# 什么是设计模式

设计模式(Design Pattern)是一套被反复使用、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可重用代码、让代码更容易被他人理解并且保证代码可靠性。

设计模式一般包含模式名称、问题、目的、解决方案、效果等组成要素，其中关键要素是模式名称、问题、解决方案和效果

# **设计模式的作用**

* 复用解决方案
* 保证代码的可靠性
* 大多数设计模式还能使软件更容易修改和维护。

# **设计模式的原则**

设计模式有7大原则：

* 单一职责原则 (Single Responsibility Principle, SRP) ：一个类只负责一个功能领域中的相应职责，当然，我们可以进行变形，让方法职责单一；
* 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，说白了，接口应该尽可能的小。
* 依赖倒转原则 (Dependence Inversion Principle, DIP)：抽象不应该依赖于细节，细节应该依赖于抽象 ，强调的是面向接口编程；
* 里氏代换原则 (Liskov Substitution Principle, LSP) ：所有引用基类对象的地方能够透明地使用其子类的对象，LSP和继承的关系非常密切；
* 开闭原则 (Open-Closed Principle, OCP) ： 软件实体应对扩展开放，而对修改关闭。
* 合成复用原则 (Composite Reuse Principle, CRP) ：尽量使用对象组合，而不是继承来达到复用的目的
* 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，其中门面模式就很好的展现了迪米特法则；

# **设计模式分类**

## 创建型模式（6个）

* 简单工厂模式
* 工厂方法模式
* 抽象工厂模式
* 单例模式
* 原型模式
* 建造者模式

## 结构型模式（7个）

* 适配器模式
* 桥接模式
* 组合模式
* 装饰模式
* 外观模式
* 享元模式
* 代理模式

## 行为型模式（11个）

* 责任链模式
* 命令模式
* 解释器模式
* 迭代器模式
* 中介者模式
* 备忘录模式
* 观察者模式
* 状态模式
* 策略模式
* 模板方法模式
* 访问者模式

# **简单工厂模式**

## 定义

简单工厂模式(Simple Factory Pattern)：定义一个工厂类，它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。因为在简单工厂模式中用于创建实例的方法是 静态(static)方法，因此简单工厂模式又被称为静态工厂方法(Static Factory Method)模式。简单工厂模式属于创建型模式。

## 类图

### 标准类图：

为了让工厂能够统一生产对象，这些对象通常会有一个公共的父类，这个类多半被定义成为抽象类或者是接口（当然这个不是必须的），如下Product，然后让各个具体的类去具体实现或者继承Product。



### 简化类图

为了简化简单工厂，可以将抽象的产品类和工厂类合并在一起，形成如下比较简单的方式，由于工厂需要被实例化，否则不能使用，所以Product不再是抽象类或者接口。



## 核心代码

### 标准模式

|  |
| --- |
| 工厂类：  public class *ProductFactory* {   public static *Product* createProduct(*String arg*) {  if ("A".equals(*arg*)) {  return new ProductA();  } else if ("B".equals(*arg*)) {  return new ProductB();  } else {  throw new RuntimeException("参数错误");  }  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Product* product = *ProductFactory*.createProduct("A");  *System*.***out***.println(product.getName());  } } |

### 简化模式

|  |
| --- |
| 工厂类（产品父类）：  public class *Product* {   /\*\*  \* 需要子类重写的方法  \* @return  \*/  protected *String* getName() {  return null;  }    /\*\*  \* 工厂方法  \* @param arg  \* @return  \*/  public static *Product* createProduct(*String arg*) {  if ("A".equals(*arg*)) {  return new ProductA();  } else if ("B".equals(*arg*)) {  return new ProductB();  } else {  throw new RuntimeException("参数错误");  }  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Product* product = *Product*.createProduct("A");  *System*.***out***.println(product.getName());  } } |

## 总结

### 优点

* 工厂类包含必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的职责，而仅仅“消费”产品，简单工厂模式实现了对象创建和使用的分离。
* 客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以在一定程度减少使用者的记忆量。
* 可以通过配置文件参数的方式，在不修改代码的情况下传入arg，从而灵活的构建出Product对象；

### 缺点

* 由于工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都要受到影响。
* 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。
* 简单工厂模式由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。
* 对于工厂类而已，每增加一种产品，就需要更改逻辑违背了开闭原则

### 适用场景

* 当需要根据不同的参数创建不同的对象，且他们有共同的抽象方法，只是具体表现行为不同的时候；
* 创建的对象相对较少，这样不会让工厂方法业务过于复杂；

## 开源项目案例

# **工厂方法模式**

## 定义

工厂方法模式(Factory Method Pattern)：定义一个用于创建对象的接口，让子类决定将哪一个 类实例化。工厂方法模式让一个类的实例化延迟到其子类。工厂方法模式又简称为工厂模式 (Factory Pattern)，又可称作虚拟构造器模式(Virtual Constructor Pattern)或多态工厂模式 (Polymorphic Factory Pattern)，工厂模式属于创建新型模式。

工厂方法模式提供一个抽象工厂接口来声明抽象工厂方法，而由其子类来具体实现工厂方法，创建具体的产品对象。

## 类图



## 核心代码

|  |
| --- |
| 抽象工厂接口：  public interface *LoggerFactory* {  *Logger* createLogger(); } |
| 具体工厂类：  public class *FileLoggerFactory* implements *LoggerFactory*{  @Override  public *Logger* createLogger() {  return new FileLogger();  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {   // 可以通过XML文件来更改具体的工厂类  *LoggerFactory* loggerFactory = new FileLoggerFactory();   *Logger* logger = loggerFactory.createLogger();   logger.writeLog("这是内容");  } } |

## 总结

### 优点

* 客户端只需要关心工厂类抽象和产品抽象，不需要关心具体的实现细节；
* 当有新的产品加入的时候，无需修改抽象工厂类和抽象产品类以及客户端，扩展性强；

### 缺点

* 每新增加一种产品，就需要新增一个工厂类，会导致类的数量很多；

### 适用场景

同一个抽象的实现不是很多，且相对固定，对于不同的场景，可能选择不同的实现的时候，比较适合使用工厂方法模式；

## 开源项目案例

# **抽象工厂模式**

## 定义

抽象工厂模式(Abstract Factory Pattern)：提供一个创建一系列相关或相互依赖对象的接口，而

无须指定它们具体的类。抽象工厂模式又称为Kit模式，抽象工厂模式属于创建型模式。

其实说白了，就是对工厂方法模式的一种优化，可以减少工厂类的数据；

同一个工厂类创建出来的对象被称为同一个产品族的；

## 类图



## 核心代码

|  |
| --- |
| 抽象工厂：  public interface *SkinFactory* {   *Button* createButton();   *TextField* createTextField();   *ComboBox* createComboBox(); } |
| 具体工厂：  public class *SpringSkinFactory* implements *SkinFactory* {  @Override  public *Button* createButton() {  return new SpringButton();  }   @Override  public *TextField* createTextField() {  return new SpringTextField();  }   @Override  public *ComboBox* createComboBox() {  return new SpringComboBox();  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *SkinFactory* skinFactory = new SpringSkinFactory();   *Button* button = skinFactory.createButton();   button.displayButton();  } } |

## 总结

### 优点

* 抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建；
* 当新增同一个产品族的时候，无需更改原有抽象；

### 缺点

* 增加新的产品等级结构麻烦，需要对原有系统进行较大的修改，甚至需要修改抽象层代码

### 适用场景

* 当系统有类似于产品族结构的时候，上面类图所示的结构，有多个产品族，每个产品族有多个产品；
* 产品类型相对固定的时候

## 开源项目案例

# **单例模式**

## 定义

单例模式(Singleton Pattern)：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。单例模式属于创建型模式。

单例模式有三个要点：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它 必须自行向整个系统提供这个实例。

## 类图



## 核心代码

单例模式主要有两种模式：饿汉式、懒汉式，写法比较多。

### 饿汉式

|  |
| --- |
| public class *EagerSingleon* {  private static *EagerSingleon* instance = new EagerSingleon();   private EagerSingleon() {  }   public static *EagerSingleon* getInstance() {  return instance;  } } |

这种方式是上来就直接创建，然而，如果这个类用不到的话，那么这个创建是没有意义的；

### 饿汉式变种（静态代码块）

|  |
| --- |
| public class *EagerStaticSingleton* {  private static *EagerStaticSingleton* instance = null;  static{  instance = new EagerStaticSingleton();  }  private EagerStaticSingleton(){};  public static *EagerStaticSingleton* getInstance(){  return instance;  } } |

这种方式和上面的是类似的，在类被加载的时候创建；

### 懒汉式（线程不安全）

|  |
| --- |
| public class *LazySingleton* {  private static *LazySingleton* instance = null;   private LazySingleton() {  }   public static *LazySingleton* getInstance() {  if (instance == null) {  instance = new LazySingleton();  }  return instance;  } } |

把对象的创建延迟到第一次使用的时候，然而，线程不安全，多线程环境下不能使用；

### 懒汉式（线程安全、效率低）

|  |
| --- |
| public class *SafeLazySingleton* {  private static *SafeLazySingleton* instance = null;   private SafeLazySingleton() {  }   public static synchronized *SafeLazySingleton* getInstance() {  if (instance == null) {  instance = new SafeLazySingleton();  }  return instance;  } } |

这种方式通过枷锁的方式让对象的创建，但是创建对象的情况很少，所以，很多时候加锁是没有意义的，效率较低

### 双重校验锁

|  |
| --- |
| public class *DoubleCheckLockSingleton* {  private static *DoubleCheckLockSingleton* instance = null;   private DoubleCheckLockSingleton() {  }   public static *DoubleCheckLockSingleton* getInstance() {  if (instance == null) {  synchronized (*DoubleCheckLockSingleton*.class) {  // 需要再次判断是存在，原因见下面：  if (instance == null) {  instance = new DoubleCheckLockSingleton();  }  }  }  return instance;  } } |

假如在某一瞬间线程A和线程B都在调用getInstance()方法，此时instance对象为null值，均能通 过instance == null的判断。由于实现了synchronized加锁机制，线程A进入synchronized锁定的代 码中执行实例创建代码，线程B处于排队等待状态，必须等待线程A执行完毕后才可以进入 synchronized锁定代码。但当A执行完毕时，线程B并不知道实例已经创建，将继续创建新的实 例，导致产生多个单例对象

### 静态内部类实现

|  |
| --- |
| public class *StaticInnerClassSingleton* {  private static class *SingletonHolder* {  private static final *StaticInnerClassSingleton* ***INSTANCE*** = new StaticInnerClassSingleton();  }   private StaticInnerClassSingleton() {  }   public static final *StaticInnerClassSingleton* getInstance() {  return *SingletonHolder*.***INSTANCE***;  } } |

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，和饿汉式不同的是，饿汉式创建当Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。

### 枚举实现

|  |
| --- |
| public enum *EnumSingleton* {  ***INSTANCE***;  public void doSomething() {  *System*.***out***.println("doSomething");  } } |

这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，使用的时候直接通过Singleton.INSTANCE的方式便可以拿到；

## 总结

### 优点

* 单例模式提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它。
* 由于在系统内存中只存在一个对象，因此可以节约系统资源；

### 缺点

* 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难
* 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色

### 适用场景

* 系统只需要一个实例对象
* 客户调用类的单个实例只允许使用一个公共访问点

## 开源项目案例

# **原型模式**

## 定义

原型模式(Prototype Pattern)：使用原型实例指定创建对象的种类，并且通过拷贝这些 原型创建新的对象。

需要注意的是通过克隆方法所创建的对象是全新的对象，它们在内存中拥有新的地址，通常对克隆所产生的对象进行修改对原型对象不会造成任何影响，每一个克隆对象都是相互独立的。原型模式属于创建型模式

克隆分为两种：深克隆和浅克隆

### 浅克隆

创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址。

### 深克隆

创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址。

总之深浅克隆都会在堆中新分配一块区域，区别在于对象属性引用的对象是否需要进行克隆（递归性的）

## 类图

这里给出深克隆和浅克隆的区别图，其中pos值得是当前对象在内存中的地址。



右面的图中，颜色和左边一样的表示对象地址没有变化的，颜色不同则表示对象地址变化了，可以看到，深度克隆，所有的对象地址都不同了，而浅克隆只有最外层的对象地址不同了，但是里面的对象都还是和之前一样，引用的是相同的地址；

## 核心代码

### 浅克隆

对于浅克隆，我们通常是让这个对象类实现Cloneable接口，调用clone方法（这个方法来自Object），

|  |
| --- |
| 最深层次对象：  @Data public class *Dog* implements *Serializable*, *Cloneable* {   private *String* color;   @Override  public *Dog* clone() {  *Object* clone = null;  try {  clone = super.clone();  } catch (*CloneNotSupportedException e*) {  *e*.printStackTrace();  }  return (*Dog*) clone;  } } |
| 最外层对象：  @Data public class *Person* implements *Serializable*, *Cloneable* {   private *String* name;   private *Son* son;   public *Person* clone() {  *Object* clone = null;  try {  clone = super.clone();  } catch (*CloneNotSupportedException e*) {  *e*.printStackTrace();  }  *Person* result = (*Person*) clone;  return result;  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Dog* dog = new Dog();  dog.setColor("black");   *Son* son = new Son();  son.setName("jim");  son.setDog(dog);   *Person* person = new Person();  person.setName("tom");  person.setSon(son);   *Person* person2 = person.clone();  // 打印true，因为地址一样  *System*.***out***.println(person.getSon() == person2.getSon());  // 打印true，因为地址一样  *System*.***out***.println(person2.getSon().getDog() == person.getSon().getDog());  } } |

实际上，我们只需要Person实现Cloneable接口即可；这里让Son和Dog也实现Cloneable，以及多实现了Serializable接口是为了后面的深克隆；

### 深克隆（利用浅克隆完成）

深克隆有两种方式实现，一种方法是每个对象在克隆的时候，除了克隆基础属性，再调用引用类型的clone方法，以Person为例，需要如下实现：

|  |
| --- |
| public *Person* deepClone1() {  *Object* clone = null;  try {  clone = super.clone();  } catch (*CloneNotSupportedException e*) {  *e*.printStackTrace();  }   // 克隆Son，需要调用它的深克隆方法  *Person* result = (*Person*) clone;  *Son* clonePerson = result.getSon().deepClone1();    result.setSon(clonePerson); // 拷贝子对象  return result; } |

这时候，需要调用deepClone1()方法进行克隆，这种克隆的弊端在于，还需要对Son的colne方法进行改造，如果层次较深，改动很多，如果引用类型的属性很多，这种做法会非常复杂

### 深克隆（字节拷贝）

有一种比较简单的方法完成深克隆，就是使用字节拷贝，这种方式适合任何对象，不过需要类实现序列化接口，但是不需要实现Cloneable 接口

|  |
| --- |
| public *Person* deepClone2() {  try {  //将对象写入到流中  *ByteArrayOutputStream* bao = new ByteArrayOutputStream();  *ObjectOutputStream* oos = new ObjectOutputStream(bao);  oos.writeObject(this);   // 从流中读出对象  *ByteArrayInputStream* bai = new ByteArrayInputStream(bao.toByteArray());  *ObjectInputStream* ois = new ObjectInputStream(bai);  return (*Person*) ois.readObject();   } catch (*Exception e*) {  *e*.printStackTrace();  }  return null; } |

这种方式我们只需要对要拷贝的对象添加对应方法即可

## 总结

### 优点

* 当创建新的对象实例较为复杂时，使用原型模式可以简化对象的创建过程，通过复制一个 已有实例可以提高新实例的创建效率
* 扩展性较好，由于在原型模式中提供了抽象原型类，在客户端可以针对抽象原型类进行编程，而将具体原型类写在配置文件中，增加或减少产品类对原有系统都没有任何影响
* 可以使用深克隆的方式保存对象的状态，使用原型模式将对象复制一份并将其状态保存起 来，以便在需要的时候使用（如恢复到某一历史状态）

### 缺点

* 需要为每一个类配备一个克隆方法，而且该克隆方法位于一个类的内部，当对已有的类进行改造时，需要修改源代码，违背了“开闭原则”

### 适用场景

* 创建新对象成本较大，新的对象可以通过原型模式对已有对象进行复制来获得，如果是相似对象，则可以对 其成员变量稍作修改；
* 需要保存历史状态的场景；

# **建造者模式**

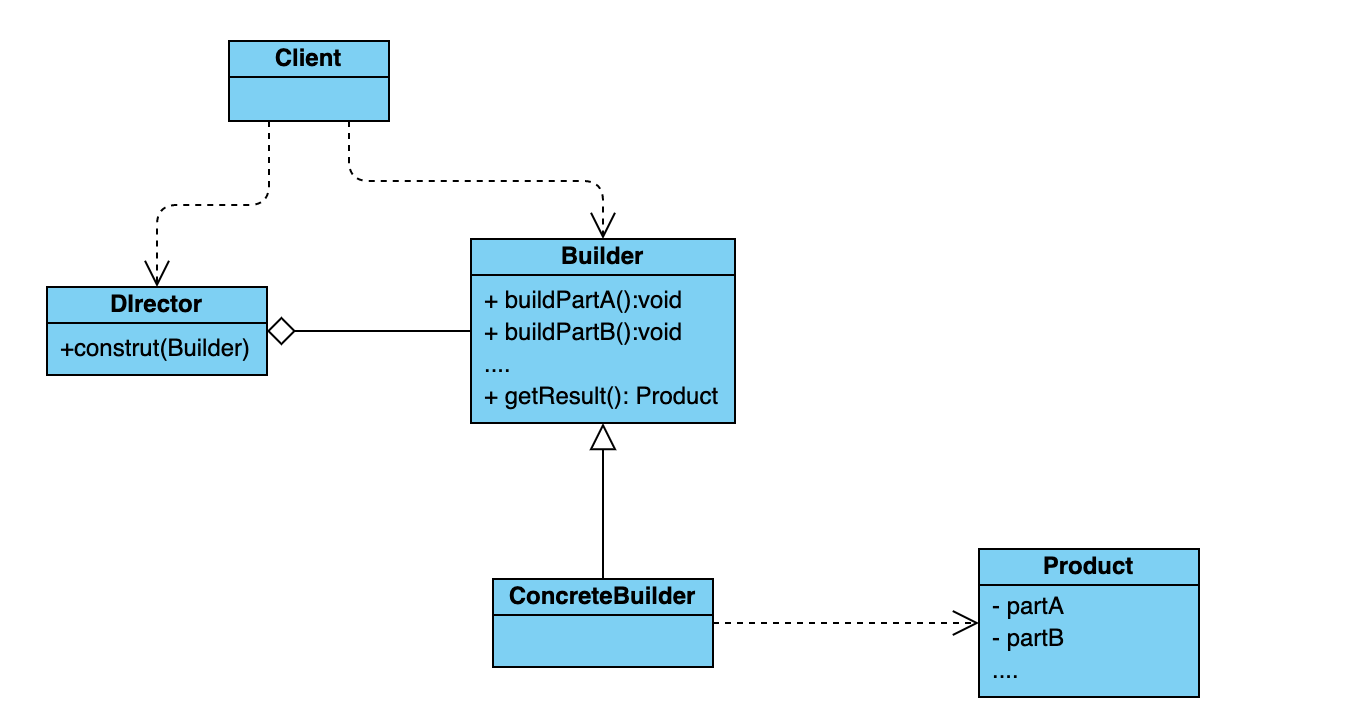
## 定义

建造者模式(Builder Pattern)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示

建造者模式一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于创建型模式。

## 类图

### 标准类图



## 核心代码

### 标准模式

|  |
| --- |
| Builder：  public abstract class *ComputerBuilder* {   protected *Computer* computer = new Computer();    // 下面是各个组成部分的构建方法  public abstract void buildCpu();   public abstract void buildHardDisk();   public abstract void buildMemory();   public abstract void buildDisplayer();   // 返回构建的结果  public *Computer* getResult() {  return computer;  } } |
| Director：  public class *ComputerDirector* {   // 引入构建者  private *ComputerBuilder* computerBuilder;   public ComputerDirector(*ComputerBuilder computerBuilder*) {  this.computerBuilder = *computerBuilder*;  }   // 构建出产品  public *Computer* buildComputer() {  computerBuilder.buildCpu();  computerBuilder.buildDisplayer();  computerBuilder.buildHardDisk();  computerBuilder.buildMemory();  return computerBuilder.getResult();  }    public *ComputerBuilder* getComputerBuilder() {  return computerBuilder;  }   public void setComputerBuilder(*ComputerBuilder computerBuilder*) {  this.computerBuilder = *computerBuilder*;  } } |
| Client：  public class *Client* {   public static void main(*String*[] *args*) {  // 可以通过配置文件的方式灵活更改Builder  *ComputerBuilder* builder = new LOLComputerBuilder();   *ComputerDirector* director = new ComputerDirector(builder);   *Computer* computer = director.buildComputer();   *System*.***out***.println(computer);   } } |

### 省略Director

为了简化结构，可以把指挥者和Builder合二为一，如下：

|  |
| --- |
| Builder：  public abstract class *ComputerDirectorBuilder* {   protected *Computer* computer = new Computer();   // 下面是各个组成部分的构建方法  public abstract void buildCpu();   public abstract void buildHardDisk();   public abstract void buildMemory();   public abstract void buildDisplayer();   // 构建并返回结果  public *Computer* builderComputer(*ComputerDirectorBuilder actorBuilder*) {  *actorBuilder*.buildCpu();  *actorBuilder*.buildDisplayer();  *actorBuilder*.buildHardDisk();  *actorBuilder*.buildMemory();  return computer;  }  } |
| Client：  public class *Client* {   public static void main(*String*[] *args*) {  *ComputerDirectorBuilder* directorBuilder = new LOLComputerDirectorBuilder();   *Computer* computer = directorBuilder.builderComputer(directorBuilder);   *System*.***out***.println(computer);   } } |

更为简单的代码是：

|  |
| --- |
| Builder：  public abstract class *ComputerDirectorBuilder* {   protected *Computer* computer = new Computer();   // 下面是各个组成部分的构建方法  public abstract void buildCpu();   public abstract void buildHardDisk();   public abstract void buildMemory();   public abstract void buildDisplayer();   // 构建并返回结果  public *Computer* builderComputer() {  this.buildCpu();  this.buildDisplayer();  this.buildHardDisk();  this.buildMemory();  return computer;  }  } |
| public class *Client* {   public static void main(*String*[] *args*) {  *ComputerDirectorBuilder* directorBuilder = new LOLComputerDirectorBuilder();   *Computer* computer = directorBuilder.builderComputer();   *System*.***out***.println(computer);   } } |

### 钩子函数

在某些场景下，可能不同的构建者略有不同，比如，某些产品没有某个部分，这时候，我们可以通过钩子函数来进行控制，如：

|  |
| --- |
| Builder：  public abstract class *ComputerDirectorBuilder* {   protected *Computer* computer = new Computer();   // 下面是各个组成部分的构建方法  ......  /\*\*  \* 钩子函数  \*  \* @return  \*/  public boolean needCpu() {  return true;  }   // 构建并返回结果  public *Computer* builderComputer(*ComputerDirectorBuilder actorBuilder*) {  if (needCpu()) {  *actorBuilder*.buildCpu();  }   *actorBuilder*.buildDisplayer();  *actorBuilder*.buildHardDisk();  *actorBuilder*.buildMemory();  return computer;  }  } |
| public class *LOLComputerDirectorBuilder* extends *ComputerDirectorBuilder* {  ....   /\*\*  \* 覆盖钩子函数  \* @return  \*/  @Override  public boolean needCpu() {  return false;  } } |

## 总结

### 优点

* 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解 耦，使得相同的创建过程可以创建不同的产品对象
* 每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体 建造者或增加新的具体建造者

### 缺点

* 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致 系统变得很庞大，增加系统的理解难度和运行成本。

### 使用场景

* 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性
* 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。

## 开源项目案例

# **适配器模式**

## 定义

适配器模式(Adapter Pattern)：将一个接口转换成客户希望的另一个接口，使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。

适配器模式分为如下几种：

* 对象适配器：适配器和适配者之间是关联关系
* 类适配器：适配器和适配者之间是继承（实现）关系
* 双向适配器：适配者和目标对象可以互相调用
* 缺省适配器：解决不需要实现全部接口方法的场景

## 类图

### 对象适配器



这里的Target可以是接口，也可以是抽象类

### 类适配器



由于Adapter要继承Adaptee，所有，Target只能是接口

### 双向适配器



### 缺省适配器



## 核心代码

### 对象适配器

|  |
| --- |
| public class *Adapter* implements *Target* {   private *Adaptee* adaptee;   public Adapter(*Adaptee adaptee*) {  this.adaptee = *adaptee*;  }   @Override  public *String* sayHelloUpCase(*String name*) {  return adaptee.sayHello(*name*).toUpperCase();  } } |

### 类适配器

|  |
| --- |
| public class *Adapter* extends *Adaptee* implements *Target* {   @Override  public *String* sayHelloUpCase(*String name*) {  return sayHello(*name*).toUpperCase();  } } |

要求目标是一个接口。

### 双向适配器

|  |
| --- |
| public class *Adapter* implements *Target*,*Adaptee* {  private *Adaptee* adaptee;  private *Target* target;   public Adapter(*Adaptee adaptee*) {  this.adaptee = *adaptee*;  }   public Adapter(*Target target*) {  this.target = *target*;  }   @Override  public *String* sayHello(*String name*) {  return target.sayHelloUpCase(*name*).toLowerCase();  }   @Override  public *String* sayHelloUpCase(*String name*) {  return adaptee.sayHello(*name*).toUpperCase();  } } |

### 缺省适配器

其实就是定义一个抽象类对接口的方法（全部或者部分）给出默认实现，用户可以通过继承这个抽象类从而减少实现不必要的接口

## 总结

### 优点

* 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，无须修改原有结构
* 增加了类的透明性和复用性，将具体的业务实现过程封装在适配者类中，对于客户端类而言是透明的，而且提高了适配者的复用性

### 缺点

* 由于JAVA的单继承特性，一次最多只能适配一个适配者类，不能同时适配多个适配者
* 类适配器模式下，适配者类不能为最终类，目标类只能是接口
* 对象适配器的实现上比类适配器要麻烦一些；

### 使用场景

* 系统需要使用一些现有的类，而这些类的接口（如方法名）不符合系统的需要，甚至没有这些类的源代码。
* 想创建一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作
* 接口方法过多，而我们只想实现其中的某几个方法的时候，可以使用缺省适配器