

    // 请一个建造别墅的工程队
    builder = new VillaBuilder;

    //设计师 指挥 工程队 干活
    director = new Director(builder);
    director->Construct();
    house  =  builder->getHouse();
    house->getWindow();
    house->getDoor();

    delete house;
    delete builder;

    //请 FlatBuilder 公寓
    builder = new FlatBuilder;
    director = new Director(builder);
    director->Construct();
    house  =  builder->getHouse();
    house->getWindow();
    house->getDoor();
    delete house;
    delete builder;
    delete director;

    system("pause");
    return ;
}
```

```
#include <string>
#include <iostream>
```

```
#include <vector>
using namespace std;

class Person  //抽象类，预留ule接口
{
public:
    virtual void createHead() = 0;
    virtual void createHand() = 0;
    virtual void createBody() = 0;
    virtual void createFoot() = 0;
};

class ThinPerson :public Person  ///实现抽象类瘦子，
{
    void createHead()
    {
        cout << "thin head" << endl;
    }
    void createHand()
    {
        cout << "thin hand" << endl;
    }
    void createBody()
    {
        cout << "thin body" << endl;
    }
    void createFoot()
    {
        cout << "thin foot" << endl;
    }
};

class FatPerson :public Person //胖子
{
    void createHead()
    {
        cout << "fat head" << endl;
    }
    void createHand()
    {
```

```
        cout << "fat hand" << endl;
    }
    void createBody()
    {
        cout << "fat body" << endl;
    }
    void createFoot()
    {
        cout << "fat foot" << endl;
    }
};

class Director
{
private:
    Person *p;//基类的指针
public:
    Director(Person *temp) //传递对象
    {
        p = temp;//虚函数实现多态
    }
    void create()
    {
        p->createHead();
        p->createHand();
        p->createBody();
        p->createFoot();
    }
};

//客户端代码：
int main()
{
    Person *p = new FatPerson();

    Director *d = new Director(p);
    d->create();
    delete d;
    delete p;
```

```
    cin.get();
    return 0;
}
```

桥接模式

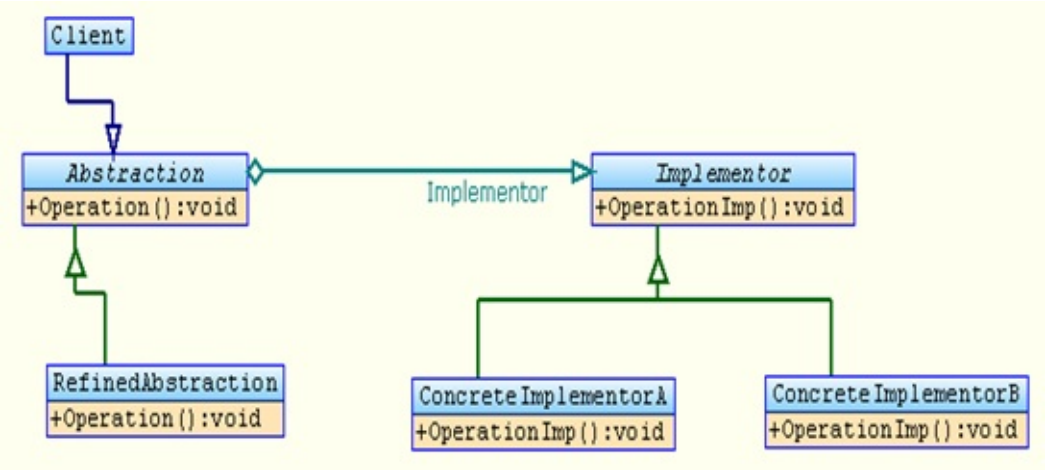
Bridge 模式又叫做桥接模式，是构造型的设计模式之一。Bridge模式基于类的最小设计原则，通过使用封装，聚合以及继承等行为来让不同的类承担不同的责任。它的主要特点是把抽象（abstraction）与行为实现（implementation）分离开来，从而可以保持各部分的独立性以及应对它们的功能扩展。

桥梁模式：将抽象化与实现化脱耦，使得二者可以独立的变化，也就是说将他们之间的强关联变成弱关联，也就是指在一个软件系统的抽象化和实现化之间使用组合

聚合关系而不是继承关系，从而使两者可以独立的变化。

早上碰到MM，要说早上好，晚上碰到MM，要说晚上好；碰到MM穿了件新衣服，要说你的衣服好漂亮哦，碰到MM新做的发型，要说你的头发好漂亮哦。不要问我“早上碰到MM新做了个发型怎么说”这种问题，自己用BRIDGE组合一下不就行了。

类图角色和职责



- Client：Bridge模式的使用者
- Abstraction：抽象类接口（接口或抽象类）维护对行为实现（Implementor）的引用
- Refined Abstraction：Abstraction子类
- Implementor：行为实现类接口 (Abstraction接口定义了基于Implementor接口的更高层次的操作)
- ConcreteImplementor：Implementor子类

适用于：桥接模式（Bridge Pattern）是将抽象部分与实现部分分离（解耦合），使它们都可以独立的变化。

车安装发动机，不同型号的车，安装不同型号的发动机。将“车安装发动机”这个抽象 和 实现进行分离；两个名字 就设计两个类。

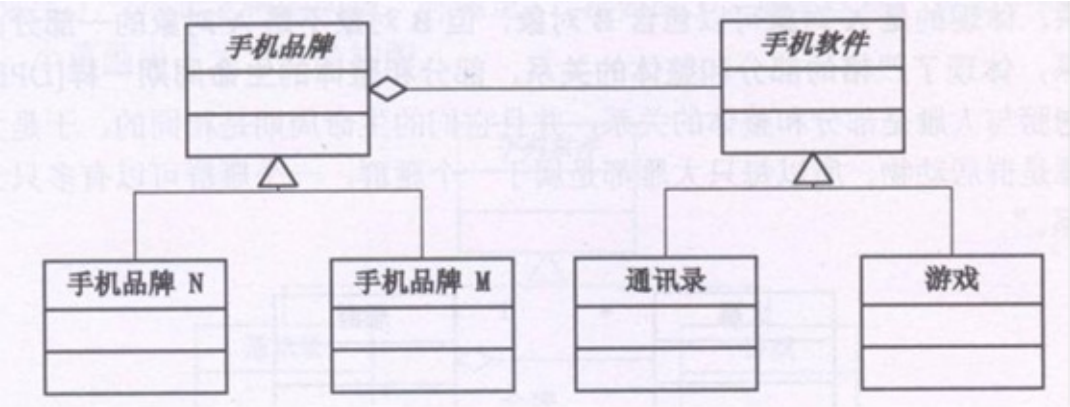
图形填颜色，不同形状的图形，填充上不同的颜色。将“图形 颜色”这个抽象和实现进行分离，两个名字，就设计两个类。

示例代码

```
#include <iostream>
using namespace std;

class Engine
{
public:
    virtual void InstallEngine() = 0;
};

class Engine4400cc : public Engine
{
public:
    virtual void InstallEngine()
    {
        cout << "我是 4400cc 发动机 安装完毕 " << endl;
    }
};

class Engine4500cc : public Engine
{
public:
    virtual void InstallEngine()
    {
        cout << "我是 4500cc 发动机 安装完毕 " << endl;
    }
};

class Car
```

```
{
public:
    Car(Engine *engine)
    {
        this->m_engine = engine;
    }
    virtual void installEngine() = 0;

protected:
    Engine *m_engine;
};

class WBM5 : public Car
{
public:
    WBM5(Engine *engine) : Car(engine){}

    virtual void installEngine()
    {
        m_engine->InstallEngine();
    }
};

class WBM6 : public Car
{
public:
    WBM6(Engine *engine) : Car(engine){}

    virtual void installEngine()
    {
        cout << "我是 王保明 WBM6 " << endl;
        m_engine->InstallEngine();
    }
};

void main()
{
    Engine    *engine = NULL;
    WBM6    *wbm6 = NULL;
```

```
    engine = new Engine4400cc;
    wbm6 = new WBM6(engine);
    wbm6->installEngine();

    delete wbm6;
    delete engine;

    system("pause");
    return ;
}
```




```
#include <iostream>
#include <string>
using namespace std;

class HandsetSoft  //手机软件
{
public:
    virtual void run() = 0;
};

class HandsetGame :public HandsetSoft  //游戏软件
{
public:
    void run()
    {
        cout << "运行手机游戏" << endl;
    }
};
```

```
class HandsetAddressList :public HandsetSoft  //通讯录软件
{
public:
    void run()
    {
        cout << "运行手机通讯录" << endl;
    }
};

class HandsetBrand  //手机品牌
{
protected:
    HandsetSoft *soft;
public:
    void setHandsetSoft(HandsetSoft *soft)
    {
        this->soft = soft;
    }
    virtual void run() = 0;
};

class HandsetBrandN :public HandsetBrand  //N品牌
{
public:
    void run()
    {
        soft->run();
    }
};

class HandsetBrandM :public HandsetBrand  //M品牌
{
public:
    void run()
    {
        soft->run();
    }
};

int main()
```

```
{
    HandsetBrand *hb;
    hb = new HandsetBrandM();

    hb->setHandsetSoft(new HandsetGame());
    hb->run();
    hb->setHandsetSoft(new HandsetAddressList());
    hb->run();

    cin.get();
    return 0;
}
```

观察者模式

Observer模式是行为模式之一，它的作用是当一个对象的状态发生变化时，能够自动通知其他关联对象，自动刷新对象状态。

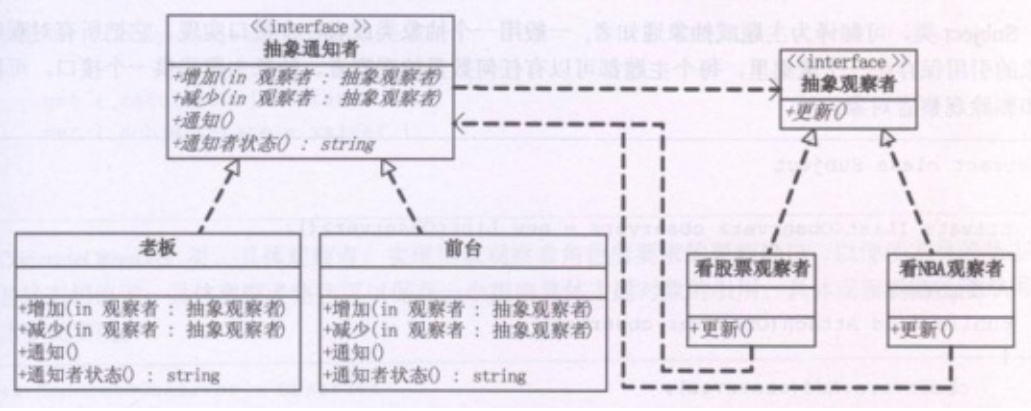
Observer模式提供给关联对象一种同步通信的手段，使某个对象与依赖它的其他对象之间保持状态同步。

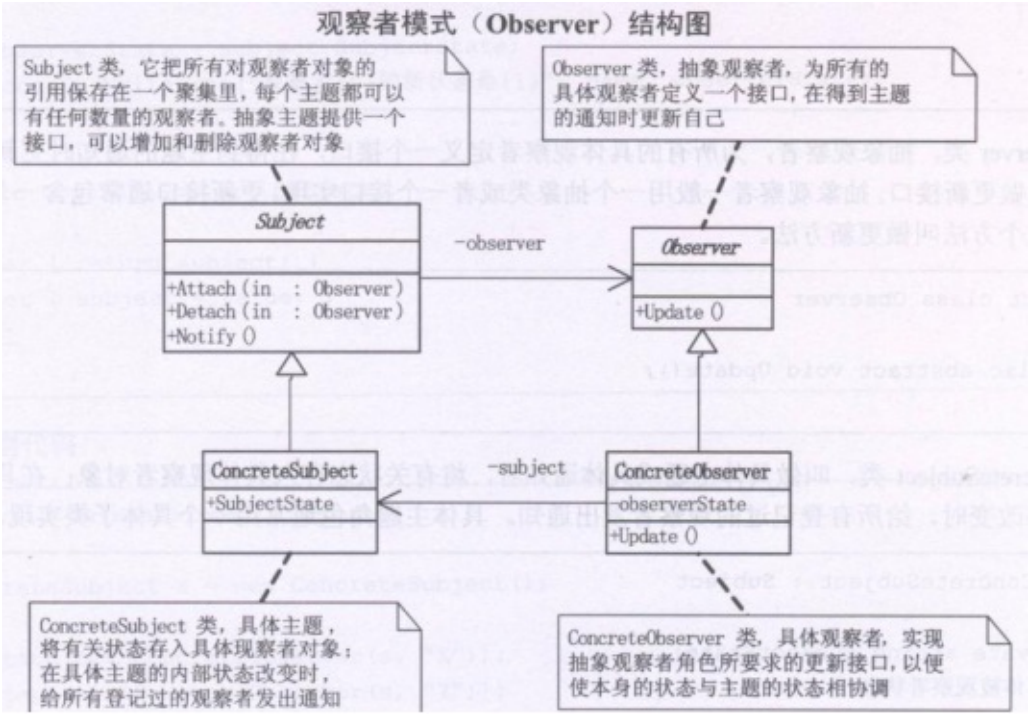
观察者模式：观察者模式定义了一种一队多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使他们能够自动更新自己。

想知道咱们公司最新MM情报吗？加入公司的MM情报邮件组就行了，tom负责搜集情报，他发现的新情报不用一个一个通知我们，直接发布给邮件组，我们作为订阅者（观察者）就可以及时收到情报啦。

监视，观察者，都有一个基类，派生，实现不同的效果监视者的类，管理所有的观察者，增加或者删除，发出消息，让观察者处理观察者的类需要接受消息并处理。

类图角色和职责





- Subject（被观察者）

被观察的对象。当需要被观察的状态发生变化时，需要通知队列中所有观察者对象。Subject需要维持（添加，删除，通知）一个观察者对象的队列列表。

- ConcreteSubject

被观察者的具体实现。包含一些基本的属性状态及其他操作。

- Observer（观察者）

接口或抽象类。当Subject的状态发生变化时，Observer对象将通过一个callback函数得到通知。

- ConcreteObserver

观察者的具体实现。得到通知后将完成一些具体的业务逻辑处理。

典型应用

- 侦听事件驱动程序设计中的外部事件
- 侦听/监视某个对象的状态变化
- 发布者/订阅者(publisher/subscriber)模型中，当一个外部事件（新的产品，消息的出现等等）被触发时，通知邮件列表中的订阅者

适用于：定义对象间一种一对多的依赖关系，使得每一个对象改变状态，则所有依赖于他们的对象都会得到通知。

使用场景：定义了一种一对多的关系，让多个观察对象（公司员工）同时监听一个主题对象（秘书），主题对象状态发生变化时，会通知所有的观察者，使它们能够更新自己。

示例代码

```
#include <iostream>
#include "string"
#include "list"
using namespace std;

class Secretary;

class PlayserObserver //观察者
{
public:
    PlayserObserver(Secretary *secretary)
    {
        this->m_secretary = secretary;
    }

    void update(string action)
    {
        cout << "action:" << action << endl;
        cout << "老板来了 我很害怕啊..." << endl;
    }

private:
    Secretary *m_secretary;
};

class Secretary
{
public:
    Secretary()
    {
```

```
        m_list.clear();
    }

    void Notify(string info)
    {
        //给所有的 观察者 发送 情报
        for ( list<PlayserObserver *>::iterator it=m_list.begin(
);
             it!=m_list.end(); it++)
        {
            (*it)->update(info);
        }
    }

    void setPlayserObserver(PlayserObserver *o)
    {
        m_list.push_back(o);
    }

private:
    list<PlayserObserver *> m_list;
};

void main()
{
    Secretary            *secretary = NULL;
    PlayserObserver        *po1 = NULL;
    PlayserObserver        *po2 = NULL;

    secretary = new Secretary;
    po1 = new PlayserObserver(secretary);
    po2 = new PlayserObserver(secretary);

    secretary->setPlayserObserver(po1);
    secretary->setPlayserObserver(po2);

    secretary->Notify("老板来了") ;
    secretary->Notify("老板走了");
    delete secretary ;
    delete po1 ;
```

```
    delete po2 ;

    system("pause");
    return ;
}
```

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

class Subject; //可以使用subject

class Observer
{
protected:
    string name;
    Subject *sub;
public:
    Observer(string name, Subject *sub)//观察者的名字， 监视与通知的
类
    {
        this->name = name;//输入名字
        this->sub = sub;//设置谁来通知我
    }
    virtual void update() = 0;//纯虚函数
};

class StockObserver :public Observer //继承，自己实现刷新函数
{
public:
    StockObserver(string name, Subject *sub) :Observer(name, sub
)
    {
    }
    void update();
};

class NBAObserver :public Observer
```

```
{
public:
    NBAObserver(string name, Subject *sub) :Observer(name, sub)

    {
    }
    void update();
};

class Subject
{
protected:
    list<Observer*> observers;///存储观察者的指针，链表
public:
    string action;
    virtual void attach(Observer*) = 0;
    virtual void detach(Observer*) = 0;
    virtual void notify() = 0;//实现监听的基类
};

class Secretary :public Subject
{
    void attach(Observer *observer)  //载入通知的列表
    {
        observers.push_back(observer);
    }
    void detach(Observer *observer)//删除
    {
        list<Observer *>::iterator iter = observers.begin();
        while (iter != observers.end())
        {
            if ((*iter) == observer)
            {
                observers.erase(iter);
            }
            ++iter;
        }
    }
    void notify()  //通知函数
    {
```

```
        list<Observer *>::iterator iter = observers.begin();
        while (iter != observers.end())
        {
            (*iter)->update();
            ++iter;
        }
    }
};

void StockObserver::update()
{
    cout << name << " 收到消息：" << sub->action << endl;
    if (sub->action == "梁所长来了!")
    {
        cout << "我马上关闭股票，装做很认眞工作的样子！" << endl;
    }
    if (sub->action == "去喝酒！")
    {
        cout << "我马上走" << endl;
    }
}

void NBAObserver::update()
{
    cout << name << " 收到消息：" << sub->action << endl;
    if (sub->action == "梁所长来了!")
    {
        cout << "我马上关闭NBA，装做很认眞工作的样子！" << endl;
    }

   if (sub->action == "去喝酒！")
   {
    cout << "我马上拍" << endl;
    }
}

int main()
{
    Subject *dwq = new Secretary();//消息监视，监视
```

```
    Observer *xs = new NBAObserver("xiaoshuai", dwq);//订阅消息
    Observer *zy = new NBAObserver("zouyue", dwq);
    Observer *lm = new StockObserver("limin", dwq);

    dwq->attach(xs);
    dwq->attach(zy);
    dwq->attach(lm);//增加到队列

    dwq->action = "去吃饭了！";
    dwq->notify();
    dwq->action = "去喝酒！";
    dwq->notify();
    cout << endl;
    dwq->action = "梁所长来了!";
    dwq->notify();
    cin.get();
    return 0;
}
```

备忘录模式

Memento模式也叫备忘录模式，是行为模式之一，它的作用是保存对象的内部状态，并在需要的时候（undo/rollback）恢复对象以前的状态。

数据库的备份，文档编辑中的撤销等功能。

备忘录模式：备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。

备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捉住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

同时跟几个MM聊天时，一定要记清楚刚才跟MM说了些什么话，不然MM发现了会不高兴的哦，幸亏我有个备忘录，刚才与哪个MM说了什么话我都拷贝一份放到备忘录里面保存，这样可以随时察看以前的记录啦

设计需要回放的软件，记录一下事物的状态。数据库备份，文档的编译，撤销，恢复

设计备忘录三大步骤

1. 设计记录的节点，存储记录，记录鼠标，键盘的状态
2. 设计记录的存储，vector、list、map、set、链表、图、数组、树
3. 操作记录的类，记录节点状态，设置节点状态，显示状态，0.1秒记录一下

应用场景

如果一个对象需要保存状态并可通过undo或rollback等操作恢复到以前的状态时，可以使用Memento模式。

- 一个类需要保存它的对象的状态（相当于Originator角色）
- 设计一个类，该类只是用来保存上述对象的状态（相当于Memento角色）
- 需要的时候，Caretaker角色要求Originator返回一个Memento并加以保存
- undo或rollback操作时，通过Caretaker保存的Memento恢复Originator对象的状态

类图角色和职责

- Originator（原生者）：需要被保存状态以便恢复的那个对象。

- Memento（备忘录）：该对象由Originator创建，主要用来保存Originator的内部状态。
- Caretaker（管理者）：负责在适当的时间保存/恢复Originator对象的状态。

适用于：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样就可以将以后的对象状态恢复到先前保存的状态。

适用于功能比较复杂的，但需要记录或维护属性历史的类；或者需要保存的属性只是众多属性中的一小部分时Originator可以根据保存的Memo还原到前一状态。

示例代码

```
#include <iostream>
#include "string"
using namespace std;

//Caretaker 管理者
// MememTo  备忘录

class MememTo
{
public:
    MememTo(string name,int age )
    {
        m_name = name;
        m_age = age;
    }

    string getName()
    {
        return m_name;
    }

    int getAge()
    {
        return m_age;
    }

    void setName(string name)
```

```
    {
        this->m_name = name;
    }

    void setAge(int age)
    {
        this->m_age = age;
    }
private:
    string    m_name;
    int        m_age;
};

class  Person
{
public:
    Person(string name,int age )
    {
        m_name = name;
        m_age = age;
    }

    string getName()
    {
        return m_name;
    }

    int getAge()
    {
        return m_age;
    }

    void setName(string name)
    {
        this->m_name = name;
    }

    void setAge(int age)
    {
        this->m_age = age;
```

```
    }

    //保存
    MememTo* createMememTo()
    {
        return new MememTo(m_name, m_age);
    }

    //还原
    void setMememTo(MememTo* memto)
    {
        this->m_age = memto->getAge();
        this->m_name = memto->getName();
    }

public:
    void printT()
    {
        cout << "m_name:" << m_name << " m_age:" << m_age << endl
;
    }

private:
    string    m_name;
    int        m_age;
};

class Caretaker
{
public:
    Caretaker(MememTo *memto)
    {
        this->memto = memto;
    }

    MememTo *getMememTo()
    {
        return memto;
    }
```

```
    void setMememTo(MememTo *memto)
    {
        this->memto = memto;
    }

private:
    MememTo *memto;
};

void main1()
{
    //MememTo *memto = NULL;
    Caretaker *caretaker = NULL;
    Person *p = new Person("张三", 32);
    p->printT();

    //创建 Person对象的一个状态
    printf("---------\n");
    caretaker = new Caretaker( p->createMememTo());
    p->setAge(42);
    p->printT();

    printf("还原旧的状态\n" );
    p->setMememTo(caretaker->getMememTo());
    p->printT();

    delete caretaker;
    delete p;
}

void main2()
{
    MememTo *memto = NULL;
    Person *p = new Person("张三", 32);
    p->printT();

    //创建 Person对象的一个状态
    printf("---------\n");
    memto = p->createMememTo();
    p->setAge(42);
```

```
    p->printT();

    printf("还原旧的状态\n" );
    p->setMememTo(memto);
    p->printT();

    delete memto;
    delete p;
}
void main()
{
    //main1();
    main2();
    system("pause");
    return ;
}
```

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

//备忘录的节点
class Memo
{
public:
    string state;
    Memo(string state) //记录当前的状态，
    {
        this->state = state;
    }
};

class Originator//类的包含备忘录的节点
{
public:
    string state;
```

```
    void setMemo(Memo *memo)
    {
        state = memo->state;
    }

    Memo *createMemo()
    {
        return new Memo(state);
    }

    void show()
    {
        cout << state << endl;
    }
};

//备忘录的集合
class Caretaker
{
public:
    vector<Memo *> memo;
    void save(Memo *memo)
    {
        (this->memo).push_back(memo);
    }
    Memo *getState(int i)
    {
        return memo[i];
    }
};

int main()
{
    Originator *og = new Originator();
    Caretaker *ct = new Caretaker();

    og->state = "on";
    og->show();
    ct->save(og->createMemo());
```

```
    og->state = "off";
    og->show();
    ct->save(og->createMemo());

    og->state = "middle";
    og->show();
    ct->save(og->createMemo());

    og->setMemo(ct->getState(1));
    og->show();

    og->setMemo(ct->getState(2));
    og->show();
    cin.get();
    return 0;
}
```

命令模式

Command模式也叫命令模式 ，是行为设计模式的一种。Command模式通过被称为Command的类封装了对目标对象的调用行为以及调用参数。

在面向对象的程序设计中，一个对象调用另一个对象，一般情况下的调用过程是：创建目标对象实例；设置调用参数；调用目标对象的方法。

但在有些情况下有必要使用一个专门的类对这种调用过程加以封装，我们把这种专门的类称作command类。

整个调用过程比较繁杂，或者存在多处这种调用。这时，使用Command类对该调用加以封装，便于功能的再利用。

调用前后需要对调用参数进行某些处理。调用前后需要进行某些额外处理，比如日志，缓存，记录历史操作等。

命令模式：命令模式把一个请求或者操作封装到一个对象中。

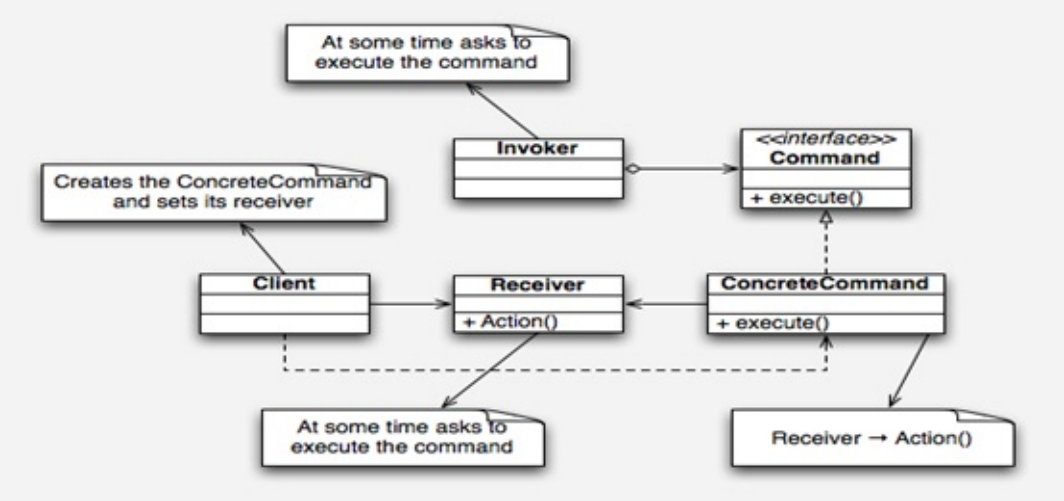
命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。

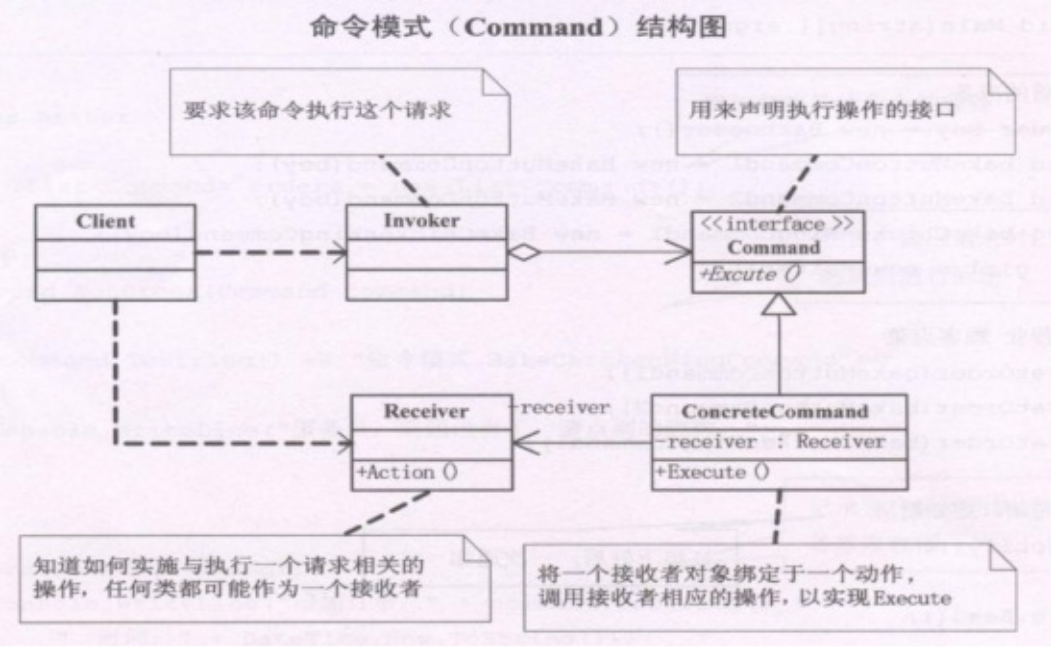
命令模式允许请求的一方和发送的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否执行，何时被执行以及是怎么被执行的。

系统支持命令的撤消。

俺有一个MM家里管得特别严，没法见面，只好借助于她弟弟在我们俩之间传送信息，她对我有什么指示，就写一张纸条让她弟弟带给我。这不，她弟弟又传送过来一个COMMAND，为了感谢他，我请他吃了碗杂酱面，哪知道他说：“我同时给我姐姐三个男朋友送COMMAND，就数你最小气，才请我吃面。”

类图角色和职责





- Command：Command命令的抽象类。
- ConcreteCommand：Command的具体实现类。
- Receiver：需要被调用的目标对象。
- Invorker：通过Invorker执行Command对象。

适用于：是将一个请求封装为一个对象，从而使你可用不同的请求对客户端进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。

示例代码

命令模式1

```
#include <iostream>
using namespace std;
```

```
class Doctor
{
public:
    void treat_eye()
    {
        cout << "医生 治疗 眼科病" << endl;
    }

    void treat_nose()
    {
        cout << "医生 治疗 鼻科病" << endl;
    }
};

class CommandTreatEye
{
public:
    CommandTreatEye(Doctor *doctor)
    {
        m_doctor = doctor;
    }

    void treat()
    {
        m_doctor->treat_eye();
    }
private:
    Doctor *m_doctor;
};

class CommandTreatNose
{
public:
    CommandTreatNose(Doctor *doctor)
    {
        m_doctor = doctor;
    }

    void treat()
```

```
    {
        m_doctor->treat_nose();
    }
private:
    Doctor *m_doctor;
};

void main2()
{
    //1 医生直接看病
    /*
    Doctor *doctor = new Doctor ;
    doctor->treat_eye();
    delete doctor;
    */

    //2 通过一个命令，医生通过命令治疗治病
    Doctor *doctor = new Doctor ;
    CommandTreatEye * commandtreateye = new CommandTreatEye(doct
or);
    commandtreateye->treat();
    delete commandtreateye;
    delete doctor;

    return ;
}

void main()
{
    main2();
    system("pause");
    return ;
}
```

命令模式2

```
#include <iostream>
#include "list"
using namespace std;
```

```
class Doctor
{
public:
    void treat_eye()
    {
        cout << "医生 治疗 眼科病" << endl;
    }

    void treat_nose()
    {
        cout << "医生 治疗 鼻科病" << endl;
    }
};

class Command
{
public:
    virtual void treat() = 0;
};

class CommandTreatEye : public Command
{
public:
    CommandTreatEye(Doctor *doctor)
    {
        m_doctor = doctor;
    }

    void treat()
    {
        m_doctor->treat_eye();
    }
private:
    Doctor *m_doctor;
};

class CommandTreatNose : public Command
{
public:
```

```
    CommandTreatNose(Doctor *doctor)
    {
        m_doctor = doctor;
    }

    void treat()
    {
        m_doctor->treat_nose();
    }
private:
    Doctor *m_doctor;
};

class BeautyNurse
{
public:
    BeautyNurse(Command *command)
    {
        this->command = command;
    }

public:
    void SubmittedCase() //提交病例 下单命令
    {
        command->treat();
    }

private:
    Command *command;
};

//护士长
class HeadNurse
{
public:
    HeadNurse()
    {
        m_list.clear();
    }
```

```
public:
    void setCommand(Command *command)
    {
        m_list.push_back(command);
    }

    void SubmittedCase() //提交病例下单命令
    {
        for (list<Command *>::iterator it=m_list.begin(); it!=m_
list.end(); it++)
        {
            (*it)->treat();
        }
    }
private:
    list<Command *> m_list;
};

void main2()
{
    //1 医生直接看病
    /*
    Doctor *doctor = new Doctor ;
    doctor->treat_eye();
    delete doctor;
    */

    //2 通过一个命令 医生通过 命令 治疗 治病
    Doctor *doctor = new Doctor ;
    Command * command = new CommandTreatEye(doctor);
    command->treat();
    delete command;
    delete doctor;

    return ;
}

void main3()
{
    //3 护士提交简历 给以上看病
```

```
    BeautyNurse        *beautynurse = NULL;
    Doctor            *doctor = NULL;
    Command            *command  = NULL;

    doctor = new Doctor ;

    //command = new CommandTreatEye(doctor);
    command = new CommandTreatNose(doctor);
    beautynurse = new BeautyNurse(command);
    beautynurse->SubmittedCase();

    delete doctor;
    delete command;
    delete beautynurse;

    return ;
}

//4 通过护士长批量的下单命令
void main4()
{
    //护士提交简历 给以上看病
    HeadNurse        *headnurse = NULL;
    Doctor            *doctor = NULL;
    Command            *command1  = NULL;
    Command            *command2  = NULL;

    doctor = new Doctor ;

    command1 = new CommandTreatEye(doctor);
    command2 = new CommandTreatNose(doctor);

    headnurse = new HeadNurse(); //new 护士长
    headnurse->setCommand(command1);
    headnurse->setCommand(command2);

    headnurse->SubmittedCase(); // 护士长批量下单命令

    delete doctor;
    delete command1;
```

```
    delete command2;
    delete headnurse;

    return ;
}

void main()
{
    //main2();
    //main3();
    main4();
    system("pause");
    return ;
}
```

示例代码3

```
#include <iostream>
#include "list"
using namespace std;

class Vendor
{
public:
    void sailbanana()
    {
        cout << "卖香蕉" << endl;
    }

    void sailapple()
    {
        cout << "卖苹果" << endl;
    }
};

class Command
{
public:
    virtual void sail() = 0;
```

```
};

class BananaCommand : public Command
{
public:
    BananaCommand(Vendor *v)
    {
        m_v = v;
    }

    Vendor *getV(Vendor *v)
    {
        return m_v;
    }

    void setV(Vendor *v)
    {
        m_v = v;
    }

    virtual void sail()
    {
        m_v->sailbanana();
    }
private:
    Vendor *m_v;
};

class AppleCommand : public Command
{
public:
    AppleCommand(Vendor *v)
    {
        m_v = v;
    }

    Vendor *getV(Vendor *v)
    {
        return m_v;
    }
```

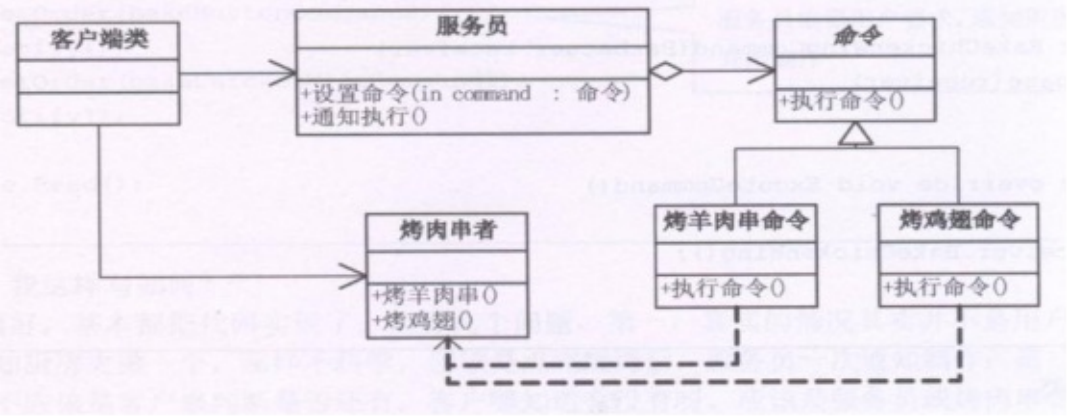
```
    void setV(Vendor *v)
    {
        m_v = v;
    }

    virtual void sail()
    {
        m_v->sailapple();
    }

private:
    Vendor *m_v;
};

class Waiter
{
public:
    Command *getCommand()
    {
        return m_command;
    }

    void setCommand(Command *c)
    {
        m_command = c;
    }

    void sail()
    {
        m_command->sail();
    }

private:
    Command *m_command;
};

class AdvWaiter
{
public:
```

```
    AdvWaiter()
    {
        m_list = new list<Command *>;
        m_list->resize(0);
    }

    ~AdvWaiter()
    {
        delete m_list;
    }

    void setCommands(Command *c)
    {
        m_list->push_back(c);
    }

    list<Command *> * getCommands()
    {
        return m_list;
    }

    void sail()
    {
        for (list<Command *>::iterator it=m_list->begin();
             it!=m_list->end(); it++ )
        {
            (*it)->sail();
        }
    }

private:
    list<Command *> *m_list;
};

//小商贩直接卖水果
void main1()
{
    Vendor *v = new Vendor;
    v->sailapple();
    v->sailbanana();
```

```
    delete v;
    return ;
}

//小商贩通过命令卖水果
void main2()
{
    Vendor *v = new Vendor;
    AppleCommand *ac = new AppleCommand(v);
    ac->sail();

    BananaCommand *bc = new BananaCommand(v);
    bc->sail();

    delete bc;
    delete ac;
    delete v;
}

//小商贩通过waiter卖水果
void main3()
{
    Vendor *v = new Vendor;
    AppleCommand *ac = new AppleCommand(v);
    BananaCommand *bc = new BananaCommand(v);

    Waiter *w = new Waiter;
    w->setCommand(ac);
    w->sail();

    w->setCommand(bc);
    w->sail();

    delete w;
    delete bc;
    delete ac;
    delete v;
}
```

```
//小商贩通过advwaiter批量下单卖水果
void main4()
{
    Vendor *v = new Vendor;
    AppleCommand *ac = new AppleCommand(v);
    BananaCommand *bc = new BananaCommand(v);

    AdvWaiter *w = new AdvWaiter;
    w->setCommands(ac);
    w->setCommands(bc);
    w->sail();

    delete w;
    delete bc;
    delete ac;
    delete v;
}

void main()
{
    //main1();
    //main2();
    //main3();
    main4();
    system("pause");
}
```

示例代码4




```
#include <iostream>
```

```
#include <string>
#include <list>

using namespace std;

class Barbecuer  //接收者执行命令
{
public:
    void bakeMutton()
    {
        cout << "烤羊肉串" << endl;
    }
    void bakeChickenWing()
    {
        cout << "烤鸡翅" << endl;
    }
};

class Command   //命令基类
{
protected:
    Barbecuer *receiver;//类的包含
public:
    Command(Barbecuer *receiver)//命令接受
    {
        this->receiver = receiver;
    }
    virtual void executeCommand() = 0;
};

class BakeMuttonCommand :public Command  //命令传送着
{
public:
    BakeMuttonCommand(Barbecuer *receiver) :Command(receiver)
    {}
    void executeCommand()
    {
        receiver->bakeMutton();
    }
};
```

```
class BakeChikenWingCommand :public Command  //命令传送着
{
public:
    BakeChikenWingCommand(Barbecuer *receiver) :Command(receiver
)
    {}
    void executeCommand()
    {
        receiver->bakeChickenWing();
    }
};

class Waiter         //服务员
{
private:
    Command *command;
public:
    void setOrder(Command *command)
    {
        this->command = command;
    }
    void notify()
    {
        command->executeCommand();
    }
};

class Waiter2   //gei多个对象下达命令
{
private:
    list<Command*> orders;
public:
    void setOrder(Command *command)
    {
        orders.push_back(command);
    }
    void cancelOrder(Command *command)
    {}
    void notify()
```

```
    {
        list<Command*>::iterator iter = orders.begin();
        while (iter != orders.end())
        {
            (*iter)->executeCommand();
            iter++;
        }
    }
};

int main()
{

    Barbecuer *boy = new Barbecuer();
    Command *bm1 = new BakeMuttonCommand(boy);
    Command *bm2 = new BakeMuttonCommand(boy);
    Command *bc1 = new BakeChikenWingCommand(boy);

    Waiter2 *girl = new Waiter2();

    girl->setOrder(bm1);
    girl->setOrder(bm2);
    girl->setOrder(bc1);

    girl->notify();

    return 0;
}
```

状态模式

State模式也叫状态模式，是行为设计模式的一种。State模式允许通过改变对象的内部状态而改变对象的行为，这个对象表现得就好像修改了它的类一样。

状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转译到表现不同状态的一系列类当中，可以把复杂的判断逻辑简化。

状态模式：状态模式允许一个对象在其内部状态改变的时候改变行为。

这个对象看上去象是改变了它的类一样。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式需要对每一个系统可能取得的状态创立一个状态类的子类。当系统的状态变化时，系统便改变所选的子类。

跟MM交往时，一定要注意她的状态哦，在不同的状态时她的行为会有不同，比如你约她今天晚上去看电影，对你没兴趣的MM就会说“有事情啦”，对你不讨厌但还没喜欢上的MM就会说“好啊，不过可以带上我同事么？”，已经喜欢上你的MM就会说“几点钟？看完电影再去泡吧怎么样？”，当然你看电影过程中表现良好的话，也可以把MM的状态从不讨厌不喜欢变成喜欢哦。

每个人、事物在不同的状态下会有不同表现（动作），而一个状态又会在不同的表现下转移到下一个不同的状态（State）。最简单的一个生活中的例子就是：地铁入口处，如果你放入正确的地铁票，门就会打开让你通过。在出口处也是验票，如果正确你就可以ok，否则就不让你通过（如果你动作野蛮，或许会有报警（Alarm））。

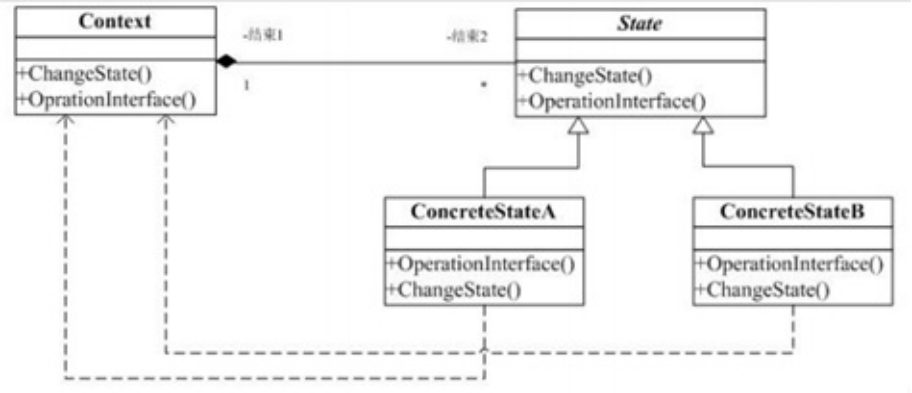
有限状态自动机（FSM）也是一个典型的状态不同，对输入有不同的响应（状态转移）。通常我们在实现这类系统会使用到很多的Switch/Case语句，Case某种状态，发生什么动作，Case另外一种状态，则发生另外一种状态。但是这种实现方式至少有以下两个问题：

1）当状态数目不是很多的时候，Switch/Case可能可以搞定。但是当状态数目很多的时候（实际系统中也正是如此），维护一大组的Switch/Case语句将是一件异常困难并且容易出错的事情。

2）状态逻辑和动作实现没有分离。在很多的系统实现中，动作的实现代码直接写在状态的逻辑当中。这带来的后果就是系统的扩展性和维护得不到保证。

类图角色和职责

State 模式就是被用来解决上的两个问题的，在 State 模式中我们将状态逻辑和动作实现分离。当一个操作中要维护大量的case分支语句，并且这些分支依赖于对象的状态。State 模式将每一个分支封装到一个独立的类中。State 模式的典型结构图为：



- Context：用户对象拥有一个State类型的成员，以标识对象的当前状态；
- State：接口或基类封装与Context的特定状态相关的行为；
- ConcreteState：接口实现类或子类实现了一个与Context某个状态相关的行为。

适用于：对象的行为，依赖于它所处的当前状态。行为随状态改变而改变的场景。

适用于：通过用户的状态来改变对象的行为。

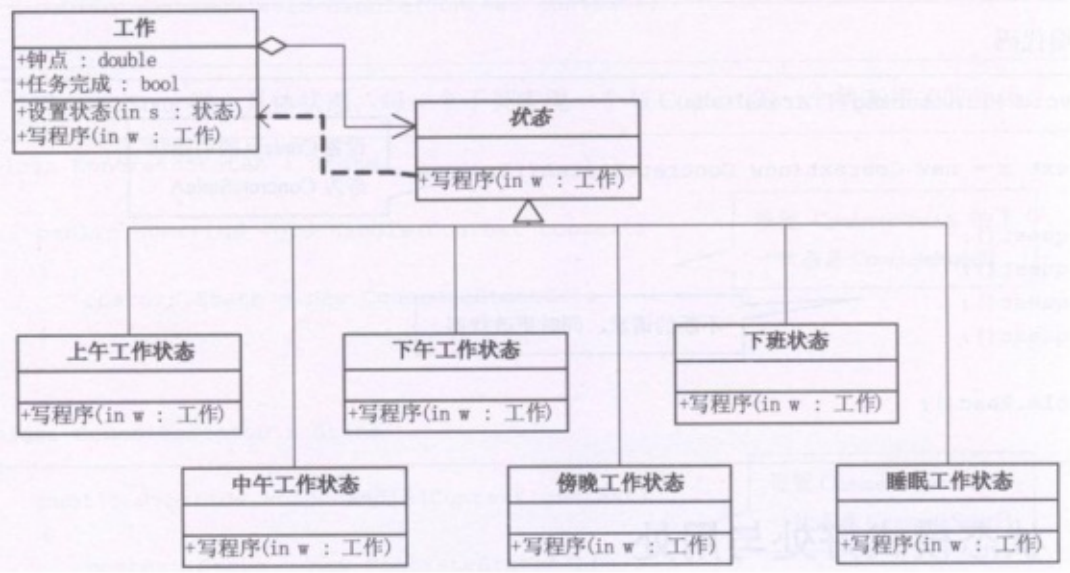
示例代码

```
#include <iostream>
using namespace std;

class Worker;

class State
{
public:
    virtual void doSomeThing(Worker *w) = 0;
```

```
};

class Worker
{
public:
    Worker();

    int getHour()
    {
        return m_hour;
    }

    void setHour(int hour) //改变状态 7
    {
        m_hour = hour;
    }

    State* getCurrentState()
    {
        return m_currstate;
    }

    void setCurrentState(State* state)
    {
        m_currstate = state;
    }

    void doSomeThing()
    {
        m_currstate->doSomeThing(this);
    }
private:
    int         m_hour;
    State    *m_currstate; //对象的当前状态
};

class State1 : public State
{
public:
    void doSomeThing(Worker *w);
```

```
};

class State2  : public State
{
public:
    void doSomeThing(Worker *w);
};

void State1::doSomeThing(Worker *w)
{
    if (w->getHour() == 7 || w->getHour()==8)
    {
        cout << "吃早饭" << endl;
    }
    else
    {
        delete w->getCurrentState();  //状态1不满足要转到状态2
        w->setCurrentState(new State2 );
        w->getCurrentState()->doSomeThing(w);
    }
}

void State2::doSomeThing(Worker *w)
{
    if (w->getHour() == 9 || w->getHour()==10)
    {
        cout << "工作" << endl;
    }
    else
    {    //状态2不满足要转到状态3，后者恢复到初始化状态
        delete w->getCurrentState();
        w->setCurrentState(new State1); //恢复到当初状态
        cout << "当前时间点：" << w->getHour() << "未知状态" << endl
;
    }
}

Worker::Worker()
{
    m_currstate = new State1;
```

```
}

void main()
{
    Worker *w1 = new Worker;
    w1->setHour(7);
    w1->doSomeThing();

    w1->setHour(9);
    w1->doSomeThing();

    delete w1;

    system("pause");
    return ;
}
```




```
#include <iostream>
#include <string>
using namespace std;

class Work;
class State;
class ForenonnState;

class State
```

```
{
public:
    virtual void writeProgram(Work*) = 0;//准柜台的基类，抽象类
};

class Work   //实施工作的类，根据状态执行不同的操作
{
public:
    int hour;
    State *current;
    Work();


    void writeProgram()
    {
        current->writeProgram(this);
    }
};

class EveningState :public State  //晚上状态
{
public:
    void writeProgram(Work *w)
    {
        cout << "当前时间: " << w->hour << "心情很好，在看《明朝那些事儿》，收获很大！" << endl;
    }
};

class AfternoonState :public State
{
public:
    void writeProgram(Work *w)
    {
        if (w->hour < 19)
        {
            cout << "当前时间: " << w->hour << "下午午睡后，工作还是精神百倍!" << endl;
        }
        else
```

```
        {
            w->current = new EveningState();
            w->writeProgram();
        }
    }
};

class ForenonnState :public State
{
public:
    void writeProgram(Work *w)
    {
        if (w->hour < 12)
        {
            cout << "当前时间: " << w->hour << "上午工作精神百倍!" <
< endl;
        }
        else
        {
            w->current = new AfternoonState();
            w->writeProgram();
        }
    }
};

Work::Work()
{
    current = new ForenonnState();
}

int main()
{
    Work *w = new Work();
    w->hour = 21;
    w->writeProgram();
    cin.get();
    return 0;
}
```

访问者模式

Visitor模式也叫访问者模式，是行为模式之一，它分离对象的数据和行为，使用Visitor模式，可以不修改已有类的情况下，增加新的操作角色和职责。

访问者模式：访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。

访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由的演化。访问者模式使得增加新的操作变的很容易，就是增加一个新的访问者类。

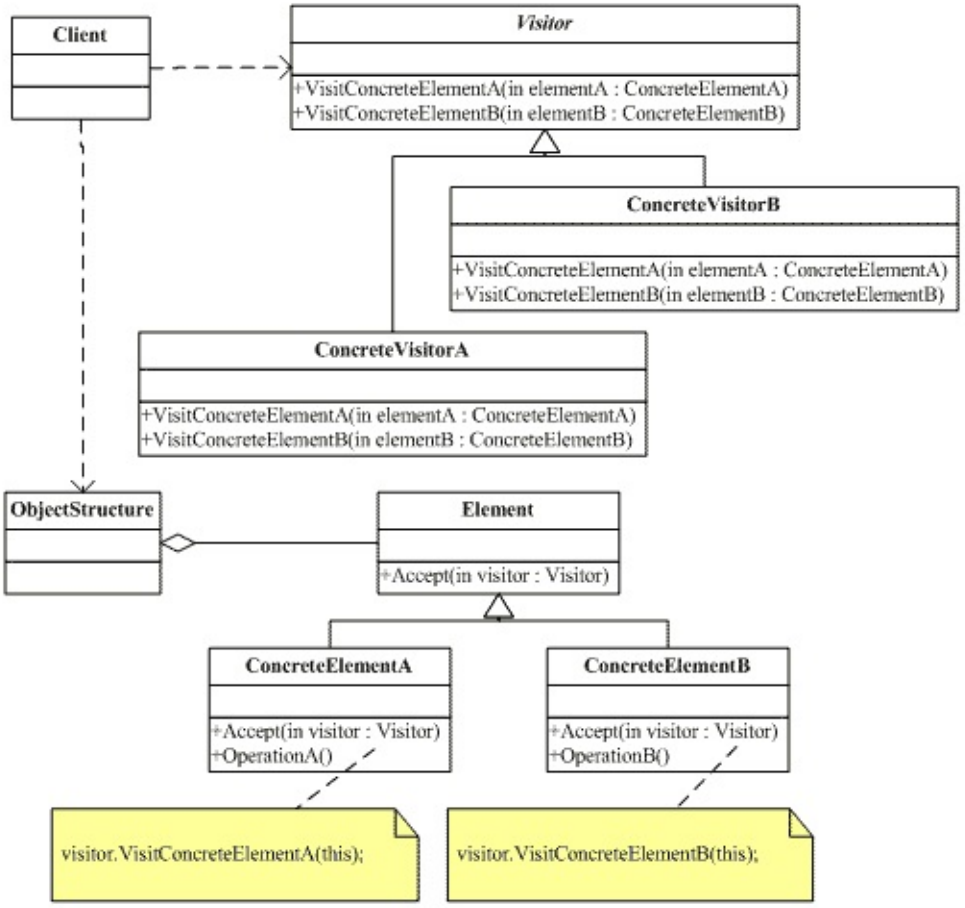
访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。当使用访问者模式时，要将尽可能多的对象浏览逻辑放在访问者类中，而不是放到它的子类中。

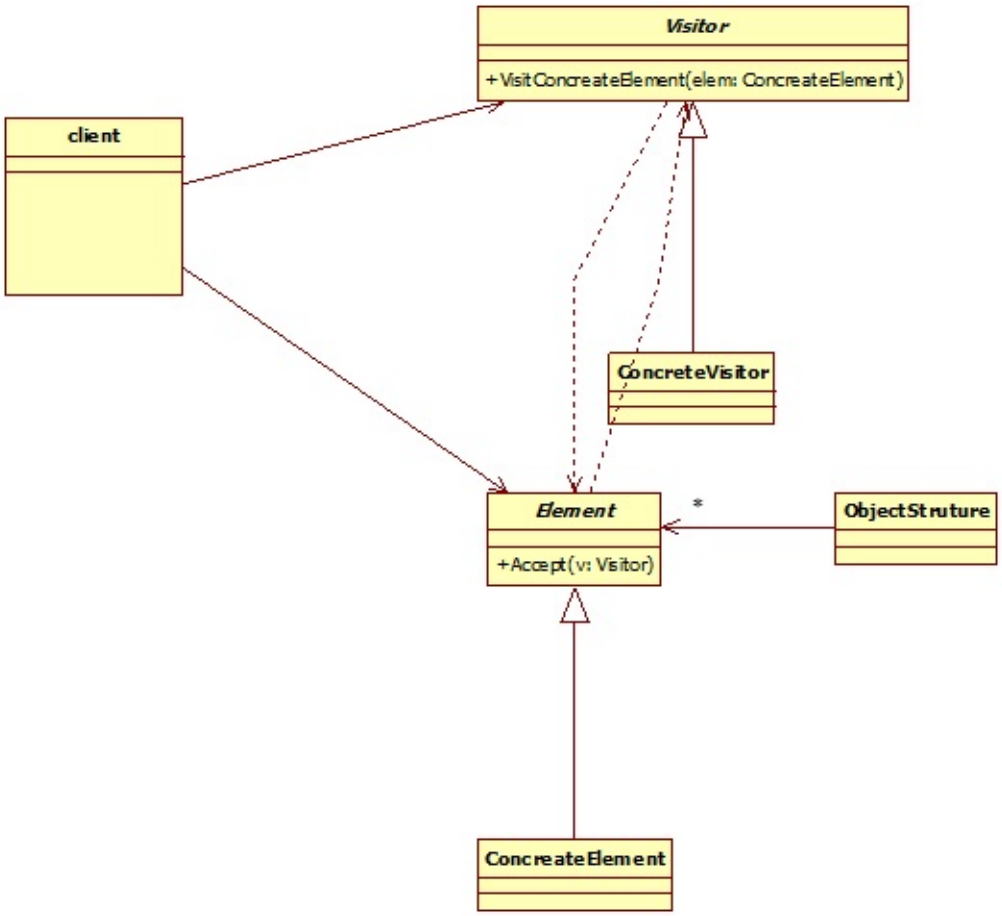
访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。

情人节到了，要给每个MM送一束鲜花和一张卡片，可是每个MM送的花都要针对她个人的特点，每张卡片也要根据个人的特点来挑，我一个人哪搞得清楚，还是找花店老板和礼品店老板做一下Visitor，让花店老板根据MM的特点选一束花，让礼品店老板也根据每个人特点选一张卡，这样就轻松多了。

访问者模式不需要改变基类，不依赖虚函数，

类图角色和职责


Client
Visitor
+VisitConcreteElementA(in elementA : ConcreteElementA)
+VisitConcreteElementB(in elementB : ConcreteElementB)
ConcreteVisitorB
+VisitConcreteElementA(in elementA : ConcreteElementA)
+VisitConcreteElementB(in elementB : ConcreteElementB)
ConcreteVisitorA
+VisitConcreteElementA(in elementA : ConcreteElementA)
+VisitConcreteElementB(in elementB : ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this);
visitor.VisitConcreteElementB(this);


Visitor
+VisitConcreateElement(elem: ConcreateElement)
client
ConcreteVisitor
Element
+Accept(v: Visitor)
*
ObjectStruture
ConcreateElement

抽象访问者（Visitor）角色：声明了一个或者多个访问操作，形成所有的具体元素角色必须实现的接口。

具体访问者（ConcreteVisitor）角色：实现抽象访问者角色所声明的接口，也就是抽象访问者所声明的各个访问操作。

抽象节点（Element）角色：声明一个接受操作，接受一个访问者对象作为一个参量。

具体节点（ConcreteElement）角色：实现了抽象元素所规定的接受操作。

结构对象（ObiectStructure）角色：有如下的一些责任，可以遍历结构中的所有元素；如果需要，提供一个高层次的接口让访问者对象可以访问每一个元素；如果需要，可以设计成一个复合对象或者一个聚集，如列（List）或集合（Set）。

适用于：把数据结构 和 作用于数据结构上的操作 进行解耦合;适用于数据结构比较稳定的场合。

访问者模式总结：

访问者模式优点是增加新的操作很容易，因为增加新的操作就意味着增加一个新的访问者。访问者模式将有关的行为集中到一个访问者对象中。

那访问者模式的缺点是是增加新的数据结构变得困难了

优缺点

访问者模式有如下的优点：

1，访问者模式使得增加新的操作变得很容易。如果一些操作依赖于一个复杂的结构对象的话，那么一般而言，增加新的操作会很复杂。而使用访问者模式，增加新的操作就意味着增加一个新的访问者类，因此，变得很容易。

2，访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。

3，访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。迭代子只能访问属于同一个类型等级结构的成员对象，而不能访问属于不同等级结构的对象。访问者模式可以做到这一点。

4，积累状态。每一个单独的访问者对象都集中了相关的行为，从而也就可以在访问的过程中将执行操作的状态积累在自己内部，而不是分散到很多的节点对象中。这是有益于系统维护的优点。

访问者模式有如下的缺点：

1，增加新的节点类变得很困难。每增加一个新的节点都意味着要在抽象访问者角色中增加一个新的抽象操作，并在每一个具体访问者类中增加相应的具体操作。

2，破坏封装。访问者模式要求访问者对象访问并调用每一个节点对象的操作，这隐含了一个对所有节点对象的要求：它们必须暴露一些自己的操作和内部状态。不然，访问者的访问就变得没有意义。由于访问者对象自己会积累访问操作所需的状态，从而使这些状态不再存储在节点对象中，这也是破坏封装的。

案例

案例需求：比如有一个公园，有一到多个不同的组成部分；该公园存在多个访问者：清洁工A负责打扫公园的A部分，清洁工B负责打扫公园的B部分，公园的管理者负责检点各项事务是否完成，上级领导可以视察公园等等。也就是说，对于同一个公园，不同的访问者有不同的行为操作，而且访问者的种类也可能需要根据时间的推移而变化（行为的扩展性）。

根据软件设计的开闭原则（对修改关闭，对扩展开放），我们怎么样实现这种需求呢？

```
#include <iostream>
#include "list"
using namespace std;

class ParkElement;

class Visitor
{
public:
    virtual void visit(ParkElement *parkelement) = 0;
};

class ParkElement
{
```

```
public:
    virtual void accept(Visitor *visit) = 0;
};

class  ParkA : public ParkElement
{
public:
    virtual void accept(Visitor *v)
    {
        v->visit(this); //公园接受访问者访问 让访问者做操作
    }
};

class  ParkB : public ParkElement
{
public:
    virtual void accept(Visitor *v)
    {
        v->visit(this); //公园接受访问者访问 让访问者做操作
    }
};

//整个公园
class Park : public ParkElement
{
public:
    Park()
    {
        m_list.clear();
    }

    void setParkElement(ParkElement *pe)
    {
        m_list.push_back(pe);
    }

public:
    virtual void accept(Visitor *v)
    {
        //v->visit(this); //公园接受访问者访问 让访问者做操作
```

```
        for (list<ParkElement *>::iterator it = m_list.begin();
it!=m_list.end(); it++ )
        {
            (*it)->accept(v);  //公园A 公园B 接受 管理者v访问
        }
    }

private:
    list<ParkElement *> m_list; //公园的每一部分，应该让公园的每一个部
分都让管理者访问
};

class VisitorA : public Visitor
{
public:
    virtual void visit(ParkElement *parkelement)
    {    //parkelement->getName();
        cout << "清洁工A 完成 公园A部分的 打扫 " << endl;
    }
};

class VisitorB : public Visitor
{
public:
    virtual void visit(ParkElement *parkelement)
    {    //parkelement->getName();
        cout << "清洁工B 完成 公园B部分的 打扫 " << endl;
    }
};

class ManagerVisitor : public Visitor
{
public:
    virtual void visit(ParkElement *parkelement)
    {    //parkelement->getName();
        cout << "管理者 访问公园 的 各个部分 " << endl;
    }
};

void main1()
```

```
{
    Visitor *vA = new  VisitorA;
    Visitor *vB = new  VisitorB;

    ParkA *parkA = new ParkA;
    ParkB *parkB = new ParkB;

    parkA->accept(vA);
    parkB->accept(vB);

    delete vA;
    delete vB;
    delete parkA;
    delete parkB;
}

void main2()
{
    Visitor *vManager = new  ManagerVisitor ;
    Park *park = new Park;

    ParkElement *parkA = new ParkA;
    ParkElement *parkB = new ParkB;

    park->setParkElement(parkA);
    park->setParkElement(parkB);

    //整个公园 接受 管理者访问
    park->accept(vManager);

    delete parkA;
    delete parkB;
    delete park;
    delete vManager;
}

void main()
{
    //main1();
    main2();
```

```
    system("pause");
    return ;
}
```

示例代码

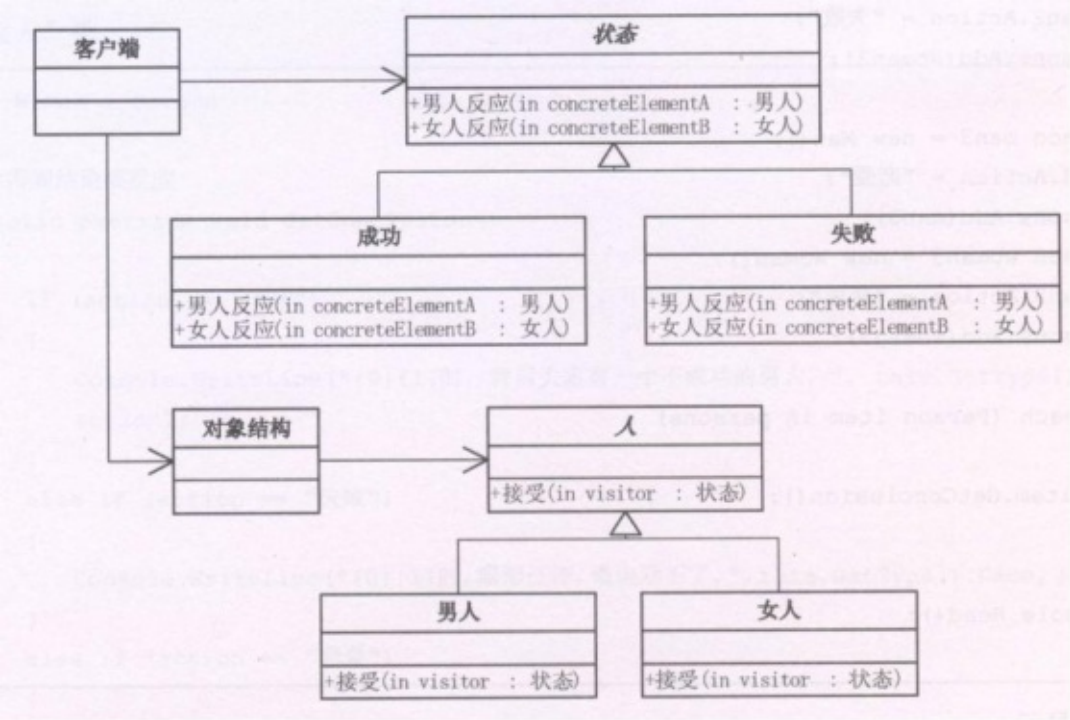
```
#include <iostream>
#include "string"
#include "list"
using namespace std;

//客户去银行办理业务
//m个客户
//n个柜员

//将要对象和要处理的操作分开，不同的柜员可以办理不同来访者的业务

class Element;

//访问者访问柜员
class Visitor
{
public:
    virtual void visit(Element *element) = 0;
};

//柜员接受客户访问
class Element
{
public:
    virtual void accept(Visitor *v) = 0;
    virtual string getName() = 0;
};

//柜员A员工
class EmployeeA : public Element
{
public:
```

```
    EmployeeA(string name)
    {
        m_name = name;
    }

    virtual void accept(Visitor *v)
    {
        v->visit(this);
    }

    virtual string getName()
    {
        return m_name;
    }
private:
    string m_name;
};

//柜员B员工
class EmployeeB : public Element
{
public:
    EmployeeB(string name)
    {
        m_name = name;
    }

    virtual void accept(Visitor *v)
    {
        v->visit(this);
    }

    string getName()
    {
        return m_name;
    }
private:
    string m_name;
};
```

```
class VisitorA : public Visitor
{
public:
    virtual void visit(Element *element)
    {
        cout << "通过" << element->getName() << "做A业务" << endl;
    }
};

class VisitorB : public Visitor
{
public:
    virtual void visit(Element *element)
    {
        cout << "通过" << element->getName() << "做B业务" << endl;
    }
};

void main1()
{
    EmployeeA *eA = new EmployeeA("柜员A");

    VisitorA *vA = new VisitorA;
    VisitorB *vB = new VisitorB;

    eA->accept(vA);
    eA->accept(vB);

    delete eA;
    delete vA;
    delete vB;
    return ;
}

//柜员B员工
class Employees : public Element
{
public:
    Employees()
    {
```

```
        m_list = new list<Element *>;
    }
    virtual void accept(Visitor *v)
    {
        for (list<Element *>::iterator it = m_list->begin();
             it != m_list->end(); it++  )
        {
            (*it)->accept(v);
        }
    }

    string getName()
    {
        return m_name;
    }
public:
    void addElement(Element *e)
    {
        m_list->push_back(e);
    }

    void removeElement(Element *e)
    {
        m_list->remove(e);
    }
private:
    list<Element *> *m_list;
    string m_name;
};

void main2()
{
    EmployeeA *eA = new EmployeeA("柜员A");
    EmployeeA *eB= new EmployeeA("柜员B");

    Employees *es = new Employees;
    es->addElement(eA);
    es->addElement(eB);
    VisitorA *vA = new VisitorA;
    VisitorB *vB = new VisitorB;
```

```
    es->accept(vA);
    cout << "-------------" << endl;
    es->accept(vB);

    delete eA;
    delete eB;
    delete vA;
    delete vB;

    return ;
}

void main()
{
    //main1();
    main2();
    system("pause");
}
```




```
#include <iostream>
#include <list>
#include <string>
```

```
using namespace std;

class Person
{
public:
    char * action;

    virtual void getConclusion()
    {

    };
};

class Man :public Person
{
public:

    void getConclusion()
    {
        if (action == "成功")
        {
            cout << "男人成功时，背后多半有一个伟大的女人。" << endl;

        }
        else if (action == "恋爱")
        {
            cout << "男人恋爱时，凡事不懂装懂。" << endl;
        }
    }
};

class Woman :public Person
{
public:

    void getConclusion()
    {
        if (action == "成功")
        {
            cout << "女人成功时，背后多半有失败的男人。" << endl;
```

```
        }
        else if (action == "恋爱")
        {
            cout << "女人恋爱时，遇到事懂也装不懂。" << endl;
        }
    }
};

int main()
{
    list<Person*> persons;

    Person *man1 = new Man();
    man1->action = "成功";
    persons.push_back(man1);

    Person *woman1 = new Woman();
    woman1->action = "成功";
    persons.push_back(woman1);

    Person *man2 = new Man();
    man2->action = "恋爱";
    persons.push_back(man2);

    Person *woman2 = new Woman();
    woman2->action = "恋爱";
    persons.push_back(woman2);

    list<Person*>::iterator iter = persons.begin();
    while (iter != persons.end())
    {
        (*iter)->getConclusion();
        ++iter;
    }

    cin.get();
    return 0;
}
```

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Man;
class Woman;
//行为
class Action
{
public:
    virtual void GetManConclusion(Man* concreteElementA)=0;
    virtual void GetWomanConclusion(Woman* concreteElementB)=0;
};
//成功
class Success : public Action
{
public:
    virtual void GetManConclusion(Man* concreteElementA)
    {
        cout<<"男人成功时，背后有个伟大的女人"<<endl;
    }
    virtual void GetWomanConclusion(Woman* concreteElementB)
    {
        cout<<"女人成功时，背后有个没用的男人"<<endl;
    }
};

//失败
class Failure : public Action
{
public:
    virtual void GetManConclusion(Man* concreteElementA)
    {
        cout<<"男人失败时，背后有个伟大的女人"<<endl;
    }
    virtual void GetWomanConclusion(Woman* concreteElementB)
    {
        cout<<"女人失败时，背后有个没用的男人"<<endl;
```

```
    }
};

//抽象人类
class Person
{
public:
    virtual void Accept(Action* visitor)=0;
};

//男人
class Man : public Person
{
public:
    virtual void Accept(Action* visitor)
    {
        visitor->GetManConclusion(this);
    }
};

//女人
class Woman : public Person
{
public:
    virtual void Accept(Action* visitor)
    {
        visitor->GetWomanConclusion(this);
    }
};

//对象结构类
class ObjectStructure
{
private:
    vector<Person*> m_personList;

public:
    void Add(Person* p)
    {
        m_personList.push_back(p);
```

```
    }
    void Display(Action* a)
    {
        vector<Person*>::iterator p = m_personList.begin();
        while (p!=m_personList.end())
        {
            (*p)->Accept(a);
            p++;
        }
    }
};

//客户端
int main()
{
    ObjectStructure * os= new ObjectStructure();
    os->Add(new Man());
    os->Add(new Woman());

    Success* success = new Success();
    os->Display(success);

    Failure* fl = new Failure();
    os->Display(fl);

    return 0;
}
```