|  |
| --- |
|  |
|  |
|  |
| 【openSCA 开发者手册】 |
|  |
| 【Developer - Guide】 |
|  |
|  |
|  |
|  |
| 版本V0.8 |
| 共 140页 |
|  |
|  |
|  |
|  |
|  |
|  |
| 上海介方信息技术有限公司 |
|  |
| 2019年7月 |
|  |

目 录

[1 概述 1](#_Toc15581764)

[1.1 什么是openSCA2.2.2？ 3](#_Toc15581765)

[1.2 如何获取openSCA2.2.2? 3](#_Toc15581766)

[1.3 文档章节介绍 3](#_Toc15581767)

[2 操作环境 5](#_Toc15581768)

[2.1 操作系统 5](#_Toc15581769)

[2.2 中间件和服务 5](#_Toc15581770)

[2.2.1 CORBA简介 5](#_Toc15581771)

[2.2.1.1 CORBA通信机制 6](#_Toc15581772)

[2.2.1.2 ORB 6](#_Toc15581773)

[2.2.1.3 接口定义语言(IDL) 7](#_Toc15581774)

[2.2.1.4 对象引用 7](#_Toc15581775)

[2.2.2 命名服务 7](#_Toc15581776)

[2.2.3 事件服务 8](#_Toc15581777)

[2.3 SCA核心框架 8](#_Toc15581778)

[3 openSCA框架核心流程 10](#_Toc15581779)

[3.1 框架启动流程介绍 10](#_Toc15581780)

[3.2 波形安装流程介绍 10](#_Toc15581781)

[3.3 波形卸载流程 11](#_Toc15581782)

[4 openSCA框架控制接口 13](#_Toc15581783)

[4.1 异常 13](#_Toc15581784)

[4.1.1 CORBA::Exception 13](#_Toc15581785)

[4.1.2 CORBA::SystemException 13](#_Toc15581786)

[4.1.3 CF::InvalidFileName 13](#_Toc15581787)

[4.1.4 CF::InvalidProfile 14](#_Toc15581788)

[4.1.5 CF::DomainManager::InvalidIdentifier 14](#_Toc15581789)

[4.1.6 CF::DomainManager::ApplicationInstallationError 14](#_Toc15581790)

[4.1.7 CF::DomainManager::ApplicationUninstallationError 14](#_Toc15581791)

[4.1.8 CF::PortSupplier::UnknownPort 14](#_Toc15581792)

[4.1.9 CF::LifeCycle::InitializeError 14](#_Toc15581793)

[4.1.10 CF::LifeCycle::ReleaseError 15](#_Toc15581794)

[4.1.11 CF::PropertySet::PartialConfiguration 15](#_Toc15581795)

[4.1.12 CF::PropertySet::InvalidConfiguration 15](#_Toc15581796)

[4.1.13 CF::UnknownProperties 15](#_Toc15581797)

[4.1.14 CF::Resource::StartError 15](#_Toc15581798)

[4.1.15 CF::ApplicationFactory::CreateApplicationRequestError 15](#_Toc15581799)

[4.1.16 CF::ApplicationFactory::CreateApplicationError 16](#_Toc15581800)

[4.1.17 CF::ApplicationFactory::InvalidInitConfiguration 16](#_Toc15581801)

[4.2 DomainManager 16](#_Toc15581802)

[4.2.1 UML 17](#_Toc15581803)

[4.2.2 identifier 17](#_Toc15581804)

[4.2.2.1 语义 18](#_Toc15581805)

[4.2.2.2 代码片段 18](#_Toc15581806)

[4.2.3 applications 19](#_Toc15581807)

[4.2.3.1 语义 19](#_Toc15581808)

[4.2.3.2 代码片段 19](#_Toc15581809)

[4.2.4 applicationFactories 19](#_Toc15581810)

[4.2.4.1 语义 19](#_Toc15581811)

[4.2.4.2 代码片段 20](#_Toc15581812)

[4.2.5 deviceManagers 20](#_Toc15581813)

[4.2.5.1 语义 20](#_Toc15581814)

[4.2.5.2 代码片段 20](#_Toc15581815)

[4.2.6 installApplication 21](#_Toc15581816)

[4.2.6.1 语义 21](#_Toc15581817)

[4.2.6.2 代码片段 21](#_Toc15581818)

[4.2.7 uninstallApplication 22](#_Toc15581819)

[4.2.7.1 语义 22](#_Toc15581820)

[4.2.7.2 代码片段 22](#_Toc15581821)

[4.3 Application 23](#_Toc15581822)

[4.3.1 UML 24](#_Toc15581823)

[4.3.2 identifier 25](#_Toc15581824)

[4.3.2.1 语义 25](#_Toc15581825)

[4.3.2.2 代码片段 26](#_Toc15581826)

[4.3.3 name 26](#_Toc15581827)

[4.3.3.1 语义 26](#_Toc15581828)

[4.3.3.2 代码片段 26](#_Toc15581829)

[4.3.4 getPort 27](#_Toc15581830)

[4.3.4.1 语义 27](#_Toc15581831)

[4.3.4.2 代码片段 27](#_Toc15581832)

[4.3.5 initialize 28](#_Toc15581833)

[4.3.5.1 语义 28](#_Toc15581834)

[4.3.5.2 代码片段 28](#_Toc15581835)

[4.3.6 releaseObject 28](#_Toc15581836)

[4.3.6.1 语义 29](#_Toc15581837)

[4.3.6.2 代码片段 29](#_Toc15581838)

[4.3.7 configure 29](#_Toc15581839)

[4.3.7.1 语义 30](#_Toc15581840)

[4.3.7.2 代码片段 30](#_Toc15581841)

[4.3.8 query 31](#_Toc15581842)

[4.3.8.1 语义 31](#_Toc15581843)

[4.3.8.2 代码片段 31](#_Toc15581844)

[4.3.9 start 32](#_Toc15581845)

[4.3.9.1 语义 32](#_Toc15581846)

[4.3.9.2 代码片段 32](#_Toc15581847)

[4.3.10 stop 33](#_Toc15581848)

[4.3.10.1 语义 33](#_Toc15581849)

[4.3.10.2 代码片段 33](#_Toc15581850)

[4.4 ApplicationFactory 33](#_Toc15581851)

[4.4.1 UML 34](#_Toc15581852)

[4.4.2 identifier 34](#_Toc15581853)

[4.4.2.1 语义 34](#_Toc15581854)

[4.4.2.2 代码片段 35](#_Toc15581855)

[4.4.3 name 35](#_Toc15581856)

[4.4.3.1 语义 36](#_Toc15581857)

[4.4.3.2 代码片段 36](#_Toc15581858)

[4.4.4 create 37](#_Toc15581859)

[4.4.4.1 语义 37](#_Toc15581860)

[4.4.4.2 代码片段 37](#_Toc15581861)

[4.5 DeviceManager 38](#_Toc15581862)

[4.5.1 UML 38](#_Toc15581863)

[4.5.2 identifier 39](#_Toc15581864)

[4.5.2.1 语义 39](#_Toc15581865)

[4.5.2.2 代码片段 40](#_Toc15581866)

[4.5.3 label 40](#_Toc15581867)

[4.5.3.1 语义 41](#_Toc15581868)

[4.5.3.2 代码片段 41](#_Toc15581869)

[4.5.4 registeredDevices 41](#_Toc15581870)

[4.5.4.1 语义 41](#_Toc15581871)

[4.5.4.2 代码片段 41](#_Toc15581872)

[5 应用组件接口 43](#_Toc15581873)

[5.1 UML 43](#_Toc15581874)

[5.2 getPort 43](#_Toc15581875)

[5.2.1 语义 44](#_Toc15581876)

[5.3 initialize 44](#_Toc15581877)

[5.3.1 语义 44](#_Toc15581878)

[5.4 releaseObject 44](#_Toc15581879)

[5.4.1 语义 45](#_Toc15581880)

[5.5 configure 45](#_Toc15581881)

[5.5.1 语义 45](#_Toc15581882)

[5.6 query 45](#_Toc15581883)

[5.6.1 语义 46](#_Toc15581884)

[5.7 start 46](#_Toc15581885)

[5.7.1 语义 46](#_Toc15581886)

[5.8 stop 46](#_Toc15581887)

[5.8.1 语义 47](#_Toc15581888)

[5.9 runTest 47](#_Toc15581889)

[5.9.1 语义 47](#_Toc15581890)

[6 逻辑设备接口 48](#_Toc15581891)

[6.1 Device 48](#_Toc15581892)

[6.1.1 UML 48](#_Toc15581893)

[6.1.2 identifier 49](#_Toc15581894)

[6.1.2.1 语义 49](#_Toc15581895)

[6.1.2.2 代码片段 49](#_Toc15581896)

[6.1.3 label 49](#_Toc15581897)

[6.1.3.1 语义 50](#_Toc15581898)

[6.1.3.2 代码片段 50](#_Toc15581899)

[6.1.4 usageState 50](#_Toc15581900)

[6.1.4.1 语义 50](#_Toc15581901)

[6.1.4.2 代码片段 51](#_Toc15581902)

[6.1.5 setUsageState 51](#_Toc15581903)

[6.1.5.1 语义 51](#_Toc15581904)

[6.1.5.2 代码片段 52](#_Toc15581905)

[6.1.6 adminState 52](#_Toc15581906)

[6.1.6.1 语义 53](#_Toc15581907)

[6.1.6.2 代码片段 53](#_Toc15581908)

[6.1.7 adminState(CF::Device::AdminType) 53](#_Toc15581909)

[6.1.7.1 语义 54](#_Toc15581910)

[6.1.7.2 代码片段 54](#_Toc15581911)

[6.1.8 operationalState 55](#_Toc15581912)

[6.1.8.1 语义 55](#_Toc15581913)

[6.1.8.2 代码片段 55](#_Toc15581914)

[6.1.9 softwareProfile 55](#_Toc15581915)

[6.1.9.1 语义 55](#_Toc15581916)

[6.1.9.2 代码片段 56](#_Toc15581917)

[6.1.10 compositeDevice 56](#_Toc15581918)

[6.1.10.1 语义 56](#_Toc15581919)

[6.1.10.2 代码片段 56](#_Toc15581920)

[6.1.11 allocateCapacity 57](#_Toc15581921)

[6.1.11.1 语义 58](#_Toc15581922)

[6.1.11.2 代码片段 58](#_Toc15581923)

[6.1.12 deallocateCapacity 58](#_Toc15581924)

[6.1.12.1 语义 59](#_Toc15581925)

[6.1.12.2 代码片段 59](#_Toc15581926)

[6.1.13 initialize 59](#_Toc15581927)

[6.1.13.1 语义 60](#_Toc15581928)

[6.1.13.2 代码片段 60](#_Toc15581929)

[6.1.14 releaseObject 60](#_Toc15581930)

[6.1.14.1 语义 60](#_Toc15581931)

[6.1.14.2 代码片段 61](#_Toc15581932)

[6.1.15 start 61](#_Toc15581933)

[6.1.15.1 语义 61](#_Toc15581934)

[6.1.15.2 代码片段 61](#_Toc15581935)

[6.1.16 stop 62](#_Toc15581936)

[6.1.16.1 语义 62](#_Toc15581937)

[6.1.16.2 代码片段 62](#_Toc15581938)

[6.1.17 configure 63](#_Toc15581939)

[6.1.17.1 语义 63](#_Toc15581940)

[6.1.17.2 代码片段 63](#_Toc15581941)

[6.1.18 query 64](#_Toc15581942)

[6.1.18.1 语义 64](#_Toc15581943)

[6.1.18.2 代码片段 64](#_Toc15581944)

[6.1.19 getPort 65](#_Toc15581945)

[6.1.19.1 语义 65](#_Toc15581946)

[6.1.19.2 代码片段 65](#_Toc15581947)

[6.1.20 runTest 66](#_Toc15581948)

[6.1.20.1 语义 66](#_Toc15581949)

[6.1.20.2 代码片段 67](#_Toc15581950)

[6.2 LoadableDevice 67](#_Toc15581951)

[6.2.1 UML 67](#_Toc15581952)

[6.2.2 load 68](#_Toc15581953)

[6.2.2.1 语义 68](#_Toc15581954)

[6.2.2.2 代码片段 69](#_Toc15581955)

[6.2.3 unload 69](#_Toc15581956)

[6.2.3.1 语义 70](#_Toc15581957)

[6.2.3.2 代码片段 70](#_Toc15581958)

[6.3 ExecutableDevice 70](#_Toc15581959)

[6.3.1 UML 70](#_Toc15581960)

[6.3.2 execute 71](#_Toc15581961)

[6.3.2.1 语义 71](#_Toc15581962)

[6.3.2.2 代码片段 72](#_Toc15581963)

[6.3.3 terminate 74](#_Toc15581964)

[6.3.3.1 语义 74](#_Toc15581965)

[6.3.3.2 代码片段 74](#_Toc15581966)

[6.4 AggregateDevice 74](#_Toc15581967)

[6.4.1 UML 75](#_Toc15581968)

[6.4.2 devices 75](#_Toc15581969)

[6.4.2.1 语义 75](#_Toc15581970)

[6.4.2.2 代码片段 75](#_Toc15581971)

[6.4.3 addDevice 76](#_Toc15581972)

[6.4.3.1 语义 76](#_Toc15581973)

[6.4.3.2 代码片段 76](#_Toc15581974)

[6.4.4 removeDevice 77](#_Toc15581975)

[6.4.4.1 语义 77](#_Toc15581976)

[6.4.4.2 代码片段 77](#_Toc15581977)

[7 端口及连接 78](#_Toc15581978)

[7.1 端口数据传输机制 78](#_Toc15581979)

[7.2 端口数据传输类型 79](#_Toc15581980)

[7.3 UsesPort接口 80](#_Toc15581981)

[7.3.1 UML 80](#_Toc15581982)

[7.3.2 getPort 80](#_Toc15581983)

[7.3.2.1 语义 81](#_Toc15581984)

[7.3.2.2 代码片段 81](#_Toc15581985)

[7.3.3 connectPort 81](#_Toc15581986)

[7.3.3.1 语义 82](#_Toc15581987)

[7.3.3.2 代码片段 82](#_Toc15581988)

[7.3.4 disconnectPort 83](#_Toc15581989)

[7.3.4.1 语义 83](#_Toc15581990)

[7.3.4.2 代码片段 83](#_Toc15581991)

[7.3.5 pushPacket 83](#_Toc15581992)

[7.3.5.1 语义 83](#_Toc15581993)

[7.3.5.2 代码片段 83](#_Toc15581994)

[7.4 ProvidesPort接口 84](#_Toc15581995)

[7.4.1 UML 84](#_Toc15581996)

[7.4.2 getPort 84](#_Toc15581997)

[7.4.2.1 语义 85](#_Toc15581998)

[7.4.2.2 代码片段 85](#_Toc15581999)

[7.4.3 connectSlot 85](#_Toc15582000)

[7.4.3.1 语义 85](#_Toc15582001)

[7.4.3.2 代码片段 86](#_Toc15582002)

[7.4.4 disconnectSlot 86](#_Toc15582003)

[7.4.4.1 语义 86](#_Toc15582004)

[7.4.4.2 代码片段 86](#_Toc15582005)

[7.4.5 getData 86](#_Toc15582006)

[7.4.5.1 语义 87](#_Toc15582007)

[7.4.5.2 代码片段 87](#_Toc15582008)

[7.5 外部程序与波形应用的连接 87](#_Toc15582009)

[8 ORB抽象接口 91](#_Toc15582010)

[8.1 UML 91](#_Toc15582011)

[8.2 bind\_object\_to\_string 91](#_Toc15582012)

[8.2.1 语义 91](#_Toc15582013)

[8.2.2 代码片段 92](#_Toc15582014)

[8.3 get\_object\_from\_string 92](#_Toc15582015)

[8.3.1 语义 92](#_Toc15582016)

[8.3.2 代码片段 93](#_Toc15582017)

[8.4 unbind\_string 93](#_Toc15582018)

[8.4.1 语义 93](#_Toc15582019)

[8.4.2 代码片段 93](#_Toc15582020)

[8.5 bind\_new\_context\_with\_string 94](#_Toc15582021)

[8.5.1 语义 94](#_Toc15582022)

[8.5.2 代码片段 94](#_Toc15582023)

[8.6 destory\_context 94](#_Toc15582024)

[8.6.1 语义 95](#_Toc15582025)

[8.6.2 代码片段 95](#_Toc15582026)

[8.7 ior\_to\_object 95](#_Toc15582027)

[8.7.1 语义 95](#_Toc15582028)

[8.7.2 代码片段 95](#_Toc15582029)

[9 域描述文件 97](#_Toc15582030)

[9.1 Software Package Descriptor(SPD) 99](#_Toc15582031)

[9.2 Software Component Descriptor(SCD) 99](#_Toc15582032)

[9.3 Software Assembly Descriptor(SAD) 100](#_Toc15582033)

[9.4 Properties Descriptor(PRF) 101](#_Toc15582034)

[9.5 Device Package Descriptor(DPD) 102](#_Toc15582035)

[9.6 Device Configuration Descriptor(DCD) 102](#_Toc15582036)

[9.7 Domain Manager Configuration Descriptor(DMD) 103](#_Toc15582037)

[10 波形应用及逻辑设备开发 104](#_Toc15582038)

[10.1 开发场景 104](#_Toc15582039)

[10.2 组件开发 105](#_Toc15582040)

[10.2.1 Sample\_Ctroller 107](#_Toc15582041)

[10.2.1.1 initialize 107](#_Toc15582042)

[10.2.1.2 getPort 108](#_Toc15582043)

[10.2.1.3 start 109](#_Toc15582044)

[10.2.1.4 stop 110](#_Toc15582045)

[10.2.1.5 releaseObject 110](#_Toc15582046)

[10.2.1.6 configure 111](#_Toc15582047)

[10.2.1.7 query 112](#_Toc15582048)

[10.2.2 TxRxComp 114](#_Toc15582049)

[10.2.2.1 start 114](#_Toc15582050)

[10.2.2.2 发送数据线程 115](#_Toc15582051)

[10.2.2.3 接收数据接口 116](#_Toc15582052)

[10.2.2.4 stop 116](#_Toc15582053)

[10.3 设备开发 117](#_Toc15582054)

[10.3.1 Zynq7035 118](#_Toc15582055)

[10.3.1.1 execute 118](#_Toc15582056)

[10.3.2 Zynq7035\_PS 119](#_Toc15582057)

[10.3.2.1 execute 119](#_Toc15582058)

[10.3.2.2 添加到父设备 120](#_Toc15582059)

[10.3.3 Zynq7035\_PL 121](#_Toc15582060)

[10.3.3.1 load 121](#_Toc15582061)

[10.3.3.2 unload 122](#_Toc15582062)

[10.3.3.3 添加到父设备 122](#_Toc15582063)

[11 术语表 125](#_Toc15582064)

[11.1 缩写 125](#_Toc15582065)

[11.2 名词解释 125](#_Toc15582066)

[11.2.1 Application 125](#_Toc15582067)

[11.2.2 ApplicationFactory 125](#_Toc15582068)

[11.2.3 API 126](#_Toc15582069)

[11.2.4 assemblycontroller 126](#_Toc15582070)

[11.2.5 Client客户端 126](#_Toc15582071)

[11.2.6 Component 126](#_Toc15582072)

[11.2.7 CORBA Component 126](#_Toc15582073)

[11.2.8 Core Application 126](#_Toc15582074)

[11.2.9 CF核心框架 126](#_Toc15582075)

[11.2.10 Device 127](#_Toc15582076)

[11.2.11 Device Configuration Description (DCD) 127](#_Toc15582077)

[11.2.12 Device Package Descriptor (DPD) 127](#_Toc15582078)

[11.2.13 Device Profile 127](#_Toc15582079)

[11.2.14 Domain域 128](#_Toc15582080)

[11.2.15 Domain Manager域管理器 128](#_Toc15582081)

[11.2.16 Domain Profile 128](#_Toc15582082)

[11.2.17 Event Service事件服务 128](#_Toc15582083)

[11.2.18 Event Channel事件通道 128](#_Toc15582084)

[11.2.19 Parent Device 129](#_Toc15582085)

[11.2.20 Port端口 129](#_Toc15582086)

[11.2.21 Properties Descriptor(PRF) 129](#_Toc15582087)

[11.2.22 Profile Descriptor 129](#_Toc15582088)

[11.2.23 Properties Descriptor 129](#_Toc15582089)

[11.2.24 Property 130](#_Toc15582090)

[11.2.25 Resource 130](#_Toc15582091)

[11.2.26 Software Assembly Descriptor (SAD) 130](#_Toc15582092)

[11.2.27 Software Component Descriptor (SCD) 130](#_Toc15582093)

[11.2.28 Software Package Descriptor (SPD) 130](#_Toc15582094)

[11.2.29 Software Profile 131](#_Toc15582095)

[11.2.30 Waveform 131](#_Toc15582096)

[11.2.31 Waveform Application 131](#_Toc15582097)

图索引

[图 1 openSCA操作环境结构图 5](#_Toc15583043)

[图 2 CORBA通信机制 6](#_Toc15583044)

[图 3框架启动流程序列图 10](#_Toc15583045)

[图 4波形安装流程序列图 11](#_Toc15583046)

[图 5波形卸载流程序列图 12](#_Toc15583047)

[图 6域管理器的UML 17](#_Toc15583048)

[图 7域管理器id 18](#_Toc15583049)

[图 8波形应用UML 25](#_Toc15583050)

[图 9波形应用id 25](#_Toc15583051)

[图 10波形应用name 26](#_Toc15583052)

[图 11波形应用端口信息 27](#_Toc15583053)

[图 12属性描述文件(PRF) 描述模型属性 30](#_Toc15583054)

[图 13应用工厂UML 34](#_Toc15583055)

[图 14应用工厂id 34](#_Toc15583056)

[图 15应用工厂id 36](#_Toc15583057)

[图 16设备管理器UML 39](#_Toc15583058)

[图 17设备管理器id 39](#_Toc15583059)

[图 18设备管理器name 41](#_Toc15583060)

[图 19组件UML 43](#_Toc15583061)

[图 20软件组件描述文件(SCD)端口信息 44](#_Toc15583062)

[图 21属性描述文件(PRF)模型属性信息 45](#_Toc15583063)

[图 22设备UML 48](#_Toc15583064)

[图 23 设备id 49](#_Toc15583065)

[图 24设备name 50](#_Toc15583066)

[图 25 adminState状态转换图 53](#_Toc15583067)

[图 26 adminState状态转换图 54](#_Toc15583068)

[图 27属性描述符(PRF)的kindtype 57](#_Toc15583069)

[图 28属性描述符文件(PRF) 模型属性信息 63](#_Toc15583070)

[图 29可加载设备UML 68](#_Toc15583071)

[图 30可执行设备UML 71](#_Toc15583072)

[图 31集合设备UML 75](#_Toc15583073)

[图 32端口数据传输机制序列图 79](#_Toc15583074)

[图 33输出端口UML 80](#_Toc15583075)

[图 34软件组件描述符(SCD)输出端口信息 81](#_Toc15583076)

[图 35软件装配描述符(SAD)组件连接信息 82](#_Toc15583077)

[图 36设备配置描述符(DCD)逻辑设备连接信息 82](#_Toc15583078)

[图 37输入端口UML 84](#_Toc15583079)

[图 38软件组件描述符(SCD)输入端口信息 85](#_Toc15583080)

[图 39软件装配描述符(SAD)中内部连接 88](#_Toc15583081)

[图 40软件装配描述符(SAD)中外部连接 88](#_Toc15583082)

[图 41 ORB抽象接口UML 91](#_Toc15583083)

[图 42 域描述文件关系 98](#_Toc15583084)

[图 43 软件包描述文件(SPD) 99](#_Toc15583085)

[图 44 接口信息 100](#_Toc15583086)

[图 45组件的SPD文件路径 100](#_Toc15583087)

[图 46波形应用包含的组件信息 101](#_Toc15583088)

[图 47 组件的连接信息 101](#_Toc15583089)

[图 48 属性描述（PRF）文件 102](#_Toc15583090)

[图 49 逻辑设备的SPD文件路径 102](#_Toc15583091)

[图 50设备管理器管理的设备信息 103](#_Toc15583092)

[图 51 域管理器配置（DMD）文件 103](#_Toc15583093)

[图 52 SampleApp波形应用 104](#_Toc15583094)

[图 53 Single\_Node节点 105](#_Toc15583095)

[图 54 SampleApp波形应用 106](#_Toc15583096)

[图 55 Single\_Node节点 117](#_Toc15583097)

1. 概述
   1. 什么是openSCA2.2.2？

openSCA2.2.2是介方基于SCA2.2.2(Software Communications Architecture 软件通信体系结构)规范开发的一套操作环境，实现了SCA2.2.2规范规定的大部分接口，包括加载波形、卸载波形、属性配置与查询、启动与停止波形等，提供了嵌入式、分布式通信系统下波形组件的部署、管理和通信能力。

openSCA2.2.2将波形应用与底层硬件接口解耦，使其具备强的可移植性。波形应用开发者无需关心底层实现，大大节省了波形开发与移植的成本。

基于openSCA2.2.2介方提供了一套波形应用开发套件sdrSDK和波形集成开发环境sdrIDE，开发者可借助于sdrSDK和sdrIDE工具进行波形应用的开发。

* 1. 如何获取openSCA2.2.2?

openSCA2.2.2开源框架代码可在Github进行获取，获取地址为：<https://github.com/JFounderSDR/openSCA.git>

sdrIDE波形应用开发套件可在Github进行获取，获取地址为：<https://github.com/JFounderSDR/sdrSDK.git>

* 1. 文档章节介绍

本文档主要包含以下几个部分：

1. **框架基础介绍**

第一章到第三章介绍了什么是openSCA2.2.2、框架获取路径、框架操作环境、框架工作流程。对于框架机制讲解以及开发时相关技术介绍。

1. **框架核心接口**

第四章到第八章介绍开发时常用接口，包括DomainManager、Application、ApplicationFactory、DeviceManager、组件、设备、端口、ORB抽象接口的介绍，通过代码片段讲解如何使用接口以及注意事项。

1. **框架配置文件**

第九章基于SCA2.2.2的域描述文件的讲解，包括SPD、SCD、SAD、PRF、DPD、DCD、DMD文件。

1. **波形及逻辑设备开发**

第十章构建开发场景，在介方开发的IDE生成的模板代码的基础上，开发波形实例和逻辑设备。

1. **术语表**

第十一章关于openSCA2.2.2框架涉及的专业术语的介绍。

1. 操作环境

操作环境(Operating Enviroment)由操作系统、中间件以及SCA核心框架提供的接口和操作组成,参见图 1。

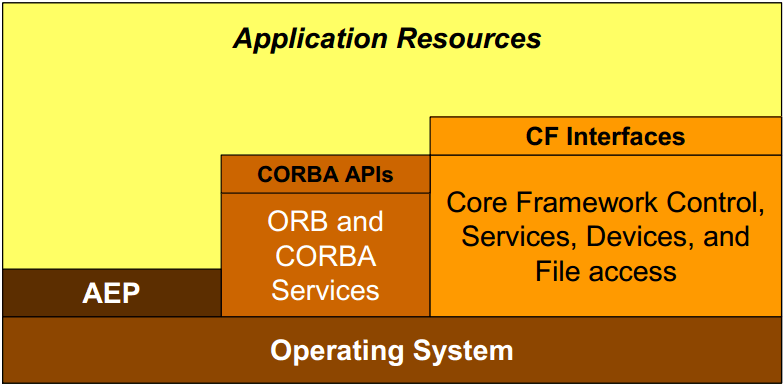


图 1 openSCA操作环境结构图

操作环境必须提供由SCA AEP(Application Enviroment Profile)所定义的功能和选项，定义AEP的目的是为了支持波形的可移植性、架构的可扩展性以及商业可行性。

* 1. 操作系统

支持Linux、VxWorks等操作系统。

* 1. 中间件和服务
     1. CORBA简介

CORBA中间件用来解决分布式异构系统间的通信问题。SCA2.2.2规范要求采用基于CORBA(Common Object Request Broker Architecture，公共对象请求代理体系)规范实现的中间件，至少需满足Minimum CORBA Specification的要求，openSCA2.2.2采用的是OCI公司的TAO(The ACE ORB)中间件。

* + - 1. CORBA通信机制

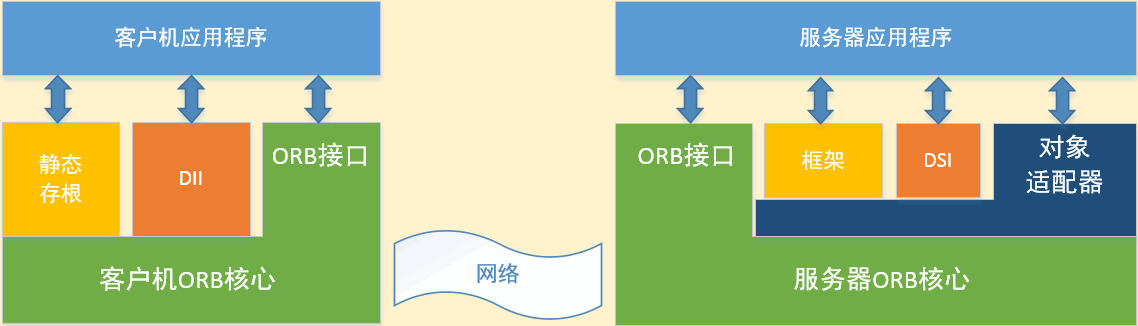


图 2 CORBA通信机制

CORBA通信机制如上图所示，客户程序提出请求，通过ORB传递到服务器应用程序，服务器应用程序再做出响应，其过程大致如下:

1. 客户机发出调用请求;
2. 客户机ORB核心通过网络将请求传送给与服务器应用程序相链接的服务器ORB核心；
3. 服务器ORB核心将请求分配给对象适配器(Object Adapter)，由它产生目标对象；
4. 对象适配器进一步将请求分配给实现目标对象的伺服程序；
5. 伺服程序执行请求后，将结果通过ORB返回给客户程序。
   * + 1. ORB

客户程序与服务程序通过ORB进行通信，ORB全称为Object Request Broker(对象请求代理)，它是CORBA能够实现分布式异构通信的核心模块，使得客户端可以透明地调用一个服务对象的方法，这个服务对象可以在本地，也可以在远程主机上。客户端通过ORB将请求发送给服务端，服务端通过ORB解析请求并进行响应，整个过程中，客户端并不知道服务程序在什么地方、用什么语言编写、运行在什么操作系统上。

* + - 1. 接口定义语言(IDL)

为调用一个分布式对象的操作，客户机必须了解这个对象所提供的接口，一个对象的接口由它所支持的操作和能够来回传输给这些操作的数据类型组成。在CORBA中，对象接口使用OMG接口定义语言(IDL)来定义，IDL不是编程语言，其目的是以与任何编程语言无关的的形式来定义对象接口，这是CORBA能够支持异构通信的关键，IDL可以映射成不同的编程语言。

* + - 1. 对象引用

客户程序通过对象引用调用目标对象的操作。对象引用是目标对象的代理，包含目标对象所在主机的IP地址、端口号、对象秘钥等信息，当客户程序调用对象引用的一个方法后，ORB会根据IP地址等信息将请求发送给正确的目标对象，目标对象响应后再通过ORB返回给客户程序。

* + 1. 命名服务

openSCA2.2.2提供了命名服务用于对象的绑定和获取。命名服务是最简单也是最基本的标准CORBA服务，提供名称到对象引用的映射：给定一个名称，该服务将返回和此名称绑定的对象引用。类似于因特网中的DNS，通过将域名转化为IP地址，访问到目标主机。

在一个分布式系统中，无论目标对象在哪个主机上运行，只要知道目标对象在命名服务上绑定的名称，调用程序就可以获取到目标对象的对象引用，进而调用目标对象的接口。

openSCA2.2.2封装了一些用户常用的命名服务的接口，详见第八节ORB抽象接口。

* + 1. 事件服务

通过对象引用调用目标对象的接口属于同步请求调用，同时CORBA也提供了事件服务用于实现异步通信。事件服务为对象之间的解耦通信提供了支持，允许发送者在单次调用的情况下向一个或者多个接收者发送事件，事件服务可以创建事件通道，发送者和接收者都连接在一个事件通道上，事件通道将事件从发送者传递到接收者，发送者与接收者无需彼此的情况。

OE提供两个标准事件通道：

Incoming Domain Management(IDM\_Channel)和Outgoing Domain Management(ODM\_Channel)。域中的组件使用IDM\_Channel事件通道将事件发送给域管理类接口(如DomainManagement、Application等)，域管理类接口使用ODM\_Channel将事件发送给外部程序。OE也支持波形开发人员创建额外的事件通道。

* 1. SCA核心框架

核心框架(Core Framework)是SCA2.2.2规范的核心内容，定义了一系列标准接口，以保证波形应用的可移植性。

DomainManager组件管理波形应用、应用工厂、硬件设备以及系统内的设备管理器。

Device、LoadableDevice、ExecutableDevice接口用于直接控制系统内的硬件设备，实现这些接口的系统组件称作逻辑设备。

还有一些组件不直接控制硬件设备，但实现了Resource接口，可作为应用组件使用。这些组件有一些列的配置和销毁操作，每个组件均可和其它组件通信。波形应用就是一个或多个提供指定服务或功能的组件的集合，通过Application接口来管理。应用工厂会根据多个因素(如硬件设备当前的可利用性、应用组件的加载需求等)将应用组件部署到一系列硬件设备上。

文件服务接口(FileManager、FileSystem以及File)用于管理域中的文件，具备读写、创建、删除、复制文件等功能。

本文档只介绍用户开发过程中常用的核心框架接口，下文会详细描述。

1. openSCA框架核心流程
   1. 框架启动流程介绍

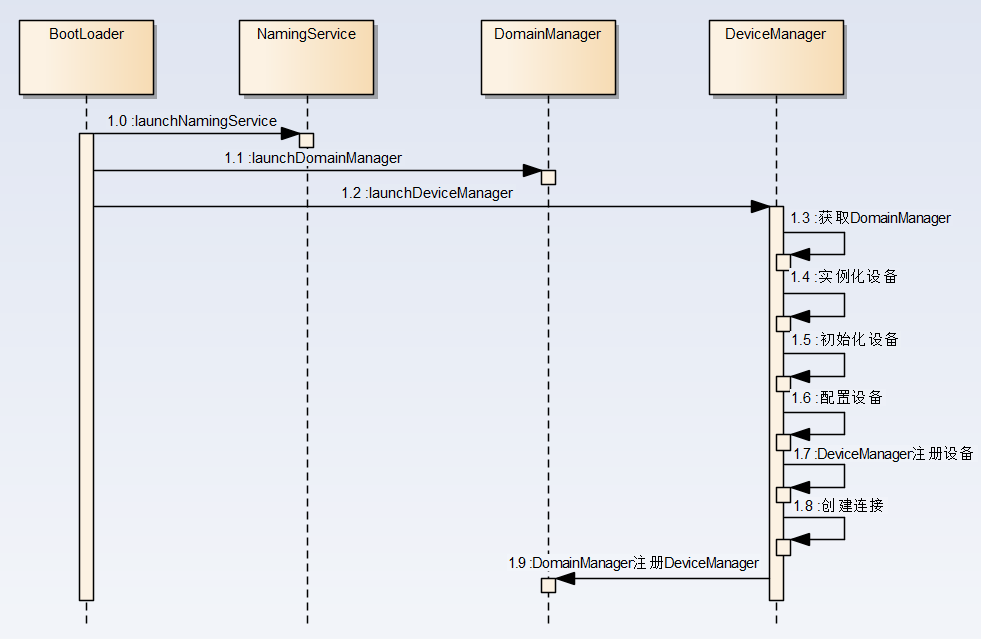


图 3框架启动流程序列图

运行BootLoader.out文件启动框架，将命名服务、域管理器、设备管理器启动，在设备管理器中实例化设备，创建连接等，具体流程如下：

1. 运行BootLoader.out启动命名服务、域管理器和设备管理器
2. 设备管理器获取域管理器，为了向域管理器注册设备管理器
3. 设备管理器实例化、初始化、配置每个设备
4. 向设备管理器注册设备
5. 建立设备间的连接
6. 在域管理器中注册设备管理器
   1. 波形安装流程介绍

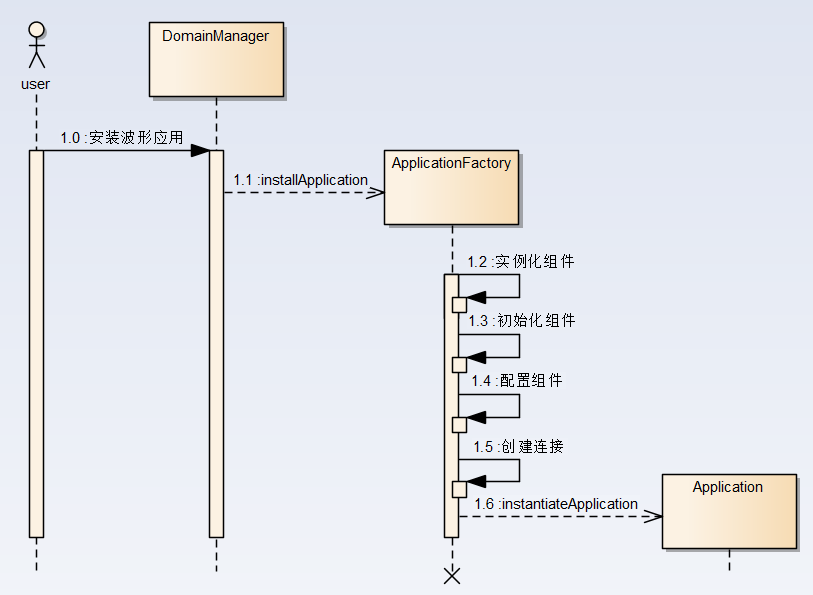


图 4波形安装流程序列图

调用域管理器的installApplication()接口，创建应用工厂，应用工厂实例化、初始化、配置波形应用的每个组件，创建组件间连接，最后实例化波形应用，具体流程如下：

1. 域管理器installApplication()接口创建应用工厂
2. 在应用工厂中实例化、初始化、配置波形应用的每个组件
3. 创建组件间的连接
4. 实例化波形应用
   1. 波形卸载流程

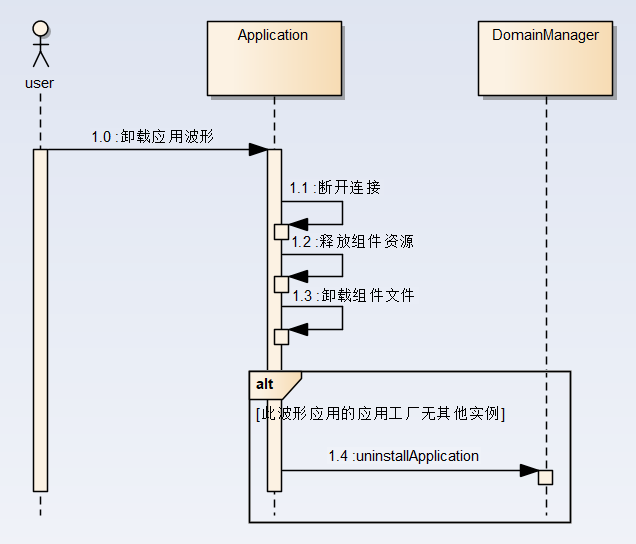


图 5波形卸载流程序列图

卸载波形应用时，需要先调用Application的releaseObject()接口，断开组件间连接、释放组件资源、卸载组件文件，如果此波形应用的应用工厂无其他波形实例，可以调用域管理器的uninstallApplication()接口，卸载应用工厂，具体流程如下：

1. 调用Application的releaseObject()接口；
2. 该接口中进行的操作为断开组件间连接、释放组件资源、卸载组件文件；
3. 如果此波形的应用工厂无其他应用实例，调用域管理器的uninstallApplication()接口，卸载应用工厂。
4. openSCA框架控制接口

框架控制接口为波形设计者提供底层软件和硬件层的抽象，为波形组件的开发提供基本的接口和服务，管理域内的软硬件资源。

* 1. 异常

框架接口抛出的异常，包含CORBA异常和核心框架CF异常。

* + 1. CORBA::Exception

CORBA异常，CORBA调用时可能抛出的异常，可以通过printCORBAException()接口打印异常信息。

try {

...

} catch (CORBA::Exception & e) {

printCORBAException(e);

} catch (...) {

...

}

* + 1. CORBA::SystemException

CORBA系统异常，CORBA调用时可能抛出的异常，可以通过printCORBASystemException()接口打印异常信息。

try {

...

} catch (CORBA::SystemException & e) {

printCORBASystemException(e);

} catch (...) {

...

}

* + 1. CF::InvalidFileName

CF::InvalidFileName异常表示将无效的文件名传递给文件服务操作。错误号应指示CF::ErrorNumberType值。该消息提供了描述文件名无效的原因的信息。

* + 1. CF::InvalidProfile

CF InvalidProfile异常表示无效的配置文件错误。

* + 1. CF::DomainManager::InvalidIdentifier

InvalidIdentifier异常表示应用程序标识符无效。

* + 1. CF::DomainManager::ApplicationInstallationError

应用程序安装未正确完成时引发ApplicationInstallation-Error异常类型。错误号应指示CF::ErrorNumberType值。该消息取决于组件，提供描述错误原因的附加信息。

* + 1. CF::DomainManager::ApplicationUninstallationError

如果未正确完成应用程序的卸载，则会引发ApplicationU-ninstallationError异常类型。 错误号应指示CF ErrorNumberType值。该消息取决于组件，提供描述错误原因的附加信息。

* + 1. CF::PortSupplier::UnknownPort

如果请求未定义的端口，引发CF::PortSupplier::UnknownPort异常。

* + 1. CF::LifeCycle::InitializeError

InitializeError异常表示组件初始化期间发生错误。该消息取决于组件，提供描述错误发生原因的附加信息。

* + 1. CF::LifeCycle::ReleaseError

ReleaseError异常表示组件releaseObject操作期间发生错误。 该消息取决于组件，提供描述错误发生原因的附加信息。

* + 1. CF::PropertySet::PartialConfiguration

PartialConfiguration异常表示Component的配置部分成功。 返回的invalidProperties指示无效的属性。

* + 1. CF::PropertySet::InvalidConfiguration

InvalidConfiguration异常表示组件的配置失败（根本没有配置）。 该消息取决于组件，提供描述错误发生原因的附加信息。返回的invalidProperties指示无效的属性。

* + 1. CF::UnknownProperties

CF UnknownProperties异常表示组件未知的一组属性

* + 1. CF::Resource::StartError

StartError异常表示尝试启动资源时发生错误。errorNumber参数应指示CF::ErrorNumberType值。该消息取决于组件，提供描述错误原因的附加信息。

* + 1. CF::ApplicationFactory::CreateApplicationRequestError

当参数CF DeviceAssignmentSequence包含一个或更多无效的应用程序组件到设备分配时，将引发CreateApplicationRequestError异常。

* + 1. CF::ApplicationFactory::CreateApplicationError

当创建请求有效但由于内部处理错误导致应用程序未成功实例化时，将引发CreateApplicationError异常。错误号应指示CF::ErrorNumberType值。该消息取决于组件，提供描述错误原因的附加信息。

* + 1. CF::ApplicationFactory::InvalidInitConfiguration

当输入initConfiguration参数无效时，将引发InvalidInitConfiguration异常。

* 1. DomainManager

DomainManager接口用于控制和配置系统域，可以逻辑地分为三类:人机界面(HCI)、注册和CF管理。

HCI操作用于配置域、获取域功能(设备和应用程序)和初始化维护功能。

注册操作用于注册/注销设备管理器、设备管理器的设备，以及在启动时或运行时用于动态设备和应用程序提取和插入的应用程序。

管理操作用于访问已注册的设备管理器和域管理器的文件管理器的接口。

还可以获取当前在域管理器中已注册或安装的设备管理器、波形应用、应用工厂和文件管理器等组件的信息。

框架会执行注册操作，开发者不用调用。下面主要介绍开发者开发时常用的接口，identifier()、applications()、applicationFactories()、deviceManagers()、installApplication()、uninstallApplication()。

* + 1. UML

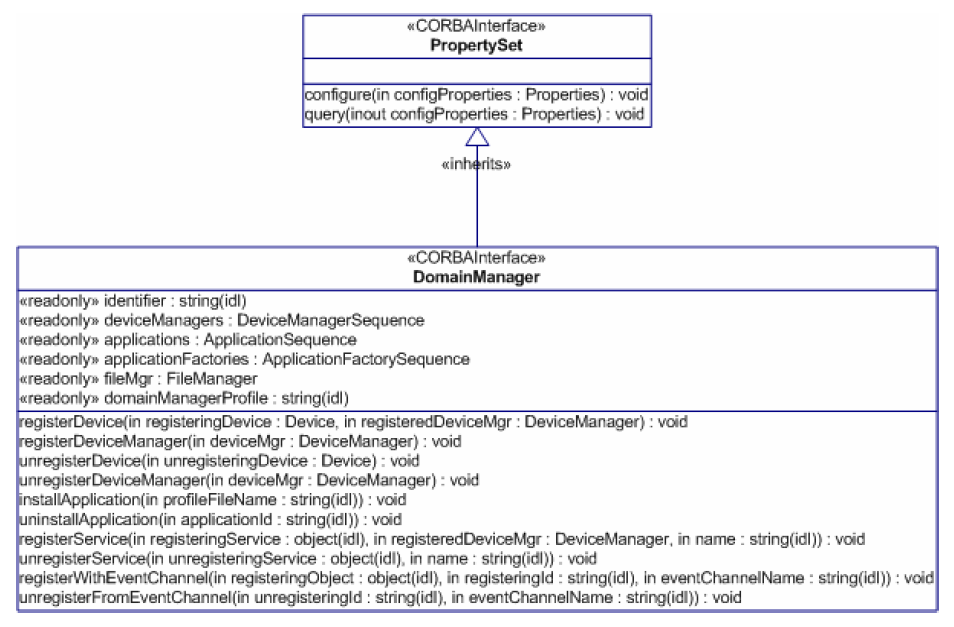


图 6域管理器的UML

* + 1. identifier

identifier()操作返回DomainManager实例的唯一标识符。标识符应与域管理器描述符(DMD)文件的domainmanagerconfiguration元素id属性值相同，参见图 7。

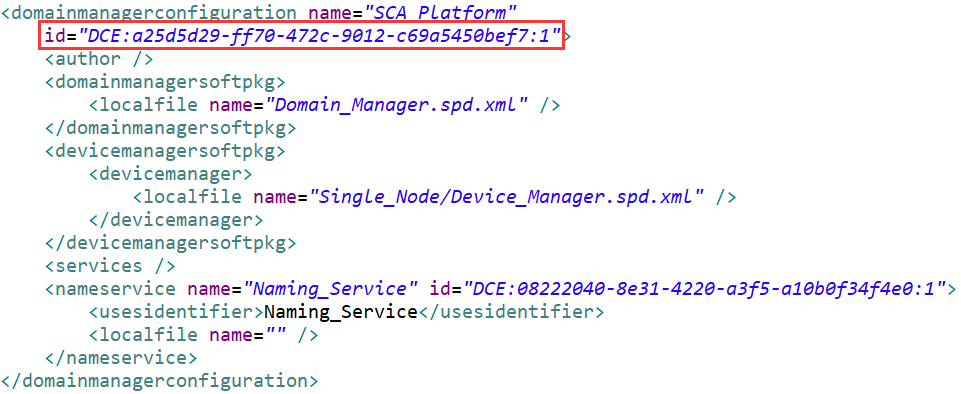


图 7域管理器id

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **identifier**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回域管理器实例唯一标识符 |

* + - 1. 代码片段

CF::DomainManager\_var dmnMgr;

openscaSupport::ORB\_Wrap \* orbWrap = ...;

CORBA::Object\_var dmnObj = CORBA::Object::\_nil();

try {

//通过调用orb的get\_object\_from\_string函数

//通过字符串id获取域管理器对象

dmnObj = orbWrap->get\_object\_from\_string(

"OpenSCA\_Domain/Domain\_Manager");

} catch (...) {

...

}

try {

//通过\_narrow函数将CORBA::Object\_var对象紧缩为域管理器对象

dmnMgr = CF::DomainManager::\_narrow(dmnObj);

} catch (...) {

...

}

//调用域管理器对象的identifier()，获得域管理器的id标识

std::string dmnMgrId = dmnMgr->identifier();

* + 1. applications

applications()操作返回域中已安装的所有波形应用。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::DomainManager::ApplicationSequence \*  **applications**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回域中已安装的所有波形应用 |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

CF::DomainManager\_var dmnMgr = ...;

CF::DomainManager::ApplicationSequence\_var apps;

try {

//调用applications函数获得域内实例化波形应用序列

apps = dmnMgr->applications();

//判断是否获取序列成功

if(CORBA::is\_nil(apps)){

//apps is null.

}

} catch (...) {

...

}

* + 1. applicationFactories

applicationFactories()操作返回域中所有已创建的应用工厂，其中每个成功安装的波形应用（SAD）文件对应一个应用工厂。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::DomainManager::ApplicationFactorySequence \*  applicationFactories()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回域中已创建的应用工厂列表 |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

CF::DomainManager\_var dmnMgr = ...;

CF::DomainManager::ApplicationFactorySequence\_var appFactorys;

try {

//调用applicationFactories函数获得域内的应用工厂序列

appFactorys = dmnMgr->applicationFactories();

//判断是否获取序列成功

if(CORBA::is\_nil(appFactorys)){

//appFactorys is null.

}

} catch (...) {

...

}

* + 1. deviceManagers

deviceManagers()操作返回域中已注册的设备管理器。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::DomainManager::DeviceManagerSequence \*  deviceManagers()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回域中已注册的设备管理器序列 |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

CF::DomainManager\_var dmnMgr = ...;

CF::DomainManager::DeviceManagerSequence\_var devMgrs;

try {

//调用deviceManagers函数获得域中已注册的设备管理器序列

devMgrs = dmnMgr->deviceManagers();

//判断是否获取序列成功

if(CORBA::is\_nil(devMgrs)){

//devMgrs is null.

}

} catch (...) {

...

}

* + 1. installApplication

installApplication()接口用于创建applicationFactory对象，再通过得到的应用工厂来创建波形应用。

波形应用需要由应用工厂创建，在安装应用前先判断是否已创建了对应的应用工厂；如果已创建，则通过已创建的应用工厂创建波形应用，否则先创建应用工厂。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  installApplication(  **const** **char** \* profileFileName)  **throw** (  CF::DomainManager::ApplicationInstallationError,  CF::InvalidFileName,  CF::InvalidProfile,  CORBA::SystemException); | |
| 参数 | (in) **const** **char** \* profileFileName | 波形应用sad.xml文件的绝对路径 |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

CF::DomainManager\_var dmnMgr = ...;

//通过域管理器获得应用工厂，可参考5.2.4.2代码片段

CF::DomainManager::ApplicationFactorySequence\_var appFactorys = ...;

std::string profileFileName = ...;

//判断应用工厂序列中是否存在该应用的应用工厂，如果存在，则通过应用工厂创建应用，

//如果不存在，先创建应用工厂

bool existAppFact = false;

CORBA::Short length = appFactorys->length();

for(CORBA::Short i = 0; i < length; ++i){

CORBA::String\_var factName = CORBA::string\_dup((\*appFactorys)[i]->name());

if(0 == strcmp(factName.in(), appName.c\_str())){

existAppFact = true;

break;

}

}

try {

if (!existAppFact) {

// profileFileName为安装波形sad.xml的绝对路径

//如：SCA\_Platform/Applications/AudioTransApp/AudioTransApp.sad.xml

dmnMgr->installApplication(profileFileName.c\_str());

}

} catch (...) {

...

}

* + 1. uninstallApplication

uninstallApplication()接口用于销毁应用工厂。

一个应用工厂可以创建多个波形应用实例，且每个波形的名称与应用工厂的名称相同。

在调用此接口前，需要先释放由该应用工厂创建的所有波形应用。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void** **uninstallApplication**(  **const** **char** \* applicationId)  **throw** (  CF::DomainManager::ApplicationUninstallationError,  CF::DomainManager::InvalidIdentifier,  CORBA::SystemException); | |
| 参数 | (in) **const** **char** \* applicationId | 波形应用实例id |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

CF::DomainManager\_var dmnMgr = ...;

//需要卸载的波形实例id

std::string applicationId = ...;

//通过域管理器获得已安装的波形应用序列对象，可参考5.2.3.2代码片段

CF::DomainManager::ApplicationSequence\_var apps = ...;

int length = apps.length();

std::string appName = "";

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

if(0 == strcmp(apps[i]->identifier(), applicationId)){

//获取将要卸载波形实例的name

appName = apps[i]->name();

try {

//释放波形应用实例资源

apps[i]-> releaseObject();

} catch (...) {

...

}

}

}

//记录已安装波形应用序列中与appName名字相同的个数，也就是该波形的应用工厂创建的实//例个数。释放资源后，已安装应用序列相应的除去该实例。

length = apps.length();

int count = 1;

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

//获取将要卸载波形实例的name，此name与applicationFactory的name相同

if(0 == strcmp(appName.c\_str(), apps[i]->name())){

++count;

}

}

//无其他实例时，则可销毁应用工厂，调用uninstallApplication

if(count = 1){

try {

dmnMgr->uninstallApplication(applicationId.c\_str());

} catch (...) {

...

}

}

* 1. Application

波形应用(Application)是由ApplicationFactory的create()接口创建的。Application提供了一个控制、配置、管理波形应用实例的统一接口。

该接口继承了资源(Resource)的所有接口,如：identifier()、name()、initialize()、releaseObject()、start()、stop()、configure()、query()、getPort()、runTest()接口。可以通过这些接口来初始化、销毁、启动、停止波形应用；配置、查询波形应用的模型属性；获取波形应用的端口。

每个波形应用都有一个称为波形控制器的组件,其作用就是实施对各个波形组件的控制,而Application接口对波形应用的控制,实际上是通过波形控制器来完成的。Application接口将start、stop、initialize、releaseObject、configure、query等操作通过波形控制器来控制、配置每个波形组件。

下面主要介绍开发者开发时常用的接口，identifier()、name()、getPort()、initialize()、releaseObject()、configure()、query()、start()、stop()。

* + 1. UML

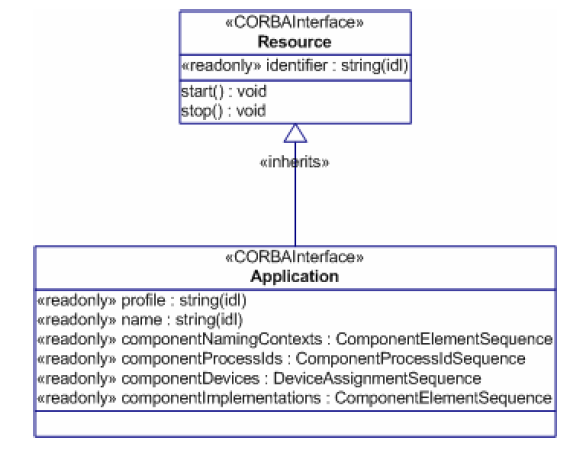


图 8波形应用UML

* + 1. identifier

identifier()接口应返回Application实例的唯一标识符。标识符应与波形装配描述符(SAD)文件的softwareassembly元素id属性值相同，参见图 9。

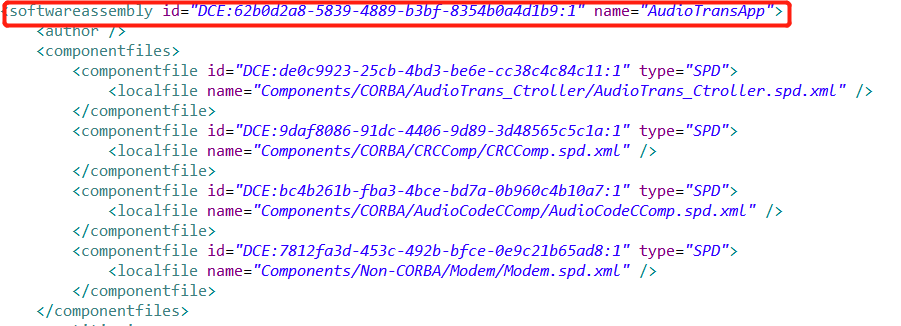


图 9波形应用id

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **identifier**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回应用实例的唯一标识符。 |

* + - 1. 代码片段

std::vector<std::string> ids;

//通过域管理器获取域内波形应用序列，可参考5.2.3.2代码片段

CF::DomainManager::ApplicationSequence \* appSeq = ...;

const char \* appName = ...;

for (CORBA::ULong i = 0; i < appSeq->length(); i++) {

std::string id;

//和获取到的序列内的应用名称比对

if ( 0 == strcmp(appName, (\*appSeq)[i]->name())) {

//获取应用的id

id = (\*appSeq)[i]->identifier();

ids.push\_back(id);

}

}

* + 1. name

name()接口返回创建波形应用的名称。应用名称获取自应用的波形装配描述(SAD)文件的softwareassembly元素的name属性，参见图 10。

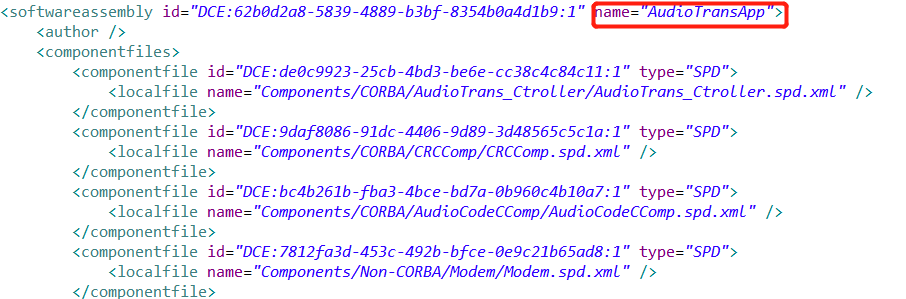


图 10波形应用name

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **name**(); |
| 返回值 | 返回所创建的波形应用的名称。 |

* + - 1. 代码片段

std::vector<std::string> ids;

//通过域管理器获取域内波形应用序列，可参考5.2.3.2代码片段

CF::DomainManager::ApplicationSequence \* appSeq = ...;

std::string name = “”;

for (CORBA::ULong i = 0; i < appSeq->length(); i++) {

name = (\*appSeq)[i]->name() ;

}

* + 1. getPort

getPort()接口根据传入的端口名称获得对应端口的对象引用。端口名称应和波形装配描述符(SAD)中externalports元素下的端口名称匹配，参见图 11。

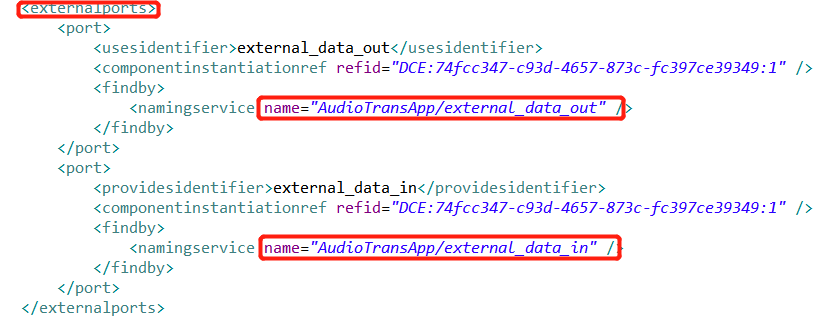


图 11波形应用端口信息

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CORBA::Object\_ptr  **getPort**(  **const** **char**\* name)  **throw** (  CORBA::SystemException,  CF::PortSupplier::UnknownPort); | |
| 参数 | (in) const char \* name | 需获取对象引用的端口名称 |
| 返回值 | 返回参数name对应的对象引用 | |

* + - 1. 代码片段

std::vector<std::string> ids;

//通过域管理器获取域内波形应用序列，可参考5.2.3.2代码片段

CF::DomainManager::ApplicationSequence \* appSeq = ...;

//端口名称

const std::string portName = ...;

CORBA::Object\_var portObj = CORBA::Object::\_nil();

for (CORBA::ULong i = 0; i < appSeq->length(); i++) {

//获取应用的符合条件的端口的对象引用

try {

portObj = (\*appSeq)[i]->getPort(portName.c\_str());

} catch (...) {

...

}

if (!CORBA::is\_nil(portObj)) {

providesport = provideObj;

} else {

//narrow port failed.

}}

* + 1. initialize

initialize()接口在应用启动之前进行一些初始化的设置，如变量赋初值，分配内存等。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **initialize**()  **throw** (  CF::LifeCycle::InitializeError,  CORBA::SystemException); |

* + - 1. 代码片段

//通过域管理器获取域内波形应用序列，可参考5.2.3.2代码片段

CF::DomainManager::ApplicationSequence \* appSeq = …;

for (CORBA::ULong i = 0; i < appSeq->length(); i++) {

//对序列中的应用进行初始化

(\*appSeq)[i]->initialize();

}

* + 1. releaseObject

releaseObject()接口终止应用程序的执行，并进行包括断开连接端口、释放资源组件、终止组建进程、更改关联设备的状态、从命名服务解除波形应用绑定等一系列操作。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **releaseObject**()  **throw** (  CF::LifeCycle::ReleaseError,  CORBA::SystemException); |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

CF::DomainManager\_var dmnMgr = ...;

//需要卸载的波形实例id

std::string applicationId = ...;

//通过域管理器获得已安装的波形应用序列对象，可参考5.2.3.2代码片段

CF::DomainManager::ApplicationSequence\_var apps = ...;

int length = apps.length();

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

if(0 == strcmp(apps[i]->identifier(), applicationId)){

try {

//释放波形应用实例资源

apps[i]->releaseObject();

} catch (...) {

...

}

}

}

* + 1. configure

configure()接口通过重写属性的id/value对实现对属性描述符(PRF)文件中的应用属性配置和修改。被配置的属性权限应为读写或只写。软件包描述符(SPD)文件包含属性描述符(PRF)文件路径参见图 12

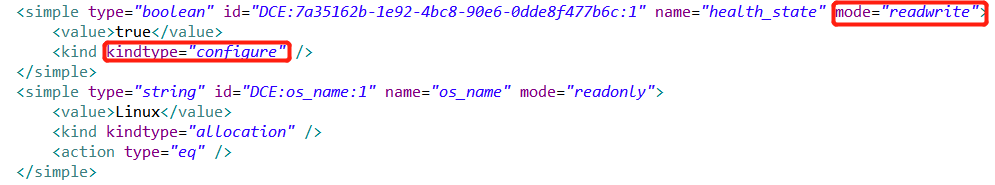


图 12属性描述文件(PRF) 描述模型属性

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **configure**(  **const** CF::Properties & configProperties)  **throw** (  CF::PropertySet::PartialConfiguration,  CF::PropertySet::InvalidConfiguration,  CORBA::SystemException); | |
| 参数 | (in) **const** CF::Properties &cofigProperties | 需要配置的属性序列 |

* + - 1. 代码片段

//通过域管理器获取域内波形应用序列，可参考5.2.3.2代码片段

CF::DomainManager::ApplicationSequence \* appSeq = ...;

//对被属性id对应的value进行赋值

CF::Properties props;

props.length(1);

props[0].id = "DCE:cfb87b0a-35b3-4fa5-9097-607d6f0ec61e:1";

//将属性值重新设置为0.0

CORBA::Float frequency = 0.0;

props[0].value >>= frequency;

//被修改属性的应用名字

const char\* appName = ...;

for(CORBA::ULong i = 0; i < appSeq->length(); i++) {

// 从应用序列中筛选目标应用

if(strcmp(appName, (\*appSeq)[i]->name()) == 0){

try {

//对属性进行重新配置

(\*appSeq)[i]->configure(props);

} catch (...) {

...

}

}

}

* + 1. query

query()接口可以根据属性描述符(PRF)文件中的属性id查询应用对应的属性值。

当参数configProperties序列长度为0时，查询操作将返回所有组件的属性；当参数configProperties序列长度不为0时，查询操作将返回configProperties参数中指定的id/value对。

查询操作的有效属性应该是kindtype为configure，且mode为readonly或readwrite的属性，参见图 12。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **query**(  CF::Properties & configProperties)  **throw** (  CF::UnknownProperties,  CORBA::SystemException); | |
| 参数 | (inout) CF::Properties & cofigProperties | 需要查询的属性 |

* + - 1. 代码片段

//通过域管理器获取域内波形应用序列，可参考5.4.3.2代码片段

CF::DomainManager::ApplicationSequence \* appSeq = ...;

//被查询属性的应用名字

const char\* appName = ...;

for(CORBA::ULong i = 0; i < appSeq->length(); i++) {

// 从应用序列中筛选目标应用

if(strcmp(appName, (\*appSeq)[i]->name()) == 0){

try {

//对属性进行查询

//1、属性参数长度不为0,根据id查询属性值

CF::Properties props;

props.length(1);

props[0].id = "DCE:cfb87b0a-35b3-4fa5-9097-607d6f0ec61e:1";

(\*appSeq)[i]->query(props);

//2、属性参数长度为0，查询所有属性

CF::Properties props;

(\*appSeq)[i]->query(props);

} catch (...) {

...

}

}

}

* + 1. start

start()接口通过调用波形控制器组件的start()接口启动波形控制器组件，波形控制器组件启动其他组件，从而启动应用。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **start**()  **throw** (  CF::Resource::StartError,  CORBA::SystemException); |

* + - 1. 代码片段

//通过域管理器获取域内波形应用序列，可参考5.4.3.2代码片段

CF::DomainManager::ApplicationSequence \* appSeq = ...;

//创建应用，可参考5.3.4.2

//目标应用名字

const char \* appName = ...;

for(CORBA::ULong i = 0; i < appSeq->length(); i++) {

// 从应用序列中筛选目标应用

if(strcmp(appName, (\*appSeq)[i]->name()) == 0) {

try {

//启动应用

(\*appSeq)[i]->start();

} catch (...) {

...

}

}

}

* + 1. stop

stop()接口通过调用波形控制器组件的stop()接口停止波形控制器组件，波形控制器组件停止其他组件的方式结束应用。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **stop**()  **throw** (  CF::Resource::StartError,  CORBA::SystemException); |

* + - 1. 代码片段

//通过域管理器获取域内波形应用序列，可参考5.4.3.2代码片段

CF::DomainManager::ApplicationSequence \* appSeq = ...;

//目标应用名字

const char \* appName = ...;

for(CORBA::ULong i = 0; i < appSeq->length(); i++) {

// 从应用序列中筛选目标应用

if(strcmp(appName, (\*appSeq)[i]->name()) == 0){

try {

//停止应用

(\*appSeq)[i]->stop();

} catch (...) {

...

}

}

}

* 1. ApplicationFactory

ApplicationFactory接口类提供了创建指定类型波形应用的操作。ApplicationFactory接口类是使用工厂设计模式设计的。ApplicationFactory仅负责创建波形应用,而没有释放波形应用的接口函数。波形的释放是通过应用(Application)的releaseObject()接口完成的。

* + 1. UML

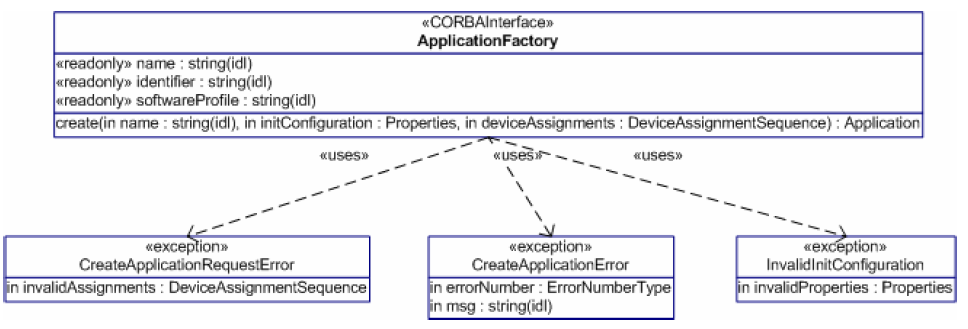


图 13应用工厂UML

* + 1. identifier

identifier()接口返回ApplicationFactory实例的唯一标识符。标识符应与应用程序工厂创建的第一个波形应用的波形装配描述(SAD)文件的softwareassembly元素id属性相同参见图 14。



图 14应用工厂id

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **identifier**(); |
| 返回值 | 返回应用工厂实例的唯一标识符。 |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

CF::DomainManager\_var dmnMgr = ...;

CF::DomainManager::ApplicationFactorySequence\_var appFacts;

try {

//调用applicationFactories函数获得域内的应用工厂序列

appFactorys = dmnMgr->applicationFactories();

//判断是否获取序列成功

if(CORBA::is\_nil(appFacts)){

//appFacts is null.

}

//获取应用工厂序列的长度

CORBA::UShort appFactSize = appFacts->length();

CORBA::Short index = -1;

for (CORBA::UShort i = 0; i < appFactSize; ++i){

//筛选目标应用工厂

if (0 == strcmp((\*appFacts)[i]->name(), "MsgTransApp")){

index = i;

break;

}

//获得该应用工厂的id

std::string appFactId = (\*appFacts)[index]->identifiter();

}

} catch (...) {

...

}

* + 1. name

name()接口返回ApplicationFactory实例的名称。标识符应与应用程序工厂创建的第一个波形应用的波形装配描述(SAD)文件的softwareassembly元素name属性相同参见图 15。



图 15应用工厂id

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **name**(); |
| 返回值 | 返回应用工厂实例的名称。 |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

//安装应用工厂，可参考5.2.6.2代码片段

CF::DomainManager\_var dmnMgr = ...;

CF::DomainManager::ApplicationFactorySequence\_var appFacts;

try {

//调用applicationFactories函数获得域内的应用工厂序列

appFactorys = dmnMgr->applicationFactories();

//判断是否获取序列成功

if(CORBA::is\_nil(appFacts)){

//appFacts is null.

}

//获取应用工厂序列的长度

CORBA::UShort appFactSize = appFacts->length();

CORBA::Short index = -1;

for (CORBA::UShort i = 0; i < appFactSize; ++i){

//根据应用工厂名称筛选目标应用工厂

if (0 == strcmp((\*appFacts)[i]->name(), "MsgTransApp")){

index = i;

break;

}

}

} catch (...) {

...

}

* + 1. create

create()接口的作用是把组成一个波形应用的各个组件部署到由软件包描述(SPD)文件定义的可执行设备上；并根据波形装配文件(SAD)文件定义,建立组件间的端口的连接关系，初始化并且运行这些组件。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CF::Application\_ptr  **create**(  **const** **char** \* name,  **const** CF::Properties & initConfiguration,  **const** CF::DeviceAssignmentSequence & deviceAssignments)  **throw** (  CF::ApplicationFactory::InvalidInitConfiguration,  CF::ApplicationFactory::CreateApplicationRequestError,  CF::ApplicationFactory::CreateApplicationError,  CORBA::SystemException); | |
| 参数 | (in) **const** **char** \* name | 创建应用的名称 |
| (in) **const** CF::Properties & initConfiguration | 初始化的模型属性 |
| (in) **const** CF::DeviceAssignmentSequence & deviceAssignments | 组件详细信息类序列 |
| 返回值 | 返回创建应用的对象引用 | |

* + - 1. 代码片段

//通过orb获得域管理器对象，可参考5.2.2.2代码片段

//安装应用工厂，可参考5.2.6.2代码片段

CF::DomainManager\_var dmnMgr = ...;

CF::DomainManager::ApplicationFactorySequence\_var appFacts;

try {

//调用applicationFactories函数获得域内的应用工厂序列

appFactorys = dmnMgr->applicationFactories();

//判断是否获取序列成功

if(CORBA::is\_nil(appFacts)){

//appFacts is null.

}

//获取应用工厂序列的长度

CORBA::UShort appFactSize = appFacts->length();

CORBA::Short index = -1;

for (CORBA::UShort i = 0; i < appFactSize; ++i){

//根据应用工厂名称筛选目标应用工厂

if (0 == strcmp((\*appFacts)[i]->name(), "MsgTransApp")){

index = i;

break;

}

}

CF::DeviceAssignmentSequence devAssig = ...;

CF::Properties props = ...;

CF::Application\_var app = CF::Application::\_nil();

//应用工厂创建应用

app = (\*appFacts)[index]->create("MsgTransApp", props, devAssig);

} catch (...) {

...

}

* 1. DeviceManager

设备管理器负责管理逻辑设备，通过域管理器获取，且在设备管理器启动后，要向域管理器注册。

设备管理器可以启动域中的设备，启动成功后，可向设备管理器注册,完成设备的初始化和配置，在域管理器中通过registerDeviceManager()接口注册设备管理器后，再调用设备管理器的registeredDevices()接口向域管理器注册设备，从而达到域管理器控制整个域资源的功能。

下面主要介绍开发者开发时常用的接口，identifier()、label()、registeredDevices()。

* + 1. UML

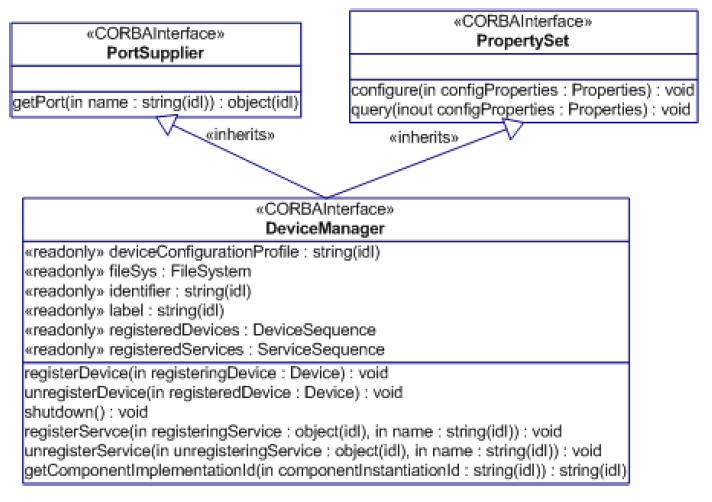


图 16设备管理器UML

* + 1. identifier

identifier()操作返回DeviceManager实例的唯一标识符。标识符应与设备配置描述符(DCD)文件的deviceconfiguration元素id属性值相同，参见图 17。

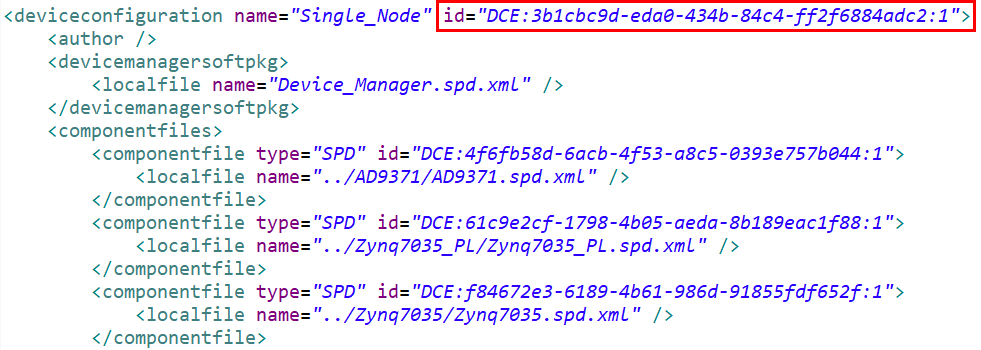


图 17设备管理器id

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **identifier**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回设备管理器实例唯一标识符 |

* + - 1. 代码片段

//需要获取的设备管理器的唯一标识符id

#define DEVICE\_MANAGER\_ID ("DCE:3b1cbc9d-eda0-434b-84c4-ff2f6884adc2:1")

CF::DomainManager\_var dmnMgr = ...;

//通过域管理器获得已注册的设备管理器序列，可参考5.2.5.2代码片段

CF::DomainManager::DeviceManagerSequence\_var devMgrs = ...;

CF::DeviceManager\_var devMgr;

CORBA::Short length = devMgrs->length();

for(CORBA::Short i=0; i<length; ++i){

//根据dcd文件中的id通过设备管理器序列，获取设备管理器

if(0 == strcmp((\*devMgrs)[i]->identifier(), DEVICE\_MANAGER\_ID)){

devMgr = CF::DeviceManager::\_duplicate((\*devMgrs)[i].in());

}

}

if(CORBA::is\_nil(devMgr)){

//devMgr is null

return;

}

...

//调用设备管理器对象的identifier()，获得设备管理器的id标识

std::string devMgrId = devMgr->identifier();

* + 1. label

label()操作返回DeviceManager实例的名称。名称应与软件包描述符(SPD)文件的softpkg元素name属性值相同，参见图 18。

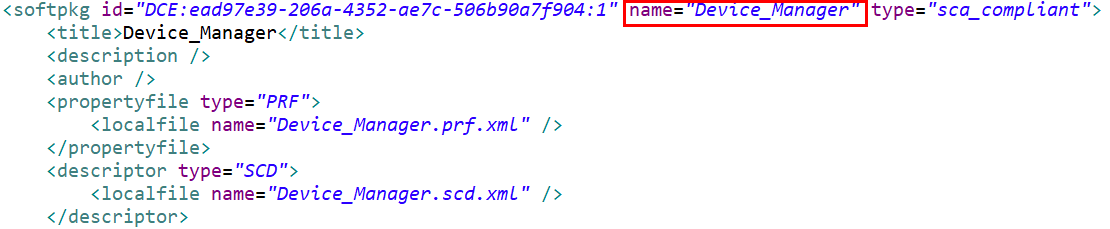


图 18设备管理器name

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **label**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回设备管理器实例的唯一名称 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，可参考

//5.5.2.2代码片段

CF::DeviceManager\_var devMgr = ...;

//调用设备管理器对象的identifier()，获得设备管理器的id标识

std::string devMgrName = devMgr->label();

* + 1. registeredDevices

registeredDevices()接口用于获取在域中已注册的设备。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::DeviceSequence \*  registeredDevices()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回域中已注册的设备序列 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，可参考

//5.5.2.2代码片段

CF::DeviceManager\_var devMgr = ...;

CF::DeviceSequence\_var devs;

try {

//调用设备管理器对象的registeredDevices()，获得域中已安装的设备

devs = devMgr->registeredDevices();

//判断是否获取序列成功

if(CORBA::is\_nil(devs)) {

//devs is null.

}

} catch (...) {

...

}

1. 应用组件接口

符合SCA规范基础应用接口的软件模块。组件构成波形应用。以下为组件接口介绍，组件接口通过应用接口调用，开发时不直接调用。

* 1. UML

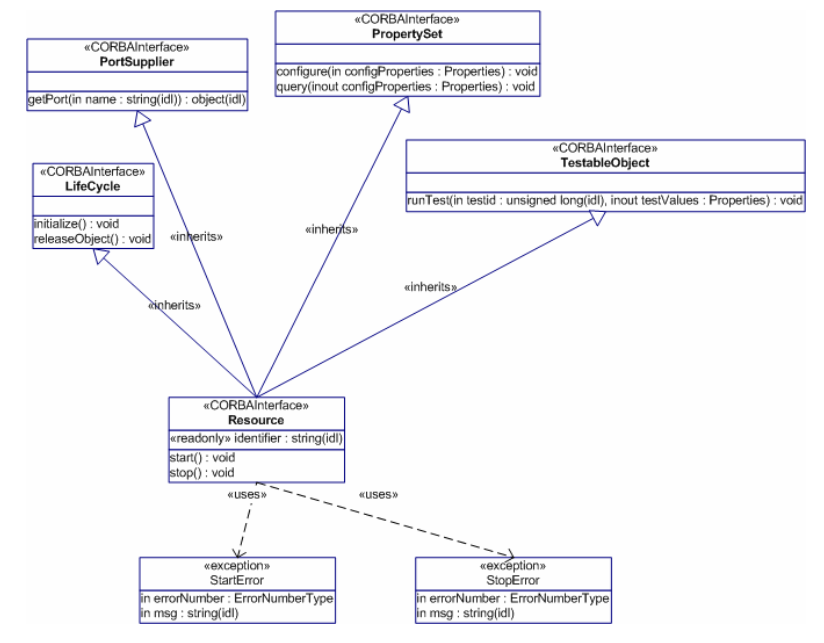


图 19组件UML

* 1. getPort

波形控制器通过getPort()接口获得组件下对应端口名字的对象引用。端口名字存放在软件组件描述(SCD)文件的port元素下，参见图 20。

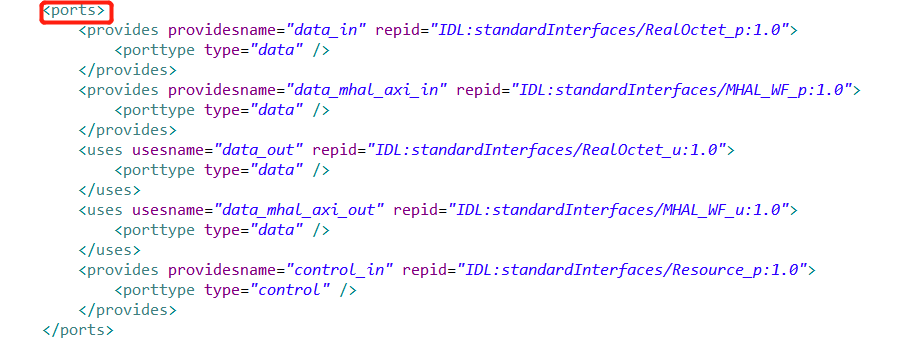


图 20软件组件描述文件(SCD)端口信息

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CORBA::Object \*  **getPort**(  **const** **char** \* name)  **throw** (  CORBA::SystemException,  CF::PortSupplier::UnknownPort); | |
| 参数 | (in)**const** **char** \* name | 端口名称 |
| 返回值 | 返回组件的日软件组件描述符(SCD)文件中相应端口名称的对象引用。 | |

* 1. initialize

initialize()操作提供一种机制将组件设置为已知的初始状态。例如，可以将数据结构设置为初始值，可以分配内存，等等，初始化行为依赖于实现。

* + 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **initialize**()  **throw** (  CF::LifeCycle::InitializeError,  CORBA::SystemException); |

* 1. releaseObject

releaseObject()操作提供一种可以销毁实例化组件的方法，释放设备在使用期间分配的所有内部内存，将组件并将其从CORBA环境中释放。

* + 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **releaseObject**()  **throw** (  CF::LifeCycle::ReleaseError,  CORBA::SystemException); |

* 1. configure

configure()操作为输入configProperties参数中指定的属性赋值。配置操作的有效属性至少应是属性描述符(PRF)文件中kindtype为configure，mode为readwrite和writeonly的模型属性，参见图 21。

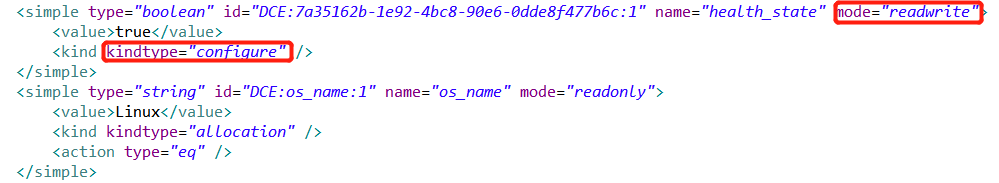


图 21属性描述文件(PRF)模型属性信息

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **configure**(  **const** CF::Properties & cofigProperties)  **throw** (  CF::PropertySet::PartialConfiguration,  CF::PropertySet::InvalidConfiguration,  CORBA::SystemException); | |
| 参数 | (in)**const** CF::Properties & cofigProperties | 需要配置的属性值 |

* 1. query

当configProperties参数长度为零时，query()接口应返回所有组件属性。如果参数长度不为零，则查询操作应仅返回configProperties参数中指定的那些属性的id/value对。

query()接口的有效属性应为所有配置属性，类型元素的kindtype属性为“configure”的简单属性， mode属性为“readwrite”或“readonly”，参见图 21。

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **query**(  CF::Properties & configProperties)  **throw** (  CF::UnknownProperties,  CORBA::SystemException); | |
| 参数 | (inout) CF::Properties & configProperties | 需要查询的属性 |

* 1. start

start()接口通过实现这个接口用来控制组件启动内部进程。在波形控制器组件中通过调用其他组件的start()接口启动波形应用的其他组件。

* + 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **start**()  **throw** (  CF::Resource::StartError,  CORBA::SystemException); |

* 1. stop

通过实现stop()接口来控制组件停止内部进程。波形控制器组件通过控制其他组件的stop()接口中断其他组件的进程。

* + 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **stop**()  **throw** (  CF::Resource::StartError,  CORBA::SystemException); |

* 1. runTest

runTest()操作允许对组件进行黑盒测试。组件要实现的测试是依赖于组件的，并在组件的属性描述符(PRF)中指定。

测试操作的有效属性应该是属性描述符(PRF)文件中kindtype为test的模型属性。

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **runTest**(  CORBA::ULong TestID,  CF::Properties & testValues)  **throw** (  CF::UnknownProperties,  CF::TestableObject::UnknownTest,  CORBA::SystemException); | |
| 参数 | (in) CORBA::ULong TestID | 测试实例id |
| (inout) CF::Properties & testValues | 待测试的模型属性 |

1. 逻辑设备接口

设备接口是域内对逻辑设备进行控制和管理的接口， 包括Device、LoadableDevice、ExecutableDevice和AggregageDevice接口。

设备管理接口主要负责逻辑设备的创建以及启动部署在这些逻辑设备上的组件。

* 1. Device

Device是对物理硬件设备的抽象，可以配置、查询设备状态（空闲、活跃、繁忙）；检查设备运行状态；初始化、销毁设备；启动、停止设备；配置、查询设备的模型属性；获取设备上的端口对象；对设备进行“黑盒”测试。

下面我们将详细介绍Device的每个接口。

* + 1. UML

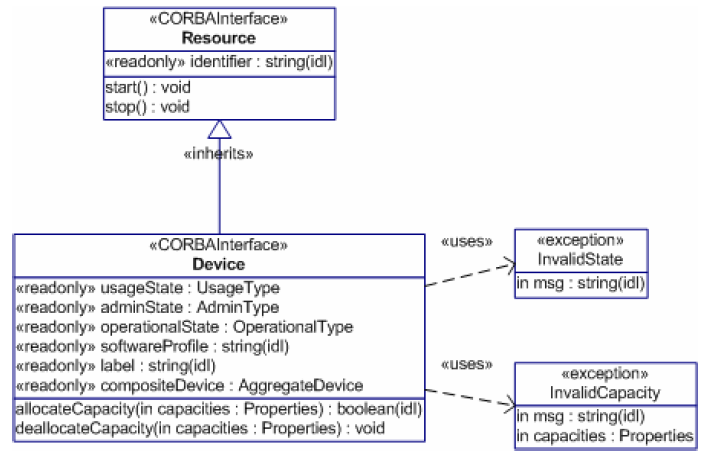


图 22设备UML

* + 1. identifier

identifier()操作返回Device实例的唯一标识符。标识符应与软件包描述符(SPD)文件的softpkg元素id属性值相同。以AD9371设备为例，参见图 23。

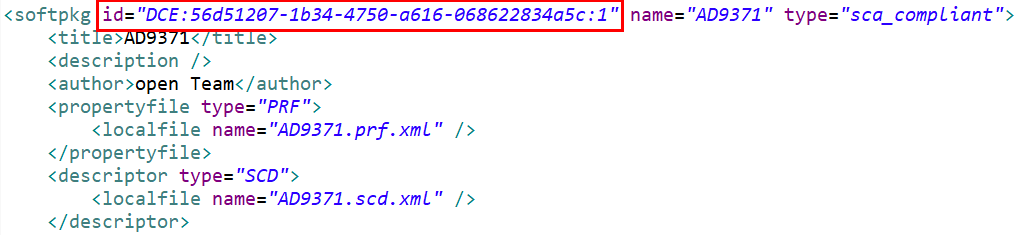


图 23 设备id

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **identifier**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回设备实例唯一标识符 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i=0; i<length; ++i){

std::string devId = (\*devs)[i]->identifier();

}

* + 1. label

label()操作返回Device实例的名称。名称应与软件包描述符(SPD)文件的softpkg元素name属性值相同。以AD9371设备为例，参见图 24。

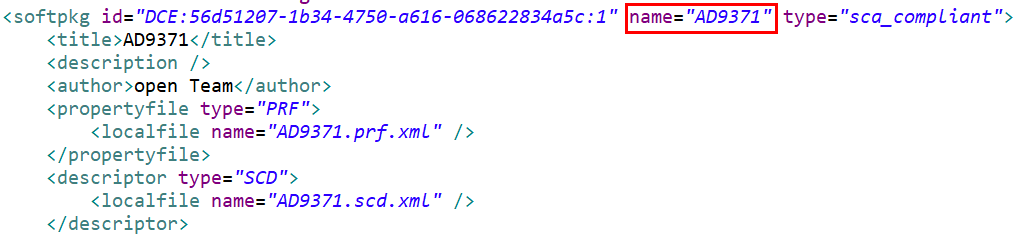


图 24设备name

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **label**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回设备实例的唯一名称 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i=0; i<length; ++i){

std::string devId = (\*devs)[i]->label();

}

* + 1. usageState

usageState()操作返回设备实例的使用状态（IDLE空闲、ACTIVE活跃、BUSY繁忙）。

IDLEVE空闲，没有在使用

ACTIVE使用活跃，在使用，剩余容量可供分配

BUSY使用繁忙，在使用，没有剩余容量可供分配

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::Device::UsageType  **usageState**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回设备实例的使用状态 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

CF::Device::UsageType devUsageState;

for(CORBA::Short i = 0; i < length; ++i){

devUsageState = (\*devs)[i]-> usageState();

//判断该设备是否还有可分配容量

if (devUsageState == CF::Device::BUSY){

//没有分配容量，抛出无效容量异常

throw CF::Device::InvalidCapacity();

}

}

* + 1. setUsageState

setUsageState()操作设置设备实例的使用状态（IDLE空闲、ACTIVE活跃、BUSY繁忙）。

IDLEVE空闲，没有在使用

ACTIVE使用活跃，在使用，剩余容量可供分配

BUSY使用繁忙，在使用，没有剩余容量可供分配

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **setUsageState**(  CF::Device::UsageType newUsageState); | |
| 参数 | (in)CF::Device::UsageType newUsageState | 设备实例新的使用状态 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

CF::Device::UsageType devUsageState;

for(CORBA::Short i = 0; i < length; ++i){

//将每个设备使用状态设置为繁忙

(\*devs)[i]->setUsageState(CF::Device::BUSY);

}

* + 1. adminState

adminState()操作返回设备实例的管理状态（LOCKED锁定、SHUTTING\_DOWN停工、UNLOCKED解锁），表示允许使用或禁止使用设备。

锁定的设置只有在管理状态为解锁时才有效。当设备的管理状态设置为锁定状态时，将解锁状态转化为停工状态，且将该设备的聚合设备设置为锁定状态；当管理状态为锁定或停工状态时，设置解锁才有效。不合法的状态转换命令将被忽略。

当设备的使用状态为空闲状态(IDLE)，将管理状态转化为锁定状态，且该设备的聚合设备设置为锁定状态。上述转换过程如下图：

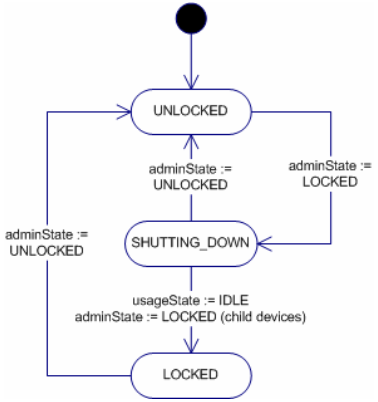


图 25 adminState状态转换图

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::Device::AdminType  **adminState**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回设备实例的管理状态 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

CF::Device::AdminType devAdminState;

for(CORBA::Short i = 0; i < length; ++i){

devAdminState = (\*devs)[i]-> adminState();

}

* + 1. adminState(CF::Device::AdminType)

adminState()操作设置设备实例的管理状态（LOCKED锁定、SHUTTING\_DOWN停工、UNLOCKED解锁），表示允许使用或禁止使用设备。

锁定的设置只有在管理状态为解锁时才有效。当设备的管理状态设置为锁定状态时，将解锁状态转化为停工状态，且将该设备的聚合设备设置为锁定状态；当管理状态为锁定或停工状态时，设置解锁才有效。不合法的状态转换命令将被忽略。

当设备的使用状态为空闲状态(IDLE)，将管理状态转化为锁定状态，且该设备的聚合设备设置为锁定状态。上述转换过程如下图：

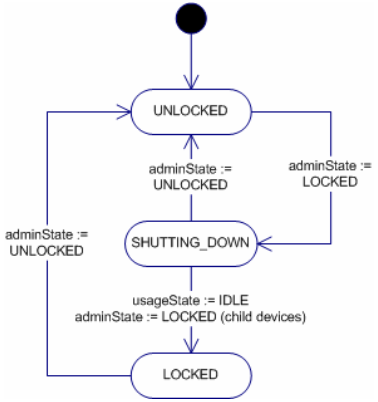


图 26 adminState状态转换图

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **adminState(**  CF::Device::AdminType adminType)  **throw** (  CORBA::SystemException); | |
| 参数 | (in)CF::Device::AdminType adminType | 设备实例新的管理状态 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

(\*devs)[i]->adminState(CF::Device::LOCKED);

}

* + 1. operationalState

operationalState()操作返回设备实例的操作状态（ENABLED启用、DISABLED禁用），表示设备是否正常运行。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::Device::OperationalType  **operationalState()**  **throw** (  CORBA::SystemException); |
| 返回值 | 返回设备实例的操作状态 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

CF::Device::OperationalType devOperationalType;

for(CORBA::Short i = 0; i < length; ++i){

devOperationalType = (\*devs)[i]-> operationalState();

}

* + 1. softwareProfile

softwareProfile()操作返回设备的软件包描述符(SPD)文件的路径，且软件包描述符(SPD)文件中的文件对象引用通过FileManager获得。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **char** \*  **softwareProfile()**  **throw** (  CORBA::SystemException); |
| 返回值 | 返回设备的软件包描述符(SPD)文件的路径 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

std::string devSoftwareProfile;

for(CORBA::Short i = 0; i < length; ++i){

devSoftwareProfile = (\*devs)[i]-> softwareProfile();

}

* + 1. compositeDevice

compositeDevice()操作返回聚合设备的对象引用。

如果一个设备有子设备，将该设备称为聚合设备，也就是父设备。

当一个设备是父设备时，返回此聚合设备的对象引用；当该设备不是父设备时，返回一个NULL。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::AggregateDevice\_ptr  **compositeDevice**()  **throw** (  CORBA::SystemException); |
| 返回值 | 返回聚合设备的对象引用 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

CF::AggregateDevice\_var aggDev;

for(CORBA::Short i = 0; i < length; ++i){

//获得此设备的对象引用

aggDev = (\*devs)[i]-> compositeDevice();

//如果对象引用不为null，则此设备为聚合设备

if (!CORBA::is\_nil(aggDev)) {  
 //获取该设备的子设备

CF::DeviceSequence\_var devSeq = aggDev->devices();

for (CORBA::UShort i = 0; i < devSeq->length(); ++i){

//释放每个子设备的资源

(\*devSeq)[i]->releaseObject();

}

}

}

* + 1. allocateCapacity

allocateCapacity()操作提供请求、分配设备承载容量(如：内存容量)的机制。

当设备的使用状态(usageState)为不繁忙(not BUSY)，操作状态(operationalState)为启用(ENABLED)，管理状态(adminState)为解锁(UNLOCKED)时，根据参数减少当前设备的承载容量。

当设备的承载容量无法分配时，allocateCapacity()接口将usageState设置为BUSY；当设备处于使用状态且有剩余承载容量时，allocateCapacity()接口需要将usageState设置为ACTIVE。

allocateCapacity()接口的参数应该为kindtype是allocation分配类型的属性，参见图 27。

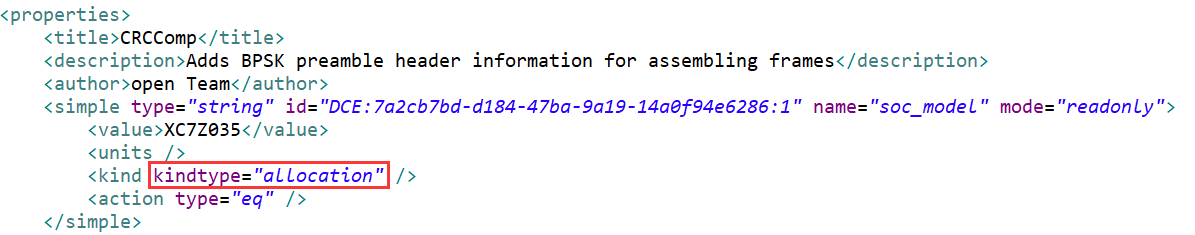


图 27属性描述符(PRF)的kindtype

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CORBA::Boolean  **allocateCapacity**(  **const** CF::Properties & capacities)  **throw** (  CF::Device::InvalidState,  CF::Device::InvalidCapacity,  CORBA::SystemException); | |
| 参数 | **(in) const** CF::Properties & capacities | kindtype为分配类型的模型属性 |
| 返回值 | 返回bool值，如果为true，设备可承载该属性；如果为false，设备承载该属性失败 | |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::Properties props;

//将属性序列长度赋为1

props.length(1);

//添加属性的id、value

props[0].id = ...;

props[0].value = ...;

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

//获得此设备的对象引用

bool allocateFlag = (\*devs)[i]-> allocateCapacity(props);

if (allocateFlag) {  
 //设备承载该属性成功

} else {

//设备承载该属性失败

}

}

* + 1. deallocateCapacity

deallocateCapacity()操作提供释放设备承载容量的机制，根据输入的属性参数调整设备承载容量。

当调整设备承载容量后，如果设备的承载容量处于使用状态时，deallocateCapacity()接口需要将usageState设置为ACTIVE；如果设备的承载容量没有被使用，deallocateCapacity()接口需要将usageState设置为IDLE。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | void  **deallocateCapacity**(  **const** CF::Properties & capacities)  **throw** (  CF::Device::InvalidState,  CF::Device::InvalidCapacity,  CORBA::SystemException); | |
| 参数 | **(in) const** CF::Properties & capacities | kindtype为分配类型的模型属性 |

* + - 1. 代码片段

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::Properties props;

//将属性序列长度赋为1

props.length(1);

//添加属性的id、value

props[0].id = ...;

props[0].value = ...;

CF::DeviceSequence\_var devs = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

(\*devs)[i]-> deallocateCapacity(props);

}

* + 1. initialize

initialize()操作提供一种机制将设备设置为已知的初始状态。例如，可以将数据结构设置为初始值，可以分配内存，可以将硬件设备配置为某种状态，等等，初始化行为依赖于实现。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **initialize**()  **throw** (  CF::LifeCycle::InitializeError,  CORBA::SystemException); |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

try{

dev->initialize();

catch(...){

...

}

* + 1. releaseObject

releaseObject()操作提供一种可以拆卸实例化设备的方法，释放设备在使用期间分配的所有内部内存，将分解设备并将其从CORBA环境中释放

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **releaseObject**()  **throw** (  CF::LifeCycle::ReleaseError,  CORBA::SystemException); |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

try{

dev->releaseObject();

catch(...){

...

}

* + 1. start

start()操作命令实现此接口的设备启动内部处理，将资源置于操作状态。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **start**()  **throw** (  CF::Resource::StartError,  CORBA::SystemException); |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

try{

dev->start();

catch(...){

...

}

* + 1. stop

stop()操作命令实现此接口的设备停止内部处理，禁用所有当前操作，并将设备置于非操作状态。停止操作不应禁止随后的配置、查询和启动操作。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **stop**()  **throw** (  CF::Resource::StopError,  CORBA::SystemException); |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

try{

dev->stop();

catch(...){

...

}

* + 1. configure

configure()操作为输入configProperties参数中指定的属性赋值。配置操作的有效属性至少应是属性描述符(PRF)文件中kindtype为configure，mode为readwrite和writeonly的模型属性。

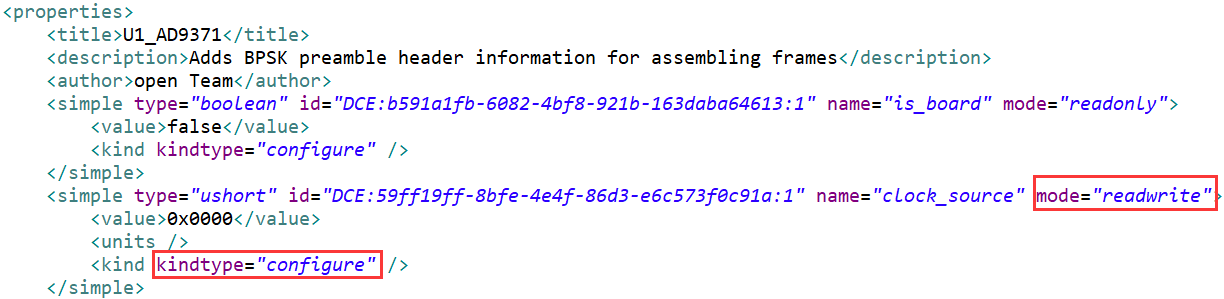


图 28属性描述符文件(PRF) 模型属性信息

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **configure**(  **const** CF::Properties & configProperties)  **throw** (  CF::PropertySet::PartialConfiguration,  CF::PropertySet::InvalidConfiguration,  CORBA::SystemException); | |
| 参数 | **(in) const** CF::Properties & configProperties | 需要被配置的模型属性 |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

//需要被设置的属性

CF::Properties props = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

try{

dev->configure(props);

catch(...){

...

}

* + 1. query

query()操作当参数configProperties为零时，查询操作将返回所有设备属性。如果参数不是零大小，查询操作将只返回configProperties参数中指定的id/值对。

查询操作的有效属性应该是属性描述符(PRF)文件中kindtype为configure，mode为readwrite或readonly的模型属性。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **query**(  CF::Properties & configProperties)  **throw** (  CF::UnknownProperties,  CORBA::SystemException); | |
| 参数 | **(in)const** CF::Properties & configProperties | 需要查询的模型属性 |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

//查询所有模型属性

CF::Properties props;

try{

dev->query(props);

catch(...){

...

}

...

//查询指定模型属性

CF::Properties props;

props.length(2);

props[0].id = ...;

props[1].id = ...;

try{

dev->query(props);

catch(...){

...

}

* + 1. getPort

getPort()操作提供了一种机制来获取特定的输入、输出端口。一个设备可以包含0到多个输入、输出端口。确切的数字在组件的软件组件描述符(SCD)文件中指定。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CORBA::Object\_ptr  **getPort**(  **const** **char** \* name)  **throw** (  CF::PortSupplier::UnknownPort,  CORBA::SystemException); | |
| 参数 | **(in)const** **char** \* name | 需要获取端口对象引用的端口名称 |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

//需要获取端口对象引用的端口名称

std::string portName = ...;

try{

dev->getPort(portName.c\_str());

catch(...){

...

}

* + 1. runTest

runTest()操作允许对设备进行黑盒测试。设备要实现的测试是依赖于设备的，并在设备的属性描述符(PRF)中指定。

测试操作的有效属性应该是属性描述符(PRF)文件中kindtype为test的模型属性。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **runTest**(  CORBA::ULong TestID,  CF::Properties & testValues)  **throw** (  CF::UnknownProperties,  CF::TestableObject::UnknownTest,  CORBA::SystemException); | |
| 参数 | **(in)** CORBA::ULong TestID | 确定应该执行的预定义的测试实现 |
| **(inout)** CF::Properties & testValues | 为要运行的特定于实现的测试提供附加信息 |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

//需要被测试的属性

CF::Properties props = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

try{

dev->runTest(props);

catch(...){

...

}

* 1. LoadableDevice

该接口通过向设备添加软件加载和卸载行为来扩展设备接口。

* + 1. UML

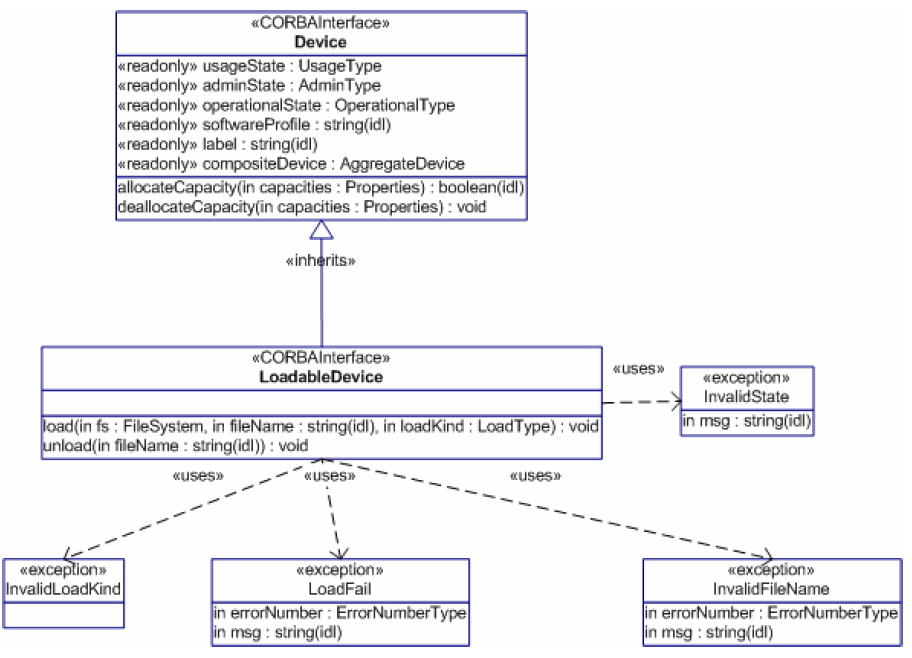


图 29可加载设备UML

* + 1. load

load()操作提供了在设备上加载软件的机制。如果该设备是可执行设备，则所加载的软件可以随后在该设备上执行。

fileName参数是相对于fs参数标识的文件系统的路径名，fileName参数所指示的同一文件的多次加载不应导致异常。但是，load操作应该考虑到会发生的这种多重加载，以便可以执行卸载操作行为。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **load**(  CF::FileSystem\_ptr fs,  **const** **char** \* fileName,  CF::LoadableDevice::LoadType loadKind)  **throw** (  CF::LoadableDevice::LoadFail,  CF::InvalidFileName,  CF::LoadableDevice::InvalidLoadKind,  CF::Device::InvalidState,  CORBA::SystemException); | |
| 参数 | (in)CF::FileSystem\_ptr fs | 需要获取端口对象引用的端口名称 |
| (in)**const** **char** \* fileName | 需要加载的文件名 |
| (in)CF::LoadableDevice::LoadType loadKind | 需要加载的文件类型 |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

CF::LoadableDevice\_var loadabledev = CF::LoadableDevice::\_narrow(dev);

//需要获取端口对象引用的端口名称

CF::FileSystem\_var fs = ...;

std::string binaryFileName = ...;

CF::LoadableDevice::LoadType loadType = ...;

try{

loadabledev->load(fs, binaryFileName, loadType);

catch(...){

...

}

* + 1. unload

unload()操作提供了卸载当前加载的软件的机制，当卸载请求的数量与指定文件的加载请求的数量匹配时，卸载操作将从设备中卸载fileName参数标识的文件。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **unload**(  **const** **char** \* fileName)  **throw** (  CF::InvalidFileName,  CF::Device::InvalidState,  CORBA::SystemException); | |
| 参数 | (in)**const** **char** \* fileName | 需要卸载的文件名 |

* + - 1. 代码片段

#define DEVICE\_ID ("DCE:4f554efe-37fd-4381-a7cb-4ab66975de0e:1")

//通过域管理器获得已注册的设备管理器序列，通过设备管理器id，获取设备管理器，通过设

//备管理器，获取已注册的设备序列，可参考5.5.4.2代码片段

CF::DeviceSequence\_var devs = ...;

CF::Device\_var dev = ...;

CORBA::Short length = devs->length();

for(CORBA::Short i = 0; i < length; ++i){

if(0 == strcmp((\*devs)[i]->identifier(), DEVICE\_ID)){

dev = (\*devs)[i];

}

}

CF::LoadableDevice\_var loadabledev = CF::LoadableDevice::\_narrow(dev);

//需要获取端口对象引用的端口名称

std::string fullCodeFile = ...;

try{

loadabledev->unload(fullCodeFile);

catch(...){

...

}

* 1. ExecutableDevice

该接口通过向设备添加执行和终止行为来扩展LoadableDevice接口。

* + 1. UML

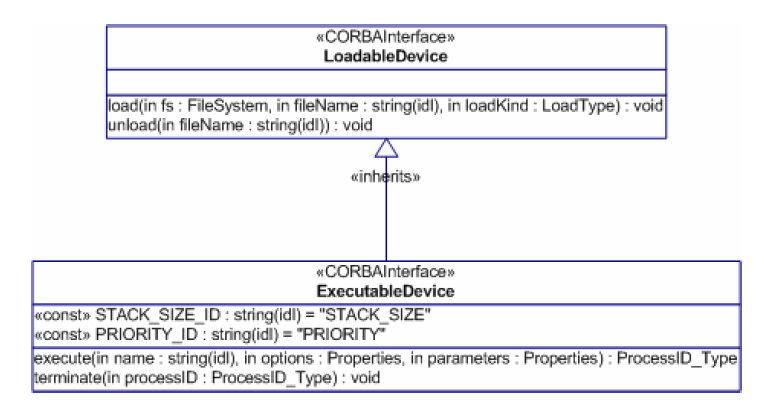


图 30可执行设备UML

* + 1. execute

execute()操作提供了在设备上启动和执行软件进程/线程的机制,使用options参数和parameters参数执行由name参数标识的文件。

parameters(id/值字符串对)参数转换为POSIX exec函数族的标准argv，其中argv(0)是函数名。执行操作将从索引1开始将parameters参数映射到argv，argv(1)映射到输入参数(0)id, argv(2)映射到输入参数(0)值，以此类推。执行操作通过操作系统执行函数传递argv。

options参数是STACK\_SIZE\_ID和PRIORITY\_ID。当指定执行操作时，应使用这些选项为给定name参数的可执行文件设置操作系统进程/线程堆栈大小和优先级。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CF::ExecutableDevice::ProcessID\_Type  **execute**(  **const** **char** \* name,  **const** CF::Properties & options,  **const** CF::Properties & parameters)  **throw** (  CF::ExecutableDevice::ExecuteFail,  CF::InvalidFileName,  CF::ExecutableDevice::InvalidOptions,  CF::ExecutableDevice::InvalidParameters,  CF::ExecutableDevice::InvalidFunction,  CF::Device::InvalidState,  CORBA::SystemException); | |
| 参数 | (in)**const** **char** \* name | 执行文件路径 |
| (in) **const** CF::Properties & options | 输入的选项参数 |
| (in) **const** CF::Properties & parameters | 输入的参数参数 |

* + - 1. 代码片段

//options参数内容从设备的软件包描述符(SPD)文件中获取

std::string codeFile = ...;

std::string entryPoint = ...;

CORBA::ULong stackSize = ...;

CORBA::ULong priority = ...;

CF::Properties options;

options.length(3);

options[0].id = CORBA::string\_dup("STACK\_SIZE\_ID");

options[0].value <<= (CORBA::ULong)stackSize;

options[1].id = CORBA::string\_dup("PRIORITY\_ID");

options[1].value <<= (CORBA::ULong)priority;

options[2].id = CORBA::string\_dup("ENTRY\_POINT");

options[2].value <<= entryPoint.c\_str();

//从属性描述符(PRF)中获取可执行的参数,

ExecParamsType execParams = ...;

CF::Properties params;

int exLen = execParams.size()；

params.length(6 + exLen);

//params从0开始将execParams存入

...

//从设备配置描述符(DCD)文件中获取

std::string devInstId = ...;

std::string devInstName = ...;

std::string devSpdPath = ...;

std::string nodeName = ...;

std::string label = ...;

//父设备的ior

std::string parentIOR = ...;

CF::DataType devId;

devId.id = CORBA::string\_dup("-ID");

devId.value <<= devInstId;

params[exLen + 0] = devId;

CF::DataType devName;

devName.id = CORBA::string\_dup("-LABEL");

devName.value <<= devInstName;

params[exLen + 1] = devName;

CF::DataType devSftfl;

devSftfl.id = CORBA::string\_dup("-SFTFL");

devSftfl.value <<= devSpdPath;

params[exLen + 2] = devSftfl;

CF::DataType node;

node.id = CORBA::string\_dup("-NODENAME");

node.value <<= nodeName;

params[exLen + 3] = node;

CF::DataType devMgrName;

devMgrName.id = CORBA::string\_dup("-DEVMGRNAME");

devMgrName.value <<= label;

params[exLen + 4] = devMgrName;

CF::DataType parentIor;

parentIor.id = CORBA::string\_dup("-COMPOSITE\_DEVICE\_IOR");

parentIor.value <<= parentIOR;

params[exLen + 5] = parentIor;

ExecutableDevice\_servant \* exDevice = new ExecutableDevice\_servant();

//通过codeFile拼接获取

std::string relativeCodePath = ...;

try {

exeDevServant->execute(relativeCodePath.c\_str(), options, params);

} catch (...) {

...

}

* + 1. terminate

terminate()操作提供了一种机制，用于终止进程/线程在使用execute()操作启动的设备上的执行。将终止processId输入参数指定的进程/线程在设备上的执行。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **terminate**(  CF::ExecutableDevice::ProcessID\_Type processId)  **throw** (  CF::Device::InvalidState,  CF::ExecutableDevice::InvalidProcess,  CORBA::SystemException); | |
| 参数 | (in) CF::ExecutableDevice::ProcessID\_Type processId | 需要终止的进程/线程id |

* + - 1. 代码片段

//需要终止的进程/线程id

unsigned long procId = ...;

CF::ExecutableDevice\_var exeDevice = ...;

try{

if (!CORBA::is\_nil(exeDevice)) {

exeDevice->terminate(procId);

}

catch(...){

...

}

* 1. AggregateDevice

AggregateDevice接口提供了从父设备中添加和删除子设备所需的行为。此接口可以通过继承提供，也可以作为用作父设备的任何设备的提供端口提供。当创建或关闭时，子设备使用此接口将自己从父设备中添加或删除。

* + 1. UML

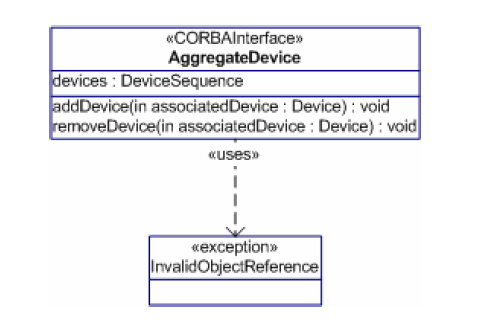


图 31集合设备UML

* + 1. devices

devices()操作获取已添加到此设备的设备列表，如果该设备与其他设备没有聚合关系，则该设备的序列长度应为零。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | CF::DeviceSequence \*  **devices**(); |

* + - 1. 代码片段

//获取PS可执行设备

std::string contextName =

"OpenSCA\_Domain/Single\_Node/Device\_Manager/Zynq7035\_PS";

CORBA::Object\_ptr obj =

m\_orbWrap->get\_object\_from\_string(contextName.c\_str());

if (CORBA::is\_nil(obj)) {

//obtain obj fail.

}

CF::ExecutableDevice\_var dev;

try {

dev = CF::ExecutableDevice::\_narrow(obj);

} catch (...) {

...

}

//获取PS可执行设备的子设备列表deviceSeq

try {

CF::DeviceSequence\_var deviceSeq = dev->compositeDevice()->devices();

} catch (...) {

...

}

* + 1. addDevice

addDevice()操作提供了将一个设备与另一个设备关联的机制。当一个设备改变状态或被拆除时，它的相关设备将受到影响。

当关联设备不存在于设备序列中时，addDevice操作应将associatedDevice参数添加到设备序列中。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **addDevice**(  CF::Device\_ptr associatedDevice)  **throw**(  CF::InvalidObjectReference); | |
| 参数 | (in)CF::Device\_ptr associatedDevice | 需要被关联的设备 |

* + - 1. 代码片段

//获取到父设备的ior

std::string compositeDevIor = ...;

//compositeDev设备的子设备的对象引用

CF::Device\_var dev = ...;

try {

CORBA::Object\_var obj =

orbWrap->ior\_to\_object(compositeDevIor.c\_str());

CF::Device\_var compositeDev = CF::Device::\_nil();

//通过ior获取父设备

compositeDev = CF::Device::\_narrow(obj.in());

compositeDev->compositeDevice()->addDevice(dev.in());

} catch(...) {

...

}

* + 1. removeDevice

removeDevice()操作提供了将一个设备与另一个设备分离的机制，从设备序列中删除associatedDevice参数关联的设备。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **removeDevice**(  CF::Device\_ptr associatedDevice)  **throw**(  CF::InvalidObjectReference); | |
| 参数 | (in)CF::Device\_ptr associatedDevice | 需要被分离的设备 |

* + - 1. 代码片段

//获取到父设备的ior

std::string compositeDevIor = ...;

//compositeDev设备的子设备的对象引用

CF::Device\_var dev = ...;

try {

CORBA::Object\_var obj =

orbWrap->ior\_to\_object(compositeDevIor.c\_str());

CF::Device\_var compositeDev = CF::Device::\_nil();

//通过ior获取父设备

compositeDev = CF::Device::\_narrow(obj.in());

compositeDev->compositeDevice()->removeDevice(dev.in());

} catch(...) {

...

}

1. 端口及连接
   1. 端口数据传输机制

openSCA2.2.2通过端口传输数据，端口分为输入端口(ProvidesPort，简称P端口)和输出端口(UsesPort，简称U端口)。

一个组件或设备可以包含0到多个输入、输出端口。确切的数字在组件的软件组件描述符(SCD)文件中指定。

调用U端口的pushPacket()发送数据，相当于调用与U端口建立连接的P端口的pushPacket()发送数据，再通过P端口的getData()接收数据。

端口中传输数据为了实现同步采用boost信号(signals2)，需要通过connectSlot()接口将外部接收数据操作与信号插槽绑定，当外部发送数据操作调用U端口的pushPacket()接口发送数据时，U端口的pushPacket()接口调用P端口的pushPacket()接口，在P端口的pushPacket()接口中，将数据写入缓存后，发送一个信号(signalDef)，触发接收数据操作，开始接收数据。端口数据传输机制的序列图如下：

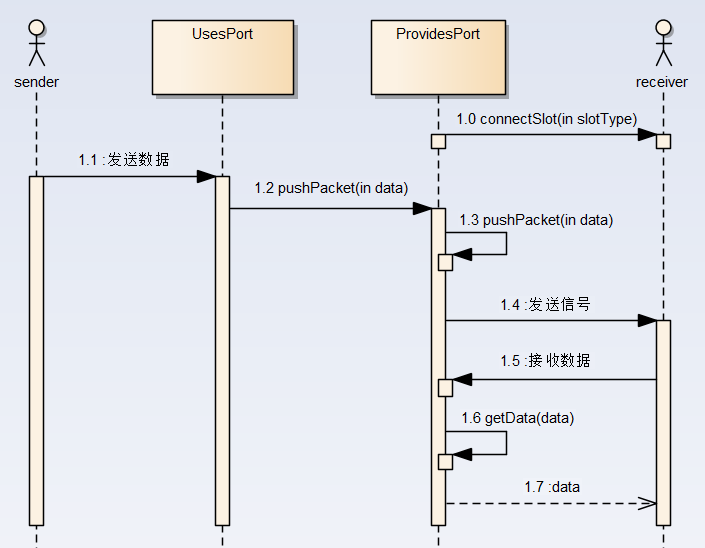


图 32端口数据传输机制序列图

//待发送的数据

JTRS::OctetSequence sendSeq = ...;

StandardInterfaces\_i::RealOctet\_u \* uPort = ...;

StandardInterfaces\_i::RealOctet\_p \* pPort = ...;

//connectSlot绑定接收数据操作

pPort->connectSlot(boost::bind(&receiveDate));

//U端口发送数据

uPort->pushPacket(sendSeq);

...

void \*

receiveData(

void \* p)

{

JTRS::OctetSequence recvSeq;

//P端口接收数据

pPort->getData(recvSeq);

}

* 1. 端口数据传输类型

端口传输的数据类型，现支持的有short、long、float、double、octet。

* 1. UsesPort接口

输出接口可以建立或断开与P端口的连接、发送数据。

下面以RealOctet\_u为例，介绍开发者常用接口，getPort()、connectPort()、disconnectPort()、pushPacket()。

* + 1. UML

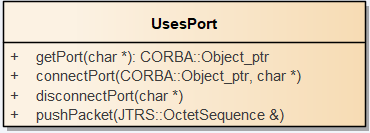


图 33输出端口UML

* + 1. getPort

getPort()操作获取输出端口的对象引用。一个波形组件或设备可以有多个输出端口，确切的数量和端口名称可以通过软件组件描述符(SCD)文件查看，参见图 34。



图 34软件组件描述符(SCD)输出端口信息

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CORBA::Object\_ptr  **getPort**(  **const** **char** \* portName); | |
| 参数 | (in) **const** **char** \* portName | 待获取对象引用的端口名称 |
| 返回值 | 返回参数portName端口的对象引用 | |

* + - 1. 代码片段

StandardInterfaces\_i::RealOctet\_u \* uPort = ...;

std::string portName = ...;

CORBA::Object\_var port = uPort->getPort(portName.c\_str());

* + 1. connectPort

connectPort()操作建立端口之间的关联，一个端口支持多个连接，为端口提供数据传输和控制的通道。

波形组件的端口连接信息可以在软件装配描述符(SAD)文件中查看，设备的端口连接信息可以在设备配置描述符(DCD)文件中查看，参见图 35、图 36。

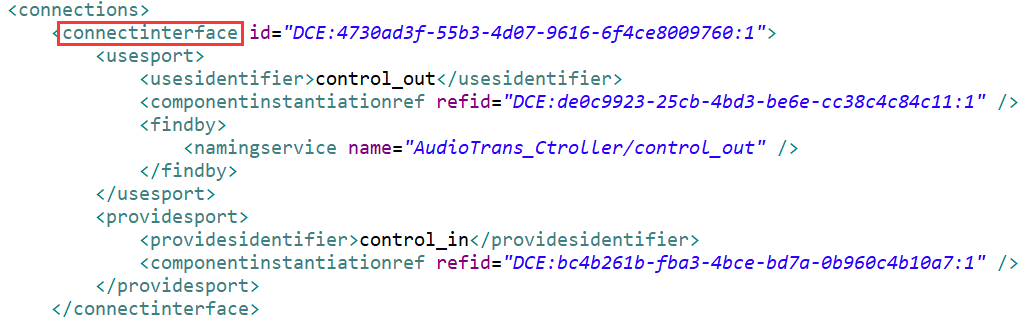


图 35软件装配描述符(SAD)组件连接信息



图 36设备配置描述符(DCD)逻辑设备连接信息

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **connectPort**(  CORBA::Object\_ptr connection,  **const** **char** \* connectionID); | |
| 参数 | (in) CORBA::Object\_ptr connection | 与U端口建立连接的P端口的对象引用 |
| (in) **const** **char** \* connectionID | 连接标识符，唯一标识此连接，作为断开连接时的判断条件 |

* + - 1. 代码片段

CF::PPort\_var usesPort = ...;

Connection \* connect = ...;

//getPort获取P端口，可参考7.3.1.2.2

CORBA::Object\_var providesObjRef = ...;

usesPort->connectPort(providesObjRef, connect->getID());

* + 1. disconnectPort

disconnectPort()操作断开参数connectionID标识的端口之间的连接。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **disconnectPort**(  **const** **char** \* connectionID); | |
| 参数 | (in) **const** **char** \* connectionID | 待断开连接的标识符 |

* + - 1. 代码片段

CF::PPort\_var usesPort = ...;

Connection \* connect = ...;

usesPort->disconnectPort(connect->getID());

* + 1. pushPacket

pushPacket()操作可以发送数据，该接口实际调用的是P端口的pushPacket()接口，P端口的pushPacket()接口将数据写入缓存后，发送一个信号，触发信号插槽绑定的接收数据接口，从而达到数据的同步传输。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **pushPacket**(  **const** JTRS::OctetSequence & data); | |
| 参数 | (in) **const** JTRS::OctetSequence & data | 待传输的数据 |

* + - 1. 代码片段

//待发送的字符串

std::string sendStr = "[\*\*\*\*\*Hello JFounder!\*\*\*\*\*]";

JTRS::OctetSequence sendSeq;

//设置序列的长度

sendSeq.length(sendStr.length());

//将待发送的字符串转为序列

memcpy(sendSeq.get\_buffer(), sendStr.c\_str(), sendStr.length());

StandardInterfaces\_i::RealOctet\_u \* usesPort = ...;

usesPort->pushPacket(sendSeq);

* 1. ProvidesPort接口

输入接口可以建立、断开信号的连接槽，接收数据。

下面以RealOctet\_p为例，介绍开发者常用接口，getPort()、connectSlot()、disconnectSlot()、getData()。

* + 1. UML

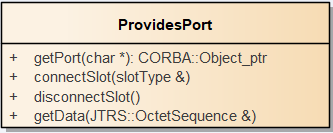


图 37输入端口UML

* + 1. getPort

getPort()操作获取输入端口的对象引用。一个波形组件或设备可以有多个输入端口，确切的数量和端口名称可以通过软件组件描述符(SCD)文件查看，参见图 38。



图 38软件组件描述符(SCD)输入端口信息

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CORBA::Object\_ptr  **getPort**(  **const** **char** \* portName); | |
| 参数 | (in) **const** **char** \* portName | 待获取对象引用的端口名称 |
| 返回值 | 返回参数portName端口的对象引用 | |

* + - 1. 代码片段

StandardInterfaces\_i::RealOctet\_p \* pPort = ...;

std::string portName = ...;

CORBA::Object\_var port = pPort->getPort(portName.c\_str());

* + 1. connectSlot

connectSlot()操作将外部接收函数与信号插槽关联，在P端口的pushPacket()接口发送信号时，可以触发外部接收函数，接收数据。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **connectSlot**(  **const** slotType & slot); | |
| 参数 | (in) **const** slotType & slot | 待绑定的外部接收函数 |

* + - 1. 代码片段

#include <boost/signals2/signal.hpp>

//connectSlot关联的函数，需要不带形参

void

receiveDate()

{

...

}

...

StandardInterfaces\_i::RealOctet\_p \* pPort = ...;

pPort->connectSlot(boost::bind(&receiveData));

* + 1. disconnectSlot

disconnectSlot()操作将外部接收函数与信号插槽断开关联，可以在停止或卸载波形应用时调用。

* + - 1. 语义

|  |  |
| --- | --- |
| 原型 | **void**  **disconnectSlot**(); |

* + - 1. 代码片段

#include <boost/signals2/signal.hpp>

...

StandardInterfaces\_i::RealOctet\_p \* pPort = ...;

pPort->disconnectSlot();

* + 1. getData

getData()操作接收U端口输出的数据。当P端口的pushPacket()接口发送信号，触发信号插槽连接的接口，运行getData()接口，获取数据，实现同步机制。

* + - 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **getData**(  JTRS::OctetSequence & data); | |
| 参数 | (out) JTRS::OctetSequence & data | 接收数据的序列容器 |

* + - 1. 代码片段

JTRS::OctetSequence recvSeq;

StandardInterfaces\_i::RealOctet\_p \* pPort = ...;

//接收数据时，接收序列，不用规定长度

pPort->getData(recvSeq);

* 1. 外部程序与波形应用的连接

通过软件装配描述符(SAD)文件可以查看波形应用端口的连接信息，波形应用端口连接可分为两种类型：内部连接、外部连接，参见图 39、图 40。

内部连接——波形应用内部端口间的连接

外部连接——波形应用与外部程序间的连接

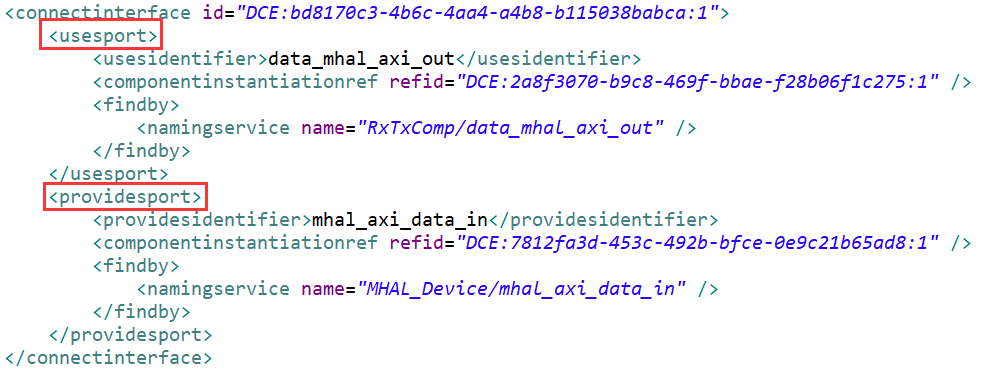
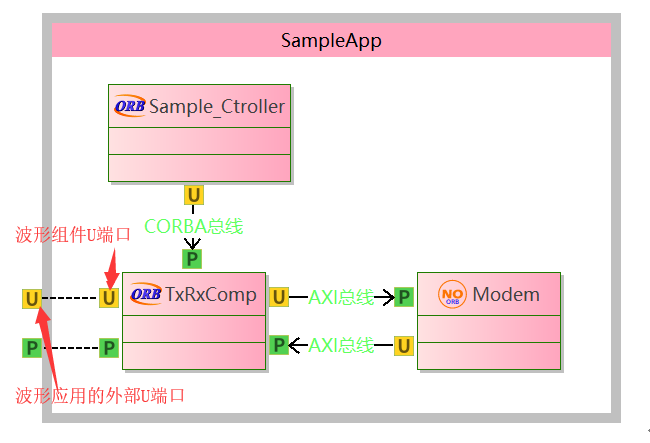


图 39软件装配描述符(SAD)中内部连接



图 40软件装配描述符(SAD)中外部连接

外部连接是通过软件装配描述符(SAD)文件中connectinterface元素下外部端口的namingservice的name值得到与其相连接端口的对象引用，而内部连接得到的是该端口的对象引用。



//通过域管理器获取域内波形应用序列，获取当前波形应用，可参考5.3.2.2代码片段

CF::Application\_var app = ...;

//波形应用的外部U端口名称

std::string usePortName = ...;

try{

//得到与波形应用的外部U端口相连的波形组件U端口的对象引用usePortObject

CORBA::Object\_var usePortObject = app->getPort(usePortName.c\_str());

if(CORBA::is\_nil(usePortObject)){

//usePortObject is nil.

return;

}

CF::PPort\_var targetPort = CF::PPort::\_narrow(usePortObject.in());

if(CORBA::is\_nil(targetPort)){

//targetPort is nil.

return;

}

//创建一个外部程序P端口对象

StandardInterfaces::RealOctet\_var providePort = ...;

//创建外部程序P端口的id对象

PortableServer::ObjectId\_var providePortId =

PortableServer::string\_to\_ObjectId("wave\_consumer\_port");

openscaSupport::ORB\_Wrap \* orbWrap = new openscaSupport::ORB\_Wrap();

CORBA::Object\_var pPort;

try {

orbWrap->poa->activate\_object\_with\_id(providePortId.in(), providePort);

//获取外部程序P端口的对象引用

pPort = orbWrap->poa->servant\_to\_reference(providePort);

} catch (...) {

...

}

if(CORBA::is\_nil(pPort)){

//pPort is nil.

return;

}

// 波形组件U端口连接外部程序P端口

targetPort->connectPort(pPort.in(), "wave\_consumer\_port");

// 波形应用的外部P端口的名称

std::string providePortName = ...;

// 得到与波形应用的外部P端口相连的波形组件P端口的对象引用providePortObject

CORBA::Object\_var providePortObject = app->getPort(providePortName.c\_str());

if(CORBA::is\_nil(providePortObject)){

//providePortObject is nil.

return;

}

//创建外部程序U端口，该U端口为CWave\_Use\_port\_i

CWave\_Use\_port\_i \* usePort = new CWave\_Use\_port\_i();

// 波形组件P端口连接外部程序U端口

usePort->connectPort(providePortObject.in(), " wave\_producer\_port");

} catch (...) {

...

}

1. ORB抽象接口

本章介绍框架中封装的常用的CORBA接口。

* 1. UML

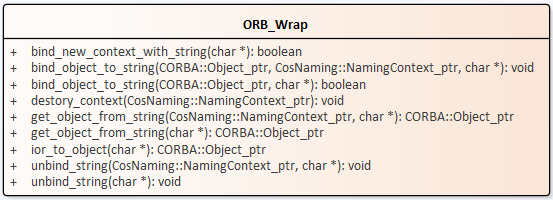


图 41 ORB抽象接口UML

* 1. bind\_object\_to\_string

bind\_object\_to\_string()操作将对象引用与指定的命名上下文绑定。

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **bind\_object\_to\_string**(  CORBA::Object\_ptr obj,  **const** CosNaming::NamingContext\_ptr nc,  **const** **char** \* name); | |
| 参数 | (in) CORBA::Object\_ptr obj | 需要绑定的对象引用 |
| (in) **const** CosNaming::NamingContext\_ptr nc | 对象引用绑定的命名上下文 |
| (in) **const** **char** \* name | 对象引用对应的上下文名称 |
|  | | |
| 原型 | **bool**  **bind\_object\_to\_string**(  CORBA::Object\_ptr obj,  **const** **char** \* name); | |
| 参数 | (in) CORBA::Object\_ptr obj | 需要绑定的对象引用 |
| (in) **const** **char** \* name | 对象引用对应的上下文名称 |
| 返回值 | 绑定成功返回true,失败返回false | |

* + 1. 代码片段

//以下为将RxTxComp对象引用与命名上下文进行绑定的代码实现

//实例化 orb

openscaSupport::ORB\_Wrap \* orbWrap = **new** openscaSupport::ORB\_Wrap();

//实例化 component servant

RxTxComp\_servant \* compServant = ...;

//将组件绑定到cosName下

std::string cosName = ...;

CF::Resource\_var gppComp\_v= compServant->\_this();

//绑定上下文

orbWrap->bind\_object\_to\_string(gppComp\_v.in(), cosName.c\_str());

...

**delete** orbWrap;

**delete** compServant;

* 1. get\_object\_from\_string

get\_object\_from\_string()操作根据指定上下文字符串获得对应的对象引用。

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CORBA::Object\_ptr  **get\_object\_from\_string**(  **const** CosNaming::NamingContext\_ptr nc,  **const** **char** \* name); | |
| 参数 | (in) **const** CosNaming::NamingContext\_ptr nc | 对象引用绑定的命名上下文 |
| (in) **const** **char** \* name | 对象引用对应的上下文名称 |
| 返回值 | 返回与命名上下文绑定的对象引用 | |
|  | | |
| 原型 | CORBA::Object\_ptr  **get\_object\_from\_string**(  **const** **char** \* name); | |
| 参数 | (in) **const** **char** \* name | 命名上下文名称 |
| 返回值 | 返回与命名上下文名字称绑定的对象引用 | |

* + 1. 代码片段

//以下为根据命名上下文名称获取与其绑定的对象引用的代码实现

//实例化 orb

openscaSupport::ORB\_Wrap \* orbWrap = **new** openscaSupport::ORB\_Wrap();

std::string contextName = ...;

//获取与contextName绑定的对象引用

CORBA::Object\_ptr obj =

orbWrap ->get\_object\_from\_string(contextName.c\_str());

* 1. unbind\_string

unbind\_string()操作将命名上下文的名称与命名上下文对象之间的绑定解除。

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **unbind\_string**(  **const** CosNaming::NamingContext\_ptr nc,  **const** **char** \* name); | |
| 参数 | (in) **const** CosNaming::NamingContext\_ptr nc | 命名上下文 |
| (in) **const** **char** \* name | 命名上下文名称 |
|  | | |
| 原型 | **void**  **unbind\_string**(  **const** **char** \* name); | |
| 参数 | (in)**const** **char** \* name | 命名上下文名称 |

* + 1. 代码片段

//以下为将命名上下文名称与其绑定的命名上下文解除绑定的代码实现

//实例化 orb

openscaSupport::ORB\_Wrap \* orbWrap = **new** openscaSupport::ORB\_Wrap();

std::string contextName = ...;

...

//解除绑定

**try** {

orbWrap->unbind\_string(contextName.c\_str());

} **catch**(...) {

...

**throw** CF::LifeCycle::ReleaseError();

}

* 1. bind\_new\_context\_with\_string

bind\_new\_context\_with\_string()操作生成一个新的命名上下文对象并绑定命名上下文名称。

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **bool**  **bind\_new\_context\_with\_string**(  **const** **char** \* name); | |
| 参数 | (in)**const** **char** \* name | 命名上下文名称 |
| 返回值 | 成功返回true,失败返回false | |

* + 1. 代码片段

//实例化 orb

openscaSupport::ORB\_Wrap \* orbWrap = **new** openscaSupport::ORB\_Wrap();

std::string contextName = ...;

...

//创建一个新的命名上下文并绑定名称

**if**( **false** == orbWrap->bind\_new\_context\_with\_string( compNaming.c\_str()))

**throw** CF::LifeCycle::InitializeError();

* 1. destory\_context

destory\_context()操作撤消一个命名上下文。只有一个上下文为空(没有任何绑定)时，才能被撤消。

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | **void**  **destory\_context**(  **const** CosNaming::NamingContext\_ptr nc); | |
| 参数 | (in) **const** CosNaming::NamingContext\_ptr nc | 命名上下文 |

* + 1. 代码片段

//实例化 orb

openscaSupport::ORB\_Wrap \* orbWrap = **new** openscaSupport::ORB\_Wrap();

CosNaming::NamingContext\_ptr nc = ...;

..

//销毁命名上下文

**try** {

orbWrap->destory\_context( nc );

} **catch**(CosNaming::NamingContext::NotEmpty) {

...

}

* 1. ior\_to\_object

ior\_to\_object()操作将ior转换为对应的对象引用。

* + 1. 语义

|  |  |  |
| --- | --- | --- |
| 原型 | CORBA::Object\_ptr  **ior\_to\_object**(  **const** **char** \* ior); | |
| 参数 | (in) **const** **char** \* ior | ior字符串 |
| 返回值 | 返回ior对应的对象引用 | |

* + 1. 代码片段

//实例化 orb

openscaSupport::ORB\_Wrap \* orbWrap = **new** openscaSupport::ORB\_Wrap();

std::string compositeDevIor = ...;

...

//子设备向父设备注册

**try** {

//将ior转换为对象引用

CORBA::Object\_var obj =

orbWrap->ior\_to\_object( compositeDevIor.c\_str() );

CF::Device\_var compositeDev = CF::Device::\_nil();

//紧缩为设备的对象引用

compositeDev = CF::Device::\_narrow(obj.in());

//注册设备

compositeDev->compositeDevice()->addDevice(compositeDev.in());

} **catch**(...) {

...

}

1. 域描述文件

系统域中的波形组件和逻辑设备由一组文件描述，这些文件统称为域描述文件。

域描述文件描述系统域中的波形组件和逻辑设备的标识、功能、属性、相互依赖关系和位置。openSCA2.2.2通过文件解析器(XMLParser)读取数据，对波形组件和逻辑设备进行配置、连接等操作。

域描述文件包括：

1. 软件包描述文件(Software Package Descriptor SPD)
2. 软件组件描述文件(Software Component Descriptor SCD)
3. 波形装配描述文件(Software Assembly Descriptor SAD)
4. 属性描述文件(Properties Descriptor PRF)
5. 设备包描述文件(Device Package Descriptor DPD)
6. 设备配置描述文件(Device Configuration Descriptor DCD)
7. 域管理器配置描述文件(DomainManager Configuration Descriptor DMD)

这些文件以XML的形式描述了组成系统的波形组件和逻辑设备的identity、 capabilities、 properties、 inter-dependencies和location等信息。各个域描述文件间的总体关系如图 42所示。

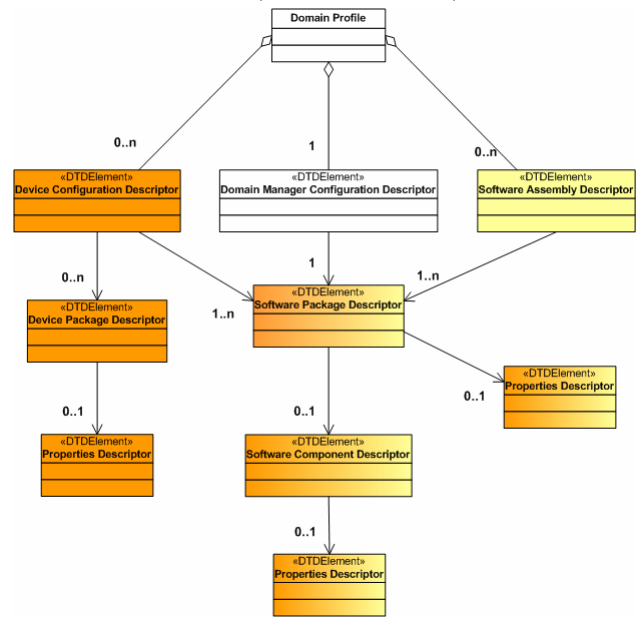


图 42 域描述文件关系

从各个域描述文件的关系可以看出,一个波形装配描述文件(SAD)实际上对应了一个波形应用,描述了该波形应用需要用到的各种组件、组件间的关系以及组件的连接信息。

* 软件包描述文件(SPD)——描述每个组件/设备的实现信息，包括入口函数、优先级、堆栈大小等
* 软件组件描述文件(SCD)——描述每个组件/设备提供和/或使用的接口信息
* 属性描述文件(PRF)——描述每个组件/设备的模型属性信息。
* 设备包描述文件(DPD)——描述逻辑设备对应的硬件设备的具体信息，如生成厂家、注册型号、设备类型等
* 设备配置描述文件(DCD)——描述了设备管理器管理的设备信息以及设备间的关系
* 域管理器配置描述文件(DMD)——描述了域管理器中服务以及设备管理器的获取信息
  1. Software Package Descriptor(SPD)

软件包描述文件(SPD)标识波形组件/逻辑设备的实现。软件包描述文件应具有.spd.xml扩展名。

软件包的一般信息，如名称、作者、实现代码信息、处理器和操作系统信息，都包含在软件包描述符文件中。

可以通过SPD文件获取组件/设备的PRF、SCD文件。



图 43 软件包描述文件(SPD)

* 1. Software Component Descriptor(SCD)

软件组件描述符(SCD)包含组件/设备提供和/或使用的接口信息。软件组件描述文件应该有.scd.xml扩展名。



图 44 接口信息

* 1. Software Assembly Descriptor(SAD)

软件装配描述文件(SAD)包含组成波形应用的组件的信息以及组件间的连接信息。应用工厂在创建波形应用时使用此信息。软件装配描述文件应具有.sad.xml扩展名。

可以通过SAD文件获取此波形应用包含组件的SPD文件。

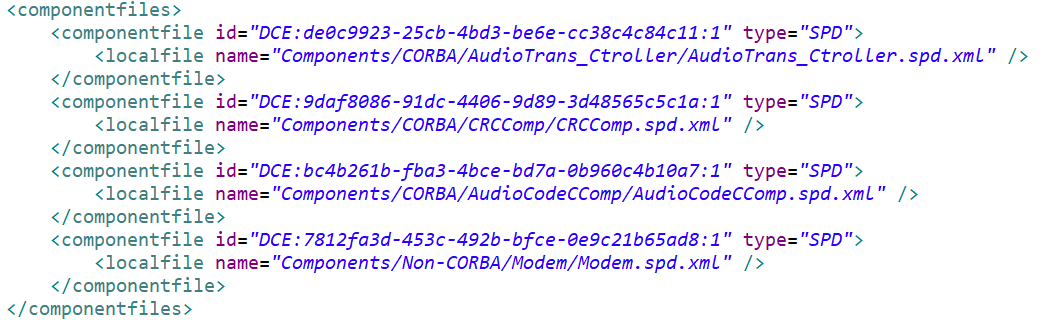


图 45组件的SPD文件路径



图 46波形应用包含的组件信息

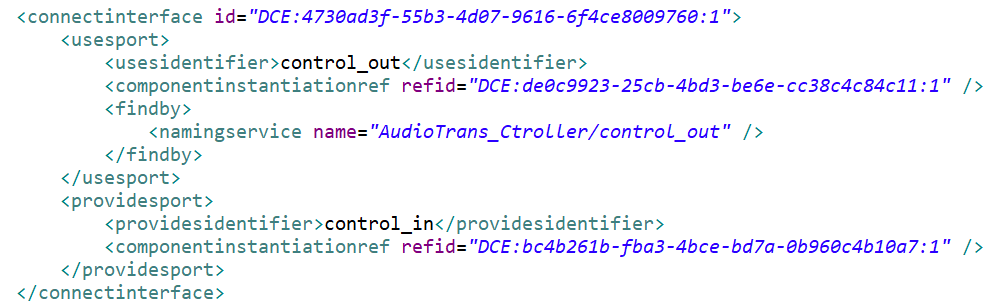


图 47 组件的连接信息

* 1. Properties Descriptor(PRF)

属性描述文件(PRF)包含适用于组件/设备的模型属性信息。属性文件应该有.prf.xml扩展名。

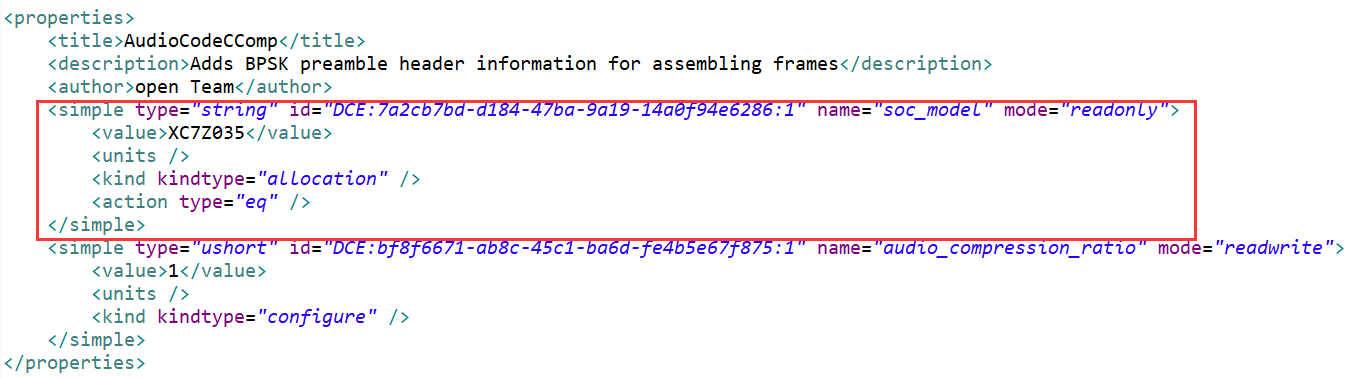


图 48 属性描述（PRF）文件

* 1. Device Package Descriptor(DPD)

设备包描述文件(DPD)描述逻辑设备对应的硬件设备的具体信息，如生成厂家、注册型号、设备类型等。设备包描述文件应具有.dpd.xml扩展名。

* 1. Device Configuration Descriptor(DCD)

设备配置描述文件(DCD)包含关于与设备管理器关联的设备的信息、如何找到域管理器以及设备的配置信息(日志、文件系统等)。设备配置描述符文件应具有.dcd.xml扩展名。

可以通过DCD文件获取逻辑设备的SPD文件。

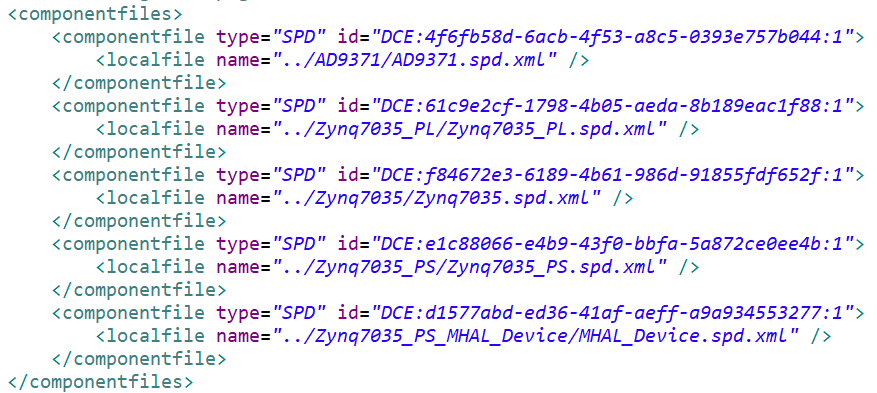


图 49 逻辑设备的SPD文件路径



图 50设备管理器管理的设备信息

* 1. Domain Manager Configuration Descriptor(DMD)

域管理器配置描述符(DMD)包含域管理器的配置信息，如何获取命名服务和设备管理器。DomainManager配置描述符文件应该具有.dmd.xml扩展名。

可以通过DMD文件获取域管理器的SPD文件，设备管理器的SPD文件和命名服务。

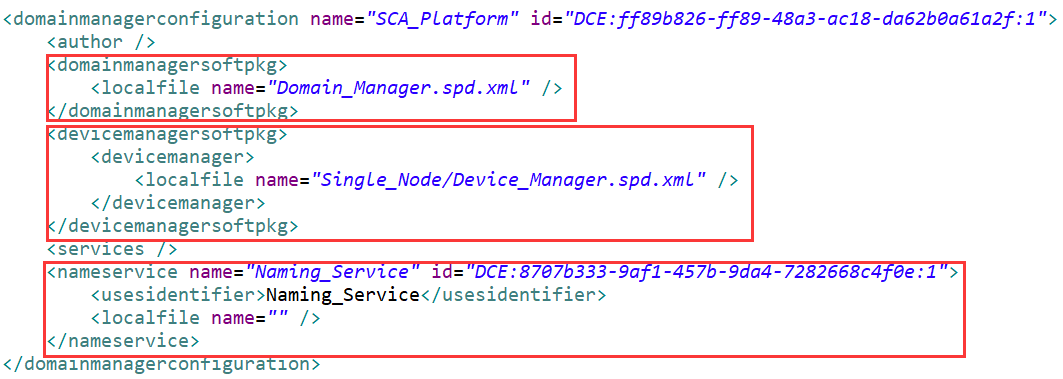


图 51 域管理器配置（DMD）文件

1. 波形应用及逻辑设备开发

本章基于jLabSDR实验平台，讲述开发者该如何开发波形应用与逻辑设备。

* 1. 开发场景

开发一个波形应用SampleApp，该波形应用的组件构成如下：

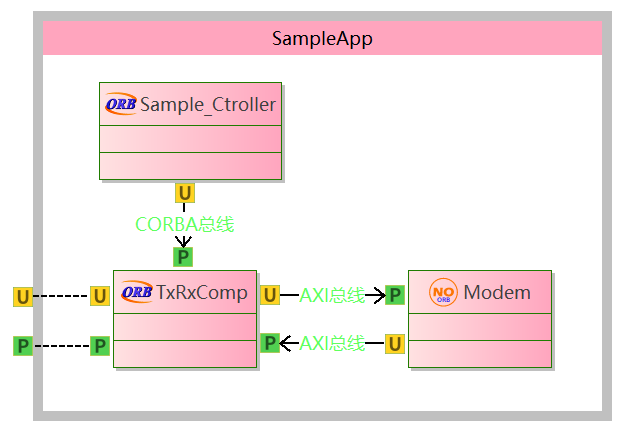


图 52 SampleApp波形应用

此波形实现了数据自发自收的功能，TxRxComp组件通过MHAL\_Device将数据发送给Modem组件，Modem组件收到数据后通过Tx天线发出，Rx天线收到数据后将再原路返回给TxRxComp组件，波形可以通过配置send\_times模型属性值来控制发送数据次数，数据传输流程如下图所示。



|  |  |
| --- | --- |
| **CORBA组件** | |
| Sample\_Ctroller | 波形控制器，控制SampleApp下的其他CORBA组件，如：TxComp组件 |
| TxRxComp | 接收或产生数据，将数据发送给MHAL\_Device设备 |
| **非CORBA组件** | |
| Modem | 运行在没有操作系统的处理器上的组件，比如FPGA |

基于jLabSDR平台抽象的逻辑设备如下：

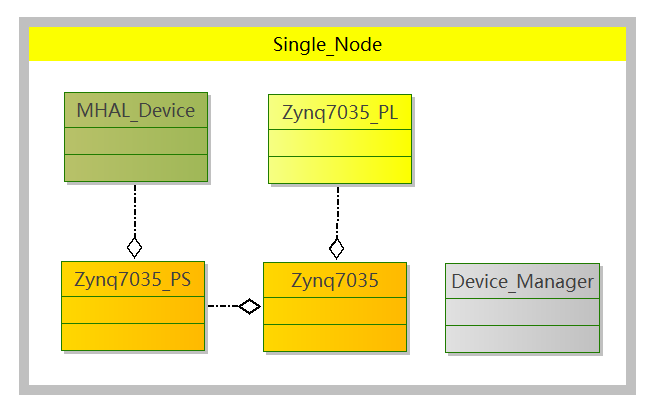


图 53 Single\_Node节点

在本场景中，Sample\_Ctroller、TxRxComp组件部署在Zynq7035\_PS设备，Modem组件部署在Zynq7035\_PL设备。

|  |  |
| --- | --- |
| **可执行设备** | |
| Zynq7035 | Zynq7035\_PS和Zynq7035\_PL的父设备,用于执行子设备 |
| Zynq7035\_PS | MHAL\_Device的父设备,用于执行子设备以及运行部署在Zynq7035\_PS上的组件 |
| **可加载设备** | |
| Zynq7035\_PL | 加载Modem组件 |
| **普通设备** | |
| MHAL\_Device | 封装硬件设备驱动接口，与硬件设备交互 |

* 1. 组件开发

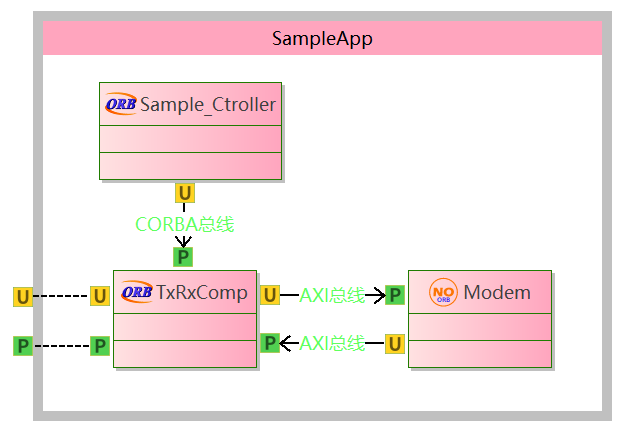


图 54 SampleApp波形应用

开发SampleApp波形应用，该波形应用的波形控制器为Sample\_Ctroller，控制TxRxComp组件的getPort()、start()、stop()、configure()、query()操作。

通过MHAL\_Device设备收发数据，需要在波形控制器initialize()接口中，获取MHAL\_Device的对象引用，在start()、stop()接口中，同时需要启动、停止MHAL\_Device。

TxRxComp组件的start()操作创建一个发送数据线程sendDataToMHAL，并且将TxRxComp组件的mhal\_axi\_data\_in输入端口信号插槽与recvDataFromMHAL()操作关联。

通过配置TxRxComp组件的send\_times的模型属性，可以configure、query发送数据的次数。

sendDataToMHAL()操作通过query()接口查询当前设置的send\_times的值，确定发送次数，调用mhal\_axi\_data\_out\_uport输出端口的pushPacket()接口发送数据。

天线接收数据后，通过MHAL\_Device发送给TxRxComp组件，TxRxComp组件的mhal\_axi\_data\_in端口的getData()接收数据。

红色部分为在模板代码基础上新添加的代码。

* + 1. Sample\_Ctroller
       1. initialize

通过initialize()接口获取MHAL\_Device的对象引用，MHAL\_Device的对象引用需要通过它的父设备Zynq7035\_PS得到。

void

**Sample\_Ctroller\_servant::initialize()**

throw (

CF::LifeCycle::InitializeError,

CORBA::SystemException)

{

DEBUG(6, [Sample\_Ctroller\_servant::initialize], " In initialize.")

control\_out\_uport = new StandardInterfaces\_i::Resource\_u(("OpenSCA\_Domain/" + m\_appName + "/Sample\_Ctroller/control\_out").c\_str());

m\_initConfig = false;

getConfigPropsFromPRF();

std::string contextName =

"OpenSCA\_Domain/Single\_Node/Device\_Manager/Zynq7035\_PS";

CORBA::Object\_var obj =

m\_orbWrap->get\_object\_from\_string(contextName.c\_str());

if (CORBA::is\_nil(obj)) {

DEBUG(1, [AudioTrans\_Ctroller\_servant::initialize], " obtain obj fail.")

return;

}

CF::ExecutableDevice\_var dev;

try {

dev = CF::ExecutableDevice::\_narrow(obj.in());

} catch (...) {

DEBUG(1, [AudioTrans\_Ctroller\_servant::initialize],

"\"CF::ExecutableDevice::\_narrow\" failed with Unknown Exception.")

}

try {

CF::DeviceSequence\_var deviceSeq = dev->compositeDevice()->devices();

for (CORBA::ULong i = 0; i < deviceSeq->length(); ++i) {

if (0 == strcmp("MHAL\_Device", deviceSeq[i]->label())) {

m\_mhalDev = CF::Device::\_duplicate(deviceSeq[i]);

break;

}

}

} catch (...) {

DEBUG(0, [AudioTrans\_Ctroller\_servant::initialize], " get MHAL device failed.")

}

if (CORBA::is\_nil(m\_mhalDev)) {

DEBUG(0, [AudioTrans\_Ctroller\_servant::initialize], " get child device failed.")

return;

}

m\_isStarted = false;

}

* + - 1. getPort

通过波形控制器的getPort()接口，可以获取到TxRxComp组件的端口对象。

#define TXRX\_COMP\_UPORT ("TxRxComp/user\_data\_out")

#define TXRX\_COMP\_PPORT ("TxRxComp/user\_data\_in")

CORBA::Object\_ptr

**Sample\_Ctroller\_servant::getPort**(

const char \* portName)

throw (

CORBA::SystemException,

CF::PortSupplier::UnknownPort)

{

DEBUG(6, [Sample\_Ctroller\_servant::getPort], "In getPort.")

CORBA::Object\_var \_port;

std::string portFullName = "OpenSCA\_Domain/" + m\_appName + "/" + portName;

if ((0 == strcmp(portName, TXRX\_COMP\_UPORT)) ||

(0 == strcmp(portName, TXRX\_COMP\_PPORT))) {

std::vector <CF::Resource\_var> comps;

comps = control\_out\_uport->getProvidesPorts();

\_port = comps[0]->getPort(portName);

return \_port.\_retn();

}

\_port = control\_out\_uport->getPort( portFullName.c\_str() );

if (!CORBA::is\_nil(\_port))

return \_port.\_retn();

//don't find any port named by protName

throw CF::PortSupplier::UnknownPort();

}

* + - 1. start

通过start()接口调用MHAL\_Device设备的start()和波形控制器控制的组件TxRxComp的start()。

void

**Sample\_Ctroller\_servant::start()**

throw (

CORBA::SystemException,

CF::Resource::StartError)

{

DEBUG(6, [Sample\_Ctroller\_servant::start], "In start.")

if(!m\_isStarted){

m\_mhalDev->start();

std::vector <CF::Resource\_var> comps;

comps = control\_out\_uport->getProvidesPorts();

DEBUG(7, [AudioTrans\_Ctroller\_servant::start], " comps.size(): " << comps.size());

for(int i = 0; i < comps.size(); i++) {

if(CORBA::is\_nil(comps[i])) {

DEBUG(7, [AudioTrans\_Ctroller\_servant::start], " get component failed. ");

break;

}

comps[i]->start();

}

m\_isStarted = true;

}

}

* + - 1. stop

通过stop()接口调用MHAL\_Device设备的stop()和波形控制器控制的组件TxRxComp的stop()。

void

**Sample\_Ctroller\_servant::stop()**

throw (

CORBA::SystemException,

CF::Resource::StopError)

{

DEBUG(6, [Sample\_Ctroller\_servant::stop], "In stop.")

if(m\_isStarted){

try {

m\_mhalDev->stop();

} catch(...) {

DEBUG(0, [AudioTrans\_Ctroller\_servant::stop], " stop error.")

}

std::vector <CF::Resource\_var> comps;

comps = control\_out\_uport->getProvidesPorts();

DEBUG(7, AudioTrans\_Ctroller\_servant::stop, " comps.size(): " << comps.size());

for(int i = 0; i < comps.size(); i++) {

if(CORBA::is\_nil(comps[i])) {

DEBUG(7, [AudioTrans\_Ctroller\_servant::stop], " get component failed. ");

break;

}

DEBUG(7, [AudioTrans\_Ctroller\_servant::stop], " call stop. ");

comps[i]->stop();

}

m\_isStarted = false;

}

}

* + - 1. releaseObject

通过releaseObject()接口调用停止stop()接口关闭线程、释放资源。

void

**Sample\_Ctroller\_servant:: releaseObject()**

throw (

CORBA::SystemException,

CF::LifeCycle::ReleaseError)

{

if (m\_isStarted) {

stop();

}

}

* + - 1. configure

通过configure()接口，设置TxRxComp组件的模型属性send\_times的值，先根据模型属性的id获取属性的对象，再通过组件的id，确定组件，调用组件的configure()设置模型属性。

此处需要考虑，安装波形应用时从每个波形组件的PRF中获取的所有模型属性的配置以及单个属性的设置：设置波形控制器的模型属性；设置波形控制器控制下的组件的属性。

#define SEND\_TIMES ("DCE:ed7c7593-226a-4564-b085-70a6aa4ba8b4:1")

#define TXRX\_COMP\_ID ("DCE:6d4c3067-04c8-48d4-abfb-8556f26b2e15:1")

void

**Sample\_Ctroller\_servant::configure(**

**const CF::Properties & configProperties)**

throw (

CORBA::SystemException,

CF::PropertySet::InvalidConfiguration,

CF::PropertySet::PartialConfiguration)

{

DEBUG(6, [Sample\_Ctroller\_servant::configure], "In configure.")

pthread\_mutex\_lock(&m\_attrMtx);

CF::Properties props = configProperties;

if(1 == props.length()){

if( 0 == strcmp( props[0].id, SEND\_TIMES)) {

std::vector <CF::Resource\_var> comps;

comps = control\_out\_uport->getProvidesPorts();

DEBUG(7, [Sample\_Ctroller\_servant::configure],

" comps.size(): " << comps.size());

for(int i = 0; i < comps.size(); i++){

if(CORBA::is\_nil(comps[i])){

DEBUG(0, [Sample\_Ctroller\_servant::configure],

" get component failed. ");

pthread\_mutex\_unlock(&m\_attrMtx);

break;

}

if( 0 == strcmp(comps[i]->identifier(), TXRX\_COMP\_ID)){

CF::Properties txRxProp;

txRxProp.length(1);

txRxProp[0] = props[0];

try{

comps[i]->configure(txRxProp);

} catch (...) {

DEBUG(0, [Sample\_Ctroller\_servant::configure],

"occur unkown exception when config." )

pthread\_mutex\_unlock(&m\_attrMtx);

throw CF::PropertySet::InvalidConfiguration();

}

pthread\_mutex\_unlock(&m\_attrMtx);

return;

}

}

}

}

PropertySet\_impl::configure(props);

pthread\_mutex\_unlock(&m\_attrMtx);

}

* + - 1. query

通过query()接口，查询TxRxComp组件的模型属性send\_times的值，先根据模型属性的id获取属性的对象，再通过组件的id，确定组件，调用组件的query()查询模型属性的值。

此处需要考虑，安装波形应用时从每个波形组件的PRF中获取的所有模型属性的查询以及单个属性的查询：查询波形控制器的模型属性的值；查询波形控制器控制下的组件的属性的值。

void

**Sample\_Ctroller\_servant::query(**

**CF::Properties & configProperties)**

throw (

CORBA::SystemException,

CF::UnknownProperties)

{

DEBUG(6, [Sample\_Ctroller\_servant::query], "In query.")

pthread\_mutex\_lock(&m\_attrMtx);

CF::Properties totalProperties;

totalProperties.length(0);

std::vector <CF::Resource\_var> comps;

comps = control\_out\_uport->getProvidesPorts();

CORBA::Short compLen = comps.size();

DEBUG(6, [Sample\_Ctroller\_servant::query], " compLen: " << compLen);

if (0 == configProperties.length()){

for(CORBA::Short i = 0; i < compLen; i++) {

if(CORBA::is\_nil(comps[i])) {

DEBUG(0, [Sample\_Ctroller\_servant::start],

" get component failed. ");

pthread\_mutex\_unlock(&m\_attrMtx);

break;

}

CF::Properties properties;

comps[i]->query(properties);

CORBA::Short len = properties.length();

DEBUG(6, [Sample\_Ctroller\_servant::query], " len: " << len);

CORBA::Short totalLen = totalProperties.length();

DEBUG(6, [Sample\_Ctroller\_servant::query],

" totalLen: " << totalLen);

totalProperties.length(totalLen + len);

for(CORBA::Short i = 0; i < len; ++i) {

totalProperties[totalLen + i] = properties[i];

}

}

PropertySet\_impl::query(configProperties);

CORBA::UShort len = configProperties.length();

CORBA::UShort oldLen = totalProperties.length();

totalProperties.length(oldLen + len);

for(CORBA::UShort i = 0; i < len; ++i) {

totalProperties[oldLen + i] = configProperties[i];

}

configProperties.length(0);

configProperties = totalProperties;

}

if(1 == configProperties.length()){

if (0 == strcmp(configProperties[0].id, SEND\_TIMES)) {

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

if(CORBA::is\_nil(comps[i])) {

DEBUG(0, [Sample\_Ctroller\_servant::start],

" get component failed. ")

pthread\_mutex\_unlock(&m\_attrMtx);

break;

}

if(0 == strcmp(comps[i]->identifier(), TXRX\_COMP\_ID)){

comps[i]->query(configProperties);

}

}

}

}

pthread\_mutex\_unlock(&m\_attrMtx);

}

* + 1. TxRxComp
       1. start

通过start()接口创建一个发送数据线程sendDataToMHAL，并且将TxRxComp组件的mhal\_axi\_data\_in输入端口信号插槽与recvDataFromMHAL()操作关联。

void

**TxRxComp\_servant::start()**

throw (

CORBA::SystemException,

CF::Resource::StartError)

{

DEBUG(6, [TxRxComp\_servant::start], "In start.")

if(!m\_isStarted){

m\_runFlag = true;

int sendTid = pthread\_create(&m\_sendThread, NULL, sendDataToMHAL, this);

if(0 != sendTid){

DEBUG(0, [TxRxComp\_servant::start],

" create sendDataToMHAL thread failed.")

m\_runFlag = false;

return;

}

mhal\_axi\_data\_in\_pport->connectSlot(

boost::bind(&TxRxComp\_servant::recvDataFromMHAL, this));

m\_isStarted = true;

}

}

* + - 1. 发送数据线程

sendDataToMHAL线程通过query()接口查询当前设置的send\_times的值，确定发送次数，调用mhal\_axi\_data\_out\_uport输出端口的pushPacket()接口发送数据。

**void \***

**sendDataToMHAL(void \* obj)**

{

TxRxComp\_servant \* txRxObj = (TxRxComp\_servant \*) obj;

CF::Properties props;

txRxObj->query(props);

int propLen = props.length();

CORBA::Short times;

for(CORBA::UShort i=0; i<propLen; ++i){

if(0 == strcmp(props[i].id, SEND\_TIMES)){

props[i].value >>= times;

}

}

DEBUG(0, [TxRxComp\_servant::configure], " times:" << times)

while(txRxObj->m\_runFlag && times){

JTRS::OctetSequence dataSeq;

std::string sendStr = "Hello JFounder!";

dataSeq.length(sendStr.length());

memcpy(dataSeq.get\_buffer(), sendStr.c\_str(), sendStr.length());

txRxObj->mhal\_axi\_data\_out\_uport->

pushPacket(txRxObj->mhal\_axi\_data\_out\_uport\_target\_LD, dataSeq);

std::cout << "send dataSeq.length:"<< dataSeq.length() << times << std::endl;

--times;

}

}

* + - 1. 接收数据接口

天线接收数据后，通过MHAL\_Device发送给TxRxComp组件，TxRxComp组件的recvDataFromMHAL()接口，通过mhal\_axi\_data\_in输入端口的getData()接收数据。

**void**

**TxRxComp\_servant::recvDataFromMHAL()**

{

CORBA::UShort targetLD;

JTRS::OctetSequence dataSeq;

mhal\_axi\_data\_in\_pport->getData(targetLD, dataSeq);

std::cout << "recv dataSeq.length:"<< dataSeq.length() << std::endl;

}

* + - 1. stop

通过stop()接口关闭线程，断开插槽。

void

**TxRxComp\_servant::stop()**

throw (

CORBA::SystemException,

CF::Resource::StopError)

{

DEBUG(6, [TxRxComp\_servant::stop], "In stop.")

if(m\_isStarted){

m\_runFlag = false;

sMsSleep(100000);

if(!pthread\_kill(m\_sendThread, 0)) {

DEBUG(3, [TxRxComp\_servant::stop],

" pthread kill start.")

int sendCancelID = pthread\_cancel(m\_sendThread);

DEBUG(2, [TxRxComp\_servant::stop],

" stop sendCancel ID:" << sendCancelID)

if(0 != sendCancelID) {

DEBUG(0, [TxRxComp\_servant::stop],

" sendDataToMHAL thread failed.")

return;

}

void \*status = NULL;

int ret = pthread\_join(m\_sendThread, &status);

m\_sendThread = 0;

}

mhal\_axi\_data\_in\_pport->disconnectSlot();

m\_isStarted = false;

}

}

* 1. 设备开发

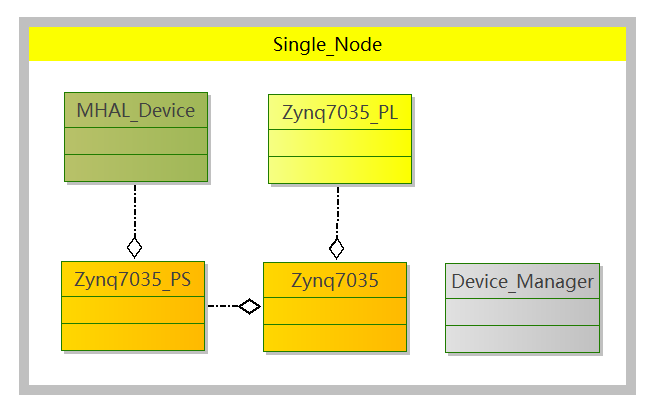


图 55 Single\_Node节点

将jLabSDR平台抽象成Single\_Node节点，包含设备如下：

1. 两个聚合设备：Zynq7035、Zynq7035\_PS，
2. 一个可加载设备：Zynq7035\_PL
3. 一个MHAL设备：MHAL\_Device
4. 一个设备管理器：Device\_Manager

聚合设备（简称父设备）可以启动其他设备（简称子设备）

运行框架时，设备管理器加载、启动父设备和MHAL设备，父设备启动子设备。

下面介绍设备各个接口的作用。

* + 1. Zynq7035

Zynq7035设备启动Zynq7035\_PS设备和Zynq7035\_PL设备，也就是执行设备的可执行的二进制文件，需在execute()接口中实现。

* + - 1. execute

CF::ExecutableDevice::ProcessID\_Type

CF::ExecutableDevice::ProcessID\_Type

U1\_Zynq7035\_servant::execute(

const char \* name,

const CF::Properties & options,

const CF::Properties & parameters)

throw (

CF::ExecutableDevice::ExecuteFail,

CF::InvalidFileName,

CF::ExecutableDevice::InvalidOptions,

CF::ExecutableDevice::InvalidParameters,

CF::ExecutableDevice::InvalidFunction,

CF::Device::InvalidState,

CORBA::SystemException)

{

DEBUG(5, [U1\_Zynq7035\_servant::execute], " In execute.")

CF::ExecutableDevice::ProcessID\_Type processId;

#ifdef \_\_SDS\_OS\_VXWORKS\_\_

processId = executeOnVxworks(name, options, parameters);

#elif defined \_\_SDS\_OS\_LINUX\_\_

processId = executeOnLinux(name, options, parameters);

#endif

return processId;

}

* + 1. Zynq7035\_PS

Zynq7035\_PS设备启动MHAL\_Device设备，也就是执行设备的可执行的二进制文件，需在execute()接口中实现。

同时它是Zynq7035的子设备，需要在server.cpp文件中添加代码，实现向父设备Zynq7035的设备序列中添加子设备。

* + - 1. execute

Zynq7035\_PS设备是MHAL\_Device的父设备,用于执行子设备以及运行部署在Zynq7035\_PS上的组件。在execute()接口实现。

CF::ExecutableDevice::ProcessID\_Type

CF::ExecutableDevice::ProcessID\_Type

U1\_Zynq7035\_servant::execute(

const char \* name,

const CF::Properties & options,

const CF::Properties & parameters)

throw (

CF::ExecutableDevice::ExecuteFail,

CF::InvalidFileName,

CF::ExecutableDevice::InvalidOptions,

CF::ExecutableDevice::InvalidParameters,

CF::ExecutableDevice::InvalidFunction,

CF::Device::InvalidState,

CORBA::SystemException)

{

DEBUG(5, [U1\_Zynq7035\_servant::execute], " In execute.")

CF::ExecutableDevice::ProcessID\_Type processId;

#ifdef \_\_SDS\_OS\_VXWORKS\_\_

processId = executeOnVxworks(name, options, parameters);

#elif defined \_\_SDS\_OS\_LINUX\_\_

processId = executeOnLinux(name, options, parameters);

#endif

return processId;

}

* + - 1. 添加到父设备

作为Zynq7035的子设备需向父设备进行addDevice()操作。子设备的对象引用只能通过父设备获取。注册在Zynq7035\_PS\_server.cpp主函数中实现，代码示例如下：

**#ifdef** \_\_SDS\_OS\_VXWORKS\_\_

**extern** "C"

**int** U1\_Zynq7035\_PS\_Main(**int** argc, **char** \* argv[])

**#elif** defined \_\_SDS\_OS\_LINUX\_\_

**int** main(**int** argc, **char** \* argv[])

**#endif**

{

**...**

///parse input parameter

**for** (**unsigned** **int** numArg=0; numArg<argc; numArg++) {

**...**

///parse devicemanager name

**if** (strcmp(argv[numArg], "-DEVMGRNAME") == 0) {

**if** (checkInParam(argc, numArg) == **false**)

exit(EXIT\_FAILURE);

devmgrName = argv[numArg+1];

}

//parser composite device ior

**if** (strcmp(argv[numArg], "-COMPOSITE\_DEVICE\_IOR") == 0){

**if** (checkInParam(argc, numArg) == **false**)

exit(EXIT\_FAILURE);

compositeDevIor = argv[numArg+1];

}

}

CF::ExecutableDevice\_var devServant\_var =

devServant->POA\_CF::ExecutableDevice::\_this();

...

**if** (orbWrap->bind\_object\_to\_string(

devServant\_var.in(), cosName.c\_str()) == **false**) {

DEBUG(0, [U1\_Zynq7035\_PS\_Main], " bind device " << devName

<< " objref with name into nameservice failed.")

**delete** devServant;

**delete** orbWrap;

**return** -1;

}

//将该设备添加到父设备

**try** {

CORBA::Object\_var obj =

orbWrap->ior\_to\_object(compositeDevIor.c\_str());

CF::Device\_var compositeDev = CF::Device::\_nil();

compositeDev = CF::Device::\_narrow(obj.in());

compositeDev->compositeDevice()->addDevice(devServant\_var.in());

} **catch**(...) {

...

}

...

**return** 0;

}

* + 1. Zynq7035\_PL

Zynq7035\_PL设备加载Modem组件，也就是加载非CORBA组件的bit文件到FPGA设备中。

同时它是Zynq7035的子设备，需要在server.cpp文件中添加代码，实现向父设备Zynq7035的设备序列中添加子设备。

* + - 1. load

Zynq7035\_PL设备加载modem组件，在load()接口中实现。

**void**

U1\_Zynq7035\_PL\_servant::**load**(

CF::FileSystem\_ptr fs,

**const** **char** \* fileName,

CF::LoadableDevice::LoadType loadKind)

**throw** (

CF::LoadableDevice::LoadFail,

CF::InvalidFileName,

CF::LoadableDevice::InvalidLoadKind,

CF::Device::InvalidState,

CORBA::SystemException)

{

//加载modem组件

}

* + - 1. unload

Zynq7035\_PL设备卸载modem组件，在unload()接口中实现。

**void**

U1\_Zynq7035\_PL\_servant::**unload**(

**const** **char** \* fileName)

**throw** (

CF::InvalidFileName,

CF::Device::InvalidState,

CORBA::SystemException)

{

//卸载modem组件

}

* + - 1. 添加到父设备

作为Zynq7035的子设备需向父设备进行addDevice()操作。子设备的对象引用只能通过父设备获取。注册在Zynq7035\_PL\_server.cpp主函数中实现，代码示例如下。

**#ifdef \_\_SDS\_OS\_VXWORKS\_\_**

**extern "C"**

**int Zynq7035\_PL\_Main(int argc, char \* argv[])**

**#elif defined \_\_SDS\_OS\_LINUX\_\_**

**int main(int argc, char \* argv[])**

**#endif**

**{**

**...**

///parse input parameter

**for** (**unsigned** **int** numArg=0; numArg<argc; numArg++) {

**...**

**if** (strcmp(argv[numArg], "-DEVMGRNAME") == 0) {

**if** (checkInParam(argc, numArg) == **false**)

exit(EXIT\_FAILURE);

devmgrName = argv[numArg+1];

}

//parser composite device ior

**if** (strcmp(argv[numArg], "-COMPOSITE\_DEVICE\_IOR") == 0){

**if** (checkInParam(argc, numArg) == **false**)

exit(EXIT\_FAILURE);

compositeDevIor = argv[numArg+1];

}

}

CF::LoadableDevice\_var devServant\_var =

devServant->POA\_CF::LoadableDevice::\_this();

...

**if** (orbWrap->bind\_object\_to\_string(

devServant\_var.in(), cosName.c\_str()) == **false**) {

**delete** devServant;

**delete** orbWrap;

**return** -1;

}

//将该设备添加到父设备

try {

CORBA::Object\_var obj =

orbWrap->ior\_to\_object(compositeDevIor.c\_str());

CF::Device\_var compositeDev = CF::Device::\_nil();

compositeDev = CF::Device::\_narrow(obj.in());

compositeDev->compositeDevice()->addDevice(devServant\_var.in());

} catch(...) {

...

}

...

**return** 0;

}

1. 术语表
   1. 缩写

|  |  |
| --- | --- |
| **Abbreviation** | **Definition** |
| AEP | Application Environment Profile |
| API | Application Program Interface |
| CF | Core Framework |
| CORBA | Common Object Request Broker Architecture |
| DCD | Device Configuration Descriptor |
| DMD | DomainManager Configuration Descriptor |
| DPD | Device Package Descriptor |
| FPGA | Field Programmable Gate Array |
| HCI | Human-Computer Interface |
| ID | Identification, Identifier |
| IOR | Interoperable Object Reference |
| IDL | Interface Definition Language |
| OE | Operating Environment |
| ORB | Object Request Broker |
| OS | Operating System |
| PRF | Properties Descriptor |
| SAD | Software Assembly Descriptor |
| SCA | Software Communications Architecture |
| SCD | Software Component Descriptor |
| SPD | Software Package Descriptor |
| UML | Unified Modeling Language |
| UUID | Universally Unique Identifier |
| XML | eXtensible Markup Language |

* 1. 名词解释
     1. Application

SCA规范定义了一个Application接口类，它为实例化应用程序的控制、配置和状态提供接口。 Application控制其组件并建立与其他应用程序的连接。

* + 1. ApplicationFactory

ApplicationFactory接口的实例化用于创建应用程序的实例。域管理器创建应用工厂。

* + 1. API

API（Application Program Interface）是包含在一组相关接口中的操作和属性的定义，这些接口提供一系列的功能。

* + 1. assemblycontroller

assemblycontroller为波形控制器，控制波形应用的CORBA组件。波形装配描述(Software Assembly Descriptor SAD)文件中的assemblycontroller元素指示作为波形控制器的组件。

* + 1. Client客户端

调用另一个组件的操作的组件。

* + 1. Component

符合SCA规范并实现一组接口的软件模块或元素。

* + 1. CORBA Component

实现一个或多个CORBA接口的软件组件。 CORBA组件由软件组件描述（SCD）文件描述。

* + 1. Core Application

一个或多个框架控制或服务接口的软件实现。

* + 1. CF核心框架

CF(Core Framework)是SCA2.2.2规范中定义的开放应用层接口和服务的集合。 CF旨在提供为软件应用程序设计者提供底层软件和硬件层抽象所需的基本（“核心”）接口集。

* + 1. Device

1. 硬件设备指的是物理硬件元件（通常是执行功能或功能集的模块）。
2. SCA定义了Device接口类。 此接口是硬件设备的抽象，用于定义该设备的功能、属性和接口。
   * 1. Device Configuration Description (DCD)

DCD描述了DeviceManager中配置哪些组件与服务，子类Device如何找到DomainManager，以及各个Device应该如何配置连接等信息。其扩展名为“dcd.xml”。

* + 1. Device Package Descriptor (DPD)

用以识别设备类,同时还包含一些为设备类所定义的特殊属性、序列号等,其主要作用是为操作和维护人员提供对硬件设备信息的配置和修改。其扩展名为“dpd.xml”。

* + 1. Device Profile

设备配置文件是域配置文件中的一组XML文件，它完整地描述了硬件设备。设备配置文件包含设备包描述(DPD)文件，设备配置描述(DCD)文件和可选的属性(PRF)文件。 有关与此硬件设备关联的软件的信息，请参阅相关的软件配置文件。

* + 1. Domain域

域定义了一组在单个域管理器控制下的硬件设备和应用程序。

* + 1. Domain Manager域管理器

DomainManager接口的实现，域管理器管理整套可用的硬件设备和应用程序。它负责设置和关闭应用程序以及将资源、设备和非CORBA组件分配给硬件设备。

* + 1. Domain Profile

SCA2.2.2规范要求为可移植的组件提供通用的信息,这种信息叫做域描述文件(Domain Profile)。这些文件以XML的形式描述了组成系统的硬件设备和软件组件的identity、 capabilities、 properties、 inter-dependencies和location等信息。

* + 1. Event Service事件服务

事件服务是一种CORBA服务，可以解耦组件之间的通信。 CORBA事件服务为组件定义了两个角色：生产者角色（生成事件数据）和消费者角色（流程事件数据）。 通过发出标准CORBA请求，在生产者和消费者之间传递事件数据。

* + 1. Event Channel事件通道

事件通道是一个干预组件，允许多个生产者异步与多个消费者进行通信。 事件通道既是消费者又是事件的供应者。 事件通道是被更改的组件（生产者）和有兴趣了解更改的组件（消费者）之间的中介。 提供更改通知的事件通道可以是通用的，众所周知的组件（传入和传出域管理事件通道），它们作为域范围框架或特定任务组件的一部分（例如，在应用程序部署时创建的临时事件通道）。

* + 1. Parent Device

父设备使用AggregateDevice接口，由一个或多个子设备组成。 父设备及其子设备密切相关并具有相同的生命周期（即删除父设备会删除所有子设备）。

* + 1. Port端口

端口接口标识源/消费者（Provides Port）或数据/命令的接收器/生成器（Uses Port）。

* + 1. Properties Descriptor(PRF)

属性文件包含适用于软件包或设备包的属性集，其扩展名为“性文件包含适用于软件。一个属性文件包含组件的属性有：配置、 测试、 执行和分配类型（ configuration, test, execute, and allocation types）。

* + 1. Profile Descriptor

配置文件描述符是域配置文件的一个元素，它包含软件包描述文件，软件程序集描述文件，DomainManager配置描述文件或设备配置描述文件的绝对路径名。

* + 1. Properties Descriptor

属性描述符是域配置文件的一个元素，包含有关适用于软件包或设备包的属性的信息，例如配置、测试、执行和分配类型。

* + 1. Property

SCA属性是包含特定类型值的变量。Configuration Properties是PropertySet接口的配置和查询操作的参数。分配属性定义资源所需的设备功能。

* + 1. Resource

基于SCA规范的Resource基础应用程序接口实现的软件组件，是所有应用程序中与SCA一致的可见组件必须实现的资源接口。

* + 1. Software Assembly Descriptor (SAD)

波形装配描述 (SAD)文件包含了组成Applicaton的所有组件的信息。 ApplicationFactory在创建一个Application时要使用这些信息。 波形装配描述(SAD)文件的扩展名为“sad.xml”。

* + 1. Software Component Descriptor (SCD)

软件组件描述文件(SCD)描述了SCA软件组件(如Resource、ResourceFactory、Device等)的信息,以及组件所提供的或使用的接口(Port)信息。其扩展名为“.scd.xml”。

* + 1. Software Package Descriptor (SPD)

一个软件包描述文件标识了一个软件组件的实施。软件包描述文件的扩展名为“spd.xml”。软件包描述文件包含了软件名称、作者、知识产权声明文件、实施的代码信息和硬件软件依赖性等信息。其主要作用是在组件启动与配置时指定应该加载的组件以及如何找到组件的属性文件。其中，组件的属性由三部分组成,依次为与实施相关的属性、描述软件包描述(SPD)文件本身的属性和软件组件描述(SCD)文件中定义的属性。

* + 1. Software Profile

软件配置文件是一组域配置文件，属于特定的SCA组件。 CORBA组件的所有软件配置文件都包括软件包描述符和软件组件描述文件（以及可选的属性描述文件），但配置文件中包含的其他文件取决于相关的SCA组件。应用程序的配置文件将包括软件程序集描述文件，域管理器配置文件包括DomainManager配置描述文件，设备管理器的配置文件包括设备配置描述文件。

* + 1. Waveform

波形是通过空中传输的信息的变换集以及相应的变换集将接收的信号转换回其信息内容。

* + 1. Waveform Application

波形应用程序是软件元素（模块或组件）的集合，其执行为特定波形定义的变换。这可能包括（SCA）应用程序组件以及核心应用程序、服务和（SCA）设备。