# 状态模式简介

状态模式用于解决系统中复杂对象的状态转换以及不同状态下行为的封装问题。当系统中的某个对象存在多个状态，这些状态之间可以进行转换，而且对象在不同状态下行为不相同时可以适应状态模式。

## 概述

1. “人有悲欢离合，月有阴晴圆缺”，包括人在内，很多事物都具有多种状态，而且在不同状态下会具有不同的行为，这些状态在特定条件下还将发生相互转换。就像水，它可以凝固成冰，也可以受热蒸发后变成水蒸气，水可以流动，冰可以雕刻，蒸汽可以扩散。在软件系统中，有些对象也像水一样具有多种状态，这些状态在某些情况下能够相互转换，而且对象在不同的状态下也将具有不同的行为。虽然可以使用复杂的条件判断语句（例如if…else语句）来进行状态的判断和转换操作，但这会导致代码的可维护性和灵活性下降，特别是在出现新的状态时代码的扩展性很差，客户端代码也需要进行相应的修改，违反了开闭原则。为了解决状态的转换问题，并使客户端代码与对象之间的耦合度降低，可以适应一种被称为状态模式的设计模式。
2. 状态模式用于解决系统中复杂对象（具有多种状态的对象）的状态转换以及不同状态下行为的封装问题。当系统中某个对象存在多个状态，这些状态之间可以进行转换，而且对象在不同状态下行为不相同时可以使用状态模式。状态模式将一个对象的状态从该对象中分离出来，封装到专门的状态类中，使得对象状态可以灵活变化。对于客户端而言，无须关心对象状态的转换以及对象所处的当前状态，无论对于何种状态的对象，客户端都可以一致处理。
3. 状态模式（State Pattern）的定义如下：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。
4. 状态模式又名状态对象（Object for States），它是一种对象行为型模式。

## 结构

在状态模式中引入了抽象状态类和具体状态类，它们是状态模式的核心，其结构如下图所示：



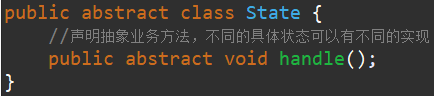
由图可知，状态模式包含以下3个角色：

1. Context（环境类）：环境类也称为上下文类，是拥有多种状态的对象。由于环境类的状态存在多样性且在不同状态下对象的行为有所不同，因此将状态独立出去形成单独的状态类。在环境类只维护一个抽象状态类State的实例，这个实例定义当前状态，在具体实现时它是一个State类型的子类对象。
2. State（抽象状态类）：它用于定义一个接口以封装与环境类的一个特定状态相关的行为，在抽象状态类中声明了各种不同状态对应的方法，而在其子类中实现这些方法，由于不同状态下对象的行为可能不同，因此在不同子类中方法的实现可能存在不同，相同的方法可以写在抽象状态类中。
3. ConcreteState（具体状态类）：它是抽象状态类的子类，每一个子类实现一个与环境类的一个状态相关的行为，每一个具体状态类对应环境的一个具体状态，不同的具体状态类的行为有所不同。

# 实现

## 实现原理

1. 在状态模式中，将对象在不同状态下的行为封装到不同的状态类中，为了让系统具有更好的灵活性和可扩展性，同时对各状态下的共有行为进行封装，需要对状态进行抽象化，引入了抽象状态类角色，其典型代码如下：



1. 在抽象状态类的子类（即具体状态类）中实现了在抽象状态类中声明的业务方法，不同的具体状态类可以提供完全不同的方法实现。在实际使用时，在一个状态类中可能会包含多个业务方法，如果在具体状态类中某些业务方法的实现完全相同，可以将这些方法移至抽象状态类，实现代码的复用。典型的具体状态类代码如下：



3、环境类维持一个对抽象状态类的引用，通过定义的setState( )方法可以向环境类注入

不同的状态对象，然后在环境类的业务方法中调用状态对象的方法。其典型代码如下：

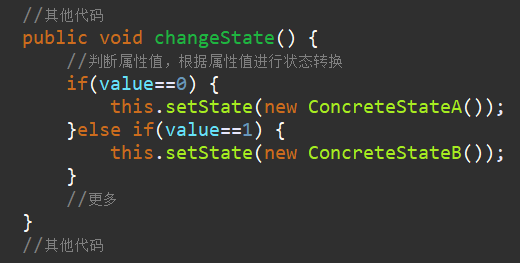


环境类实际上是真正拥有状态的对象，只是将环境类中与状态有关的代码提取出来封装到专门的状态类。在状态模式结构图中，环境类Context与抽象状态类State之间存在依赖或双向关联关系。

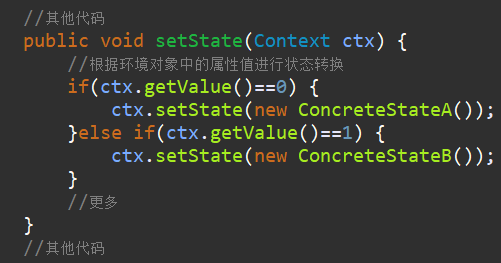
## 状态转换

在状态模式使用过程中，一个对象在状态之间相互转换通常有两种方式：

1. 统一由环境类来负责状态之间的转换，此时环境类还充当了状态管理器（State Manager）角色，在环境类的业务方法中通过对某些属性值的判断实现状态转换，还可以提供一个专门的方法用于实现属性判断和状态转换，代码片段如下：



1. 由具体状态类来负责状态之间的转换，可以在具体状态类的业务方法中判断环境类的某些属性值，再根据情况为环境类设置新的状态对象，实现状态转换。同样，也可以提供一个专门的方法来负责属性值的判断和状态转换。此时状态类与环境类之间存在依赖关联关系，因为状态类需要访问环境类中的属性值，具体状态类ConcreteStateA的代码片段如下：



## 实例

实例说明：某软件公司要为一银行开发一套信用卡业务系统，银行账户（Account）是该系统的核心类之一，通过分析，该软件公司的开发人员发现在系统账户中存在3种状态，且在不同状态下账户存在不同的行为，具体说明如下：

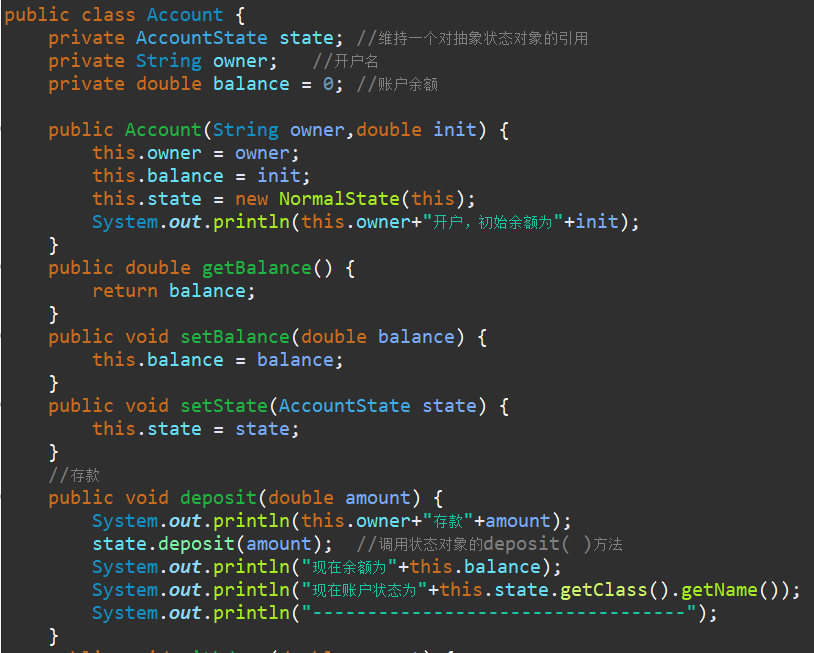
1. 如果账户中的余额大于等于0，则账户的状态为正常状态（Normal State），此时用户既可以向该账户存款也可以从该账户取款。
2. 如果账户中的余额小于0，并且大于-2000，则账户的状态为透支状态（Overdraft State），此时用户既可以向该账户存款也可以从该账户取款，但需要按天计算利息。
3. 如果账户中的余额等于-2000，那么状态为受限状态（Restricted State），此时用户只能向该账户存款，不能从中取款，同时将按天计算利息

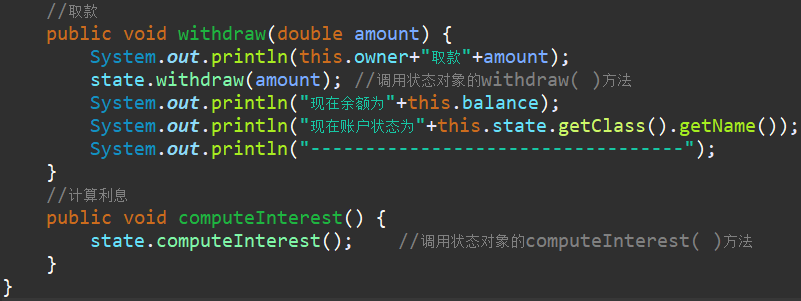
通过对银行账户类进行分析，其结构图如下：



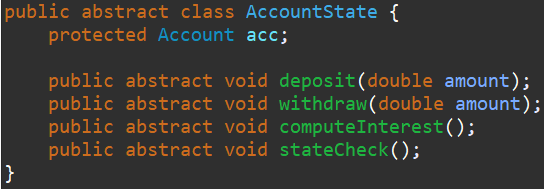
图中，NormalState表示正常状态，OverdraftState表示透支状态，RestrictedState表示受限状态。在这3种状态下账户对象拥有不同的行为，方法deposit( )用于存款，withdraw( )用于取款，computeInterset( )用于计算利息，stateCheck( )用于在每一次执行存款和取款后根据余额来判断是否要进行状态转换并实现状态转换，相同的方法在不同的状态下会有不同的实现。Account充当环境类角色，AccountState充当抽象状态类，NormalState、OverdraftState和RestrictedState充当具体状态角色。

1. Account：银行账户，充当环境类

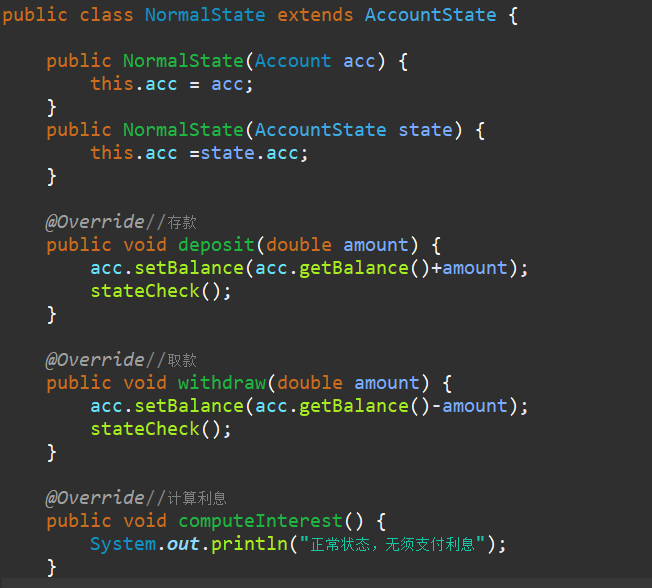


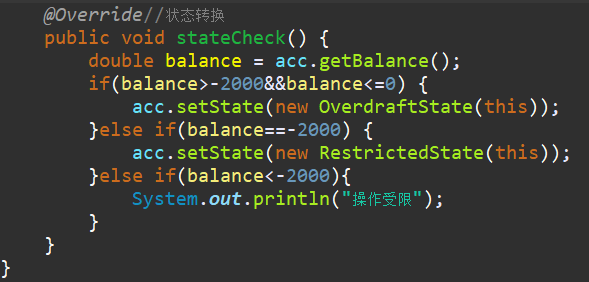


1. AccountState：账户状态类，充当抽象状态类



1. NormalState：正常状态类，充当具体状态类





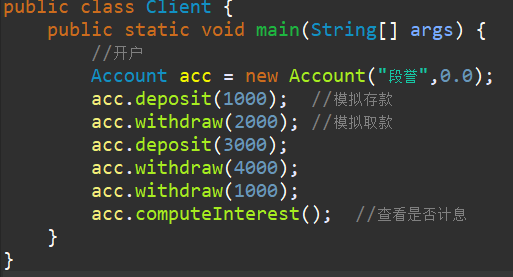
1. OverdraftState：透支状态类，充当具体状态类



1. RestrictedState：受限状态类，充当具体状态类



1. Client：客户端测试类

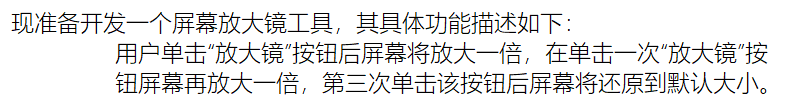


 运行结果如下：

从运行结果可以看出，由于对象状态不一样，因此3次输出结果均有所差异。开户后默认状态为正常状态；第一次取款后账户状态由正常状态（Normal State）变为透支状态（Overdraft State）；第二次取款后账户状态由正常状态（Normal State）变为受限状态（Restricted State）；第三次取款时，由于账户此时已经为受限状态，因此取款失败。这3次取款操作提现了对象在不同状态下具有不同的行为，而且对象的转换是自动的，客户端无须关心其转换细节。

## 使用环境类实现状态转换

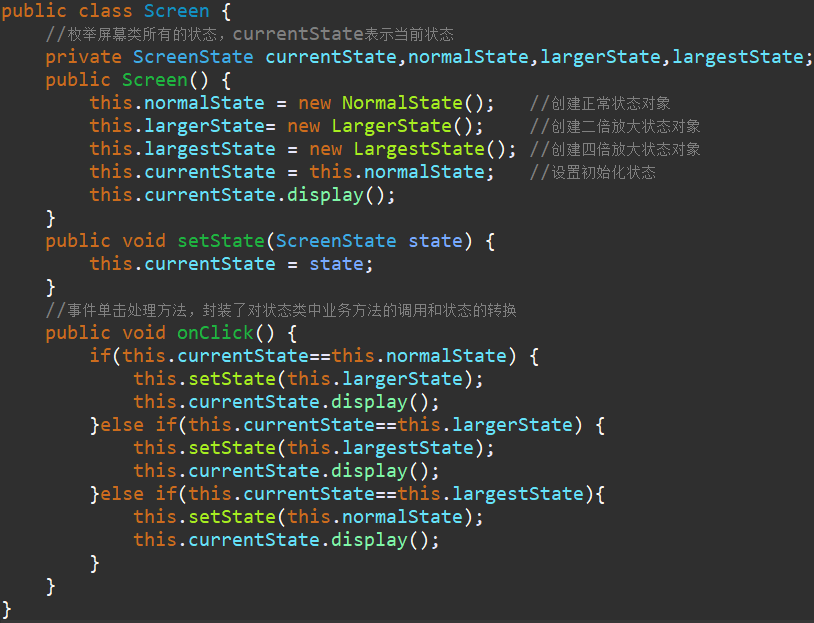
在状态模式中实现状态转换时，具体状态类可通过调用环境类Context的setState( )方法进行状态的转换操作，也可以统一由环境类Context来实现状态的转换。但无论是在环境类中还是在具体状态类中实现状态转换，增加新的具体状态都需要修改环境类或者其他具体状态类，否则系统无法转换到新增状态。但是对于客户端来说无须关心状态类，可以为环境类设置默认的状态类，而将状态的转换工作交给具体状态类或环境类，具体的转换细节对于客户端而言是透明的。上例“银行账户状态转换”中，在每一个具体状态类中都包含一个stateCheck( )方法，在该方法内部实现状态的转换。除此之外还可以通过环境类来实现状态转换，环境类作为一个状态管理器，统一实现各种状态之间的转换操作。下面通过一个包含循环状态的简单实例来说明如何使用环境类实现状态转换：



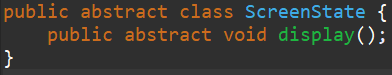
现使用状态模式来设计该屏幕放大镜，通过分析可以定义3个屏幕状态类（NormalState、LargerState和LargestState）来对应屏幕的3种状态，分别是正常状态、二倍放大状态和四倍放大状态，屏幕类充当环境类，其结构如下图所示：



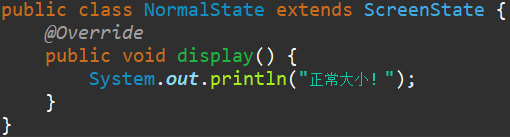
1. Screen：屏幕类，充当环境类



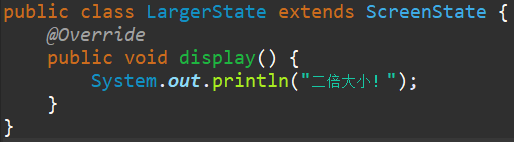
1. ScreenState：屏幕状态类，充当抽象状态类



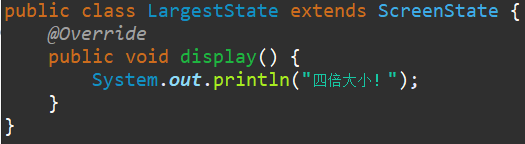
1. NormalState：正常状态类，充当具体状态类



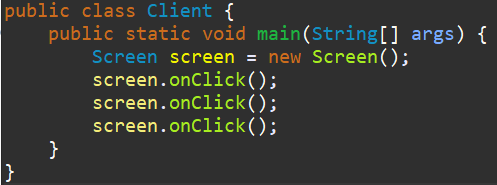
1. LargerState：二倍状态类，充当具体状态类



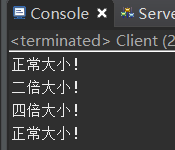
1. LargestState：四倍状态类，充当具体状态类



1. Client：客户端测试类



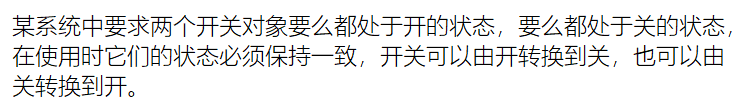
运行结果如下：



该实例中，所有的状态转换操作都由环境类Screen来实现，此时环境类充当了状态管理器角色。需要需要增加新的状态，例如“八倍状态类”，需要修改环境类，这在一定程度上违背了开闭原则，但对其他状态类没有任何影响。

## 共享状态

在有些情况下，同一类型（环境类）的不同对象可能需要共享同一个状态，如果希望在系统中实现多个环境对象共享一个或多个状态对象，那么需要将这些状态对象定义为环境类的静态成员对象。下面通过一个简单实例来说明如何实现共享状态。



使用状态模式来实现开关的设计，其结构图如下：

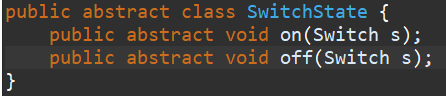


图中，Switch充当环境类，SwitchState充当抽象状态类，OnState充当开状态，OffState充当关状态。

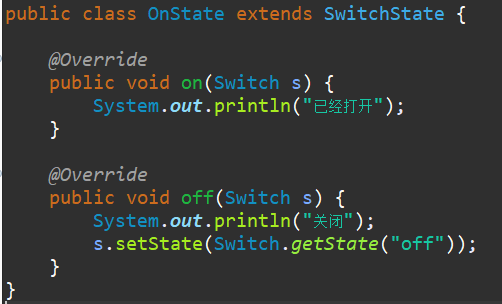
1. Switch：开关类，充当环境类



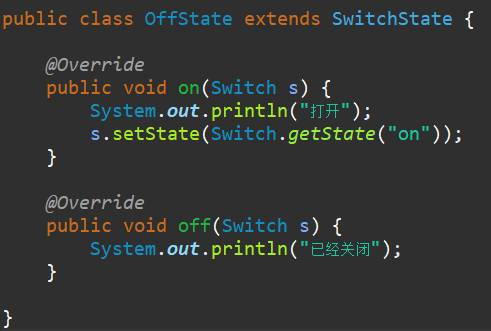
1. SwitchState：开关状态类，充当抽象状态类



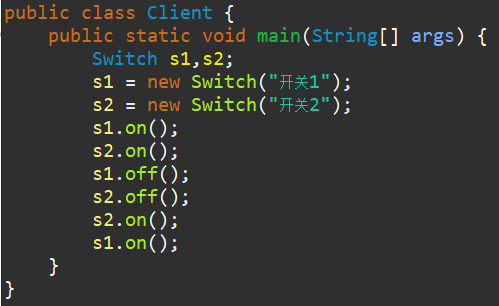
1. OnState：开状态类，充当具体状态类

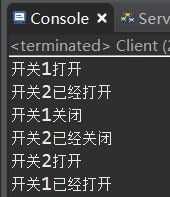


1. OffState：关状态类，充当具体状态类



1. Client：客户端测试类



运行结果如下：

从运行结果可以得知：两个开关状态共享相同的状态，如果第一个开关关闭，则第二个开关也将关闭，再次关闭时将输出“已经关闭”；打开时也将得到类似结果。

# 优缺点及适用环境

状态模式将一个对象在不同状态系的不同行为封装在一个个状态中，通过设置不同的状态对象可以让环境对象拥有不同的行为，而状态转换的细节对于客户端而言是透明的，方便了客户端的使用。在实际开发中，状态模式具有较高的使用频率，在工作流、游戏等软件中状态模式都得到了广泛的应用，例如公文状态的转换、游戏中角色的升级等。

## 优点

状态模式的优点主要如下：

1. 状态模式封装了状态的转换规则，在状态模式中可以将状态的转换代码封装在环境类或者具体状态类中，可以对状态转换代码进行集中管理，而不是分散在一个个业务方法中。
2. 状态模式将所有与某个状态有关的行为放到一个类中，只需要注入一个不同的状态对象即可使环境对象拥有不同的行为。
3. 状态模式允许状态转换逻辑与状态对象合成一体，而不是提供一个巨大的条件语句块，状态模式可以避免使用庞大的条件语句将业务方法和状态转换代码交织在一起。
4. 状态模式可以让多个同一类型的环境类对象共享一个状态对象，从而减少系统中对象的个数。

## 缺点

状态模式的缺点主要如下：

1. 状态模式会增加系统中类和对象的个数，导致系统运行开销很大。
2. 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱，增加系统设计的难度。
3. 状态模式对开闭原则的支持并不太好，增加新的状态类需要修改那些负责状态转换的代码，否则无法转换到新增状态；而且修改某个状态类的行为也需要修改对应类的源代码。

## 适用环境

在以下情况下可以考虑使用状态模式：

1. 对象的行为依赖于它的状态（例如某些属性值），状态的改变将导致行为的变化。
2. 在代码中包含大量与对象状态有关的条件语句，这些条件语句的出现会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，并且导致客户类与类库之间的耦合增强。