C/C++与设计模式基础课程

传智扫地僧

设计模式基础

1 设计模式编程基础

1.1 设计模式前言

模式

在一定环境中解决某一问题的方案，包括三个基本元素--问题，解决方案和环境。

大白话：**在一定环境下，用固定套路解决问题。**

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脉络，如同大厦的结构一样。

学习设计模式的意义

提高职业素养，关注学员在行业内的长期发展。

“我眼中的设计模式”

把简单的问题复杂化（标准化），把环境中的各个部分进行抽象、归纳、解耦合。

不是多神秘的东西，我们初学者也能学的会。要有信心。

学习设计模式的方法

对初学者：

积累案例，大于背类图。

初级开发人员：

多思考、多梳理，归纳总结；

尊重事物的认知规律，注意事物临界点的突破。不可浮躁。

中级开发人员

合适的开发环境，寻找合适的设计模式，解决问题。

多应用

对经典组合设计模式的大量、自由的运用。要不断的追求。

设计模式的分类

Gang of Four 的“Design Patterns: Elements of Resualbel Software”书将设计模式归纳为三大类型，共 23 种。

创建型模式： 通常和对象的创建有关，涉及到对象实例化的方式。(共 5 种模式)

结构型模式： 描述的是如何组合类和对象以获得更大的结构。(共 7 种模式)

行为型模式： 用来对类或对象怎样交互和怎样分配职责进行描述。(共 11 种模式)

创建型模式用来处理对象的创建过程，主要包含以下 5 种设计模式：

1，工厂方法模式（Factory Method Pattern）的用意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中。

2，抽象工厂模式（Abstract Factory Pattern）的意图是提供一个创建一系列相关或者相互依赖的接口，而无需指定它们具体的类。

3，建造者模式（Builder Pattern）的意图是将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

4，原型模式（Prototype Pattern）是用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

5，单例模式（Singleton Pattern）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。

结构型模式用来处理类或者对象的组合，主要包含以下 7 种设计模式：

6，代理模式（Proxy Pattern）就是为其他对象提供一种代理以控制对这个对象的访问。

7，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。

8，适配器模式（Adapter Pattern）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。

9，桥接模式（Bridge Pattern）是将抽象部分与实际部分分离，使它们都可以独立的变化。

10，组合模式（Composite Pattern）是将对象组合成树形结构以表示“部分--整体”的层次结构。使得用户对单个对象和组合对象的使用具有一致性。

11，外观模式（Facade Pattern）是为子系统中的一组接口提供一个一致的界面，此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。

12，享元模式（Flyweight Pattern）是以共享的方式高效的支持大量的细粒度的对象。

行为型模式用来对类或对象怎样交互和怎样分配职责进行描述，主要包含以下 11 种设计模式：

13，模板方法模式（Template Method Pattern）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。

14，命令模式（Command Pattern）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。

15，责任链模式（Chain of Responsibility Pattern），在该模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。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，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

16，策略模式（Strategy Pattern）就是准备一组算法，并将每一个算法封装起来，使得它们可以互换。

17，中介者模式（Mediator Pattern）就是定义一个中介对象来封装系列对象之间的交互。终结者使各个对象不需要显示的相互调用 ，从而使其耦合性松散，而且可以独立的改变他们之间的交互。

18，观察者模式（Observer Pattern）定义对象间的一种一对多的依赖关系，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被自动更新。
19，备忘录模式（Memento Pattern）是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。
20，访问者模式（Visitor Pattern）就是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。
21，状态模式（State Pattern）就是对象的行为，依赖于它所处的状态。
22，解释器模式（Interpreter Pattern）就是描述了如何为简单的语言定义一个语法，如何在该语言中表示一个句子，以及如何解释这些句子。
23，迭代器模式（Iterator Pattern）是提供了一种方法顺序来访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。

1.2 设计模式基本原则

最终目的：高内聚，低耦合

1) 开放封闭原则 (OCP,Open For Extension, Closed For Modification Principle)
类的改动是通过增加代码进行的，而不是修改源代码。

2) 单一职责原则 (SRP,Single Responsibility Principle)
类的职责要单一，对外只提供一种功能，而引起类变化的原因都应该只有一个。

3) 依赖倒置原则 (DIP,Dependence Inversion Principle)
依赖于抽象(接口),不要依赖具体的实现(类)，也就是针对接口编程。

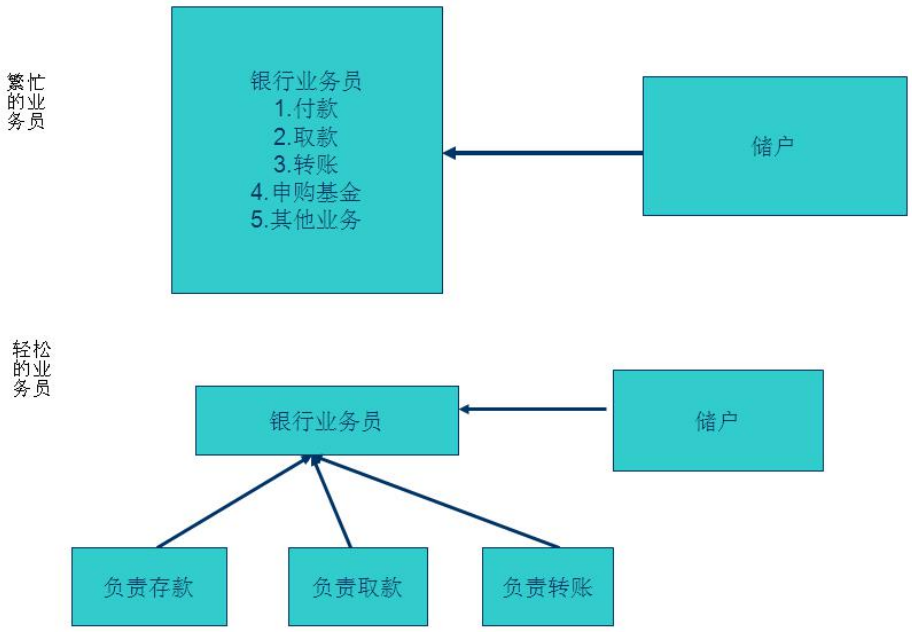
4) 接口隔离原则 (ISP,Interface Segegation Principle)
不应该强迫客户的程序依赖他们不需要的接口方法。一个接口应该只提供一种对外功能，不应该把所有操作都封装到一个接口中去。

5) 里氏替换原则 (LSP, Liskov Substitution Principle)
任何抽象类出现的地方都可以用他的实现类进行替换。实际就是虚拟机制，语言级别实现面向对象功能。

6) 优先使用组合而不是继承原则(CARP,Composite/Aggregate Reuse Principle)
如果使用继承，会导致父类的任何变换都可能影响到子类的行为。
如果使用对象组合，就降低了这种依赖关系。

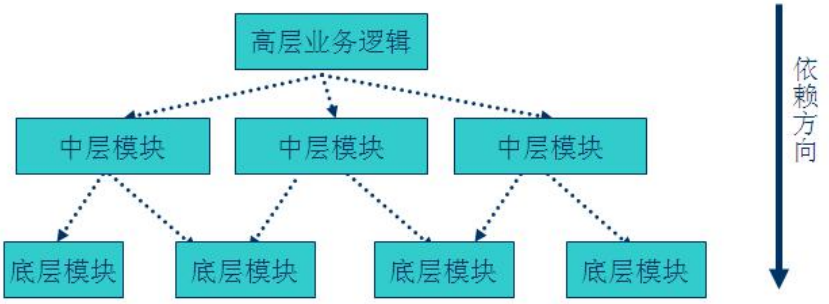
7) 迪米特法则(LOD,Law of Demeter)
一个对象应当对其他对象尽可能少的了解，从而降低各个对象之间的耦合，提高系统的可维护性。例如在一个程序中，各个模块之间相互调用时，通常会提供一个统一的接口来实现。这样其他模块不需要了解另外一个模块的内部实现细节，这样当一个模块内部的实现发生改变时，不会影响其他模块的使用。(黑盒原理)

案例图
开闭原则案例



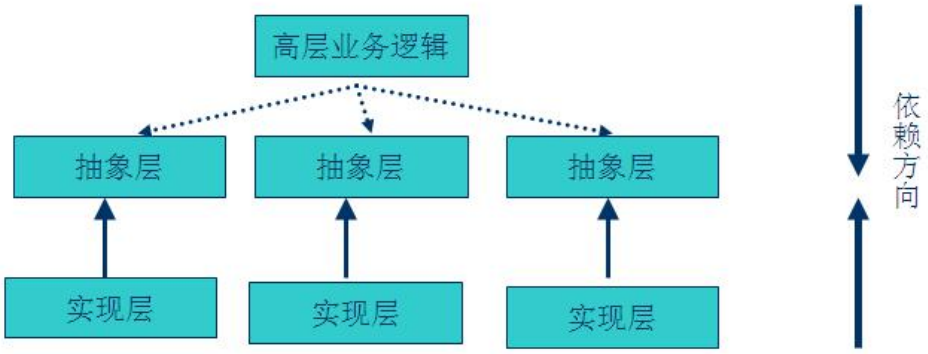
依赖倒转

1）



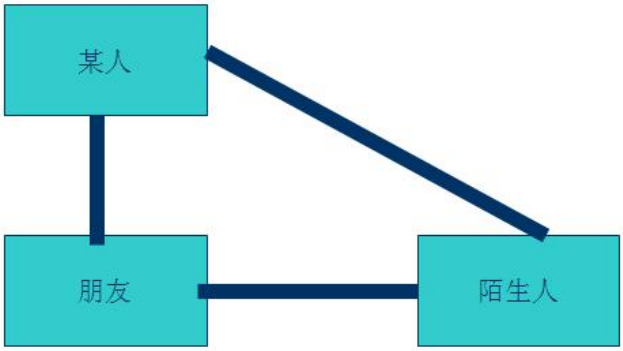
传统的过程式设计倾向于使高层次的模块依赖于低层次的模块，抽象层依赖于具体的层次。

2）

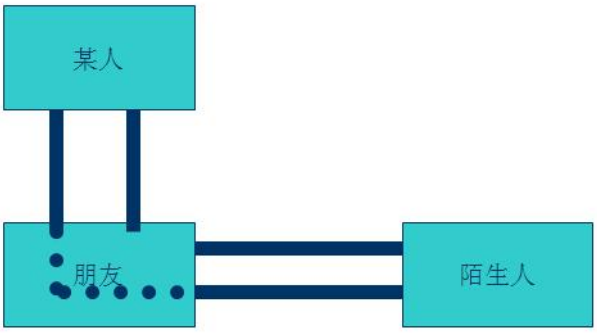


迪米特法则

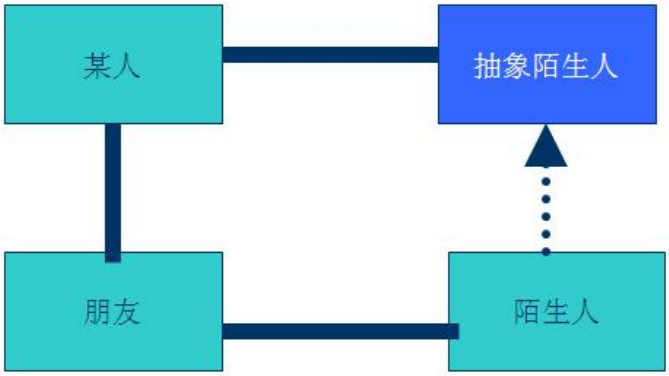
1)和陌生人说话



2)不和陌生人说话



3) 与依赖倒转原则结合 某人和 抽象陌生人说话 让某人和陌生人进行解耦合



2 创建型模式

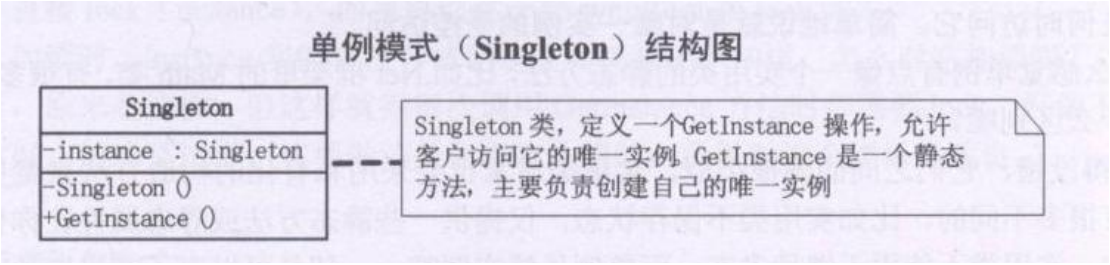
2.1 单例模式

2.2.1 概念

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。



2.2.2 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：
- 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。

2.2.3 实现单例步骤常用步骤

a) 构造函数私有化
b) 提供一个全局的静态方法（全局访问点）
c) 在类中定义一个静态指针，指向本类的变量的静态变量指针

2.2.4 饿汉式单例和懒汉式单例

懒汉式

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

//懒汉式
class   Singelton
{
private:
    Singelton()
    {
        m_singer = NULL;
        m_count = 0;
        cout << "构造函数 Singelton ... do" << endl;
    }
```

```
public:
    static Singelton *getInstance()
    {
        if (m_singer == NULL )  //懒汉式：1 每次获取实例都要判断 2 多线程会有问题
        {
            m_singer = new Singelton;
        }
        return m_singer;
    }
    static void printT()
    {
        cout << "m_count: " << m_count << endl;
    }

private:
    static Singelton *m_singer;
    static int m_count;
};

Singelton *Singelton::m_singer = NULL;  //懒汉式 并没有创建单例对象
int Singelton::m_count = 0;


void main01_1()
{
    cout << "演示 懒汉式" << endl;
    Singelton *p1 = Singelton::getInstance(); //只有在使用的时候，才去创建对象。
    Singelton *p2 = Singelton::getInstance();
    if (p1 != p2)
    {
        cout << "不是同一个对象" << endl;
    }
    else
    {
        cout << "是同一个对象" << endl;
    }
    p1->printT();
    p2->printT();

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

```
//////////////////////////////////////////////////////////////////////////
//俄汉式

class   Singelton2
{
private:
    Singelton2()
    {
        m_singer = NULL;
        m_count = 0;
        cout << "构造函数 Singelton ... do" << endl;
    }

public:
    static Singelton2 *getInstance()
    {
//      if (m_singer == NULL )
//      {
//          m_singer = new Singelton2;
//      }
        return m_singer;
    }
    static void Singelton2::FreeInstance()
    {
        if (m_singer != NULL)
        {
            delete m_singer;
            m_singer = NULL;
            m_count = 0;
        }
    }
    static void printT()
    {
        cout << "m_count: " << m_count << endl;
    }

private:
    static Singelton2 *m_singer;
    static int m_count;
};

Singelton2 *Singelton2::m_singer = new Singelton2; //不管你创建不创建实例，均把实例 new 出来
```

```
int Singelton2::m_count = 0;

void main()
{
    cout << "演示 饿汉式" << endl;

    Singelton2 *p1 = Singelton2::getInstance(); //只有在使用的时候，才去创建对象。
    Singelton2 *p2 = Singelton2::getInstance();
    if (p1 != p2)
    {
        cout << "不是同一个对象" << endl;
    }
    else
    {
        cout << "是同一个对象" << endl;
    }
    p1->printT();
    p2->printT();
    Singelton2::FreeInstance();
    Singelton2::FreeInstance();


    system("pause");
}
```

2.2.5 多线程下的懒汉式单例和饿汉式单例

```
//1"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。
//2 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。
//3 提供释放资源的函数
```

讨论: 这是因为 C++中构造函数并不是线程安全的。

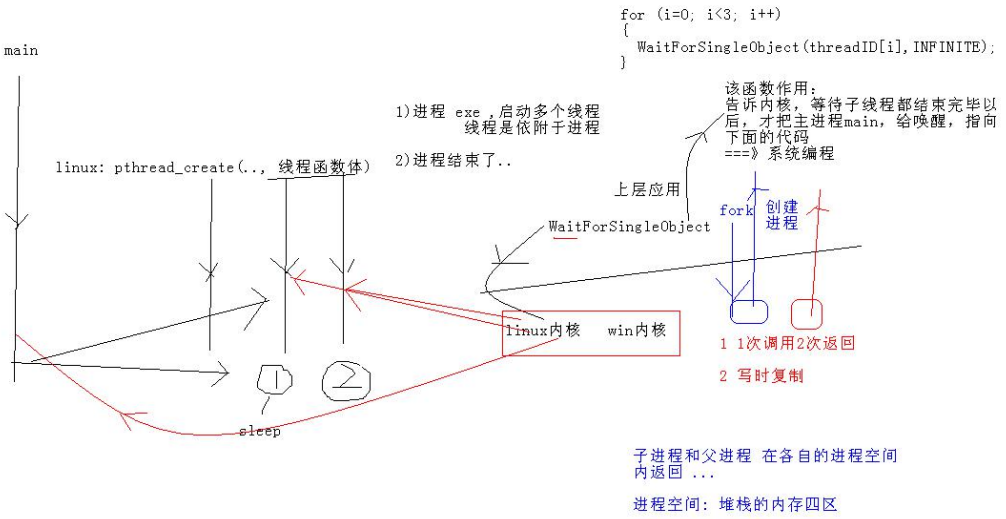
C++中的构造函数简单来说分两步：

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。

多线程下的懒汉式问题抛出：




```
#include "stdafx.h"
#include "windows.h"
#include "winbase.h"
#include <process.h>
#include "iostream"

using namespace std;
class Singelton
{
private:
    Singelton()
    {
        count ++;
        cout<<"Singelton 构造函数 begin\n"<<endl;
        Sleep(1000);
        cout<<"Singelton 构造函数 end\n"<<endl;

    }
private:
    //防止拷贝构造和赋值操作
    Singelton(const Singelton &obj) { ;}
    Singelton& operator=(const Singelton &obj)     { ;}
public:
    static Singelton *getSingelton()
```

```
    {
        //1"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。
        //2 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。
        //3 提供释放资源的函数
        return single;
    }

    static Singelton *releaseSingelton()
    {
        if (single != NULL) //需要判断
        {
            cout<<"释放资源\n"<<endl;
            delete single;
            single = NULL;
        }
        return single;
    }
    void pirntS() //测试函数
    {
        printf("Singelton printS test count:%d \n", count);
    }

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

//note 静态变量类外初始化
Singelton *Singelton::single = new Singelton();
int Singelton::count = 0;

int _tmainTTT(int argc, _TCHAR* argv[])
{
    Singelton *s1 = Singelton::getSingelton();
    Singelton *s2 = Singelton::getSingelton();
    if (s1 == s2)
    {
        cout<<"ok....equal"<<endl;
    }
    else
    {
        cout<<"not.equal"<<endl;
    }
```

```
    s1->pirntS();
    Singelton::releaseSingelton();
    cout <<"hello...."<<endl;
    system("pause");
    return 0;
}

unsigned int threadfunc2(void *myIpAdd)
{
    int id = GetCurrentThreadId();
    printf("\n threadfunc%d \n", id);
    return 1;
}

void threadfunc(void *myIpAdd)
{
    int id = GetCurrentThreadId();
    printf("\n threadfunc%d \n", id);
     Singelton::getSingelton()->pirntS();
    return ;
}

 int _tmain(int argc, _TCHAR* argv[])
{
    int i = 0;
    DWORD dwThreadId[201], dwThrdParam = 1;
    HANDLE hThread[201];
    int threadnum = 3;

    for (i=0; i<threadnum; i++)
    {
        //hThread[i] = (HANDLE)_beginthreadex( NULL, 0, &threadfunc, NULL,
0,&dwThreadId[i] );
        hThread[i] = (HANDLE)_beginthread(&threadfunc, 0 , 0 );
        if (hThread[i] == NULL)
        {
            printf("begin thread %d error!!!\n", i);
            break;
        }
    }
    //等待所有的子线程都运行完毕后,才执行 这个代码
    for (i=0; i<threadnum; i++)
    {
        WaitForSingleObject( hThread[i], INFINITE );
```

```
    }
    printf("等待线程结束\n");
    for (i=0; i<threadnum; i++)
    {
        //CloseHandle( hThread[i] );
    }
    Singelton::releaseSingelton();
    cout <<"hello...."<<endl;
    system("pause");
    return 0;
}
```

2.2.6 多线程下懒汉式单例的 Double-Checked Locking 优化

新建 MFC 对话框应用程序。
方便使用临界区类对象，同步线程

```
// MFC Diagram 应用程序
#include "stdafx.h"
#include "01 单例优化.h"
#include "01 单例优化 Dlg.h"
#include "afxdialogex.h"

#include "iostream"
using namespace std;

//临界区
static CCriticalSection cs;
//man pthread_create()
class Singleton
{
private:
    Singleton()
    {
        TRACE("Singleton begin\n");
        Sleep(1000);
        TRACE("Singleton end\n");

    }
    Singleton(const Singleton &);
    Singleton& operator = (const Singleton &);

public:
```

```
    static void printV()
    {
        TRACE("printV..\n");
    }

//请思考 ;懒汉式的 Double-Check 是一个经典问题 !为什么需要 2 次检查 “if(pInstance == NULL)”
场景：假设有线程 1、线程 2、线程 3，同时资源竞争。
//1）第 1 个、2 个、3 个线程执行第一个检查，都有可能进入黄色区域（临界区）
//2）若第 1 个线程进入到临界区，第 2 个、第 3 个线程需要等待
//3）第 1 个线程执行完毕，cs.unlock()后，第 2 个、第 3 个线程要竞争执行临界区代码。
//4）假若第 2 个线程进入临界区，此时第 2 个线程需要再次判断 if(pInstance == NULL)”，若第一个线程已经创建实例；第 2 个线程就不需要再次创建了。保证了单例；
//5）同样道理，若第 2 个线程，cs.unlock()后，第 3 个线程会竞争执行临界区代码；此时第 3 个线程需要再次判断 if(pInstance == NULL) 通过检查发现实例已经 new 出来，就不需要再次创建；保证了单例。

    static Singleton *Instantialize()
    {
        if(pInstance == NULL)   //double check
        {
            cs.Lock(); //只有当 pInstance 等于 null 时，才开始使用加锁机制 二次检查
            if(pInstance == NULL)
            {
                pInstance = new Singleton();
            }
            cs.Unlock();
        }
        return pInstance;
    }
    static Singleton *pInstance;

};

Singleton* Singleton::pInstance = 0;

void CMy01 单例优化 Dlg::OnBnClickedButton1()
{
    CCriticalSection   cs;
    cs.Lock();

    cs.Unlock();
    // TODO: 在此添加控件通知处理程序代码
}
```

```
void threadfunc(void *myIpAdd)
{
    int id = GetCurrentThreadId();
    TRACE("\n threadfunc%d \n", id);
    Singleton::Instantialize()->printV();
    //Singelton::getSingelton()->pirntS();
}

void CMy01 单例优化 Dlg::OnBnClickedButton2()
{
    int i = 0;
    DWORD dwThreadId[201], dwThrdParam = 1;
    HANDLE hThread[201];
    int threadnum = 3;

    for (i=0; i<threadnum; i++)
    {
        //hThread[i] = (HANDLE)_beginthreadex( NULL, 0, &threadfunc, NULL,
0,&dwThreadId[i] );
        hThread[i] = (HANDLE)_beginthread(&threadfunc, 0 , 0 );
        if (hThread[i] == NULL)
        {
            TRACE("begin thread %d error!!!\n", i);
            break;
        }
    }

    for (i=0; i<threadnum; i++)
    {
        WaitForSingleObject( hThread[i], INFINITE );
    }
    TRACE("等待线程结束\n");
    for (i=0; i<threadnum; i++)
    {
        //CloseHandle( hThread[i] );
    }
    //Singelton::releaseSingelton();

    TRACE("ddddd\n");
}
```

2.2.7 程序并发机制扩展阅读

程序的并发执行往往带来与时间有关的错误，甚至引发灾难性的后果。这需要引入同步机制。使用多进程与多线程时，有时需要协同两种或多种动作，此过程就称同步（Synchronization）。引入同步机制的第一个原因是为了控制线程之间的资源同步访问，因为多个线程在共享资源时如果发生访问冲突通常会带来不正确的后果。例如，一个线程正在更新一个结构，同时另一个线程正试图读取同一个结构。结果，我们将无法得知所读取的数据是新的还是旧的，或者是二者的混合。第二个原因是有时要求确保线程之间的动作以指定的次序发生，如一个线程需要等待由另外一个线程所引起的事件。

为了在多线程程序中解决同步问题，Windows 提供了四种主要的同步对象，每种对象相对于线程有两种状态——信号状态（signal state）和非信号状态（nonsignal state）。当相关联的同步对象处于信号状态时，线程可以执行（访问共享资源），反之必须等待。这四种同步对象是：

（1）事件对象（Event）。事件对象作为标志在线程间传递信号。一个或多个线程可等待一个事件对象，当指定的事件发生时，事件对象通知等待线程可以开始执行。它有两种类型：自动重置（auto-reset）事件和手动重置（manual-reset）事件。

（2）临界区（Critical Section）。临界区对象通过提供一个进程内所有线程必须共享的对象来控制线程。只有拥有那个对象的线程可以访问保护资源。在另一个线程可以访问该资源之前，前一个线程必须释放临界区对象，以便新的线程可以索取对象的访问权。

（3）互斥量（Mutex Semaphore）。互斥量的工作方式非常类似于临界区，只是互斥量不仅保护一个进程内为多个线程使用的共享资源，而且还可以保护系统中两个或多个进程之间的的共享资源。

（4）信号量（Semaphore）。信号量可以允许一个或有限个线程访问共享资源。它是通过计数器来实现的，初始化时赋予计数器以可用资源数，当将信号量提供给一个线程时，计数器的值减 1，当一个线程释放它时，计数器值加 1。当计数器值小于等于 0 时，相应线程必须等待。信号量是 Windows98 同步系统的核心。从本质上讲，互斥量是信号量的一种特殊形式。

Windows/NT 还提供了另外一种 Windows95 没有的同步对象：可等待定时器（Waitable Timer）。它可以封锁线程的执行，直到到达某一具体时间。这可以用于后台任务。

同步问题是多线程编程中最复杂的问题，后面的 linux 系统编程中，还会有更深入的介绍。

2.2.8 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：

1、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。

2、 MSDN,有关线程同步 interlocked 相关的知识。

3、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。

4、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。

5、C++全局和静态变量初始化顺序的研究(CSDN)。

6、四人帮的经典之作：设计模式

7、windows 核心编程(Jeffrey Richter)

2.2 简单工厂模式

2.2.1 什么是简单工厂模式

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。

2.2.2 模式中包含的角色及其职责

1.工厂（Creator）角色

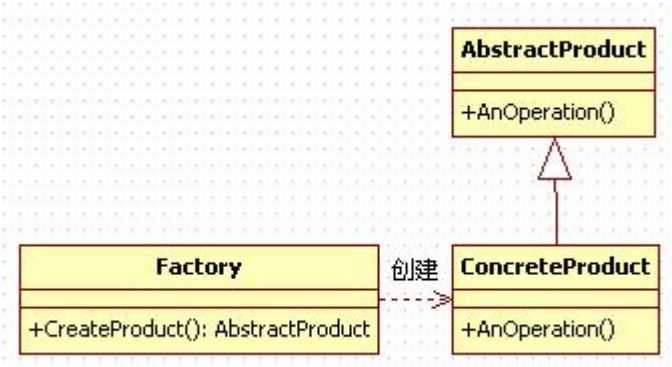
简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类可以被外界直接调用，创建所需的产品对象。

2.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。

3.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

3 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。

2.2.3 案例

```
include "iostream"
using namespace std;

//思想： 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。
//元素分析：
//抽象产品类：水果类
//具体的水果了：香蕉类、苹果类、梨子
//优点 适用于不同情况创建不同的类时
//缺点 客户端必须要知道基类和工厂类，耦合性差 增加一个产品，需要修改工厂类

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

protected:
private:
};


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

    }
protected:
private:
};


class Pear : public Fruit
{
```

```
public:
    virtual void getFruit()
    {
        cout<<"梨子"<<endl;

    }
protected:
private:
};

class Factory
{
public:
    static Fruit* Create(char *name)
    {
        Fruit *tmp = NULL;
        if (strcmp(name, "pear") == 0)
        {
            tmp = new Pear();
        }
        else if (strcmp(name, "banana") == 0)
        {
            tmp = new Banana();
        }
        else
        {
            return NULL;
        }
        return tmp;
    }
protected:
private:
};

void main41()
{
    Fruit *pear =   Factory::Create("pear");
    if (pear == NULL)
    {
        cout<<"创建 pear 失败\n";
    }
    pear->getFruit();

    Fruit *banana =   Factory::Create("banana");
```

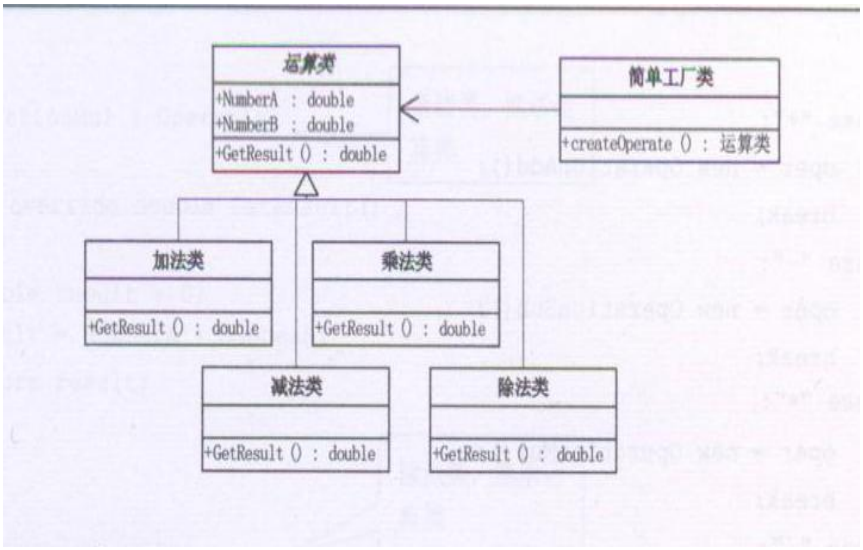
```
    banana->getFruit();

    system("pause");
}
```

2.2.4 练习

主要用于创建对象。新添加类时，不会影响以前的系统代码。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。

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



（工厂类与基类为关联关系）

```
#include "iostream"
using namespace std;
//需求：//模拟四则运算；
        //用操作符工厂类生产操作符（加减乘除），进行结果运算

//运算符抽象类 COperation
//加减乘除具体的类   (注意含有 2 个操作数)
//工厂类 CCalculatorFactory
//核心思想 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

class COperation
{
public:
    int first;
    int second;
```

```
public:
    virtual double GetResult() = 0;
private:
};

class AddOperation : public COperation
{
public:
    double GetResult()
    {
        return first + second;
    }
private:
};

class SubOperation : public COperation
{
public:
    double GetResult()
    {
        return first - second;
    }
private:
};

class CCalculatorFactory
{
public:
    static COperation*CreateOperation(char cOperator)
    {
        COperation * tmp = NULL;
        switch(cOperator)
        {
        case '+':
            tmp = new AddOperation();
            break;
        case '-':
            tmp = new SubOperation();
            break;
        default:
            tmp = NULL;
        }
        return tmp;
```

```
    }
};

void main()
{
    COperation *op1 = CCalculatorFactory::CreateOperation('+');
    op1->first = 10;
    op1->second = 20;
    cout<<op1->GetResult()<<endl;

    COperation *op2 = CCalculatorFactory::CreateOperation('-');
    op2->first = 10;
    op2->second = 20;
    cout<<op2->GetResult()<<endl;

    cout<<"hello...\n";
    system("pause");
}
```

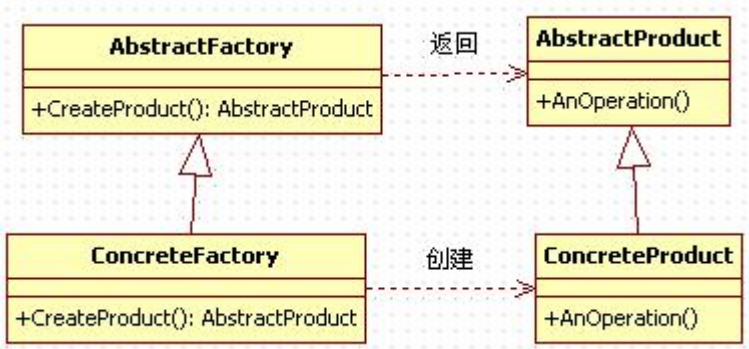
2.3 工厂模式

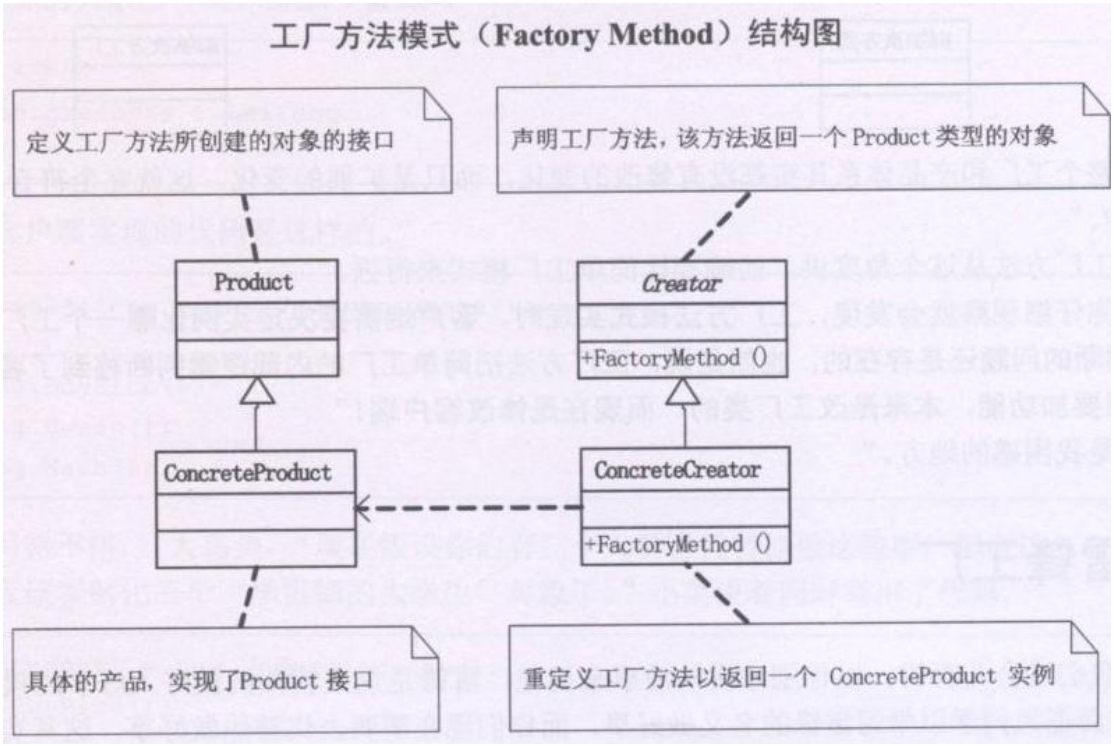
2.3.1 概念

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。
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。

2.3.2 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





2.3.3 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。

```
#include "iostream"
using namespace std;
```

```
class Fruit
{
public:
    virtual void sayname()
    {
        cout<<"fruit\n";
    }
};

class FruitFactory
{
public:
    virtual Fruit* getFruit()
    {
        return new Fruit();
    }
};


//香蕉
class Banana : public Fruit
{
public:
    virtual void sayname()
    {
        cout<<"Banana \n"<<endl;
    }
};

//香蕉工厂
class BananaFactory : public   FruitFactory
{
public:
    virtual Fruit* getFruit()
    {
        return new Banana;
    }
};



//苹果
class Apple : public Fruit
{
```

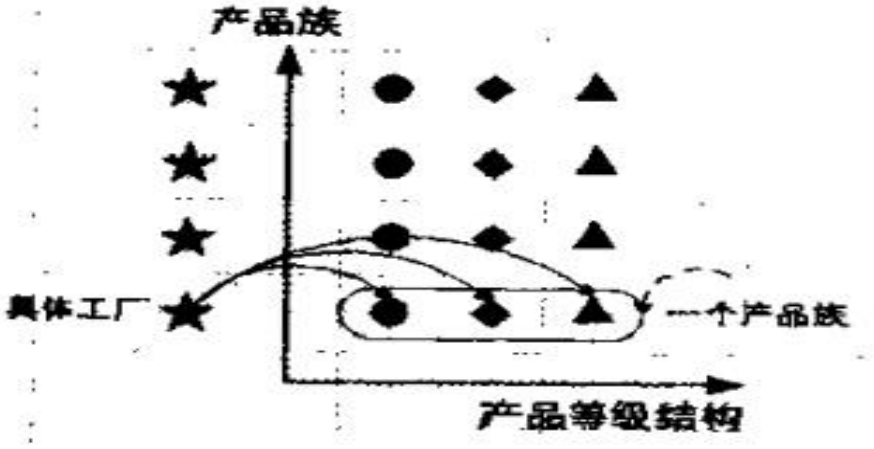
```
public:
	virtual void sayname()
	{
		cout<<"Apple \n"<<endl;
	}
};

//苹果工厂
class AppleFactory : public   FruitFactory
{
public:
	virtual Fruit* getFruit()
	{
		return new Apple;
	}
};

void main()
{
	FruitFactory * ff   = NULL;
	Fruit *fruit = NULL;

	//1
	ff = new BananaFactory();
	fruit = ff->getFruit();
	fruit->sayname();

	delete fruit;
	delete ff;

	//2 苹果
	ff = new AppleFactory();
	fruit = ff->getFruit();
	fruit->sayname();

	delete fruit;
	delete ff;

	cout<<"hello....\n";
	system("pause");
}
```

2.4 抽象工厂

2.4.1 概念

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。

2.4.2 产品族和产品等级结构



备注1：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。===》抽象工厂存在原因

解释: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 (产品族)
北方香蕉/苹果/梨子

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）

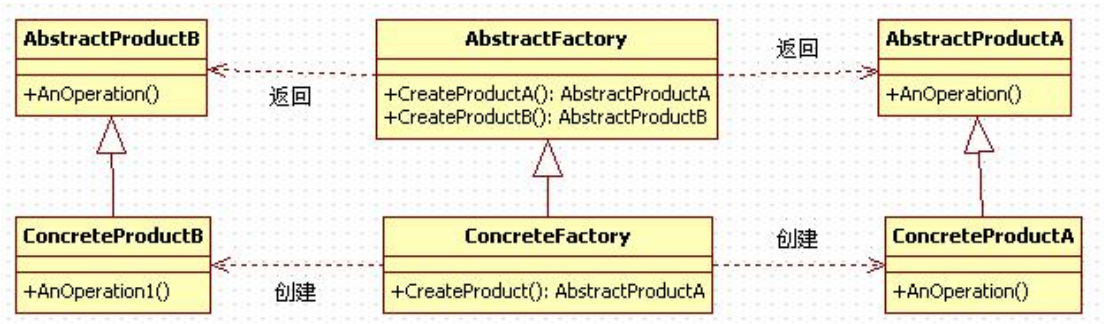
2.4.3 模式中包含的角色及其职责

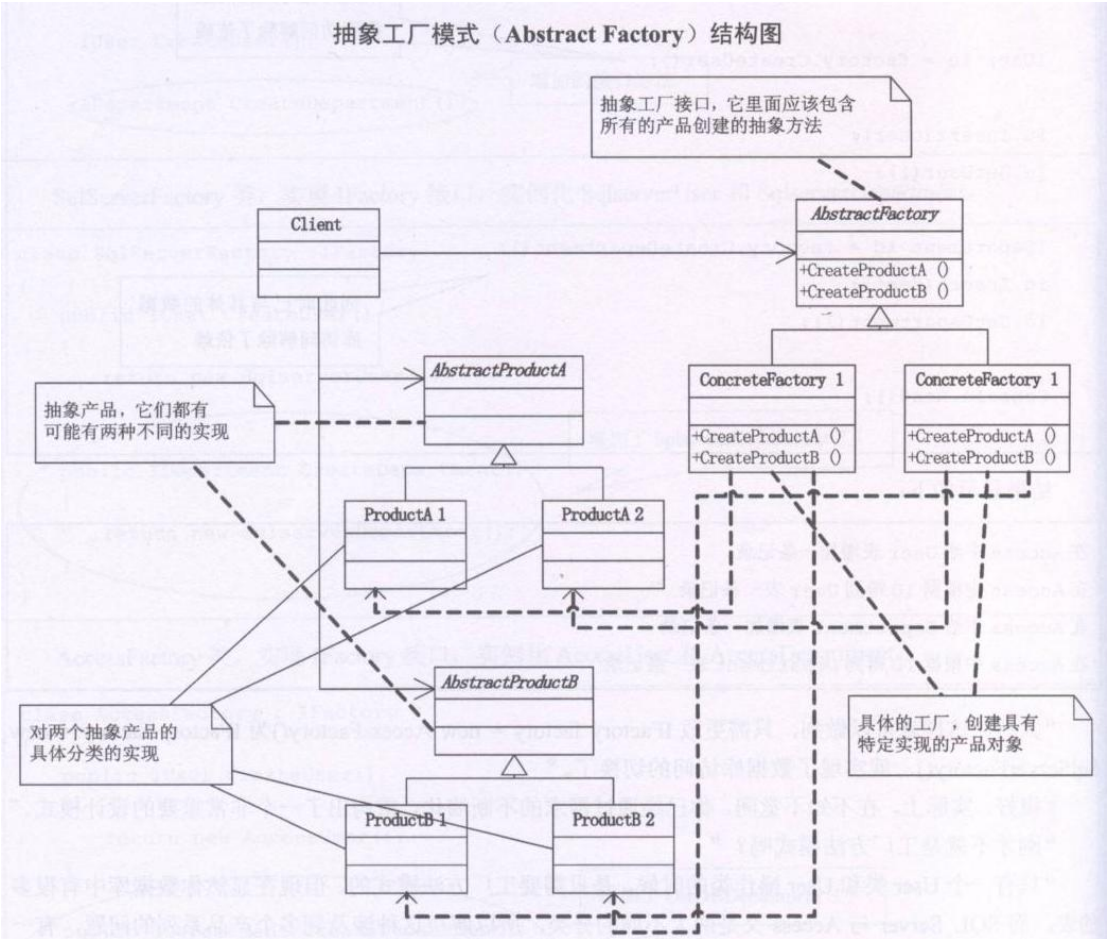
1. 抽象工厂（Creator）角色
   抽象工厂模式的核心，包含对多个产品结构的声明，任何工厂类都必须实现这个接口。

2. 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。

3. 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。

4. 具体产品（Concrete Product）角色
   抽象模式所创建的具体实例对象





2.4.4 案例

```
class Fruit
{
public:
	virtual void sayname()
	{
		cout<<"fruit\n";
	}
};

class FruitFactory
{
public:
	virtual Fruit* getApple()
	{
		return new Fruit();
	}
	virtual Fruit* getBanana()
	{
		return new Fruit();
	}
};

//南方香蕉
class SouthBanana : public Fruit
{
public:
	virtual void sayname()
	{
		cout<<"South Banana \n"<<endl;
	}
};


//南方苹果
class SouthApple : public Fruit
{
public:
	virtual void sayname()
	{
		cout<<"South Apple \n"<<endl;
```

```
    }
};


//北方香蕉
class NorthBanana : public Fruit
{
public:
    virtual void sayname()
    {
        cout<<"North Banana \n"<<endl;
    }
};


//北方苹果
class NorthApple : public Fruit
{
public:
    virtual void sayname()
    {
        cout<<"North Apple \n"<<endl;
    }
};

class SourthFruitFactory : public FruitFactory
{
public:
    virtual Fruit* getApple()
    {
        return new SouthApple();
    }
    virtual Fruit* getBanana()
    {
        return new SouthBanana();
    }
};

class NorthFruitFactory : public FruitFactory
{
public:
    virtual Fruit* getApple()
    {
        return new NorthApple();
```

```
    }
    virtual Fruit* getBanana()
    {
        return new NorthBanana();
    }
};

void main()
{
    FruitFactory * ff   = NULL;
    Fruit *fruit = NULL;

    ff = new SourthFruitFactory();
    fruit = ff->getApple();
    fruit->sayname();
    fruit = ff->getBanana();
    fruit->sayname();

    delete fruit;
    delete ff;

    ff = new NorthFruitFactory();
    fruit = ff->getApple();
    fruit->sayname();
    fruit = ff->getBanana();
    fruit->sayname();

    delete fruit;
    delete ff;

    cout<<"hello....\n";
    system("pause");
}
```

2.5 建造者模式

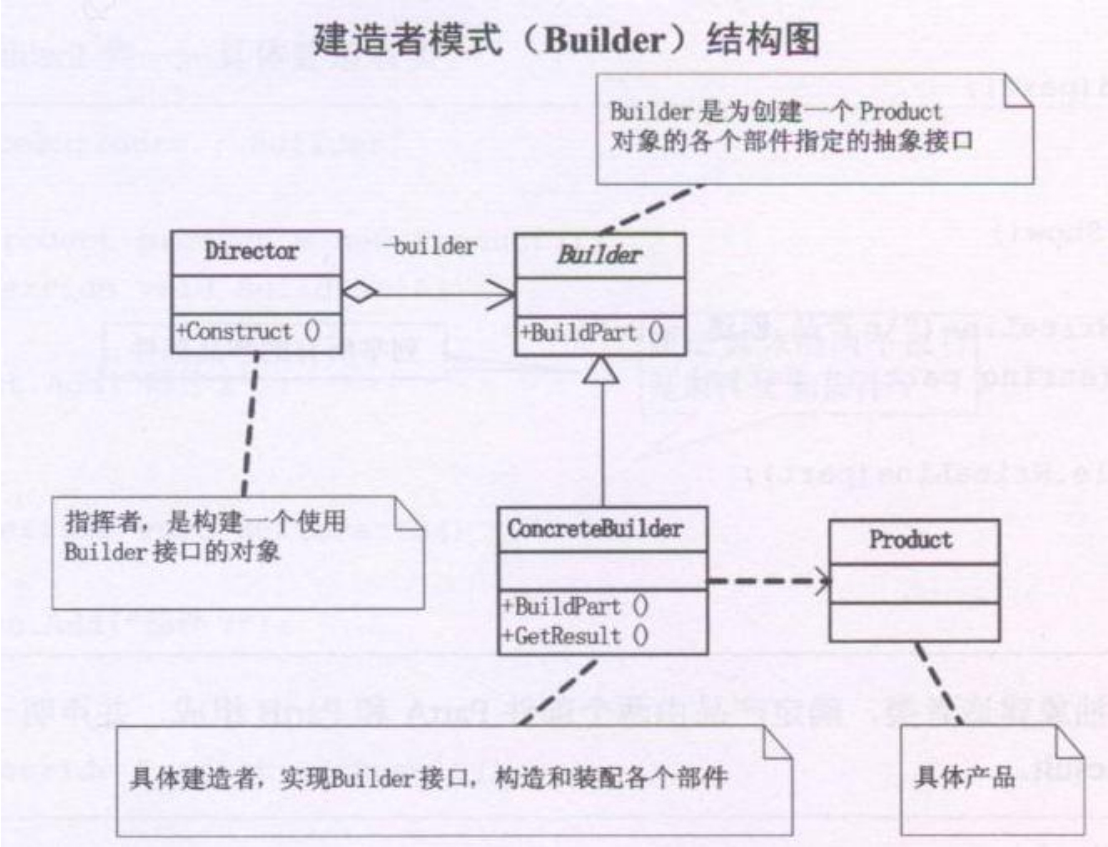
概念

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

角色和职责



1) Builder：为创建产品各个部分，统一抽象接口。
2) ConcreteBuilder：具体的创建产品的各个部分，部分 A， 部分 B，部分 C。
3) Director：构造一个使用 Builder 接口的对象。
4) 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。

创建者模式和工厂模式的区别

Factory 模式中：

1、有一个抽象的工厂。
2、实现一个具体的工厂---汽车工厂。
3、工厂生产汽车 A，得到汽车产品 A。

4、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:

1、引擎工厂生产引擎产品，得到汽车部件 A。

2、轮胎工厂生产轮子产品，得到汽车部件 B。

3、底盘工厂生产车身产品，得到汽车部件 C。

4、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。

5、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。

案例

关键字：建公寓工程队 FlatBuild 别墅工程队 VillaBuild 设计者Director

```
#include <iostream>
using namespace std;
#include "string"
class House
{
public:
    void setFloor(string floor)
    {
        this->m_floor = floor;
    }
    void setWall(string wall)
    {
        this->m_wall = wall;
    }
    void setDoor(string door)
    {
        this->m_door = door;
```

```
    }

    //
    string getFloor()
    {
        return m_floor;
    }
    string setWall()
    {
        return   m_wall;
    }
    string setDoor()
    {
        return m_door;
    }

protected:
private:
    string   m_floor;
    string   m_wall;
    string   m_door;
};

class Builder
{
public:
    virtual void makeFloor() = 0;
    virtual void makeWall() =   0;
    virtual void makeDoor() = 0;
    virtual House *GetHouse() = 0;
};

//公寓
class FlatBuild : public Builder
{
public:
    FlatBuild()
    {
        pHouse = new House;
    }
    virtual void makeFloor()
    {
        pHouse->setFloor("flat Door");
    }
```

```
    virtual void makeWall()
    {
        pHouse->setWall("flat Wall");
    }
    virtual void makeDoor()
    {
        pHouse->setDoor("flat Door");
    }
    virtual House *GetHouse()
    {
        return pHouse;
    }

private:
    House *pHouse;
};

//别墅
class VillaBuild : public Builder
{
public:
    VillaBuild()
    {
        pHouse = new House;
    }
    virtual void makeFloor()
    {
        pHouse->setFloor("villa floor");
    }
    virtual void makeWall()
    {
        pHouse->setWall("villa Wall");
    }
    virtual void makeDoor()
    {
        pHouse->setDoor("villa Door");
    }
    virtual House *GetHouse()
    {
        return pHouse;
    }
private:
    House *pHouse;
};
```

```
class Director
{
public:
	void Construct(Builder *builder)
	{
		builder->makeFloor();
		builder->makeWall();
		builder->makeDoor();
	}
protected:
private:
};


void main()
{
	//客户直接造房子
	House *pHose = new House;
	pHose->setDoor("wbm 门");
	pHose->setFloor("wbmFloor");
	pHose->setWall("wbmWall");
	delete pHose;


	/* //工程队直接造房子
	Builder *builder = new FlatBuild;
	builder->makeFloor();
	builder->makeWall();
	builder->makeDoor();
	*/

	//指挥者（设计师）指挥 工程队 和 建房子
	Director *director = new Director;

	//建公寓
	Builder *builder = new FlatBuild;
	director->Construct(builder); //设计师 指挥 工程队干活
	House *house = builder->GetHouse();
	cout << house->getFloor() << endl;
	delete house;
	delete builder;

	//建别墅
```

```
    builder = new VillaBuild;
    director->Construct(builder); //设计师 指挥 工程队干活
    house = builder->GetHouse();
    cout << house->getFloor() << endl;
    delete house;
    delete builder;

    delete director;

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

2.6 原型模式 prototype

概念

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。

1）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。

2）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。

3）根据对象克隆深度层次的不同，有浅度克隆与深度克隆。

角色和职责

Prototype 模式典型的结构图为：

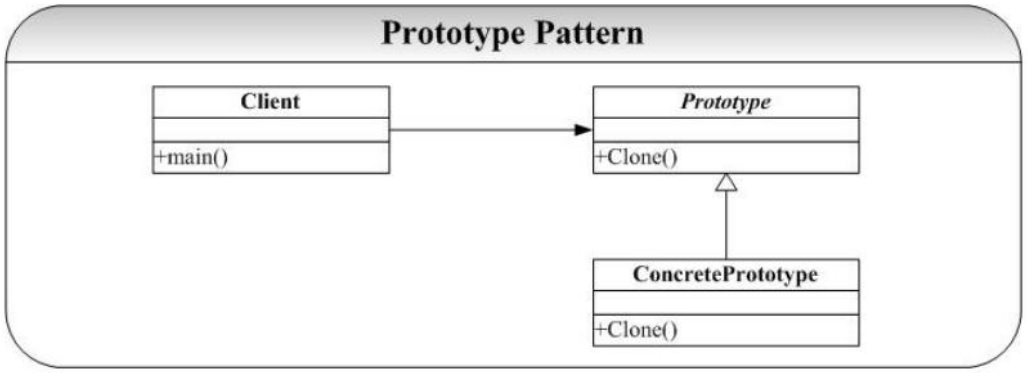


图 2-1：Prototype Pattern 结构图

Prototype 模式提供了一个通过已存在对象进行新对象创建的接口（Clone），Clone（）实现和具体的实现语言相关，在 C++中我们将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。

案例

```
class Person
{
public:
    virtual Person *Clone() = 0;
    virtual void printT() = 0;

};

class JavaProgrammer : public Person
{
public:
    JavaProgrammer()
    {
        this->m_name = "";
        this->m_age = 0;
        m_resume = NULL;
    }
    JavaProgrammer(string name, int age)
    {
        this->m_name = name;
        this->m_age = age;
        m_resume = NULL;
    }

    ~JavaProgrammer()
    {
        if (m_resume!= NULL)
        {
            free(m_resume);
            m_resume = NULL;
        }
    }
    virtual Person *Clone()
    {
        JavaProgrammer *p = new JavaProgrammer;
```

```
        *p = *this;
        return p;
    }

    void setResume(char *resume)
    {
        m_resume = new char[strlen(resume) + 1];
        strcpy(m_resume, resume);
    }

    virtual void printT()
    {
        cout << "m_name:" << m_name << "\t" << "m_age:" << m_age <<
endl;
        if (m_resume != NULL)
        {
            cout << m_resume << endl;
        }
    }
protected:
private:
    string   m_name;
    int      m_age;
    char     *m_resume;
};

void main()
{
    JavaProgrammer javaperson1("张三", 30);
    javaperson1.setResume("我是 java 程序员");
    Person *p2 = javaperson1.Clone();  //对象具有自我复制功能 注意深拷贝和浅拷
贝问题
    p2->printT();

    delete p2;

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

3 结构型模式

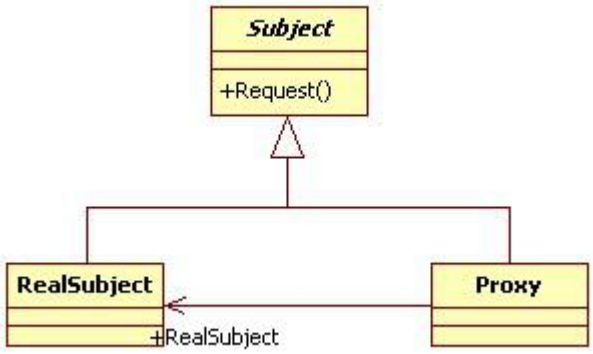
3.1 代理模式

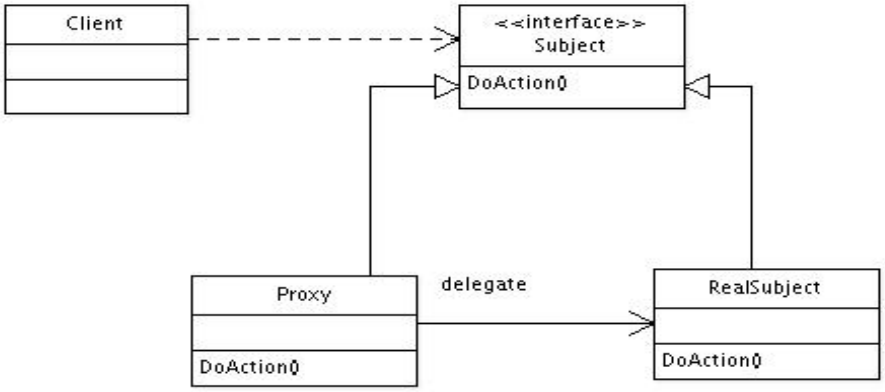
概念

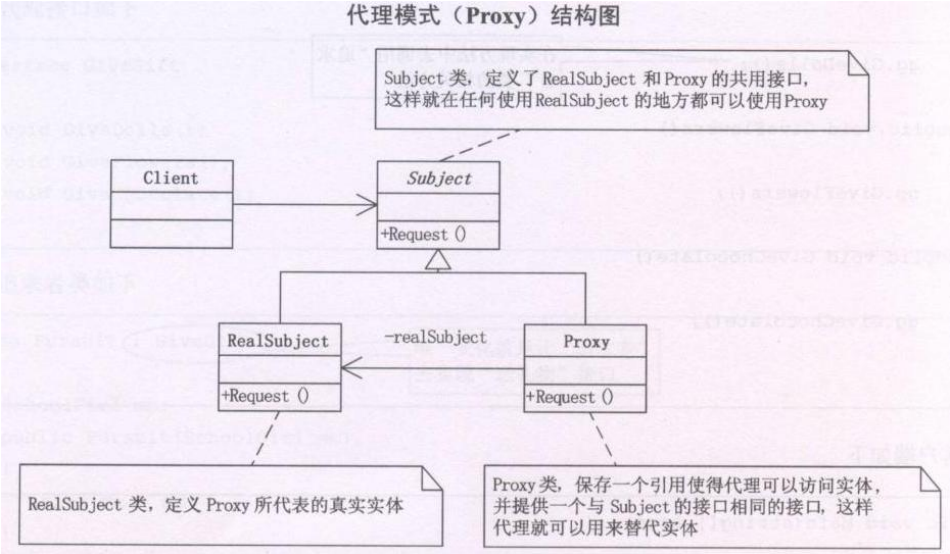
Proxy 模式又叫做代理模式，是构造型的设计模式之一，它可以为其他对象提供一种代理（Proxy）以控制对这个对象的访问。

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。

类图角色和职责







subject（抽象主题角色）：

　　真实主题与代理主题的共同接口。

RealSubject（真实主题角色）：

　　定义了代理角色所代表的真实对象。

Proxy（代理主题角色）：

　　含有对真实主题角色的引用，代理角色通常在将客户端调用传递给真是主题对象之前或者之后执行某些操作，而不是单纯返回真实的对象。

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

理论模板

理论模型

提示：a 中包含 b 类；a、b 类实现协议类 protocol

```
#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;
}
```

案例

案例 2 cocos2d-x 中应用程序代理类

```
出版社被代理对象，要卖书
淘宝、当当网（网上书店），代理对象
客户端通过当当网进行买书。
```

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

/*
subject（抽象主题角色）：
    真实主题与代理主题的共同接口。
    RealSubject（真实主题角色）：
    定义了代理角色所代表的真实对象。
```

```
    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。

    提示：a 中包含 b 类；a、b 类实现协议类 protocol

*/

class   Subject
{
public:
    virtual void SaleBook() = 0;
protected:
private:
};


class   RealSubject : public Subject
{
public:
    virtual void SaleBook()
    {
        cout << "实体店买书....\n";
    }
protected:
private:
};

class   ProxyTaoBao : public Subject
{
public:
    virtual void SaleBook()
    {
        //
        Double11();
        RealSubject rs;
        rs.SaleBook();
        Double11();
    }
    void Double11()
    {
        cout << "Double11 打折 半价" << endl;
    }
protected:
```

```
private:
};


class   ProxyTaoBao2 : public Subject
{
public:
    void SetRealSubject(RealSubject *rs)
    {
        m_s = rs;
    }
    virtual void SaleBook()
    {
        Double11();
        m_s->SaleBook();
    }
    void Double11()
    {
        cout << "Double11 打折 半价" << endl;
    }
protected:
private:
    RealSubject *m_s;
};

void main61()
{
    ProxyTaoBao *ptb = new ProxyTaoBao;
    ptb->SaleBook();
    delete ptb;
    return ;
}
```

```
#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;
protected:
private:
};
```

```
//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;
    }
protected:
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);
    }

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

练习

1 定义真实玩家（李逍遥）
2 定义代理玩家
3 代理玩家，代替李逍遥，进行升级打怪

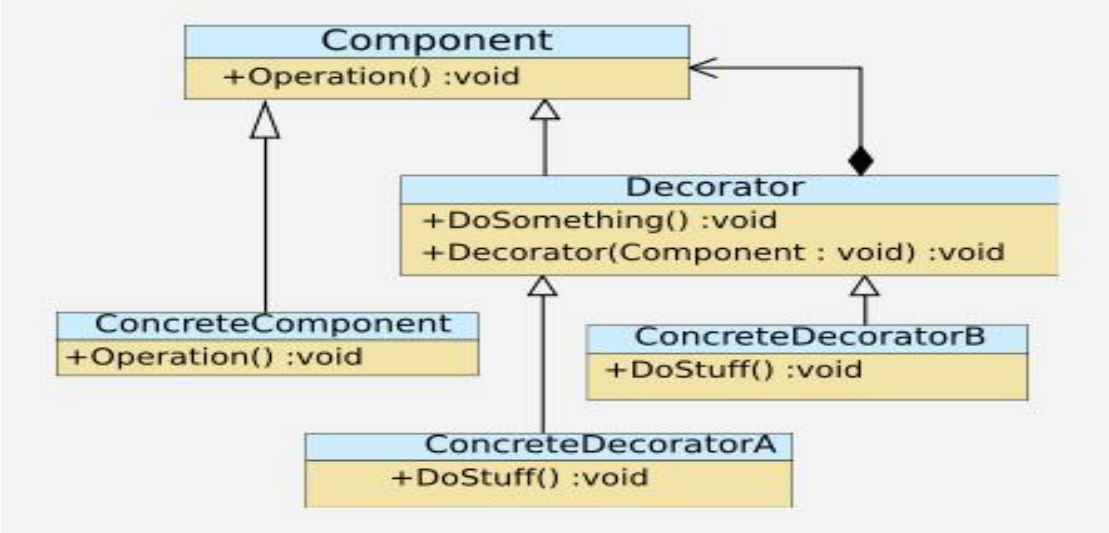
3.2 装饰模式

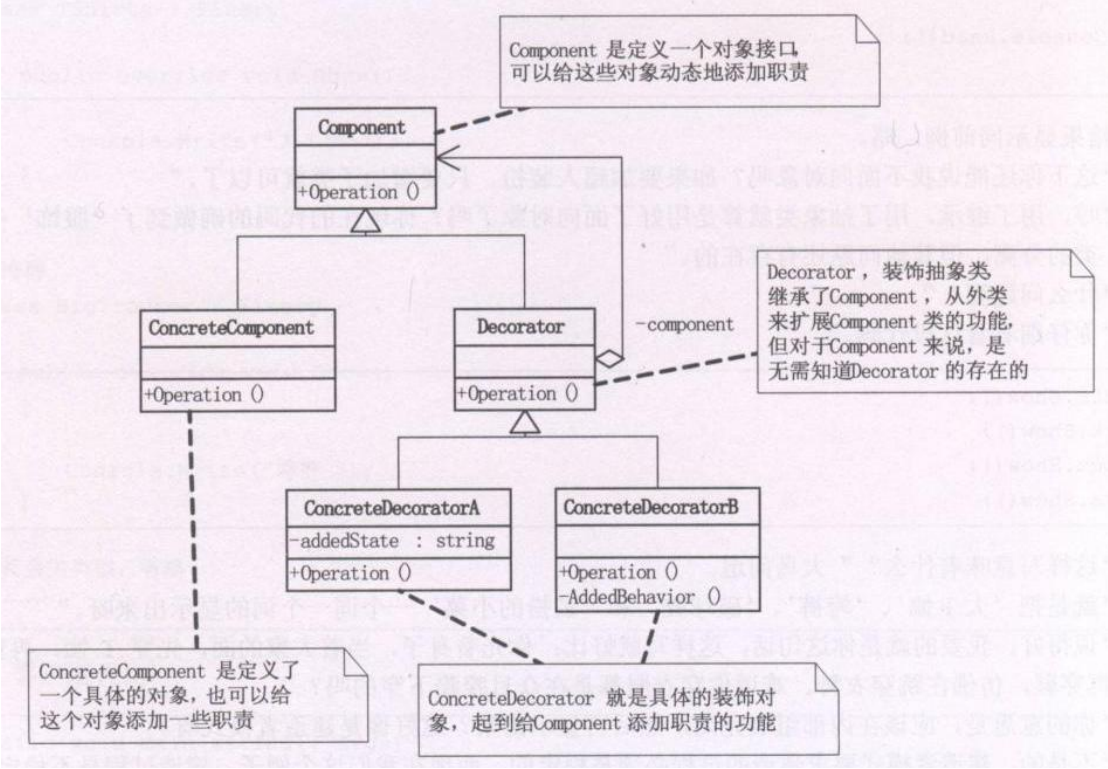
概念

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。

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。

案例

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

class Car
{
public:
    virtual void show() = 0;
protected:
private:
};

class RunCar : public Car
{
public:
    void run()
    {
        cout << "可以跑" << endl;
    }
```

```
    virtual void show()
    {
        run();
    }
protected:
private:
};

class SwimCarDirector : public Car
{
public:
    SwimCarDirector(Car *p)
    {
        m_p = p;
    }

    void swim()
    {
        cout << "可以游" << endl;
    }

    virtual void show()
    {
        m_p->show();
        swim();
    }
private:
    Car *m_p;
};

class FlyCarDirector : public Car
{
public:
    FlyCarDirector(Car *p)
    {
        m_p = p;
    }

    void fly()
    {
        cout << "可以飞" << endl;
    }
    virtual void show()
    {
```

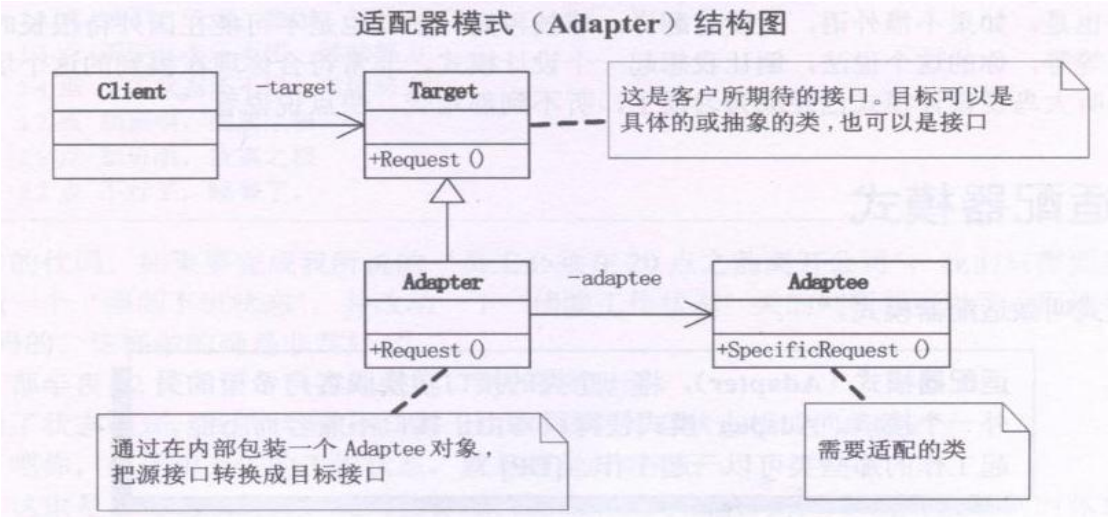
```
        m_p->show();
        fly();
    }
private:
    Car *m_p;
};

void main()
{
    Car *runcar = NULL;
    runcar = new RunCar;
    runcar->show();

    cout <<"车开始装饰 swim"<<endl;
    SwimCarDirector *swimCar = new SwimCarDirector(runcar);
    swimCar->show();

    cout <<"车开始装饰 fly"<<endl;
    FlyCarDirector *flyCar = new FlyCarDirector(swimCar);
    flyCar->show();

    delete flyCar;
    delete swimCar;
    delete runcar;

    return ;
}
```

3.3 适配器模式 adapter

概念

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。

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。

案例

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

class Current18v
{
public:
    void use18vCurrent()
    {
        cout << "使用 18v 的交流电" << endl;
    }
protected:
private:
};


class Current220v
{
public:
    void use220vCurrent()
    {
        cout << "使用 220v 的交流电" << endl;
```

```
    }
protected:
private:
};


class Adapter : public Current18v
{
public:
    Adapter(Current220v *p220v)
    {
        m_p220v = p220v;
    }
    void use18vCurrent()
    {
        cout << "adapter 中使用电流" << endl;
        m_p220v->use220vCurrent();
    }
protected:
private:
    Current220v *m_p220v;
};

void main()
{
    Current220v *p220v = new Current220v;
    Adapter *padapter = new Adapter(p220v);
    padapter->use18vCurrent();

    delete p220v;
    delete padapter;
    system("pause");
    return ;
}
```

练习

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。

思路： Player， 抽象的球员 （Attack、 Defense）

class TransLater: public Player 适配器

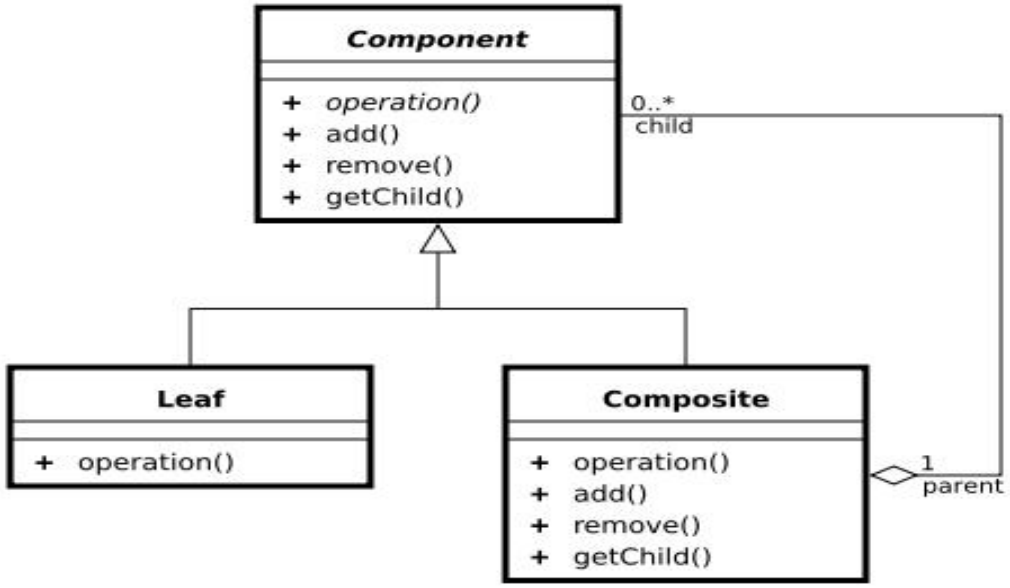
class Center : public Player 被适配的对象

3.4 组合模式

概念

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。

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”

案例

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

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

class File : public IFile
{
public:
	File(string name)
	{
		m_list = NULL;
		m_name = "";
		m_name = name;
	}
	~File()
	{
		if (m_list != NULL)
		{
			delete m_list;
		}
	}
	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:
    list<IFile *> *    m_list;
    string         m_name;

};

class Folder : public IFile
{
public:
    Folder(string name)
    {
        m_name = name;
        m_list = new list<IFile *>;
    }
    ~Folder()
    {
        if (m_list == NULL)
        {
            delete m_list;
        }
    }
    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:
    list<IFile *> *    m_list;
    string             m_name;

};

void showTree(IFile *ifile, int level)
{
    list<IFile *> *l = NULL;
    int i = 0;
    for (i=0; i<level; i++)
    {
        printf("\t");
    }
    ifile->display();

    l = ifile->getChild();
    if (l != NULL)
    {
        for (list<IFile *>::iterator it=l->begin(); it!=l->end(); it++)
        {
            if ( (*it)->getChild() == NULL)
            {
                for (i=0; i<=level; i++) //注意 <=
                {
                    printf("\t");
                }
                (*it)->display();
            }
            else
            {
                showTree((*it), level + 1);
            }

        }
    }
}
```

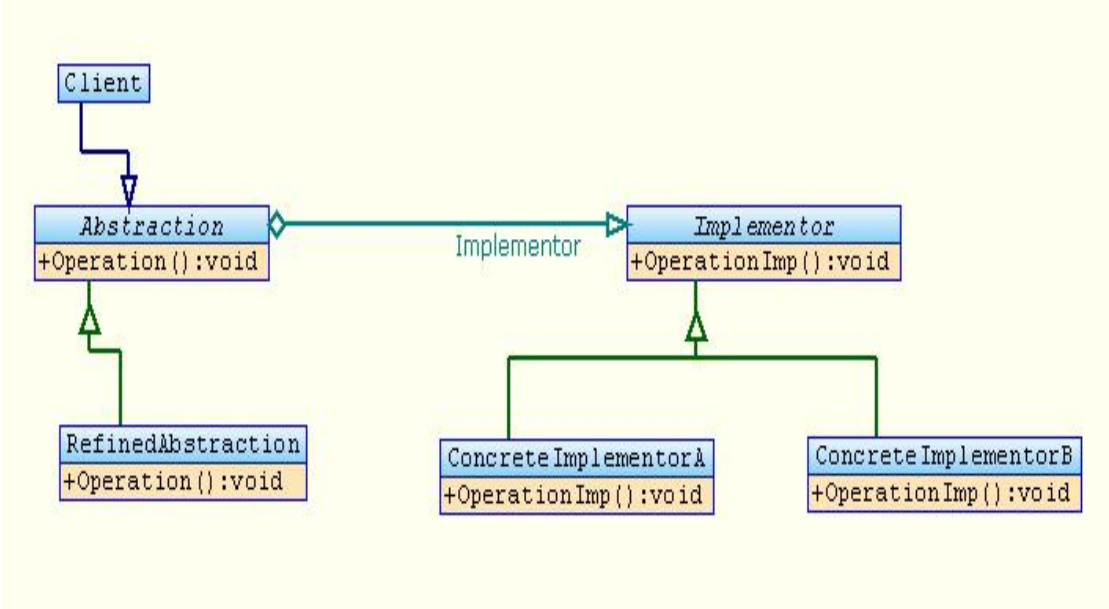
```
void main()
{
	Folder *root = new Folder("C:");

	Folder *dir1 = new Folder("111dir");
	File *txt1 = new File("aaa.txt");

	Folder *dir12 = new Folder("222dir");
	//dir12->display();
	File *txt12 = new File("222.txt");
	//txt12->display();



	root->display();
	root->add(dir1);
	root->add(txt1);

	dir1->add(dir12);
	dir1->add(txt12);

	/*
	list<IFile *> *l = dir1->getChild();
	for (list<IFile *>::iterator it=l->begin(); it!=l->end(); it++)
	{
		(*it)->display();
	}
	*/
	//开发一个递归函数 现在根结点下的所有子结点
	cout << "测试递归函数" << endl;

	showTree(root, 0);

	delete txt12;
	delete dir12;
	delete dir1;
	delete txt1;
	delete root;
	cout<<"hello..."<<endl;
	system("pause");
	return ;
}
```

3.5 桥接模式 bridge

概念

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。

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

图形 填 颜色 不同形状的图形，填充上 不同的颜色

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；
将“图形 填 颜色”这个抽象 和 实现 进行分离，两个名字，就设计两个类

案例

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

class MyCar1
{
public:
    virtual void installEngine() = 0;
};

class BMW5 : public MyCar1
{
public:
    virtual void installEngine()
    {
        cout << "BMW5 3500CC" << endl;
    }
};

class BMW6 : public MyCar1
{
public:
    virtual void installEngine()
    {
        cout << "BMW6 4000CC" << endl;
    }
};


class Jeep11 : public MyCar1
{
public:
    virtual void installEngine()
    {
        cout << "Jeep11 1100CC" << endl;
    }
```

```
};


class Jeep12 : public MyCar1
{
public:
    virtual void installEngine()
    {
        cout << "Jeep12 1200CC" << endl;
    }
};

//不同的车型，不同型号，安装不同类型的发动机，会引起子类的泛滥
//问题引出
void main1601()
{
    Jeep12 *j12 = new Jeep12;
    j12->installEngine();
    delete j12;
    return ;
}

class MyCar2
{

public:
    virtual void installEngine3500() = 0;
    virtual void installEngine4000() = 0;
    virtual void installEngine1100() = 0;
    virtual void installEngine1200() = 0;
};

class BMW : public MyCar2
{
public:
    virtual void installEngine3500()
    {
        cout << "3500CC" << endl;
    }
    virtual void installEngine4000()
    {
        cout << "4000CC" << endl;
    }
    virtual void installEngine1100()
```

```
    {
        cout << "1100CC" << endl;
    }
    virtual void installEngine1200()
    {
        cout << "1200CC" << endl;
    }
};

//这样的设计 不符合开闭原则
void main1602()
{
    BMW *bmw5 = new BMW;
    bmw5->installEngine3500();
}

//需要把“安装发动机”这个事，做很好的分解；把抽象 和 行为实现 分开
//发动机是一个名次，专门抽象成一个类；类中含有一个成员函数，安装发动机

class Engine
{
public:
    virtual void installEngine() = 0;
};

class Engine4000 : public Engine
{
public:
    virtual void installEngine()
    {
        cout << "安装发动机 Engine4000" << endl;
    }
};

class Engine3500 : public Engine
{
public:
    virtual void installEngine()
    {
        cout << "安装发动机 Engine 3500" << endl;
    }
};

class Car
```

```
{
public:
    Car(Engine *pengine)
    {
        m_engine = pengine;
    }
    virtual void installEngine() = 0;

protected:
    Engine *m_engine;
};

class BMW7 :public Car
{
public:
    BMW7(Engine *p) : Car(p)
    {

    }

    //注意车的安装  和 发动机的安装 不同之处
    virtual void installEngine()
    {
        cout << "BMW7 " ;
        m_engine->installEngine();
    }
protected:
private:
};

void main163()
{
    Engine4000 *e4000 = new Engine4000;
    BMW7 *bmw7 = new BMW7(e4000);
    bmw7->installEngine();

    delete bmw7;
    delete e4000;
}
void main()
{
    //main1601();
    //main1602();
    main163();
```

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

练习

在上述案例场景之中，让 jeep11 车，安装上 1100CC 发动机。
方法：添加 Jeep11 类，添加 Engine1100CC 发动机类，在 main 函数中编写测试案例，体会设计模式的开闭原则。

抽象一个图形 AbstractShape，含有操作 draw（），可以画圆形、正方形
抽象一个颜色类 color，含有 getColor()：string
让 AbstractShape 持有一个 color 的引用。形成桥接模式。
创建类 class Circle : public AbstractShape
Class Square : publice AbstractShape
创建类
Class Red : public color
Class Green : public color
测试： 用红色 画 正方形

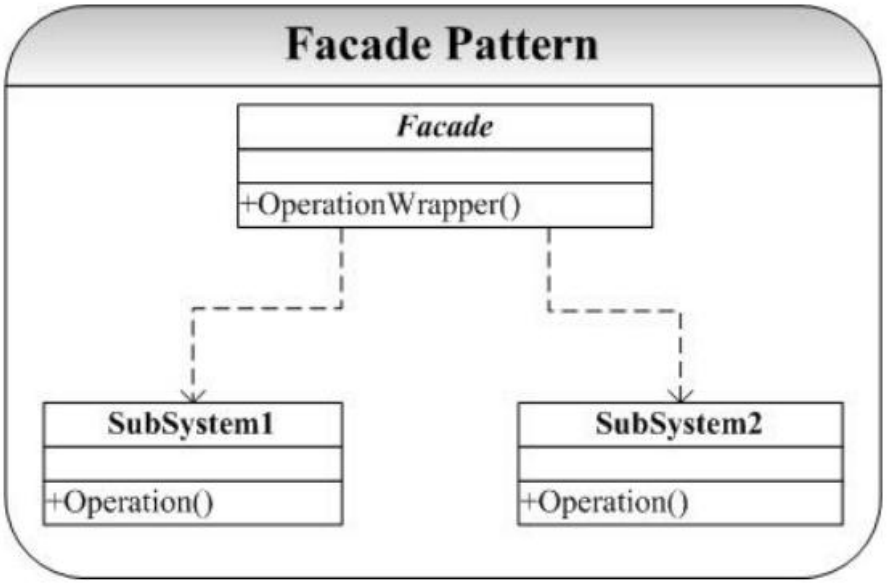
```
Void main()
{
    Color color = new Red();
    AbstractShape *shap = new Square(color);
    shap.draw();
}
```

3.6 外观模式 façade

概念

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。

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。

案例

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

class SystemA
{
public:
    void doThing()
    {
        cout << "systemA do...." << endl;
    }
};

class SystemB
```

```
{
public:
    void doThing()
    {
        cout << "systemA do...." << endl;
    }
};

class SystemC
{
public:
    void doThing()
    {
        cout << "systemA do...." << endl;
    }
};

class Facade
{
public:
    Facade()
    {
        a = new SystemA;
        b = new SystemB;
        c = new SystemC;
    }
    ~Facade()
    {
        delete a;
        delete b;
        delete c;
    }

    void doThing()
    {
        a->doThing();
        b->doThing();
        c->doThing();
    }

protected:
private:
    SystemA *a;
    SystemB *b;
```

```
    SystemC *c;
};


void main1414()
{
    /*
    SystemA *a = new SystemA;
    SystemB *b = new SystemB;
    SystemC *c = new SystemC;

    a->doThing();
    b->doThing();
    c->doThing();

    delete a;
    delete b;
    delete c;
    */

    Facade *f = new Facade;
    f->doThing();
    delete f;
    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

练习案例

基金（股票、外汇、期货、国债）

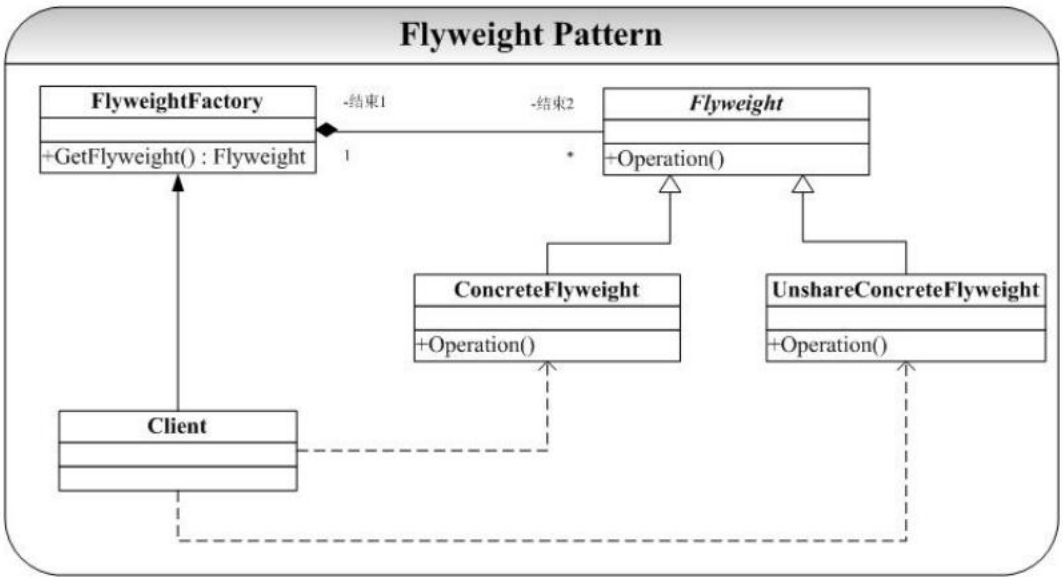
3.7 享元模式 flyweight

概念

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）。

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。

案例

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

class Person
{
public:
    Person(string name, int age, int sex)
    {
        this->name = name;
        this->age = age;
        this->sex = sex;
    }
    string getName()
    {
        return name;
    }
    int getAge()
    {
        return age;
    }
    int getSex()
    {
        return sex;
    }
protected:
    string   name;
    int      age;
    int      sex; //1 男 2 女
};

class Teacher : public Person
{
public:
    Teacher(string id, string name, int age, int sex) : Person(name, age, sex)
    {
        this->id = id;
    }

    string getId()
    {
        return id;
    }
```

```
    void printT()
    {
        cout << "id:" <<id << "\t" << "name:" <<name << "\t" << "age:"
<<age << "\t" << "sex:" <<sex << "\t" << endl;
    }
private:
    string id;
};

class TeacherFactory
{
public:
    TeacherFactory()
    {
        m_tpool.empty();
    }
    ~TeacherFactory()
    {
        //内存管理 永远是 c++程序员的痛
        while (!m_tpool.empty()) //在工厂中创建老师结点，在工厂中销毁老师结点
        {
            Teacher *tmp = NULL;
            map<string, Teacher *>::iterator it = m_tpool.begin();
            tmp = it->second;
            m_tpool.erase(it);
            delete tmp;
        }
    }
    //通过 Teacher 的 pool，来存放老师结点，在 TeacherFactory 中创建老师、销毁老
师
    Teacher *getTeacher(string tid)
    {
        string   name;
        int      age;
        int      sex;

        Teacher *tmp = NULL;
        map<string, Teacher*>::iterator it =   m_tpool.find(tid);
        if (it == m_tpool.end())
        {
            cout << "id 为: " << tid << " 的老师不存在,系统为你创建该老师，请输
入以下信息" <<endl;
            cout << "请输入老师姓名 : ";
            cin >> name;
```

```
            cout << "请输入老师年龄：";
            cin >> age;
            cout << "请输入老师性别 1 男 2 女：";
            cin >> sex;
            tmp = new Teacher(tid, name, age, sex);
            m_tpool.insert(pair<string, Teacher*>(tid, tmp));
        }
        else
        {
            tmp = (it->second);
        }
        return tmp;
    }

private:
    map<string, Teacher *> m_tpool;
};


void main()
{
    /*
    Teacher *t1 = new Teacher("001", "小李", 30, 1);
    Teacher *t2 = new Teacher("002", "小张", 30, 1);
    Teacher *t3 = new Teacher("001", "小李", 30, 1);
    Teacher *t4 = new Teacher("004", "小吴", 30, 1);
    //
    cout << "t1 t3 的 工号一样，但是也不是同一个人 " << endl;
    delete t1;
    delete t2;
    delete t3;
    delete t4;
    */
    TeacherFactory *teacherFactory = new TeacherFactory;
    Teacher *t1 = teacherFactory->getTeacher("001");
    t1->printT();

    Teacher *t2 = teacherFactory->getTeacher("001");
    t2->printT();
    delete teacherFactory;
    system("pause");
    return ;
}
```

4 行为型模式

4.1 模板模式 template

概念

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。

应用场景

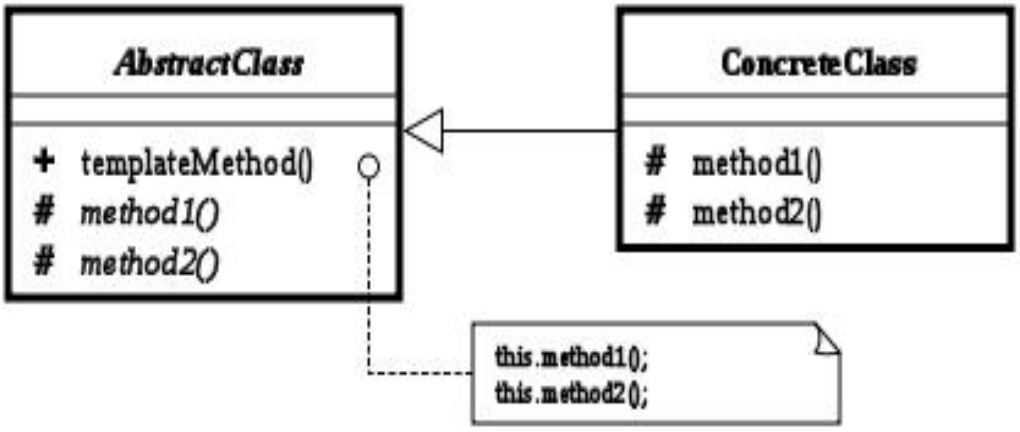
Template Method 模式一般应用在具有以下条件的应用中：

- 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

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

案例

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

class MakeCar
{
public:
    virtual void makeHead() = 0;
    virtual void makeBody() = 0;
    virtual void makeTail() = 0;

public:    //把一组行为 变成 一个模板
    void make()
    {
        makeHead();
        makeBody();
        makeTail();
```

```
    }

protected:
private:
};

class MakeBus : public MakeCar
{
public:
    virtual void makeHead()
    {
        cout << "bus 组装 车头" << endl;
    }
    virtual void makeBody()
    {
        cout << "bus 组装 车身" << endl;
    }
    virtual void makeTail()
    {
        cout << "bus 组装 车尾" << endl;
    }
protected:
private:
};

class MakeJeep : public MakeCar
{
public:
    virtual void makeHead()
    {
        cout << "Jeep 组装 车头" << endl;
    }
    virtual void makeBody()
    {
        cout << "Jeep 组装 车身" << endl;
    }
    virtual void makeTail()
    {
        cout << "Jeep 组装 车尾" << endl;
    }
protected:
private:
};
```

```
void main()
{
	MakeCar *bus = new MakeBus;

	//bus->makeHead();
	//bus->makeBody();
	//bus->makeTail();
	bus->make();

	MakeCar *jeep = new MakeJeep;
	//jeep->makeHead();
	//jeep->makeBody();
	//jeep->makeTail();
	jeep->make();

	delete bus;
	delete jeep;

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

4.2 命令模式 command

概念

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。

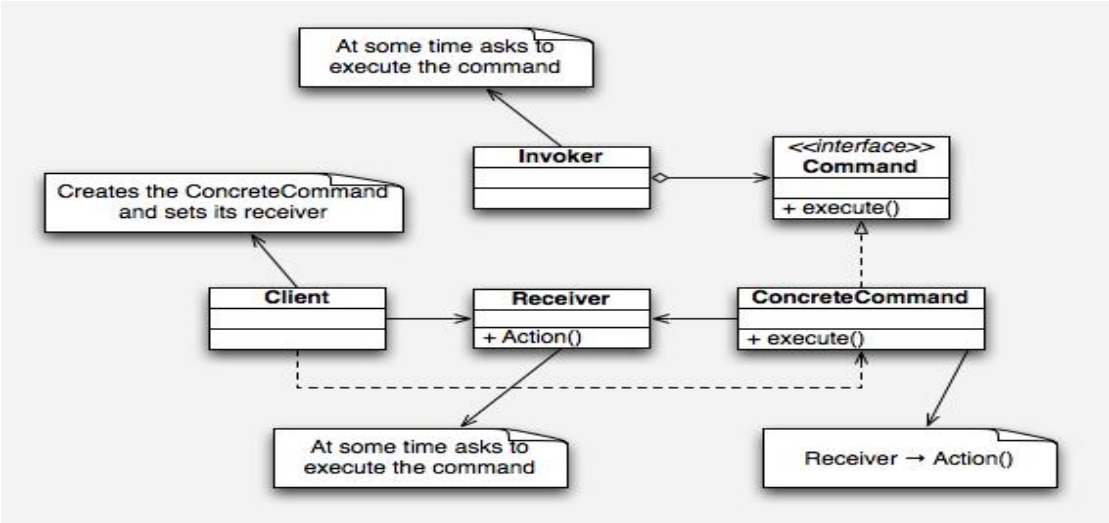
在面向对象的程序设计中，一个对象调用另一个对象，一般情况下的调用过程是：创建目标对象实例；设置调用参数；调用目标对象的方法。

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。

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。

案例

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

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();
    }
protected:
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();
    }
protected:
private:
    Vendor *m_v;
};

class Waiter
{
public:
    Command *getCommand()
    {
        return m_command;
    }
    void setCommand(Command *c)
    {
        m_command = c;
    }
    void sail()
    {
        m_command->sail();
    }
protected:
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();
        }
    }
protected:
private:
    list<Command *> *m_list;
};

//小商贩 直接 卖 水果
void main25_01()
{
    Vendor *v = new Vendor;
    v->sailapple();
    v->sailbanana();

    delete v;
    return ;
}

//小商贩 通过命令 卖 水果
void main25_02()
{
    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 main25_03()
{
    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 main25_04()
{
    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()
{
    //main25_01();
    //main25_02();
    //main25_03();
    main25_04();
    system("pause");
}
```

练习

医院护士（医导）、医生；
护士把简历给医生；医生看病

4.3 责任链模式

概念

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。

角色和职责

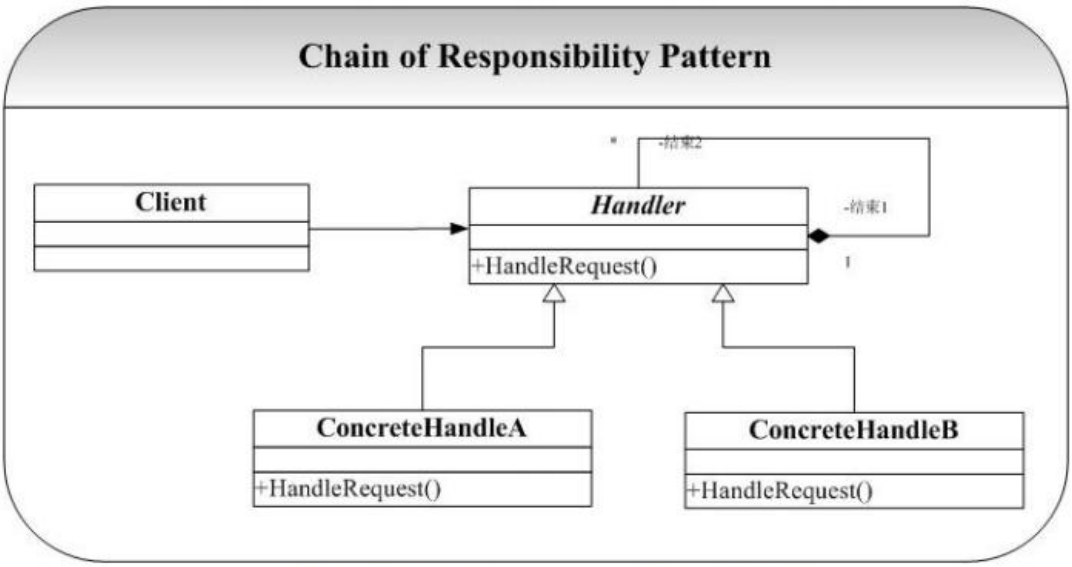


图 2-1：Chain of Responsibility Pattern 结构图

Handler

处理类的抽象父类。

concreteHandler

具体的处理类。

责任链优缺点

优点：

1。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。

2。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。

3。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；

案例

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

```
{
public:
    virtual void HandleCar() = 0;

public:
    CarHandle *setNextHandle(CarHandle *carhandle)
    {
        this->carhandle = carhandle;
        return this->carhandle;
    }
protected:
    CarHandle *carhandle;
};

class CarHandleHead : public CarHandle
{
public:
    virtual void HandleCar()
    {
        cout << "处理车头" << endl;
        if (this->carhandle != NULL)
        {
            carhandle->HandleCar();
        }
    }
};

class CarHandleBody : public CarHandle
{
public:
    virtual void HandleCar()
    {
        cout << "处理车身" << endl;
        if (this->carhandle != NULL)
        {
            carhandle->HandleCar();
        }
    }
};

class CarHandleTail : public CarHandle
{
public:
    virtual void HandleCar()
```

```
    {
        cout << "处理车尾" << endl;
        if (this->carhandle != NULL)
        {
            carhandle->HandleCar();
        }
    }
};

void main()
{
    CarHandleHead *head = new CarHandleHead;
    CarHandleBody *body = new CarHandleBody;
    CarHandleTail *tail = new CarHandleTail;

    head->setNextHandle(body);
    body->setNextHandle(tail);
    tail->setNextHandle(NULL);

    //处理
    head->HandleCar();
    delete head;
    delete body;
    delete tail;
    system("pause");
    return ;
}
```

4.4 策略模式

概念

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 。

角色和职责

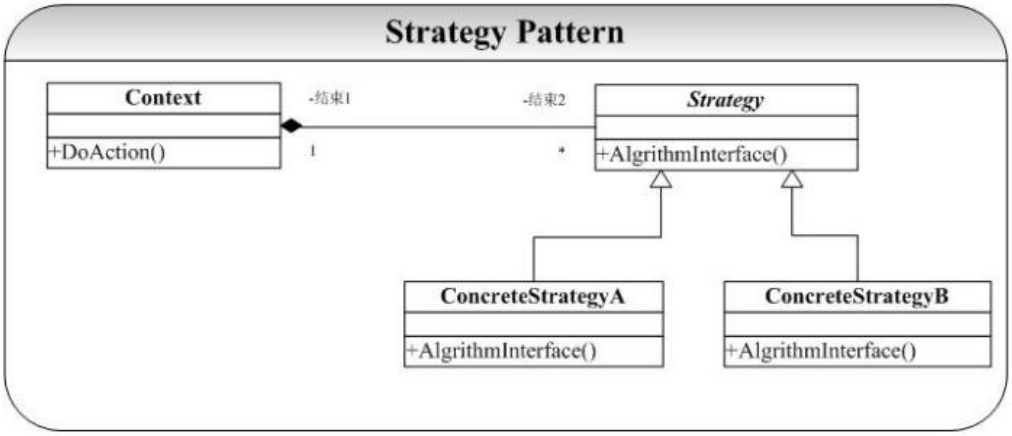


图 2-1：Strategy Pattern 结构图

这里的关键就是将算法的逻辑抽象接口（DoAction）封装到一个类中（Context），再通过委托的方式将具体的算法实现委托给具体的 Strategy 类来实现（ConcreteStrategeA 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：

1. 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。

2. 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。

3. 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：
1. 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。
2. 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。

案例

```
//Symmetric encryption
class Strategy
{
public:
    virtual void SymEncrypt() = 0;
};

class Des : public Strategy
{
public:
    virtual void SymEncrypt()
    {
        cout << "Des 加密" << endl;
    }
};

class AES : public Strategy
{
public:
    virtual void SymEncrypt()
    {
        cout << "AES 加密" << endl;
    }
};


class Context
{
public:
    Context(Strategy *strategy)
    {
        p = strategy;
    }
```

```
    void Operator()
    {
        p->SymEncrypt();
    }
private:
    Strategy *p;
};


//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()
{
    /* 不符合开闭原则
    Strategy *strategy = NULL;
    strategy = new AES;
    strategy->SymEncrypt();
    delete strategy;

    strategy = new Des;
    strategy->SymEncrypt();
    delete strategy;
    */
    Strategy *strategy = NULL;
    Context *ctx = NULL;

    strategy = new AES;
    ctx = new Context(strategy);
    ctx->Operator();
    delete strategy;
    delete ctx;

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

练习

商场促销有策略 A（0.8 折） 策略 B（消费满 200，返券 100），用策略模式模拟场景。

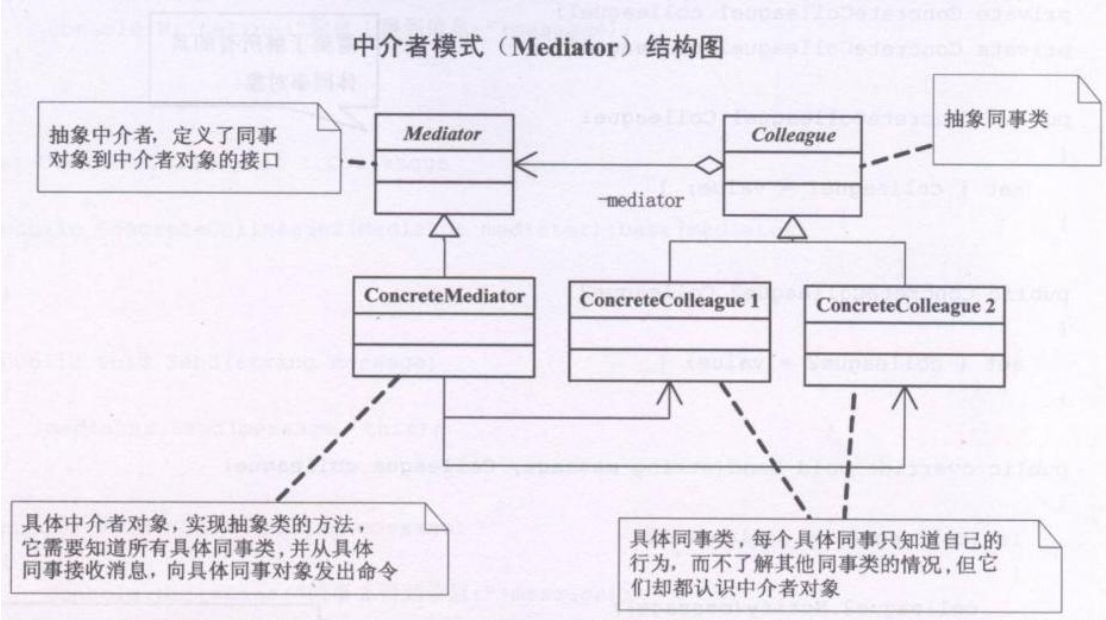
4.5 中介者模式 mediator

概念

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。

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。

模式优点

1，将系统按功能分割成更小的对象，符合类的最小设计原则
2，对关联对象的集中控制
3，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）。
4，有利于提高类的重用性

案例

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

class Person
{
public:
    Person(string name, int sex, int condit)
    {
        m_name = name;
        m_sex = sex;
        m_condition = condit;
    }
    string getName()
    {
        return m_name;
    }
    int getSex()
    {
        return m_sex;
    }
    int getCondit()
    {
        return m_condition;
    }
    virtual void getParter(Person *p) = 0;

protected:
    string    m_name;    //
```

```
    int       m_sex; //1 男   2 女
    int       m_condition; //123456789;
};

class Man : public Person
{
public:
    Man(string name, int sex, int condit):Person(name, sex, condit)
    {
        ;
    }
    virtual void getParter(Person *p)
    {
        if (this->getSex() == p->getSex())
        {
            cout << "No No No 我不是同性恋" << endl;
        }
        if (this->getCondit() == p->getCondit())
        {
            cout << this->getName() << " 和 " << p->getName() << "绝配"
<< endl;
        }
        else
        {
            cout << this->getName() << " 和 " << p->getName() << "不配"
<< endl;
        }
    }
protected:

};

class Woman : public Person
{
public:
    Woman(string name, int sex, int condit):Person(name, sex, condit)
    {
        ;
    }
    virtual void getParter(Person *p)
    {
        if (this->getSex() == p->getSex())
        {
            cout << "No No No 我不是同性恋" << endl;
```

```
        }
        if (this->getCondit() == p->getCondit())
        {
            cout << this->getName() << " 和 " << p->getName() << "绝配"
<< endl;
        }
        else
        {
            cout << this->getName() << " 和 " << p->getName() << "不配"
<< endl;
        }
    }
protected:

};

//以上  Woman  Man 类的太紧密 需要解耦合
void main1901()
{

    Woman *w1 = new Woman("小芳", 2, 4);
    Man *m1 = new Man("张三", 1, 3);
    Man *m2 = new Man("李四", 1, 4);

    w1->getParter(m1);
    w1->getParter(m2);

    delete w1;
    delete m1;
    delete m2;

    return ;
}

class Mediator ;
class Person2
{
public:
    Person2(string name, int sex, int condition, Mediator *m)
    {
        m_name = name;
        m_sex = sex;
        m_condition = condition;
        m_m = m;
```

```
    }
    string getName()
    {
        return m_name;
    }
    int getSex()
    {
        return m_sex;
    }
    int getCondit()
    {
        return m_condition;
    }

    Mediator *getMediator()
    {
        return m_m;
    }
public:
    virtual void getParter(Person2 *p) = 0;

protected:
    string   m_name;    //
    int      m_sex; //1 男   2 女
    int      m_condition; //123456789;
    Mediator *m_m;
};


class Mediator
{
public:
    Mediator()
    {
        pMan = NULL;
        pWoman = NULL;
    }
    void setWoman(Person2 *p)
    {
        pWoman = p;
    }
    void setMan(Person2 *p)
    {
        pMan = p;
```

```
    }

    void getPartner()
    {
        if (pMan->getSex() == pWoman->getSex())
        {
            cout << "No No No 我不是同性恋" << endl;
        }
        if (pMan->getCondit() == pWoman->getCondit())
        {
            cout << pMan->getName() << " 和 " << pWoman->getName()
<< "绝配" << endl;
        }
        else
        {
            cout << pMan->getName() << " 和 " << pWoman->getName()
<< "不配" << endl;
        }
    }

protected:
private:
    Person2*pMan;
    Person2*pWoman;
};

class Woman2 : public Person2
{
public:
    Woman2(string name, int sex, int condition, Mediator *m) : Person2(name,
sex, condition, m)
    {
        ;
    }
    virtual void getParter(Person2 *p)
    {
        this->getMediator()->setWoman(this);
        this->getMediator()->setMan(p);
        this->getMediator()->getPartner();
    }
private:
};

class Man2 : public Person2
```

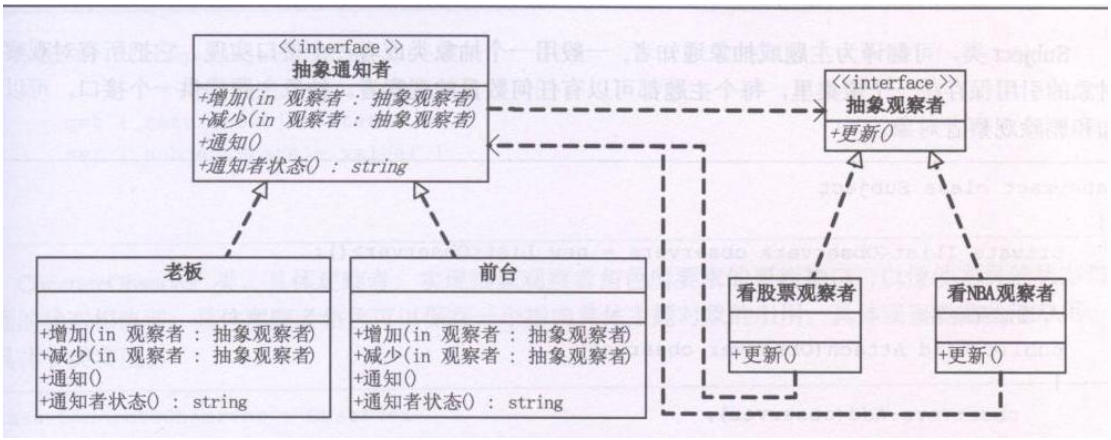
```
{
public:
    Man2(string name, int sex, int condition, Mediator *m) : Person2(name, sex, condition, m)
    {
        ;
    }
    virtual void getParter(Person2 *p)
    {
        this->getMediator()->setMan(this);
        this->getMediator()->setWoman(p);
        this->getMediator()->getPartner();
    }
private:
};

void main1902()
{
    Mediator *mediator = new Mediator;
    Woman2 *w1 = new Woman2("小芳", 2, 4, mediator);
    Man2 *m1 = new Man2("张三", 1, 3, mediator);
    Man2 *m2 = new Man2("李四", 1, 4, mediator);

    w1->getParter(m1);
    w1->getParter(m2);

    delete w1;
    delete m1;
    delete m2;
    delete mediator;
}
void main()
{
    //main1901(); //问题的引出
    main1902(); //用中介者模式 进行优化
    system("pause");
}
```

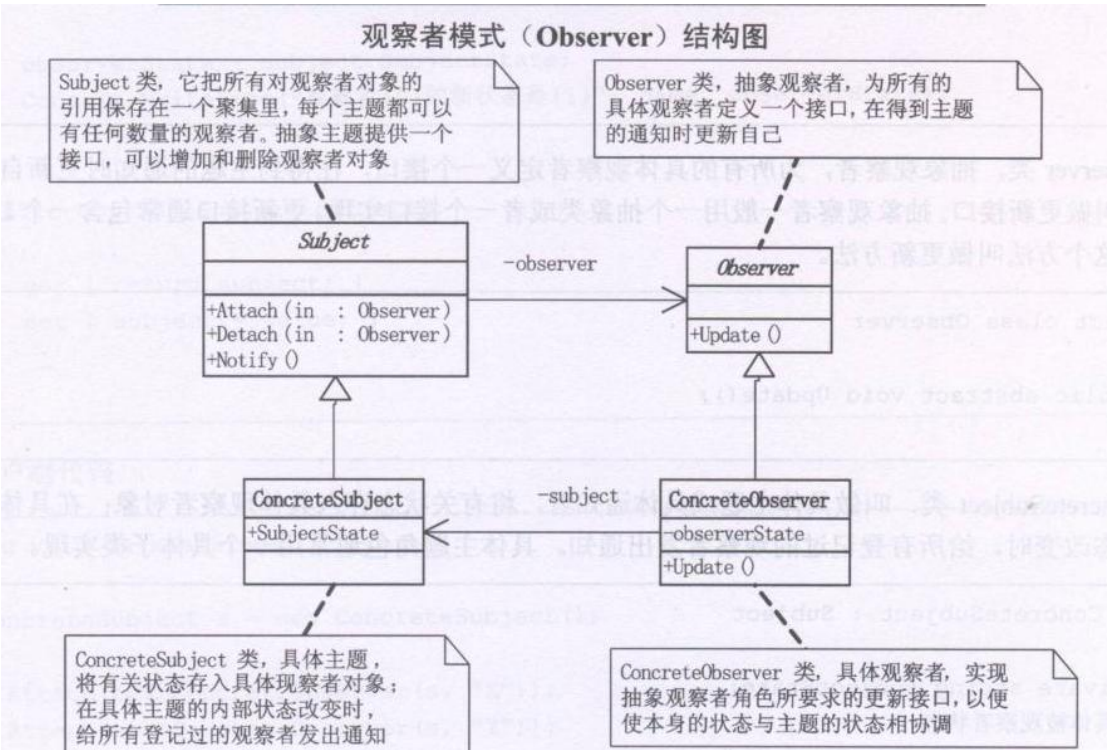
4.6 观察者模式 observer

概念

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。

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。

案例

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

class Secretary;

//玩游戏的同事类（观察者）
class PlayserObserver
{
public:
    PlayserObserver(string name, Secretary *secretary)
    {
        m_name = name;
        m_secretary = secretary;
    }
    void update(string action)
    {
        cout << "观察者收到 action:" << action << endl;
    }
private:
    string        m_name;
    Secretary   *m_secretary;
};

//秘书类（主题对象，通知者）
class Secretary
{
public:
    void addObserver(PlayserObserver *o)
    {
        v.push_back(o);
    }
    void Notify(string action)
    {
        for (vector<PlayserObserver *>::iterator it= v.begin(); it!=v.end();
it++ )
        {
            (*it)->update(action);
        }
    }
```

```
    void setAction(string action)
    {
        m_action = action;
        Notify(m_action);
    }
private:
    string m_action;
    vector<PlayserObserver *> v;
};

void main()
{
    //subject 被观察者
    Secretary *s1 = new Secretary;

    //具体的观察者 被通知对象
    PlayserObserver *po1 = new PlayserObserver("小张", s1);
    //PlayserObserver *po2 = new PlayserObserver("小李", s1);
    s1->addObserver(po1);
    //s1->addObserver(po2);
    s1->setAction("老板来了");
    s1->setAction("老板走了");
    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

练习

利用观察者模式实现发布者/订阅者(publisher/subscriber)模型。

4.7 备忘录模式 mememto

概念

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。

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。

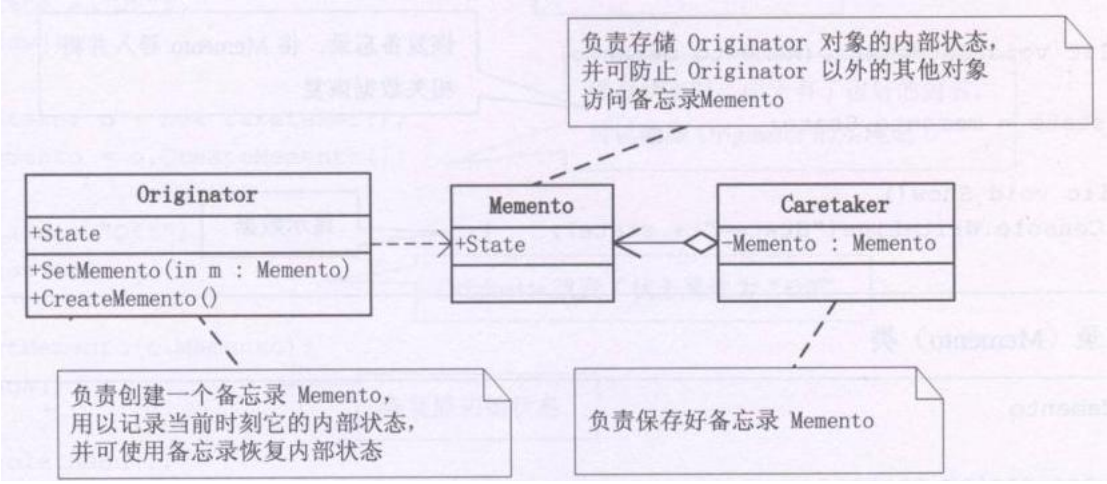
1）一个类需要保存它的对象的状态（相当于 Originator 角色）

2）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）

3）需要的时候，Caretaker 角色要求 Originator 返回一个 Memento 并加以保存

4）undo 或 rollback 操作时，通过 Caretaker 保存的 Memento 恢复 Originator 对象的状态

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。

案例

```
#include <iostream>
```

```
using namespace std;
#include "string"

class MememTo
{
public:
    MememTo(string name, int age)
    {
        this->m_name = name;
        this->m_age = age;
    }
    void setName(string name)
    {
        this->m_name = name;
    }
    string getName()
    {
        return m_name;
    }
    void setAge(int   age)
    {
        this->m_age = age;
    }
    int getAge()
    {
        return m_age;
    }
protected:
private:
    string   m_name;
    int      m_age;
};

class Person
{
public:
    Person(string name, int age)
    {
        this->m_name = name;
        this->m_age = age;
    }
    void setName(string name)
    {
        this->m_name = name;
```

```
    }
    string getName()
    {
        return m_name;
    }
    void setAge(int   age)
    {
        this->m_age = age;
    }
    int getAge()
    {
        return m_age;
    }
    void printT()
    {
        cout << "name: " << m_name << "age: " << m_age << endl;
    }

public:

    //创建备份
    MememTo *createMememTo()
    {
        return new MememTo(m_name, m_age);
    }

    //恢复备份
    void SetMememTo(MememTo *memto)
    {
        m_name = memto->getName();
        m_age = memto->getAge();
    }

protected:
private:
    string   m_name;
    int      m_age;

};

//管理者
class Caretaker
{
public:
```

```
    Caretaker(MememTo *mem)
    {
        this->m_memto = mem;
    }
    MememTo *getMememTo()
    {
        return m_memto;
    }
    void setMememTo(MememTo *mem)
    {
        this->m_memto = mem;
    }
protected:
private:
    MememTo *m_memto;
};

void main23_01()
{
    Person *p = new Person("张三", 18);
    p->printT();

    //创建备份
    Caretaker *ct = new Caretaker(p->createMememTo());

    p->setAge(28);
    p->printT();

    //恢复信息
    p->SetMememTo(ct->getMememTo());
    p->printT();

    delete p;
    delete ct->getMememTo();

    return ;
}

void main23_02()
{
    Person *p = new Person("张三", 18);
    p->printT();

    //创建备份
```

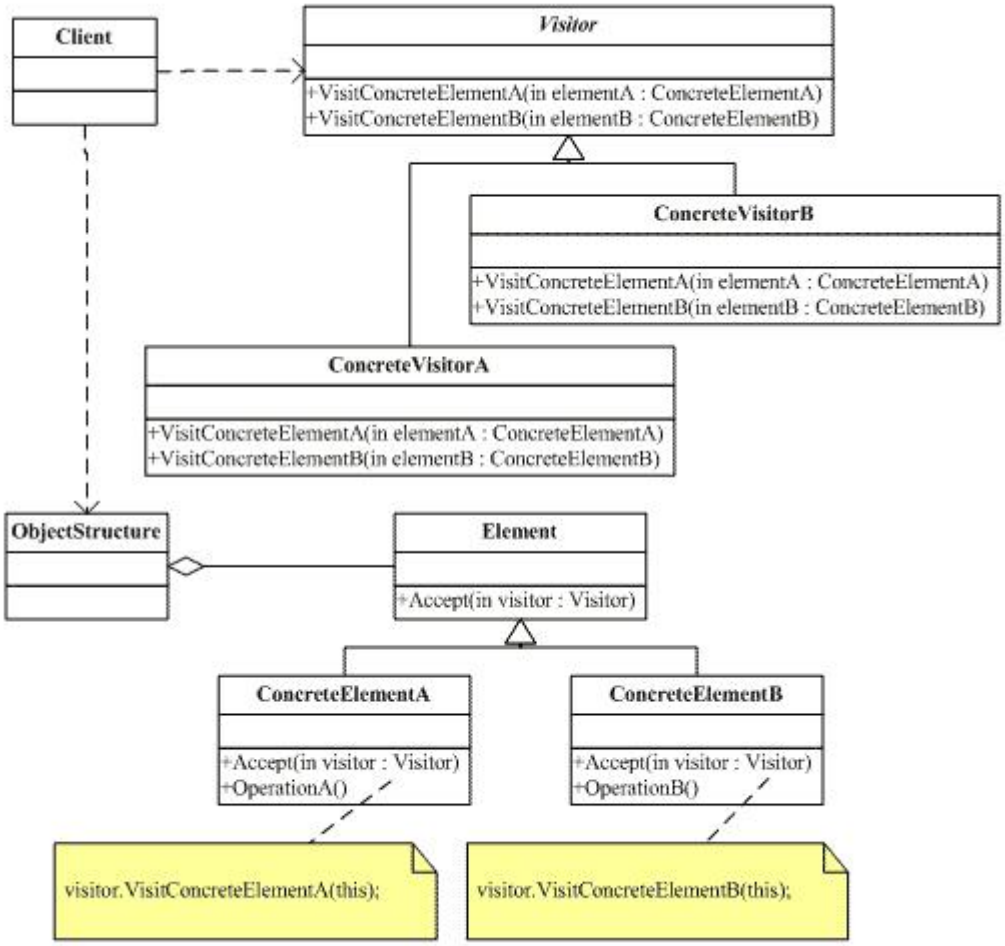
```
    MememTo * membak = p->createMememTo();
    p->setAge(28);
    p->printT();

    //恢复信息
    p->SetMememTo(membak);
    p->printT();

    delete p;
    delete membak;
}

void main()
{
    //main23_01();
    main23_02();
    system("pause");
}
```

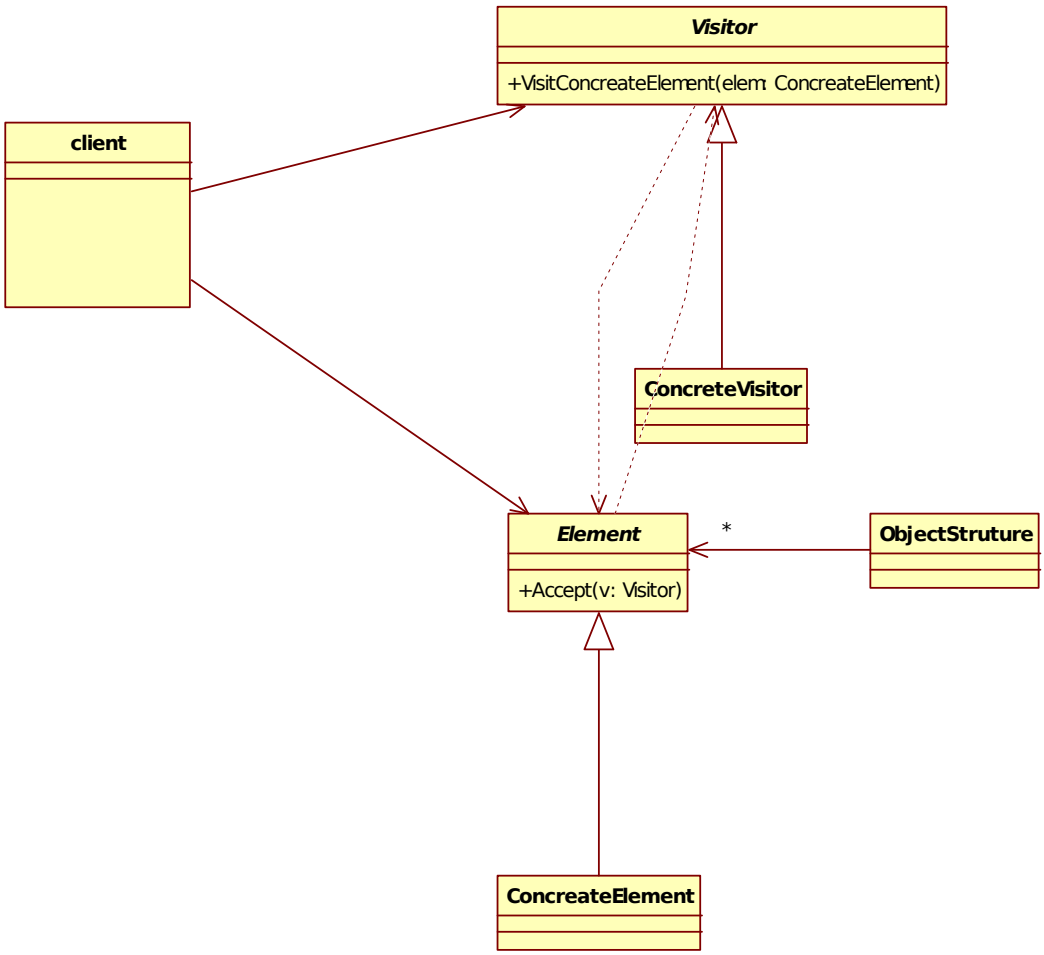
4.8 访问者模式 visitor

概念

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。

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);



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>
using namespace std;
#include "list"
#include "string"

class   ParkElement;
```

```
//不同的访问者 访问公园完成不同的动作
class Visitor
{
public:
    virtual void visit(ParkElement *park) = 0;
};

class ParkElement //每一个
{
public:
    virtual void accept(Visitor *v) = 0;
};

class ParkA : public ParkElement
{
public:
    virtual void accept(Visitor *v)
    {
        v->visit(this);
    }
};

class ParkB : public ParkElement
{
public:
    virtual void accept(Visitor *v)
    {
        v->visit(this);
    }
};

class Park   : public ParkElement
{
public:
    Park()
    {
        m_list.clear();
    }
    void setPart(ParkElement *e)
    {
        m_list.push_back(e);
    }
public:
```

```
    void accept(Visitor *v)
    {
        for ( list<ParkElement *>::iterator it=m_list.begin(); it != m_list.end();
it++)
        {
            (*it)->accept(v);
        }
    }

private:
    list<ParkElement *> m_list;
};

class VisitorA : public Visitor
{
public:
    virtual void visit(ParkElement *park)
    {
        cout << "清洁工 A 访问公园 A 部分，打扫卫生完毕" << endl;
    }
};

class VisitorB : public Visitor
{
public:
    virtual void visit(ParkElement *park)
    {
        cout << "清洁工 B 访问 公园 B 部分，打扫卫生完毕" << endl;
    }
};
class VisitorManager : public Visitor
{
public:
    virtual void visit(ParkElement *park)
    {
        cout << "管理员 检查整个公园卫生打扫情况" << endl;
    }
};

void main()
{
    VisitorA *visitorA = new VisitorA;
    VisitorB *visitorB = new VisitorB;
```

```
    ParkA *partA = new ParkA;
    ParkB *partB = new ParkB;

    //公园接受访问者 a 访问
    partA->accept(visitorA);
    partB->accept(visitorB);

    VisitorManager *visitorManager = new VisitorManager;
    Park * park = new Park;
    park->setPart(partA);
    park->setPart(partB);
    park->accept(visitorManager);

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

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

//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 main26_01()
{
    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 main26_02()
{
    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()
{
    //main26_01();
    main26_02();
    system("pause");
}
```

4.9 状态模式 state

概念

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。

- **问题**

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。

■ 模式选择

State 模式就是被用来解决上面列出的两个问题的，在 State 模式中我们将状态逻辑和动作实现进行分离。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 模式将每一个分支都封装到独立的类中。State 模式典型的结构图为：

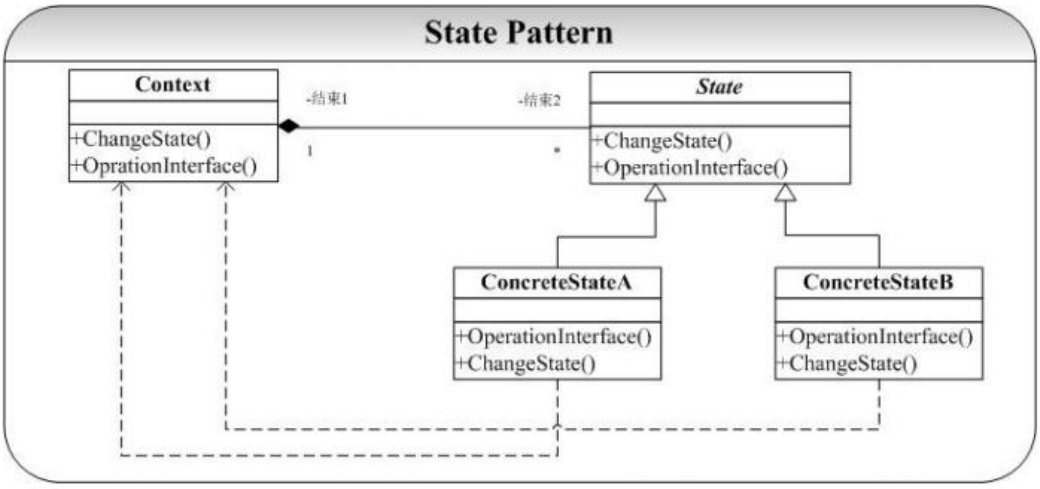


图 2-1：State Pattern 结构图

角色和职责

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。

案例

```
#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)
    {
        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
    {
        delete w->getCurrentState(); //状态 2 不满足 要转到状态 3 后者恢复到初始
化状态
        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;
    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

4.10 解释模式 interpreter

概念

一些应用提供了内建（Build-In）的脚本或者宏语言来让用户可以定义他们能够在系统中进行的操作。Interpreter 模式的目的就是使用一个解释器为用户提供一个一门定义语言的语法表示的解释器，然后通过这个解释器来解释语言中的句子。

Interpreter 模式提供了这样的一个实现语法解释器的框架，笔者曾经也正在构建一个编译系统 Visual CMCS，现在已经发布了 Visual CMCS1.0 (Beta)，请大家访问 Visual CMCS 网站获取详细信息。

角色和职责

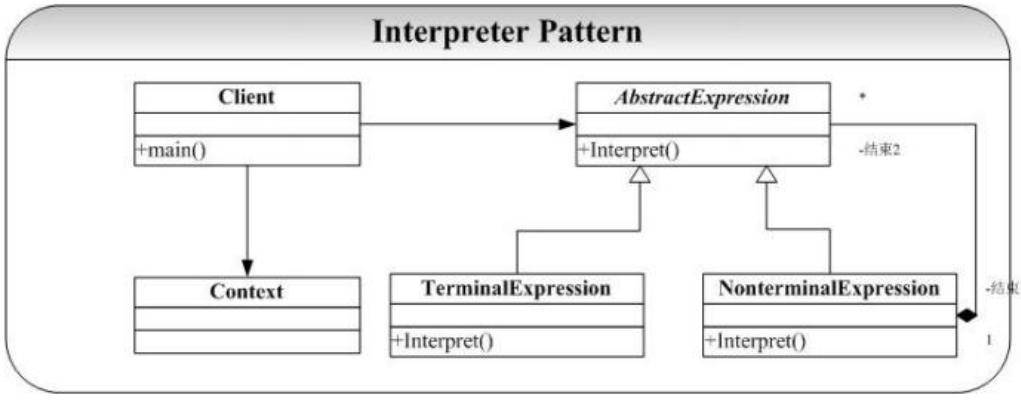
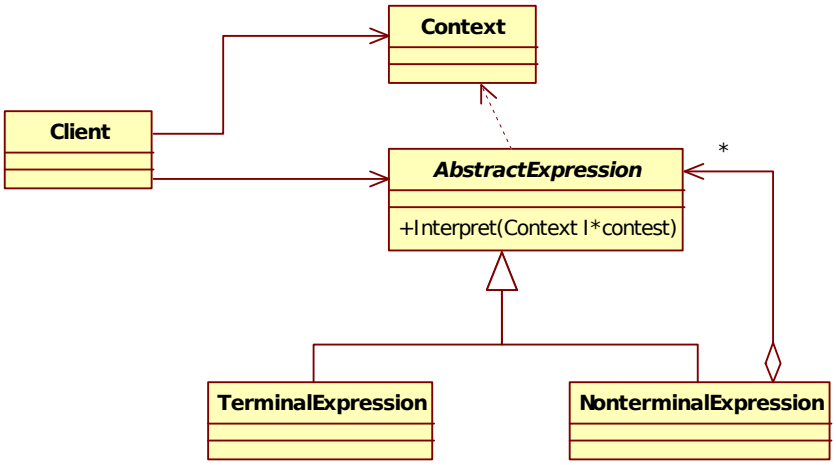


图 2-1：Interpreter Pattern 结构图

Interpreter 模式中，提供了 TerminalExpression 和 NonterminalExpression 两种表达式的解释方式，Context 类用于为解释过程提供一些附加的信息（例如全局的信息）。



Context

解释器上下文环境类。用来存储解释器的上下文环境，比如需要解释的文法等。

AbstractExpression

解释器抽象类。

ConcreteExpression

解释器具体实现类。

案例

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

class Context
{
public:
    Context(int num)
    {
        m_num = num;
    }
public:
    void setNum(int num)
    {
        m_num = num;
    }
    int getNum()
    {
        return m_num;
    }
```

```
    void setRes(int res)
    {
        m_res = res;
    }
    int getRes()
    {
        return m_res;
    }

private:
    int m_num;
    int m_res;

};

class Expression
{
public:
    virtual void interpreter(Context *context) = 0;
};

class PlusExpression : public Expression
{
public:
    virtual void interpreter(Context *context)
    {
        int num = context->getNum();
        num ++ ;
        context->setNum(num);
        context->setRes(num);
    }
};

class MinusExpression : public Expression
{
public:
    virtual void interpreter(Context *context)
    {
        int num = context->getNum();
        num -- ;
        context->setNum(num);
        context->setRes(num);
    }
};
```

```
void main()
{
    Context *pcxt = new Context(10);
    Expression *e1 = new PlusExpression;
    e1->interpreter(pcxt);
    cout << "PlusExpression:" << pcxt->getRes() << endl;

    Expression *e2 = new MinusExpression;
    e2->interpreter(pcxt);
    cout << "MinusExpression:" << pcxt->getRes() << endl;

    delete e2;
    delete e1;
    system("pause");
    return ;
}
```

4.11 迭代器模式 iterator

概念

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。

1.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

2.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？

1，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。

2，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。

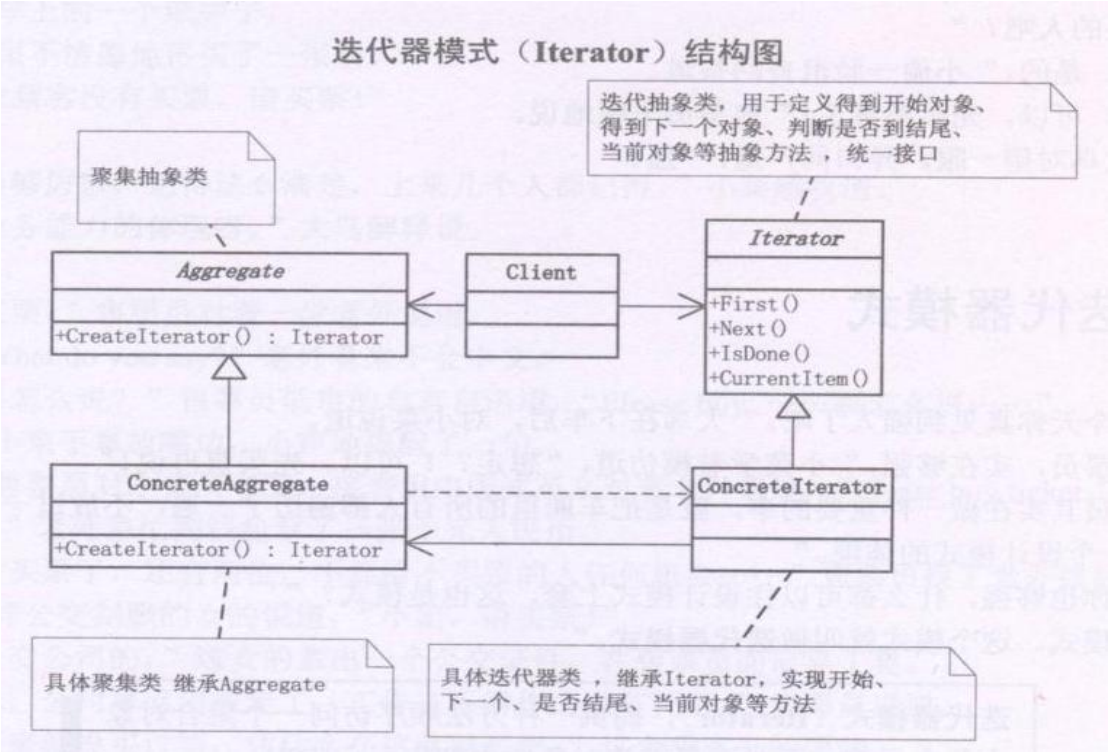
Iterator 模式就是为了有效地处理按顺序进行遍历访问的一种设计模式，简单地说，Iterator 模式提供一种有效的方法，可以屏蔽聚集对象集合的容器类的实现细节，而能对容器内包含的对象元素按顺序进行有效的遍历访问。所以，Iterator 模式的应用场景可以归纳为满足以下几个条件：

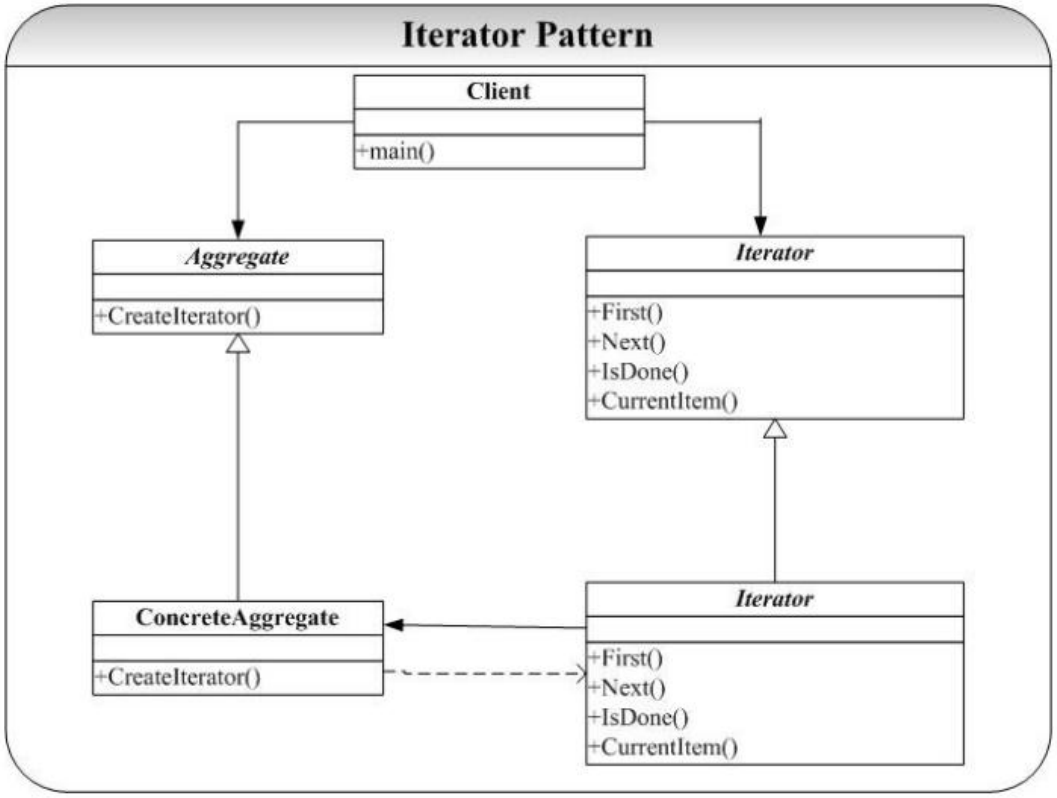
- 访问容器中包含的内部对象
- 按顺序访问

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

案例

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

typedef int Object ;
#define SIZE 5
```

```
//注意类的顺序
class MyIterator
{
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() = 0;
    virtual Object CurrentItem() = 0;
};

class Aggregate
{
public:
    virtual Object getItem(int index) = 0;
    virtual MyIterator *CreateIterator() = 0;
    virtual int getSize() = 0;
};

class ContreteIterator : public MyIterator
{
public:
    ContreteIterator(Aggregate *ag)
    {
        _ag = ag;
        _idx = 0;
    }
    ~ContreteIterator()
    {
        _ag = NULL;
        _idx = 0;
    }

    virtual void First()
    {
        _idx = 0;
    }
    virtual void Next()
    {
        if (_idx <    _ag->getSize())
        {
            _idx ++;
        }
    }
```

```
    virtual bool IsDone()
    {
        return (_idx == _ag->getSize());
    }
    virtual Object CurrentItem()
    {
        return _ag->getItem(_idx);
    }

protected:
private:
    int             _idx;
    Aggregate  *_ag;
};

class ConcreteAggregate : public Aggregate
{
public:
    ConcreteAggregate()
    {
        for (int i=0; i<SIZE; i++)
        {
            object[i] = i+1;
        }
    }
    virtual ~ConcreteAggregate()
    {

    }
    virtual Object getItem(int index)
    {
        return object[index];
    }
    virtual MyIterator *CreateIterator()
    {
        return new ContreteIterator(this);
    }
    virtual int getSize()
    {
        return SIZE;
    }
protected:
private:
    Object object[SIZE];
```

```
};

void main21()
{
    // 创建一个集合
    Aggregate *ag = new ConcreteAggregate();
    // 创建一个遍历这个集合的 迭代器
    MyIterator *it = ag->CreateIterator();


    //通过迭代器 遍历 集合
    for (; !(it->IsDone()); it->Next() )
    {
        cout << it->CurrentItem() << " ";
    }

    //清理相关资源
    delete it;
    delete ag;
}

void main()
{
    main21();
    system("pause");
    return ;
}
```

```
EngPerson
{
Public:
    Void dowork1(Car *car)
{
    Car. Run();
}
Void dowork2()
{
m_car.run()
}
Private:
    Car m_car;
}
```

5 类与类之间关系

类与类之间的关系对于理解面向对象具有很重要的作用，以前在面试的时候也经常被问到这个问题，在这里我就介绍一下。
类与类之间存在以下关系:
(1)泛化(Generalization)
(2)关联(Association)
(3)依赖(Dependency)
(4)聚合(Aggregation)
UML 图与应用代码例子:

UML 表示和代码表示
//依赖(虚线): 一个类是另外一个类的函数参数或者函数返回值
//关联(实线) 关联 张三 有车 一个类 是 另外一个类的成员变量
//聚合(菱形实线)：整体和部分的关系 汽车 发动机 (汽车可以选择各个型号的发动机)
//组合(实心形加实线)：生命体,整体和部分的关系 汽车 发动机 (人 和 五脏六腑)

1 泛化(Generalization)

[泛化]
表示类与类之间的继承关系，接口与接口之间的继承关系，或类对接口的实现关系。一般化的关系是从子类指向父类的，与继承或实现的方法相反。
[具体表现]
父类 父类实例＝new 子类()
[UML 图](图 1.1)

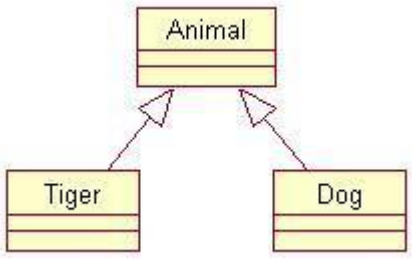


图 1.1 Animal 类与 Tiger 类,Dog 类的泛化关系

[代码表现]

```
class Animal{}
class Tiger extends Animal{}
public class Test
{
   public void test()
   {
```

```
    Animal a=new Tiger();
  }
}
```

2 依赖(Dependency)

对于两个相对独立的对象，当一个对象负责构造另一个对象的实例，或者依赖另一个对象的服务时，这两个对象之间主要体现为依赖关系。

[具体表现]

依赖关系表现在局部变量，方法的参数，以及对静态方法的调用

[现实例子]

比如说你要去拧螺丝，你是不是要借助(也就是依赖)螺丝刀(Screwdriver)来帮助你完成拧螺丝(screw)的工作

[UML 表现](图 1.2)

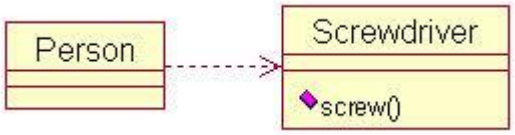


图 1.2 Person 类与 Screwdriver 类的依赖关系

[代码表现]

```
public class Person{
  /** 拧螺丝 */
  public void screw(Screwdriver screwdriver){
    screwdriver.screw();
  }
   Void drive（Car &mycar）{}

}
```

3 关联(Association)

对于两个相对独立的对象，当一个对象的实例与另一个对象的一些特定实例存在固定的对应关系时，这两个对象之间为关联关系。

[具体表现]

关联关系是使用实例变量来实现

[现实例子]

比如客户和订单，每个订单对应特定的客户，每个客户对应一些特定的订单；再例如公司和员工，每个公司对应一些特定的员工，每个员工对应一特定的公司

[UML 图] (图 1.3)

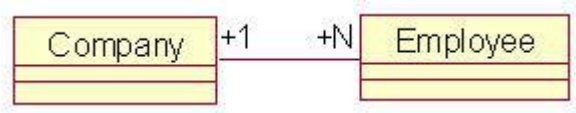


图 1.3 公司和员工的关联关系

[代码表现]

```
public class Company{
   private Employee employee;
   public Employee getEmployee(){
      return employee;
   }
   public void setEmployee(Employee employee){
      this.employee=employee;
   }
   //公司运作
   public void run(){
      employee.startWorking();
   }

}
```

测试案例：

```
Person
{
Public:
     Void dowork1(Car *car) //借别人的车去上班 可以借用 A、B 的车，依赖于别人
     {
        Car. Run();
      }
     Void dowork2()
     {
       m_car.run() //开自己的车上班
     }
     Private:
        Car m_car;
}
```

Person 和 Car 是关联关系
Person 和 car 是依赖关系 Void dowork1(Car *car)

4 聚合（Aggregation）

当对象 A 被加入到对象 B 中，成为对象 B 的组成部分时，对象 B 和对象 A 之间为聚集

关系。聚合是关联关系的一种，是较强的关联关系，强调的是整体与部分之间的关系。
[具体表现]
与关联关系一样，聚合关系也是通过实例变量来实现这样关系的。关联关系和聚合关系来语法上是没办法区分的，从语义上才能更好的区分两者的区别。
[关联与聚合的区别]
(1)关联关系所涉及的两个对象是处在同一个层次上的。比如人和自行车就是一种关联关系，而不是聚合关系，因为人不是由自行车组成的。
聚合关系涉及的两个对象处于不平等的层次上，一个代表整体，一个代表部分。比如电脑和它的显示器、键盘、主板以及内存就是聚集关系，因为主板是电脑的组成部分。
(2)对于具有聚集关系（尤其是强聚集关系）的两个对象，整体对象会制约它的组成对象的生命周期。部分类的对象不能单独存在，它的生命周期依赖于整体类的对象的生命周期，当整体消失，部分也就随之消失。比如张三的电脑被偷了，那么电脑的所有组件也不存在了，除非张三事先把一些电脑的组件（比如硬盘和内存）拆了下来。
[UML 图](图 1.4)

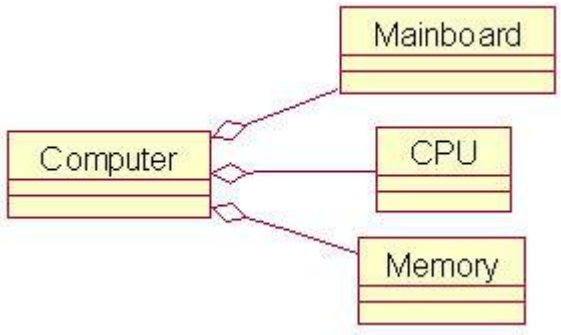
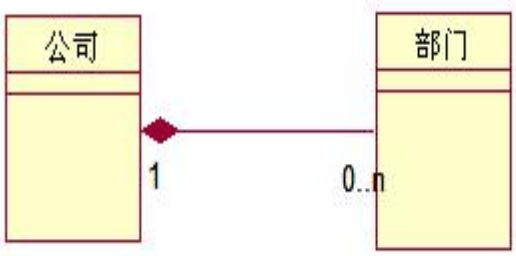


图 1.3 电脑和组件的聚合关系

[代码表现]
```
public class Computer{
   private CPU cpu;
   public CPU getCPU(){
      return cpu;
   }
   public void setCPU(CPU cpu){
      this.cpu=cpu;
   }
   //开启电脑
   public void start(){
      //cpu 运作
      cpu.run();
   }
}
```

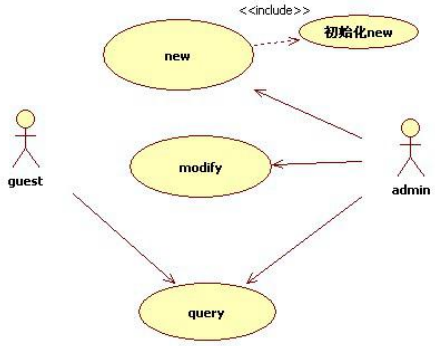
5 组合关系

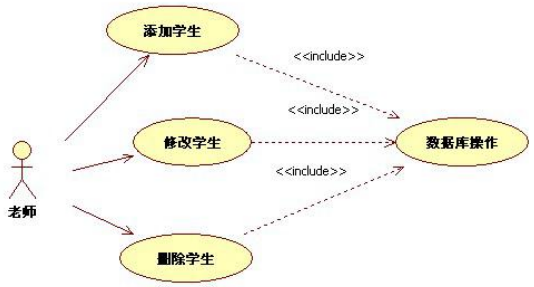
代表整体的对象负责代表部分对象的生命周期。公司不存在，部门也没有意义了。
再例如：人和五脏六腑、四肢的关系。组合关系。

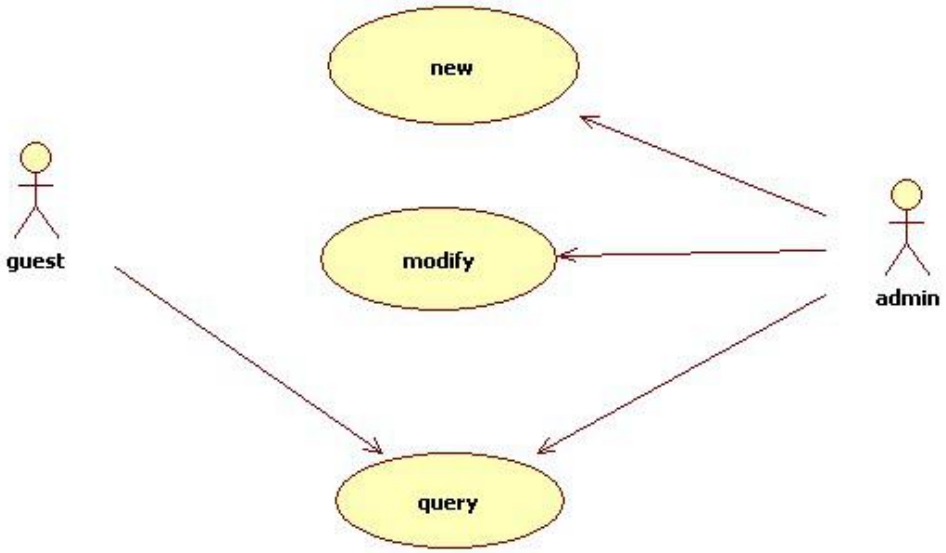


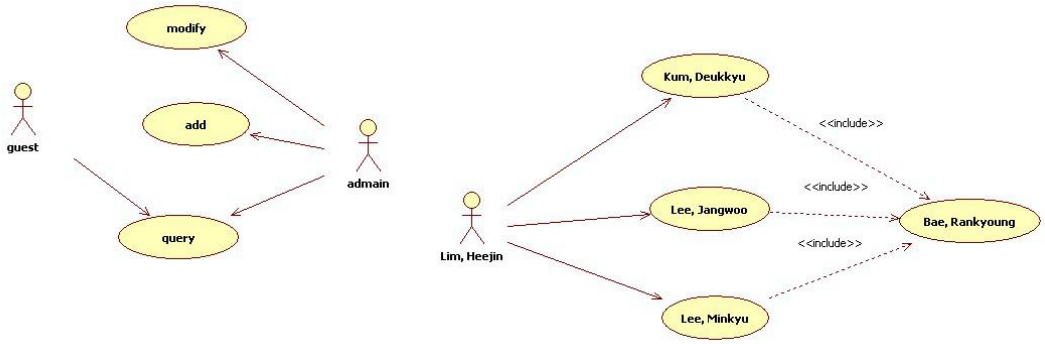
6UML 图

1 用例图

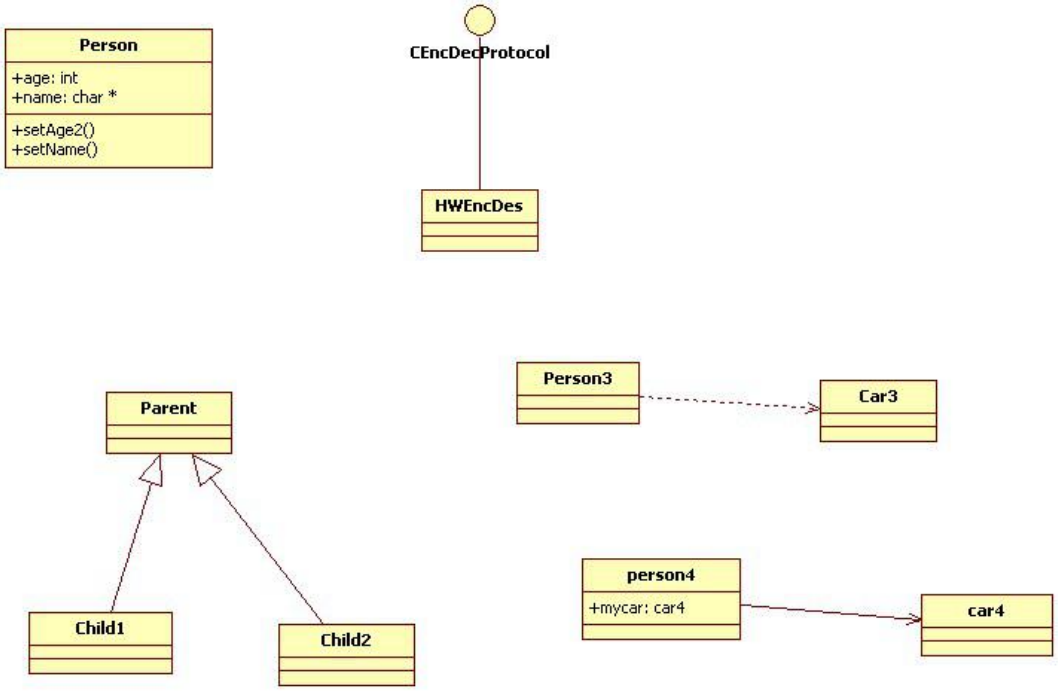




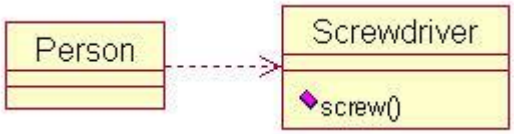

new
guest
modify
admin
query


modify
guest
add
admain
query
Kum, Deukkyu
<<include>>
<<include>>
Lee, Jangwoo
Bae, Rankyoung
<<include>>
Lim, Heejin
Lee, Minkyu

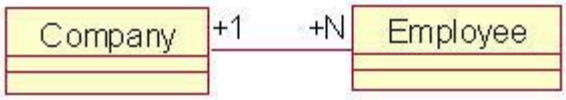
2 类图



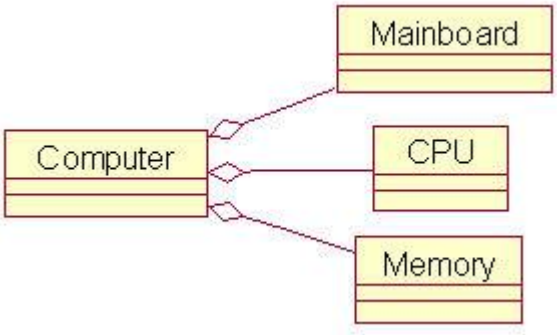
依赖



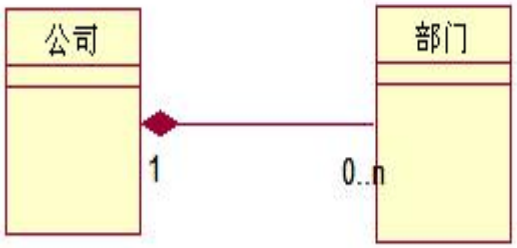
关联



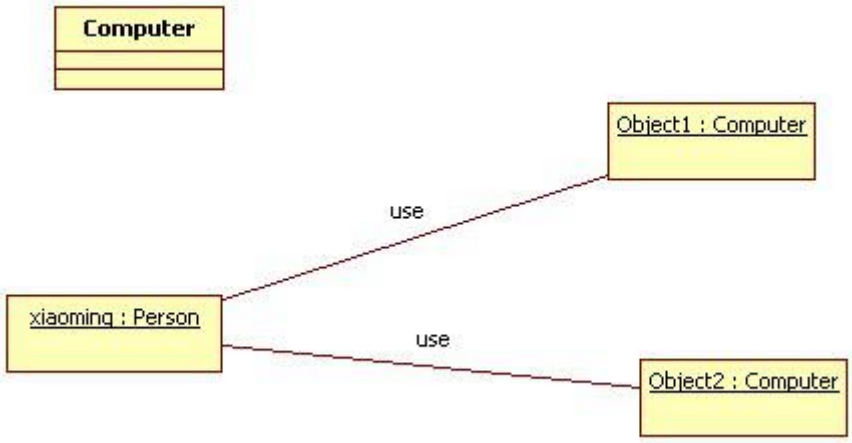
聚合



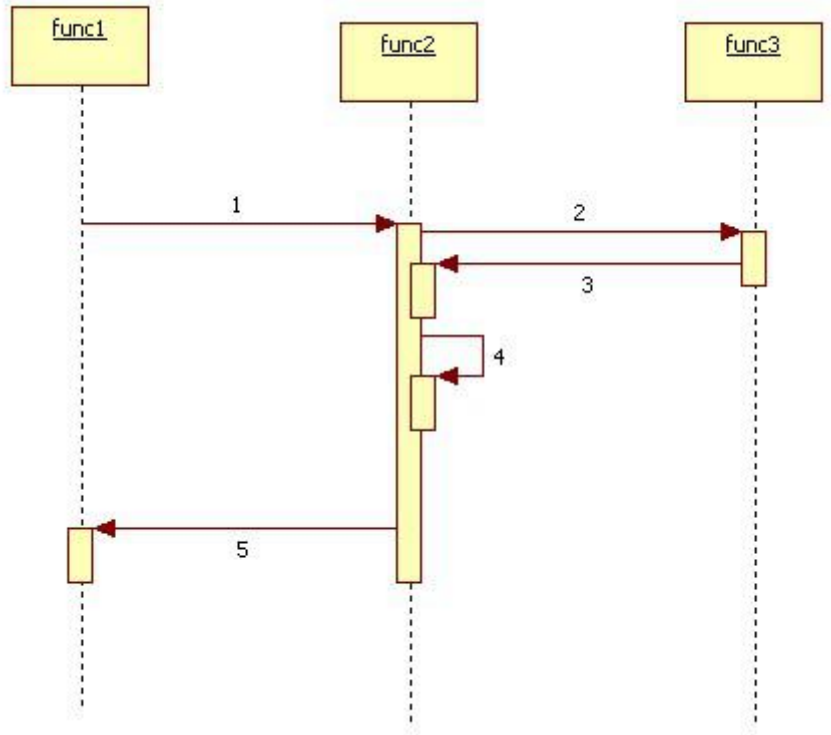
组合


公司
部门
1
0..n

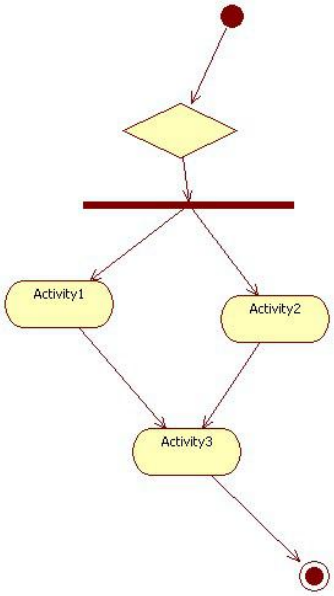
3 对象图

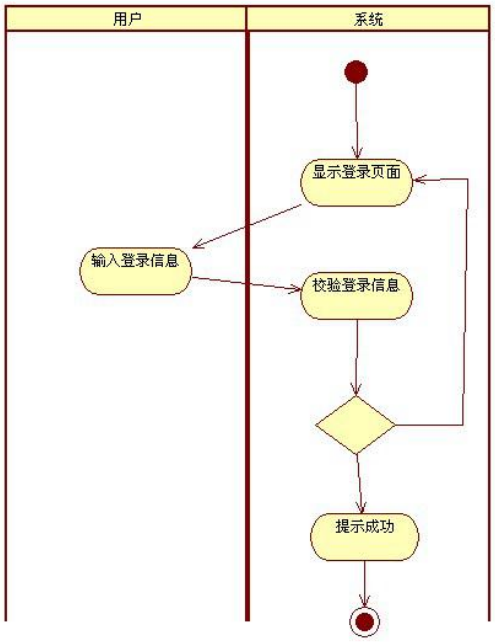

Computer
Object1 : Computer
use
xiaoming : Person
use
Object2 : Computer

4 时序图

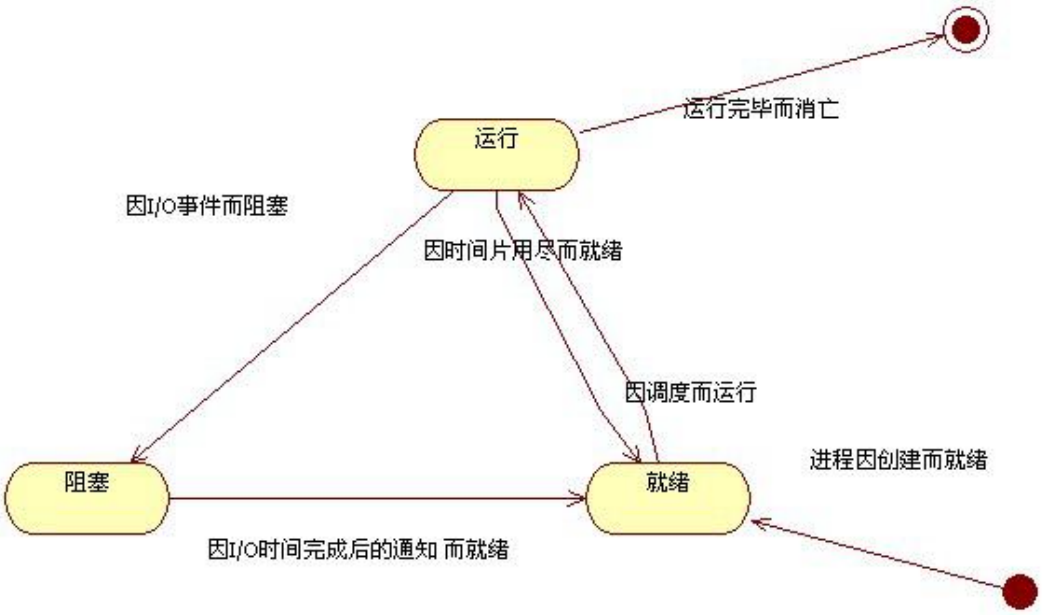

func1
func2
func3
1
2
3
4
5

5 活动图

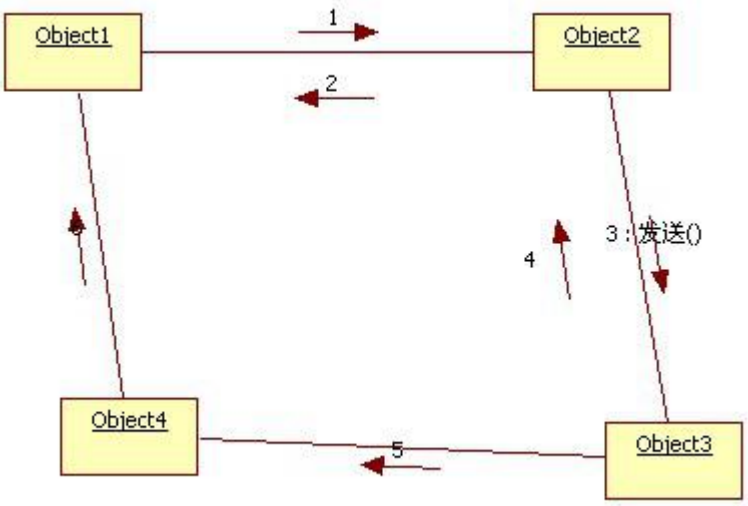

Activity1
Activity2
Activity3


用户
系统
显示登录页面
输入登录信息
校验登录信息
提示成功

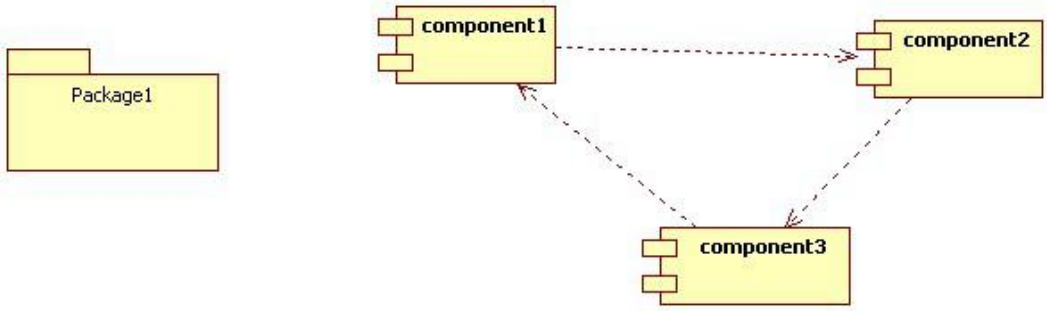
6 状态图


运行
运行完毕而消亡
因I/O事件而阻塞
因时间片用尽而就绪
因调度而运行
阻塞
就绪
进程因创建而就绪
因I/O时间完成后的通知 而就绪

7 协作图


Object1
Object2
1
2
3 : 发送()
4
Object4
Object3
5

8 和 9 包图和组件图



10 部署图