第10章-设计模式

1. 概念

设计模式是由先人总结的一些经验规则，被我们反复使用后、被多数人知晓认可的、然后经过分类编排，形成了一套针对代码设计经验的总结，设计模式主要是为了解决某类重复出现的问题而出现的一套成功或有效的解决方案，设计模式的出现提高代码可复用性、扩展性、可维护性、灵活性、稳健性以及安全可靠性的解决方案。
设计模式一般分为3大类**创建型模式、结构型模式、行为型模式**，共计23种。
设计模式的本质是面向对象设计原则的实际运用，是对类的封装性、继承性和多态性以及类的纵、横关系的充分理解。
值得注意的是：设计模式只是一个程序设计指导思想，在具体的程序设计开发中，必须根据设计的应用系统的特点和要求来适当的选择，而不是一味地追求设计模式，否则可能过犹不及。

2. 单例模式

单例模式（Singleton Pattern）：属于创建型模式。3个要点：

1. 当前类最多只能创建一个实例。
2. 它必须自己创建这个实例（而不是调用者创建）。
3. 它必须自己向整个系统提供全局访问点访问这个实例。

注意：构造私有化保证了对象不能在类外创建。

2.1 懒汉式

不会自动初始化对象，直到第一次调用获取实例接口时，才会创建对象执行构造，初始化对象空间。是一种“时间换空间”的做法。实例最后需要手动销毁。
实现一：

```
class CSingleton {
private:
    static CSingleton* m_pSin;  //存储创建唯一的实例的地址
    CSingleton() {}
    CSingleton(const CSingleton&) = delete;    //被放弃使用
    ~CSingleton() {}
public:
    //多线程下会失效，可能创建多个实例，需要加锁来解决
    static CSingleton* CreateSingleton(){ //创建一个对象
        //多线程下：加锁
        if (!m_pSin) {
            m_pSin = new CSingleton;
```

```
        }
        //多线程下：解锁
        return m_pSin;
    }
    static void DestorySingleton(CSingleton *& p) {
        if (p) {
            delete p;
            p = nullptr;
            m_pSin = nullptr;
        }
    }
};
//类外初始化
CSingleton* CSingleton::m_pSin = nullptr;
```

对象的生命周期可控，在不需要的时候可以手动回收对象空间，节省资源，避免浪费。

实现二：

```
class CSingleton {
private:
    CSingleton(){}
    CSingleton(const CSingleton&) = delete;    //被放弃使用
    ~CSingleton() {}
public:
    //多线程下不会失效，编译器创建静态局部对象保证其原子性。
    static CSingleton* CreateSingleton() {
        static CSingleton sin;
        return &sin;
    }
};
```

第一次调用公共接口时创建唯一的实例，且不需要手动回收，直到程序退出时资源才被系统回收。

2.2 饿汉式

在程序创建之初自动的创建了对象，他是一个全局的对象，生命周期直到程序结束，所以不需要提供销毁对象的方法。是一种“空间换时间”的做法。不存在多线程下的安全问题。

```
class CSingleton {
private:
    static CSingleton sin;
private:
```

```
    CSingleton() :m_a(10) {}
    CSingleton(const CSingleton&) = delete;    //被放弃使用
    ~CSingleton() {}
public:
    //不会有多线程失效的问题
    static CSingleton* CreateSingleton() {
        return &sin;
    }
};
CSingleton CSingleton::sin;
```

单例模式优点:

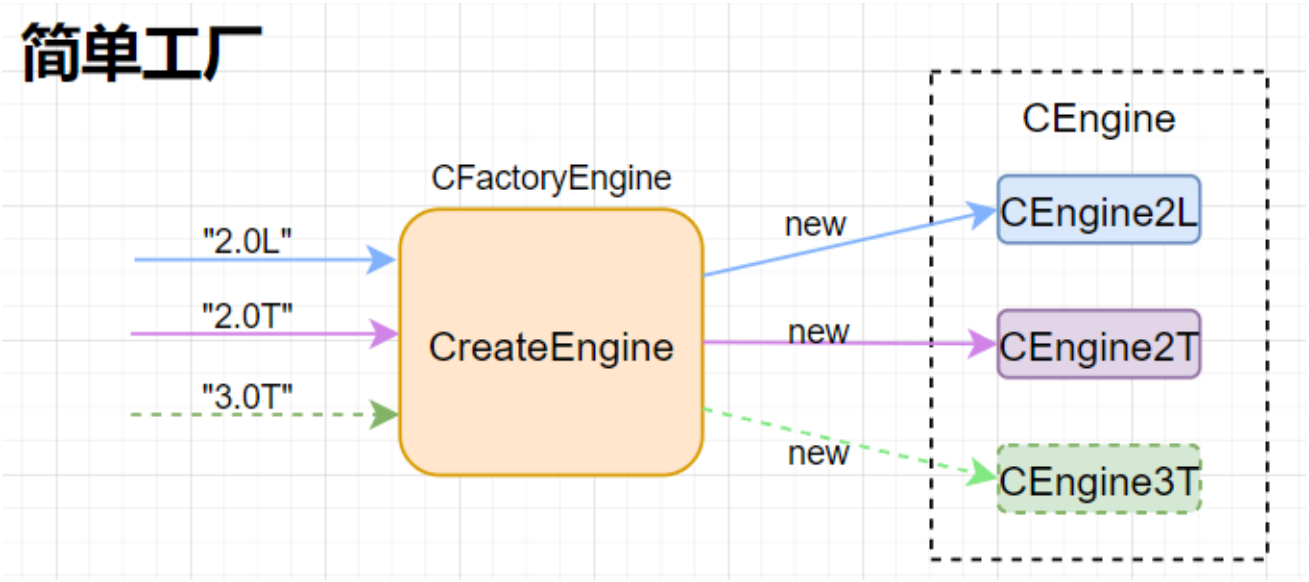
1. 单例模式提供了严格的对唯一实例的创建、访问和销毁，安全性高。
2. 单例模式的实现可以节省系统资源。

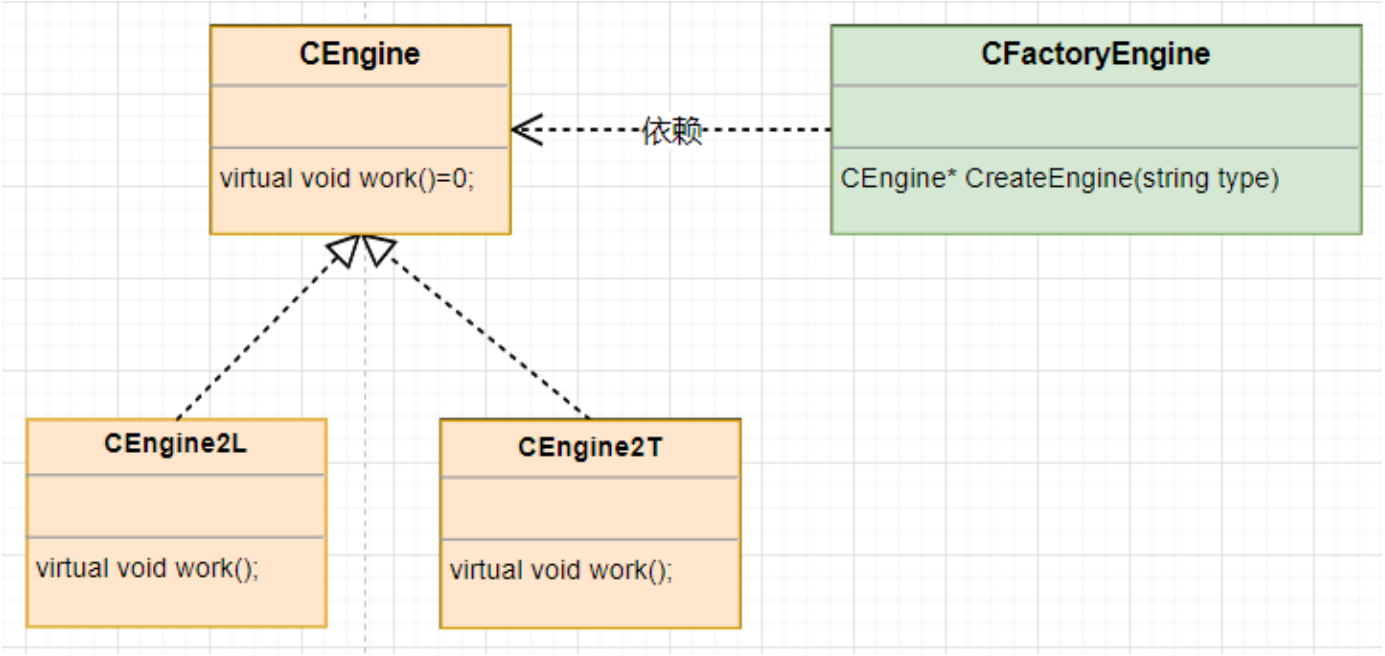
3. 工厂模式

工厂模式（Factory Pattern）：主要用来集中创建对象的，如果在任何使用的地方创建对象那就造成了类或方法之间的耦合，如果要更换对象那么在所有使用到的地方都要修改一遍，不利于后期的维护，也违背了开闭设计原则，如果使用工厂来创建对象，那么就彻底解耦合了，如果要修改只需要修改工厂类即可。工厂模式最大的优势：解耦。

3.1 简单工厂

简单工厂提供了一个集中创建所有类型产品的方法，根据对应的标识创建对应的产品。
模型图





发动机的类：

```
class CEngine {
public:
    virtual void work() = 0; //纯虚函数
};
class CEngine2L:public CEngine {
public:
    virtual void work() {
        cout << "2.0 自然吸气发动机正在工作" << endl;
    }
};
class CEngine2T :public CEngine {
public:
    virtual void work() {
        cout << "2.0 涡轮增压发动机正在工作" << endl;
    }
};
```

创建发动机的工厂类：

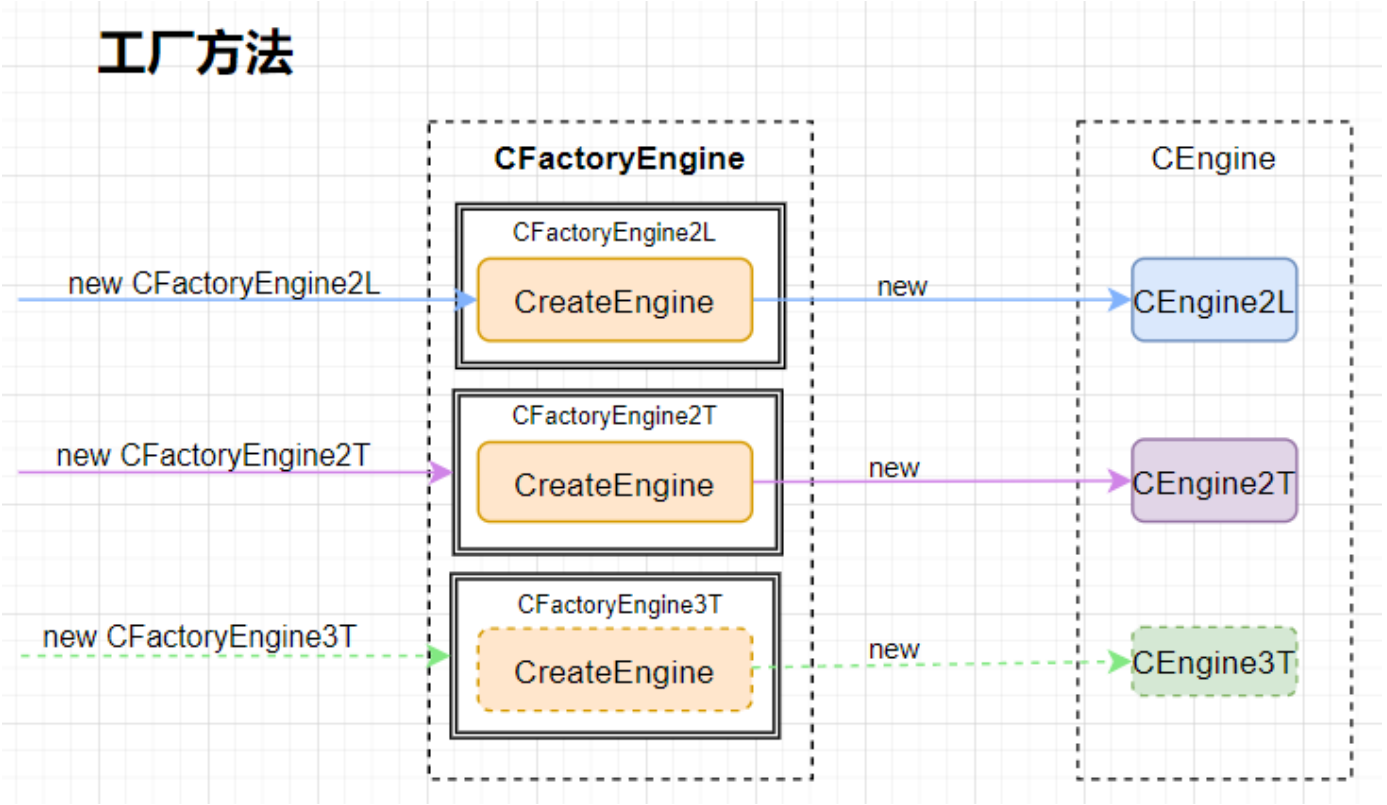
```
class CFactoryEngine {
public:
    CEngine* CreateEngine(string type) {  //创建所有类型的产品
        if (type == "2.0L")
            return  new CEngine2L;
        else if (type == "2.0T")
            return new CEngine2T;
        else
```

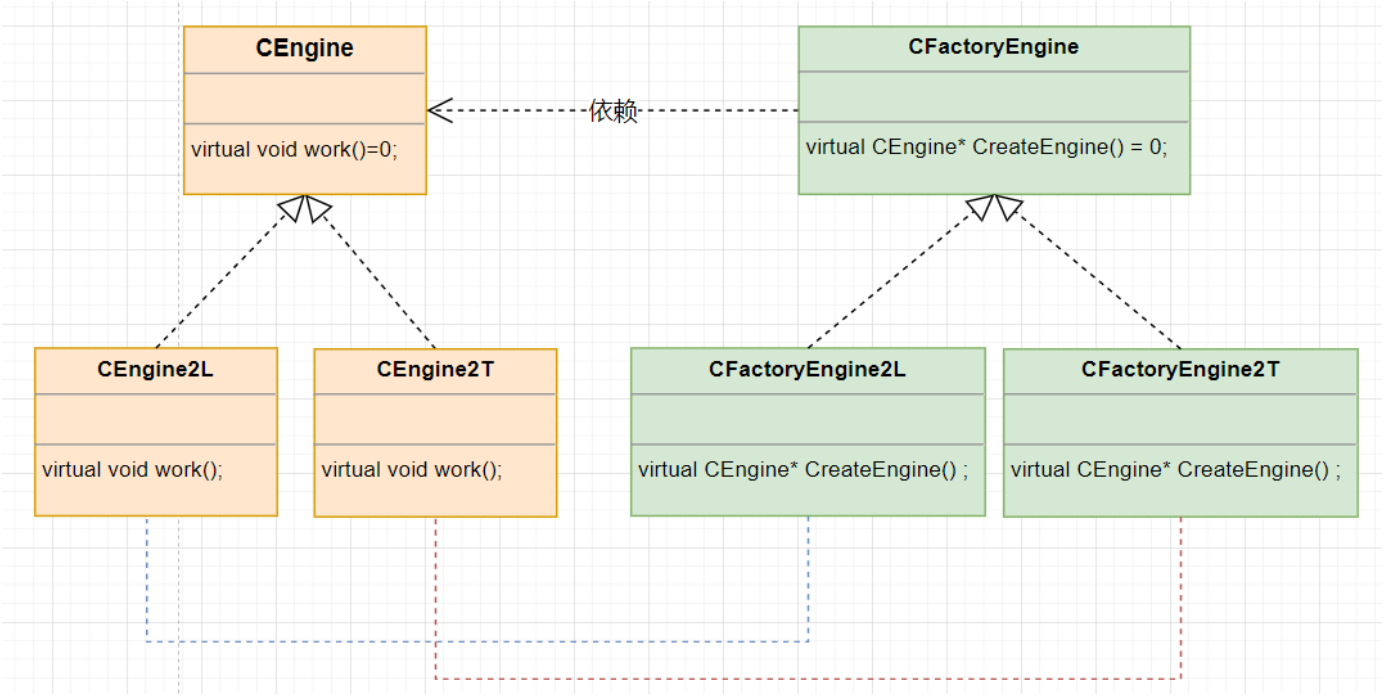
```
9                 return  nullptr;
10        }
11    };
```

上例中 `CreateEngine` 方法 包含了所有类型发动机（`CEngine2L`，`CEngine2T`）的创建，的当增加新的产品种类时，除了要增加对应的类外，还要修改工厂的集中创建方法，违背了开闭原则，所以适合创建**种类比较少**且**比较固定**的对象，对于复杂的业务环境就不太适应了。

3.2 工厂方法

由于简单工厂存在着弊端（违背开闭原则），所以在其基础上，进一步将工厂类进行抽象，拆分多个类型的工厂，每个工厂创建对应类型的类对象。
模型图：





将工厂抽象（抽象类），增加新的具体工厂与发动机种类一一对应。

```
class CFactoryEngine {
public:
    virtual CEngine* CreateEngine() = 0;//抽象接口
};

class CFactoryEngine2L :public CFactoryEngine {
public:
    virtual CEngine* CreateEngine() {
        return  new CEngine2L;
    }
};
class CFactoryEngine2T :public CFactoryEngine {
public:
    virtual CEngine* CreateEngine() {
        return  new CEngine2T;
    }
};
```

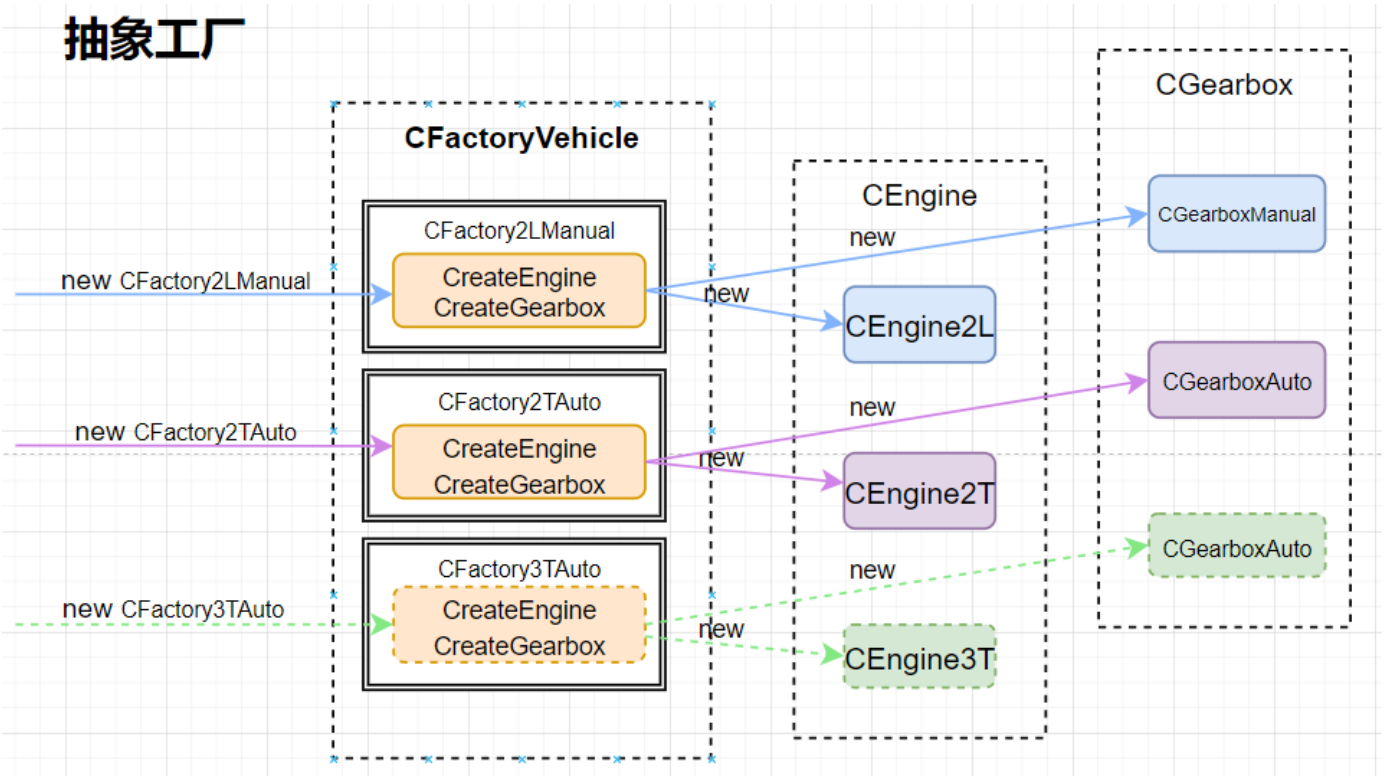
工厂方法模式每个子类工厂对应一个具体类型的对象，比如`CFactoryEngine2L` 对应`CEngine2L` 对象， 遵循了开闭原则，提高了扩展性（如果增加了新的类型的发动机那么原有的发动机工厂不用改动），如果对象种类较多，那么一个都要对应一个工厂，需要大量的工厂得不偿失。

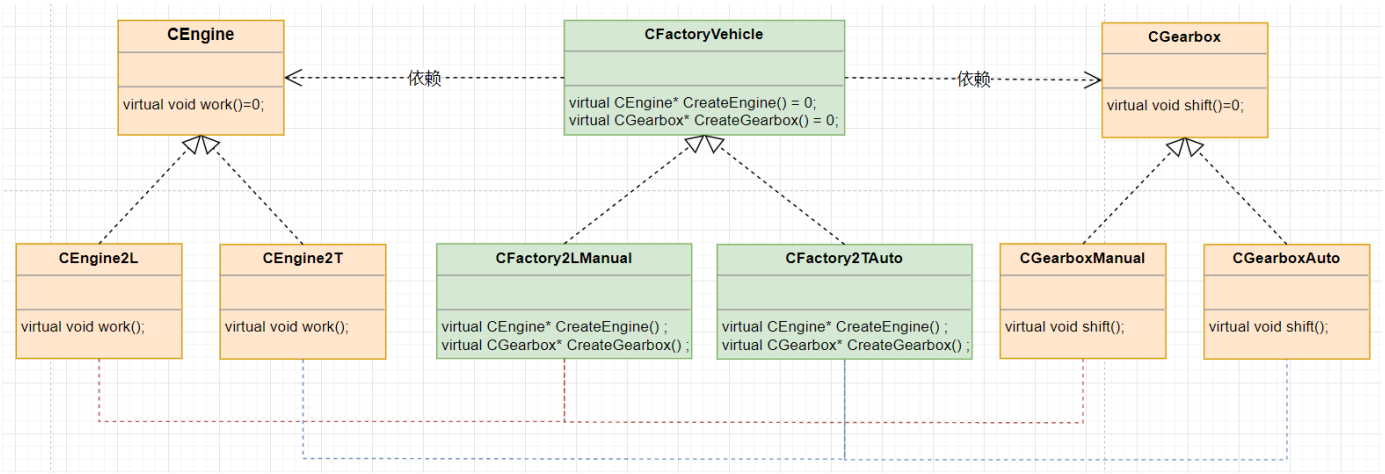
3.3 抽象工厂

抽象工厂（Abstract Factory Pattern）和工厂方法的模式基本一样，区别在于，工厂方法是生产一个具体的产品，而抽象工厂可以用来生产一组相同，有相对关系的产品，重点在于一组，一批，一系

列。

模型图：





一系列产品，如：发动机和变速箱：

```
class CEngine {
public:
    virtual void work() = 0; //纯虚函数
};

class CEngine2L :public CEngine {
public:
    virtual void work() {
        cout << "2.0 自然吸气发动机正在工作" << endl;
    }
};

```

```
class CEngine2T :public CEngine {
public:
    virtual void work() {
        cout << "2.0 涡轮增压发动机正在工作" << endl;
    }
};

class CGearbox {
public:
    virtual void work() = 0; //纯虚函数
};
class CGearboxAuto :public CGearbox {
public:
    virtual void work() {
        cout << "自动变速箱正在工作" << endl;
    }
};
class CGearboxManual :public CGearbox {
public:
    virtual void work() {
        cout << "手动变速箱正在工作" << endl;
    }
};
```

抽象工厂类:

```
class CFactory {
public:
    //增加了创建多种产品的方法，支持创建发动机和变速箱
    virtual CEngine* CreateEngine() = 0;
    virtual CGearbox* CreateGearbox() = 0;
};

class CFactory2LManual:public CFactory {
public:
    virtual CEngine* CreateEngine() {
        return  new CEngine2L;
    }
    virtual CGearbox* CreateGearbox() {
        return new CGearboxManual;
    }
};

```

```
class CFactory2TAuto :public CFactory {
public:
    virtual CEngine* CreateEngine() {
        return  new CEngine2T;
    }
    virtual CGearbox* CreateGearbox() {
        return new CGearboxAuto;
    }
};
```

对于具体工厂类可以根据需求自由组合产品，如需求增加 `CEngine2L` 和 `CGearboxAuto` 的组合，则新建工厂类`CFactory2LAuto`即可。

三种工厂方式总结：

1、对于简单工厂和工厂方法来说，两者的使用方式实际上是一样的，如果对于产品的分类和名称是确定的，数量是相对固定的，推荐使用简单工厂模式。

2、抽象工厂用来解决相对复杂的问题，适用于一系列、大批量的对象生产。