工厂模式

不使用工厂模式

```
public class BMW320 {
    public BMW320(){
        System.out.println("制造-->BMW320");
    }
}

public class BMW523 {
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}

public class Customer {
    public static void main(String[] args) {
        BMW320 bmw320 = new BMW320();
        BMW523 bmw523 = new BMW523();
    }
}
```

使用简单工厂模式

产品类

```
abstract class BMW {
    public BMW(){

    }
}

public class BMW320 extends BMW {
    public BMW320() {
        System.out.println("制造-->BMW320");
    }
}
public class BMW523 extends BMW{
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}
```

工厂类

```
public class Factory {
    public BMW createBMW(int type) {
        switch (type) {

        case 320:
            return new BMW320();

        case 523:
            return new BMW523();

        default:
            break;
        }
        return null;
    }
}
```

客户类

```
public class Customer {
    public static void main(String[] args) {
        Factory factory = new Factory();
        BMW bmw320 = factory.createBMW(320);
        BMW bmw523 = factory.createBMW(523);
    }
}
```

1) 工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑，用来创建产品
2) 抽象产品角色：它一般是具体产品继承的父类或者实现的接口。
3) 具体产品角色：工厂类所创建的对象就是此角色的实例。在java中由一个具体类实现。

工厂方法模式

工厂方法模式去掉了简单工厂模式中工厂方法的静态属性，使得它可以被子类继承。这样在简单工厂模式里集中在工厂方法上的压力可以由工厂方法模式里不同的工厂子类来分担。

工厂方法模式组成：

1)抽象工厂角色： 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在java中它由抽象类或者接口来实现。

2)具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。

3)抽象产品角色：它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。

4)具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。

（开闭原则)当有新的产品产生时，只要按照抽象产品角色、抽象工厂角色提供的合同来生成，那么就可以被客户使用，而不必去修改任何已有 的代码。

产品类

```
 abstract class BMW {
    public BMW(){

    }
}
public class BMW320 extends BMW {
    public BMW320() {
        System.out.println("制造-->BMW320");
    }
}
public class BMW523 extends BMW{
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}
```

创建工厂类:

```
interface FactoryBMW {
    BMW createBMW();
}

public class FactoryBMW320 implements FactoryBMW{

    @Override
    public BMW320 createBMW() {

        return new BMW320();
    }

}
public class FactoryBMW523 implements FactoryBMW {
    @Override
    public BMW523 createBMW() {

        return new BMW523();
    }
}
```

客户类

```
public class Customer {
    public static void main(String[] args) {
        FactoryBMW320 factoryBMW320 = new FactoryBMW320();
        BMW320 bmw320 = factoryBMW320.createBMW();

        FactoryBMW523 factoryBMW523 = new FactoryBMW523();
        BMW523 bmw523 = factoryBMW523.createBMW();
    }
}
```

策略模式

定义

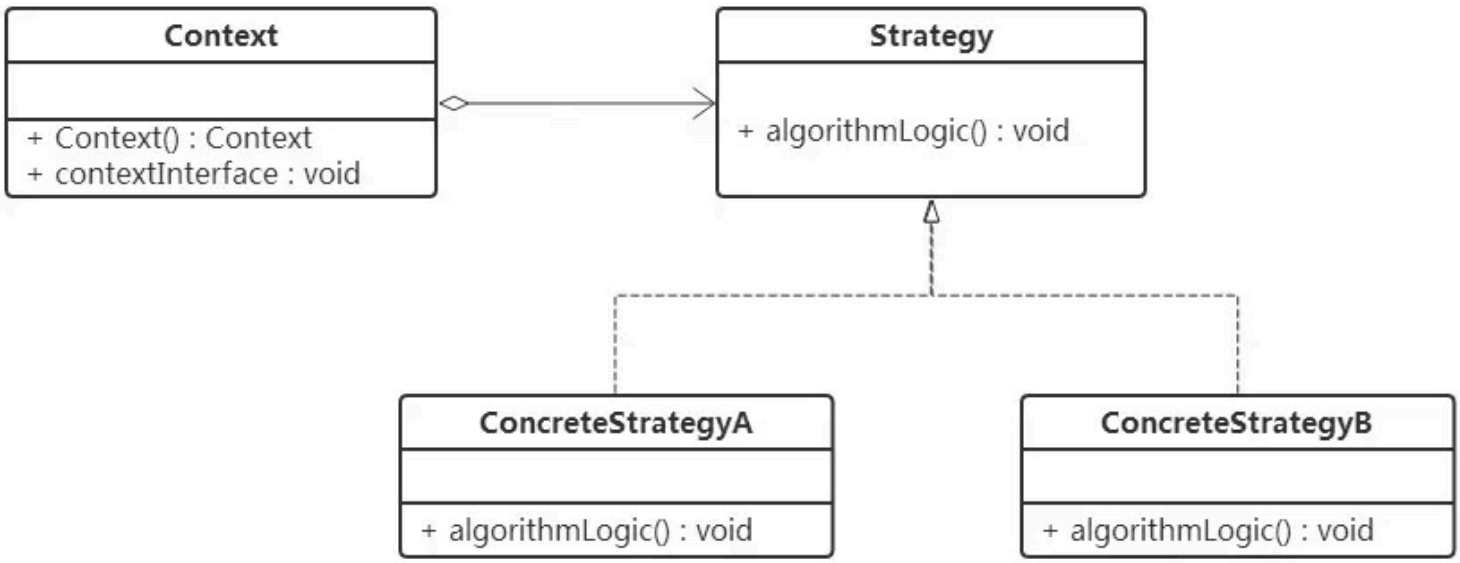
定义一组算法，将每一个算法封装起来，从而使它们可以相互切换。

特点

1）一组算法，那就是不同的策略。

2）这组算法都实现了相同的接口或者继承相同的抽象类，所以可以相互切换。

UML



策略模式涉及到的角色有三个：

- 封装角色：上层访问策略的入口，它持有抽象策略角色的引用。
- 抽象策略角色：提供接口或者抽象类，定义策略组必须拥有的方法和属性。
- 具体策略角色：实现抽象策略，定义具体的算法逻辑。

```
// Context持有Strategy的引用，并且提供了调用策略的方法，
  public class Context {

    private Strategy strategy;

    /**
     * 传进的是一个具体的策略实例
     * @param strategy
     */
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    /**
     * 调用策略
     */
    public void contextInterface() {
        strategy.algorithmLogic();
    }

}
```

```java
// 抽象策略角色，定义了策略组的方法
public interface Strategy {

    public void algorithmLogic();

}
```

```java
// 具体策略角色类
public class ConcreteStrategyA implements Strategy{

    @Override
    public void algorithmLogic() {
        // 具体的算法逻辑（）
    }
}
```

```java
// 客户端
public class Client {

    public static void main(String[] args) {
        // 操控比赛，这场要输
        Context context = new Context(new ConcreteStrategyA());
        context.contextInterface();
    }
}
```

策略模式的优点

（1）策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免代码重复。

（2）使用策略模式可以避免使用多重条件(if-else)语句。多重条件语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重条件语句里面，比使用继承的办法还要原始和落后。

策略模式的缺点

（1）客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道算法或行为的情况。

（2）由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。

工厂结合策略实战

课堂代码

单例设计模式(面试重点)

概念：单例对象的类必须保证只有一个实例存在

适用场景： 单例模式只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等。如：

1. 需要频繁实例化然后销毁的对象。
2. 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
3. 有状态的工具类对象。

4. 频繁访问数据库或文件的对象。

常见写法：

1. 饿汉式

```
public class Singleton {
 /**
 * 优点：没有线程安全问题，简单
 * 缺点：提前初始化会延长类加载器加载类的时间；如果不使用会浪费内存空间；  不能传递参数
 */
 private static final Singleton instance = new Singleton();
 private Singleton(){};
 public static Singleton getInstance(){
     return instance;
 }
}
```

2. 懒汉式

```


public class Singleton{
/**
 * 优点:解决线程安全，延迟初始化( Effective Java推荐写法)
 */
private Singleton(){}
public static Singleton getInstance () {
    return Holder.SINGLE_TON;
}
private static class Holder{
    private static final Singleton SINGLE_TON = new Singleton();
}
}
```

3. 双重检查锁（double checked locking）

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

volatile指令重排序

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

1. 在单线程环境下不能改变程序运行的结果；
2. 存在数据依赖关系的不允许重排序

uniqueSingleton = new Singleton();
分配内存空间
初始化对象
将对象指向刚分配的内存空间
但是有些编译器为了性能的原因，可能会将第二步和第三步进行重排序，顺序就成了：
分配内存空间
将对象指向刚分配的内存空间
初始化对象
现在考虑重排序后，两个线程发生了以下调用：

Time	Thread A	Thread B
T1	检查到uniqueSingleton为空	
T2	获取锁	
T3	再次检查到uniqueSingleton为空	
T4	为uniqueSingleton分配内存空间	
T5	将uniqueSingleton指向内存空间	
T6		检查到uniqueSingleton不为空
T7		访问uniqueSingleton（此时对象还未完成初始化）
T8	初始化uniqueSingleton	

在这种情况下，T7时刻线程B对uniqueSingleton的访问，访问的是一个初始化未完成的对象。
使用了volatile关键字后，重排序被禁止，所有的写（write）操作都将发生在读（read）操作之前。

4. 单例模式的破坏

```
Singleton sc1 = Singleton.getInstance();
Singleton sc2 = Singleton.getInstance();
System.out.println(sc1); // sc1, sc2是同一个对象
System.out.println(sc2);
/*通过反射的方式直接调用私有构造器*/
Class<Singleton> clazz = (Class<Singleton>) Class.forName("com.learn.example.Singleton");
Constructor<Singleton> c = clazz.getDeclaredConstructor(null);
c.setAccessible(true); // 跳过权限检查
Singleton sc3 = c.newInstance();
Singleton sc4 = c.newInstance();
System.out.println("通过反射的方式获取的对象sc3: " + sc3);  // sc3, sc4不是同一个对象
System.out.println("通过反射的方式获取的对象sc4: " + sc4);
```

```
//防止反射获取多个对象的漏洞
private Singleton() {
if (null != SingletonClassInstance.instance)
    throw new RuntimeException();
}
```

5. spring中bean的单例

    1. 当多个用户同时请求一个服务时，容器会给每一个请求分配一个线程，这时多个线程会并发执行该请求对应的业务逻辑（成员方法），此时就要注意了，如果该处理逻辑中有对单例状态的修改（体现为该单例的成员属性），则必须考虑线程同步问题。
    有状态就是有数据存储功能。有状态对象(Stateful Bean)，就是有实例变量的对象，可以保存数据，是非线程安全的。在不同方法调用间不保留任何状态。

无状态就是一次操作，不能保存数据。无状态对象(Stateless Bean)，就是没有实例变量的对象 .不能保存数据，是不变类，是线程安全的。

2. 实现

```
public abstract class AbstractBeanFactory implements ConfigurableBeanFactory{
/**
* 充当了Bean实例的缓存，实现方式和单例注册表相同
*/
private final Map singletonCache=new HashMap();
public Object getBean(String name)throws BeansException{
   return getBean(name,null,null);
}
...
public Object getBean(String name,Class requiredType,Object[] args)throws BeansException{
  //对传入的Bean name稍做处理，防止传入的Bean name名有非法字符(或则做转码)
  String beanName=transformedBeanName(name);
  Object bean=null;
  //手工检测单例注册表
  Object sharedInstance=null;
  //使用了代码锁定同步块，原理和同步方法相似，但是这种写法效率更高
  synchronized(this.singletonCache){
     sharedInstance=this.singletonCache.get(beanName);
   }
  if(sharedInstance!=null){
     ...
     //返回合适的缓存Bean实例
     bean=getObjectForSharedInstance(name,sharedInstance);
  }else{
    ...
    //取得Bean的定义
    RootBeanDefinition mergedBeanDefinition=getMergedBeanDefinition(beanName,false);
     ...
    //根据Bean定义判断，此判断依据通常来自于组件配置文件的单例属性开关
    //<bean id="date" class="java.util.Date" scope="singleton"/>
    //如果是单例，做如下处理
    if(mergedBeanDefinition.isSingleton()){
       synchronized(this.singletonCache){
        //再次检测单例注册表
         sharedInstance=this.singletonCache.get(beanName);
         if(sharedInstance==null){
            ...
           try {
              //真正创建Bean实例
              sharedInstance=createBean(beanName,mergedBeanDefinition,args);
              //向单例注册表注册Bean实例
               addSingleton(beanName,sharedInstance);
           }catch (Exception ex) {
              ...
           }finally{
              ...
          }
         }
       }
      bean=getObjectForSharedInstance(name,sharedInstance);
    }
   //如果是非单例，即prototpye，每次都要新创建一个Bean实例
   //<bean id="date" class="java.util.Date" scope="prototype"/>
   else{
```

```
        bean=createBean(beanName,mergedBeanDefinition,args);
    }
}
...
return bean;
}
}
```

命令模式

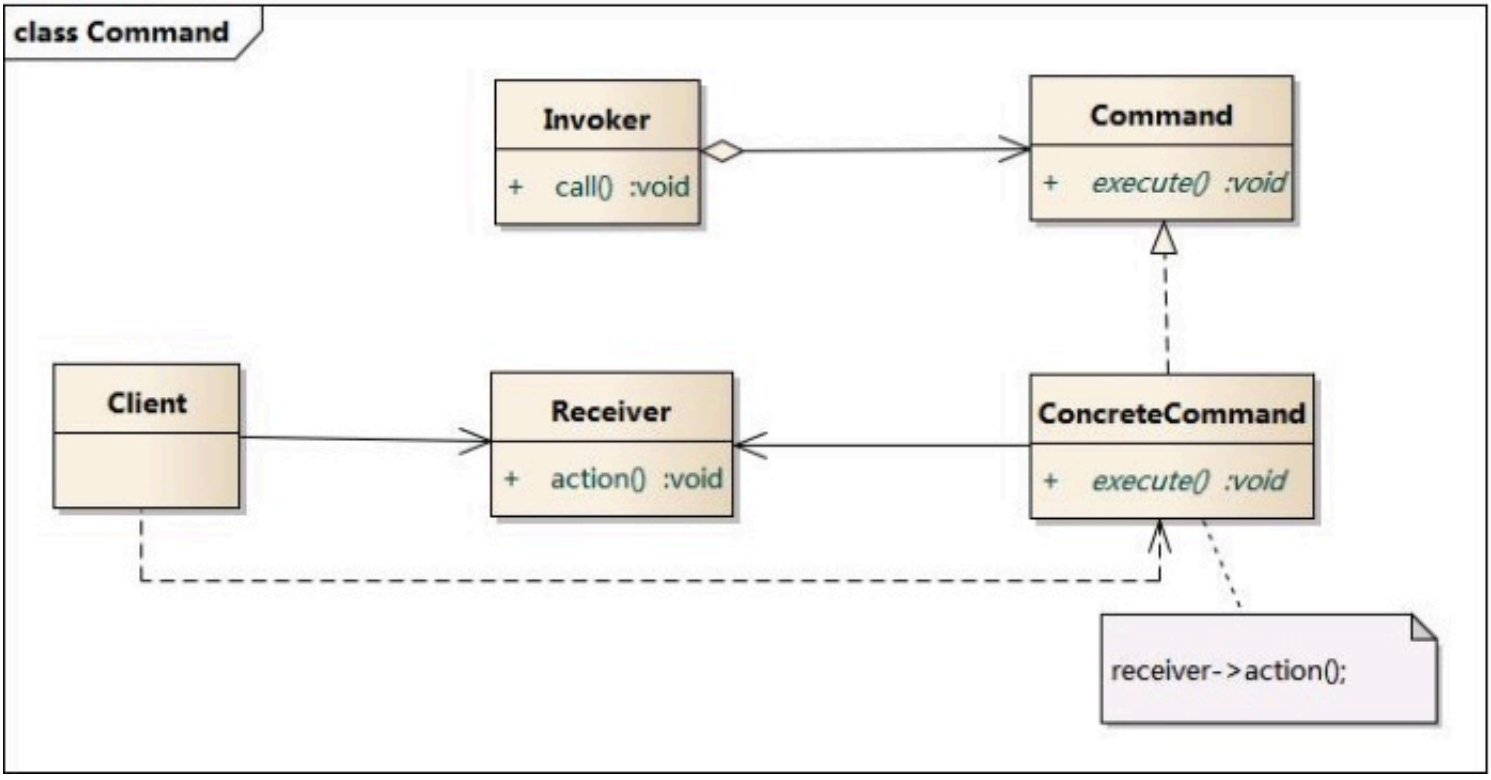
1.模式动机
在软件设计中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，我们只需在程序运行时指定具体的请求接收者即可，此时，可以使用命令模式来进行设计，使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

命令模式可以对发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。这就是命令模式的模式动机。

2.模式定义
命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。

3.模式结构
命令模式包含如下角色：

Command: 抽象命令类
ConcreteCommand: 具体命令类
Invoker: 调用者
Receiver: 接收者
Client:客户类



4.优点
命令模式的优点

降低系统的耦合度。

新的命令可以很容易地加入到系统中。
可以比较容易地设计一个命令队列和宏命令（组合命令）。
可以方便地实现对请求的Undo和Redo。
5.缺点
命令模式的缺点

使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。
6.样例代码
接收者角色类

```
  public class Receiver {
    /**
     * 真正执行命令相应的操作
     */
    public void action(){
        System.out.println("执行操作");
    }
}
```

抽象命令角色类

```
public interface Command {
    /**
     * 执行方法
     */
    void execute();
}
```

具体命令角色类

```
public class ConcreteCommand implements Command {
    //持有相应的接收者对象
    private Receiver receiver = null;
    /**
     * 构造方法
     */
    public ConcreteCommand(Receiver receiver){
        this.receiver = receiver;
    }
    @Override
    public void execute() {
        //通常会转调接收者对象的相应方法，让接收者来真正执行功能
        receiver.action();
    }

}
```

请求者角色类

```
public class Invoker {
    /**
     * 持有命令对象
     */
    private Command command = null;
    /**
     * 构造方法
     */
    public Invoker(Command command){
        this.command = command;
    }
    /**
     * 行动方法
     */
    public void action(){

        command.execute();
    }
}
```

客户端角色类

```
public class Client {

    public static void main(String[] args) {
        //创建接收者
        Receiver receiver = new Receiver();
        //创建命令对象，设定它的接收者
        Command command = new ConcreteCommand(receiver);
        //创建请求者，把命令对象设置进去
        Invoker invoker = new Invoker(command);
        //执行方法
        invoker.action();
    }

}
```