状态

状态，在现实环境中，指物质所处的状况，物质处于不同的状态时，表现出的行为也是不同的，比如液态的水，固态的冰，气态的水蒸气。在虚拟环境中同样，有状态的实例在不同状态下表现出的行为也是不同的。

一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做状态，这样的对象叫做有状态的(stateful)对象，这样的对象状态是从事先定义好的一系列值中取出的。当一个这样的对象与外部事件产生互动时，其内部状态就会改变，从而使得系统的行为也随之发生变化。

状态模式

在设计模式中，有一种模式和状态相关，叫状态模式。GOF对状态模式的定义：当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了它的类。状态模式中的行为是由状态来决定的，不同的状态对应了不同的行为。

状态模式的角色：

（1）上下文环境（Context）：它定义了客户程序需要的接口并维护一个具体状态角色的实例，将与状态相关的操作委托给当前的Concrete State对象来处理。

（2）抽象状态角色（State）：定义一个接口以封装使用上下文环境的的一个特定状态相关的行为。

（3）具体状态角色（ConcreteState）：实现抽象状态定义的接口。

状态模式，解决内在状态的改变，状态变化则行为变化，客户端调用时可能会在一个调用过程中出现不同状态而调用不同的行为。

状态模式，通常是自我控制状态的改变（通常由Context类判断状态的变化）。

状态模式的优点：

（1）状态模式将不同状态所对应的行为彼此分隔开来，降低程序的耦合，从而在新增或修改状态时，可以避免程序互相影响。

（2）状态模式将状态的逻辑处理变化交由上下文对象Context管理，便于客户端的调用。

状态模式的缺点：

不同的状态对应不同的类文件，增加了系统文件个数，不便于维护管理。

当一个类A的某个成员变量的值变化时，可能导致多个行为表现得不同。将该成员变量封装成类型的模式，即为状态模式（state pattern）。

状态决定行为

先不考虑状态转换，很容易看到状态决定行为的场景。“朋友来了有好酒，若是那豺狼来了，迎接它的有猎枪”。不在状态，该赢的比赛都会输。

这种Man没有城府，当成员变量isHappy取值不同时，Man的sayHello()和sayGoodbye()等多个行为表现得不同。

在简单场景中，if-else更简洁。然而，关键成员变量可以是int或枚举类型表示的好感度，或某个类型如顾客(土豪、凡人、钓丝)。

随着状态取值可能性的添加，分支块越来越大，sayHello()和sayGoodbye()代码都变得庞大。

特别是增加新的状态值，在分支结构下违反OCP。

从编程技巧上看，状态模式和策略模式、工厂方法模式一样，以多态来重构分支结构。

重构的要点：①以状态类State替代boolean、int、枚举类型或类型(注意这种情况)的分支判断参数，显然有多少分支，State将对应有多少子类。②状态类State将所有具有上述分支判断的方法，提取为自己的接口，State的子类分别给出配套的实现。

在不考虑状态转换时，单纯从代码上看，状态模式的State封装多个方法，而策略模式仅封装一个方法。此时，

状态模式与策略模式的关系，如同抽象工厂与工厂方法的关系。

虽然状态模式封装多个方法，与抽象工厂一样，不会出现[3.2.4方法类型化]的例程3-5那样的组合爆炸，因为多个方法在一个状态下是配套的，而非任意的组合。

yqj2065在设计模式学习难度系数排名中，状态模式给的难度系数高达5分（10分制）。如果仅仅是策略模式基础上的一变多，难度系数应该是1.难过

在Man的代码中是通过设置isHappy的值改变其状态，在Man2中是通过设置State的对象类型改变其状态。状态之间的变换由外界控制，或者说，多种状态是分割的、无关的。这种情况下，学习状态模式的难度系数最多是1。

但是，状态模式最有趣的地方正是其状态的变迁。将涉及一个比较高端的术语Finite State Machine 有限状态机。

状态模式也属于行为类设计模式，它的使用方式跟策略模式颇为相似，首先咱们还是先看下GOF对状态模式的定义：当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。状态模式中的行为是由状态来决定的，不同的状态对应了不同的行为。状态模式和策略模式的结构几乎完全一样，但它们的目的、本质却完全不一样。状态模式的行为是平行的不可替换的，策略模式的行为是彼此独立可相互替换的。

首先看下状态模式的结构图：

wKioL1MQmY7RM8UDAADpmQ9AN4Q008.jpg

状态模式的角色：

（1）上下文环境（Context）：它定义了客户程序需要的接口并维护一个具体状态角色的实例，将与状态相关的操作委托给当前的Concrete State对象来处理。

（2）抽象状态角色（State）：定义一个接口以封装使用上下文环境的的一个特定状态相关的行为。

（3）具体状态角色（ConcreteState）：实现抽象状态定义的接口。

下面看状态模式的具体代码实现：

1、第一个文件IWaterState.java

1

2

3

4

5

6

7

8

9

/\*\*

\* 水的状态

\*/

public interface IWaterState {

/\*\*

\* 输出水的状态

\*/

public void printState();

}

2、第二个文件IceWaterStateImpl.java

1

2

3

4

5

6

7

8

9

10

/\*\*

\* 冰水状态实现类

\*/

public class IceWaterStateImpl implements IWaterState {

@Override

public void printState() {

// TODO Auto-generated method stub

System.out.println("水的状态：冰水！");

}

}

3、第三个文件WarmWaterStateImpl.java

1

2

3

4

5

6

7

8

9

10

/\*\*

\* 温水状态实现类

\*/

public class WarmWaterStateImpl implements IWaterState {

@Override

public void printState() {

// TODO Auto-generated method stub

System.out.println("水的状态：温水！");

}

}

4、第四个文件WasteWaterStateImpl.java

1

2

3

4

5

6

7

8

9

10

/\*\*

\* 沸水状态实现类

\*/

public class WasteWaterStateImpl implements IWaterState {

@Override

public void printState() {

// TODO Auto-generated method stub

System.out.println("水的状态：沸水！");

}

}

5、第五个文件WaterContext.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

/\*\*

\* 状态上下文实现

\*/

public class WaterContext {

/\*\*

\* 状态对象

\*/

private IWaterState state = null;

/\*\*

\* 设置状态对象

\* @param i

\*/

public void setState(int i){

if(i == 0){

state = new IceWaterStateImpl();

System.out.println("正在加热...");

}else if(i == 1){

state = new WarmWaterStateImpl();

System.out.println("正在加热...");

}else if(i == 2){

state = new WasteWaterStateImpl();

System.out.println("加热完成！");

}

}

/\*\*

\* 获得状态对象

\* @return

\*/

public IWaterState getState(){

return state;

}

}

6、第六个文件TestMain.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

/\*\*

\* 测试Main方法

\*/

public class TestMain {

public static void main(String [] args){

IWaterState state = null;

for(int i=0;i<3;i++){

try {

WaterContext context = new WaterContext();

context.setState(i);

state = context.getState();

state.printState();

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

看了如上代码，我们发现状态模式真的和策略模式极为相似，但其实虽然用法上相似，但思想层面上却极为不同。它们所解决的问题不同：状态模式，解决内在状态的改变，状态变化则行为变化，客户端调用时可能会在一个调用过程中出现不同状态而调用不同的行为。策略模式，解决内部算法的改变，客户端选择具体实现算法，选择完后不会再改变。它们解决问题的方法不同：状态模式，通常是自我控制状态的改变（通常由Context类判断状态的变化）。策略模式，由外部指定使用什么样的策略（通常由客户端传入参数，Context根据外界需求生成具体实现对象）。

7、状态模式的适用场景：

状态模式适用于一个对象的行为取决于它的状态, 并且它必须在运行时刻根据状态改变它的行为。

8、状态模式的优点：

（1）状态模式将不同状态所对应的行为彼此分隔开来，降低程序的耦合，从而在新增或修改状态时，可以避免程序互相影响。

（2）状态模式将状态的逻辑处理变化交由上下文对象Context管理，便于客户端的调用。

9、状态模式的缺点：

不同的状态对应不同的类文件，增加了系统文件个数，不便于维护管理。

在阎宏博士的《JAVA与模式》一书中开头是这样描述状态（State）模式的：

　　状态模式，又称状态对象模式（Pattern of Objects for States），状态模式是对象的行为模式。

　　状态模式允许一个对象在其内部状态改变的时候改变其行为。这个对象看上去就像是改变了它的类一样。

状态模式的结构

　　用一句话来表述，状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式的示意性类图如下所示：

状态模式所涉及到的角色有：

　　●　　环境(Context)角色，也成上下文：定义客户端所感兴趣的接口，并且保留一个具体状态类的实例。这个具体状态类的实例给出此环境对象的现有状态。

　　●　　抽象状态(State)角色：定义一个接口，用以封装环境（Context）对象的一个特定的状态所对应的行为。

　　●　　具体状态(ConcreteState)角色：每一个具体状态类都实现了环境（Context）的一个状态所对应的行为。

　　源代码

　　环境角色类

复制代码

public class Context {

//持有一个State类型的对象实例

private State state;

public void setState(State state) {

this.state = state;

}

/\*\*

\* 用户感兴趣的接口方法

\*/

public void request(String sampleParameter) {

//转调state来处理

state.handle(sampleParameter);

}

}

复制代码

　　抽象状态类

复制代码

public interface State {

/\*\*

\* 状态对应的处理

\*/

public void handle(String sampleParameter);

}

复制代码

　　具体状态类

复制代码

public class ConcreteStateA implements State {

@Override

public void handle(String sampleParameter) {

System.out.println("ConcreteStateA handle ：" + sampleParameter);

}

}

复制代码

复制代码

public class ConcreteStateB implements State {

@Override

public void handle(String sampleParameter) {

System.out.println("ConcreteStateB handle ：" + sampleParameter);

}

}

复制代码

　　客户端类

复制代码

public class Client {

public static void main(String[] args){

//创建状态

State state = new ConcreteStateB();

//创建环境

Context context = new Context();

//将状态设置到环境中

context.setState(state);

//请求

context.request("test");

}

}

复制代码

　　从上面可以看出，环境类Context的行为request()是委派给某一个具体状态类的。通过使用多态性原则，可以动态改变环境类Context的属性State的内容，使其从指向一个具体状态类变换到指向另一个具体状态类，从而使环境类的行为request()由不同的具体状态类来执行。

使用场景

　　考虑一个在线投票系统的应用，要实现控制同一个用户只能投一票，如果一个用户反复投票，而且投票次数超过5次，则判定为恶意刷票，要取消该用户投票的资格，当然同时也要取消他所投的票；如果一个用户的投票次数超过8次，将进入黑名单，禁止再登录和使用系统。

　　要使用状态模式实现，首先需要把投票过程的各种状态定义出来，根据以上描述大致分为四种状态：正常投票、反复投票、恶意刷票、进入黑名单。然后创建一个投票管理对象（相当于Context）。

　　系统的结构图如下所示：

　　源代码

　　抽象状态类

复制代码

public interface VoteState {

/\*\*

\* 处理状态对应的行为

\* @param user 投票人

\* @param voteItem 投票项

\* @param voteManager 投票上下文，用来在实现状态对应的功能处理的时候，

\* 可以回调上下文的数据

\*/

public void vote(String user,String voteItem,VoteManager voteManager);

}

复制代码

　　具体状态类——正常投票

复制代码

public class NormalVoteState implements VoteState {

@Override

public void vote(String user, String voteItem, VoteManager voteManager) {

//正常投票，记录到投票记录中

voteManager.getMapVote().put(user, voteItem);

System.out.println("恭喜投票成功");

}

}

复制代码

　　具体状态类——重复投票

复制代码

public class RepeatVoteState implements VoteState {

@Override

public void vote(String user, String voteItem, VoteManager voteManager) {

//重复投票，暂时不做处理

System.out.println("请不要重复投票");

}

}

复制代码

　　具体状态类——恶意刷票

复制代码

public class SpiteVoteState implements VoteState {

@Override

public void vote(String user, String voteItem, VoteManager voteManager) {

// 恶意投票，取消用户的投票资格，并取消投票记录

String str = voteManager.getMapVote().get(user);

if(str != null){

voteManager.getMapVote().remove(user);

}

System.out.println("你有恶意刷屏行为，取消投票资格");

}

}

复制代码

　　具体状态类——黑名单

复制代码

public class BlackVoteState implements VoteState {

@Override

public void vote(String user, String voteItem, VoteManager voteManager) {

//记录黑名单中，禁止登录系统

System.out.println("进入黑名单，将禁止登录和使用本系统");

}

}

复制代码

　　环境类

复制代码

public class VoteManager {

//持有状体处理对象

private VoteState state = null;

//记录用户投票的结果，Map<String,String>对应Map<用户名称，投票的选项>

private Map<String,String> mapVote = new HashMap<String,String>();

//记录用户投票次数，Map<String,Integer>对应Map<用户名称，投票的次数>

private Map<String,Integer> mapVoteCount = new HashMap<String,Integer>();

/\*\*

\* 获取用户投票结果的Map

\*/

public Map<String, String> getMapVote() {

return mapVote;

}

/\*\*

\* 投票

\* @param user 投票人

\* @param voteItem 投票的选项

\*/

public void vote(String user,String voteItem){

//1.为该用户增加投票次数

//从记录中取出该用户已有的投票次数

Integer oldVoteCount = mapVoteCount.get(user);

if(oldVoteCount == null){

oldVoteCount = 0;

}

oldVoteCount += 1;

mapVoteCount.put(user, oldVoteCount);

//2.判断该用户的投票类型，就相当于判断对应的状态

//到底是正常投票、重复投票、恶意投票还是上黑名单的状态

if(oldVoteCount == 1){

state = new NormalVoteState();

}

else if(oldVoteCount > 1 && oldVoteCount < 5){

state = new RepeatVoteState();

}

else if(oldVoteCount >= 5 && oldVoteCount <8){

state = new SpiteVoteState();

}

else if(oldVoteCount > 8){

state = new BlackVoteState();

}

//然后转调状态对象来进行相应的操作

state.vote(user, voteItem, this);

}

}

复制代码

　　客户端类

复制代码

public class Client {

public static void main(String[] args) {

VoteManager vm = new VoteManager();

for(int i=0;i<9;i++){

vm.vote("u1","A");

}

}

}

复制代码

　　运行结果如下：

　　从上面的示例可以看出，状态的转换基本上都是内部行为，主要在状态模式内部来维护。比如对于投票的人员，任何时候他的操作都是投票，但是投票管理对象的处理却不一定一样，会根据投票的次数来判断状态，然后根据状态去选择不同的处理。

认识状态模式

　　●　　状态和行为

　　所谓对象的状态，通常指的就是对象实例的属性的值；而行为指的就是对象的功能，再具体点说，行为大多可以对应到方法上。

　　状态模式的功能就是分离状态的行为，通过维护状态的变化，来调用不同状态对应的不同功能。也就是说，状态和行为是相关联的，它们的关系可以描述为：状态决定行为。

　　由于状态是在运行期被改变的，因此行为也会在运行期根据状态的改变而改变。

　　●　　行为的平行性

　　注意平行线而不是平等性。所谓平行性指的是各个状态的行为所处的层次是一样的，相互独立的、没有关联的，是根据不同的状态来决定到底走平行线的哪一条。行为是不同的，当然对应的实现也是不同的，相互之间是不可替换的。

　　而平等性强调的是可替换性，大家是同一行为的不同描述或实现，因此在同一个行为发生的时候，可以根据条件挑选任意一个实现来进行相应的处理。

　　大家可能会发现状态模式的结构和策略模式的结构完全一样，但是，它们的目的、实现、本质却是完全不一样的。还有行为之间的特性也是状态模式和策略模式一个很重要的区别，状态模式的行为是平行性的，不可相互替换的；而策略模式的行为是平等性的，是可以相互替换的。

　　● 　　环境和状态处理对象

　　在状态模式中，环境(Context)是持有状态的对象，但是环境(Context)自身并不处理跟状态相关的行为，而是把处理状态的功能委托给了状态对应的状态处理类来处理。

　　在具体的状态处理类中经常需要获取环境(Context)自身的数据，甚至在必要的时候会回调环境(Context)的方法，因此，通常将环境(Context)自身当作一个参数传递给具体的状态处理类。

　　客户端一般只和环境(Context)交互。客户端可以用状态对象来配置一个环境(Context)，一旦配置完毕，就不再需要和状态对象打交道了。客户端通常不负责运行期间状态的维护，也不负责决定后续到底使用哪一个具体的状态处理对象。

状态机

一、状态机描述

状态机理论最初的发展在数字电路设计领域。在数字电路方面，根据输出是否与输入信号有关，状态机可以划分为Mealy型和Moore型状态机；根据输出是否与输入信号同步，状态机可以划分为异步和同步状态机。而在软件设计领域，状态机设计的理论俨然已经自成一体。Moore型状态机的输出只和当前状态有关，和输入无关，如果在软件设计领域设计出这种类型的状态机，则该状态机接受的事件都是无内蕴信息的事件（输入）。Mealy型状态机的输入是由当前状态和输入共同决定，对应到软件设计领域，则该状态机接收的事件含有内蕴信息，并且影响状态机的输出。显然，这种划分在软件设计领域毫无意义。虽然软件设计领域的状态机也有同步和异步的划分，但和数字电路方面的同步异步已经不同。

除了《数字电路》，涉及到状态机的课程就是《编译原理》了（本人属计算机专业，其它专业是否涉及到状态机就不清楚了）。下面简单回顾一下《编译原理》里有关有限状态机的描述。在编译原理课程里面，对有限状态机的描述仅限在编译领域，特定状态，针对输入字符，发生状态改变，没有额外的行为，另编译原理里有限状态机的构成要素，还包含唯一的初始状态和一个终态集。数学语言描述如下：一个有限状态机M是一个五元组，M=(K,E,T,S,Z)。其中（1)K是一个有穷集，其中的每个元素称为状态(2)E是一个有穷字母表，它的每个元素称为一个输入字符（3）T是转换函数，是K×E->K上的映射(4)S是K中的元素，是唯一的一个初态（5） Z是K的一个子集，是一个终态集，或者叫结束集。很明显，状态机在编译原理里的讲解已经特化，输入被定位为字符集，状态改变的时候没有额外动作发生。

与编译原理中的状态机不同，软件设计领域中通用状态机的输入不是字符集，而是被称作事件的结构（可以是结构体，也可以是类对象），并且特定的状态下，针对发生的事件，不仅发生状态改变，而且产生动作。借鉴编译原理中状态机的初始状态和终态，通用状态机的数学语言描述如下：一个通用有限状态机M是一个七元组，M={K,E,T,M,F,S,Z}。其中（1）K是一个有穷集，其中的每个元素称为状态(2)E是一个有穷集，它的每个元素称为一个事件（3）T是转换函数，是K×E->K上的映射（4）M是一个有穷集，它的每个元素称为动作（5）F是动作映射函数，是K×E->M上的映射（6)S是K中的元素，是唯一的一个初态（7） Z是K的一个子集，是一个终态集，或者叫结束集。实用的状态机可以做进一步的优化，首先，可以把 （3）（5）整合在一起，做一个K×E->{K,M}的映射，其次从实用性的角度出发，禁止状态接收空事件（无输入的情况下，状态发生改变），作为弥补，为每个状态增加进入动作和离开动作，第三，鉴于定时器在系统中，尤其是在状态机中的重要性，可以为每个状态增加定时器以及超时后的状态转换。本文后面的讲述以及实现暂不考虑把定时器特化，如果需要，可以在状态的进入动作中初始化定时器（另：关于定时器，以后会写文章《系统设计之 定时器》）。

二、状态机分类（后文中如无特别说明，则状态机指软件设计领域的通用有限状态机）

依据状态之间是否有包含关系，分以下两种

（1）常规状态机。状态机中的所有状态是不相交的、互斥的。

（2）层次状态机。状态机中的状态之间要么是互斥的，要么是真包含的，可以用树性结构来描述这些状态集，包含其它状态的状态称为枝节点，不包含其它状态的状态称为叶节点，为方便单树描述，总是设计一个状态包含所有的状态节点，称为根节点。状态机的状态只能停留在叶节点，而不能停留在枝节点，每个枝节点需要指定一个子节点为它的默认子节点，以便状态机进入枝节点的时候能够停留到叶节点。

三、状态机实现

（1）switch/case if/else方式实现。用于少量状态（3个及其以下）的时候，不需要引入专门的状态机模块。这种方式不能编写通用的状态机模块，不再多说。

（2）面向过程方式：宏是实现面向过程方式的通用方式。虽然在状态机层面还是可以用面向对象的方式封装，这里还是把它称为面向过程的方式。

1.常规状态机模块实现。这个状态机涉及到机构由上而下为：

顶层结构是状态机：当前状态id，缺省操作，状态表，

状态表：状态数组

状态结构：状态id，状态名，进入操作，退出操作，缺省操作，状态事件表（数组）

状态事件结构：操作，事件，下一状态的id

状态机的算法是由状态机的结构决定的。实现如下：

一、状态机描述

状态机理论最初的发展在数字电路设计领域。在数字电路方面，根据输出是否与输入信号有关，状态机可以划分为Mealy型和Moore型状态机；根据输出是否与输入信号同步，状态机可以划分为异步和同步状态机。而在软件设计领域，状态机设计的理论俨然已经自成一体。Moore型状态机的输出只和当前状态有关，和输入无关，如果在软件设计领域设计出这种类型的状态机，则该状态机接受的事件都是无内蕴信息的事件（输入）。Mealy型状态机的输入是由当前状态和输入共同决定，对应到软件设计领域，则该状态机接收的事件含有内蕴信息，并且影响状态机的输出。显然，这种划分在软件设计领域毫无意义。虽然软件设计领域的状态机也有同步和异步的划分，但和数字电路方面的同步异步已经不同。

除了《数字电路》，涉及到状态机的课程就是《编译原理》了（本人属计算机专业，其它专业是否涉及到状态机就不清楚了）。下面简单回顾一下《编译原理》里有关有限状态机的描述。在编译原理课程里面，对有限状态机的描述仅限在编译领域，特定状态，针对输入字符，发生状态改变，没有额外的行为，另编译原理里有限状态机的构成要素，还包含唯一的初始状态和一个终态集。数学语言描述如下：一个有限状态机M是一个五元组，M=(K,E,T,S,Z)。其中（1)K是一个有穷集，其中的每个元素称为状态(2)E是一个有穷字母表，它的每个元素称为一个输入字符（3）T是转换函数，是K×E->K上的映射(4)S是K中的元素，是唯一的一个初态（5） Z是K的一个子集，是一个终态集，或者叫结束集。很明显，状态机在编译原理里的讲解已经特化，输入被定位为字符集，状态改变的时候没有额外动作发生。

与编译原理中的状态机不同，软件设计领域中通用状态机的输入不是字符集，而是被称作事件的结构（可以是结构体，也可以是类对象），并且特定的状态下，针对发生的事件，不仅发生状态改变，而且产生动作。借鉴编译原理中状态机的初始状态和终态，通用状态机的数学语言描述如下：一个通用有限状态机M是一个七元组，M={K,E,T,M,F,S,Z}。其中（1）K是一个有穷集，其中的每个元素称为状态(2)E是一个有穷集，它的每个元素称为一个事件（3）T是转换函数，是K×E->K上的映射（4）M是一个有穷集，它的每个元素称为动作（5）F是动作映射函数，是K×E->M上的映射（6)S是K中的元素，是唯一的一个初态（7） Z是K的一个子集，是一个终态集，或者叫结束集。实用的状态机可以做进一步的优化，首先，可以把 （3）（5）整合在一起，做一个K×E->{K,M}的映射，其次从实用性的角度出发，禁止状态接收空事件（无输入的情况下，状态发生改变），作为弥补，为每个状态增加进入动作和离开动作，第三，鉴于定时器在系统中，尤其是在状态机中的重要性，可以为每个状态增加定时器以及超时后的状态转换。本文后面的讲述以及实现暂不考虑把定时器特化，如果需要，可以在状态的进入动作中初始化定时器（另：关于定时器，以后会写文章《系统设计之 定时器》）。

二、状态机分类（后文中如无特别说明，则状态机指软件设计领域的通用有限状态机）

依据状态之间是否有包含关系，分以下两种

（1）常规状态机。状态机中的所有状态是不相交的、互斥的。

（2）层次状态机。状态机中的状态之间要么是互斥的，要么是真包含的，可以用树性结构来描述这些状态集，包含其它状态的状态称为枝节点，不包含其它状态的状态称为叶节点，为方便单树描述，总是设计一个状态包含所有的状态节点，称为根节点。状态机的状态只能停留在叶节点，而不能停留在枝节点，每个枝节点需要指定一个子节点为它的默认子节点，以便状态机进入枝节点的时候能够停留到叶节点。

三、状态机实现

（1）switch/case if/else方式实现。用于少量状态（3个及其以下）的时候，不需要引入专门的状态机模块。这种方式不能编写通用的状态机模块，不再多说。

（2）面向过程方式：宏是实现面向过程方式的通用方式。虽然在状态机层面还是可以用面向对象的方式封装，这里还是把它称为面向过程的方式。

1.常规状态机模块实现。这个状态机涉及到机构由上而下为：

顶层结构是状态机：当前状态id，缺省操作，状态表，

状态表：状态数组

状态结构：状态id，状态名，进入操作，退出操作，缺省操作，状态事件表（数组）

状态事件结构：操作，事件，下一状态的id

状态机的算法是由状态机的结构决定的。实现如下：

以上说明实现原理，有特殊需要的话可以自己定制状态机，比如上面的状态事件表数组的上限取的是单个状态中事件项的最大值，也可以定义为所有事件的个数，这样的话事件也不需要查询，可以象状态样直接定位，只是状态事件表会浪费一些存储空间。上面的FSM\_EVENT仅仅是个例子，实际开发根据需要定义不同的union。上面的算法也是假定状态表的状态定义是从0开始，顺序递增的。

对外部调用而言，最后的状态机结构和事件执行的方法可以封装为对象。下面举例说明状态机的定义(事件和状态都应该是enum类型，这里直接使用数字，仅为说明问题而已)。

三、状态机实现

（2）面向过程方式

2、层次状态机模块实现。

与常规状态机相比，它的FSM\_STATE结构没有default\_func,多了 FSM\_STATE\_ID parent; FSM\_STATE\_ID default\_child;两个结构。状态机初始化的时候可以指定默认状态，为了防止指定的状态非叶结点，增加fsm\_init方法。该状态机的事件处理算法简单描述如下：（1）首先在当前状态以及其祖先状态的状态事件表中搜索匹配事件，如果搜索到，保存操作以及目的状态标识；（2）在old栈中保存当前状态到根节点的路径，在new栈中保存目的状态到根节点的路径；（3）将old栈中的顶层元素依次与new栈的顶层元素匹配，如果匹配则都出栈，不匹配，停止；（4）当前的old栈中节点即为该事件导致的退出状态，从栈低扫描到栈顶，依次执行exit\_func;（5）执行以前保存的操作；（6）扫描new栈，从栈顶到栈低依次执行enter\_func；（7）最后检测目的状态是否是叶节点状态，否，则依次进入default\_child节点，并执行enter\_func。模块实现代码如下：

三、状态机实现

（3）面向对象方式 常规&层次

四、状态机分析

五、状态机回路检测

六、状态机使用

另介绍boost中同步异步状态机

有限状态机

有限状态机（Finite State Machine，FSM）又称有限自动机（Finite Automation，FA），简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的模型。

有穷自动机看上去就像是一个有向图，其中状态是图的节点，而状态转换则是图的边。此外这些状态中还必须有一个初始状态和至少一个接受状态。

状态，存储了关于过去的信息，就是说：它反映从系统开始到现在时刻的输入变化。

转移指示状态变更，并且用必须满足确使转移发生的条件来描述它。

动作是在给定时刻要进行的活动的描述。有多种类型的动作：

进入动作（entry action）：在进入状态时进行

退出动作：在退出状态时进行

输入动作：依赖于当前状态和输入条件进行

转移动作：在进行特定转移时进行

FSM（有限状态机）可以使用状态图（或状态转移图）来表示。此外可以使用多种类型的状态转移表。下面展示最常见的表示：当前状态（B）和条件（Y）的组合指示出下一个状态（C）。完整的动作信息可以只使用脚注来增加。包括完整动作信息的FSM定义可以使用状态表。



有两个不同的群组：接受器／识别器和变换器。

接受器和识别器

接受器和识别器（也叫做序列检测器）产生一个二元输出，说要么“是”要么“否”来回答输入是否被机器接受。所有FSM的状态被称为要么接受要么不接受。在所有输入都被处理了的时候，如果当前状态是接受状态，输入被接受，否则被拒绝。作为规则，输入是符号（字符）；动作不使用。

现在我们来看看有穷自动机怎么处理输入的字符串：

一开始，自动机处于初始状态

输入字符串的第一个字符，这时自动机会查询当前状态上与输入字符相匹配的边，并沿这条边转换到下一个状态。

继续输入下一个字符，重复第二步，查询当前状态上的边并进行状态转换

当字符串全部输入后，如果自动机正好处于接受状态上，就说该自动机接受了这一字符串。

机器还可以被描述为定义了一个语言，它包含了这个机器所接受而非拒绝的所有字词；我们称这个语言被这个机器接受。通过定义，FSM接受的语言是正则语言 - 就是说，如果一个语言被某个FSM接受，那么它是正则的（cf. Kleene的定理）。

开始状态

开始状态通常用“没有起点的箭头”指向它来表示（Sipser (2006）p.34）

接受状态

接受状态是机器成功的进行了它的程序之后的状态，它通常表示为双重圆圈。

接受状态出现的下面确定有限状态自动机例子的状态图的左边，它确定二进制输入是否包含偶数个0: S1（它也是开始状态）指示已经输入了偶数个0的状态因此被定义为接受状态。

变换器

变换器使用动作基于给定输入和／或状态生成输出。它们用于控制应用。常分为两种类型：

Moore机(摩尔型有限状态机)

只使用进入动作的FSM，就是说输出只依赖于状态。Moore模型的好处是行为的简单性。图1的例子展示了一个电梯门的Moore FSM。这个状态机识别两个命令：“command\_open”和“command\_close”触发状态变更。在状态“Opening”中的进入动作 (E:)开启电机开门，在状态“Closing”中的进入动作以反方向开启电机关门。状态“Opened”和“Closed”不进行任何动作。它们信号通知外部世界（比如其他状态机）情况：“门开着”或“门关着”。

Mealy机(米利型有限状态机)

只使用输入动作的FSM，就是说输出依赖于输入和状态。Mealy FSM的使用经常导致状态数目的简约。在图4中的例子展示了实现同上面Moore机同样行为的Mealy FSM（行为依赖于实现的FSM执行模型，比如对虚拟FSM可工作但对事件驱动FSM不行）。有两个输入动作（I:）：“开启电机关门如果command\_close下达”和“反向开启电机开门如果command\_open下达”。

在实践中经常使用混合模型。

进一步可区分为确定型（DFA）和非确定型（NDFA、GNFA）自动机。在确定型自动机中，每个状态对每个可能输入只有精确的一个转移。在非确定型自动机中，给定状态对给定可能输入可以没有或有多于一个转移。这个区分在实践而非理论中更有用，因为存在算法把任何NDFA转换成等价的DFA，尽管这种转换一般会增加自动机的复杂性。

只有一个状态的FSM叫做组合FSM并只使用输入动作。这个概念在多个FSM要一起工作的情况下是有用的，这时把纯组合部分看作一种形式的FSM来适合设计工具可能是方便的。

FSM逻辑

FSM的下一个状态和输出是由输入和当前状态决定的。

数学模型

依据类型不同有多种定义。接受器有限状态机是五元组 (∑, S, s\_0, , F)，这里的：

∑是输入字母表（符号的非空有限集合）。

S是状态的非空有限集合。

s\_0是初始状态，它是S的元素。在非确定有限状态自动机中，s\_0是初始状态的集合。

是状态转移函数：: S × ∑ → S。

F是最终状态的集合，S的（可能为空）子集。

变换器有限状态自动机是六元组(∑, Γ, S, s\_0, , ω)，这里的：

∑a是输入字母表（符号的非空有限集合）。

Γ是输出字母表（符号的非空有限集合）。

S是状态的非空有限集合。

s\_0是初始状态，它是S的元素。在非确定有限状态自动机中，s\_0是初始状态的集合。

是状态转移函数：: S × ∑ → S。

ω是输出函数。

如果输出函数是状态和输入字母表的函数（ω: S × ∑ → Γ），则定义对应于Mealy模型，它可以建模为Mealy机。如果输出函数只依赖于状态 (ω: S →Γ），则定义对应于Moore模型，它可建模为Moore机。根本没有输出函数的有限状态机叫做半自动机或转移系统。

优化

优化一个FSM意味着找到带有极小数目个状态的进行同样功能的机器。一种可能是使用蕴涵表或Moore简约过程。另一种可能是无环FSA的自底向上算法。

确定有限状态自动机

确定有限状态自动机或确定有限自动机（Deterministic Finite Automation, DFA）是一个能实现状态转移的自动机。对于一个给定的属于该自动机的状态和一个属于该自动机字母表∑的字符，它都能根据事先给定的转移函数转移到下一个状态（这个状态可以是先前那个状态）。

确定有限状态自动机\mathcal{A}是由

一个非空有限的状态集合Q

一个输入字母表\Sigma（非空有限的字符集合）

一个转移函数\delta: Q \times \Sigma \rarr Q（例如：\delta \left( q,\sigma \right) = p, \left( p,q \in Q, \sigma \in \Sigma \right)）

一个开始状态s \in Q

一个接受状态的集合F \sube Q

所组成的5-元组。因此一个DFA可以写成这样的形式：\mathcal{A} = \left( Q,\Sigma,\delta,s,F \right)。

工作方式（非正式的语义）

确定有限状态自动机从起始状态开始，一个字符接一个字符地读入一个字符串w \in \Sigma^\*（这里的{}^\*指示Kleene星号算子。），并根据给定的转移函数一步一步地转移至下一个状态。在读完该字符串后，如果该自动机停在一个属于F的接受状态，那么它就接受该字符串，反之则拒绝该字符串。

扩展转移函数

为了在保证严谨的前提下，方便地叙述关于DFA的内容，我们定义如下扩展的转移函数：

\delta^\*: Q \times \Sigma^\* \rarr Q。

\delta^\* \left( q,w \right)是自动机从状态q顺序读入字符串w后达到的状态

扩展转移函数递归的定义为：

\delta^\* \left( q,\epsilon \right) = q

\delta^\* \left( q,u\sigma \right) = \delta(\delta^\*(q,u),\sigma), \forall u \in \Sigma^\*, \sigma \in \Sigma

工作方式（正式的语义）

对于一个确定有限状态自动机\mathcal{A} = \left( Q,\Sigma,\delta,s,F \right)，如果\delta^\* \left( s,w \right) \in F，我们就说该自动机接受字符串w，反之则表明该自动机拒绝字符串w。

被一个确定有限自动机接受的语言（或者叫“被识别的语言”）定义为：\mathcal{L} ( \mathcal{A} ) = \{ w \in \Sigma^\* | \mathcal{A}~接受字符串~w \}，也就是由所有被接受的字符串组成的集合。

DFA与有向图

除了数学上的严谨表述，通常为了讨论方便，也使用状态图直观地表示DFA。不难发现，对于一个给定的DFA，存在唯一一个对应的有向图（但是严格意义上一个有向图不能确定出唯一一个DFA）。有向图的每个结点对应一个状态，每条有向边对应一种转移。习惯上将结点画成两个圈表示接受状态，一个圈表示拒绝状态。用一条没有起点的边指向起始状态。

除了在表述上方便以外，在研究某些问题（如“给定的DFA的语言是否为无穷集合”）时，状态图也提供了有效的解法。

利弊

DFA是一种实际的计算模型，因为有平凡的线性时间、恒定空间的在线算法模拟在输入流上的DFA。给定两个DFA有有效算法找到识别它们所识别语言的并集、交集和补集的DFA。还有有效算法确定一个DFA是否接受任何给定字符串，一个DFA是否接受所有字符串，两个DFA是否识别同样的语言，和对特定正则语言找到状态数目最小的DFA（最小DFA）。

在另一方面，DFA在可识别的语言上有严格的限制—很多简单的语言，包括需要多于恒定空间来解决的任何问题，不能被DFA识别。经典的DFA不能识别的简单语言的例子是括号语言，就是由正确配对的括号组成的语言，比如 (()())。由形如anbn的字符串组成的语言，就是有限数目个a，随后是相等数目个b。可以证明没有DFA有足够状态来识别这种语言（通俗地说，因为需要至少2n个状态，而n是不恒定的）。

其它

能被确定有限状态自动机识别的语言是正则语言。

确定有限状态自动机是非确定有限状态自动机的一种极限形式。

确定有限状态自动机在计算能力上等价于非确定有限状态自动机。

没有接受状态列表并没有指定开始状态的确定有限状态机叫做转移系统或半自动机。

例子

下面是一个确定有限状态自动机的例子。

\mathcal{A}的状态图

确定有限状态自动机\mathcal{A} = \left( Q,\Sigma,\delta,s,F \right)

Q = \{ S\_1 , S\_2 \}

\Sigma = \{0,1\}

s=S\_1

F=\{S\_1\}

\delta由下面的状态转移表定义：

0

1

S1 S2 S1

S2 S1 S2

对应的转移函数为：

\delta(S\_1,0)=S\_2

\delta(S\_1,1)=S\_1

\delta(S\_2,0)=S\_1

\delta(S\_2,1)=S\_2

状态S\_1表示在输入的字符串中有偶数个0，而S\_2表示有奇数个0。在输入中1不改变自动机的状态。当读完输入的字符串的时候，状态将显示输入的字符串是否包含偶数个0。

\mathcal{A}能识别的的语言是\mathcal{L} ( \mathcal{A} ) = \{ w | \#\_0(w) \equiv 0~(mod~2) \}。用正则表达式表示为：(1^\*(01^\*0)^\*)^\*。

封闭性及一些运算

封闭性

确定有限状态自动机的交，并，差，补，连接，替换，同态，逆同态等运算是封闭的，也就是说确定有限状态自动机通过这些运算产生的新的自动机也是确定有限状态自动机。

补运算

\mathcal{A}=(Q,\Sigma,\delta,s,F)是一个DFA，那么由补运算产生的新DFA定义为：\bar{\mathcal{A}}=(Q,\Sigma,\delta,s,Q-F)。显然只要将\mathcal{A}中接受的状态设为不接受的状态，同时把不接受的状态设为接受的状态就得到\bar{\mathcal{A}}。补运算的复杂度是：O(\left| Q \right|)。

交运算和并运算

有两个DFA，\mathcal{A}\_1=(Q\_1,\Sigma,\delta\_1,s\_1,F\_1)和\mathcal{A}\_2=(Q\_2,\Sigma,\delta\_2,s\_2,F\_2)，那么由这两个DFA创造出来的新的自动机定义为：\mathcal{B}= (Q\_1 \times Q\_2,\Sigma,\delta\_\mathcal{B},(s\_1,s\_2),M)。其中M \sube Q\_1 \times Q\_2，\left( s\_1 , s\_2 \right)为\mathcal{B}的开始状态，\delta\_\mathcal{B}为\mathcal{B}的转移函数，且作如下定义：\forall q\_1 \in Q\_1,~q\_2 \in Q\_2,~\sigma \in \Sigma : \delta\_\mathcal{B}((q\_1,q\_2),\sigma) = (\delta\_1 (q\_1,\sigma),\delta\_2 (q\_2,\sigma) )。

当M = F\_1 \times F\_2时，由上述方法得到的\mathcal{B}就是DFA \mathcal{A}\_1和\mathcal{A}\_2的交运算，记作：\mathcal{B} = \mathcal{A}\_1 \cap \mathcal{A}\_2。也就是说对于读入的字符串w，当且仅当\mathcal{A}\_1和\mathcal{A}\_2同时接受w的时候\mathcal{B}接受w。

当M = Q\_1 \times F\_2 \bigcup F\_1 \times Q\_2时，由上述方法得到的\mathcal{B}就是DFA \mathcal{A}\_1和\mathcal{A}\_2的并运算，记作：\mathcal{B} = \mathcal{A}\_1 \cup \mathcal{A}\_2。也就是说对于读入的字符串w，只要\mathcal{A}\_1或\mathcal{A}\_2中至少有一个接受w，\mathcal{B}就接受w。

交运算和并运算的复杂度都是O(\left| Q\_1 \right| \left| Q\_2 \right| \left| \Sigma \right|)。

同态和逆同态运算

一个同态函数h: \Sigma^\* \rarr \Gamma^\*可以递归的定义为：

~h(\epsilon)=\epsilon

~h(u\sigma)=h(u)h(\sigma)

于是则有~h(uv)=h(u)h(v)。（以上所述中~\epsilon为空字符，~u,v \in \Sigma^\* , \sigma \in \Sigma）

\mathcal{L} \sube \Sigma^\* : h(\mathcal{L}) := \{ h(w) ~| ~w \in \mathcal{L} \}：对于接受语言L的DFA，只要将其中代表~\delta(q,\sigma)的边替换成一个序列~h(\sigma)并在其中加入不属于原DFA状态的新状态，就产生了接受语言h(L)的DFA。

\mathcal{L} \sube \Gamma^\* : h^{-1}(\mathcal{L}) := \{ w ~| ~h(w) \in \mathcal{L} \}：定义一个~Q,\Sigma,s,F都不变的新DFA，并定义新的转移函数为~\delta'(q,\sigma) := \delta^\*(q,h(\sigma))，则~(Q,\Sigma,\delta',s,F)就是逆同态运算产生的新DFA。

此外替换运算和逆同态运算的方法近似。

最小自动机

等价类自动机

对于一个正则语言，接受该语言的等价类自动机是一个~(Q,\Sigma,\delta,s,F)的5-元组。其定义如下：

Q是等价关系~L的等价类的集合：[x], x \in \Sigma^\*的集合

~s=[\epsilon]

F = \{ [x] ~| ~x \in L \}

~\delta([x],\sigma) = [x\sigma]

~L被称为Nerode关系，是Myhill-Nerode定理的基础。简单的来说就是对于任意~x,y,z \in \Sigma^\*，如果 xz \in L \Leftrightarrow yz \in L，那么x~Ly。

唯一性

对于任意给定的确定有限状态自动机都能找到一个与之计算能力等价的最小确定有限状态自动机，简称最小自动机。该最小自动机中状态的数量等于能识别相同语言的等价类自动机中等价关系的数量，我们可以称最小自动机和等价类自动机“实际上”是相等的，也就是同构。非正式的说法是：对于最小自动机上的任意状态都可以通过一个同构函数变换成等价类自动机上的一个状态。

能识别一个正则语言的等价类自动机是唯一的，因此能识别该语言的最小自动机也是唯一的。

算法

定义一个非等价关系：N := \{(p,q) ~| ~p,q \in Q, \exists w \in \Sigma^\* : \delta^\*(p,w) \in F \leftrightarrow \delta^\*(q,w) \notin F \}，如下步骤可以得到这个集合N：

如果p \in F, ~q \notin F，就给所有的状态对(p,q)和(q,p)打上标记

重复步骤3，直到所标记的状态对没有变化为止

对于未标记的状态对(p,q)和σ，如果~(\delta(p,\sigma),\delta(q,\sigma))被标记过了就把(p,q)也标记上

以上所有标记了的状态对的集合就是非等价关系N

以下是由一个任意DFA转换到一个最小DFA的步骤：

把所有不能从开始状态达到的状态删除

通过上述标记算法计算非等价关系N

一步一步将不属于N的状态对中的两个状态合成一个状态

这样就得到了接受相同语言的最小自动机。复杂度为O(\left| Q \right| ^2 \left| \Sigma \right|)。

非确定有限状态自动机

非确定有限状态自动机或非确定有限自动机（NFA）是对每个状态和输入符号对可以有多个可能的下一个状态的有限状态自动机。这区别于确定有限状态自动机（DFA），它的下一个可能状态是唯一确定的。尽管DFA和NFA有不同的定义，在形式理论中可以证明它们是等价的；就是说，对于任何给定NFA，都可以构造一个等价的DFA，反之亦然：通过使用幂集构造。两种类型的自动机只识别正则语言。非确定有限自动机有时被称为有限类型的子移位（subshift）。非确定有限状态自动机可推广为概率自动机，它为每个状态转移指派概率。

非确定有限自动机是Michael O. Rabin和Dana Scott在1959年介入的[1]，他们证明了它与确定自动机的等价性。

目录 [隐藏]

1 直观介绍

2 形式定义

3 性质

4 NFA-ε的性质

5 实现

6 例子

7 NFA与DFA

8 NFA-ε的应用

9 引用

9.1 注释

9.2 参考文献

直观介绍

NFA同DFA一样，消耗输入符号的字符串。对每个输入符号它变换到一个新状态直到所有输入符号到被耗尽。

不像DFA，非确定意味着对于任何输入符号，它的下一个状态不是唯一确定的，可以是多个可能状态中的任何一个。因此在形式定义中，一般都谈论状态幂集的子集：转移函数不提供一个单一状态，而是提供所有可能状态的某个子集。

一种扩展的NFA是NFA-λ（也叫做NFA-ε或有ε移动的NFA），它允许到新状态的变换不消耗任何输入符号。例如，如果它处于状态1，下一个输入符号是a，它可以移动到状态2而不消耗任何输入符号，因此就有了歧义：在消耗字母a之前系统是处于状态1还是状态2呢?由于这种歧义性，可以更加方便的谈论系统可以处在的可能状态的集合。因此在消耗字母a之前，NFA-ε可以处于集合{1,2}内的状态中的任何一个。等价的说，你可以想象这个NFA同时处于状态1和状态2:这给出了对幂集构造的非正式提示：等价于这个NFA的DFA被定义为此时处于状态q={1,2}中。不消耗输入符号的到新状态的变换叫做λ转移或ε转移。它们通常标记着希腊字母λ或ε。

接受输入的概念类似于DFA。当最后的输入字符被消耗的时候，NFA接受这个字符串，当且仅当有某个转移集合把它带到一个接受状态。等价的说，它拒绝这个字符串，如果不管应用什么转移，它都不能结束于接受状态。

形式定义

通常定义两种类似类型的NFA: NFA和“有ε-移动的NFA”。普通的NFA被定义为5-元组(Q, Σ, T, q0, F)，它构成自

状态的有限集合Q

输入符号的有限集合Σ

转移函数T : Q×Σ → P(Q)

“初始”（或“开始”）状态q0，有着q0 ∈ Q

“接受”（或“最终”）状态的集合F，有着F ⊆ Q

这里的P(Q)指示Q的幂集。“有ε-移动的NFA”（有时也叫做“NFA-ε”或“NFA-λ”）修订转移函数为允许空串ε作为可能的输入，结果为

T : Q ×(Σ ∪{ε})→ P(Q)。

可以证明普通NFA和有ε移动的NFA是等价的，给定任何一个都可以构造出识别同样语言的另一个。

性质

机器开始于任意初始状态并读取来自它的符号表的符号的字符串。自动机使用状态转移函数T来使用当前状态，和刚读入的符号或空串来确定下一个状态。但是，“NFA的下一个状态不只依赖于当前输入事件，还依赖于任意数目的后续输入事件。直到这些后续事件出现才有可能确定这个机器所处的状态” [2]。如果在自动机完成读取的时候，它处于接受状态，则称NFA接受了这个字符串，否则称为它拒绝了这个字符串。

NFA接受的所有字符串的集合是NFA接受的语言。这个语言是正则语言。

对于所有NFA都可以找到接受同样语言的一个确定有限状态自动机（DFA）。所以出于实现（可能）更简单的机器的目的把现存的NFA转换成DFA是可行的。这是使用幂集构造进行的，这可能导致在必需状态的数目上的指数增长。幂集构造的形式证明在这里给出。

对于有状态的可数无限集合的非确定有限自动机，幂集构造给出有状态的连续统的确定自动机因为可数无限集合的幂集是连续统：2^{\aleph \_{0}}=\aleph \_{1})。在这种情况下，为了使状态转移有意义，状态的集合必须被赋予一个拓扑。这种系统叫做拓扑自动机。

NFA-ε的性质

对于所有p,q\in Q，写p{\stackrel {\epsilon }{\rightarrow }}q当且仅当从p沿着零或更多个\epsilon 箭头前进可到达q。换句话说，p{\stackrel {\epsilon }{\rightarrow }}q当且仅当存在q\_{1},q\_{2},\cdots q\_{k}\in Q这里的k\geq 0使得

q\_{1}\in T(p,\epsilon ),q\_{2}\in T(q\_{1},\epsilon )\cdots ,q\_{k}\in T(q\_{k-1},\epsilon ),q\in T(q\_{k},\epsilon )。

对于任何p\in Q，从p可到达的状态的集合叫做p的ε-闭包，并写为

\,E(\{p\})=\{q\in Q:p{\stackrel {\epsilon }{\rightarrow }}q\}。

对于P\subset Q的任何子集，定义P的ε-闭包为

E(P)=\bigcup \limits \_{p\in P}E(\{p\})。

ε-转移是传递的，这可以证明，对于所有q\_{0},q\_{1},q\_{2}\in Q和P\subset Q，如果q\_{1}\in E(\{q\_{0}\})且q\_{2}\in E(\{q\_{1}\})，则q\_{2}\in E(\{q\_{0}\})。

类似的，如果q\_{1}\in E(P)且q\_{2}\in E(\{q\_{1}\})则q\_{2}\in E(P)

设x是字母表Σ上的字符串。一个NFA-ε M接受字符串x，如果存在x的形如x1x2 ... xn的表示，这里的xi ∈(Σ ∪{ε})，和状态序列 p0,p1, ..., pn二者，这里的pi ∈ Q，满足下列条件：

p0 \in E({q0})

pi \in E(T(pi-1, xi ))对于i = 1, ..., n

pn \in F。

特别注意某些字母xi可以是{ε}；它们不是选择自单独的Σ，而是来自Σ ∪{ε}。

实现

有多种方式实现NFA:

转换成等价的DFA。在某些情况下这导致在自动机大小上的指数爆炸，从而辅助空间正比于在NFA中状态的数目（因为状态值存储要求给在NFA中的每个状态最多一位）。

保持机器当前可能处在的所有状态的集合数据结构。在消耗最后一个输入符号的时候，如果这些状态之一是最终状态，则机器接受这个字符串。在最坏的情况下，这要求辅助空间正比于在NFA中状态的数目；如果集合结构为每个NFA状态使用一位，则这个方式等价于前者。

建立多个复件。对于每个n路决定，NFA建立这个机器的直到n-1个复件。每个都进入单独的状态。如果在耗尽最后的输入符号的时候，至少一个NFA复件处在接受状态，则NFA就接受它。（这也要求关于NFA状态数目的线性存储，因为对所有NFA状态都可能有一个机器）。

通过NFA的转移结构明确的传播（propagate）记号（token）并在一个记号到达最终状态的时候匹配。这在NFA应当编码触发转移的事件的额外上下文的时候是有用的。（对使用这种技术来跟踪对象引用的实现可查看Tracematches）。

例子

下面的例子展示一个NFA M，带有二进制字母表，它确定输入是否包含偶数个0或偶数个1。设M =(Q, Σ, T, s0, F)这里的

Σ = {0, 1},

Q = {s0, s1, s2, s3, s4},

E({s0}) = { s0, s1, s3 }

F = {s1, s3}，而

转移函数T定义自下列状态转移表：

0

1

ε

S0

{}

{}

{S1, S3}

S1

{S2}

{S1}

{}

S2

{S1}

{S2}

{}

S3

{S3}

{S4}

{}

S4

{S4}

{S3}

{}

M的状态图是：

NFAexample.svg

M可以被看作两个DFA的并集：一个有状态{S2, S1}而另一个有状态{S3, S4}。

M的语言可以描述为如下正则表达式给出的正则语言：

(1^{\*}(01^{\*}01^{\*})^{\*})\cup (0^{\*}(10^{\*}10^{\*})^{\*})

NFA与DFA

NFA与确定有限自机动（DFA，或简称FA）的辨识能力是一样的，因而两者是等价的。每个FA也可以写成NFA的形式，只要把转换函式由\delta (q\_{n-1},x)=q\_{n}改写成\delta (q\_{n-1},x)=\{q\_{n}\}就可以，即是与之等价的NFA的转换函数的输出结果即是FA的转换函数的输出结果的单元集。反之，对任何NFA M=(Q, Σ, δ, q0, F)来说，如果它可以接受语言L，则必定存在某个FA M1=(Q1, Σ, δ1, q1, F1)，也可以接受L。可以从“状态”的定义下手“消除”NFA的不确定性。NFA的转换函数的输出结果本为状态集合Q的子集合，现在把这一个子集合当成一个状态看待，即是FA M1中的状态是NFA中状态集合的子集合。这技巧叫做子集合的建构（subset construction）。即是

Q\_{1}=2^{Q}{\text{ and }}q\_{1}=\{q\_{0}\}{\text{ for }}q\in Q\_{1}{\text{ and }}a\in \Sigma ,\delta \_{1}(q,a)=\bigcup \_{p\in q}\delta (p,a)

F\_{1}=\{q\in Q\_{1}|q\cap F\neq 0\}

NFA-ε的应用

NFA和DFA是等价的，如果一个语言可被一个NFA识别，则它也可以被一个DFA识别，反之亦然。这种等价性的建立是重要和有用的。有用是因为构造识别给定语言的NFA有时比构造这个语言的DFA要容易很多。重要是因为NFA可以用来减少建立计算理论中很多重要性质的数学工作的复杂性。例如，使用NFA比使用DFA证明下列性质要更加容易：

（i）两个正则语言的并集是正则的。

（ii）两个正则语言的串接是正则的。

（iii）一个正则语言的Kleene闭包是正则的。

让我用有限状态机（FSM）来举个例子，在FSM中就包含了逻辑的分离和抽象，它有“状态”这个概念，这就是一个逻辑块，它的逻辑块也可以重用，但它对于逻辑的关联做的相对比较薄弱，由状态自己来决定何时跳转，并且跳转比较随意，所以逻辑的关联性比较模糊，这就导致FSM在多状态的情况下很难维护。所以后来有了层次化的有限状态机（HFSM），部分解决了逻辑关联模糊的问题，但FSM的设计原理导致它并没有办法从根本上解决问题。但对于状态和跳转都不是很复杂的功能，FSM是个不错的选择。

对于FSM，每个节点表示一个状态

在简明 状态模式（5.8）中，状态之间的变换由外界控制，或者说，多种状态是分割的、无关的。状态模式最有趣的地方正是讨论其状态的变迁。

1.引子

空调(air-condition)的遥控器有两个按钮(更多的按钮奋斗在后面的例子中引入)，power/电源键和cool/制冷键。空调的运行呈现3个状态，停止/Off、仅送风/FanOnly、制冷/Cool。起始状态为Off，状态变化图如下所示。

这是简化的有限状态机(Finite State Machine、FSM或者Finite State Automata)图形，使用了状态图的3个元素：①气泡，表示状态(state)；②连接状态的箭头表示转换（transition）；③箭头上的标记前者为事件(event)。

状态的转换，看图说话。按power键，则Off→FanOnly、Cool→Off等；按cool，则Off→Off (没有画出来，喜欢全面一点就自己画吧)。

对于这种简单的状态的转换，yqj2065还是喜欢分支语句。微笑，简洁明快。

[java] view plaincopy

例程 4-5 简洁明快

package property.state.stateMachine;

import static tool.Print.\*;//pln

/\*\*

\* 空调Aircon。简单的模型：

\* 遥控器有两个按钮(更多的按钮在下面的例子中引入)，power电源键和cool制冷键。

\* 空调的运行呈现3个状态，停止/Off、仅送风/FanOnly、制冷/Cool。

\* 起始状态为Off

\* @author (yqj2065)

\* @version 0.1

\*/

public class Aircon0{

// off，FanOnly，AC

private int state=0;//起始状态为Off

public int getState(){return state;}

//两个Action

public void power(){//按power键

if(state==0){//off

state=1;

pln("start Fan");

}else if(state==1){//FanOnly

state=0;

pln("stop Fan");

}else{

state=0;

pln("stop Cool");

}

}

public void cool(){//按制冷键

if(state==0){//off

pln("nothing");

}else if(state==1){//FanOnly

state=2;

pln("start Cool");

}else{

state=1;

pln("stop Cool");

}

}

}

package property.state.stateMachine;

public class ACCtrl{

public static void test(){

Aircon0 ac = new Aircon0();//power() cool()

System.out.println("Current State:" + ac.getState());

ac.cool();

ac.power();

ac.cool();

ac.cool();

ac.power();

ac.power();

}

}

测试代码的输出：

Current State:0

nothing

start Fan

start Cool

stop Cool

stop Fan

start Fan

在此基础上，可以花10分钟练习一下，采用状态模式修改上述代码。我们使用enum编写状态类层次。其结构如下：

[java] view plaincopy

例程 4 6 enum State

enum State0{

OFF{

@Override void power(){

}

@Override void power(){

}

},FANONLY{

},

COOL{ };

public abstract void power();

public abstract void cool();

}

(本来是应该将State1作为Aircon1的内部类的。放在外边，power()等需要添加参数Aircon1，变为power(Aircon1 ac)).

现在，丰富有限状态机的细节，增添④动作(action)，如事件(event)相应的动作和状态的动作。

为此，在enum State1中，除了状态模式 提取的接口外，添加了状态机的各种动作/action methode

void entry(Aircon1 ac){pln("→"+ac.state.name());}

void exit(Aircon1 ac){p(ac.state.name()+"→ ");}

void startCool(){ p("start Cool"); }

void stopCool(){ p("stop Cool"); }

void startFan(){ p("start Fan"); }

void stopFan(){ p("stop Fan"); }

每个power(Aircon1 ac)、cool(Aircon1 ac)的方法体结构都是：

this.exit(ac);

//如果有的话，事件(event)相应的动作，如stopFan();

ac.state =OFF; //下一个状态

ac.state.entry(ac);

[java] view plaincopy

package property.state.stateMachine;

import static tool.Print.\*;//pln

/\*\*

\* 本来是应该将State1作为Aircon1的内部类的。现在放在外边，

\* power()等需要变为power(Aircon1 ac)

\*/

public enum State1{

OFF{

@Override void exit(Aircon1 ac){super.exit(ac);startFan();}

@Override void power(Aircon1 ac){

this.exit(ac);

ac.state =FANONLY;

ac.state.entry(ac);

}

@Override void cool(Aircon1 ac){

pln("nothing");

}

},FANONLY{

@Override void power(Aircon1 ac){

this.exit(ac);

stopFan();

ac.state =OFF;

ac.state.entry(ac);

}

@Override void cool(Aircon1 ac){

this.exit(ac);

ac.state =COOL;

ac.state.entry(ac);

}

},

COOL{

@Override void exit(Aircon1 ac){super.exit(ac);stopCool();}

@Override void entry(Aircon1 ac){startCool();super.entry(ac);}

@Override void power(Aircon1 ac){

this.exit(ac);

stopFan();

ac.state =OFF;

ac.state.entry(ac);

}

@Override void cool(Aircon1 ac){

this.exit(ac);

ac.state =FANONLY;

ac.state.entry(ac);

}

};

//状态模式 提取的接口

abstract void power(Aircon1 ac);

abstract void cool(Aircon1 ac);

//状态机的各种动作action methode

void entry(Aircon1 ac){pln("→"+ac.state.name());}

void exit(Aircon1 ac){p(ac.state.name()+"→ ");}

void startCool(){ p("start Cool"); }

void stopCool(){ p("stop Cool"); }

void startFan(){ p("start Fan"); }

void stopFan(){ p("stop Fan"); }

}

空调Aircon1的修改版本。

[java] view plaincopy

package property.state.stateMachine;

import static tool.Print.\*;//pln

/\*\*

\* 空调Aircon1。使用状态模式重构Aircon0，使用enum State1编写状态类层次。

\* @author (yqj2065)

\* @version 0.1

\*/

public class Aircon1{

State1 state= State1.OFF;//private改默认，删除getState()。

//两个Action

public void power(){//按power键

state.power(this);

}

public void cool(){//按制冷键

state.cool(this);

}

/\*\*

\* ACCtrl的代码。

\*/

public static void test(){

Aircon1 ac = new Aircon1();

System.out.println("Current State:" + ac.state.name());

ac.cool();

ac.power();

ac.cool();

ac.cool();

ac.power();

ac.power();

ac.power();

}

}

对应测试操作的输出：“OFF→”表示离开OFF状态，而“→FANONLY”...

Current State:OFF

nothing

OFF→ start Fan→FANONLY

FANONLY→ start Cool→COOL

COOL→ stop Cool→FANONLY

FANONLY→ stop Fan→OFF

OFF→ start Fan→FANONLY

FANONLY→ stop Fan→OFF

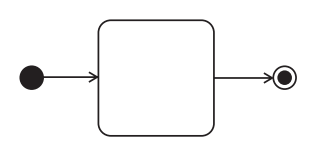
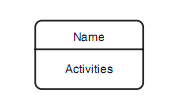
2.分层状态机

对于状态较多的状态机，通常将具有许多公共的特性的状态合并到一起。例如FANONLY和COOL构成的Running状态。

状态机中的hierarchical states，我们可以使用组合模式处理。（还没有单独写组合模式，尴尬）。但是，又不一定能够完美地使用组合模式，例如Running到Off，所有的Running的内部状态在PoverEvent时都转化到OFF，很好；OFF到Running，不是所有Running的内部状态都要调用其entry。在使用enum(不好搞类层次)时，使用责任链吧。

4. 状态图细节

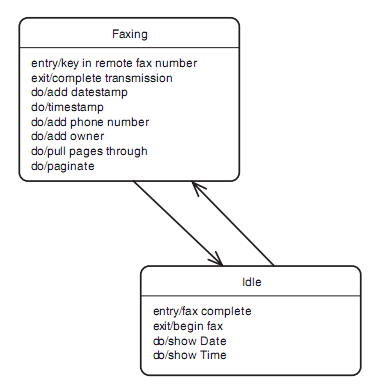
    何谓 State 状态 : Generally speaking, the state of an object depends on the collective value of the object ’ s instance variables. In some cases, most of an object ’ s attributes are fairly static once set, and one attribute is dynamic and plays a prominent role in the class ’ s logic. This attribute may represent the state of the entire object and may even be named state.

    4.1 State

    You can subdivide a state icon into areas that show the state ’ s name and activities活动 .

    3 frequently used categories of activities are entry (what happens when the system enters the state), exit (what happens when the system leaves the state), and do (what happens while the system is in the state).



    4.2 Transition s (Details: Event[Guard Condition]/Action)

    You can also add some details to the transition lines. You can indicate an event that causes a transition to occur (atrigger event ) and the computation (the action ) that executes and makes the state change happen.

    A guard condition : when it’s met, the transition takes place. 通常将超时作为监护条件，∵可以认为此时没有任何event.



•源状态 Source State ：即受转换影响的状态

•目标状态 Target State ：当转换完成后，对象的状态

•触发事件 (Trigger) Event ：用来为转换定义一个事件，包括调用、改变、信号、时间四类事件

•监护条件 (Guard Condition) ：布尔表达式，决定是否激活转换、

•动作 (Action) ：转换激活时的操作

几个实例：

