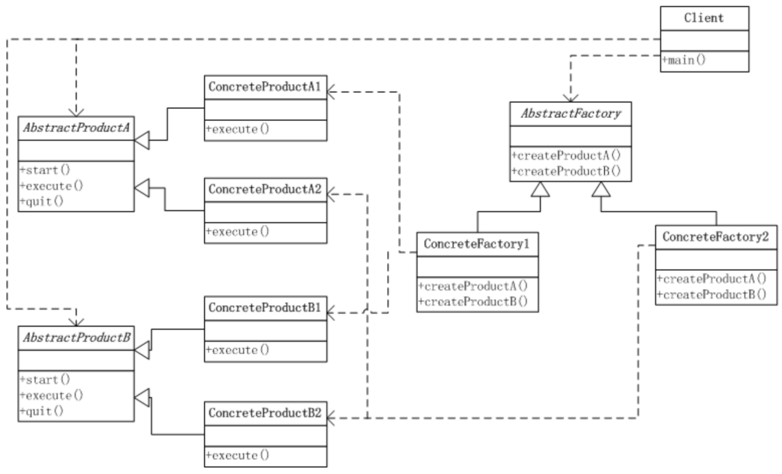
Abstract Factory 抽象工厂

动机（Motivation）:在软件系统中，经常面临着“一系列相互依赖的对象”的创建工作；同时，由于需求的变化，往往存在更多系列对象的创建工作。如何应对这种变化？如何绕过常规的对象创建方法(new)，提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？

定义:提供一个接口，让该接口负责创建一系列“相关或者相互依赖的对象”，无需指定它们具体的类。——《设计模式》GoF

- 客户`Client`
- 抽象工厂接口`AbstractFactory`
- 抽象工厂的实现类`ConcreteFactory`
- 抽象产品接口`AbstractProduct`
- 产品实现类`ConcreteProduct`



要点总结: 如果没有应对“多系列对象构建”的需求变化，则没有必要使用Abstract Factory模式，这时候使用简单的工厂完全可以。“系列对象”指的是在某一特定系列下的对象之间有相互依赖、或作用的关系。不同系列的对象之间不能相互依赖。Abstract Factory模式主要在于应对“新系列”的需求变动。其缺点在于难以应对“新对象”的需求变动

意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

主要解决：主要解决接口选择的问题。

何时使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

如何解决：在一个产品族里面，定义多个产品。

关键代码：在一个工厂里聚合多个同类产品。

优点：当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

缺点：产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

使用场景： 1、QQ 换皮肤，一整套一起换。 2、生成不同操作系统的程序。

注意事项：产品族难扩展，产品等级易扩展。

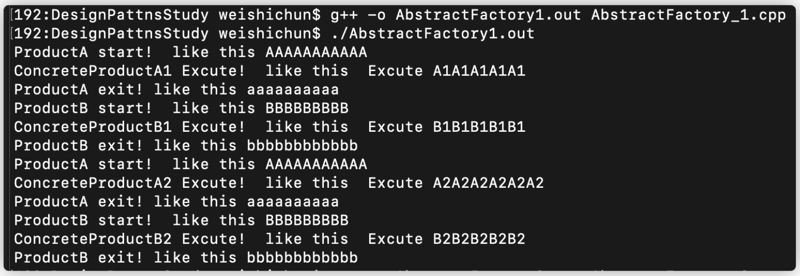
```
#include <iostream>
using namespace std;
class AbstractProductA{
public:
```

```cpp
    AbstractProductA(){}
    virtual ~AbstractProductA(){}
    void start(){
        cout << "ProductA start!  like this AAAAAAAAAAA" << endl;
    }
     void Exit(){
         cout << "ProductA exit! like this aaaaaaaaaa" <<  endl;
     }
     virtual void Excute() = 0;
};
class ConcreteProductA1 : public AbstractProductA{
public:
    ConcreteProductA1(){};
    ~ConcreteProductA1(){};
    void Excute(){
        cout << "ConcreteProductA1 Excute!  like this  Excute A1A1A1A1A1" << endl;
    }
};
class ConcreteProductA2 : public AbstractProductA{
public:
    ConcreteProductA2(){};
    ~ConcreteProductA2(){};
    void Excute(){
        cout << "ConcreteProductA2 Excute!  like this  Excute A2A2A2A2A2A2" << endl;
    }
};
class AbstractProductB{
public:
    AbstractProductB(){}
    virtual ~AbstractProductB(){}
    void start(){
        cout << "ProductB start!  like this BBBBBBBBB" << endl;
    }
    void Exit(){
        cout << "ProductB exit! like this bbbbbbbbbbbb" <<  endl;
    }
    virtual void Excute() = 0;
};
class ConcreteProductB1 : public AbstractProductB{
public:
    ConcreteProductB1(){};
    ~ConcreteProductB1(){};
    void Excute(){
        cout << "ConcreteProductB1 Excute!  like this  Excute B1B1B1B1B1" << endl;
    }
};
class ConcreteProductB2 : public AbstractProductB{
public:
    ConcreteProductB2(){};
    ~ConcreteProductB2(){};
    void Excute(){
        cout << "ConcreteProductB2 Excute!  like this  Excute B2B2B2B2B2" << endl;
    }
};
```

```cpp
class AbstractFactory{
public:
    AbstractFactory(){}
    virtual ~AbstractFactory(){}
    virtual AbstractProductA * CreateProductA() = 0;
    virtual AbstractProductB * CreateProductB() = 0;
};
class ConcretFactory1 : public AbstractFactory{
public:
    ConcretFactory1(){}
    ~ConcretFactory1(){}
    AbstractProductA * CreateProductA(){
        return new ConcreteProductA1();
    }
    AbstractProductB * CreateProductB(){
        return new ConcreteProductB1();
    }
};
class ConcretFactory2 : public AbstractFactory{
public:
    ConcretFactory2(){}
    ~ConcretFactory2(){}
    AbstractProductA * CreateProductA(){
        return new ConcreteProductA2();
    }
    AbstractProductB * CreateProductB(){
        return new ConcreteProductB2();
    }
};
int main(){
    AbstractFactory * pFactory = new ConcretFactory1();
    AbstractProductA * pProductA = pFactory->CreateProductA();
    pProductA->start();
    pProductA->Excute();
    pProductA->Exit();
    AbstractProductB * pProductB = pFactory->CreateProductB();
    pProductB->start();
    pProductB->Excute();
    pProductB->Exit();
    delete pFactory;
    delete pProductA;
     delete pProductB;

     pFactory = new ConcretFactory2();
     pProductA = pFactory->CreateProductA();
     pProductA->start();
     pProductA->Excute();
     pProductA->Exit();
     pProductB = pFactory->CreateProductB();
     pProductB->start();
     pProductB->Excute();
     pProductB->Exit();
     delete pFactory;
     delete pProductA;
    delete pProductB;

    return 0;
}
```

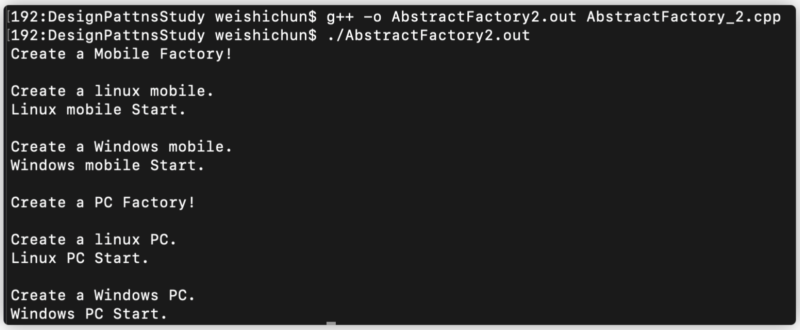


```
192:DesignPattnsStudy weishichun$ g++ -o AbstractFactory1.out AbstractFactory_1.cpp
192:DesignPattnsStudy weishichun$ ./AbstractFactory1.out
ProductA start!  like this AAAAAAAAAAA
ConcreteProductA1 Excute!  like this  Excute A1A1A1A1A1
ProductA exit! like this aaaaaaaaaa
ProductB start!  like this BBBBBBBBB
ConcreteProductB1 Excute!  like this  Excute B1B1B1B1B1
ProductB exit! like this bbbbbbbbbbbb
ProductA start!  like this AAAAAAAAAAA
ConcreteProductA2 Excute!  like this  Excute A2A2A2A2A2A2
ProductA exit! like this aaaaaaaaaa
ProductB start!  like this BBBBBBBBB
ConcreteProductB2 Excute!  like this  Excute B2B2B2B2B2
ProductB exit! like this bbbbbbbbbbbb
```

```
#include <iostream>
using namespace std;
class CLinux{
public:
    virtual ~CLinux() {}
    virtual void Start() = 0;
};
class CWindows{
public:
     virtual ~CWindows() {}
     virtual void Start() = 0;
 };
 class CLinuxMobile : public CLinux{
 public:
     CLinuxMobile(){
         cout << "Create a linux mobile." << endl;
     }
     virtual ~CLinuxMobile(){}
     void Start(){
         cout << "Linux mobile Start. " << endl;
     }
 };
 class CLinuxPC: public CLinux{
 public:
     CLinuxPC(){
         cout << "Create a linux PC." << endl;
     }
     virtual ~CLinuxPC(){}
     void Start(){
         cout << "Linux PC Start. " << endl;
     }
 };
 class CWindowsMobile : public CWindows{
 public:
     CWindowsMobile(){
         cout << "Create a Windows mobile." << endl;
     }
     virtual ~CWindowsMobile(){}
```

```cpp
    void Start(){
        cout << "Windows mobile Start. " << endl;
    }
};
class CWindowsPC: public CWindows{
public:
    CWindowsPC(){
        cout << "Create a Windows PC." << endl;
    }
    virtual ~CWindowsPC(){}
    void Start(){
        cout << "Windows PC Start. " << endl;
    }
};
class CFactory{
public:
    virtual ~CFactory(){}
    virtual CLinux * CreateLinuxProduct() = 0;
    virtual CWindows * CreateWindowsProduct() = 0;
};
class CMobileFactory : public CFactory{
public:
   CMobileFactory(){
       cout << "Create a Mobile Factory! " << endl << endl;
   }
   virtual ~CMobileFactory(){}
   CLinux * CreateLinuxProduct(){
       return new CLinuxMobile();
   }
   CWindows * CreateWindowsProduct(){
       return new CWindowsMobile();
   }
};
class CPCFactory : public CFactory{
public:
   CPCFactory(){
       cout << "Create a PC Factory! " << endl << endl;
   }
   virtual ~CPCFactory(){}
   CLinux * CreateLinuxProduct(){
       return new CLinuxPC();
   }
   CWindows * CreateWindowsProduct(){
       return new CWindowsPC();
   }
};
void CreateProduct(CFactory * pFactory){
    CLinux *pLinux= pFactory->CreateLinuxProduct();
    pLinux->Start();
    cout << endl;
    CWindows *pWindows = pFactory->CreateWindowsProduct();
    pWindows->Start();
    delete pLinux;
    delete pWindows;
```

```cpp
}
int main()
{
    CFactory *pFactory = new CMobileFactory();
    CreateProduct(pFactory);
    delete  pFactory;
    cout << endl;
     pFactory = new CPCFactory();
     CreateProduct(pFactory); delete pFactory; return 0;
 }
```



```
192:DesignPattnsStudy weishichun$ g++ -o AbstractFactory2.out AbstractFactory_2.cpp
192:DesignPattnsStudy weishichun$ ./AbstractFactory2.out
Create a Mobile Factory!

Create a linux mobile.
Linux mobile Start.

Create a Windows mobile.
Windows mobile Start.

Create a PC Factory!

Create a linux PC.
Linux PC Start.

Create a Windows PC.
Windows PC Start.
```