设计模式

- 设计模式（Design pattern）代表了**最佳的实践**，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的**解决方案**。这些解决方案是众多软件开发人员经过相当长的**一段时间的试验和错误总结出来的**。
- 设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计**经验的总结。**使用设计模式是为了**重用代码、让代码更容易被他人理解、保证代码可靠性。**项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。
- **不要为了使用设计模式而使用，不要为了使用设计模式而使用，不要为了使用设计模式而使用**

开闭原则

由Bertrand Meyer提出的开闭原则（Open Closed Principle）是指，软件应该对扩展开放，而对修改关闭。这里的意思是在**增加新功能的时候，能不改代码就尽量不要改**，如果只增加代码就完成了新功能，那是最好的。

一共有 23 种设计模式。这些模式可以分为三大类：

创建型模式（Creational Patterns）：核心思想是要把对象的创建和使用相分离，这样使得两者能相对独立。

结构型模式（Structural Patterns）：结构型模式主要涉及**如何组合各种对象**以便获得更好、更灵活的结构。虽然面向对象的继承机制提供了最基本的子类扩展父类的功能，但结构型模式不仅仅简单地使用继承，而更多地通过**组合与运行期的动态组合**来实现更灵活的功能。

行为型模式（Behavioral Patterns）：主要涉及算法和对象间的职责分配。通过使用对象组合，行为型模式可以描述一组对象应该如何协作来完成一个整体任务。

这次课主要讲解下列：

创建型模式：单例模式、简单工厂模式、工厂方法模式、抽象工厂模式、生成器模式(建造者模式)

结构型模式：装饰器模式

行为型模式：策略模式、观察者模式

单例模式

一个类只有一个实例

注意：

- 1、单例类只能有一个实例。
- 2、单例类必须**自己创建自己**的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。

例子

- 1、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。
- 2、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

优点

- 1、在内存里只有一个实例，减少了内存的开销。
- 2、避免对资源的多重占用（比如多个程序写文件操作）。

几种实现方式

Kotlin的实现方式请见 这里

下面给出Java的几种方式

1、懒汉式，线程不安全

可能有多个线程同时进入了 if (instance == null) ,就**会实例化多次**

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

2、懒汉式，线程安全

必须加锁 synchronized 才能保证单例，但加锁**会影响效率。**当1个线程进getInstance之后,其他的线程只能等待,降低了性能

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
    public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

3、饿汉式

类加载时就初始化，没有用到的时候也存在，**浪费了内存**

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
    return instance;
    }
}
```

4、双重校验锁（DCL，即 double-checked locking)

最好的一种。这种方式采用双锁机制，**安全且在多线程情况下能保持高性能。**

当两个并发线程(thread1和thread2)访问synchronized代码块时，在同一时刻只能有一个线程得到执行，另一个线程受阻塞，必须等待当前线程执行完这个代码块以后才能执行该代码块。

java代码

```
public class Singleton {
    private volatile static Singleton instance;
    //volatile让变量每次在使用的时候，都从主存中取。而不是从各个线程的“工作内存”。volatile变量对于每次使用，线程都能得到当前volatile变量的最新值
    private Singleton (){}
    public static Singleton getInstance() {
    if (instance == null) {                  //(1)
        synchronized (Singleton.class) {
            if (instance == null) {          //(2)
                instance = new Singleton();
            }
        }
    }
    return instance;
    }
}
```

如果只有（1）处的判空，没有（2）处的判空，则当有两个线程进入if后，其中一个线程进入synchronized代码块，产生了一个实例，执行完了这个代码块；然后下一个线程进入synchronized代码块，**会出现又产生一个实例的问题。**

如果只有（2）处的判空，没有（1）处的判空，当有两个线程时，则若已经有了实例，某一个线程进入synchronized代码块，if语句判断为假，执行完synchronized代码块；之后另一个线程再进入。**后面进入的线程在刚才一直在等待前一个线程执行完synchronized代码块，浪费时间**，降低了代码效率。如果1、2处判空都有，则当已经有了实例是，直接第一个判断为假，节省了时间，提高了效率。

Kotlin代码

```
class KtSingleton {
    companion object {
        val instance: KtSingleton by lazy(mode =
LazyThreadSafetyMode.SYNCHRONIZED) {
            //利用LazyThreadSafetyMode.SYNCHRONIZED参数来保证线程安全
            KtSingleton()
        }
    }
}
```

观察者模式

当对象间存在一对多关系时，推荐使用观察者模式（Observer Pattern）。当一个对象被修改时，则会自动通知依赖它的对象，自动更新。

实例：

- 西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作。
- 服务器发送推送新闻，手机客户端接收新闻。
- 气象站的气象数据发生改变的时候,客户端也要对这种变化作出反应。比如这里的客户端就是手机,需要手机能够实时刷新当前的天气状况。可以有两种方式来实现
    1. 由客户端不断的向服务器请求数据,当服务器的数据发生变化之后,客户端也就能够知道这种变化（轮询）（缺点：一直会占用网络，**浪费客户端和服务器的性能**）
    2. 由服务器主动向客户端推送消息,当数据发生改变的时候,服务器将这种变化推送到客户端（推送，在数据发生变化之后再通知客户端数据发生了变化）（优点：比轮询更**节省资源**）

使用场景：

- 一个对象的改变将导致其他一个或多个对象也发生改变。
- 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

观察者模式有两个部分：

1.可观察对象 `Observable` （例如上例中的服务器，气象站，菩萨）它在数据改变的时候通知观察者

2.观察者 `Observer` （例如上例中的客户端，老乌龟）当 `Observable` 中的数据发生改变后会回调其中的方法来通知 `Observer` 数据发生了改变。

①我们在 `Observer` 中持有一个 `Observable` 对象,在通知更新时不传参数,而是在 `Observer` 中通过 `Observable` 对象来获取数据。

②也可以把数据作为参数直接传递给 Observer , Observer 就不需要持有一个 Observable 对象 此处不再赘述

最基本的写法

见代码（kotlin不讲，有兴趣可以自己尝试）

Kotlin用委托方式：

Delegates.observable() 函数接受两个参数: 第一个是初始化值, 第二个是属性值变化事件的响应器(handler)。在属性赋值后会执行事件的响应器(handler)，它有三个参数：被赋值的属性、旧值和新值：

```
var news: String by Delegates.observable("昨天的新闻") {
    property, old, new ->
    Log.d("property name is ${property.name}", "旧值：$old -> 新值：$new")
}
```

MVVM里的View观察ViewModel里的数据

```
//var mldTodayOnHistoryList: MutableLiveData<List<TodayOnHistoryItem>>
mainViewModel.mldTodayOnHistoryList?.observe(
    mActivity,
    //通过对象表达式实现一个匿名内部类的对象
    object : Observer<List<TodayOnHistoryItem>> {
        override fun onChanged(t: List<TodayOnHistoryItem>) {
            //do something...
        }
    })
```

生成器模式(建造者模式)

使用多个简单的对象一步一步构建成一个复杂的对象。一个 Builder 类会一步一步构造最终的对象。

当一个类的构造函数参数个数超过4个，而且这些参数有些是可选的参数，考虑使用构造者模式。

解决的问题

当一个类的构造函数参数超过4个，而且这些参数有些是可选的时，我们通常有两种办法来构建它的对象。 例如我们现在有如下一个类计算机类`Computer`，其中cpu与ram是必填参数，而其他3个是可选参数，那么我们如何构造这个类的实例呢,通常有两种常用的方式：

```
public class Computer {
    private String cpu;//必须
    private String ram;//必须
    private int usbCount;//可选
    private String keyboard;//可选
    private String display;//可选
}
```

第一：折叠构造函数模式（telescoping constructor pattern ），这个我们经常用,如下代码所示

```
public class Computer {
     ...
    public Computer(String cpu, String ram) {
        this(cpu, ram, 0);
    }
    public Computer(String cpu, String ram, int usbCount) {
        this(cpu, ram, usbCount, "罗技键盘");
    }
    public Computer(String cpu, String ram, int usbCount, String keyboard) {
        this(cpu, ram, usbCount, keyboard, "三星显示器");
    }
    public Computer(String cpu, String ram, int usbCount, String keyboard,
String display) {
        this.cpu = cpu;
        this.ram = ram;
        this.usbCount = usbCount;
        this.keyboard = keyboard;
```

```
        this.display = display;
    }
}
```

第二种：Javabean 模式，如下所示

```
public class Computer {
        ...

    public String getCpu() {
        return cpu;
    }
    public void setCpu(String cpu) {
        this.cpu = cpu;
    }
    public String getRam() {
        return ram;
    }
    public void setRam(String ram) {
        this.ram = ram;
    }
    public int getUsbCount() {
        return usbCount;
    }
...
}
```

那么这两种方式有什么弊端呢？ 第一种主要是使用及阅读不方便。第二种方式在构建过程中对象的状态容易发生变化，造成错误。因为那个类中的属性是分步设置的，所以就容易出错。

为了解决这两个问题，builder模式就横空出世了。

实例：

- 1、去快餐店，汉堡、可乐、薯条、炸鸡等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。
- 2、图片处理框架Glide，网络请求框架Retrofit等都使用了此模式。
- ```
  mRetrofit = new Retrofit.Builder()
          .baseUrl("https://v1.alapi.cn/") //设置网络请求的Url地址
          .addConverterFactory(GsonConverterFactory.create()) //设置数据解析器
          .build();
  ```
- ```
  Glide.with(context).load(item.data?.header?.icon).into(ivAvatar)
  ```
- 3、Android中的AlertDialog对话框的构建过程就是建造者模式的典型应用

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("标题");
builder.setMessage("内容");
builder.setPositiveButton("确定", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {}
});
builder.setNegativeButton("取消", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {}
});
builder.show();
```

例示代码

例示1

```
public class Computer {
    private final String cpu;//必须
    private final String ram;//必须
    private final int usbCount;//可选
    private final String keyboard;//可选
    private final String display;//可选

    private Computer(Builder builder){
        this.cpu=builder.cpu;
        this.ram=builder.ram;
        this.usbCount=builder.usbCount;
        this.keyboard=builder.keyboard;
        this.display=builder.display;
    }
    public static class Builder{
        private String cpu;//必须
        private String ram;//必须
        private int usbCount;//可选
        private String keyboard;//可选
        private String display;//可选

        public Builder(String cpu,String ram){
            this.cpu=cpu;
            this.ram=ram;
        }

        public Builder setUsbCount(int usbCount) {
            this.usbCount = usbCount;
            return this;
        }
        public Builder setKeyboard(String keyboard) {
            this.keyboard = keyboard;
            return this;
        }
        public Builder setDisplay(String display) {
            this.display = display;
            return this;
```

```
        }
        public Computer build(){
            return new Computer(this);
        }
    }
}
```

如何使用

在客户端使用链式调用，一步一步的把对象构建出来。

```
Computer computer=new Computer.Builder("因特尔","三星")
                .setDisplay("三星24寸")
                .setKeyboard("罗技")
                .setUsbCount(2)
                .build();
```

策略模式

定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

实例：

- 1、诸葛亮的锦囊妙计，每一个锦囊就是一个策略。 刘备派诸葛亮攻打一座城池，诸葛亮锦囊里有攻城锤攻城，攻城梯爬上城墙攻城，与城主谈判三种锦囊。
- 
- 2、旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略。
- 3、商场对会员打折，算法(策略)一：对初级会员没有折扣。算法(策略)二：对中级会员提供10%的促销折扣。算法(策略)三：对高级会员提供20%的促销折扣。

使用场景：

- 1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。
    - 2、一个系统需要动态地在几种算法中选择一种。
- 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

优点：

- 1、算法可以自由切换。诸葛亮在攻城时就可以随时选择不同的锦囊
- 2、避免使用多重条件判断。
- 3、扩展性良好。

缺点：

策略模式造成很多的策略类。~~如果诸葛亮锦囊妙计太多了可能要拿一辆车来专门运送锦囊~~

策略模式的主要角色

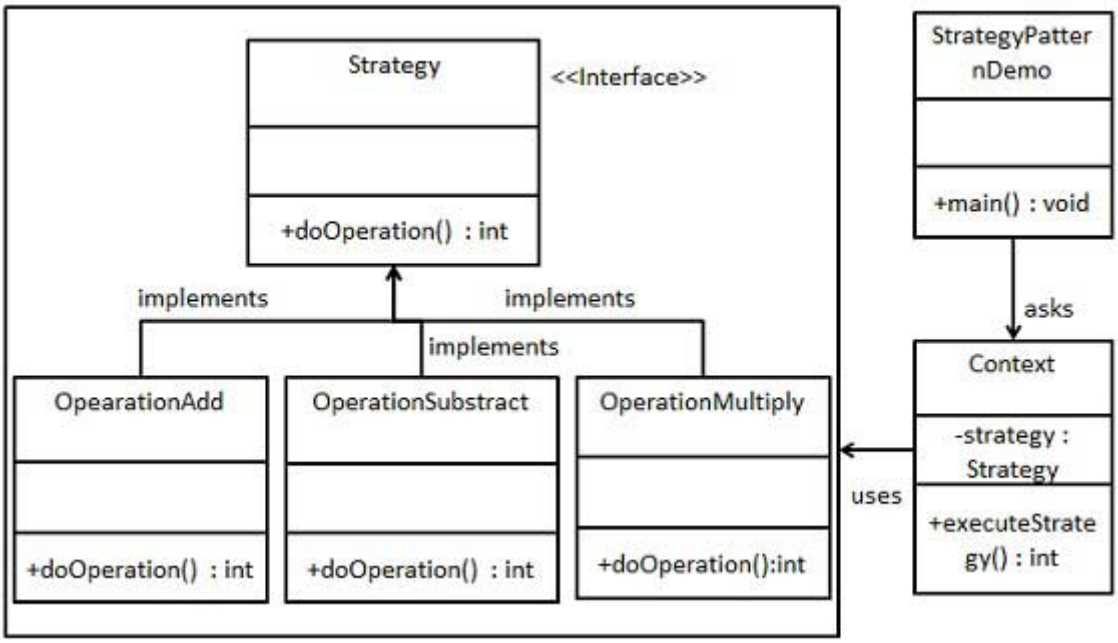
- 抽象策略（Strategy）类：定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法，一般使用接口或抽象类实现。（诸葛亮要攻城）
- 具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，提供具体的算法实现。（锦囊里具体的攻城方法）
- 环境（Context）类：持有一个策略类的引用，最终给客户端调用。（把所有锦囊用一个大袋子装起来）

实现

现在我们用策略模式来制作一个计算器，那就有加减乘除四种策略

创建一个定义活动的 *Strategy* 接口和实现了 *Strategy* 接口的实体策略类。*StrategyContext* 是一个使用了某种策略的类。

我们使用 *StrategyContext* 和策略对象来演示 *StrategyContext* 在它所配置或使用的策略改变时的行为变化。



例示代码

Java代码

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

```
public class OperationSubtract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

```
public class StrategyContext {
    private Strategy strategy;

    public StrategyContext(Strategy strategy){
        this.strategy = strategy;
    }

    public int executeStrategy(int num1, int num2){
        return strategy.doOperation(num1, num2);
    }
}
```

```
//Activity
val strategyContext = StrategyContext(OperationAdd())
Log.d("strategy---Java", "7 + 8 = ${strategyContext.executeStrategy(7, 8)}")
strategyContext.setStrategy(OperationSubtract())
Log.d("strategy---Java", "7 - 8 = ${strategyContext.executeStrategy(7, 8)}")
strategyContext.setStrategy(OperationMultiply())
Log.d("strategy---Java", "7 * 8 = ${strategyContext.executeStrategy(7, 8)}")
```

Kotlin代码

其实与Java代码几乎一样

```
interface KtStrategy {
    fun doOperation(num1: Int, num2: Int): Int
}
```

```
class KtOperationAdd: KtStrategy {
    override fun doOperation(num1: Int, num2: Int): Int {
        return num1 + num2
    }
}
```

```
class KtOperationSubtract: KtStrategy{
    override fun doOperation(num1: Int, num2: Int): Int {
        return num1 - num2
    }
}
```

```
class KtOperationMultiply: KtStrategy{
    override fun doOperation(num1: Int, num2: Int): Int {
        return num1 * num2
    }
}
```

```
class KtStrategyContext(private var strategy: KtStrategy){
    fun executeStrategy(num1: Int, num2: Int): Int{
        return strategy.doOperation(num1, num2);
    }
}
```

```
//Activity
val strategyContextKt = KtStrategyContext(KtOperationAdd())
Log.d("strategy---Kotlin", "7 + 8 = ${strategyContextKt.executeStrategy(7, 8)}")
strategyContextKt.strategy = KtOperationSubtract()
Log.d("strategy---Kotlin", "7 - 8 = ${strategyContextKt.executeStrategy(7, 8)}")
strategyContextKt.strategy = KtOperationMultiply()
Log.d("strategy---Kotlin", "7 * 8 = ${strategyContextKt.executeStrategy(7, 8)}")
```

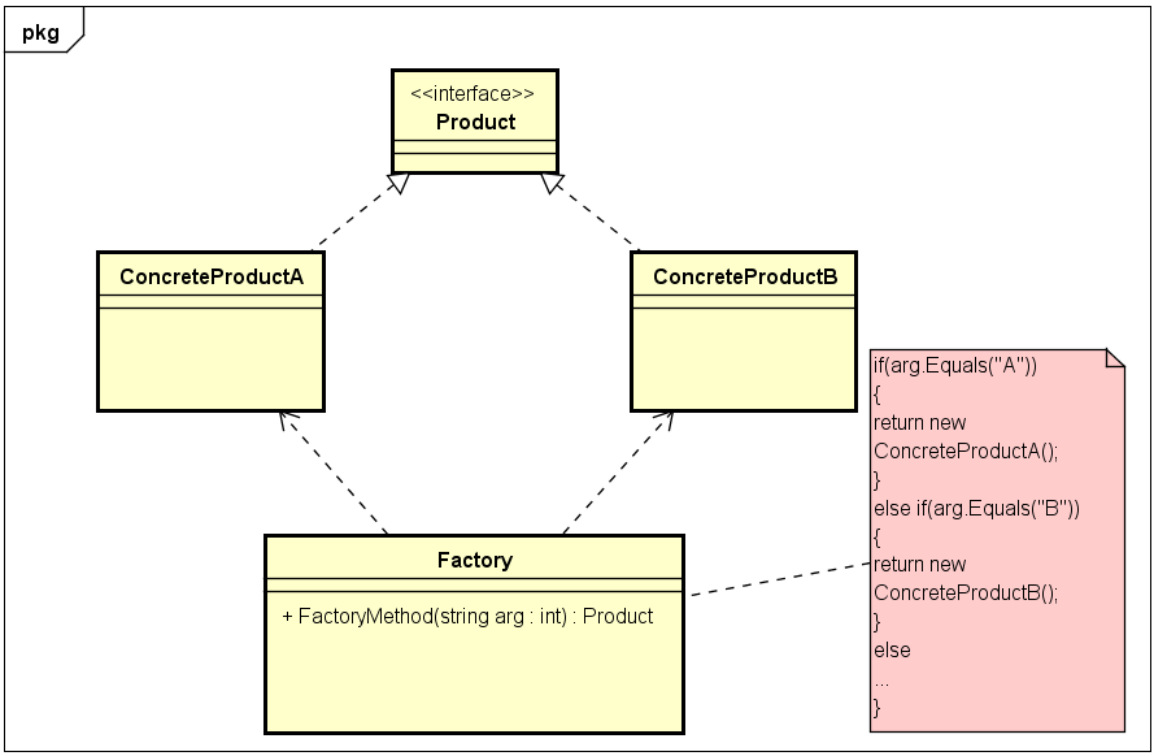
输出：

```
2020-07-10 11:00:52.125 13428-13428/com.redrock.designpattern D/strategy---Java:
7 + 8 = 15
2020-07-10 11:00:52.125 13428-13428/com.redrock.designpattern D/strategy---Java:
7 - 8 = -1
2020-07-10 11:00:52.125 13428-13428/com.redrock.designpattern D/strategy---Java:
7 * 8 = 56
2020-07-10 11:00:52.126 13428-13428/com.redrock.designpattern D/strategy---
Kotlin: 7 + 8 = 15
2020-07-10 11:00:52.126 13428-13428/com.redrock.designpattern D/strategy---
Kotlin: 7 - 8 = -1
2020-07-10 11:00:52.126 13428-13428/com.redrock.designpattern D/strategy---
Kotlin: 7 * 8 = 56
```

简单工厂

定义：

定义一个工厂类，他可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类，在简单工厂模式中用于被创建实例的方法通常为静态(static)方法,因此简单工厂模式又被成为静态工厂方法(Static Factory Method)



理解：

鼠标工厂，专业生产鼠标，给参数0，生产戴尔鼠标，给参数1，生产惠普鼠标。

示例代码

此处只提供Kotlin代码

```
interface Product {
    fun getPrice(): Int
    fun getName(): String
}
```

```
class ProductA(): Product {
    override fun getPrice(): Int {
        return 100
    }

    override fun getName(): String {
        return "ProductA"
    }
}
```

```
class ProductB(): Product {
    override fun getPrice(): Int {
        return 200
    }

    override fun getName(): String {
        return "ProductB"
    }
}
```

```
class ProductC(): Product {
    override fun getPrice(): Int {
        return 300
    }

    override fun getName(): String {
        return "ProductC"
    }
}
```

```
class Factory() {
    fun getProduct(type: String): Product =
            when (type) {
                "ProductA" -> ProductA()
                "ProductB" -> ProductB()
                "ProductC" -> ProductC()
                else -> ProductA()
            }
}
```

```
val productA: Product = Factory().getProduct("ProductA")
Log.d("simplefactory---", "name: ${productA.getName()}; price:
${productA.getPrice()}")
val productB: Product = Factory().getProduct("ProductB")
Log.d("simplefactory---", "name: ${productB.getName()}; price:
${productB.getPrice()}")
val productC: Product = Factory().getProduct("ProductC")
Log.d("simplefactory---", "name: ${productC.getName()}; price:
${productC.getPrice()}")
```

输出：

```
2020-07-10 15:08:44.831 25441-25441/com.redrock.designpattern D/simplefactory--
-: name: ProductA; price: 100
2020-07-10 15:08:44.831 25441-25441/com.redrock.designpattern D/simplefactory--
-: name: ProductB; price: 200
2020-07-10 15:08:44.831 25441-25441/com.redrock.designpattern D/simplefactory--
-: name: ProductC; price: 300
```

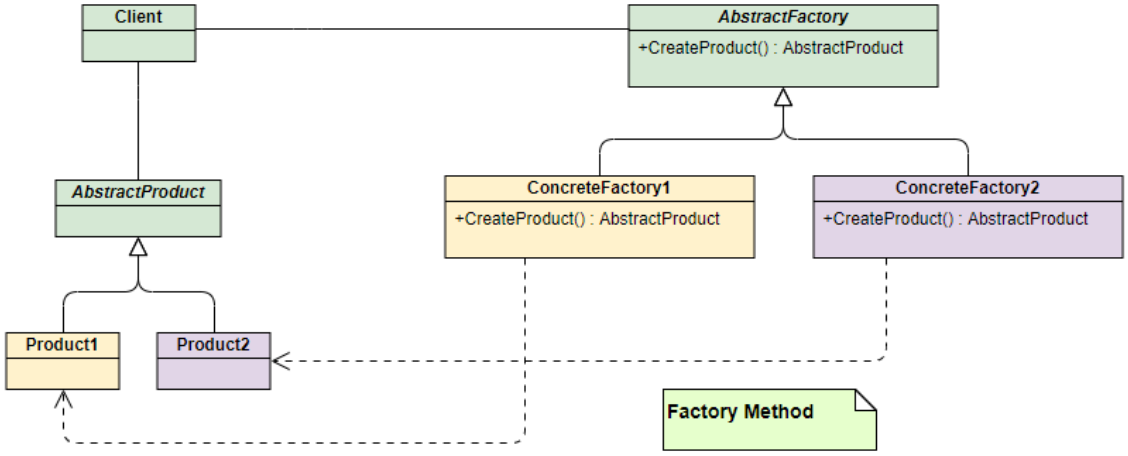
当用户需要新增产品ProductD时，必须在工厂类的生产方法中增加对应的判断分支，所以简单工厂违背了开放封闭原则（开放封闭原则是所有面向对象原则的核心。软件设计本身所追求的目标就是封装变化、降低耦合，而开放封闭原则正是对这一目标的最直接体现。）这时候就需要使用工厂模式了。

工厂模式

工厂方法模式是**对简单工厂模式进一步的解耦**，因为在工厂方法模式中是**一个子类对应一个工厂类**，而这些工厂类都**实现于一个抽象接口。**这相当于是把原本会因为业务代码而庞大的简单工厂类，拆分成了一个个的工厂类，这样代码就不会都耦合在同一个类里了。

我们在创建对象时不会对客户端暴露创建逻辑，并且是**通过使用一个共同的接口**来指向新创建的对象。

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。**和简单工厂模式中工厂负责生产所有产品相比，工厂方法模式将生成具体产品的任务分发给具体的产品工厂，每一个工厂负责一个指定的产品



实例：

需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。

使用场景：

例如日志记录器：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。

理解：

工厂模式也就是鼠标工厂是个父类，有生产鼠标这个接口。戴尔鼠标工厂，惠普鼠标工厂继承它，可以分别生产戴尔鼠标，惠普鼠标。生产哪种鼠标不再由参数决定，而是创建鼠标工厂时，例如若创建戴尔鼠标工厂，则后续直接调用鼠标 `工厂.生产鼠标()` 即可生产戴尔鼠标。

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，不需要知道是怎么造出来的，调用者只关心产品的接口。

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。**

例示代码

此处只提供Kotlin代码

```
interface Product2 {
    fun getPrice(): Int
    fun getName(): String
}
```

```
class Product2A: Product2 {
    override fun getPrice(): Int {
        return 100
    }

    override fun getName(): String {
        return "Product2A"
    }
}
```

```
class Product2B: Product2 {
    override fun getPrice(): Int {
        return 200
    }

    override fun getName(): String {
        return "Product2B"
    }
}
```

```
class Product2C: Product2 {
    override fun getPrice(): Int {
        return 300
    }

    override fun getName(): String {
        return "Product2C"
    }
}
```

```
interface Factory2 {
    fun createProduct(): Product2
}
```

```
class Factory2A : Factory2{
    override fun createProduct(): Product2 {
        return Product2A()
    }
}
```

```
class Factory2B : Factory2{
    override fun createProduct(): Product2 {
        return Product2B()
    }
}
```

```
class Factory2C : Factory2{
    override fun createProduct(): Product2 {
        return Product2C()
    }
}
```

```
val factoryA = Factory2A()
val productA: Product2 = factoryA.createProduct()
Log.d("factorymethod---", "name: ${productA.getName()}; price:
${productA.getPrice()}")
val factoryB = Factory2B()
val productB: Product2 = factoryB.createProduct()
Log.d("factorymethod---", "name: ${productB.getName()}; price:
${productB.getPrice()}")
val factoryC = Factory2C()
val productC: Product2 = factoryC.createProduct()
Log.d("factorymethod---", "name: ${productC.getName()}; price:
${productC.getPrice()}")
```

输出：

```
2020-07-10 15:31:45.321 29194-29194/com.redrock.designpattern D/factorymethod--
-: name: Product2A; price: 100
2020-07-10 15:31:45.321 29194-29194/com.redrock.designpattern D/factorymethod--
-: name: Product2B; price: 200
2020-07-10 15:31:45.321 29194-29194/com.redrock.designpattern D/factorymethod--
-: name: Product2C; price: 300
```

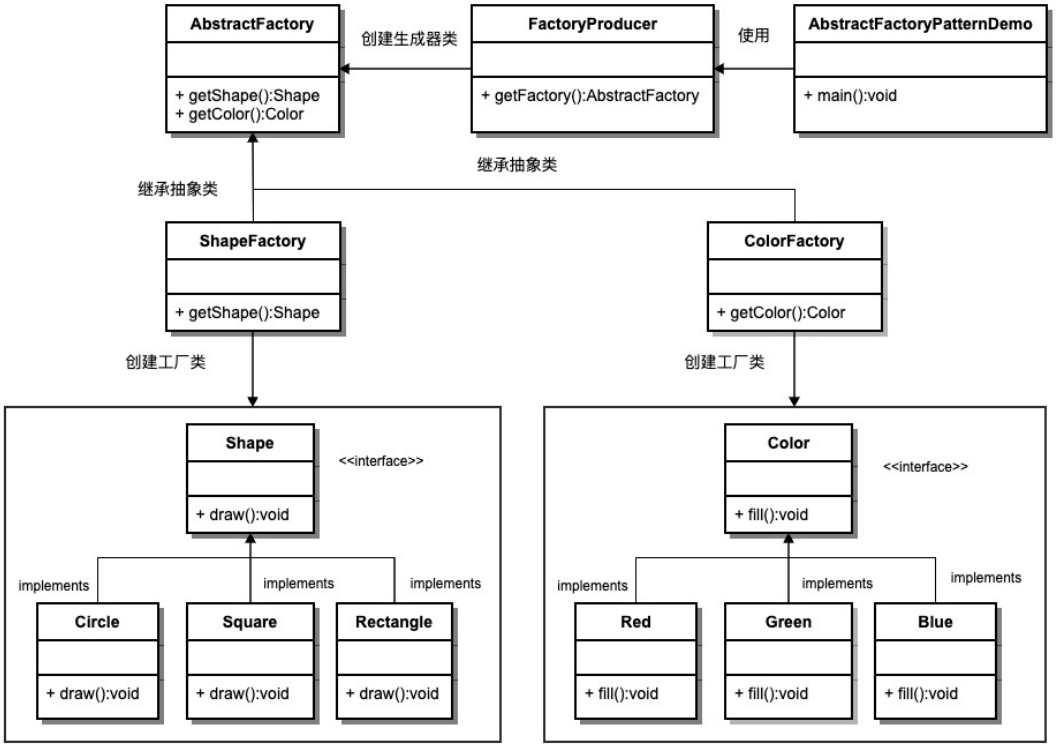
工厂方法把简单工厂的内部逻辑判断转移到了客户端代码来进行，各个不同功能的实例对象的创建代码，也没有耦合在同一个工厂类里，这也是工厂方法模式对简单工厂模式解耦的一个体现。工厂方法模式克服了简单工厂会违背开-闭原则的缺点，又保持了封装对象创建过程的优点。

当需要增加一个新产品ProductD,只需要新建对应的FactoryD来实现生产功能即可,对原有的代码没有任何影响，符合开放封闭原则，但是由于每增加一个产品，都需要新增对应的生产工厂，导致增加额外的开发工作量。

抽象工厂模式

抽象工厂与工厂方法模式的区别在于：抽象工厂是可以**生产多个产品**的。

在抽象工厂模式中,围绕一个超级工厂创建其他工厂，每个工厂都负责创建一个对象族 。将一些相关的具体类组成一个“具体类族”，由同一个工厂来统一生产，这就是“抽象工厂模式”的基本思想。



理解：

假设现在抽象工厂模式不仅生产食物，同时生产玩具。 也就是工厂是个父类，有生产食物，生产玩具两个接口。 混凝土工厂继承它，可以生产混凝土食物+混凝土玩具**（混凝土族）**，乡村工厂继承它，可以生产乡村土+乡村玩具**（乡村族）**。 创建工厂时，若创建混凝土工厂，则后续 `工厂.生产食物()`，则生产混凝土食物； `工厂.生产玩具()`，则生产混凝土玩具。

> **在抽象工厂模式中，假设我们需要增加一个工厂**

假设我们要增加一个外国工厂，和其他工厂一样，继承工厂，之后创建外国土，继承食物类。创建外国玩具，继承玩具类。

优点：

当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

示例代码

此处只提供Kotlin代码

```
interface IProduct {
    fun description()
}
```

```
interface IMouse: IProduct {}
```

```
interface IKeyboard: IProduct {}
```

```
class HPMouse :IMouse{
    override fun description() {
        Log.d("abstractfactory---", "HPMouse")
    }
}
```

```
class HPKeyboard :IKeyboard{
    override fun description() {
        Log.d("abstractfactory---", "HPKeyboard")
    }
}
```

```
class DellMouse :IMouse{
    override fun description() {
        Log.d("abstractfactory---", "DellMouse")
    }
}
```

```
class DellKeyboard :IKeyboard{
    override fun description() {
        Log.d("abstractfactory---", "DellKeyboard")
    }
}
```

```
interface ComputerFactory {
    fun createKeyboard(): IKeyboard
    fun createMouse(): IMouse
}
```

```
class HPFactory :ComputerFactory{
    override fun createKeyboard(): IKeyboard {
        return HPKeyboard()
    }

    override fun createMouse(): IMouse {
        return HPMouse()
    }
}
```

```
class DellFactory :ComputerFactory{
    override fun createKeyboard(): IKeyboard {
        return DellKeyboard()
    }

    override fun createMouse(): IMouse {
        return DellMouse()
    }
}
```

```
val hPFactory: ComputerFactory = HPFactory()
val hPKeyboard: IKeyboard = hPFactory.createKeyboard()
hPKeyboard.description()
val hPMouse: IMouse = hPFactory.createMouse()
hPMouse.description()

val dellFactory: ComputerFactory = DellFactory()
val dellKeyboard: IKeyboard = dellFactory.createKeyboard()
dellKeyboard.description()
val dellMouse: IMouse = dellFactory.createMouse()
dellMouse.description()
```

输出：

```
2020-07-10 16:20:58.459 6477-6477/com.redrock.designpattern D/abstractfactory--
-: HPKeyboard
2020-07-10 16:20:58.459 6477-6477/com.redrock.designpattern D/abstractfactory--
-: HPMouse
2020-07-10 16:20:58.459 6477-6477/com.redrock.designpattern D/abstractfactory--
-: DellKeyboard
2020-07-10 16:20:58.459 6477-6477/com.redrock.designpattern D/abstractfactory--
-: DellMouse
```

装饰器模式

允许向一个现有的对象添加新的功能，同时又不改变其结构。这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

意图：

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

主要解决：

一般的，我们为了扩展一个类经常使用继承方式实现，随着扩展功能的增多，子类会很膨胀。

如何解决：

将具体功能职责划分，同时继承装饰者模式。

何时使用：

在不想增加很多子类的情况下扩展类。

优点：

装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

缺点：

多层装饰比较复杂。

应用实例：

不论一幅画有没有画框都可以挂在墙上，但是通常都是有画框的，并且实际上是画框被挂在墙上。在挂在墙上之前，画可以被蒙上玻璃，装到框子里；这时画、玻璃和画框形成了一个物体。

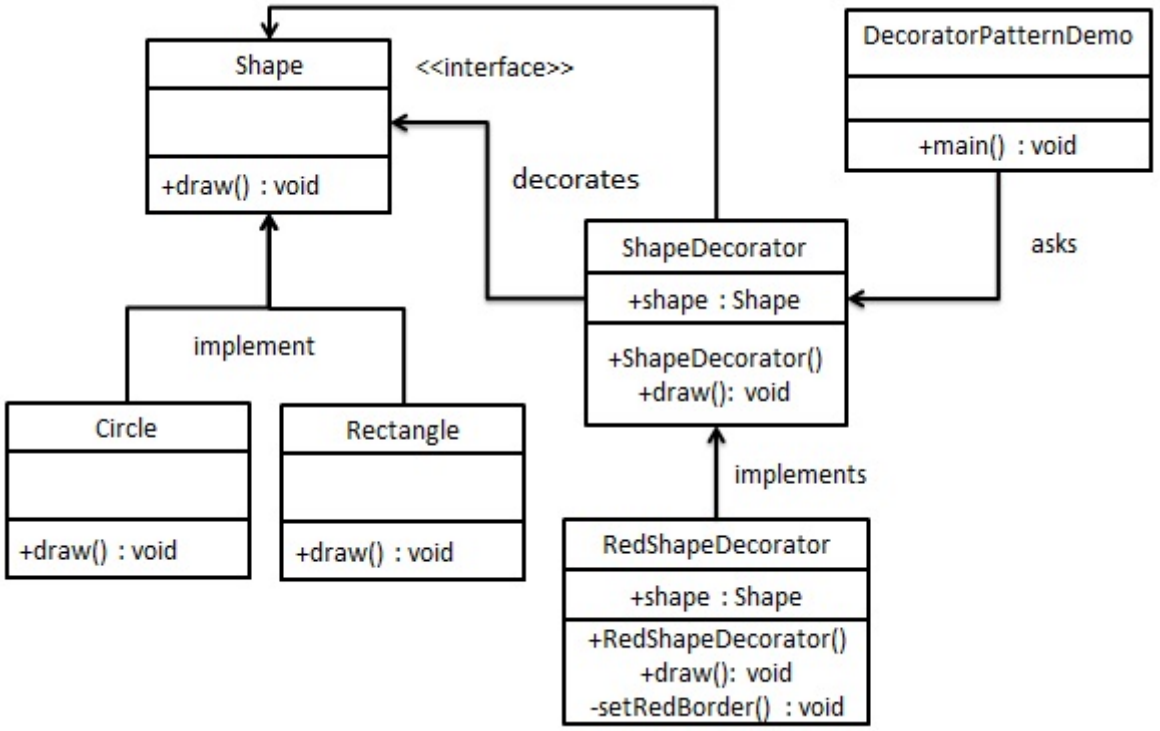
实现

假设我们现在要生成不同颜色的圆形和长方形。

我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类。然后我们创建一个实现了 *Shape* 接口的抽象装饰类 *ShapeDecorator*，并把 *Shape* 对象作为它的实例变量。

RedShapeDecorator 是实现了 *ShapeDecorator* 的实体类。

DecoratorPatternDemo，我们的演示类使用 *RedShapeDecorator* 来装饰 *Shape* 对象。



示例代码：

此处只提供Kotlin代码

```
interface Shape {
    fun draw()
}
```

```
class Rectangle :Shape{
    override fun draw() {
        Log.d("decorator---", "Shape: Rectangle")
    }
}
```

```
class Circle :Shape{
    override fun draw() {
        Log.d("decorator---", "Shape: Circle")
    }
}
```

```
abstract class ShapeDecorator(private var decoratedShape: Shape) : Shape {
    override fun draw() {
        decoratedShape.draw()
    }
}
```

```
class RedShapeDecorator(decoratedShape: Shape) : ShapeDecorator(decoratedShape)
{
    override fun draw() {
        super.draw()
        setRedBorder(decoratedShape)
    }

    private fun setRedBorder(decoratedShape: Shape) {
        Log.d("decorator---", "Border Color: Red")
    }
}
```

```
val circle: Shape = Circle()
val redCircle: ShapeDecorator = RedShapeDecorator(Circle())
val redRectangle: ShapeDecorator = RedShapeDecorator(Rectangle())
Log.d("decorator---", "Circle with normal border")
circle.draw()

Log.d("decorator---", "Circle of red border")
redCircle.draw()

Log.d("decorator---", "Rectangle of red border")
redRectangle.draw()
```

输出：

```
2020-07-10 18:03:42.586 25633-25633/com.redrock.designpattern D/decorator---:
Circle with normal border
2020-07-10 18:03:42.586 25633-25633/com.redrock.designpattern D/decorator---:
Shape: Circle
2020-07-10 18:03:42.586 25633-25633/com.redrock.designpattern D/decorator---:
Circle of red border
2020-07-10 18:03:42.586 25633-25633/com.redrock.designpattern D/decorator---:
Shape: Circle
2020-07-10 18:03:42.586 25633-25633/com.redrock.designpattern D/decorator---:
Border Color: Red
2020-07-10 18:03:42.586 25633-25633/com.redrock.designpattern D/decorator---:
Rectangle of red border
2020-07-10 18:03:42.586 25633-25633/com.redrock.designpattern D/decorator---:
Shape: Rectangle
2020-07-10 18:03:42.586 25633-25633/com.redrock.designpattern D/decorator---:
Border Color: Red
```

面向对象设计原则

1 单一职责（Single Responsibility Principle）

这个原则顾名就可以思义，就是**一个类应该只负责一个职责**，术语叫：**仅有一个引起其变化的原因。**简单点说：**一个类**中应该是一**组相关性很高的函数**及**数据**的封装。看起来简单，但是做起来就难了，这可能是六大原则中最难以熟练掌握的一个原则了，它高度依赖程序员的自身素质及业务场景。

2 开闭原则（Open Close Principle）

它是面向对象最重要的设计原则，由Bertrand Meyer（勃兰特.梅耶）在1988年出版的《面向对象软件构造》。中提出的。

本尊照片，图片来源

定义如下

> 开闭原则(Open-Closed Principle, OCP)：一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展。

提倡一个类一旦开发完成，后续增加新的功能就不应该通过修改这个类来完成，而是通过继承，增加新的类。 大家想必都听过软件需求不断变化的那个段子，在软件开发这个行当唯一不变的就是变化本身。那为什么应该对修改关闭呢，因为你一旦修改了某个类就有可能破坏系统原来的功能，就需要重新测试。其实我知道你们此刻在想什么，回忆一下自己的日常工作，有几个遵守了这个原则，都是需求来了就找到原来的类，进去改代码呗，^_^。看看有指导原则尚且如此，没有的话就更加乱套了。

那么是不是就一定不能修改原来的类的，当然不是了，我们都是成年人了，要清楚的认识到，这个世界**不是非黑即白**的。当我们发现原来的类已经烂到家了，当然在有条件的情况下及时重构，避免系统加速腐败。

3 里氏替换原则（Liskov Substitution Principle）

这个原则的的提出是Barbara Liskov

定义如下：

> 里氏代换原则(Liskov Substitution Principle, LSP)：**所有引用基类（父类）的地方必须能透明地使用其子类的对象。**

简单点说，一个软件系统中**所有用到一个类的地方**都**替换成其子类**，系统应该**仍然可以正常工作**。这个原则**依赖面向对象的继承特性**和**多态特性**，这个原则我们有意无意中使用的就比较多了。因为一个优秀的程序员一定面向抽象（接口）编程的，如果你不是，说明你还有很大的进步空间。

例如我们有如下的代码，一个图形的基类`Shap`,以及它的两个子类`Rectangle`，`Triangle`,安装里式替换原则，所有使用`Shape`的地方都可以安全的替换成其子类。

```
//基类
public abstract class Shape {
    public abstract void draw();
}
//子类矩形
public class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("绘制矩形");
    }
}
//子类三角形
public class Triangle extends Shape {
    @Override
    public void draw() {
```

```
        System.out.println("绘制三角形");
    }
}
```

写一个使用Shape类的函数

```
public static void main(String[] args) {
        //使用Shape的子类Triangle 的实例来替换Shape的实例，程序工作正常
        drawShape(new Triangle());
    }
    private static void drawShape(Shape shape){
        System.out.println("开始画图");
        shape.draw();
        System.out.println("结束画图");
    }
```

输出结果：

```
开始画图
绘制三角形
结束画图
```

如上代码所示：本来`drawShape()`函数需要一个`Shape`的实例，而我们却传给他一个其子类的实例，但是它正常工作了。我们使用`Shape`的子类`Triangle`的实例来替换`Shape`的实例，程序工作正常。这个原则也非常重要而常用，面向抽象编程。

4 依赖倒置原则（Dependence Inversion Principle）

这个原则的提倡者正是大名鼎鼎的 Robert C. Martin，人称**Bob大叔**

定义

> 依赖倒转原则(Dependency Inversion Principle, DIP)：抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。
> 关键点： 1. 高层模块不应该依赖低层模块，两者都应该依赖其抽象 2. 抽象不应该依赖细节 3. 细节应该依赖抽象

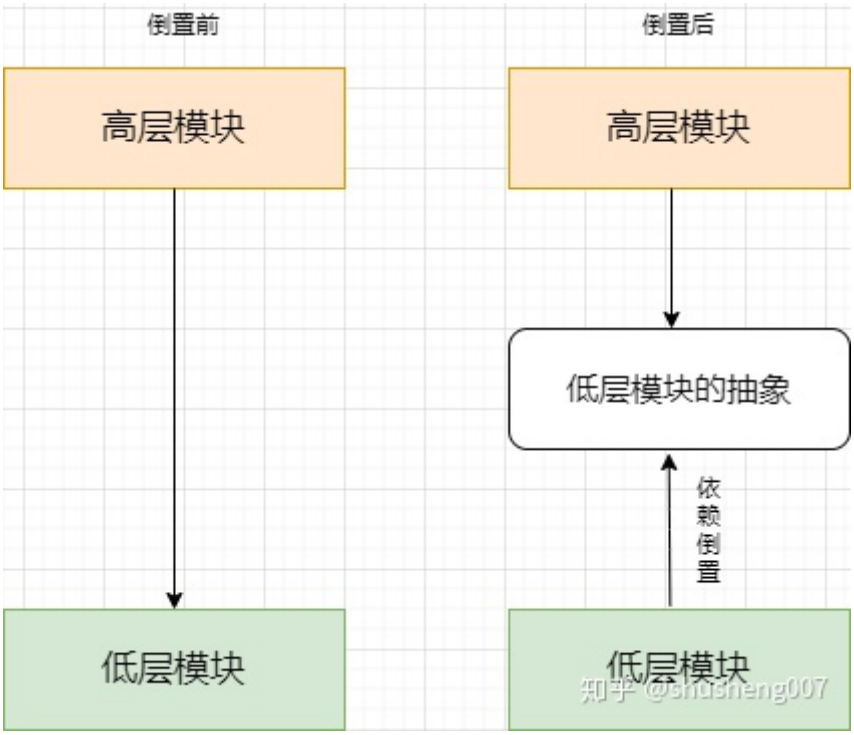
看到上面的介绍是不是已经懵了,因为这个概念人家是从软件设计的角度说的，我们应该将其对应到我们具体的实践当中去理解，例如**Java**领域

抽象：java中的抽象类或者接口 （如上面代码中的Shape 抽象类） 细节：java中的具体实现类（如上面代码中的Rectangle 和Triangle 实体类） 高层模块：java中的调用类（例如上面代码中`drawShape(Shape shape)`函数的类） 低层模块：java中的实现类（细节）

依赖倒置又叫依赖倒转，关键在倒置上，啥叫倒置，那不倒置的时候是什么样的？如下面图所示

正常情况下：调用类（高层模块）应该依赖**具体实现类**（低层模块实现细节）

倒置后：高层模块与低层模块都依赖了实现类的**接口**（低层模块的细节抽象），底层模块的依赖箭头向上了，所以叫依赖倒置了。



例如菜鸟程序员林潼会这么写代码

```
private static void drawRectangle (Rectangle rectangle){
        rectangle.draw();
    }
    private static void drawTriangle  (Triangle triangle){
        triangle.draw();
    }
```

而聪明的其他学姐则会

```
private static void drawShape(Shape shape){
        shape.draw();
    }
```

那么我的代码会有什么问题呢，假设现在产品经理觉得矩形不好看，让我将矩形换成五角形，那么我就要同时修改调用类和增加一个绘制类，而其他学姐的代码只需要增加一个五角形的绘制类，这就遵循了**开关闭原则**

所以我们要对接口编程，举几个具体的例子：声明方法参数的类型，实例变量的类型，方法的返回值类型，类型强制转换等等场景。

我的代码直接依赖了实现细节，而学姐们的代码依赖的是实现细节的抽象（依赖倒置了）。

与依赖倒置（**DIP**）相关的还有依赖注入(**di- dependency injection**)，控制翻转(**Ioc—Inversion of Control**)，记住他们不是同一个东西。

5 接口隔离原则（Interface Segregation Principle)

> 接口隔离原则(Interface Segregation Principle, ISP)：使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。

其实这个原则是很容易理解的，就是让调用者依赖的接口尽可能的小。比如一只鸟和一只猫，都是动物，鸟能飞，但不能让猫也会飞。

6 迪米特原则（Law of Demeter 又名Least Knowledge Principle)

迪米特法则来自于1987年美国东北大学(Northeastern University)一个名为“Demeter”的研究项目，又称最少知识原则(LeastKnowledge Principle, LKP)，其定义如下：

> 迪米特法则(Law of Demeter, LoD)：一个软件实体应当尽可能少地与其他实体发生相互作用。

一个类应该对自己需要调用的类知道得最少，类的内部如何实现、如何复杂都与调用者或者依赖者没关系，调用者或者依赖者只需要知道他需要的方法即可，其他的我一概不关心