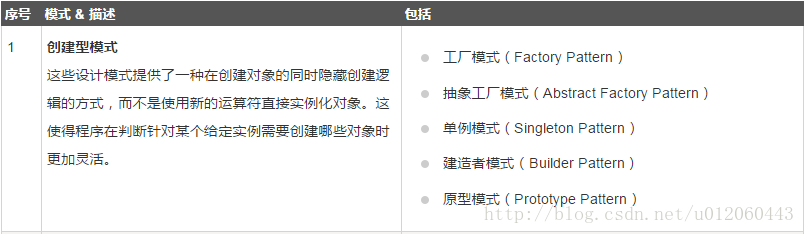
设计模式类型：总共有 23 种设计模式。

这些模式可以分为三大类：

创建型模式（Creational Patterns、

结构型模式（Structural Patterns）

行为型模式（Behavioral Patterns）







而我们的电子商务系统，常用以下几种模式：

代理模式：给一个对象创建一个代理对象，通过代理对象可以使用该对象的功能。

案例：根据文件类型，将文件存储到不同服务。

享元模式

定义：运用共享技术来有效地支持大量细粒度对象的复用。它通过共享已经存在的对象来大幅度减少需要创建的对象数量、避免大量相似类的开销，从而提高系统资源的利用率。

与单例的区别：

单例是对象只能自己创建自己，整个应用中只有1个对象

享元模式根据需要共享，不限制被谁创建（有可能有多个对象实例）

优点：特定环境下，相同对象只要保存一份，这降低了系统中对象的数量，从而降低了系统中细粒度对象给内存带来的压力。

缺点：为了使对象可以共享，需要将一些不能共享的状态外部化，这将增加程序的复杂性。

享元组件逻辑操作对象： SupplementSource

SupplementSource 该对象主要用于给当前线程填充共享数据，以及变更访问方法和访问信息等信息的逻辑操作

public class SupplementSource extends LogComponent{

/\*\*\*\*

\* 填充参数

\* @param username

\*/

public SupplementSource(String username, String sex, String role) {

super(username, sex, role);

}

/\*\*\*\*

\* 业务逻辑，完善不同方法的日志记录

\* @param args 长度为2，第1个是方法名字，第2个是方日志信息

\*/

@Override

void supplementLogContent(String... args) {

super.setMethodName(args[0]);

super.setMessage(args[1]);

}

}

装饰者模式

定义：动态的向一个现有的对象添加新的功能，同时又不改变其结构。它属于结构型模式。

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。

订单价格结算分析

@Component(value = "orderMoneySum")

public class OrderMoneySum implements MoneySum {

@Autowired

private ItemDao itemDao;

//总金额计算

@Override

public void sum(Order order) {

//商品单价\*总数量

Item item = itemDao.findById(order.getItemId());

order.setPaymoney(item.getPrice() \* order.getNum());

order.setMoney(item.getPrice() \* order.getNum());

}

}

策略模式

定义：策略模式是对算法的包装，把使用算法的责任和算法本身分隔开，委派给不同的对象管理。

简单来说就是就定义一个策略接口，子类策略去实现该接口去定义不同的策略。然后定义一个环境（Context，也就是需要用到策略的对象）类，以策略接口作为成员变量，根据环境来使用具体的策略。

优点：

1、算法可以自由切换。

2、避免使用多重条件判断。

3、扩展性良好。

缺点：

1、策略类会增多。

2、所有策略类都需要对外暴露。

工厂模式

定义：定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂类当中。这满足创建型模式中所要求的“创建与使用相分离”的特点。

支付渠道获取工厂创建：创建PayFactory 用于获取支付渠道的实例，我们这里通过映射的key获取Spring容器中实例的id值，然后从Spring容器中根据id获取对应实例，因此该工厂需要实现接口ApplicationContextAware 来获取容器。

@Data

@Component

@ConfigurationProperties(prefix = "pay")

public class PayFactory implements ApplicationContextAware{

//Spring容器

private static ApplicationContext applicationContext;

//支付键值对信息

private Map<String,String> paymap;

/\*\*\*

\* 创建支付通道，从paymap中获取对应通道的实例名字，从applicationContext获取通道实例

\*/

public PayChannel createChannel(String key){

return applicationContext.getBean(paymap.get(key),PayChannel.class);

}

/\*\*\*

\* 获取容器

\* @param applicationContext

\* @throws BeansException

\*/

@Override

public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {

PayFactory.applicationContext = applicationContext;

}

}

状态模式

定义：对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为。

优点：

1、封装了转换规则。

2、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。

3、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。

缺点：

1、状态模式的使用必然会增加系统类和对象的个数。

2、状态模式对"开闭原则"的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代

码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

发通知消息行为定义： SendMsgBehavior 用于实现给商家发送消息通知发货,这里模拟发送消息的行为

@Component("sendMsgBehavior")

public class SendMsgBehavior implements State {

@Override

public void doAction(Order order) {

System.out.println("订单支付");

order.setState(this);

}

@Override

public void execute(){

System.out.println("订单变更为已支付，需要通知商家发货！");

}

}