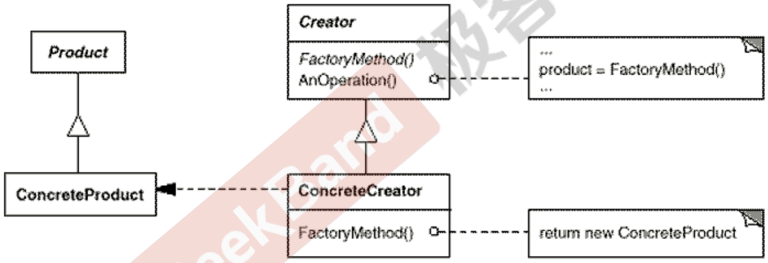
定义:定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使得一个类的实例化延迟（目的：解耦，手段：虚函数）到子类。

——《设计模式》GoF

动机: 在软件系统中，经常面临着创建对象的工作；由于需求的变化，需要创建的对象的具体类型经常变化。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“具体对象创建工作”的紧耦合?

核心思想：将工厂抽象出来，将产品抽象出来，子工厂负责new 子产品指针，返回抽象产品类指针，外面调用时只需要生成基类工厂指针，调用创建产品函数，就可以对该产品进行具体的操作，优点是能够将添加产品完全独立出来不再修改内部代码。



意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决：主要解决接口选择的问题。

何时使用：明确地计划不同条件下创建不同实例时。

如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码：创建过程在其子类执行。

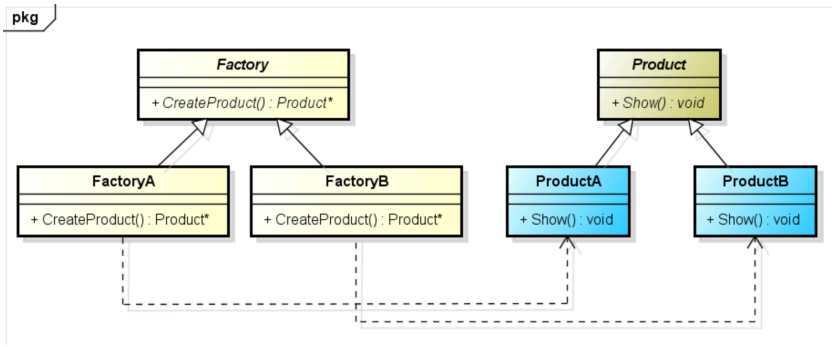
优点： 1、一个调用者想创建一个对象，只要知道其名称就可以了。 2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。 3、屏蔽产品的具体实现，调用者只关心产品的接口。

缺点：每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

使用场景： 1、日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。 2、数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。 3、设计一个连接服务器的框架，需要三个协议，"POP3"、"IMAP"、"HTTP"，可以把这三个作为产品类，共同实现一个接口。

注意事项：作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

其实这才是正宗的工厂模式，简单工厂模式只是一个简单的对创建过程封装。工厂方法模式在简单工厂模式的基础上增加对工厂的基类抽象，不同的产品创建采用不同的工厂创建（从工厂的抽象基类派生），这样创建不同的产品过程就由不同的工厂分工解决：FactoryA专心负责生产ProductA，FactoryB专心负责生产ProductB，FactoryA和FactoryB之间没有关系；如果到了后期，如果需要生产ProductC时，我们则可以创建一个FactoryC工厂类，该类专心负责生产ProductC类产品。由于FactoryA、FactoryB和FactoryC之间没有关系，当加入FactoryC加入时，对FactoryA和FactoryB的工作没有产生任何影响，那么对代码进行测试时，只需要单独对FactoryC和ProductC进行单元测试，而FactoryA和FactoryB则不用进行测试，则可省去大量无趣无味的测试工作。



- Factory（抽象工厂）：是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象的工厂类必须实现这个接口。
- ConcreteFactory（具体工厂）：实现抽象工厂接口的具体工厂类，包含与应用程序密切相关的逻辑，并且被应用程序调用以创建产品对象。
- Product（抽象产品）：所创建对象的基类，也就是具体产品的共同父类或共同拥有的接口。
- ConcreteProduct（具体产品）：实现了抽象产品角色所定义的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应。

优缺点

优点:

- 克服了简单工厂模式违背开放-封闭原则的缺点，又保留了封装对象创建过程的优点，降低客户端和工厂的耦合性。所以说，“工厂方法模式”是“简单工厂模式”的进一步抽象和推广。

缺点:

- 每增加一个产品，相应的也要增加一个子工厂，加大了额外的开发量。

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

class ICar
{
public:
    virtual string Name() = 0;
    virtual ~ICar() {}
};
class Benze : public ICar
{
public:
    string Name()
    {
        return "Benze Car";
    }
};
class Bmw : public ICar
{
public:
    string Name()
    {
        return "Bmw Car";
    }
};
```

```
class Audio : public ICar
{
public:
    string Name()
    {
        return "Audio Car";
    }
};
class IFactory
{
public:
    virtual ICar *CreateCar() = 0;
    virtual ~IFactory() {}
};
class BenzeFactory : public IFactory
{
public:
    ICar *CreateCar()
    {
        return new Benze();
    }
};
class BmwFactory : public IFactory
{
public:
    ICar *CreateCar()
    {
        return new Bmw();
    }
};
class AudioFactory : public IFactory
{
public:
    ICar *CreateCar()
    {
        return new Audio();
    }
};
int main()
{
    IFactory *pF = new BenzeFactory();
    ICar *pCar = pF->CreateCar();
    cout << "Benze Factory create: " << pCar->Name() << endl;
    delete pF;
    delete pCar;
    pF = new BmwFactory();
    pCar = pF->CreateCar();
    cout << "Bmw Factory create: " << pCar->Name() << endl;
    delete pF;
    delete pCar;
    pF = new AudioFactory();
    pCar = pF->CreateCar();
    cout << "Audio Factory create: " << pCar->Name() << endl;
    return 0;
```

```
81 }
```

```
192:DesignPattnsStudy weishichun$ ls FactoryMethod
FactoryMethod_1.cpp        FactoryMethod工厂方法.pdf
192:DesignPattnsStudy weishichun$ g++ -o FactoryMethod.out FactoryMethod_1.cpp
192:DesignPattnsStudy weishichun$ ./FactoryMethod.out
Benze Factory create: Benze Car
Bmw Factory create: Bmw Car
Audio Factory create: Audio Car
192:DesignPattnsStudy weishichun$
```