iOS三种工厂设计模式

前言:

- 三种工厂设计模式指的是:简单工厂模式、工厂方法模式、抽象工厂模式都是常见创建型模式的一种
- 创建型模式:处理在不指定对象具体类型的情况下创建对象的问题

工厂方法模式:

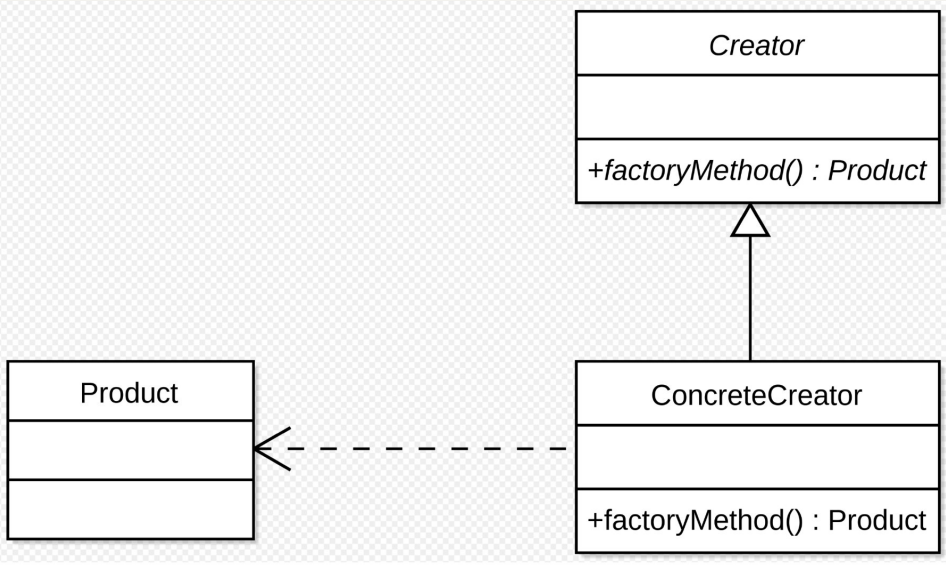
定义:

- 定义一个创建对象的接口，但让实现这个接口的类来决定实例化哪个类。工厂方法让类的实例化推迟到子类中进行

背景:

- 创建一个对象常常需要复杂的过程，所以不适合包含在一个复用对象中。创建对象可能会导致大量的重复代码，可能会需要复用对象访问不到的信息，也可能提供不了足够级别的抽象，还可能并不是复用对象概念的一部分。工厂方法模式通过定义一个单独的创建对象的方法来解决这些问题。由子类实现这个方法来创建具体类型的对象

UML:



图中角色如下:

- 抽象工厂角色:工厂通常是一个用来创建其他对象的对象。工厂是构造方法的抽象，用来实现不同的分配方案。
- 具体工厂角色:实现产品的具体对象,是抽象工厂的子类
- 产品角色:由具体对象实现的实例

例子:

有一个 `Button` 类表示按钮，另有它的两个子类 `WinButton` 和 `MacButton` 分别代表Windows和Mac风格的按钮，那么这几个类和用于创建它们的工厂类在iOS中可以如下实现:

```swift
//swift的几个Button(产品)
protocol Button {}
struct WinButton:Button {}
struct MacButton:Button {}

//他们的工厂类
protocol ButtonFactory {
    static func createButton()->Button;
}
struct WinButtonFactory: ButtonFactory {
    static func createButton() -> Button {
        return WinButton.init()
    }
}
struct MacButtonFactory:ButtonFactory {
    static func createButton() -> Button {
        return MacButton.init()
    }
}

//客户端调用
let winBtn =
WinButtonFactory.createButton();//winBtn的创建
let macBtn =
MacButtonFactory.createButton();//macBtn创建
```

```objectivec
//objective-c的几个Button(产品)
Button.h
@interface Button : NSObject
@end
Button.m
@implementation Button
@end
```

```
8
9  WinButton.h
10 @interface WinButton : Button
11 @end
12 WinButton.m
13 @implementation WinButton
14 @end
15
16 MacButton.h
17 @interface MacButton : Button
18 @end
19 MacButton.m
20 @implementation MacButton
21 @end
22
23 //他们的工厂类
24 ButtonFactory.h
25 @interface ButtonFactory : NSObject
26 + (Button *)createButton;
27 @end
28 ButtonFactory.m
29 @implementation ButtonFactory
30 + (Button *)createButton{
31     return [Button new];
32 }
33 @end
34
35 WinButtonFactory.h
36 @interface WinButtonFactory : ButtonFactory
37 @end
38 WinButtonFactory.m
39 @implementation WinButtonFactory
40 + (Button *)createButton{
41     return [WinButton new];
42 }
43 @end
44
45 MacButtonFactory.h
46 @interface MacButtonFactory : ButtonFactory
47 @end
48 MacButtonFactory.m
49 @implementation MacButtonFactory
50 + (Button *)createButton{
51     return [WinButton new];
52 }
53 @end
54
```

```
//客户端调用
Button *winBtn = [WinButtonFactory createButton];//winBtn的创建
Button *macBtn = [MacButtonFactory createButton];//macBtn创建
```

变种:

虽然工厂方法模式的背后动机是允许子类选择创建对象的具体类型，但是使用工厂方法模式也有一些其他的好处，其中很多并不依赖于子类。因此，有时候也会创建不使用多态性创建对象的工厂方法，以得到使用工厂方法的其他好处。

工厂"方法"而非工厂"类":

如果抛开设计模式的范畴，“工厂方法”这个词也可以指作为“工厂”的方法，这个方法的主要目的就是创建对象，而这个方法不一定在单独的工厂类中。这些方法通常作为静态方法，定义在方法所实例化的类中。

例子:

```swift
//swift
struct Button {
    var product:String

    static func createWinButton()->Button{
        return Button(productName: "WinButton")
    };
    static func createMacButton()->Button{
        return Button(productName: "MacButton")
    };
    private init(productName:String) {
        product = productName
    }
}

//客户端调用
let winButton = Button.createWinButton()//winBtn的创建
let macBtn = Button.createMacButton()//macBtn的创建
```

```objective-c
//objective-c
Button.h
@interface Button : NSObject
@property (nonatomic,copy)NSString *product;
```

```
+ (Button *)creatWinButton;
+ (Button *)creatMacButton;
@end
Button.m
@implementation Button
+ (Button *)creatWinButton{
    return [[Button alloc] initWithProductName:@"WinButton"];
}
+ (Button *)creatMacButton{
    return [[Button alloc] initWithProductName:@"MacButton"];
}
-(instancetype)initWithProductName:(NSString *)product{
    self = [super init];
    if (self) {
        self.product = product;
    }
    return self;
}
@end

//客户端调用
Button *winBtn = [Button creatWinButton];//winBtn的创建
Button *macBtn = [Button creatMacButton];//macBtn的创建
```

apple也有类似的设计:

```
//apple NSnumber

//客户端调用
NSNumber *aChar =[NSNumber

initWithBool:YES];//__NSCFBoolean
NSNumber *anInt = [NSNumber numberWithInt:1];//__NSCFNumber
NSNumber *aFloat = [NSNumber numberWithFloat:1.0];
NSNumber *aDouble = [NSNumber numberWithDouble:1.0];
```

返回的结果是类簇,也算是抽象工厂模式的一种应用

简单工厂:

普通的工厂方法模式通常伴随着对象的具体类型与工厂具体类型的一一对应，客户端代码根据需要选择合适的具体类型工厂使用。然而，这种选择可能包含复杂的逻辑。这时，可以创建一个单一的工厂类，用以包含这种选择逻辑，根据参数的不同选择实现不同的具体对象。这个工厂类不需要由每个具体产品实现一个自己的具体的工厂类，所以可以将工厂方法设置为静态方法。

例子:

```swift
//swift
protocol Button {}
struct WinButton:Button {}
struct MacButton:Button {}

enum ButtonType {
    case Win
    case Mac
}
struct ButtonFactory {
    static func createButton(type:ButtonType)->Button{
        let button:Button
        switch type {
        case ButtonType.Win:
            button = WinButton()
        case ButtonType.Mac:
            button = MacButton()
        }
        return button
    }
}
let winBtn = ButtonFactory.createButton(type:ButtonType.Win)//winBtn的创建
let macBtn = ButtonFactory.createButton(type:ButtonType.Mac)//macBtn的创建
```

```
//objective-c的几个Button(产品)
Button.h
@interface Button : NSObject
@end
Button.m
@implementation Button
@end

WinButton.h
```

```
@interface WinButton : Button
@end
WinButton.m
@implementation WinButton
@end

MacButton.h
@interface MacButton : Button
@end
MacButton.m
@implementation MacButton
@end

//他们的工厂
ButtonFactory.h
typedef enum {
    WinButtonType,
    MacButtonType
}ButtonType;
@interface ButtonFactory : NSObject
+ (Button *)createButtonWithType:(ButtonType)type;
@end
ButtonFactory.m
@implementation ButtonFactory
+ (Button *)createButtonWithType:(ButtonType)type{
    Button *newBtn = nil;
    switch (type) {
        case WinButtonType:
            newBtn = [WinButton new];
            break;
        case MacButtonType:
            newBtn = [MacButton new];
        default:
            newBtn = [Button new];
            break;
    }
    return newBtn;
}
@end

//客户端调用:
Button *winBtn3 = [ButtonFactory

 createButtonWithType:WinButtonType];//winBtn的创建
Button *macBtn3 = [ButtonFactory

 createButtonWithType:MacButtonType];//macBtn的创建
```

适用性:

工厂方法，适用于面向接口编程（programming to interface）与实现依赖反转原则。 下列情况可以考虑使用工厂方法模式：

- 创建对象需要大量重复的代码。可以把这些代码写在工厂基类中。
- 创建对象需要访问某些信息，而这些信息不应该包含在复合类中。
- 创建对象的生命周期必须集中管理，以保证在整个程序中具有一致的行为。 对象创建时会有很多参数来决定如何创建出这个对象。
- 业务对象的代码作者希望隐藏对象的真实类型，而构造函数一定要真实的类名才能用等等

局限性:

- 重构已经存在的类会破坏客户端代码
- 因为工厂方法所实例化的类具有私有的构造方法，所以这些类就不能扩展了。因为任何子类都必须调用父类的构造方法，但父类的私有构造方法是不能被子类调用的
- 如果确实扩展了工厂方法所实例化的类（例如将构造方法设为保护的，虽然有风险但也是可行的），子类必须具有所有工厂方法的一套实现。否则将会返回一个父类的实例，而不是所希望的子类实例。

参考:

维基百科-工厂方法