简单工厂模式

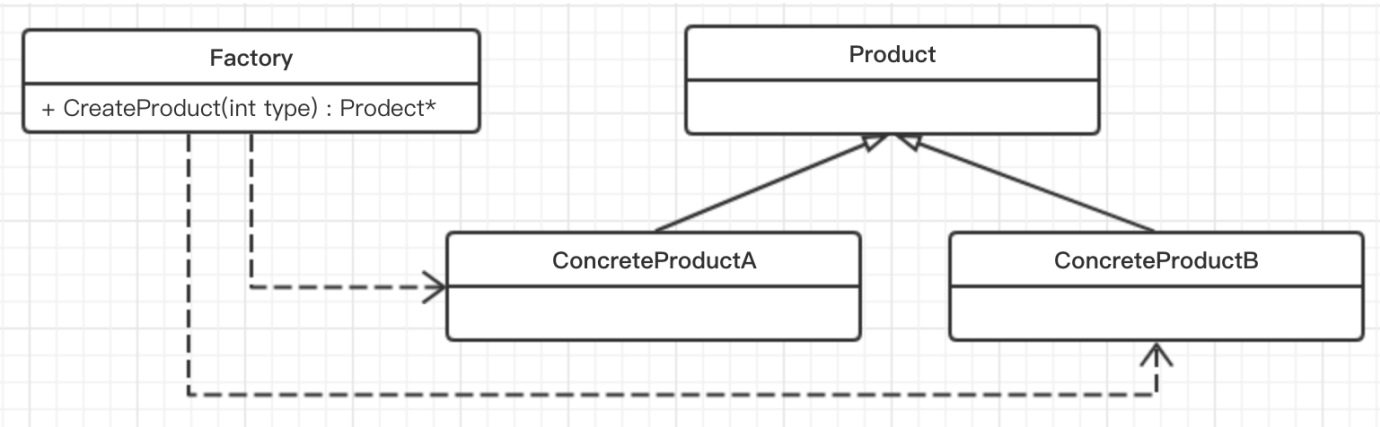
资料 C++简单工厂模式

01.简述

简单工厂模式 又叫静态工厂方法模式，属于创建型模式，简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品的实例。

注意 :简单工厂模式并不属于23中GOF设计模式之一，它是工厂模式家族中最简单实用的模式，可以理解为不同工厂模式的一个特殊实现。

02.UML结构图



- Factory：整个模式的核心，负责实现创建具体产品实例的逻辑（由外部type决定创建那个产品）
- Product：抽象产品是所有产品的基类，负责定义所有产品的公共接口
- ConcreteProduct：具体产品

03.优缺点

优点：

- 工厂类包含了必要的逻辑判断，根据外界指定的类型来决定创建何种产品，实现了创建于使用的分离
- 客户端无需关心具体产品如何创建于组织，仅需要知道具体产品所对应的类型即可

缺点：

- 由于工厂类集中了所有产品的创建逻辑，职责过重，一旦无法正常工作则整个系统都会受到影响
- 每当需要添加产品时，就需要修改工厂类，违背了封闭原则

04.代码实现

```
//
//  T2_20190122.h
//  DesignPattern
//
//  Created by shadot on 2019/1/22.
//  Copyright © 2019 shadot. All rights reserved.
//

#ifndef T2_20190122_h
#define T2_20190122_h

#include <iostream>

using namespace std;

//
//C++简单工厂模式
//

//抽象产品
class ICar
{
public:
    virtual string Name() = 0; //汽车名称
};

//具体产品A
class ACar : public ICar
{
public:
    string Name(){
        return "ACar";
    }
};

//具体产品B
class BCar : public ICar
{
public:
    string Name(){
        return "BCar";
    }
};
```

```

//工厂
class Factory
{
public:
    enum TYPE {A, B};

    ICar* CreateCar(TYPE type){
        ICar* car = nullptr;

        switch (type) {
            case A:
                car = new ACar();
                break;

            case B:
                car = new BCar();
                break;

            default:
                break;
        }

        return car;
    }
};

#endif /* T2_20190122_h */

    //20190122简单工厂模式
    {
        Factory* pFactory = new Factory();

        ICar* pCar1 = pFactory->CreateCar(Factory::A);
        cout << pCar1->Name() << endl;

        ICar* pCar2 = pFactory->CreateCar(Factory::B);
        cout << pCar2->Name() << endl;
    }



```

```

//
// main.cpp
// DesignPattern
```

```cpp
//
// Created by shadot on 2019/1/8.
// Copyright © 2019 shadot. All rights reserved.
//

#include <iostream>
using namespace std;

//运算的抽象基类
class Operation{
public:
    Operation(){
        m_numberX = 0.0;
        m_numberY = 0.0;
    }

    void Setxy(double x, double y){
        m_numberX = x;
        m_numberY = y;
    }

    virtual double GetResult() = 0;

protected:
    double m_numberX;
    double m_numberY;
};

//加减乘除具体实现类
class OperationAdd : public Operation{
public:
    OperationAdd():Operation(){}

    double GetResult(){
        return m_numberX + m_numberY;
    }
};

class OperationSub : public Operation{
public:
    OperationSub():Operation(){}

    double GetResult(){
        return m_numberX - m_numberY;
    }
};
```

```
50
51 class OperationMul : public Operation{
52 public:
53     OperationMul():Operation(){}
54
55     double GetResult(){
56         return m_numberX * m_numberY;
57     }
58 };
59
60 class OperationDiv : public Operation{
61 public:
62     OperationDiv():Operation(){}
63
64     double GetResult(){
65         if (m_numberY == 0)
66             return 0;
67         return m_numberX / m_numberY;
68     }
69 };
70
71 //简单运算工厂类
72 class OperationFactory
73 {
74 public:
75     enum EMOperate {Add, Sub, Mul, Div};
76
77     OperationFactory(){}
78
79     Operation* CreateOperate(EMOperate operate){
80         Operation* pOper = nullptr;
81
82         switch (operate) {
83             case Add:
84                 pOper = new OperationAdd(); break;
85             case Sub:
86                 pOper = new OperationSub(); break;
87             case Mul:
88                 pOper = new OperationMul(); break;
89             case Div:
90                 pOper = new OperationDiv(); break;
91             default: break;
92         }
93         return pOper;
94     }
95 };
```

```
96
97  int main(int argc, const char * argv[]) {
98
99      Operation* pOper = OperationFactory().CreateOperate(OperationFactory::Sub);
100     pOper->Setxy(1, 2);
101     auto result = pOper->GetResult();
102
103     return 0;
104 }
```

简单运算工厂负责如何去实例化对象，以后如果需要再次进行扩展，只是在基于Operation类进行派生，然后重新实现GetResult()方法，再在工厂类switch中添加分支即可