状态

状态，指在对象的生命周期中的某个条件或者状况，在此期间对象将满足某些条件、执行某些活动活活等待某些事件。所有对象都有状态，状态是对象执行了一系列活动的结果，当某个事件发生后，对象的状态将发生变化。在不同状态下，同一对象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。

行为

行为指的就是对象的功能，大多行为是有对应的方法或者处理的。

行为的平行性和平等性

平行性指的是各个状态下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。

平等性强调的是可替换性，分别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。

如果行为是平行性的，则不可相互替换的；如果行为是平等性的，则是可以相互替换的。

状态决定行为

状态之间可以转换

状态之间的变换由外界控制

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。

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）。

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。

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。在简单场景（≦3个状态）中，switch-case， if-else更简洁。

在状态模式中的多种状态是彼此独立、无关的。

状态机

状态机的概念是来自硬件的。描述一系列状态转换的电路叫状态机。主要用来实现一个数字系统设计中的控制部分。运行模式类似于CPU，但和CPU相比，具有结构简单、易读易懂等特点。

对于无限个状态（无限状态机，Infinite State machine，ISM）是难以检证的，所以这里所说的状态机通常是指有限状态机或有穷状态机，即Finite State Machine，FSM。

状态模式可以允许客户端改变状态的转换行为，而状态机则是能够自动改变状态，状态机是一个比较独立的而且复杂的机制。状态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。但是由于一些原因并不会执行初始化（initialization），而是直接通过一个节点进入状态是允许的，则此节点称之为进入节点（Entry Point）。

退出节点（Exit Point）

转移（Transitions）是两个状态之间的一种关系，表示对象将在源状态（Source State）中，因为预先定义的触发器的发生导致警界条件满足时进入目标状态（Target State）。

触发器（Trigger）：是转移的诱因，可以是一个信号，事件、条件变化（a change in some condition）和时间表达式。

警界条件（Guard Condition）：当警界条件满足时，事件才会引发转移（Transition）。

结果（Effect）：对象状态转移后的结果。

状态可以有返回自身状态的转移，称之为自身转移（Self-Transitions）。

动作（Actions）是一个可执行的原子操作,也就是说动作是不可中断的，其执行时间是可忽略不计的。



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

对象状态转移后的结果显示在转移线上，如果目标状态有许多转移，而且每个转移有相同的结果，这时把转移后的结果（Effect）展示在目标状态中（Target State）更好一些，可以定义进入动作（Entry Action ）和退出动作（Exit Action）

嵌套在另外一个状态中的状态称之为子状态（sub-state）,一个含有子状态的状态被称作组合状态（Compound States）。

历史状态（History States）是一个伪状态（Pseudostate）,其目的是记住从组合状态中退出时所处的子状态，当再次进入组合状态，可直接进入这个子状态，而不是再次从组合状态的初态开始。

并发区域（Concurrent Regions）

状态图可以分为区域，而区域又包括退出或者当前执行的子状态。说明组合状态在某一时刻可以同时达到多个子状态。

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

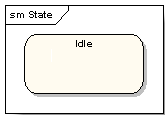
输入动作：依赖于当前状态和输入条件进行

转移动作：在进行特定转移时进行

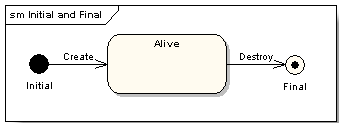
描述状态机的方法有：状态转移图（状态图）、状态转换表、时序图

状态图

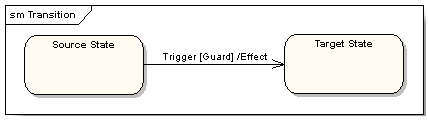
状态用圆角矩形表示。



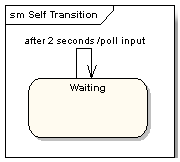
初态（Initial States）用实心圆点表示，终态（Final States）用圆形内嵌圆点表示。



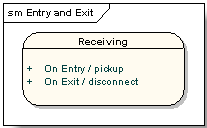
转移（Transitions）



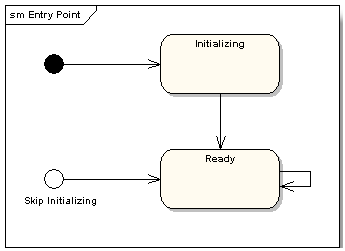
自身转移（Self-Transitions）



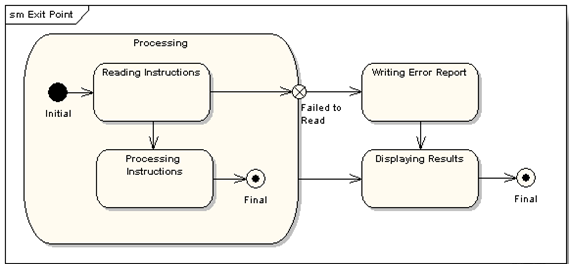
动作（Actions）



进入节点（Entry Point）

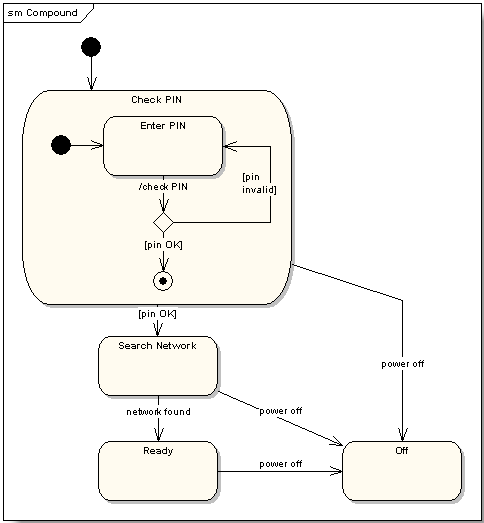


退出节点（Exit Point）

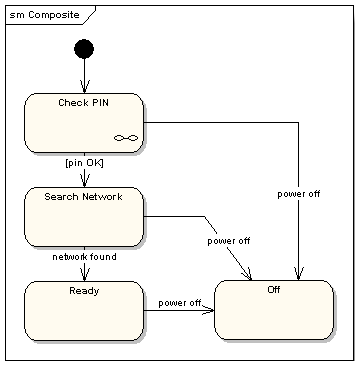


组合状态（Compound States）。

如下图，【Check PIN】是组合状态，【Enter PIN】是子状态。

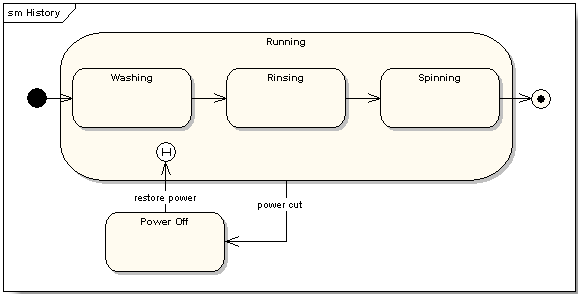


也可用以下方式进行描述



如上图，状态机【Check PIN】的细节被分割到另外一个图中了。

历史状态（History States）

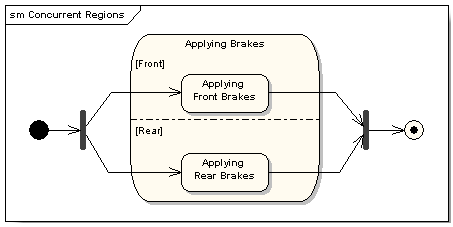


在上图的状态图中，正常的状态顺序是:【Washing】- >【Rinsing】->【Spinning】。

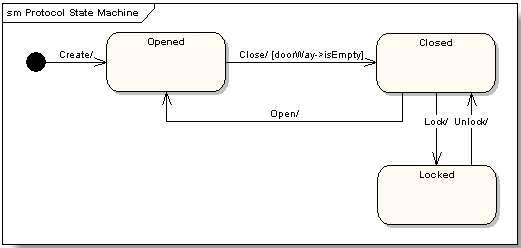
如果是从状态【Rinsing】突然停电（Power Cut）退出，,洗衣机停止工作进入状态【Power Off】，当电力恢复时直接进入状态【Running】。

并发区域（Concurrent Regions）

如下图刹车系统，同时进入前刹车【Applying Front Brakes】状态和后刹车【Applying Rear Brakes】状态。



例子



如图状态机描述了门对象的生存期间的状态序列，引起转移的事件，以及因状态转移而伴随的动作（Action）.

状态有Opened、Closed、Locked。

事件有 Open、Close、Lock和Unlock。

注意：

1、并不是所有的事件都会引起状态的转移，比如当门是处于【Opened】状态，不能进行【Lock】事件。

2、转移（Transition）有警备条件（guard condition），比如只有doorWay->isEmpty 条件满足时，才会响应事件。

状态图重点在于描述对象的状态及其状态之间的转移，状态图的基本元素主要有：状态、转移、动作、自身转移、组合状态、进入节点、退出节点、历史状态、并发区域等，状态中的事件分为调用事件（Call）、变化事件（Change）、时间事件（Time）和信号事件(Singal)。

状态转换表



当前状态（B）和条件（Y）的组合指示出下一个状态（C）。

状态机的设计步骤：

步骤1 分析设计要求，列出状态机所有状态，对所有状态进行编码

步骤2 设计状态机的状态图

步骤3 用硬件描述语言描述状态机

a.定义信号

b.状态转换进程

c.状态存储进程

d.选择描述方式

有限状态机有两个不同的群组：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型状态机的输出只和当前状态有关，和输入无关。Moore模型的好处是行为的简单性。比如一个电梯门的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型状态机的输入是由当前状态和输入共同决定。Mealy FSM的使用经常导致状态数目的简约。比如一个电梯门的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。

FSM的下一个状态和输出是由输入和当前状态决定的。

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。

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）。

能被确定有限状态机识别的语言是正则语言。

确定有限状态机是非确定有限状态机的一种极限形式。

确定有限状态机在计算能力上等价于非确定有限状态机。

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。

非确定有限状态机可以转化为确定有限状态机。

非确定有限状态机接受正则语言。

有限状态机的缺点

每一种有限状态机均功能唯一，即设计好之后无法完成其他原理不同的工作；

因为其状态有限，当所要描述的系统的状态太多时，可能确定的有限状态机无能为力；

有一些任务是有限状态机无法完成的，比如它可以判断输入的0、1数列中0或1的个数是否为奇数或偶数，但是无法判断0是否比1多或者相反。

非正统

过于底层

逻辑受限

需要自定义扩展

难以标准化

非自主的

难以并发

大规模支持较差

劳动密集型

行业进步

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。

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。

curEvent = getEvent();

curState = getCurState();

switch(curState) {

case state1: {

switch(curEvent ) {

// TODO...

setCurState();

break;

}

break;

}

...

}

（2）面向过程方式：

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。

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。

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：

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。

（3）面向对象方式 常规&层次

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|)。

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。

一般都用switch/case if/else方式实现。在少量状态（3个及其以下）的时候，不需要引入专门的状态机模块。

常规状态机模块实现涉及到的结构由上而下为：

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

通常FSM包含几个要素：状态的管理、状态的监控、状态的触发、状态触发后引发的动作。

基于表结构的状态机设计方法：建立相应的状态表和动作查询表，根据状态表、事件、动作表定位相应的动作处理函数，执行完成后再进行状态的切换。

对于状态较多的状态机，通常的设计会维护一个庞大的二维矩阵，所有状态耦合在一起，这往往导致维护困难，由于可能存在许多公共的特性，也会导致许多状态具有相同的处理函数。针对这些问题我们可以通过设计分层状态机来解决，主要的思想就是根据不同的功能模块设计出多个状态机，各个状态机分布在不同的层次上。上层状态机调用下层状态机时，上层状态机入栈，下层状态机变为当前处理状态机。通常我们使用堆栈来保存当前状态机的上层状态机信息。

游戏开发之状态机的实现与优化

引言

你是否还在面对乱作一团的代码束手无策？你是否仍然觉得复杂的逻辑无从下手？你是否觉得游戏AI高端得毫无头绪？本文将以一个复杂的弹窗逻辑和RPG游戏挂机AI的实现为案例，讲述状态机的概念及其写法。

本文分为以下部分：

有限状态机（finite-state machine）：对状态机一些概念的解释。

案例对照：将一个复杂弹窗的普通写法和状态机编程两种实现进行对比。这里状态机的实现是多个if-else的最简单的状态机实现。

有限状态机的优势：通过上面的对比总结状态机的优势。

如何优雅地使用状态机：以游戏挂机自动刷怪的AI为例，提供状态模式的代码实现。

状态机的使用场景：对状态机的使用做了一些扩充。

总结：对本文内容的总结。

参考资料：文中部分概念的来源以及扩展阅读的链接。

对状态机一无所知的读者可以顺序看下去；写了不少逻辑，却依旧编不好繁复代码的，可以从案例对照开始阅读，相信可以让你对编程有个新的把握；会用状态机，却用得不优雅的读者，可以直接空降如何优雅地使用状态机，状态模式的实现在等着你钻研；会用一百种不同的方法花式写状态机的读者，可以直接去看文末的参考资料，希望对你有所帮助~

有限状态机（finite-state machine）

有限状态机，又称有限状态自动机，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数学模型。[1]

有限状态机可以将复杂的逻辑简化为有限个稳定状态，在稳定状态中判断事件。其中有限不是指有限次处理，而是有限个稳定状态，并且有限状态机是一个闭环系统，可以用有限的状态处理无尽的事务。

例如，灯的开关就是一个非常简单的有限状态机。它有两种状态：开或关。这两个状态的切换是通过手指的输入产生的。打开开关，产生从关到开的状态变换；关闭开关，产生从开到关的状态变换。

light

状态机由下列几部分组成：

状态集(States)：包括现态和次态在内的一系列状态，用来描述状态机所处的状态。

事件(Event)：又被称为“条件”，当满足条件时，将会触发一个动作，或者执行一次状态的迁移。

动作(Action)：条件满足后执行的动作。动作执行完毕后，可以迁移到新的状态，也可以仍旧保持原状态。动作不是必需的，当条件满足后，也可以不执行任何动作，直接迁移到新状态。

转换(Transition)：通过转换函数将状态从现态迁移到次态的动作。迁移后次态变为现态。

最著名的有限状态机可能是艾伦·图灵假想的设备——图灵机，他在1936年论文《关于可计算数字》中写道：这是一个预示着现代可编程计算机的机器，它们可以通过对无限长的磁带上的符号进行读写和擦除操作来进行任何逻辑运算。[2]

有限状态机实际上是一个有向图，由状态节点和状态转义函数组成。因此，当游戏策划交给你一个模块的流程图时，完全可以将流程图简化成一个或多个状态图，并进行实现。

案例对照

下面，我讲列举非状态机和状态机编程两种代码进行对比。

当我们写一个弹窗时，需求往往是这样：点击打开按钮，显示弹窗；点击关闭按钮，弹窗消失。这和本文一开始的电灯状态很相似，但这样一个简单的逻辑，并不需要使用复杂的状态机进行控制，我们可以直接对相应的按钮进行事件绑定。

example 1:

1

2

3

4

5

6

7

8

//MainUI.cpp init 函数内

//打开按钮

Button \*openBtn = Button::create();

openBtn-&gt;addClickEventListener([=] (this)

{

MyAlertDialog \*dialog = MyAlertDialog::create();

dialog-&gt;show();

}

1

2

3

4

5

6

7

8

//MyAlertDialog.cpp init 函数内

//关闭按钮

Button \*closeBtn = Button::create();

closeBtn-&gt;addClickEventListener([=] (this)

{

//关闭按钮在弹窗内部

this-&gt;dismiss();

}

但很多时候需求是复杂的，我们需要的弹窗可能是这样：弹窗开启前插入两个动画，动画间有0.5秒延迟，动画播完后1秒打开弹窗，弹窗打开后4s自动关闭或点击关闭按钮关闭，延迟2s后弹窗消失，关闭后主页产生变化。

我们仍不使用状态机编程，最终代码如下：

example 2:

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

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

//MainUI.cpp

bool MainUI::init()

{

...

//打开按钮

Button \*openBtn = Button::create();

openBtn-&gt;addClickEventListener([=] (this)

{

runActionBeforeShowDialog();

}

...

m\_dialog = MyAlertDialog::create();//调整为成员变量进行控制，需在头文件中声明并在构造中置为nullptr

m\_dialog-&gt;setDismissFunc(std::bind(&amp;MainUI::dismissDialog, this));

return true;

}

//动画

void MainUI::runActionBeforeShowDialog()

{

Action \*action1 = SomeAction::create(2.0f);

Action \*action2 = OtherAction::create(1.5f);//第二个动画

CallFunc \*callback = CallFunc::create(std::bind(&amp;MainUI::showDialog, this));

Sequence \*seq = Sequence::create(action, DelayTime::create(0.5f), action2, DelayTime::create(1.f), callback, nullptr);//增加延迟

this-&gt;runAction(seq);

}

//优化：把打开弹窗的代码整理成函数

void MainUI::showDialog()

{

dialog-&gt;show();

}

void MainUI::dismissDialog()

{

dialog-&gt;dismiss();

...do something...

scheduleOnce(...);//关闭弹窗后新的定时器操作

}

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

//MyAlertDialog.cpp

bool MyAlertDialog::init()

{

...

//关闭按钮

Button \*closeBtn = Button::create();

closeBtn-&gt;addClickEventListener([=] (this)

{

//延迟两秒关闭

scheduleOnce(std::bind(&amp;MyAlertDialog::m\_dismissFunc, this), 2.0f);

}

//延迟4s自动关闭,关闭延迟两秒程序员偷懒未做

scheduleOnce(std::bind(&amp;MyAlertDialog::m\_dismissFunc, this), 4.0f);

...

return true;

}

void setDismissFunc(std::function&lt;void()&gt; func)

{

m\_dismissFunc = func;

}

其实，在处理这样的逻辑时，我们已经将不同块的需求整理成了不同的状态，从弹窗打开到关闭无非经历了如下步骤：

开始

点击打开，显示动画

动画结束，延迟1s，显示弹窗

4s后自动开始关闭

点击关闭按钮直接进入关闭

延迟2s，弹窗消失

结束，MainUI处理其他逻辑

但是，由于没有引入状态机，上述代码从清晰简单的弹窗逻辑变成了充斥着回调和定时器的代码堆砌。如果此时流程中出现问题，很难迅速定位，导致整体效率的下降。

根据上述步骤，列出状态表：

当前状态 条件 状态转换

开始 点击开始按钮 显示动画

显示动画 1秒后自动切换 弹窗开

弹窗开 点击关闭或4秒后 弹窗关

弹窗关 2秒后 弹窗消失（结束）

引入状态机来控制逻辑，最简单的写法如下：

example 3:

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

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

//MainUI.cpp

enum class MAINUI\_DIALOG\_STATE = {

READY,

SHOW\_ANIMATION,

OPEN,

CLOSE,

DISMISS,//调用关闭后2s，弹窗才会消失

END,

}

bool MainUI::init()

{

...

m\_state = MAINUI\_DIALOG\_STATE.READY;

m\_timeout = 0;//存储时间间隔，作为延迟的判断条件

//打开按钮

Button \*openBtn = Button::create();

openBtn-&gt;addClickEventListener([=] (this)

{

//runActionBeforeShowDialog();

setState(MAINUI\_DIALOG\_STATE.SHOW\_ANIMATION);//点击打开，展示动画

}

...

m\_dialog = MyAlertDialog::create();//调整为成员变量进行控制，需在头文件中声明并在构造中置为nullptr

m\_dialog-&gt;setDismissFunc(std::bind(&amp;MainUI::dismissDialog, this));

return true;

}

//动画

void MainUI::runActionBeforeShowDialog()

{

Action \*action1 = SomeAction::create(2.0f);

Action \*action2 = OtherAction::create(1.5f);//第二个动画

//CallFunc \*callback = CallFunc::create(std::bind(&amp;MainUI::showDialog, this));不再需要回调

Sequence \*seq = Sequence::create(action, DelayTime::create(0.5f), action2, nullptr);//移除回调和回调前的延迟

this-&gt;runAction(seq);

}

//优化：把打开弹窗的代码整理成函数

void MainUI::showDialog()

{

dialog-&gt;show();

}

void MainUI::dismissDialog()

{

setState(MAINUI\_DIALOG\_STATE.CLOSE);//主动点关闭，状态变为CLOSE

}

void MainUI::update(float dt)

{

m\_timeout += 1;//每次update自加1

//通过当前状态判断是否进入下一状态

if (m\_state == MAINUI\_DIALOG\_STATE.SHOW\_ANIMATION)

{

if (m\_timeout &gt; (4.f + 1.f) \* 60)//cocos2d-x每秒60帧，此处即为1s延迟 + 4s动画时间

{

setState(MAINUI\_DIALOG\_STATE.OPEN);//延迟一秒，打开弹窗

}

}

else if (m\_state == MAINUI\_DIALOG\_STATE.OPEN)

{

if (m\_timeout &gt; 4.f \* 60)//4s后自动关闭

{

setState(MAINUI\_DIALOG\_STATE.CLOSE)

}

}

else if (m\_state == MAINUI\_DIALOG\_STATE.CLOSE)

{

if (m\_timeout &gt; 2.f \* 60)//2s后弹窗消失

{

setState(MAINUI\_DIALOG\_STATE.DISMISS)

}

}

else if (m\_state == MAINUI\_DIALOG\_STATE.DISMISS)

{

setState(MAINUI\_DIALOG\_STATE.END)//状态结束，没有延迟

}

}

void MainUI::setState(MAINUI\_DIALOG\_STATE state)

{

m\_timeout = 0;//进入新状态时，时间间隔清零

if (state == MAINUI\_DIALOG\_STATE.SHOW\_ANIMATION)

{

runActionBeforeShowDialog();

}

else if (state == MAINUI\_DIALOG\_STATE.OPEN)

{

showDialog();

}

else if (state == MAINUI\_DIALOG\_STATE.CLOSE)

{

//do nothing

}

else if (state == MAINUI\_DIALOG\_STATE.DISMISS)

{

//do nothing

m\_dialog-&gt;dismiss();

}

else if (state == MAINUI\_DIALOG\_STATE.END)

{

//状态结束

}

m\_state = state;

}

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

//MyAlertDialog.cpp

bool MyAlertDialog::init()

{

...

//关闭按钮

Button \*closeBtn = Button::create();

closeBtn-&gt;addClickEventListener([=] (this)

{

//延迟两秒关闭

//scheduleOnce(std::bind(&amp;MyAlertDialog::m\_dismissFunc, this), 2.0f);

m\_dismissFunc();//无需在这里处理延迟，调用函数设置关闭状态即可

}

//延迟4s自动关闭,关闭延迟两秒程序员偷懒未做

//scheduleOnce(std::bind(&amp;MyAlertDialog::m\_dismissFunc, this), 4.0f);

//此处延迟已统一由MainUI进行处理

...

return true;

}

void setDismissFunc(std::function&lt;void()&gt; func)

{

m\_dismissFunc = func;

}

通过example 2、3的对比，我们可以看出，使用状态机，不仅让代码更加清晰，而且将逻辑都放在了MainUI处理，包括弹窗的显示和消失，弹窗只关注自身内部的变化，不去对自己进行dismiss的操作，使逻辑解耦。并且在这一过程中任何一个步骤出现问题，都能很快进行定位，并直接对相应状态下的代码进行调整，不会影响其他的状态。

同时我们可以看到状态机的四个部分，首先在枚举中定义了所有的状态，用m\_state表示现态；在update函数和按钮响应事件中设置动作触发的事件；动作触发后执行响应逻辑并通过转移函数进行状态的切换；而setState则是状态的转移函数。

有限状态机的优势

通过上述案例，我们可以得出有限状态机的五个优点：

编程快速简单。编写有限状态机的方法有很多种，并且几乎所有的实现方法都非常简单。本文中将会提供几种状态机的实现方法及其利弊。

易于调试。将游戏逻辑分解成不同的状态，使得问题的定位和修改变得方便。

很少的计算开销。有限状态机几乎不占用珍贵的处理器时间，因为它本质上遵守硬件编码规则，只需要对if-else进行处理。

直觉性。在生活中，人们总是自然地把事物思考为处在一种或另一种状态。“进入状态”、“状态不佳”也是我们常见的。在编码中，将游戏逻辑分解成一系列状态并创建相应的规则去处理是非常容易的。

灵活性。代码增删变得方便快捷。

事实上，在写逻辑的时候已经潜在地使用了状态，只是没有把状态抽象出来，而是直接按流程去编写代码，使用响应、回调的方式做逻辑处理，这样使得在增删流程，后期维护时代码耦合过深，难以维护，最终不得不进行重构。而且当逻辑出现问题时，很难直接定位问题，降低了调试效率。

如何优雅地使用状态机

上述给出的只是最简单的状态机，适合较少状态之间的切换.当逻辑变得庞杂的时候，if-else的逻辑将变成一场噩梦。状态的切换会让我们难以把握程序的现状。往后的扩展也会变得相当困难。

这里我就要向大家介绍，如何优雅地使用状态机。

我们在开发游戏的时候，经常会碰到游戏AI，在编写游戏AI时，我们通常会选择有限状态机。

一般来说，在设计角色、怪物、NPC的时候，很有可能都是继承自同一个基类，此时状态机就不宜写成上面那种格式。我们可以先将状态写成一个抽象类：

1

2

3

4

5

6

7

8

class State

{

public:

virtual ~State() {}

virtual void Enter(Player\*) = 0;

virtual void Execute(Player\*) = 0;

virtual void Exit(Player\*) = 0;

}

这里预留了Enter和Exit的接口，方便做状态切换时的动作。上述三个接口都有一个Player的指针作为传参。这里我不想以简单我怪物的逻辑作为示例来讲解，现在很多RPG类的手游都提供了挂机刷怪的逻辑，点开这个设置，角色就会自动跑到附近的副本里刷怪升级，减轻玩家的负担。

这里我设定一个逻辑，开始挂机时，自动寻找附近副本，刷怪，刷怪需要体力值，体力过低时会自动回城休息，刷怪获得物品占满物品栏时会自动回城贩卖。达到设定要求时挂机停止。如图2所示。

chart2

根据上述条件，我们可以得出Player的类。

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

33

34

Class Player : public BaseGameEntity

{

private:

State\* m\_pCurrentState;

location m\_location;//当前位置

int m\_gold;//金币数

int m\_exp;//经验数

int m\_strength;//体力值

int m\_goods;//物品数

public:

Player(int uid);

void update();

//状态转移函数

void ChangeState(State\* newState);

}

void Player::update()

{

if (m\_pCurrentState)

{

m\_pCurrentState-&gt;Execute(this);

}

}

void Player::ChangeState(State\* newState)

{

//现态退出时的动作

m\_pCurrentState-&gt;Exit(this);

m\_pCurrentState = newState;

//次态进入时的动作

m\_pCurrentState-&gt;Enter(this);

}

通过图2我们可以看到，一共有四个状态：

\* 挂机：将角色移动到副本中，寻找附近的怪物击杀，获取经验和金钱，扣除体力。若经验到达设定值，则停止挂机。

\* 回城休息：角色体力过低，自动移动位置到城里休息。休息完毕回到挂机状态。

\* 回城贩卖：角色背包装满，回城自动贩卖，若金钱到达设定值，则停止挂机，否则回到挂机状态。

\* 结束：挂机过程结束，角色回城。

以挂机状态为例，实现这个状态只需要直接将State类继承过来。

1

2

3

4

5

6

7

8

9

class AutoState : public State

{

public:

AutoState() {}

virtual void Enter(Player\* player);

virtual void Execute(Player\* player);

virtual void Exit(Player\* player);

}

根据逻辑补齐接口：

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

void AutoState::Enter(Player\* player)

{

//寻找副本

player-&gt;ChangeLocation(dungeon);

}

void AutoState::Execute(Player\* player)

{

//认为每次执行就击杀了一个怪物

player-&gt;AddGold(1);

player-&gt;AddExp(1);

player-&gt;AddGoods(1);

player-&gt;DecreaseStrength();

//背包装满，则回城贩卖

if (player-&gt;PocketsFull())

{

player-&gt;ChangeState(new GoBackAndSellState());

}

//体力值过低，则回城休息

if (player-&gt;NeedRest())

{

player-&gt;ChangeState(new GoBackAndRestState());

}

}

void AutoState::Execute(Player\* player)

{

cout &lt;&lt; &quot;\n&quot; &lt;Uid()) &lt;&lt; &quot;: &quot;&lt;&lt; &quot;I&#039;m leaving the dungeon!&quot;;

}

上面的代码简单地讲述了如何使用状态模式编写一段游戏AI，上述的实现方式就是状态模式[3]。为了方便讲解，这里所列举出的状态都是比较独立的，以便于我们对状态机本身的理解和状态模式的把握。

通过这几段代码，和上面example 3作对比，我们可以发现新的写法丢弃了繁重的if-else结构，通过类的继承的方式来实现整个逻辑，这样不仅简化了逻辑的编写，也让我们搭建游戏框架变得更加方便。状态的增删也仅需要新建和移除状态子类即可，十分快捷。

当然，细心的朋友可能发现，我们在每次切换状态的时候都做了一次new的操作，在状态切换频繁的时候会消耗很多资源。这里可以具体问题具体分析，究竟是直接new，还是将子类写成单例，则需要读者根据需求自己把握了。

状态机的使用场景

状态机的使用场景非常广泛，除了上述在游戏中处理UI逻辑和编写游戏AI时需要使用状态机编程以外，还有很多地方会用到状态机。

状态机本身广泛应用于硬件控制电路设计中，比如比较经典的电梯、洗衣机的控制。

软件中如正则表达式[4]、词法分析，网络协议如下图所示的TCP/IP协议[5]等，可以说有限状态机是无处不在的。

chart3

当然，任何编程规范都不宜被滥用。在最初的时候，example 1就已经是比较合适的写法了，没有必要过度追求编程规范，反而会降低开发效率。本文中只是以一个复杂的弹窗（结算动画、中奖提示等类型）讲述状态机的优势，在实际应用场景中，游戏主逻辑、游戏大厅等具有复杂UI交互的类，都可以考虑使用状态机来进行代码编写，细分状态，保证代码的健壮性，方便以后扩展新的特性。

本文侧重游戏开发中的状态机，这里提到的一些使用场景在文末参考资料部分附上了链接，有兴趣的朋友可以进行深入阅读。

总结

在游戏开发中，状态机有利于处理复杂模块的逻辑，降低耦合度，方便扩展特性。

简单实现的状态机会面临if-else过多所造成的难以维护的问题，而状态模式则是实现状态机的最优解法，在细节处仍有不少可优化的地方。

状态机应用广泛，但不宜滥用状态机。

<http://erlang.shiningray.cn/otp-design-principles/gen_fsm.html>

<https://github.com/jakesgordon/javascript-state-machine>

<http://mtydev.net/?p=553>