面向状态编程模式探索

State-Oriented Programming

海贵青
twitter: @garyhai
email: gary@XL59.com

议程

- 问题
- 构建
- SOP vs OOP
- 实践

迁移面临的问题

- 系统工程师的问题
  - 如何在Erlang/OTP平台上进行系统分析和建模？
- 开发人员的问题
  - 如何适应函数式编程模式？
  - 如何理解Erlang系统？
  - 如何理解分布式计算系统？
- 测试人员的问题
  - 重新学习基于Erlang/OTP平台的测试方式和工具。
  - 需要学习Erlang语言及OTP平台。
- 业务人员的问题
  - 如何描述和展示业务逻辑？
  - 如何与研发人员用同一套语言讨论业务逻辑？

需求总结

- 系统分析人员需要一种新的建模工具。
  - 传统的OO模式已经不适合
- 开发人员需要一种编程模板。
  - 降低函数式编程的门槛，减缓冲击
- 测试人员需要一套简洁的测试方法。
  - 不需要深入学习Erlang/OTP平台
- 研发团队需要Erlang及分布式系统的隐喻。
- 业务人员需要一种简单易懂的方式描述业务逻辑。

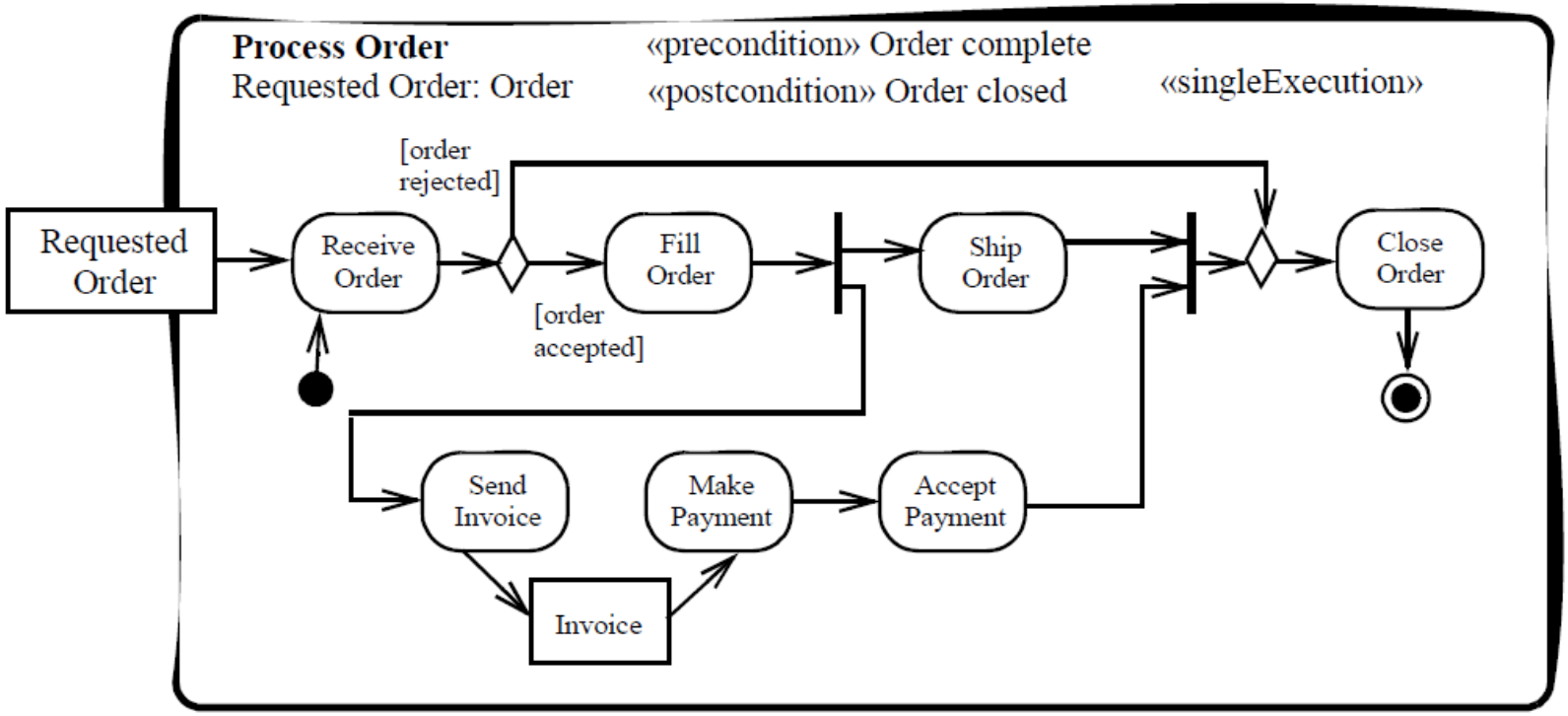
出路：面向状态

- 系统分析人员面向状态进行系统分析设计
  - 采用流程图及UML状态图作为可视化工具。
  - 面向并发、面向进程、面向函数进行面向状态的抽象与建模。
- 开发人员使用面向状态的编程模板。
  - 只需要关注本状态下的函数编程
  - 不采用防御性编程模式
  - 不考虑单元测试代码
- 测试人员使用面向状态的测试模板。
  - 归一化的状态函数只需要一种单元测试用力方法。
  - 进程日志统计建立系统运行状态报告及异常警示。
  - 状态实例统计数据建立函数单元正常运作的指导线。
- 以状态机系统作为Erlang/OTP平台及分布式系统的隐喻
  - 多细胞生物体
- 以传统流程图方式描述业务逻辑。
  - 直观，自然。业务人员与研发人员共享的一种可视化语言。

生命体也是分布式并发系统

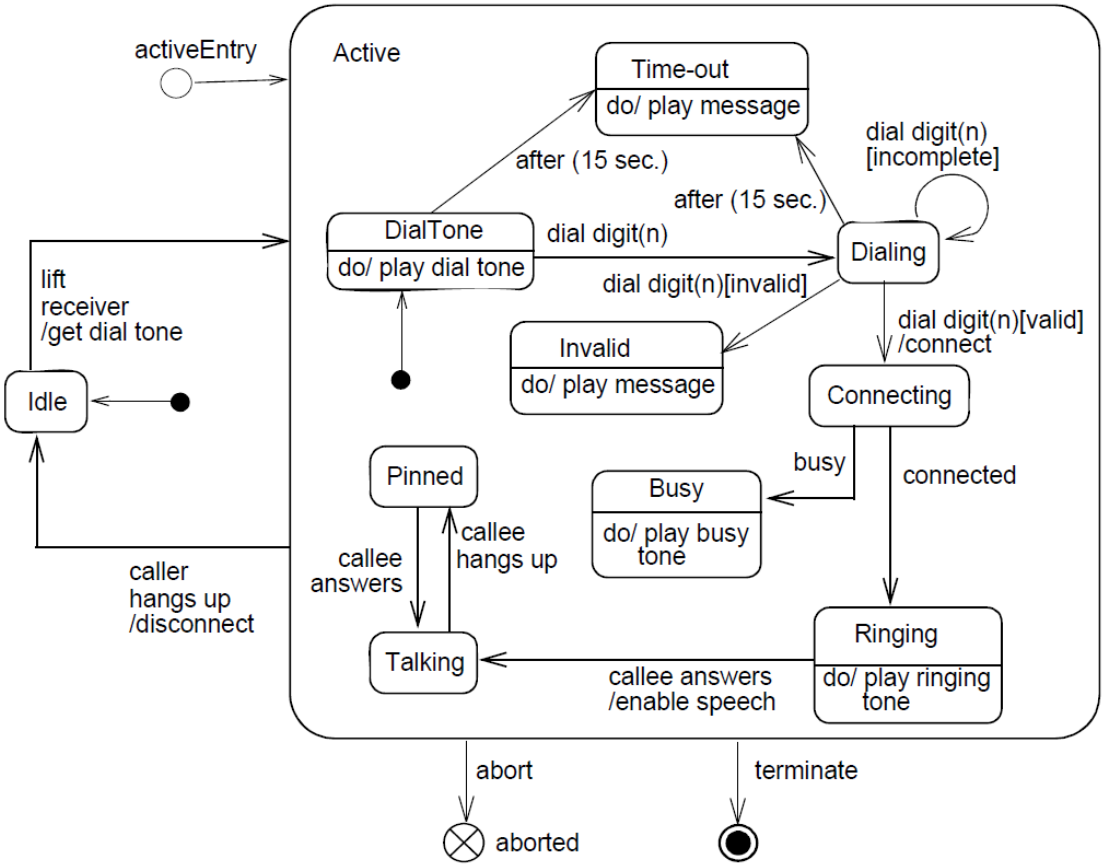
- 复杂的生命体都是由小的生命体：细胞组成的。
- 细胞之间是松耦合的，通过一致的方式进行沟通交互。
- 生命体的组成部分与组成生命体本身可能是大不相同的。
- 细胞通过大量简单叠加规则构建复杂生命体，互相之间有很高的相似性。
- 通过自组织自适应生命体的形态也在生长变化。

工具：UML活动图



- 活动图与状态图的等效
  - 单一进程中的action等效状态图的State

工具：UML状态图



- 状态与函数的等效。

敢不敢踏入雷池？

- Erlang基于Lambda算式构建的准函数式编程系统。
  - 程序员的知识背景通常是面向过程的图灵系统。
- Side Effect：函数式编程语言设计中的大忌。
  - 函数式编程尽量规避状态和变量。
  - 传统程序开发人员视变量和状态为空气和水。
- 数据传递效率：函数直接调用的效率最高。
  - 跨状态机传递数据只能通过消息事件。

抑制边界效应(Side Effect)

- 分离代码与数据
  - 把状态分解为逻辑部分（函数）和数据部分（状态对象）。
- 分离常量与变量
  - 环境数据、配置数据与输入输出数据分离。
- 数据采用状态对象函数封装
  - 采用函数闭包而不是静态数据结构存储。
- 为每个状态机创建独立的状态对象
  - 数据对象被状态机独占访问。
  - 改变的数据在一个状态机中迁移。
- 通过状态实体管理状态对象及状态机
  - 一个状态机的状态保持唯一性和完备性。
  - 一个状态实体可以建立多个状态机同时执行。

等效性

- 图灵机系统与Lambda演算是等效的
  - Lambda可以模拟图灵机，反之亦然。
- 命令式编程与函数式编程的等效
  - 函数式编程可以认为是移除了流程控制的命令式编程模式。
  - 模式匹配、运算顺序加上递归调用实现了类似于命令式编程的时序控制机制。
- 面向状态的编程模式模糊了命令式编程和函数式编程的界限。
  - 状态即是函数
  - 状态机即是流程控制
  - 流程控制部分抽离出来作为公共部分之后，剩下的是面向函数编程。
- 流程图与状态图的等效
  - 同进程的流程图的每个环节等效一个状态。
  - 进程中运行的每个函数等效一个状态。

归一化

- 归一化函数形态
  - 状态作为一种统一的函数形式。y = f(x)。
- 归一化状态对象形态
  - 采用函数闭包形式封装状态对象，使其更加灵活高效。
- 归一化函数式编程与命令式编程。
  - 把命令式编程的流程控制部分分离出来，交由状态机执行。去除流程控制的命令式编程即为函数式编程。
- 归一化状态机与状态实例。
  - 状态机本身也可以看作是一个状态实例，有entry/do/exit行为。局部与整体近似。
- 归一化对象间的关系。
  - 采用松耦合的Link关系建立对象间扁平的网状结构。
- 归一化监控跟踪模式。
  - 每个状态机及状态实例有起止时间及事件记录，每个状态有统计数据。
  - 把静态日志变为动态流程，使系统可重现可自省。
- 归一化面向并发的编程方式。

动态、自洽与自适应

- 参考Moore Machine模型：O = f(S)
  - 只需要entry action
  - 输出只与状态有关
- 状态迁移前后加入Guard函数
  - 迁移前映射输出directive到next state
  - 迁移后完成状态日志
  - 保障状态机的唯一、完备及有限。
- 当前状态函数可以不用关心下一状态。
  - 通过Guard函数进行映射
- 输出与状态映射是可配置数据。
  - 无需重新编译即可修改状态机及流程。
- 自适应与自愈
  - Guard函数可根据状态统计数据选定流程路径。
  - 状态执行异常可进行回滚自愈尝试。
- 异常侦测
  - Loop Detection
  - Dead Lock Detection
  - Abnormal Detection

返璞归真，顺其自然

- Everything is an Erlang process → Everything is a finite state machine.
- Concurrency oriented programming → State oriented programming.
- OTP多采用事件驱动开发模式，舍本逐末。
- 采用SOP开发模式，清晰自然。
  - 即使不做任何编程模板，Erlang语言本身也能够很自然的用SOP模型描述。
- Flow-based programming vs Message-based programming.
- 晦涩艰深的函数式编程 vs 简单直接的命令式编程。
  - TIOBE排名中，Erlang已经跌出前50名。

Erlang/OTP平台反思

- ECUG为什么要改名？
  - 去Erlang化？
- Erlang语言在Tiobe排名中为何逐年下降？
  - 应该随着云计算而第二春才对啊
- Erlang为什么这么大？
  - 第一次下载编译安装Erlang你花了多长时间？
  - 其他声明型语言为何footprint那么小？
- OTP平台入门为什么那么难？
  - 新手理解gen_fsm, gen_server要花费多长时间？
- Erlang OTP为何很难构建大项目？
  - 命名冲突，注册冲突
- Erlang语言为什么不采用纯函数式编程模式？
  - 时域函数和频域函数哪个更自然易懂？
- OTP平台为何采用事件驱动的编程模式？
  - 函数式编程模式排斥面向进程的编程模式
  - 那个时期流行事件驱动模式。
  - 那个时期的硬件能力弱，事件驱动模式是一种高效的并发编程方法。

Erlang的复兴

- Erlang需要一个精简的内核。
  - 是否可以把OTP从distribution中分离出去，按需加载。
- Erlang需要一套简洁的开发模式。
  - 面向状态编程
  - 面向状态机建模
  - Erlang是图灵系统与Lambda演算系统完美结合。
- Erlang需要一套简单的平台。
  - 归一化的编程模板。
  - 细致的日志调试跟踪能力。
  - 自洽、自省的运行时能力。

建模目标

- 构建：基础单元通过简单规则叠加来构建复杂系统。（分形原理）。
- 解构：基础元素只有两种：状态和状态机。
- 理解：局部与总体的相似性，基础元素的一致性让复杂系统更容易理解。
- 管理：模块清晰的边界与一致的行为更易于管理监控。
- 规范：规则简单，组件一致，接口统一。

SOP的概念

- 面向状态机进行抽象。
- 面向状态进行编程。
- 状态实体：对系统中实体的抽象，一个状态实体通常包含一个（或者多个）状态机。
- 状态是某个状态实体在特定情况下的表现形式。
- 状态机呈现的状态集合中的状态是互斥完备的。也就是说一个状态机同时只能呈现一种状态，只能呈现自己状态集合中的有限状态。
- 流程是状态机的运行过程，是状态实体的活动序列，由一系列状态的实例构成的单向时序。

SOP理念

- State是融合函数式编程与命令式编程的关键点，State本身就是函数。
- FSM是State的容器，State在FSM中参数化，FSM本身就是一个进程。
- Process是FSM的实例，是FSM的一条工作路径。
- Process之间通过Link/Monitor机制建立彼此间网状的扁平化关联。

Why not gen_fsm?

- gen_fsm把原本面向状态流程的模式转变成了事件驱动模式。
  - 我个人很讨厌事件驱动模式编程，肮脏破碎。
- gen_fsm把状态边界搞得很模糊，很破碎。
- gen_fsm可能有点过度设计。Erlang进程本身已经是非常干净漂亮的状态机了。
- gen_fsm没有提供灵活的状态加载机制，只能通过本地函数实现。
- gen_fsm没有提供统一的log、debug数据。
- gen_fsm没有提供对状态流转过程的控制接口。

- 不采用消息驱动模式，写一个状态机试试看。

Erlang语言与SOP浑然天成

- 无需任何工作，Erlang语言内置支持SOP。

- Erlang语言天然是SOP的。
  - 进程 ➔ 状态机/状态机实例/流程
  - 函数 ➔ 状态
  - link ➔ 关联
- Erlang语言宣言
  - Everything is an Erlang process.
    - Everything is a finite state machine.
  - Concurrency oriented programming.
    - State oriented programming.

还可以做得更好

- 把函数表现的更像State。
  - 归一化代表State的函数形式：O = f(S)。
- 分离状态逻辑和状态数据
  - 状态逻辑即为函数f
  - 状态数据即为f的输入参数S，higher-order function
  - 用函数闭包而不是数据结构封装数据，更加灵活。
- 动态调整、配置流程。
  - S` = prestate(O, S)
  - O` = poststate(O, S)
  - 当前状态不必关心后续状态实现
  - 通过配置实现输出与状态的映射

还可以更加神奇

- 日志。
  - 横向，以每个流程为线，串起流经状态实例。
  - 纵向，以状态、状态机为线，串起每个实例化运行数据。
- 回溯与重现
  - 流程日志可以重现该状态机运行过程
  - 运行中可以回卷到流程的前几步。
- 自省
  - 循环检测。
    - 状态迁移次数检定。
    - 状态环回检定。
    - 单流程同一状态流经次数检定。
  - 病态检测
    - 流程或状态实例执行参数偏离统计正常值判定。
    - 死锁及堵塞判定。
- 自愈与自适应
  - 根据自省及状态信息，进行回卷、迂回、启用旧版本等操作。

还可以更加时髦

- 像做Web页面一样做Erlang应用。
- 每个状态可以对应一个配置页面。
- 每个状态机相当于一个Web目录。
- 每个流程对应一个Web Session。
- 通过REST方式访问每个状态及状态机。
- 通过Json ←→Erlang Term映射进行互操作和持久化。
- 采用Web页面作为Erlang的表示层。
- 通过Web页面控制状态的颗粒度以对应具体的业务逻辑。

SOP建模

- 状态对象SO：O = f(S)。
  - O为输出 {status, result, S'}
  - f表达状态逻辑函数
  - S状态数据，采用Closure封装，表现为高阶函数。
- 状态机对象FSM：FSM ≈ S。
  - FSM = {SO1, SO2, …, SOn}
  - 状态机是一个容器。
  - 状态机被其当前状态表达。
- 流程：状态机的实例，是一个Erlang Process。
  - 一个Erlang Process只能运行一个流程，每个流程同时只能表达一个状态。
- 关联：状态机之间通过Link/Monitor建立联系。
  - 状态机通过关联建立局部运行环境。
  - 建立Mesh Network。
  - 呈现扁平的网状有限结构。

状态数据Closure

- 状态流转过程中传递的数据即为状态实体。
- 状态数据采用函数方式封装。
  - 更加灵活
  - 也许更加高效
  - 更符合Functional Programming
- 支持统一的交互方式: State(Command, Data)
  - State(get, Key) → {ok, Value} | {error, Error}
  - State(set, {Key, Value}) → {update, NewState} | {ok, Result} | {error, Error}
- 特例情况
  - 当状态数据在其他进程中被修改，则此状态机实例接收到{state_update, NewState, From}

状态实体

- 状态实体是状态数据的超集，绝大多数情况二者等同。
- 状态实体支持的通用函数。
  - Entry(State, Args) → {ok, StartState, NewState} | {error, Error}
    - 初始化状态机实例
  - Exit(State, Reason)
    - 状态机停机回调
  - Prestate(Next, State) → {ok, Entry, NewState} | {error, Error}
    - 状态映射与预处理
  - Poststate(Status, Result, State) → {OStatus, OResult, OState}
    - 日志，后处理与流程修饰。

状态数据传递优化

- 尽量避免以异步方式获取状态数据。
- 数据采用函数闭包封装，频繁访问的数据以本地变量的形式保存。
- 只有当数据对象改变时才生成新的Closure，显式返回。
  - State(set, {Key, Value}) → {update, NewState} | {ok, Result} | {error, Error}
- 简单的数据结构可以采用Record/Tuple方式短期替代Closure。
  - Entry函数返回的NewState不一定是函数。

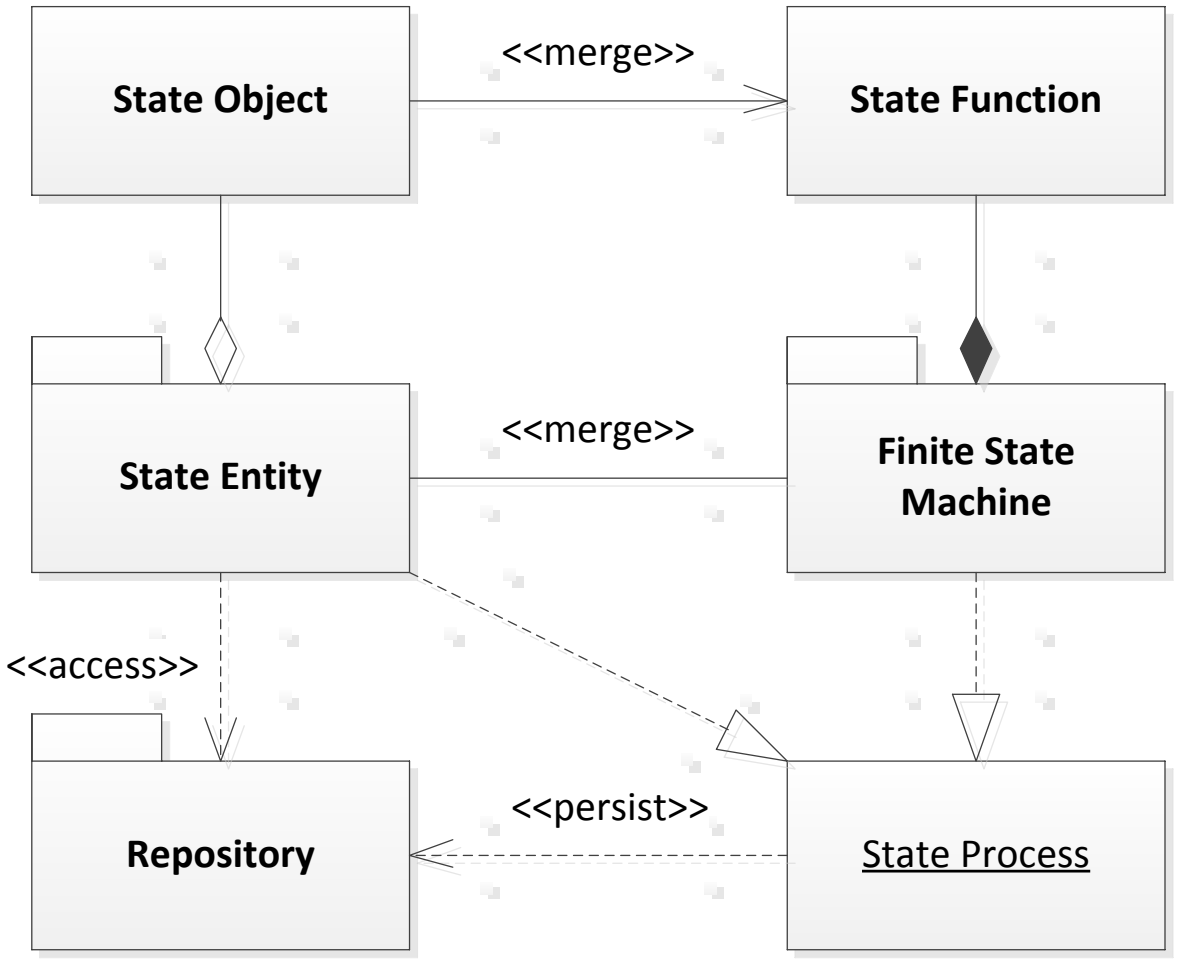
状态

- 状态函数的形式为：
  state_name(State) → {Status, Directive, NewState}
- 通常情况下State参数是高阶函数形式，有些情况是自定义简单数据结构。
  - 通过状态机的Entry函数进行转换。
- 状态函数返回值中不一定包含下一个状态名称，函数体内也不需要知晓下一个可能状态的名称。
  - 可以返回特定的Directive，通过prestate函数根据配置映射为下一个状态名称。
- 一个状态对象包含3部分内容
  - 状态函数：状态执行代码。
  - 状态配置：状态实体数据，包括输入数据。
  - 状态关系：返回值与下一个状态的映射关系。

状态机

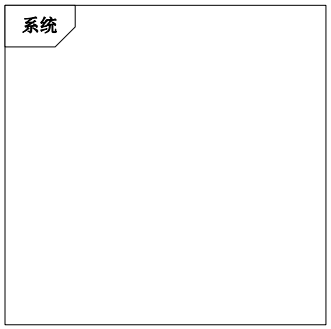
- 状态机的静态表现形式通常为一个Erlang Library Module，包含一系列状态函数。
- 状态机动态形式可以表现为一个状态实体函数，通过其内部的prestate函数动态动态构建状态函数。
- 一个状态机可以同时产生多个流程，每个流程所流转的状态实体可以共用（不鼓励）。

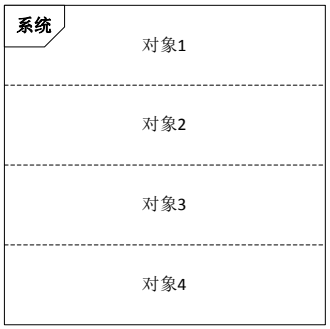
SOP模型

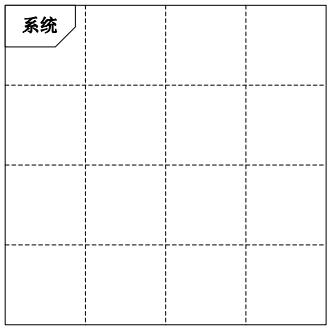

State Object
<<merge>>
State Function
State Entity
<<merge>>
Finite State Machine
<<access>>
Repository
<<persist>>
State Process

空域与时域

- OOP侧重于静态空间建模。
- SOP侧重于动态时序建模。
- 二者可以结合来进一步细化抽象的颗粒度。
- 状态机可以认为是OO与SO的结合。

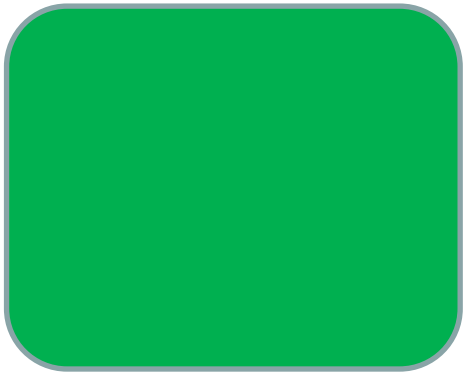




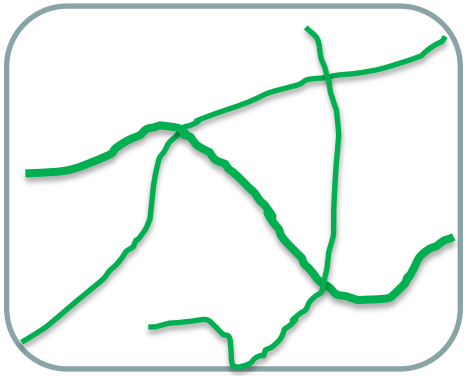


发散的平面与聚焦的线条

- OOP是对最终设计物的抽象，采用防御性设计模式，抽象程度和设计边界很难把握。
- SOP只面对具体工作流程，通过异常处理限定为有限状态机，边界清晰，通透清澈。
- （谁能告诉我怎么画虚线？）



面向对象编程



面向状态编程

面、线、点

- 如果把系统当作一个平面，SOP设计相当于在画线描图，未触及的地方要留白。OOP则更像是做油画。
- 状态机是系统平面上的线条。
- 状态是构成线条的点。

分布并发系统抽象

- OOP倾向于把设计对象抽象为单一复杂系统。（系统论）
- SOP倾向于把设计对象分解为松耦合、分布运算的状态机。（还原论）
- 现实世界是面向并发的分布系统。
- 巨系统通常是由分离而相似的组成元素通过简单叠加、共同作用构成的。
- 我们可以使用简单可计算的元素构建不可计算的复杂系统。反之我们无法从不可计算系统中还原基本元素和初始状态。

- 结论：SOP比OOP更“自然”，更符合本次论坛主题☺

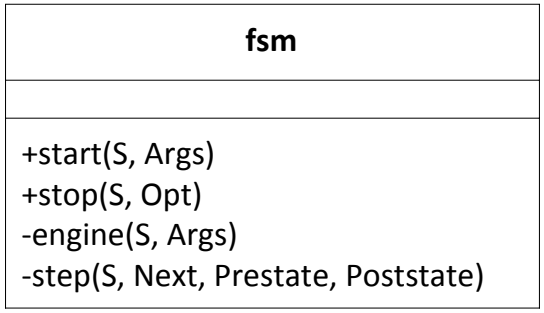
抽象 vs 具象

- OOP是站在观察者角度，通过对实体对象的观察进行归纳抽象。
- SOP直接作为参与者以具体的执行逻辑去构建、组成这个系统，不需要抽象或者只需要轻度的抽象。
- OOP建立的是想象中的系统，而系统在构建好之前通常是难以捉摸的，对系统人员的要求很高。
- SOP表达就是系统某个具体的执行元素。系统是这些元素共同作用的结果。

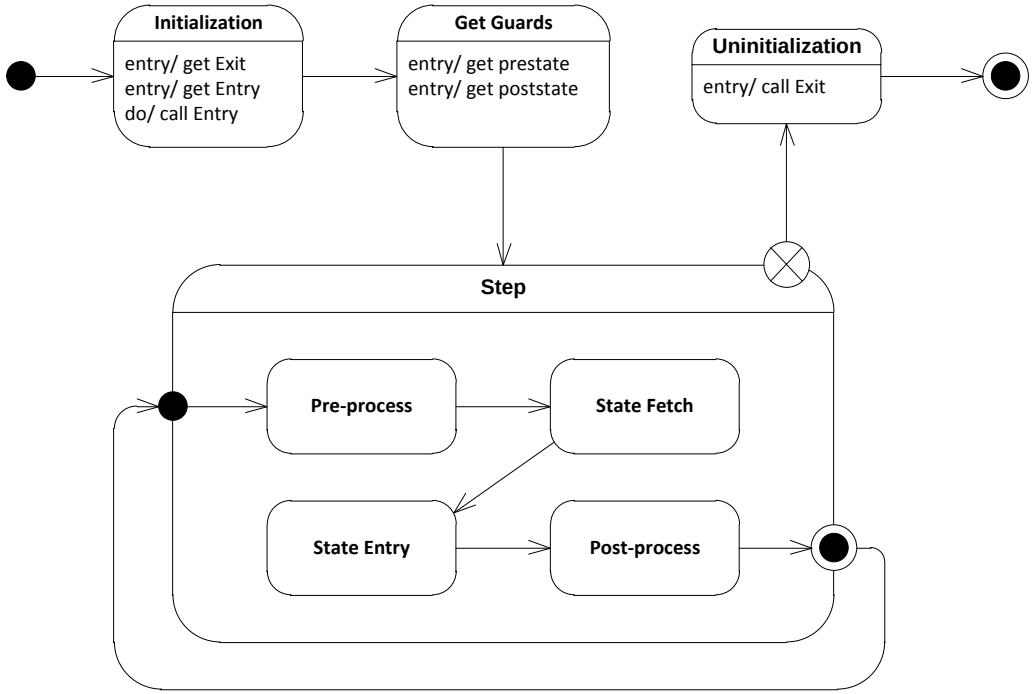
Web应用架构时代

- Getting Real.
- Modeling is dying.
- Application is real, not model.

状态机引擎



Class Diagram



State Diagram

状态机实体

- 静态的状态机实体可以采用Erlang Module进行封装。
- 动态的状态机实体可以就是一个state/2函数。
- 对外可以只提供一个state/2函数作为接口；对内提供一个用于封装状态机对象的state/3函数。
- state/2相当于OO中的静态函数，用于对于实体类全局的操作。
- state/3是针对某个状态对象实例的操作。外部使用时通常封装成state/2形式：

```
wrap(S) ->
    fun(Command, Data) ->
        state(Command, Data, S)
    end.
```

- 各个状态函数统一采用state_name(State)封装。其中State通常是封装后的状态对象。
- 内部的执行函数可以通过state(Function, Data)形式调用。根据情况运行于全局或者局部某个状态。
- 每个状态函数负责在自己状态下的事件处理。
- prestate主要用于next state映射，poststate则通常使用公用的函数。