**设计模式笔记**

1. **设计模式入门**

OO基础:抽象、封装、多态、继承

OO原则:

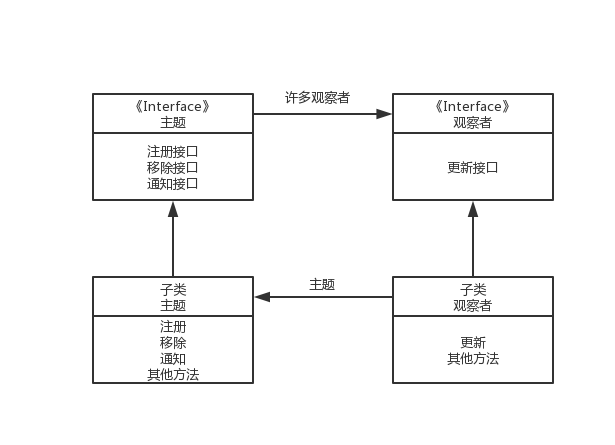
1. 封装变化
2. 多用组合,少用继承
3. 针对接口编程而非实现

OO模式之策略模式：

定义算法族,分别封装起来,让其之间可以相互替换,次模式让算法的变化独立于使用算法的客户

1. **观察者模式**

**性质:**观察者模式的本质就是出版者+订阅者。或者说,主题(出版者)和观察者(订阅者)。

**OO模式之观察者模式：**

定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖都会收到通知且自动更新。

**观察者模式的类图：**

**松耦合：**

当两个对象之间松耦合，他们依然可以交互，但是不太清楚彼此的细节

**设计原则：**

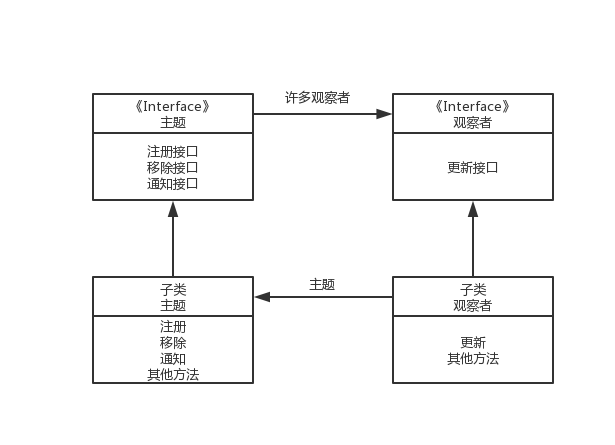
为了交互对象间的松耦合而努力

松耦合的设计之所以能够建立富有弹性的OO系统以应对变化，是因为对象之间的依赖降到了最低

1. **装饰者模式**

**设计原则:**

类应当对扩展开放，对修改关闭

**OO模式之装饰者模式：**

动态地将责任附加到对象上,若要扩展功能，装饰者提供比继承更有弹性的替代方案

**装饰者模式的类图:**

实际上装饰者模式利用了组合来替代继承

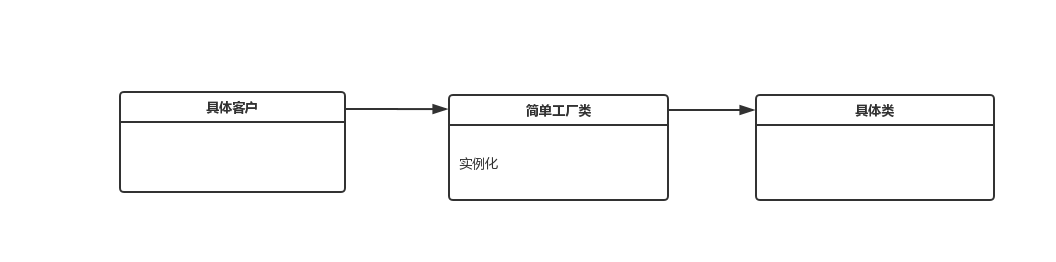
**OO原则(扩展)：**

1. 封装变化
2. 多用组合、少用继承
3. 针对接口编程而非实现
4. 为交互对象之间松耦合而努力
5. 对扩展开放、对修改关闭

**四、工厂模式**

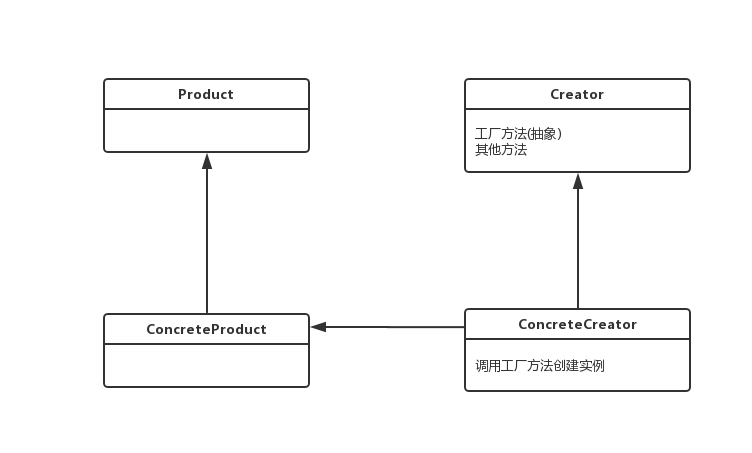
**1.简单工厂**

简单工厂并非一个具体的设计模式，反而更像一种编程习惯。



其本质是将实例化的过程用一个单独的类封装起来,这个类也被称作简单工厂类,如下图所示

**2.OO模式之工厂方法模式**

工厂方法模式定义了一个创建对象的接口,但由子类决定要实例化的类到底是哪一个,工厂方法让类把实例化推迟到子类

如上图所示,Creator是一个类,其实现了所有操纵产品的方法,但不实现工厂方法

ConcreteCreator实现了工厂方法,以创建实际的产品

Product是一个抽象产品类,作为所有产品实现的共同接口

ConcreteProduct代表具体的产品

**3.OO原则之依赖倒置原则**

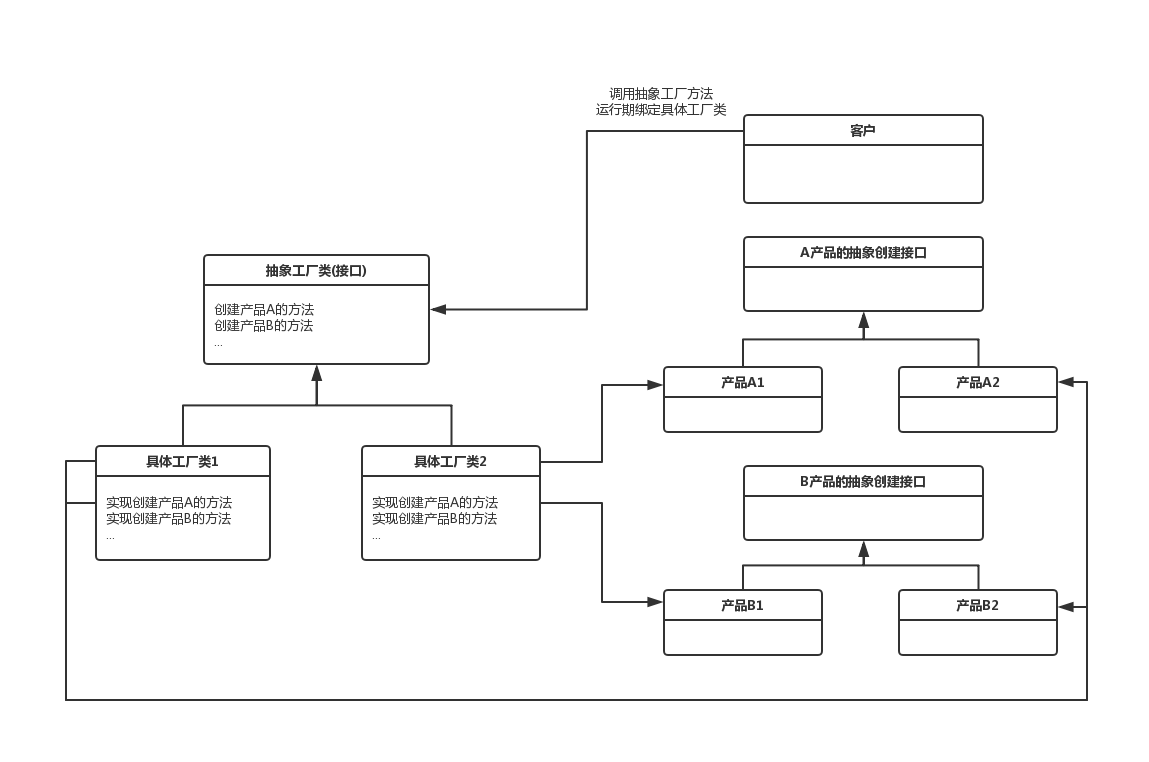
要依赖抽象而不要依赖具体类

在应用工厂方法后,高层组件和低层组件都依赖于抽象方法,符合依赖倒置原则.想要遵循该原则,工厂方法并非唯一的技巧,却是最有威力的技巧之一！

**4.OO模式之抽象工厂模式**

抽象工厂模式提供一个接口,用于创建相关或者依赖对象的家族,而不用明确指定具体类

抽象工厂允许客户使用抽象的接口来创建一组相关的产品,而不需要知道或关心实际产出的产品是什么。这样一来客户就从具体的产品中被解耦。



产品等级结构:即产品的继承结构(比如某一品牌汽车下属的所有汽车类型)

产品族:在抽象工厂模式中,产品族指的是由一个工厂生产的位于不同产品等级结构中的一组产品

**工厂方法和抽象工厂方法的共同点与区别：**

1. 所有工厂都是用来封装对象的创建
2. 工厂方法使用继承(is - a),把对象的创建委托给子类,子类实现工厂方法来创建对象
3. 抽象工厂使用组合(has - a),对象的创建被实现在工厂接口所暴露出来的方法中
4. 工厂方法允许将实例化推迟到子类
5. 遵循依赖倒置原则
6. 抽象工厂创建对象家族(有多个抽象产品类),而工厂方法(只有唯一抽象产品类)仅仅创建单一对象
7. 实际上抽象工厂中每一类产品的创建方法都与工厂方法相同(即抽象类只提供创建接口,子类负责具体创建实例)
8. 抽象工厂方法只是纯粹使用具体工厂方法来创建产品
9. 工厂方法中抽象创建者所实现的代码通常会用到子类创建的具体类型
10. 当增加产品等级结构时,抽象工厂方法不符合开闭原则
11. 工厂方法针对一个产品等级结构,抽象工厂方法增对多个产品等级结构

**五、单件(例)模式**

1. **定义**

单件模式确保一个类只有一个示例,并提供一个全局访问点

1. **场景**

有一些对象实际上我们只需要全局唯一,比如线程池、缓存、对话框、驱动对象等。这些类只能创造一个实例。

1. **实现方法**

单件模式的经典实现(懒汉模式)是把构造器私有化，然后声明一个静态类方法(为类所有而非实例所有)用于实例化。同时在类中绑定一个静态变量用于记录此类的唯一实例。

注:

(1).一般把静态类方法声明为public的,这样才可以在类外部调用该方法

(2).而记录该类的唯一实例对应的静态变量应是private的,只能通过类的静态方法访问

(3).静态方法只能访问静态成员

例：

Public class singleton{

Public:

Static singleton getInstance(){if(cur == nullptr) cur = new singleton(); return cur;}

Private:

Static singleton \*cur;

Singleton(){};

}

懒汉模式是通过时间换取空间,即再方法调用之后才创建对象,在多线程环境下存在风险

**Extra:饥汉模式**

例：

Public class singleton{

Public:

Static::singleton &getInstance(){return cur;}

Private:

Static Singleton cur;

Singleton(){};

}

这种方式在单例定义时就进行了实例化，用空间换取时间,所以全局只有唯一实例cur,故不会有线程安全问题；

**Extra2:改进的懒汉模式,智能指针 + 锁 (双重检查枷锁)**

#include <iostream>

#include <memory> // shared\_ptr

#include <mutex> // mutex

// version 2:

// with problems below fixed:

// 1. thread is safe now

// 2. memory doesn't leak

class Singleton{

public:

typedef std::shared\_ptr<Singleton> Ptr;

~Singleton(){

}

Singleton(Singleton&)=delete;

Singleton& operator=(const Singleton&)=delete;

static Ptr get\_instance(){

// "double checked lock"

if(m\_instance\_ptr==nullptr){

std::lock\_guard<std::mutex> lk(m\_mutex);

if(m\_instance\_ptr == nullptr){

m\_instance\_ptr = std::shared\_ptr<Singleton>(new Singleton);

}

return m\_instance\_ptr;

}

}

private:

Singleton(){

std::cout<<"constructor called!"<<std::endl;

}

static Ptr m\_instance\_ptr;

static std::mutex m\_mutex;

};

// initialization static variables out of class

Singleton::Ptr Singleton::m\_instance\_ptr = nullptr;

std::mutex Singleton::m\_mutex;

int main(){

Singleton::Ptr instance = Singleton::get\_instance();

Singleton::Ptr instance2 = Singleton::get\_instance();

return 0;

}

**Extra3: 最推荐的懒汉式单例(magic static )——局部静态变量**

class Singleton

{

public:

~Singleton(){

std::cout<<"destructor called!"<<std::endl;

}

Singleton(const Singleton&)=delete;

Singleton& operator=(const Singleton&)=delete;

static Singleton& get\_instance(){

static Singleton instance;

return instance;

}

private:

Singleton(){

std::cout<<"constructor called!"<<std::endl;

}

};

int main(int argc, char \*argv[])

{

Singleton& instance\_1 = Singleton::get\_instance();

Singleton& instance\_2 = Singleton::get\_instance();

return 0;

}

这个方法的巧妙之处在于利用了静态局部变量的特性：

（1）该变量在全局数据区分配内存；

（2）静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；

（3）静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0；

（4）它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束(静态全局变量的作用域则是全局)。

这种方法又叫做 Meyers' SingletonMeyer's的单例， 是著名的写出《Effective C++》系列书籍的作者 Meyers 提出的。所用到的特性是在C++11标准中的Magic Static特性：

如果当变量在初始化的时候，并发同时进入声明语句，并发线程将会阻塞等待初始化结束。

这样保证了并发线程在获取静态局部变量的时候一定是初始化过的，所以具有线程安全性。

这种方法具有以下优点：

1. 通过局部静态变量的特性保证了线程安全 (C++11, GCC > 4.3, VS2015支持该特性);
2. 不需要使用共享指针，代码简洁；
3. 注意在使用的时候需要声明单例的引用 Single& 才能获取对象。

**补充：**

注意静态全局变量和普通全局变量的区别：

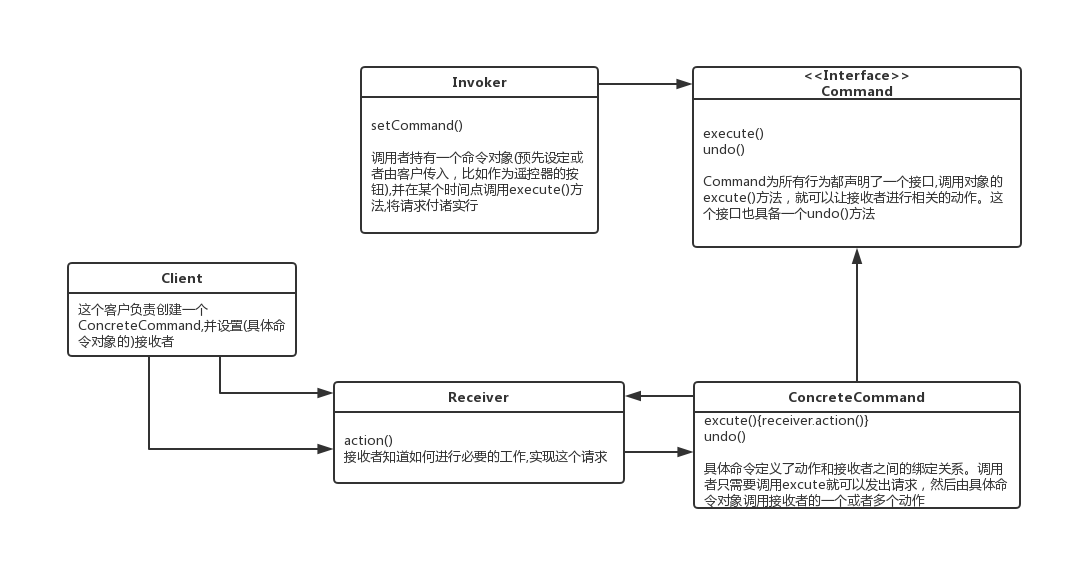
1）全局变量是不显式用 static 修饰的全局变量，全局变量默认是有外部链接性的，作用域是整个工程，在一个文件内定义的全局变量，在另一个文件中，通过 extern 全局变量名的声明，就可以使用全局变量。

2）全局静态变量是显式用 static 修饰的全局变量，作用域是声明此变量所在的文件，其他的文件即使用 extern 声明也不能使用。

**六、命令模式**

**定义:** 将“请求”封装成对象,以便使不同的请求、请求队列或者请求日志来参数化其他对象。命令模式也可以撤销操作。

**类图**

****

各部分的作用:

**客户:** 负责创建或指定具体命令,并相应地设置接收者

**调用者:** 真正负责调用具体命令对象方法的那个对象

**命令类:** 每个具体命令类中都已绑定好具体的接收者,调用命令类对象的接口将调用接收者的具体方法

**接收者:** 响应命令,执行相应动作的对象

打个比方,客户也就是人本身，而调用者就是手里的遥控器，不同按钮中已经封装好具体的命令对象与相应方法，接收者就是遥控器所作用的对象。

**作用：**

命令模式允许我们将动作封装为命令对象，这样就可以随心所欲地存储、调用和传递它们！

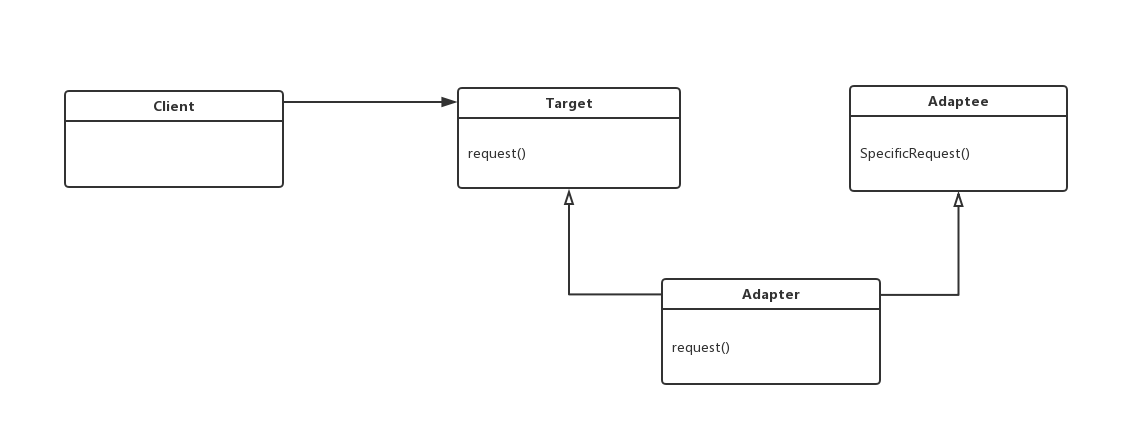
**使用情景：**

当需要将发出请求地对象和执行请求的对象解耦的时候，使用命令模式！

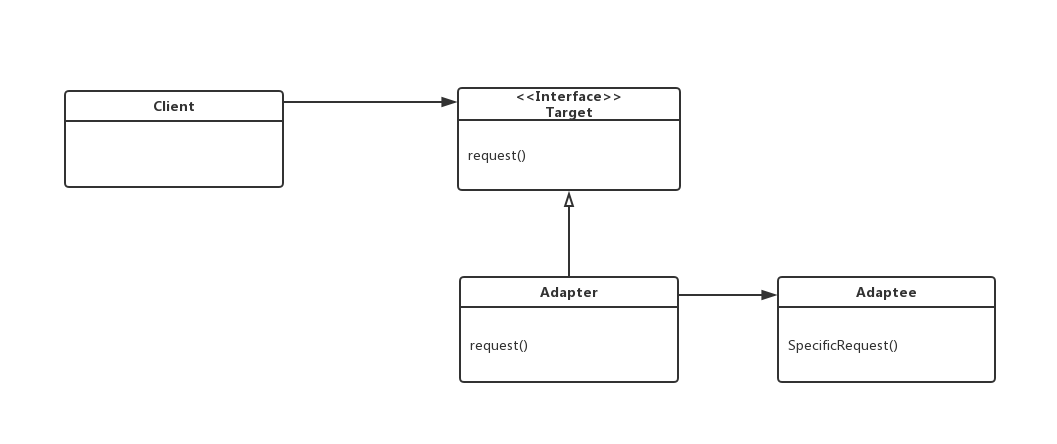
**七、适配器模式与外观模式**

**定义**：适配器模式将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的类之间合作无间。

**类图：**



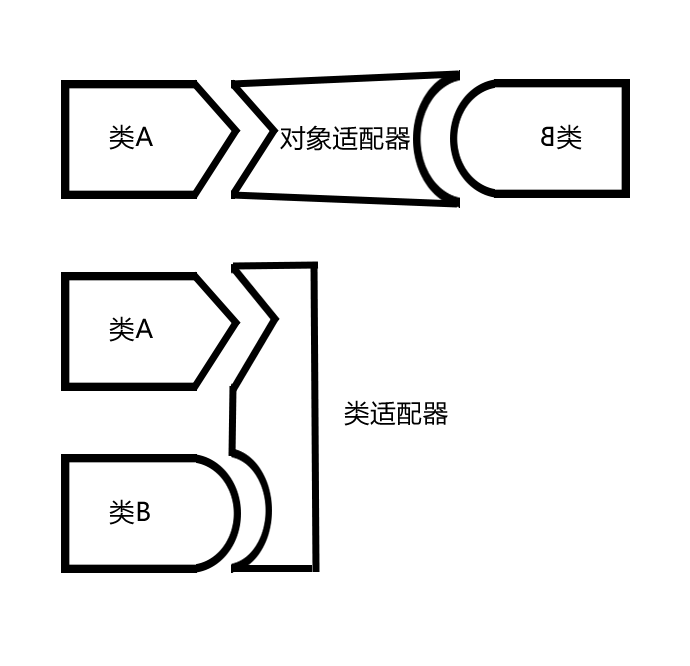
**图7.1 类适配器**



**图7.2 对象适配器**

**类适配器和对象适配器的区别**：

1. 类适配器使用的是多重继承的思想,即直接继承自被适配类和待适配类，由此适配器类同时拥有了二者的所有共有方法
2. 对象适配器使用的是组合的思想,一般做法是提供一个统一的,被适配类的抽象,由其具体的适配器子类中传入待适配类型的引用。从而实现同时拥有两个对象的成员方法，以实现接口适配。
3. 具体比喻的话,类适配器是并行的、对象适配器是串行的，如下图所示



**图7.3 对象适配器与类适配器的区别**

**外观模式定义:** 外观模式提供了一个统一的接口,用来访问子类中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

简单而言，想要使用外观模式，只需要创建一个接口简化而统一的类，用来包装子系统中一个或者多个复杂的类即可(使用组合)。

**外观模式对应的OO原则—最少知识原则**：最少知识原则告诉我们要减少对象间的交互，只留下几个“密友”。即只和“密友”交谈。

当我们正在设计一个系统，不管是任何对象，都要注意其交互的有哪些类，并注意这些类是如何交互的。

这个原则希望我们在设计中，不要让太多类耦合在一起，免得修改一部分而影响其他部分。

**如何在具体的设计中实现最少知识原则？本原则提供了一些方针，我们应该只调用属于以下范围的方法：**

1. **对象本身**
2. **被当作方法的参数而传递进来的对象**
3. **此方法所创建或者实例化的对象**
4. **对象的任何组件**

**注 ：外观和适配器都可以包装许多类，但是外观的意图是简化接口而适配器则是转换接口**