设计模式学习

设计模式学习
- 工厂模式
  - 介绍
  - 简单工厂模式
  - 工厂方法模式
  - 抽象工厂模式
  - 缺陷
  - 解决办法
  - 模板工厂
  - 产品注册模板类+单例工厂模板类
- 单例模式
  - 介绍
  - 懒汉版（Lazy Singleton）
  - 饿汉版（Eager Singleton）
- 观察者模式
  - 介绍
  - 动机
  - 使用场景
  - 结构
  - 优缺点
  - 例子

工厂模式

介绍

属于创建型模式，提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

简单来说，使用了C++**多态**的特性，将存在**继承**关系的类，通过一个工厂类创建对应的子类（派生类）对象。在项目复杂的情况下，可以便于子类对象的创建。

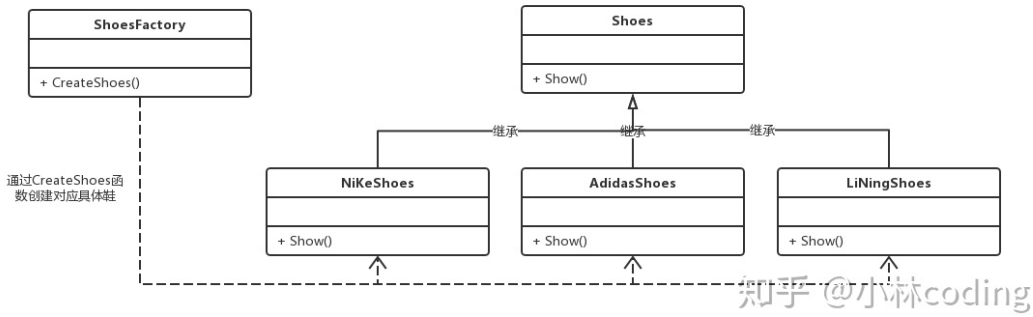
工厂模式的实现方式可分别**简单工厂模式、工厂方法模式、抽象工厂模式**，每个实现方式都存在优和劣。

简单工厂模式

具体的情形：

鞋厂可以指定生产耐克、阿迪达斯和李宁牌子的鞋子。哪个鞋炒的火爆，老板就生产哪个，看形势生产。

UML图：



简单工厂模式的结构组成：

1. 工厂类：工厂模式的核心类，会定义一个用于创建指定的具体实例对象的接口。
2. 抽象产品类：是具体产品类的继承的父类或实现的接口。
3. 具体产品类：工厂类所创建的对象就是此具体产品实例

简单工厂模式的特点：

工厂类封装了创建具体产品对象的函数。

简单工厂模式的缺陷：

扩展性非常差，新增产品的时候，需要去修改工厂类。

简单工厂模式的代码：

Shoes为鞋子的抽象类（基类），接口函数为Show()，用于显示鞋子广告。

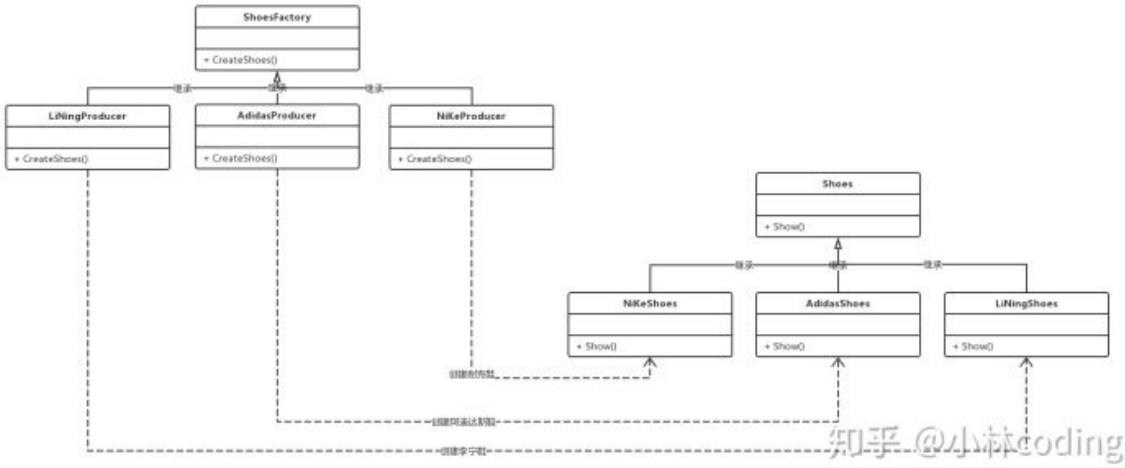
NiKeShoes、AdidasShoes、LiNingShoes为具体鞋子的类，分别是耐克、阿迪达斯和李宁鞋牌的鞋，它们都继承于Shoes抽象类。

工厂方法模式

具体情形：

现各类鞋子抄的非常火热，于是为了大量生产每种类型的鞋子，则要针对不同品牌的鞋子开设独立的生产线，那么每个生产线就只能生产同类型品牌的鞋。

UML图：



工厂方法模式的结构组成：

1. 抽象工厂类：工厂方法模式的核心类，提供创建具体产品的接口，由具体工厂类实现。
2. 具体工厂类：继承于抽象工厂，实现创建对应具体产品对象的方式。
3. 抽象产品类：它是具体产品继承的父类（基类）。
4. 具体产品类：具体工厂所创建的对象，就是此类。

工厂方法模式的特点：

- 工厂方法模式抽象出了工厂类，提供创建具体产品的接口，交由子类去实现。
- 工厂方法模式的应用并不只是为了封装具体产品对象的创建，而是要把具体产品对象的创建放到具体工厂类实现。

工厂方法模式的缺陷：

- 每新增一个产品，就需要增加一个对应的产品的具体工厂类。相比简单工厂模式而言，工厂方法模式需要更多的类定义。
- 一条生产线只能一个产品。

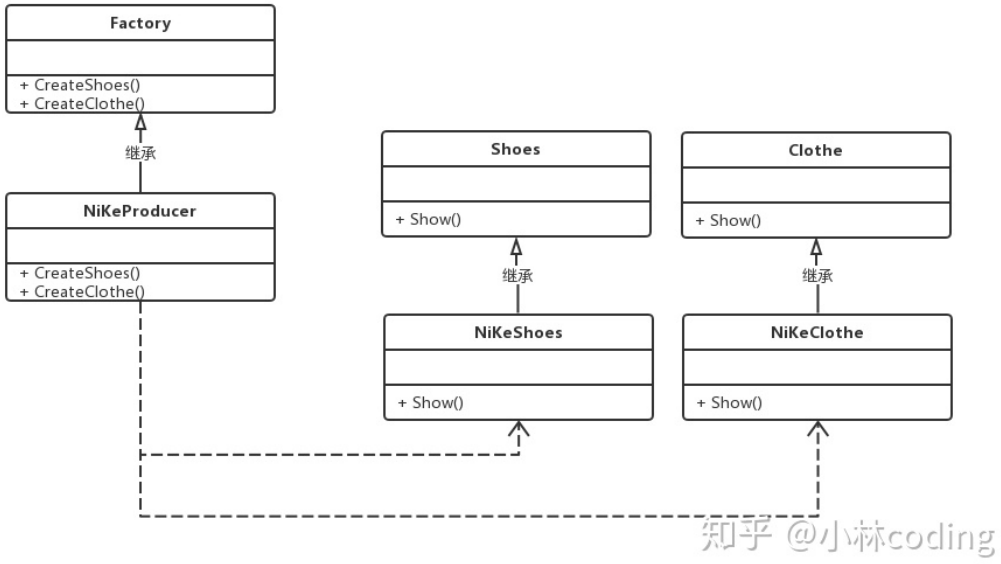
main函数针对每种类型的鞋子，构造了每种类型的生产线，再由每个生产线生产出对应的鞋子。需注意的是具体工厂对象和具体产品对象，用完了需要通过delete释放资源。

抽象工厂模式

具体情形：

鞋厂为了扩大了业务，不仅只生产鞋子，把运动品牌的衣服也一起生产了。

UML图：



抽象工厂模式的结构组成（和工厂方法模式一样）：

1. 抽象工厂类：工厂方法模式的核心类，提供创建具体产品的接口，由具体工厂类实现。
2. 具体工厂类：继承于抽象工厂，实现创建对应具体产品对象的方式。
3. 抽象产品类：它是具体产品继承的父类（基类）。
4. 具体产品类：具体工厂所创建的对象，就是此类。

抽象工厂模式的特点：

提供一个接口，可以创建多个产品族中的产品对象。如创建耐克工厂，则可以创建耐克鞋子产品、衣服产品、裤子产品等。

抽象工厂模式的缺陷：

同工厂方法模式一样，新增产品时，都需要增加一个对应的产品的具体工厂类。

缺陷

以上三种工厂模式，在新增产品时，都存在一定的缺陷。

- 简单工厂模式，，需要去修改工厂类，这违背了开闭法则。
- 工厂方式模式和抽象工厂模式，都需要增加一个对应的产品的具体工厂类，这就会增大了代码的编写量。

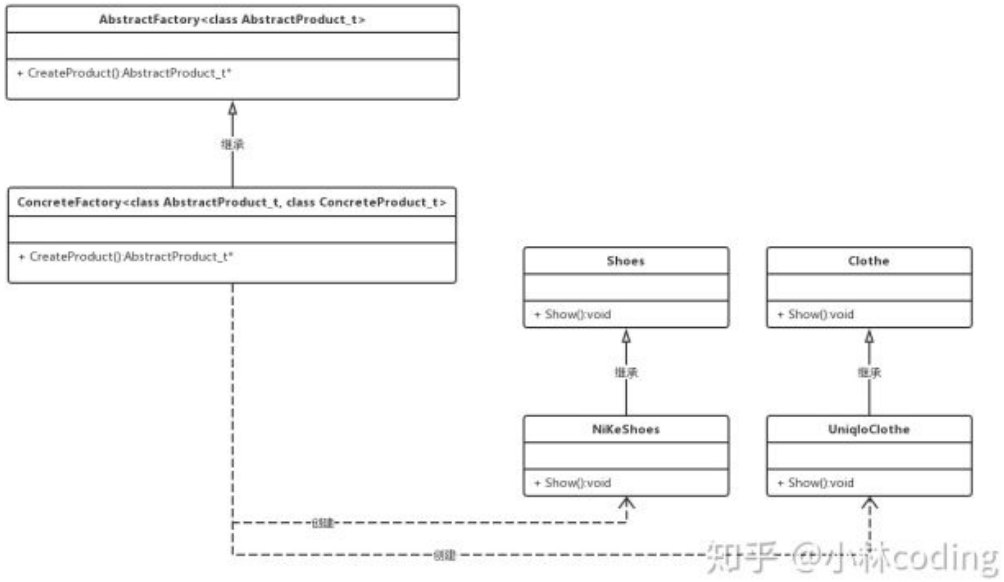
解决办法

将工厂类的封装性提高，达到新增产品时，也不需要修改工厂类，不需要新增具体的工厂类。封装性高的工厂类特点是扩展性高、复用性也高。

模板工厂

针对工厂方法模式封装成模板工厂类，那么这样在新增产品时，是不需要新增具体的工厂类，减少了代码的编写量。相当于用模板来解决

UML图：



代码描述：

AbstractFactory为抽象模板工厂类，其中模板参数：AbstractProduct_t`产品抽象类，如Shoes、Clothe

ConcreteFactory为具体模板工厂类，其中模板参数：AbstractProduct_t产品抽象类（如Shoes、Clothe），ConcreteProduct_t 产品具体类（如NiKeShoes、UniqloClothe）

```
// 抽象模板工厂类
// 模板参数：AbstractProduct_t 产品抽象类
template <class AbstractProduct_t>
class AbstractFactory
{
public:
    virtual AbstractProduct_t *CreateProduct() = 0;
    virtual ~AbstractFactory() {}
};

// 具体模板工厂类
// 模板参数：AbstractProduct_t 产品抽象类，ConcreteProduct_t 产品具体类
template <class AbstractProduct_t, class ConcreteProduct_t>
class ConcreteFactory : public AbstractFactory<AbstractProduct_t>
{
public:
    AbstractProduct_t *CreateProduct()
    {
        return new ConcreteProduct_t();
    }
};

int main()
{
    // 构造耐克鞋的工厂对象
    ConcreteFactory<Shoes, NiKeShoes> nikeFactory;
    // 创建耐克鞋对象
    Shoes *pNiKeShoes = nikeFactory.CreateProduct();
    // 打印耐克鞋广告语
    pNiKeShoes->Show();
```

```
    // 构造优衣库衣服的工厂对象
    ConcreteFactory<Clothe, UniqloClothe> uniqloFactory;
    // 创建优衣库衣服对象
    Clothe *pUniqloClothe = uniqloFactory.CreateProduct();
    // 打印优衣库广告语
    pUniqloClothe->Show();
}
```

产品注册模板类+单例工厂模板类

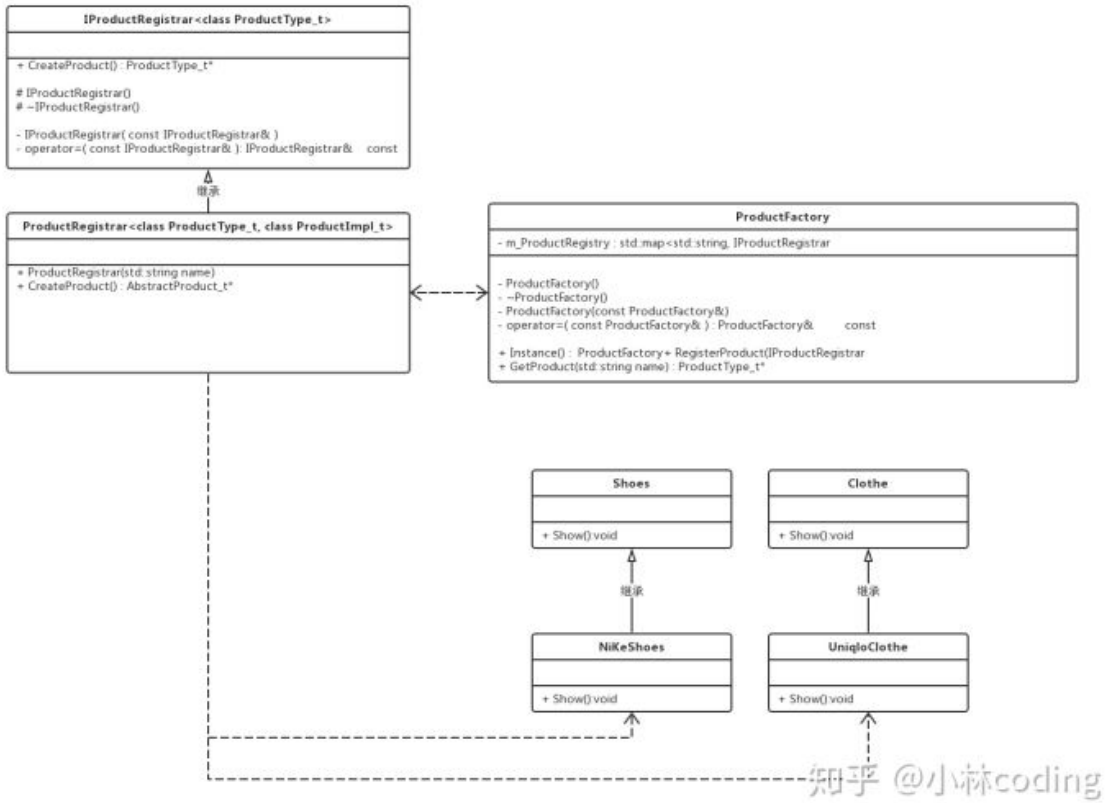
前面的模板工厂虽然在新增产品的时候，不需要新增具体的工厂类，但是缺少一个可以统一随时随地获取指定的产品对象的类。

还有改进的空间，我们可以把产品注册的对象用**std::map**的方式保存，通过**key-valve**的方式可以轻松简单的获取对应的产品对象实例。

实现大致思路：

- 把产品注册的功能封装成产品注册模板类。注册的产品对象保存在工厂模板类的**std::map**，便于产品对象的获取。
- 把获取产品对象的功能封装成工厂模板类。为了能随时随地获取指定产品对象，则把工厂设计成单例模式

UML图：



具体代码参考网站https://zhuanlan.zhihu.com/p/83537599

单例模式

介绍

单例模式(Singleton Pattern，也称为单件模式)，使用最广泛的设计模式之一。这个设计模式主要目的是想在整个系统中只能出现类的一个实例，即一个类只有一个对象。并提供一个访问它的全局访问点，该实例被所有程序模块共享。

单例模式解决的痛点：节约资源，节省时间。从两个方面来看:

1. 由于频繁使用的对象，可以节省**创建对象**所花费的时间，这对于那些重量级的对象而言，是很重要的。
2. 因为不需要频繁创建对象，我们的GC压力也减轻了，而在GC中会有STW(stop the world)，从这一方面也节约了GC的时间

单例模式的缺点：简单的单例模式设计开发都比较简单，但是复杂的单例模式需要考虑线程安全等并发问题，引入了部分复杂度。

定义一个单例类：

1. 私有化它的构造函数，以防止外界创建单例类的对象；
2. 使用类的私有静态指针变量指向类的唯一实例；
3. 使用一个公有的静态方法获取该实例。

懒汉版（Lazy Singleton）

单例实例在第一次被使用时才进行初始化，这叫做延迟初始化。

```
// version 1.0
class Singleton
{
private:
    static Singleton* instance;
private:
    Singleton() {};
    ~Singleton() {};
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
public:
    static Singleton* getInstance()
        {
        if(instance == NULL)
            instance = new Singleton();
        return instance;
    }
};

// init static member
Singleton* Singleton::instance = NULL;
```

当程序结束的时候，指针不会被delete，会造成内存泄露，有两种解决方式：

1. 使用智能指针
2. 使用静态的嵌套类对象

对于第二种解决方法，代码如下

```
// version 1.1
class Singleton
{
private:
    static Singleton* instance;
private:
    Singleton() { };
    ~Singleton() { };
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
private:
    class Deletor {
    public:
        ~Deletor() {
            if(Singleton::instance != NULL)
                delete Singleton::instance;
        }
    };
    static Deletor deletor;// 在单例类内定义私有的专门用于释放的静态成员。
public:
    static Singleton* getInstance() {
        if(instance == NULL) {
            instance = new Singleton();
        }
        return instance;
    }
    // 线程安全版本 (Java)
    // 能在多线程中很好的工作，但是每次调用getInstance()方法时都需要进行同步，造成不必要的同步开销
    static synchronized Singleton* getInstance(){……}
};

// init static member
Singleton* Singleton::instance = NULL;
```

在程序运行结束时，系统会调用静态成员`deletor`的析构函数，该析构函数会删除单例的唯一实例。使用这种方法释放单例对象有以下特征：

- 在单例类内部定义专有的嵌套类。
- 在单例类内定义私有的专门用于释放的静态成员。
- 利用程序在结束时析构全局变量的特性，选择最终的释放时机。

缺点：

- 当需要【getInstance()方法调用时】才创建，看上去好像没什么问题，但是当有多个线程同时调用getInstance方法时，此时刚好对象没有初始化，两个线程同时通过了`instance == NULL`的校验，将会创建两个`LazySingleton`对象，必须加锁来使得`getInstance`是线程安全的。

饿汉版（Eager Singleton）

指单例实例在程序运行时被立即执行初始化

```
// version 1.3
class Singleton
```

```
{
private:
    static Singleton instance;
private:
    Singleton(); // 构造函数私有化，保证外部不能调用构造函数创建对象，创建对象的行为只能由这个类决定
    ~Singleton();
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
public:
    static Singleton& getInstance() { // 只能通过getInstance获取对象
        return instance;
    }
}

// initialize defaultly
Singleton Singleton::instance;
```

C++规定，non-local static 对象的初始化发生在main函数执行之前

由于在main函数之前初始化，所以没有线程安全的问题。但是潜在问题在于no-local static对象（函数外的static对象）在不同编译单元中的初始化顺序是未定义的。也即，static Singleton instance;和 static Singleton& getInstance()二者的初始化顺序不确定，如果在初始化完成之前调用 getInstance() 方法会返回一个未定义的实例。

缺点：

- 如果一直没用到getInstance，有点浪费资源

优点：

- 由ClassLoad保证线程安全

总结：

- Eager Singleton 虽然是线程安全的，但存在潜在问题；
- Lazy Singleton通常需要加锁来保证线程安全，但局部静态变量版本在C++11后是线程安全的；
- 局部静态变量版本（Meyers Singleton）最优雅。
- 优点：

  > 只创建了一个实例，节省内存开销
  >
  > 减少了系统的性能开销，创建对象回收对象对性能都有一定的影响
  >
  > 避免对资源的多重占用
  >
  > 在系统设置全局的访问点，优化和共享资源优化

- 缺点：

  > 不适用于变化的对象
  >
  > 单例模式中没有抽象层，扩展有困难
  >
  > 与单一原则冲突。一个类应该只实现一个逻辑，而不关心它是否单例，是不是单例应该由业务决定

- 应用场景：

  > 创建对象需要消耗的资源过多时

> 需要定义大量的静态常量和静态方法的环境，比如工具类【感觉是最常见应用场景】

观察者模式

介绍

观察者模式(Observer Pattern)： 定义对象间一种一对多的依赖关系，使得当每一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

观察者模式是一种**对象行为型模式**

动机

将一个系统设计成一系列相互协作的类有一个常见的副作用：需要维护相关对象之间的一致性。

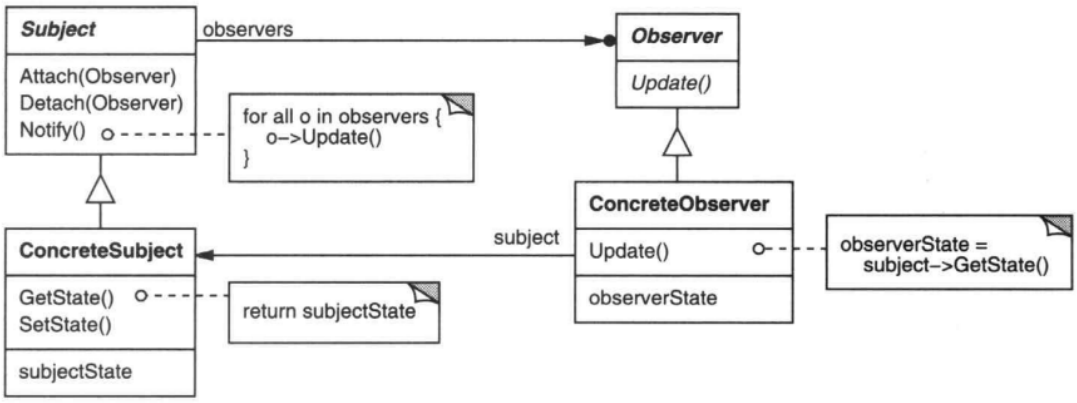
观察者模式定义一种交互，即发布-订阅：

- 一个对象当自身状态发生改变时，会发出通知，但是并不知道谁是他的接收者，但每个接收者都会接收到通知，这些接受者称为观察者。
- 作为对通知的响应，每个观察者都将查询目标状态，然后改变自身的状态以和目标状态进行同步。

使用场景

- 使对象封装为独立的改变和使用；
- 一个对象改变同时需要改变其它对象，而不知道具体有多少对象需要改变；
- 不希望对象是紧耦合的。

结构



Subject：目标，知道它的观察者，提供注册和删除观察者对象的接口

Observer：观察者，为那些在目标发生改变时需获得通知的对象定义一个更新接口

ConcreteSubject：具体目标，存储对象状态，状态改变时，向各个观察者发出通知

ConcreteObserver：具体观察者，维护一个指向ConcreteSubject对象的引用，存储有关状态，实现更新接口update，使自身状态与目标的状态保持一致

优缺点

1. 目标和观察者之间松耦合

2. 支持广播通信：Subject发送的通知不需要指定它的接受者。通知被自动广播给所有已向该目标对象登记的有关对象
3. 意外的更新：看似无害的操作可能会引起观察者错误的更新

例子

高数课，ABCD四位同学，A是好学生，去上课，B在寝室睡觉，C在网吧打游戏，D在学校外陪女友逛街。他们约定，如果要点名了，A在QQ群里吼一声，他们立刻赶到教室去。