AlexOne

八、工厂模式

目录

1.1 简单工厂

```
#include <iostream>

using namespace std;

// 1.简单工厂
// 定义一个用于创建对象的接口，让子类决定实例化哪一个类，将实例化延迟到子类。有选择地实例化
//
#if 0
             SingleCore
            /    |    \
           /     |     \
          /      |      \
SingleCoreA SingleCoreB SingleCoreC
           \     |     /
            \    |    /
              Factory
#endif

enum CoreType{
    COREA,COREB,COREC
};

class SingleCore{
public:
    virtual void work()=0;
    //virtual ~SingleCore(){}
};

class SingleCoreA:public SingleCore{
public:
    void work(){
        cout<<"SingleCoreA"<<endl;
    }
};

class SingleCoreB:public SingleCore{
public:
    void work(){
        cout<<"SingleCoreB"<<endl;
    }
};

class SingleCoreC:public SingleCore{
public:
    void work(){
        cout<<"SingleCoreC"<<endl;
    }
};

class Factory{
public:
    SingleCore * createSingleCore(enum CoreType type){
        if(type==COREA)
            return new SingleCoreA;
        if(type==COREB)
            return new SingleCoreB;
        if(type==COREC)
            return new SingleCoreC;
        //else return NULL;
    }
};

int main(){
    Factory fac;     //创建工厂
    // 生产A类产品
    SingleCore *pa = fac.createSingleCore(COREA);
    pa->work();

    // 生产B类产品
    SingleCore *pb = fac.createSingleCore(COREB);
    pb->work();

    // 生产C类产品
    SingleCore *pc = fac.createSingleCore(COREC);
    pc->work();

    delete pa; delete pb; delete pc;

    return 0;
}
```

1.2 简单工厂的升级

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

// 1.简单工厂
// 定义一个用于创建对象的接口，让子类决定实例化哪一个类，将实例化延迟到子类
#if 0
             SingleCore
            /    |    \
           /     |     \
          /      |      \
SingleCoreA SingleCoreB SingleCoreC
           \     |     /
            \    |    /
```

```cpp
              Factory
#endif

enum CoreType{
    COREA,COREB,COREC
};

class SingleCore{
public:
    virtual void work()=0;
};

class SingleCoreA:public SingleCore{
public:
    void work(){
        cout<<"SingleCoreA"<<endl;
    }
};

class SingleCoreB:public SingleCore{
public:
    void work(){
        cout<<"SingleCoreB"<<endl;
    }
};

class SingleCoreC:public SingleCore{
public:
    void work(){
        cout<<"SingleCoreC"<<endl;
    }
};

class Factory{
public:
    virtual SingleCore * createSingleCore() = 0;
};

class FactoryCA:public Factory{
public:
    SingleCore *createSingleCore(){
        return new SingleCoreA;
    }
};

class FactoryCB:public Factory{
public:
    SingleCore *createSingleCore(){
        return new SingleCoreB;
    }
};

class FactoryCC:public Factory{
public:
    SingleCore *createSingleCore(){
        return new SingleCoreC;
    }
};

// 加入内存管理
```

```
int main(){
    Factory *fca = new FactoryCA();
    fca->createSingleCore()->work();

    Factory *fcb = new FactoryCB();
    fcb->createSingleCore()->work();

    return 0;
}
```

2.1 抽象工厂

```
#include <iostream>
using namespace std;
// https://blog.csdn.net/taiyang1987912/article/details/43148913
// https://blog.csdn.net/taiyang1987912/article/category/2859245/2?
// product.h
class Product
{
    public:
        virtual ~Product() = 0;
    protected:
        Product() {cout<<" Product"<<endl;}
};
// 上面有了虚析构，下面的析构可以不写
class ConcreteProduct1 : public Product
{
    public:
        ConcreteProduct1() {cout<<" ConcreteProduct1"<<endl;}
        ~ConcreteProduct1(){cout<<"~ConcreteProduct1"<<endl;}
};

class ConcreteProduct3 : public Product
{
    public:
        ConcreteProduct3() {cout<<" ConcreteProduct3"<<endl;}
        ~ConcreteProduct3(){cout<<"~ConcreteProduct3"<<endl;}
};

// product纯虚析构可以有实现
Product::~Product()
{
    cout<<"~Product"<<endl;
}
// factory
class Factory
{
    public:
        virtual ~Factory() = 0;
        virtual Product *CreateProduct() = 0;
        void setFactoryMethod(int flag);
    protected:
        Factory():_flag(0){}
        int _flag;
};

class ConcreteFactory : public Factory
{
    public:
        ConcreteFactory() {cout<<" ConcreteFactory"<<endl;}
        ~ConcreteFactory(){cout<<"~ConcreteFactory"<<endl;}
        virtual Product *CreateProduct();
};
Factory::~Factory()
{
}
void Factory::setFactoryMethod(int flag)
{
    _flag = flag;
}

Product *ConcreteFactory::CreateProduct()  //创建操作
{
    if(_flag == 1)
        return new ConcreteProduct1();     //子类中决定要实例化哪一个类
    else if(_flag == 3)
        return new ConcreteProduct3();
    //else return NULL;
}

int main()
{
    Factory *fac = new ConcreteFactory();

    //ConcreteFactory通过参数延时决定具体到底创建哪一个Product的子类
    fac->setFactoryMethod(3);
    Product *pro = fac->CreateProduct();

    if (pro) {
        delete pro;
        pro = NULL;
    }
    if (fac) {
        delete fac;
        fac = NULL;
    }
    return 0;
}
```

2.2 抽象工厂

```
#include <iostream>
//using namespace std;
using std::cout;
using std::endl;

class AbstractProductA
{
    public:
        virtual ~AbstractProductA() {cout<<" AbstractProductA"<<endl;}
    protected:
        AbstractProductA() {cout<<"~AbstractProductA"<<endl;}
};

class AbstractProductB
{
```

```cpp
    public:
        virtual ~AbstractProductB() {cout<<"~AbstractProductB"<<endl;}
    protected:
        AbstractProductB() {cout<<" AbstractProductB"<<endl;}
};
class ProductA1 : public AbstractProductA
{
    public:
        ProductA1() {cout<<" ProductA1"<<endl;}
        ~ProductA1(){cout<<"~ProductA1"<<endl;}
};
class ProductA2 : public AbstractProductA
{
    public:
        ProductA2() {cout<<" ProductA2"<<endl;}
        ~ProductA2(){cout<<"~ProductA2"<<endl;}
};
class ProductB1 : public AbstractProductB
{
    public:
        ProductB1() {cout<<" ProductB1"<<endl;}
        ~ProductB1(){cout<<"~ProductB1"<<endl;}
};
class ProductB2 : public AbstractProductB
{
    public:
        ProductB2() {cout<<" ProductB2"<<endl;}
        ~ProductB2(){cout<<"~ProductB2"<<endl;}
};

// factory
class AbstractFactory
{
    public:
        virtual AbstractProductA * CreateProductA() = 0;
        virtual AbstractProductB * CreateProductB() = 0;
        virtual ~AbstractFactory(){
            if (pointA1) delete pointA1;
            if (pointA2) delete pointA2;
            if (pointB1) delete pointB1;
            if (pointB2) delete pointB2;
        }
    protected:
        AbstractFactory(){
            pointA1 = NULL;
            pointA2 = NULL;
            pointB1 = NULL;
            pointB2 = NULL;
        }
        AbstractProductA * pointA1;
        AbstractProductA * pointA2;
        AbstractProductB * pointB1;
        AbstractProductB * pointB2;
};

class ConcreteFactory1 : public AbstractFactory
{
    public:
        ConcreteFactory1() {cout<<" ConcreteFactory1"<<endl;}
        ~ConcreteFactory1(){cout<<"~ConcreteFactory1"<<endl;}
        AbstractProductA * CreateProductA(){
            return pointA1 = new ProductA1();
        }
        AbstractProductB * CreateProductB(){
            return pointB1 = new ProductB1();
        }
};

class ConcreteFactory2 : public AbstractFactory
{
    public:
        ConcreteFactory2() {cout<<" ConcreteFactory2"<<endl;}
        ~ConcreteFactory2(){cout<<"~ConcreteFactory2"<<endl;}
        AbstractProductA * CreateProductA(){
            return pointA2 = new ProductA2();
        }
        AbstractProductB * CreateProductB(){
            return pointB2 = new ProductB2();
        }
};

int main(){
    AbstractFactory *cf1 = new ConcreteFactory1();
    cf1->CreateProductA();
    cf1->CreateProductB();

    AbstractFactory *cf2 = new ConcreteFactory2();
    cf2->CreateProductA();
    cf2->CreateProductB();

    delete cf1;
    delete cf2;
    return 0;
}
```