CTI详细设计

作者:祁伟

时间:2013年4月21日

**目录**

[CTI详细设计 1](#_Toc435553162)

[前言： 10](#_Toc435553163)

[1. 引言 11](#_Toc435553164)

[1.1 编写目的 11](#_Toc435553165)

[1.2 背景 11](#_Toc435553166)

[1.3 参考资料 11](#_Toc435553167)

[2. 程序系统的结构 11](#_Toc435553168)

[3. 状态机设计 11](#_Toc435553169)

[3.1 介绍 12](#_Toc435553170)

[3.1.1 核心结构 12](#_Toc435553171)

[3.1.2 组合状态 12](#_Toc435553172)

[3.2 基本元素 13](#_Toc435553173)

[3.2.1 <FSM> 13](#_Toc435553174)

[3.2.1.1 属性明细 13](#_Toc435553175)

[3.2.1.2 子元素 13](#_Toc435553176)

[3.2.2 <state> 13](#_Toc435553177)

[3.2.2.1 属性明细 14](#_Toc435553178)

[3.2.2.2 子元素 14](#_Toc435553179)

[3.2.3 <event> 14](#_Toc435553180)

[3.2.3.1 属性明细 14](#_Toc435553181)

[3.2.3.2 子元素 15](#_Toc435553182)

[3.2.4 <transition> 15](#_Toc435553183)

[3.2.4.1 属性明细 15](#_Toc435553184)

[3.2.4.2 子元素 15](#_Toc435553185)

[3.2.5 <onentry> 15](#_Toc435553186)

[3.2.5.1 属性明细 16](#_Toc435553187)

[3.2.5.2 子元素 16](#_Toc435553188)

[3.2.6 <onexit> 16](#_Toc435553189)

[3.2.6.1 属性明细 16](#_Toc435553190)

[3.2.6.2 子元素 16](#_Toc435553191)

[3.2.7 合法的状态结构和说明 17](#_Toc435553192)

[3.2.8 事件 17](#_Toc435553193)

[3.2.8.1 事件描述 17](#_Toc435553194)

[3.2.9 Error 17](#_Toc435553195)

[3.2.10 选择和执行跳转 18](#_Toc435553196)

[3.2.11 ID 18](#_Toc435553197)

[3.3 状态机可执行内容 18](#_Toc435553198)

[3.3.1 介绍 18](#_Toc435553199)

[3.3.2 <raise> 19](#_Toc435553200)

[3.3.2.1 属性明细 19](#_Toc435553201)

[3.3.2.2 子元素 19](#_Toc435553202)

[3.3.3 <log> 19](#_Toc435553203)

[3.3.3.1 概述 19](#_Toc435553204)

[3.3.3.2 属性明细 19](#_Toc435553205)

[3.3.3.3 子元素 20](#_Toc435553206)

[3.3.4 <timer> 20](#_Toc435553207)

[3.3.4.1 属性明细 20](#_Toc435553208)

[3.3.4.2 子元素 20](#_Toc435553209)

[3.3.5 <sleep> 20](#_Toc435553210)

[3.3.5.1 属性明细 21](#_Toc435553211)

[3.3.5.2 子元素 21](#_Toc435553212)

[3.3.6 其他可执行内容 21](#_Toc435553213)

[3.3.7 执行可执行内容 21](#_Toc435553214)

[3.3.8 扩展性可执行模块 21](#_Toc435553215)

[3.4 状态机数据模块和数据操作 22](#_Toc435553216)

[3.4.1 介绍 22](#_Toc435553217)

[3.4.2 <datamodel> 22](#_Toc435553218)

[3.4.2.1 属性明细 22](#_Toc435553219)

[3.4.2.2 子元素 22](#_Toc435553220)

[3.4.3 <data> 22](#_Toc435553221)

[3.4.3.1 属性明细 22](#_Toc435553222)

[3.4.3.2 子元素 22](#_Toc435553223)

[3.4.4 <scriptmodel> 23](#_Toc435553224)

[3.4.4.1 属性明细 23](#_Toc435553225)

[3.4.4.2 子元素 23](#_Toc435553226)

[3.4.5 <script> 23](#_Toc435553227)

[3.4.5.1 属性明细 23](#_Toc435553228)

[3.4.5.2 子元素 23](#_Toc435553229)

[3.4.6 <script> 23](#_Toc435553230)

[3.4.6.1 属性明细 23](#_Toc435553231)

[3.4.6.2 子元素 24](#_Toc435553232)

[3.4.7 表达式 24](#_Toc435553233)

[3.4.7.1 条件表达式 24](#_Toc435553234)

[3.4.7.2 本地表达式 24](#_Toc435553235)

[3.4.7.3 合法的数据值和值类型表达式 24](#_Toc435553236)

[3.4.8 系统变量 24](#_Toc435553237)

[3.4.8.1 事件的内部结构 25](#_Toc435553238)

[3.5 状态机外部通信 25](#_Toc435553239)

[3.5.1 介绍 25](#_Toc435553240)

[3.5.2 <send> 25](#_Toc435553241)

[3.5.2.1 属性明细 26](#_Toc435553242)

[3.5.2.2 子元素 27](#_Toc435553243)

[3.5.2.3 发送目标 27](#_Toc435553244)

[3.5.2.4 发送类型 27](#_Toc435553245)

[3.5.2.5 数据传递 27](#_Toc435553246)

[3.5.2.6 实现 28](#_Toc435553247)

[3.6 一致性 28](#_Toc435553248)

[3.6.1 文档一致性 28](#_Toc435553249)

[3.7 FSM解析器算法 28](#_Toc435553250)

[3.7.1 非正式语义 28](#_Toc435553251)

[3.7.2 预定义 28](#_Toc435553252)

[3.7.2.1 状态 28](#_Toc435553253)

[3.7.2.2 transition目标 28](#_Toc435553254)

[3.7.2.3 原子状态 28](#_Toc435553255)

[3.7.2.4 组合状态 29](#_Toc435553256)

[3.7.2.5 源状态 29](#_Toc435553257)

[3.7.2.6 目标状态 29](#_Toc435553258)

[3.7.2.7 外部事件 29](#_Toc435553259)

[3.7.2.8 内部事件 29](#_Toc435553260)

[3.7.3 原则和约束 29](#_Toc435553261)

[3.7.3.1 封装 29](#_Toc435553262)

[3.7.3.2 执行完成 29](#_Toc435553263)

[3.7.3.3 结束 29](#_Toc435553264)

[3.7.4 算法 30](#_Toc435553265)

[3.7.4.1 数据类型 30](#_Toc435553266)

[3.7.4.2 全局变量 30](#_Toc435553267)

[3.7.4.3 函数和功能 30](#_Toc435553268)

[3.7.4.3.1 函数mainEventLoop() 30](#_Toc435553269)

[3.8 Schema 30](#_Toc435553270)

[3.9 数据模型 30](#_Toc435553271)

[3.10 外部模块 30](#_Toc435553272)

[3.10.1 XML 模块 30](#_Toc435553273)

[3.10.2 日志模块 30](#_Toc435553274)

[3.10.3 脚本解析模块 30](#_Toc435553275)

[3.11 代码实现 30](#_Toc435553276)

[3.11.1 库的选用 30](#_Toc435553277)

[3.11.2 日志模块的选用 31](#_Toc435553278)

[3.11.3 脚本解析模块选用 31](#_Toc435553279)

[3.12 问题 31](#_Toc435553280)

[3.12.1 不满足执行条件 31](#_Toc435553281)

[3.12.2 组合操作 31](#_Toc435553282)

[3.12.3 多个事件逻辑错误 31](#_Toc435553283)

[4. 软电话设计说明 33](#_Toc435553284)

[4.1 程序描述 33](#_Toc435553285)

[4.2 功能 33](#_Toc435553286)

[4.3 性能 33](#_Toc435553287)

[4.4 输入项 33](#_Toc435553288)

[4.4.1 用户界面操作 33](#_Toc435553289)

[4.4.2 API 33](#_Toc435553290)

[4.4.3 来自CTIServer的消息 34](#_Toc435553291)

[4.5 输出项 34](#_Toc435553292)

[4.5.1 软电话发送给CTIServer的消息 34](#_Toc435553293)

[4.5.2 发送给API的消息 35](#_Toc435553294)

[4.5.3 界面显示信息 35](#_Toc435553295)

[4.6 流程逻辑 35](#_Toc435553296)

[4.6.1 初始化状态 36](#_Toc435553297)

[4.6.2 未登录状态 36](#_Toc435553298)

[4.6.3 登陆中状态 36](#_Toc435553299)

[4.6.4 服务器返回状态 36](#_Toc435553300)

[4.7 算法 36](#_Toc435553301)

[4.7.1 连接方式 36](#_Toc435553302)

[4.7.2 状态机 36](#_Toc435553303)

[4.7.3 软电话消息发送 36](#_Toc435553304)

[4.7.4 软电话消息接收 37](#_Toc435553305)

[4.7.5 软电话消息解析 37](#_Toc435553306)

[4.7.6 软电话API 37](#_Toc435553307)

[4.7.7 全局变量 37](#_Toc435553308)

[4.7.8 软电话配置脚本 37](#_Toc435553309)

[4.8 限制条件 38](#_Toc435553310)

[4.9 测试计划 38](#_Toc435553311)

[4.10 尚未解决的问题 38](#_Toc435553312)

[4.11 代码实现 38](#_Toc435553313)

[4.11.1 Web版本 38](#_Toc435553314)

[4.11.1.1 特点 38](#_Toc435553315)

[4.11.1.2 结构图 39](#_Toc435553316)

[4.11.1.3 逻辑 40](#_Toc435553317)

[4.11.1.4 界面参考 40](#_Toc435553318)

[5. CardAdapter模块设计说明 40](#_Toc435553319)

[5.1 程序描述 40](#_Toc435553320)

[5.2 功能 40](#_Toc435553321)

[5.3 性能 41](#_Toc435553322)

[5.4 输入项 41](#_Toc435553323)

[5.4.1 来自IVR的放音消息 41](#_Toc435553324)

[5.4.2 来自CTIServer的分机注册消息 41](#_Toc435553325)

[5.4.3 来自CTIServer的呼叫分机消息 41](#_Toc435553326)

[5.4.4 来自CTIServer的录音消息 42](#_Toc435553327)

[5.5 输出项 42](#_Toc435553328)

[5.5.1 返回给IVR的放音结果消息 42](#_Toc435553329)

[5.5.2 返回给CTIServer的分机注册消息 42](#_Toc435553330)

[5.5.3 返回给CTIServer的呼叫分机结果消息 42](#_Toc435553331)

[5.5.4 返回给CTIServer的录音消息 42](#_Toc435553332)

[5.6 流程逻辑 43](#_Toc435553333)

[5.7 算法 44](#_Toc435553334)

[5.7.1 连接方式 44](#_Toc435553335)

[5.7.2 消息队列 44](#_Toc435553336)

[5.7.3 配置项 44](#_Toc435553337)

[5.7.3.1 全局配置 44](#_Toc435553338)

[5.7.3.2 分机配置 44](#_Toc435553339)

[5.7.4 状态机 44](#_Toc435553340)

[5.7.4.1 分机外呼 45](#_Toc435553341)

[5.7.4.2 分机呼入 45](#_Toc435553342)

[5.7.4.1 状态描述 46](#_Toc435553343)

[5.7.5 消息发送 46](#_Toc435553344)

[5.7.6 消息接收 46](#_Toc435553345)

[5.7.7 软电话消息解析 47](#_Toc435553346)

[5.7.8 变量 47](#_Toc435553347)

[5.7.8.1 全局变量 47](#_Toc435553348)

[5.7.8.2 分机变量 47](#_Toc435553349)

[5.7.9 CardAdapter配置脚本 47](#_Toc435553350)

[5.7.10 StateMachine类 48](#_Toc435553351)

[5.7.11 extension 类 49](#_Toc435553352)

[5.8 限制条件 49](#_Toc435553353)

[5.9 测试计划 49](#_Toc435553354)

[5.10 尚未解决的问题 49](#_Toc435553355)

[5.11 代码实现 49](#_Toc435553356)

[5.11.1 Keygoe版本 49](#_Toc435553357)

[5.11.1.1 特点 49](#_Toc435553358)

[5.11.1.2 结构图 49](#_Toc435553359)

[5.11.1.3 逻辑 50](#_Toc435553360)

[6. IVR流程模块设计说明 50](#_Toc435553361)

[6.1 程序描述 50](#_Toc435553362)

[6.2 功能 50](#_Toc435553363)

[6.2.1 语音播放 50](#_Toc435553364)

[6.2.1.1 属性明细 50](#_Toc435553365)

[6.2.1.2 子元素 50](#_Toc435553366)

[6.2.1.3 示例 51](#_Toc435553367)

[6.2.2 按键接收 51](#_Toc435553368)

[6.3 性能 51](#_Toc435553369)

[6.4 输入项 51](#_Toc435553370)

[6.4.1 放音完成消息 51](#_Toc435553371)

[6.5 输出项 51](#_Toc435553372)

[6.5.1 放音命令 51](#_Toc435553373)

[6.6 流程逻辑 52](#_Toc435553374)

[6.6.1 转人工过程 52](#_Toc435553375)

[6.7 算法 52](#_Toc435553376)

[6.8 限制条件 52](#_Toc435553377)

[6.9 测试计划 52](#_Toc435553378)

[6.10 尚未解决的问题 52](#_Toc435553379)

[6.11 代码实现 53](#_Toc435553380)

[7. ACD模块设计说明 53](#_Toc435553381)

[7.1 程序描述 53](#_Toc435553382)

[7.2 功能 53](#_Toc435553383)

[7.2.1 交换功能 53](#_Toc435553384)

[7.2.2 排队功能 53](#_Toc435553385)

[7.2.3 路由功能 53](#_Toc435553386)

[7.3 性能 53](#_Toc435553387)

[7.3.1 实时性 53](#_Toc435553388)

[7.3.2 并发性 53](#_Toc435553389)

[7.4 输入项 53](#_Toc435553390)

[7.5 输出项 53](#_Toc435553391)

[7.6 流程逻辑 53](#_Toc435553392)

[7.7 算法 53](#_Toc435553393)

[7.8 限制条件 54](#_Toc435553394)

[7.9 测试计划 54](#_Toc435553395)

[7.10 尚未解决的问题 54](#_Toc435553396)

[7.11 代码实现 54](#_Toc435553397)

[8. CTIServer模块设计说明 54](#_Toc435553398)

[8.1 和软电话之间的连接 54](#_Toc435553399)

[8.2 软电话 54](#_Toc435553400)

[8.2.1 初始状态 54](#_Toc435553401)

[8.2.2 未登录状态 55](#_Toc435553402)

[8.2.3 暂停状态 55](#_Toc435553403)

[8.2.4 振铃状态 55](#_Toc435553404)

[8.2.5 话后状态 56](#_Toc435553405)

[8.2.6 通话状态 56](#_Toc435553406)

[8.2.7 保持状态 56](#_Toc435553407)

[8.2.8 就绪状态 57](#_Toc435553408)

[8.2.9 转接中状态 57](#_Toc435553409)

[8.2.10 会议中状态 57](#_Toc435553410)

[8.2.11 会议状态 57](#_Toc435553411)

# 前言：

本文档主要目是对《CTI概要设计》的下一任务的文档描述，对《CTI概要设计》进行细化描述，其中涉及到详细的数据结构、逻辑描述、代码结构、代码实现方式、算法选用。项目中有一些模块用到的功能，尽量使用开源项目，以减少工作量，增加程序健壮性。

# 引言

## 编写目的

对《CTI概要设计》进行细化，描写好各个模块的数据结构、逻辑功能流程、对算法的选择，使用伪码描写出各个模块的代码结构，细化出各个模块之间的接口，使各个模块之间的代码可以同步进行编写。

## 背景

文档的编写已经将《CTI需求分析》和《CTI概要设计》完成，当然还有很多缺陷和错误，但是已经有了项目的思路。感谢一些朋友的支持，希望自己能将这个项目完成。

## 参考资料

百度百科：详细设计

百度百科：详细设计说明书

《CTI概要设计》

# 程序系统的结构

参考 《CTI概要设计》2.4 结构。

# 状态机设计

## 前言

程序中多处的状态使用状态机来描述，包括对一些事件的处理也使用状态机来实现，所以状态机在程序中占的比例较重，最初此部分的实现想找开源的状态机直接使用，这样会省去很多的开发时间，开源项目的设计比较完善，并且代码质量比自己写的要好，也经过了很多人的验证，稳定性要好。

经过了在网络和开源社区的查找，scxml 状态机比较适合此项目的应用，此项目是W3C的定义的一项标准，现在还处于草稿阶段，但总设计上看还是不错的，接触到了状态机实现的很多理念。基于此标准实现的代码还不多，使用C++语言实现的在开源社区找到了sscxml和uscxml，基于java实现的有apache组织下的一个项目。

由于此项目是使用C++来编写，所以对apache的项目代码只做了大概的浏览。一直在研究W3C的文档和C++版本的实现。W3C的文档是英文版，阅读比较慢，经过了大概一周的时间发现此标准的设计是很丰富，但不符合此项目对状态机的要求。此项目的状态机中每个状态下可以有N个子状态，子状态和父状态是平等的。但是在W3C标准中将每个状态下的所有子状态和此状态定义成一个组合状态，只要进入了组合状态中的任一状态，必定进入了父状态，进入了父状态也必须同时进入一个子状态。由于有设计方面的冲突，所以不可能直接使用scxml标准，计划研究一下此标准的实现，然后修改少量的代码加入到自己的项目中。

最先研究的是sscxml，此项目是较早使用C++来实现scxml的，已经有2年没人维护了。此项目的代码过于混乱，在提供的源代码中有很多错误，无法编译通过，很快放弃了此项目。然后研究的是uscxml项目。此项目的代码庞大、在社区比较活跃，并且在Windows和linux上都有实现，很快决定使用此项目。分析了两周时间后才发现此代码的实现并不理想，代码中还存在很多BUG，在修改上也没那么容易，并且每个状态机都会启动一个线程，如果在程序中启用大量的状态机实例不知道会造成什么样的后果。相比apache的项目代码质量要差很多。

因此本人想自己实现一个状态机，状态机的主要定义使用w3c中的。修改了状态机中的复合状态，在状态机中每个状态和其他的任何状态都是平等的。其他的细节也有些修改，但只是一些表达方式上的不同，对概念上没有区别。

项目中的应用方面有限，并不计划实现scxml中的所有功能，只实现其中的部分功能，满足要求即可。

## 介绍

此状态机是一个基于事件的通用型状态机设计。

核心结构提出了核心状态机概念，执行内容包括了一组可扩展的动作，以响应事件；数据模块定义了数据结构和对数据的修改；外部通信提供了与外部实体进行通信的功能。

### 核心结构

基本的状态机概念包括 state、transition、event。每个状态都包含了event去定义如何响应收到的事件。事件可以由状态机自己或者外部实体产生。在传统的状态机中，这个状态机总是处于一个状态，这个状态被成为当前状态。当遇到一个事件，这个状态机检查当前状态下的event，当找到一个与event相匹配，执行event元素下的transition它从当前状态转移到transition指定的状态，这个目标状态将变为新的当前状态。

在现在的状态机中有几个扩展概念，在状态机匹配event后会有其他动作（详见执行内容部分），特别的是，每个状态都可能包含<onentry>和<onexit>动作，匹配事件后也可能包含其他动作。如果状态机执行转换T从状态S1到状态S2，它首先执行 S1的onexit，然后执行T,然后执行S2的onentry，其次，transition 也可以添加 ‘cond’属性作为可计算的条件值。transition 当属性’cond’当计算值为 true执行，当’cond’属性缺失时默认值为true。如果有多个 transition 匹配，第一个会被执行。因此，下面的例子，当遇到 事件 e 和x等于1 时，跳转到s1，当遇到事件e并且x不等于1是，跳转的s2，其他情况跳转到s3。

<state id=s">

<event event="e">

<transition cond="x==1" target="s1"/>

<transition target="s2"/>

</event>

<event event="\*">

<transition target="s3"/>

</event>

</state>

### 组合状态

在Harel 状态机中有一个重要的概念，就是状态可以包含它的内部结构，尤其是，一个 <state> 元素可以嵌套<state>元素，这样的状态叫做组合状态并且称其为父状态，嵌套状态称其为子状态，这些子状态也可以有自己的嵌套并且可以嵌套任意深度。最终，我们会到达未包含任何子状态的状态。这样的状态成为原子状态。在此状态机设计中并没有严格的组合状态概念，但父状态和子状态也存在一定的对事件匹配的逻辑关系。

组合状态影响到事件匹配的选择，当查找<event>元素时，状态机首先查找当前的活动状态，如果在当前状态中没有匹配，状态机会查找它的父状态，然后是父状态的父状态，以此类推。因此在祖先状态中的匹配事件将是子孙状态下没有匹配到event元素的默认匹配。如果在任何状态下都没有配合，事件将被丢弃（可以在顶层状态中添加通配符匹配要丢弃的事件）。

## 基本元素

名词解释：

**NMTOKEN**：NMTOKEN是Name Token的简写，中文意思就是名称记号。它表示由一个或多个字母、数字、句点、连字号或底线所组成的一个名称。除了第一个字符位置外，NMTOKEN属性的设置值中也可以是冒号。

**IDREFS**：id值引用，此id值表示在脚本文档中 state 元素的id属性。

### <fsm>

【此部分是标准】

最外层元素，版本信息，实际的状态机都是它的子元素。注意在任意时刻，只有一个状态是活动状态。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| initial | false | none | IDREFS | none | 在此脚本中有效的state ID | 此脚本的初始状态。 |
| name | false | none | NMTOKEN | none | 任何有效的NMTOKEN | 状态机的名字，只是为了信息显示。 |
| version | true | none | decimal | none | 有效的数字 | 状态机版本号，例如 1.0 |

#### 子元素

1、<state> 可以出现0到多次。

2、<datamodel> 定义部分或全部的数据模块，可以出现0到多次。

3、<scriptmodel> 提供了脚本能力，可以出现0到多次。

一个符合逻辑的状态机必须至少包含一个<state>元素，当系统初始化时，状态机必须进入到指定的初始状态，如果没有指定，必须进入第一个状态。

### <state>

【此部分是标准】

此标签代表了一个状态。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| id | false | none | NMTOKEN | none | 任何有效的NMTOKEN | 状态的标志，如果没有此属性，状态机将不能查找到此状态。 |
| name | false | none | NMTOKEN | none | 任何有效的NMTOKEN | 状态机的名字，只是为了信息显示。 |
| description | false | none | NMTOKEN | none | 任何有效的NMTOKEN | 状态的描述 |
| type | false | none | NMTOKEN | none | 任何有效的NMTOKEN | 扩展字段，在用状态机实现特定业务时，指定类型，此类型的状态执行特定的功能。 |

#### 子元素

1、<onentry> 当进入<state>时执行的内容，可以出现0到多次。

2、<onexit> 当离开<state>时执行的内容，可以出现0到多次。

3、<event> 当遇到事件时匹配的事件元素，可以出现0到多次。

4、<state> 此状态的子状态，可以出现0到多次。

### <event>

【此部分是标准】

此部分实现了状态机中事件的匹配，在<event>元素中包含的都是事件匹配后要执行的内容。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| event | false | 合法的正则表达式 | 正则表达式 | none | 任何有效的正则表达式 | 根据正则表达式匹配事件。 |
| cond | false | 状态机可以执行的脚本 | 布尔表达式 | true | 状态机所支持的脚本表达式 | 事件匹配的一个条件，计算值为true时，此条件通过。 |

#### 子元素

1、<transition> 当匹配event后执行的状态转换，可以出现0到多次。

2、<log> 当匹配event后执行的日志记录功能，可以出现0到多次。

3、<script> 当匹配event后执行的脚本功能，可以出现0到多次。

4、<send> 当匹配event后执行的和外部模块通信功能，可以出现0到多次。

5、<timer>当匹配event后执行的定时器功能，可以出现0到多次。

6、<raise>当匹配event后执行的产生内部事件功能，可以出现0到多次。

7、<sleep>当执行sleep元素后状态机休眠指定的时间，可以出现0到多次。

【注意：如果省略event属性，认为消息名称是匹配的，如果省略cond属性，默认此值是true。】

【注意：如果第一个transition执行后，余下的元素都不再执行。】

### <transition>

【此部分是标准】

transition 是在各个状态之间实现跳转。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| cond | false | 所支持的脚本表达式 | 布尔表达式 | true | 任何状态机支持的布尔表达式 | 当表达式为’true’时，执行 transition。缺省值为true。 |
| target | true | none | IDREFS | none | 指定的有效state id。 | 状态机会跳转到符合此id的state。 |

#### 子元素

none

【当没有cond属性时，状态机会假定有一个默认值是true的cond属性。】

【一个符合逻辑的状态机脚本，必须指定 target。】

### <onentry>

【此部分是标准】

当进入一个状态后包含在此元素下的内容都会被执行。

#### 属性明细

none.

#### 子元素

<onentry>下可以包含任何以下元素，也可以根据需要扩展其他可执行元素。

1、<log>，可以出现0到多次。

2、<script>，可以出现0到多次。

3、<send>，可以出现0到多次。

4、<timer>，可以出现0到多次。

5、<raise>可以出现0到多次。

6、<sleep>可以出现0到多次。

【当进入一个状态后状态机必须按文档顺序执行<onentry>元素下的内容。必须把每个执行部分分割成一个单独的块来执行。】

### <onexit>

【此部分是标准】

当离开一个状态后包含在此元素下的内容都会被执行。

#### 属性明细

none.

#### 子元素

<onexit>下可以包含任何以下元素，也可以根据需要扩展其他可执行元素。

1、<log>，可以出现0到多次。

2、<script>，可以出现0到多次。

3、<send>，可以出现0到多次。

4、<timer>，可以出现0到多次。

5、<raise>可以出现0到多次。

6、<sleep>可以出现0到多次。

当离开一个状态后状态机必须按文档顺序执行<onexit>元素下的内容。必须把每个执行部分分割成一个单独的块来执行。

【注意：一个符合逻辑的状态机脚本在<onexit>中不能包含<transition>元素。】

【注意：在onexit元素下扩展的其他元素必须保证不可以直接引起状态变化。】

### 合法的状态结构和说明

【此部分是标准】

【定义：一个<state> 元素被进入并处于活跃状态并且不会自动离开退出】

【定义：状态机的结构是改变当前的活跃状态并执行相应的操作。】

当在初始化的时候fsm解析器会指定状态机的初始状态。每个 transition 会将状态机的当前状态改变为另一个状态。一个符合逻辑的状态机文档必须指定状态机是一个合法的状态结构，合法的状态结构需要满足以下条件之一：

1、所有内容必须是包含在<fsm>下的子元素。

2、状态机中<transition>元素的‘target’属性指定的状态都必须在状态机中。

### 事件

【此部分是标准】

事件是状态机中的一个基本概念，用来驱动状态机的状态转变（也包括操作）。事件的内部结构是依赖具体平台的，只需要观察下列外部接口：

1. fsm解析器必须通过’\_event’变量来存取在事件中包含的数据。
2. fsm解析器存取事件的名字必须通过’\_event’变量，必须使用相同的名字去匹配对应的事件属性。

在大部分情况下，在脚本执行过程中引发的事件是<send>和<raise>元素。但是，也有一部分事件是由解析器产生的。详情请见错误和事件。

#### 事件描述

就像是一个事件的名字，事件描述是一个用来匹配事件名称的可见字符串。

【定义：如果事件描述和事件名称匹配，这个<event>就是匹配的（cond 属性另外描述）。】

【定义：一个事件描述和事件名称相匹配，如果事件描述中有前缀标记，事件名称中也需要精确的匹配。在所有的情况下，标记的匹配是大小写敏感的。事件描述也可以使用正则表达式，这样可以进行有选择的匹配。】

例如，一个 <event> 有’event’属性为”error\*” 会匹配事件名称 “error”,”error.send”,”error.send.failed”，等等。但是不会匹配事件的名称”errors.my.custom”,”errorhandler.mistake”,”error.send” or “foobar”。

### Error

【此部分是标准】

一个符合要求的状态机处理器，当处理过程中遇到错误，必须产生一个以”error.”为前缀的事件到内部事件队列中，处理此事件必须像其他事件一样（当时此产生的错误事件不会立即执行，需要等到此事件处理完成才会顺序处理内部事件）。预定义了'error.communication' 和 'error.execution'事件，error.communication是和外部通信时产生的事件，error.execution是在状态机执行过程中产生的事件。

### 选择和执行跳转

【此部分是标准】

以下是简单的定义。

【定义：一个transition 元素T 被事件E激活在事件S中，T的源状态是Ｓ，T父元素<event>匹配E的名称，T没有’cond’属性或者’cond’属性计算值为’true’。】

【一个transition的源状态是包含此元素的状态，transition元素的目标状态应该由其属性’target’指定，transition执行完成后target指定的状态应该处于当前状态。】

【定义：一个transition的父亲应该是<event>。目标state 由’target’属性指定。】

【包含在transition源状态中的<onexit>元素集合应该在transition 触发时执行，如果没有找到’target’所指定的目标<onexit>不执行。】

【transition目标状态的<onentry>集合会在transition执行时执行，如果一个transition没有找到’target’，<onentry>不执行。】

【在一组可执行序列中，如果一个transition被执行，transition顺序排列下的其他元素不再执行】

准备执行一个transition ，解析器必须退出当前状态，然后必须执行要进入state 的entry。

退出一个状态，必须执行当前状态下的 <onexit>。

进入一个状态，fsm解析器必须将此状态设置为当前状态，必须执行此状态的<onentry>下所包含的内容。

### ID

【此部分是标准】

一个符合逻辑的脚本文档，在一个状态机中所有的”id”属性值必须是唯一的，当此属性省略后，解析器必须在加载是自动生成一个唯一的id(注意这样的id 不能在其他地方被引用，因为作者在编辑时还不知道他的存在，特别是，一个state使用一个系统生成的id也不能用作transition的target)。在一个符合逻辑的状态机脚本中，id在状态机中必须唯一，但是也有一些情况作者不用提供他们，解析器会自动生成一个新的唯一id，但不是在加载的时候而是在每次执行的时候。再者，’idexpr’属性能自动生成id，解析器会为其生成任何格式的id，只要他们是唯一的。

## 状态机可执行内容

### 介绍

【此节是说明】

可执行部分允许状态机做一些事情，他提供了修改数据模块和与外部实体交互的方式。可执行内容包括一些可执行的动作，transition就是其中之一。详情中，可执行内容包括<event>、<onentry>和<onexit>元素在内。当状态机处于状态转移时，在离开状态时会执行<onexit>中的可执行内容，接下来是<transition>，接下来是要进入状态的<onentry>。

标准的可执行内容“引发事件”定义参见3.3.2<raise>，和外部实体的通信参见3.5.2<send>，日志信息参见3.3.7<log>，执行脚本参见3.4.8<script>。状态机的实现允许定义自己平台指定的可扩展性执行内容。

### <raise>

【此节是标准】

<raise>元素会在当前会话中引起一个事件。注意，事件不会在当前可执行内容模块执行完成前和所有的内部事件处理完成前处理。例如，假定<raise>元素首先发生在一个S状态的<onentry>中的可执行内容ec1和ec2中，事件e1已经在内部事件队列中，当S状态进入后，直到ec1和ec2执行完毕和e1被处理后，一个事件才会被<raise>产生。详细的事件处理见详细的fsm解释算法。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| event | true | none | NMTOKEN | none |  | 指定事件名称，它将会在 ‘event’属性中匹配。 |
| cond | false | 状态机可以执行的脚本 | 布尔表达式 | true | 状态机所支持的脚本表达式 | 执行元素匹配的一个条件，计算值为true时，此条件通过。 |

#### 子元素

无

fsm 解析器必须将产生的事件放在状态机内部事件队列的结尾。

### <log>

【此节是标准】

<log>是运行状态机时生成日志信息。当状态机解析器执行到此元素时，才会产生一个日志信息，日志模块的实现可根据平台特性输出到屏幕、文件、远端模块。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| level | false | none | 字符串值 | none | 具体有效值依赖实现平台所定。 | 状态机输出的日志类型，如：fatal、warning、error、info、debug、trace等。 |
| type | false | none |  | text | text或script | 指定此元素的内容是字符串常量或者是可执行脚本。 |
| cond | false | 状态机可以执行的脚本 | 布尔表达式 | true | 状态机所支持的脚本表达式 | 执行元素匹配的一个条件，计算值为true时，此条件通过。 |

#### 子元素

log元素内的内容是需要记录的日志内容，可以是字符串常量或者计算结果为字符串常量的可执行脚本。

此方法对于消息的显示和记录依赖于平台。fsm解析器必须保证<log>的执行不会影响到脚本的解析。

### <timer>

【此部分是标准】

当遇到此元素时，解析器会启动一个计时器，计时器到超时后会向外部事件队列添加一个timer事件。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| idexpr | true | 所支持的脚本表达式 | 可执行脚本 | none | 任何状态机所支持的表达式。 | 解析器会根据此表达式产生一个timer ID，在产生的事件中会携带此ID. |
| interval | true | 整型值 | 整型值 | 0 |  | 事件定时间隔，单位毫秒。 |
| cond | false | 状态机可以执行的脚本 | 布尔表达式 | true | 状态机所支持的脚本表达式 | 执行元素匹配的一个条件，计算值为true时，此条件通过。 |

#### 子元素

none

【定时器的功能需要根据平台特性实现，精确度也根据具体实现而定。】

### <sleep>

【此部分是标准】

当执行此元素时，解析器会暂停此状态机直到指定时间。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| interval | true | >=0 | 整型值 | 0 |  | 休眠时间，单位毫秒。 |
| cond | false | 状态机可以执行的脚本 | 布尔表达式 | true | 状态机所支持的脚本表达式 | 执行元素匹配的一个条件，计算值为true时，此条件通过。 |

#### 子元素

none

### 其他可执行内容

【比部分是标准】

以下的元素是可执行内容，定义在脚本中指定的地方。可以定义在允许可执行内容的任何地方，但是不能定义在别处。

<script>提供了脚本能力。

<send>发送一个事件到指定的地方。

### 执行可执行内容

【比部分是标准】

在任何可以出现执行内容的地方，可以出现任意次数。一组顺序可执行内容元素被称为块。例如，在<event>元素下的内容，如果transition从状态S1到状态S2，这里有2个可执行内容块被执行:一个S1状态的<onexit>，还有一个S2的<onentry>。解析器必须顺序执行在块内的元素。如果在处理元素过程中引起了一个错误，解析器必须不处理这个块内剩下的元素（其他块内的不受影响）。

<raise>引发的事件处理在执行可以执行内容期间要像对待其他事件一样。特别指出，错误事件不能被移出队列。

### 扩展性可执行模块

【此部分是标准】

具体实现可以提供额外的可执行内容对应实现的特性。特定的平台功能是不限制的，除了必须不会引起transition和状态的变化（间接引发事件触发transition除外）。注意状态机对待可执行内容触发的transition像一个单独的阻塞操作直到可执行内容执行完毕。例如，当要转入S状态，解析器会停止处理任何事件和跳转直到S 状态的<onentry>完成。这些对于可执行内容是比较重要的，包括特定平台的扩展功能。

## 状态机数据模块和数据操作

### 介绍

【此部分是介绍】

数据模块提供了在状态机内部存储、读取、修改数据集合的能力。此功能并没有指定特殊的数据类型，但是相对定义了一组抽象的能力可以通过各种语言实现，像ECMAScript或者XML/XPath。可以选择他们支持的数据模型来实现，除此之外内在的数据结构，数据模块定义表达式集合。这些表达式在数据模型里被用在指定的地方，将计算后的值赋值给本地，或者当作布尔条件。最后，数据模型包括系统变量集合，见“系统变量定义”，被fsm解析器自动维护。

数据模型是通过<datamodel>元素定义的，其中包含了零或多个<data>元素，每一个定义了单独的数据元素初始值。这些值可是内部的或者是外部的。可以通过可执行脚本更新。最终，可以被<script>元素在脚本语言中使用。

### <datamodel>

【此部分是标准】

<datamodel>是用来封装任意数量的<data>元素的，每一项定义了一个单独的数据对象，数据对象的确切性质依赖于数据模型使用的语言。

#### 属性明细

none

#### 子元素

<data>元素可以出现0到多次，每个实例被命名为数据元素。

### <data>

【此部分是标准】

<data>元素是用来声明数据和初始值。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| id | true |  | ID | none |  | 此项数据的名称 |
| expr | false | none | 表达式 | none |  | 数据值的表达式。 |

#### 子元素

none

### <scriptmodel>

【此部分是标准】

<scriptmodel>元素为状态机提供了执行脚本能力，可以封装任意数量的<script>元素。

#### 属性明细

none

#### 子元素

<script>元素可以出现0到多次，每个实例中包含一段合法的脚本或脚本文件，包含在scriptmodel元素下的script元素忽略cond属性。

状态机初始化后，将会执行此模块。

### <script>

【此部分是标准】

<script>元素在状态机中添加了脚本执行能力，可以嵌入脚本操作状态机中定义全局变量和局部变量。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| src | false |  | NKTOKEN | none | 文件名称 | script元素包含的外部文件，文件中是脚本内容，与在元素内编写代码效果是一致的。 |
| cond | false | 状态机可以执行的脚本 | 布尔表达式 | true | 状态机所支持的脚本表达式 | 执行元素匹配的一个条件，计算值为true时，此条件通过。 |

#### 子元素

出现在<script>元素中的脚本将会被执行。

在一个符合逻辑的状态机文档中，必须指定内容。

fsm解析器当遇到可执行内容中的<script>元素时，必须执行其中的脚本内容

### 表达式

【此部分是标准】

fsm 包括三种表达式，将在以下描述。不同的数据模型将会支持不同的语言。但是有些表达式确定的属性常量将会忽略语言。

#### 条件表达式

条件表达式是被用在’cond’属性中，一个条件表达式布被计算成一个布尔值，如果计算过程中遇到错误，fsm 解析器必须将表达式的值取’false’并且产生’error.execution’错误到内部事件队列中。条件表达式中的变量依赖于数据模型。在一个符合逻辑的fsm文档中条件表达式不应该影响其他的数据。

#### 本地表达式

本地表达式是指被用在执行模块中，包含数据模型的表达式。具体的表达式依赖于数据模型。

#### 合法的数据值和值类型表达式

字符串常量或者值常量。

### 系统变量

【此部分是标准】

FSM 解析器必须维护一部分在程序中比较有用的数据，以提供在状态机中可以使用程序中的数据功能，这部分数据我们定义为“系统变量”,实现必须提供以下系统变量，还可以包含其他的。

1、\_event。FSM解析器必须使用变量’\_event’控制包含当前事件名称和数据的结构。结构的详细类型依赖于被使用的数据模型。当一个事件被从内部或外呼事件队列进行处理时FSM 解析器必须绑定\_event变量，并且必须一直保持到下一个事件被处理。当第一个事件到来前FSM解析器不能绑定\_event变量。因此当状态机启动时\_event是未绑定的。

2、\_sessionid。当状态机启动当前session时FSM解析器必须绑定变量\_sessionid。解析器必须保持此变量直到session结束。

3、\_name。在状态机启动加载时FSM解析器必须绑定变量\_name为<FSM>的’name’属性。解析器必须保持变量直到session结束。

设置系统变量会在未来的版本中扩展。变量名称以’\_’开始表示是被系统使用的。一个符合逻辑的FSM 文档在其他变量中必须不以’\_’开头。

这些变量的具体实现会在脚本语言中使用。当遇到在试图改变系统变量时，解析器必须产生一个错误’error.execution’并禁止改变变量值。

#### 事件的内部结构

事件的内部结构反映了\_event变量。这个变量可以被<event>的判断使用或者在脚本语言中使用。

FSM解析器必须保证event事件有以下的字段，不管是内部事件还是外部事件。

1. name。是用来表示事件名字的字符串。FSM解析器必须设置事件的name字段。将被用来匹配<event>的’event’属性值。注意<event>也可以通过’cond’属性来添加布尔表达式。
2. type。此字段用来描述事件类型。标志事件来源。比如’internal’（内部事件），’external’（外部事件）。
3. sendid。如果发送实体指定了此值，解析器必须设置此字段的值。
4. data。此字段表示发送者在事件中包含了什么数据，FSM解析器为了能匹配数据可以重定义数据格式，但不能修改值。如果不能转换，此字段必须为空并且产生一个’error.execution’在内部事件队列中。

## 状态机外部通信

### 介绍

【此部分是说明】

外部通信能力允许FSM和外部模块发送和接受事件。 <send>提供了”fire and forget” 能力发送事件到任意目标，包括其他的FSM会话。具体的事件格式依赖于使用了那一种IO处理，每种实现会支持一种或多种处理，并且FSM作者可以选择一种合适的类型终端。

### <send>

【此部分是标准】

<send>元素是用来向外部模块发送事件和数据，包括外部FSM解析器，或者在当前session中引发事件。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| event | false | 不能和’eventexpr’同时出现 |  | none |  | 标志事件名称的字符串 |
| eventexpr | false | 不能和’event’同时出现 | 值表达式 | none |  | FSM解析器必须计算他的值并代替’event’值。 |
| target | false | 不能和targetexpr同时出现 |  |  |  | 指定将要发送的目标 |
| targetexpr | false | 不能和target同时出现 | 值表达式 |  |  | 此表达式是状态机可以执行的一段脚本，脚本的结果为指定将要发送的目标 |
| type | false | 不能和typeexpr同时出现 |  | none | 依赖于具体实现 | 指定发送消息的类型。依赖于具体的实现平台。 |
| typeexpr | false | 不能和type同时出现 | 值表达式 | none | 依赖于具体实现 | 此表达式是状态机可以执行的一段脚本，脚本的结果为指定发送消息的类型。依赖于具体的实现平台。 |
| id | false | 不能和’idexpr’同时出现 |  |  | 有效字符串 | 字符串的字面值用来标志属于那个<send>。 |
| idexpr | false | 不能和’id’同时出现 | 值表达式 |  | 有效表达式 | 将会生成一个id值。 |
| from | false | 不能和fromexpr同时出现 |  | none | 消息发送者 | 消息的发送者。 |
| fromexpr | false | 不能和from同时出现 | 值表达式 | none | 消息发送者 | 此表达式是状态机可以执行的一段脚本，脚本的结果为消息的发送者。 |
| dest | false | 不能和destexpr同时出现 |  | none | 消息接收者 | 消息的接收者。 |
| destexpr | false | 不能和dest同时出现 | 值表达式 | none | 消息接收者 | 此表达式是状态机可以执行的一段脚本，脚本的结果为消息的接收者。 |
| cond | false | 状态机可以执行的脚本 | 布尔表达式 | true | 状态机所支持的脚本表达式 | 执行元素匹配的一个条件，计算值为true时，此条件通过。 |

#### 子元素

任何xml类型的元素，子元素中的内容是send元素的扩展功能，可以携带任何xml节点类型的参数，具体实现依赖于平台。在子元素中需要指定一个xml 属性”type”，当type 属性为”script”时，此子元素的内容是执行脚本，需要在执行send元素的时候执行，其他情况或者是”text”，此子元素的内容是字符串常量。

一个符合逻辑的FSM文档必须指定’event’、’type’,’target’。

#### 发送目标

<send>操作的target指定了事件的目的地。目标由’target’属性表明或者’targetexpr’属性执行结果得出。

#### 发送类型

<send>操作的type指定了发送到目标后目标程序该怎么处理。

#### 数据传递

<send>元素中如果需要将更多的消息发送给其他模块，需要在<send>元素中添加属性和添加子元素两种方式，这两种方式也可以一起使用。

1. 在<send>元素中添加属性，添加的属性FSM状态机不做修改，保持原格式发送给目标。
2. 在<send>元素中添加子元素，在子元素中有一个参数type用来指定子元素的内容是执行脚本或者是字符串常量，将保持xml格式将数据发送给目标。

额外的数据传递都需要目标模块能够对接收到的数据进行正常的处理。

#### 实现

定义一个虚拟类，有一个字符串属性（name）是实例的描述（对应与send的target目标），此类中有一个虚函数负责处理状态机中的send 结果，具体的功能模块必须继承此类将虚函数实现。状态机初始化时将实现类的指针添加到FSM状态机中（FSM状态机中需要有一个map来保存子类的指针），当遇到<send>元素时，状态机根据’target’属性查找内部的map，找到对应的实现类指针，并调用虚函数。

## 一致性

【此部分是标准】

### 文档一致性

以下的一致性描述所有的FSM文档都要满足。

1. 文档的根元素必须是<FSM>。
2. 文档必须符合在这里指定的所有语法约束，包括所有的”属性约束”,在属性表格中的”有效值”字段，和包含的子元素定义。
3. 文档任何添加的语法必须符合在数据模型中定义的约束。

## FSM解析器算法

此部分包含了FSM解析的算法。可以按照自己选择的任何方法实现FSM的解析，但是必须使用以下定义的算法。

### 非正式语义

【此部分是说明】

以下的定义、原则和约束是为规范算法提供了一个背景，为了更好的理解。

### 预定义

#### 状态

<state>元素

#### transition目标

一个 state。

#### 原子状态

没有子状态的状态。

#### 组合状态

包含其他状态的状态。

#### 源状态

包含transition 的状态相对于此transition是源状态。

#### 目标状态

transition 中的’target’属性所指定的状态。

#### 外部事件

出现在外部事件队列中的事件，一般情况是由外部模块发送的。

#### 内部事件

出现在内部事件队列中的事件，这样的事件一般是由平台自己引发的。

### 原则和约束

我们在这里写出来一些额外原则和约束，在语义上，FSM应该符合它：

#### 封装

FSM处理器是一个单纯的事件处理器，只有一个办法将数据传递给FSM状态机，那就是发送外部事件。只有一个方法从FSM状态机获取数据那就是接收事件。

#### 执行完成

一个状态机执行完成的语义是，执行完一个外部事件，并且执行完由于此外部事件而产生的微步。

#### 结束

一个微步执行是一定会结束的，但是一个宏步不一定会结束，因为它可能会包含无限的微步，这样的语法是允许的。

### 算法

#### 数据类型

#### 全局变量

以下的变量时状态机对象的全局变量

global internalQueue

global externalQueue

global running

#### 函数和功能

##### 函数mainEventLoop()

此函数是状态机的事件执行函数，执行事件队列中的事件，或者running标志为false。

在最外层循环中，状态机执行外部事件。当执行完此外部事件后，继续执行当前存在的内部事件，内部事件完成后，判断是否状态机结束事件，如果是结束事件将running标志为false跳出外部循环。如果外部队列和内部队列都为空跳出外部循环。

## Schema

## 数据模型

<datamodel>元素中包含了状态机中全局范围内有效的数据，必须保证在文档中其他任何地方可以通过脚本进行访问这些元素。

## 外部模块

### XML 模块

状态机的基本描述是基于xml语言的，需要选择一个合适的xml解析器来实现状态机。除了需要支持基本的xml操作外还需要支持XPath标准。

### 日志模块

状态机需要日志记录功能，此部分模块可以使用合适的开源项目。

### 脚本解析模块

状态机中需要支持一定的脚本语言，一般脚本语言的解析都有实现，可以使用合适的开源项目，将开源项目集成到系统中。

## 代码实现

### 库的选用

程序中的基本功能都是使用xml来描述，所以都xml库的依赖比较重，需要选用一个合适的xml来实现程序，程序的设计是想基于夸平台来实现，所以在xml库的选择上需要选择夸平台的xml来实现。开始设计时选择的xerces xml库，基于C++ 方式实现，在Windows和Linux中都很成熟。在使用xerces实现状态机的过程中遇到一个问题，xerces在实现xpath的时候遇到了异常，本以为这是一个完善的xml库，没有怀疑是库的问题，在分析了源代码后发现是对xpath只是实现了一部分。此时面临着两个选择修改xerces源代码、换另一个xml库。对xerces的xpath部分源代码分析了2天后认为修改代码需要的工作量太大，又选取了libxml2 来实现xml功能。Libxml2 是一个比较完善的xml库，实现了w3c中的dom和xpath定义，在windows和linux上有很多应用，网上的评论也很好，美中不足的是没有C++版本的实现。虽然也可以和C++项目良好的兼容，但在代码的编写方式上风格不统一。在调试过程有遇到了一个问题，在使用xmlParseFile函数时总是出现无法捕获的异常，造成程序直接退出，后发现是受log4cplus库的影响，只要在程序中使用了log4cplus::initialize() 函数，就一定会发现此异常，具体原因不明，将log4cplus升级成1.1.2版本后暂时避免了此问题，但是使用log4cplus静态库版本还是会发生异常，此问题暂时不做处理，等基本功能完成后再分析。

### 日志模块的选用

选择了log4cplus 开源项目作为状态机的日志模块，此项目支持跨平台使用。

### 脚本解析模块选用

经过考虑在状态机中先实现对JS脚本的支持，JS脚本的解析器选用了SpiderMonkey项目。

## 问题

### 不满足执行条件

问题：如果在一个状态时，收到的事件执行前需要执行其他操作，操作成功后才能执行当前事件。如东进语音卡 放音前需要将通道设备和媒体设备连接，在当前状态下收到放音事件，当前条件下不能直接放音，需要先执行连接媒体操作，当返回成功事件后才能放音。现在的状态机事件是无记忆功能，收到放音事件后，执行了连接媒体操作就会放弃放音事件。

### 组合操作

问题：组合操作，当收到的事件中需要执行的操作不是原子操作时，此状态机难于执行。例如当收到放音事件，事件中包含多个语音来源，比如一个是语音文件，一个是内存索引放音，由于硬件的原因，这两个个语音需要播放完成一个后再播放另一个。

### 多个事件逻辑错误

当通道上连续收到多个事件，但是状态机的处理事件不及时的情况下，会造成问题。例如：一个通道上连续收到三个事件“呼入”、“挂机”、“呼入”，只要这三个事件的间隔足够小，状态机收到第一个“呼入”事件时，通道的状态已经变成 第二个电话的“呼入”。

假设状态机的流程是 呼入->应答->放音，那么将会产生的结果是：  


此情况下回收到呼入2的两次 摘机事件，第一次是成功，第二次是 重复摘机 事件。也有更严重的情况会影响电话接起。  


# 软电话设计说明

## 程序描述

此程序模块是系统中坐席人员操作的接口，一般称作软电话或SoftPhone，是系统应用人员最重要的系统接口，坐席人员主要通过此程序和系统交互，完成系统中的所有业务功能。此程序包括界面部分和逻辑部分，程序在每个坐席的客户端计算机上运行，依靠TCP端口和CTIServer交互。

## 功能

此程序是对物理电话的一种逻辑功能映射，然后再在此程序上扩展出适应业务需求的功能，包括物理电话上的语音通话控制功能（软电话上并不包括语音处理功能，以后可能会包含。）和一些业务、操作人员相关的功能。



软电话是CTIServer和使用人员之间的桥梁，坐席人员的操作通过软电话发送给CTIServer，CTIServer将结果发送给软电话处理。

## 性能

此程序是在客户端运行，是一个实时控制程序，要求及时性，对延迟为人员感觉不到为底线，兼容性要求尽量做到不依赖终端业务系统。

## 输入项

软电话可以处理输入消息和输出结果，输入消息主要来自3个方面：

1. 用户对软电话的操作。
2. 其他程序调用软电话的程序接口。
3. CTIServer发送给软电话的消息。

### 用户界面操作

软电话提供界面接口供坐席人员操作，软电话的所期望有功能都需要在界面上提供。在界面上相应的动作需要编写正确的功能函数。



### API

1、登陆函数 Login(string UserID,string PassWord);

2、登出函数 Logout();

3、应答函数Answer();

4、挂机函数Onhook();

### 来自CTIServer的消息

1、登录结果消息

<message name=”Login” >

< result> success </ result>

</message>

2、振铃消息

<message name=”Ringing” >

< CallerNumber ></ CallerNumber >

< CalleeNumber ></ CalleeNumber>

< SkillGroup ></ SkillGroup >

</message>

3、应答成功消息

<message name=”AnswerResult” >

< result> success </ result >

</message>

4、挂机消息

<Event name=”offline” ></Event>

## 输出项

输出项包括经处理后需要返回的信息，包括：

1. 给CTIServer的信息。
2. 返回给接口函数信息。
3. 需要在界面显示的信息。

### 软电话发送给CTIServer的消息

1、登录消息

<cmd Name=”Login”>

< AgentID ></ AgentID >

< PassWord ></ PassWord >

< Extension ></ Extension >

</cmd>

2、登出消息

<cmd Name=”LogOut” >

< AgentID ></ AgentID >

</cmd>

3、应答命令

<cmd Name=”Answer” >

< AgentID ></ AgentID >

</cmd>

4、挂机命令

<cmd Name=”OnHook” >

< AgentID ></ AgentID >

</cmd>

### 发送给API的消息

1、来电振铃消息

Ringing(string CallerNumber,string CalleeNumber);

### 界面显示信息



## 流程逻辑

在以前设计的软电话状态中是在CTIServer和本地各处理一个状态机，经常会造成两方的状态冲突。现在的设计中软电话无本地状态（有几个极少数的状态，只是没有业务逻辑状态），软电话的状态在CTIServer中保存，软电话客户端只是显示从CTIServer收到的状态。

软电话状态图



软电话的全部流程如下

### 初始化状态

软电话刚启动时处于初始化状态，此状态是读取配置，初始化一些信息。

### 未登录状态

初始化完成后跳转到未登录状态。

### 登陆中状态

坐席输入用户名和密码登陆后，软电话跳转到登陆中状态，等待CTIServer的登录结果。

### 服务器返回状态

软电话收到CTIServer的消息，成功后进入 一个“服务器返回状态（暂时叫这个名字吧）”，此状态下接受CTIServer返回的状态、消息、事件。软电话程序需要解析这些消息，改变软电话的状态显示，提示警告信息等功能。

## 算法

### 连接方式

软电话通过TCP/IP连接将消息发送给CTIServer，每个软电话可以单独和CTIServer建立一个连接，也可以将多个软电话的消息通过一个连接发送。软电话发送和接收的消息通过参消息中的AgentID来区分。

### 状态机



上图是软电话的状态，初始状态、未登录状态、登录中状态这三个状态在软电话程序中维护，登录成功后软电话程序不需要自己维护状态，要根据CTIServer返回的状态消息决定自己的状态。

### 软电话消息发送

软电话发送给CTIServer的消息都是xml格式，消息中包含消息类型，消息名称，AgentID，参数。

例如：

<cmd Name=”Login”>

< AgentID ></ AgentID >

< PassWord ></ PassWord >

< Extension ></ Extension >

</cmd>

Cmd是消息类型，消息类型包括cmd、event、msg（message）三种。

软电话对CTIServer的操作、命令是cmd消息，对CTIServer的事件通知，没有具体的操作是event消息，对CTIServer的消息通知使用msg消息。

Name 属性中是消息名称，指定此消息所要做的操作。

子元素是消息中的参数。

### 软电话消息接收

如果是每个软电话和CTIServer建立单独的连接，CTIServer发送的消息直接到达软电话。

如果是多个软电话使用同一个连接和CTIServer交互，CTIServer发送给软电话的消息经过同一个端口发出，消息接收器，需要将消息解析，根据AgentID发送到指定的软电话。

CTIServer 发送给软电话的消息有state（状态）、event（事件）、msg（消息）三种类型。

<state name="登陆中状态" id="BDBD5896-AA13-4A56-F646-451314AA7B8D" Memo="">

<parameter></parameter>

</state>

<event name=”” memo=”” ></event>

<msg name=”” memo=””></msg>

### 软电话消息解析

软电话收到消息后需要对xml文本进行解析。

State消息是状态消息，解析后根据结果改变软电话的状态。

Event消息是事件消息，暂时无定义

Msg消息是通知消息，例如警告消息等，需要根据消息做出相应提示。

### 软电话API

将软电话发送的xml消息封装到一个函数中，定义好需要传递的参数，函数体根据参数组合成像对应的xml消息，然后发送给CTIServer。

### 全局变量

软电话中有一些实现定义好的软电话变量，在整个软电话程序中生效，可以在程序中直接调用，软电话量名以”softphone\_”开始，其他部分和c 语言命名方式相同。

如$AgentID、$exstension。

### 软电话配置脚本

<?xml version="1.0" encoding="UTF-8"?>

<SoftPhone>

<config>

<softphone\_AgentID></softphone\_AgentID>

<softphone\_Extension></softphone\_Extension>

<softphone\_CTIServerAddr></softphone\_CTIServerAddr>

</config>

<StateMachine>

<state name="初始化状态" id="D8596801-45EA-7E50-FFCB-07CC5D38DC9E" memo="">

</state>

<state name="未登录状态" id="E3265B06-45C7-2F19-67C2-2E2D31DEB8D0" Memo="描述">

</state>

<state name="登陆中状态" id="BDBD5896-AA13-4A56-F646-451314AA7B8D" Memo="">

</state>

<state name="服务器返回状态" id="A39D4123-0709-90AD-6D7C-D938B76B87C5" Memo="">

</state>

</StateMachine>

</SoftPhone>

## 限制条件

暂无

## 测试计划

暂无

## 尚未解决的问题

暂无

## 代码实现

### Web版本

最初的软电话软件实现计划使用Web版本软电话。

#### 特点

使用Web版本软电话有的好处:

1、可以免安装和配置，使用软电话时直接访问Web服务器即可，软电话的配置可以在通过Web页面统一配置保存在Web 服务器上，不需要在每个客户端配置。

2、兼容性好，使用Web程序不需要为不同的客户端系统编写程序，因为现在大部分操作系统上都有浏览器程序。由于Web程序的标准在各个系统和浏览器程序中是统一的，所以编写的程序可以很容易的兼容多种系统和浏览器。

3、和业务系统集成，由于和CTI中间件集成的系统都是Web程序，使用Web软电话能更好的集成在一起。

4、减少CTIServer和软电话的连接，使用Web软电话可以在Web服务器上和CTIServer只建立一个TCP连接，各个软电话通过这一个连接和CTIServer交互。

坏处：

每次使用软电话都需要从服务器加载软电话程序，需要时间。

对Web程序不了解，会增加开发时间。

#### 结构图



实现方式如上图

CTIServer上监听一个TCP/IP端口，Web服务器连接此端口，和CTIServer建立连接，软电话启动时，通过浏览器访问Web服务，将软电话脚本程序加载到客户端运行，软电话和Web服务器通讯，软电话和CTIServer之间的消息都通过Web服务器转发。

现在的实现中还有几个问题没有验证

1、JS版本的程序是否在客户端运行。

2、由于事件消息都是xml格式，JS中是否有已经完成的xml解析器。

3、每个软电话需要独立的配置文件（softphone.xml），软电话加载时，是否可以从服务器上加载到本地。

#### 逻辑



#### 界面参考



# CardAdapter模块设计说明

## 程序描述

此程序模块是对语音卡功能的一个封装，完成基本的语音通话功能，根据语音卡API直接控制硬件，封装成可以供CTIServer和IVR直接操作的接口，根据CTIServer和IVR的消息对办卡完成一定的业务流程，对不同的语音卡需要编写不同的程序。

## 功能

控制语音卡，将语音卡的通道资源转换成分机资源，每个通道配置一个分机号码，可以实现分机之间互相通话，其他接口调用时，直接调用分机号码。根据收到的命令消息，转化成可以直接用板卡API实现的逻辑，包括所有语音卡上支持的功能，如：放音、收号、录音、信号音等。



## 性能

此程序是在服务端运行，每一种板卡只有一个程序实例，是一个实时控制程序，要求及时性，对延迟为人员感觉不到为底线。

## 输入项

输入项包括来自CTIServer和IVR的命令，和来自语音卡的事件。

### 来自IVR的放音消息

<cmd Name=Play>

< extension></ extension >

< files >

<file></file>

</ files >

</cmd>

### 来自CTIServer的分机注册消息

<cmd name=register>

<extension type=FXS></extension>

</cmd>

### 来自CTIServer的呼叫分机消息

<cmd name=call>

<src></src>

<dest ></dest>

</cmd>

### 来自CTIServer的录音消息

<cmd name=Record>

<extension></extension>

</cmd>

## 输出项

输出项包括经处理后需要对CTIServer和IVR返回的信息和对语音卡的操作。

### 返回给IVR的放音结果消息

<msg Name=PlayResult>

< Extension ></ Extension >

<Result></Result>

<Reason></reason>

<memo></memo>

</msg>

### 返回给CTIServer的分机注册消息

<msg Name=RegisterResult >

< Extension ></ Extension >

<Result></Result>

<Reason></reason>

<memo></memo>

</msg>

### 返回给CTIServer的呼叫分机结果消息

<msg Name=CallResult >

<src></src>

<dest></dest>

<Result></Result>

<Reason></reason>

<memo></memo>

</msg>

### 返回给CTIServer的录音消息

<msg Name=RecordResult >

< Extension ></ Extension >

<Result></Result>

<Reason></reason>

<memo></memo>

</msg>

## 流程逻辑

流程图



步骤：

1、加载配置文件

2、根据配置文件内容，初始化语音卡资源。

3、为每个初始化后的分机生成一个状态机，状态机脚本在配置文件中指定。

4、初始化TCP/IP监听端口，等待CTIServer和IVR的连接。

5、收到CTIServer或者IVR的消息后放入各自消息队列，等待处理模块处理。

6、处理模块根据收到的消息执行状态机。将状态机返回的结果传递给命令处理模块。

7、命令处理模块根据结果对语音卡操作或者给CTIServer 、IVR发送消息。

## 算法

### 连接方式

程序根据配置文件为CTIServer和IVR各建立一个TCP/IP监听端口，等待连接，连接后新建立线程处理CTIServer和IVR通讯。

### 消息队列

从端口接收后的消息取出后先放入队列，等待处理，此队列采用先进先出方式。

### 配置项

配置内容从作用范围上分为全局配置、分机配置两类。

#### 全局配置

1、分机号码长度

2、拨号间隔时长

3、号码收齐标识符

#### 分机配置

1、分机号

2、通道号

3、状态机

4、分机类型

### 状态机

每个程序配置的分机需要关联一个状态机，事务处理模块根据状态机的命令执行相应的功能。

#### 分机外呼



#### 分机呼入



#### 状态描述

在每个状态下，收到事件后经匹配进入下一状态，进入状态后会执行此状态下的已些动作。

例如 分机处于空闲状态，收到分机摘机事件，改变为摘机状态，到摘机状态后会执行

1、对分机发送等待拨号音。

2、发送消息给CTIServer。

3、启动分机计时器。

<state name="摘机状态" displayname="摘机状态" id="C6E00A9F-90F1-5D53-88D2-FEDA7AE85192" memo="">

<action application="playtone" data="等待拨号音"></action>

<action application="SendToCTIServer" data="extension=channel\_var\_extension 摘机"></action>

<action application="timer" data="10000"></action>

<condition filed="channel" expression="dialtone">

<action></action>

</condition>

</state>

### 消息发送

发送给CTIServer的消息都是xml格式，消息中包含消息类型，消息名称，extension，参数。

例如：

<event Name=”offhook”>

< Extension ></ Extension >

</event>

event是消息类型，消息类型包括cmd、event、msg（message）三种。

Name 属性中是消息名称，指定此消息的内容。

子元素是消息中的参数。

根据要发送的目的地，将消息放到对应的发送队列中，消息发送线程会自动将队列中的消息发送。

### 消息接收

CTIServer 和IVR会根据业务流程对CardAdapter发送消息，CardAdapter收到消息后进行解析，根据消息执行状态机流程。

CTIServer 发送给软电话的消息有cmd（命令）、event（事件）、msg（消息）三种类型。

IVR发送给软电话的消息有event（事件）、cmd（命令）等。

例如：<cmd Name=Play>

< extension></ extension >

< files >

<file></file>

</ files >

</cmd>

<msg name=”” memo=””></msg>

### 软电话消息解析

收到消息后需要对xml文本进行解析。

cmd消息是命令，例如IVR的放音命令。

Event消息是事件消息，例如分机的摘机、挂机、拨号事件。

Msg消息是通知消息，例如警告消息等，需要根据消息做出相应提示。

### 变量

按照变量的作用域的不同，可以分为全局变量和分机变量两种，这些变量都都已在脚本或配置文件中引用。

#### 全局变量

程序中有一些实现定义好的变量，在整个程序中生效，可以在程序中直接调用，CardAdapter量名以”card\_”开始，其他部分和c 语言命名方式相同。

如card\_KeygoeIP、card\_KeygoePort。

#### 分机变量

分机变量以”exten\_”，开始，其他部分和c语言命名方式相同，变量在每个分机或通道中有效。

### CardAdapter配置脚本

<?xml version="1.0" encoding="UTF-8"?>

<CardAdapter>

<config>

</config>

<StateMachies>

<StateMachie1>

<state name="空闲状态" displayname="空闲状态" id="65E376F0-2FA9-4F4C-3A2B-FEEAF714BC20" memo="">

<action></action>

<condition filed="channel" expression="offhook">

<action application="goto" data="C6E00A9F-90F1-5D53-88D2-FEDA7AE85192"></action>

</condition>

</state>

<state name="摘机状态" displayname="摘机状态" id="C6E00A9F-90F1-5D53-88D2-FEDA7AE85192" memo="">

<action application="playtone" data="等待拨号音"></action>

<action application="SendToCTIServer" data="extension=channel\_var\_extension 摘机"></action>

<action application="timer" data="10000"></action>

<condition filed="channel" expression="dialtone">

<action></action>

</condition>

</state>

</StateMachie1>

</StateMachies>

</CardAdapter>

### StateMachine类

状态机类，负责维护分机的状态，接收事件返回状态结果。

Class StateMachine{

}

此类的主要功能和接口

1、以xml文件为参数的构造函数。

2、以xml Node指针为参数的构造函数。

3、返回当前状态

4、传递事件

5、返回命令

6、返回消息

### extension 类

Extension类是程序中通道资源的基础类，包含StateMachine类变量，定义一些分机资源的接口，使用虚拟类。设计到具体的硬件时，需要继承此类，实现此类定义的虚函数。

Class extension{

}

## 限制条件

暂无

## 测试计划

暂无

## 尚未解决的问题

暂无

## 代码实现

使用的开源模块xml解析模块、日志模块。

### Keygoe版本

最初的CardAdapter 计划使用东进Keygoe交换机实现。

#### 特点

暂无

#### 结构图



实现方式如上图

根据配置文件在CardAdapter上为CTIServer和IVR各监听以个TCP/IP端口，CTIServer和IVR会主动连接此端口，CardAdapter收到连接后会新建立一个通信线程，来维护和CTIServer和IVR的连接。

#### 逻辑

# IVR流程模块设计说明

## 程序描述

此模块在系统中负责IVR脚本的解析，系统启动时将IVR脚本加载到内存中，当收到事件时，根据IVR脚本的状态机处理执行相应的业务流程。当有多路通话时，在程序中有多个状态机处于活跃状态，这几个状态机之间是相互独立的。此模块是基于状态机实现的，属于状态机的上层应用程序。

## 功能

解析由状态机定义的IVR流程，完成业务处理。在IVR业务中会有多种原子功能相互组合，形成复杂的业务流程。各个功能的组合类似于Visio中的图形组合，各个原子功能是使用状态机中的元素来描述。

### 语音播放

此原子功能使用状态机中的状态元素来定义，一切格式需要符合状态机语法。当执行到此功能时，状态机会执行放音操作（让其他模块）。

#### 属性明细

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **名称** | **必须** | **约束** | **类型** | **默认值** | **有效值** | **描述** |
| id | true |  |  |  | 有效字符串 | 功能id，在此session中必须唯一 |
| name | false |  |  | none | 有效字符串 | 名称描述 |
| type | true | 必须为”play” |  |  |  | 此功能的类型 |
| memo | false |  |  | none | 有效字符串 | 此功能的描述 |

#### 子元素

1、<onentry><send></send></onentry> 元素，包含需要播放的语音文件，在进入到此功能后自动执行的元素。<send>元素的target属性表示接收此命令的模块，type属性在这里一直是”cmd”，from 属性表明此session，dest属性表示接收此消息的session，event属性表明发送的消息是什么命令，这里固定为”playFile”。具体需要播放的语音文件，包含在此元素的子元素<file>中。

2、<event>元素，此功能下可以处理的事件，当执行语音播放功能后，必需要可以处理playfinished（放音完成）和dtmf（按键打断）两个事件，其他的事件可以作者自己添加。

#### 示例

<state id="welcome" name=”” type=”” memo=””>

<onentry>

//发送语音播放命令

<send target="acd" type="cmd" idlocation="extension.number; " from="extension.number;" dest="forward" event="playFile" >

<file>'./wav/welcome.wav'</file>

</send>

</onentry>

<event event=”playfinished” >

//执行功能跳转或者其他操作

</event>

<event event=”dtmf” >

//执行功能跳转或者其他操作

</event>

</state>

### 按键接收

此功能可以介绍并存储收到的按键。

## 性能

是一个实时的多并发处理程序，要求对事件处理迅速。

## 输入项

IVR模块是一个业务解析器，主要是将业务流程解析成其他模块可以执行的命令，然后发送给对应的模块，其他模块执行完毕后将结果发送给IVR模块。IVR模块可以接收的消息有以下：

### 放音完成消息

<event event=”playfinished”/>

## 输出项

IVR模块是一个业务解析器，主要是将业务流程解析成其他模块可以执行的命令，然后发送给对应的模块。IVR模块发送给其他模块的消息有：

### 放音命令

<send target="acd" type="cmd" idlocation="5000" from="5000" dest="1000" event="playFile">

<file>'./wav/welcome.wav'</file>

</send>

## 流程逻辑

IVR模块的逻辑是解析IVR脚本，并根据状态机的运行和CTIServer和CardAdapter交互。



### 转人工过程

在IVR中转人工后,IVR将转人工的参数传送给CTI,由CTI来执行转人工或组功能,IVR流程处于等待过程,CTI将转接结果返回会IVR流程。

## 算法

代码的实现可以自己选用任何合适的语言和算法，但是必须实现以下定义的算法。

## 限制条件

暂无

## 测试计划

暂无

## 尚未解决的问题

暂无

## 代码实现

# ACD模块设计说明

## 程序描述

ACD(Automatic Call Distribution)即自动呼叫分配，将接入的呼叫中心系统的来电、呼出电话按特定规则自动转接到正确的座席、分机或进行其它自动处理如排队或留言等。

## 功能

### 交换功能

呼叫中心在本质上也是更加智能化的交换机，故呼叫中心ACD应具有程控交换机最基本的话务交换功能。

### 排队功能

ACD必须具有话务排队的功能，所谓排队是指在内线都忙的情况下，外部再打来一个电话时按一定规则进行等待，一旦内线空闲时进行接入。

### 路由功能

拨入ACD的话务可以根据一定的规则转发到响应的坐席终端或其他的一些的终端(如IVR、外呼时线路选择 )。

## 性能

### 实时性

ACD处理属于即时处理，要求1ms之内返回结果。

### 并发性

1000个/秒请求的并发能力。

## 输入项

Xml格式，例如：<from=”1000” dest=”acd” event=”0” data=””>

## 输出项

Xml格式，例如：<from=”acd” dest=”1000” cmd=”hangup” data=””>

<from=”acd” dest=”2000” cmd=”transfer” data=”1000”>

## 流程逻辑

ACD模块工作在话务控制层，负责电话的会话控制、路由选择。当一通电话在进行呼入时（包括呼入组、外呼、）状态机将呼入消息发送给ACD，ACD收到消息后启动一个状态机，由状态机根据配置规则处理此消息，处理结果为继续将电话前转或者挂机，如果是挂机，ACD返回处理结果。如果是前转ACD重新生成事件交给目标分机处理，目标分机将处理结果返回给ACD，如果是转接失败ACD根据脚本继续处理，此session结束时才销毁ACD状态机。

## 算法

使用状态机实现

## 限制条件

无

## 测试计划

无

## 尚未解决的问题

无

## 代码实现

# CTIServer模块设计说明

CTIServer模块是系统中各个模块的连接模块，处在系统中的节点位置，并负责和软电话客户端的连接、和媒体模块的交互。

## 和软电话之间的连接

CTI和软电话通信使用两种方式，1、服务器和每个客户端建立单独的连接。2、服务器只使用一个连接和软电话事件代理连接，所有的客户端消息通过此连接发送给服务器，使用AgentID区分是哪个客户端的事件，此情况适用于CTI服务器<->软电话事件代理<->软电话模式。

为了保持兼容性，CTI服务器的设计模式如下：

1. CTI服务器上启用一个监听端口，监听每个客户端发起的连接请求，连接后创建一个单独的连接和线程处理和这个客户端的事件，这个连接中只有一个客户端发起登录请求。
2. CTI服务器上启动一个监听端口，和上面的监听端口共用，软电话事件代理服务器发起连接请求，连接后创建一个单独的连接和线程处理软电话事件代理的事件，事件中包括多个软电话的登录、退出，状态事件。每个消息中都包含AgentID来区分消息的操作对象。
3. 为了保持兼容性，无论是那种客户端的连接模式，发送的消息中都需要包含AgentID，用来区分消息的操作对象。

## 软电话

### 初始状态



“初始状态”是软电话程序启动后的第一个状态，程序初始化一些参数，初始化完成后后自动跳转到“未登录状态”。

### 未登录状态



未登录状态时软电话还没有和CTIServer建立连接。

### 暂停状态



登录成功后自动跳转到暂停状态。

### 振铃状态



### 话后状态



### 通话状态



### 保持状态



### 就绪状态



### 转接中状态



### 会议中状态



### 会议状态

