**1.何为生成器模式**（需要规范生产的时候使用）

将一个复杂对象的构建与它的表现分离，使得同样的构建过程可以创建不同的表现

【1】案例一

上面的表述可能有些抽象，我们用生活中的例子为大家说明下：

有时候我们虽然在同一家外卖店点的相同的商品，有可能体验会相差很大。

第一次，卧槽，这家店的葱爆牛肉炒饭好好吃啊，老子下次还要点！！  
第二次，卧槽，我点的葱爆牛肉炒饭怎么没牛肉，牛肉哪里去了！！  
第三次，老子不信邪，再点一次葱爆牛肉炒饭看看，卧槽，老板忘记放盐了！！Boom.........

为什么同一家同一款商品而且还是同一个师傅做的，有时候差异性会那么大呢？这时我们再想想，你平时点的肯德基(这次不用金拱门做例子了，到时候你们还以为我收了广告费/(ㄒoㄒ)/~~)外卖会出现口味差异那么大的情况吗？同学们可能会说，别人是专业的啊！有一套整体规范的制作流程。没错，对于同样一款商品在肯德基中的制作流程可能细节到了使用多少克配料，油炸的时长具体是几分钟，这些都是有严格的把控的。通过这个简单的例子应该能帮助我们对生成器模式中的对象构建和表现分离这一思想有所启发。

【2】案例2

拿魔兽世界来说（作者没玩过，哪里说得不对请多多见谅，能明白意思就好）

里面的角色分成许多种族：人族、兽族  
虽然种族不一样但是每个种族都会有一些相同的职业：战士、术士  
然而因为种族不同，每个职业当中的一些加成也不一样。  
假如一个角色的有如下间接属性：力量（strength）、耐力（Stamina）、智力（Intelligence）、敏捷（Agility）。  
直接属性就两个：攻击力（Aggressiveness）、防御力（Protection）

我们认为：  
力量越高 攻击 和 防御 越高。  
耐力越高 攻击欲望就下降了，所以攻击下降， 防御变高。  
智力越高越懂得命中要害，但是会让自己更加危险，所以攻击变高，防御下降。  
敏捷越高越会躲避敌人攻击，但是攻击的时机变少给了，所以攻击下降，防御变高

由于种族和职业导致每种间接属性对直接属性的加成有所不同，列表如下：

* 种族 影响着初始值

人族：对于力量和耐力加成不高，智力和敏捷加成很高

|  | **属性** |
| --- | --- |
| 力量（Strength） | 20 |
| 耐力（Stamina） | 20 |
| 智力（Intelligence） | 30 |
| 敏捷（Agility） | 30 |

兽族：力量和耐力加成高，智力和敏捷加成低

|  | **属性** |
| --- | --- |
| 力量（Strength） | 30 |
| 耐力（Stamina） | 30 |
| 智力（Intelligence） | 20 |
| 敏捷（Agility） | 20 |

* 职业 影响着副属性对主属性的加成

战士：力量、耐力 和 敏捷的攻击加成增高

|  | **防御** | **攻击** |
| --- | --- | --- |
| 力量（Strength） | +2 | +2 |
| 耐力（Stamina） | +2 | -1 |
| 智力（Intelligence） | - 2 | +2 |
| 敏捷（Agility） | +1 | -1 |

术士：智力增加攻击力和防御力

|  | **防御** | **攻击** |
| --- | --- | --- |
| 力量（Strength） | +1 | +1 |
| 耐力（Stamina） | +2 | -1 |
| 智力（Intelligence） | + 5 | +5 |
| 敏捷（Agility） | +2 | -1 |

从上面的表格我们举一个例子：  
兽人战士每增加一点力量，会增加3点防御 和 2点攻击  
人族法师每增加一点智力，会增加3点防御 和 5点攻击 （尼玛 我要是策划我估计我能被打死）

1.你根本就不知道用户会创建什么种族 什么职业，所以只能动态加载。  
2.这里只有4种情况，但实际当中可能有十几个职业和属各种族，所以根本没办法一一举例出来，而且每增加一个种族这个枚举就会成倍的增加。  
的·这里生成器模式就用上了。

我们这里先从代码开始讲，然后再开始分析生成器模式。

作者：sqatm  
链接：https://www.jianshu.com/p/92e13802980f  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

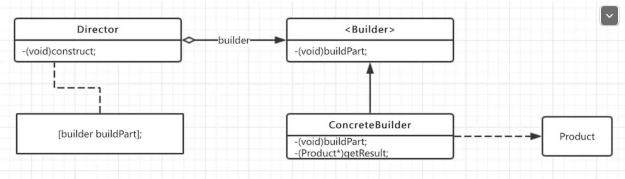
**2.何时使用生成器模式**

* 需要创建涉及各种部件的复杂对象。创建对象的算法应该独立于部分的装配方式，常见例子是构建组合对象。
* 构建过程是稳定可抽象的，而构建对象所需的部件是可变动的

**3.生成器模式静态类结构图**

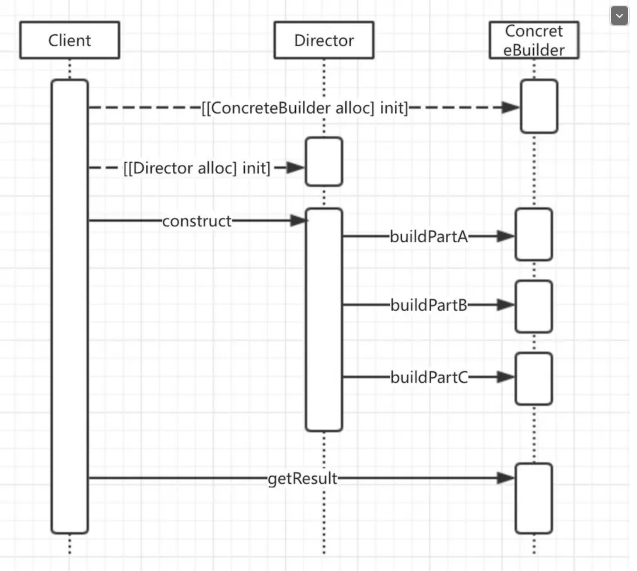
生成器模式由Director(指导者)、Builder(生成器)、Product(产品)三者构成。

**Director(汉堡制作流程)：**抽象对象构建流程  
**Builder(汉堡制作师傅)：**抽象对象构建部件方法  
**Product(汉堡)：**实际产品类

生成器模式由Director(指导者)、Builder(生成器)、Product(产品)三者构成。

**Director(汉堡制作流程)：**抽象对象构建流程  
**Builder(汉堡制作师傅)：**抽象对象构建部件方法  
**Product(汉堡)：**实际产品类

**4.生成器模式时序图**

这里客户端首先知悉并生成Director和ConcreteBuilder，将它们用于今后的协同工作。当Client发送construct消息给Director时，该方法告诉Director要构建什么。Director执行内部构建流程，这里Director和ConcreteBuilder是一个聚合关系。构建成功后客户端通过getResult方法从ConcreteBuilder中直接取回产品。

这里可能有人要问，为什么要让Client知悉Director，直接用construct方法返回构建的产品即可。没错，如果消除ConcreteBuilder那么跟普通的工厂模式十分相似了。这里也就是生成器模式和工厂模式的区分点，前面也有提到过该模式的核心点就是对可复用的构建流程和构建组件进行抽象化，这样便于以后模块的复用。同样的构建流程，替换不同的ConcreteBuilder即可生成其他相似的产品，也保障了同类型性产品制作的准确性。

1. **实际案例**

代码以上述肯德基生产汉堡为基准：

师傅**WCQDirector** 、机器**WCQConcreteBuilder**（程序**WCQBuilder**）、汉堡**WCQHamburger**

**WCQDirector.h**

#import <Foundation/Foundation.h>

#import "WCQBuilder.h"

@interface WCQDirector : NSObject

@property (nonatomic, strong) id<WCQBuilder> concreteBuilder;  //指定一个懂程序的师傅

- (void)construct;  // 师傅要得到面包时需要设备下的命令

@end

**WCQDirector.m**

#import "WCQDirector.h"

@implementation WCQDirector

- (void)construct {

    [\_concreteBuilder addBread];

    [\_concreteBuilder addSauce];

    [\_concreteBuilder addMeat];

}

@end

**WCQBuilder.h  // 程序协议：**程序工作时候，会提示制作汉堡需要添加什么东西

#import <Foundation/Foundation.h>

@protocol WCQBuilder <NSObject>

- (void)addBread; //添加面包

- (void)addSauce; //添加酱料

- (void)addMeat;  //添加肉饼

@end

**WCQConcreteBuilder.h**

#import <Foundation/Foundation.h>

#import "WCQBuilder.h"

@interface WCQConcreteBuilder : NSObject<WCQBuilder> // 生产汉堡的机器需要遵守程序指令，这样才能得到面包

- (id)getHamburger；// 遵守了程序，就会得到面包

@end

**WCQConcreteBuilder.m //** 生产汉堡的机器去完成制作汉堡的工作（汉堡不同，就是这边添加的原料不同）

#import "WCQConcreteBuilder.h"

#import "WCQHamburger.h"

@interface WCQConcreteBuilder ()

@property (nonatomic, strong) WCQHamburger \*hamburger;

@end

@implementation WCQConcreteBuilder

- (instancetype)init {

    if (self = [super init]) {

        \_hamburger = [[WCQHamburger alloc] init];

    }

    return self;

}

-(id)getHamburger {

    return \_hamburger;

}

#pmark - mark  WCQBuilder

- (void)addBread {

    \_hamburger.bread = @"小麦面包";

}

- (void)addSauce {

    \_hamburger.sauce = @"芝士酱";

}

- (void)addMeat {

    \_hamburger.meat = @"牛肉饼";

}

@end

**WCQHamburger.h //** 汉堡需要的原料

#import <Foundation/Foundation.h>

@interface WCQHamburger : NSObject

@property (nonatomic, copy) NSString \*bread;

@property (nonatomic, copy) NSString \*sauce;

@property (nonatomic, copy) NSString \*meat;

@end

客户端调用示例：

- (void)viewDidLoad {

    [super viewDidLoad]

    // Do any additional setup after loading the view, typically from a nib.

// 指定一台机器A

    WCQConcreteBuilder \*concreteBuilder = [[WCQConcreteBuilder alloc] init];

// 指定一个师傅A

    WCQDirector \*director = [[WCQDirector alloc] init];

// 默认给到师傅，师傅就会直接做的话，可以把这个逻辑抽到实现内部去

// 把机器A给师傅A

    director.concreteBuilder = concreteBuilder;

// 师傅开始下命令，机器就会提示，师傅按提示加入原料（或者事先已经准备好，机器自行添加即可），机器就会自己开始制作过程

    [director construct];

 // 指定一个汉堡，这个汉堡就是机器A生产出来的汉堡

    WCQHamburger \*hamburger = [concreteBuilder getHamburger];

}

从上述代码示例可以看到，制作汉堡的流程已抽象到Director中，每次生成汉堡都将执行该流程 [director construct];，这就避免了制作同种汉堡时可能造成的口味偏差问题(少放酱料等问题)。同时如果需要制作其他口味的汉堡只需更改汉堡制作程序(**ConcreteBuilder**)即可。