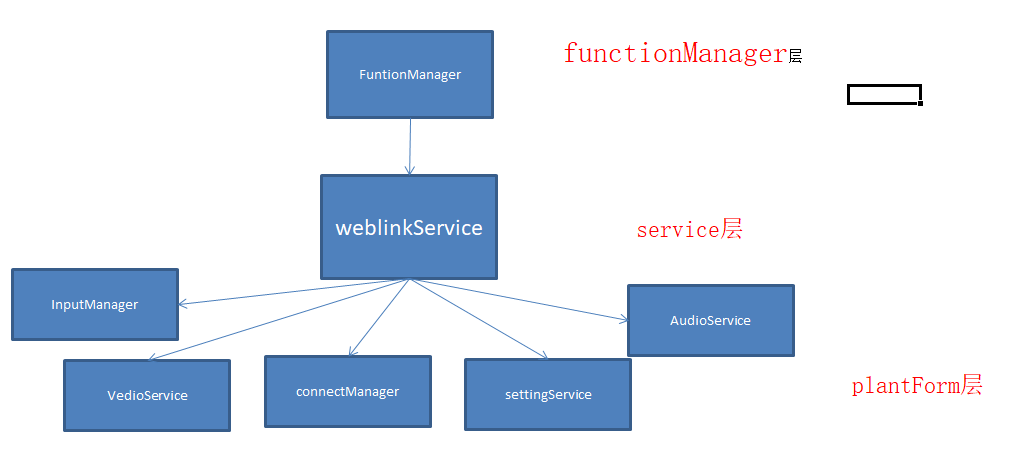
### Weblink与其他模块的关系



#### 1 weblink和vedioService模块

在webLink使用vedioService服务。这时，需要创建一个类用来继承vedioService的proxy类。



#### 2 weblink和AudioService模块

#### 3 weblink和vedioService模块

#### 4 weblink和SettingService模块

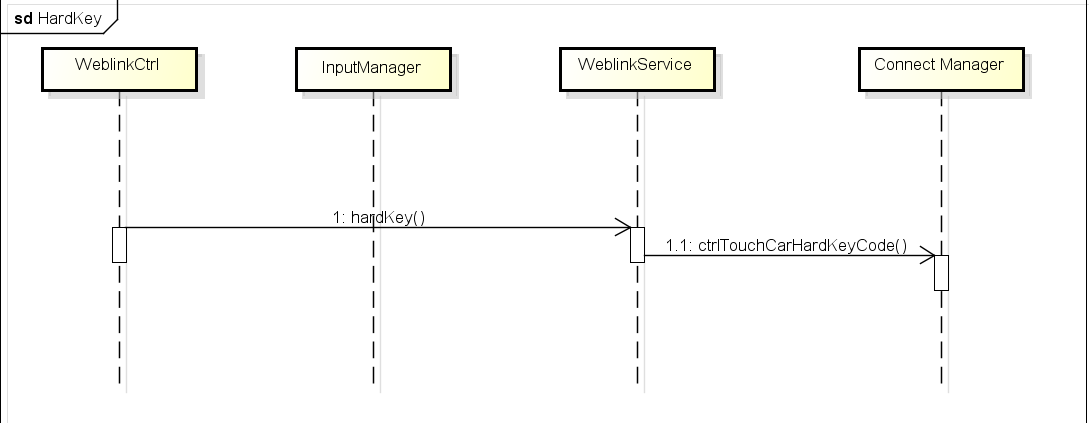
#### 5 weblink和ConnectManager模块

#### 6 weblink和functionManager模块

### WebLink服务核心方法实现

#### 1 SendHardkeyEvent

##### 时序图



##### 具体实现

发送实体按键事件，当通过weblink操作车机时，按下上下曲实体按钮，触发事件。

在proxy中发送的sendHardkeyEvent请求，在WebLinkStubBase中被接收，然后转到WebLinkServiceManager中被处理。

|  |
| --- |
| void WeblinkServiceManager::sendHardkeyEvent(const WLHardKey& key, const WLHardKeyStatus& status)  WEBLINKLOGD("%s key = %d, status = %d", \_\_FUNC\_\_, key, status);  //判断传过来的硬件的状态是否为放开，同时判断客户端是否连接成功。  if((status == WLHardKeyStatus\_RELEASE) && (m\_client != NULL) && (m\_client->IsConnected()) && gstreamerFlag)  {  // if(m\_isClientBusy)  // {  // //when show WaitIndicator, allow hard key event.  // WEBLINKLOGD("%s m\_isClientBusy & event = %d", \_\_FUNC\_\_, event);  // return;  // }  DTLOG\_INFO(0x70302A, "sendHardkey -- key = %d", key);  //获取与客户端连接对象  IWebLinkConnection\* conn = m\_client->GetConnection();  if(conn)  {  switch(key)  {  case WLHardKey\_HOME:  conn->SendCommand(new CBrowserCommand(CBrowserCommand::eHome));  break;  case WLHardKey\_PREV:  conn->SendCommand(new CBrowserCommand(CBrowserCommand::eBack));  break;  default:  WEBLINKLOGD("%s key = %d not recognized.", \_\_FUNC\_\_, key);  break;  }  }  }  else  {  DTLOG\_PTRASSERT(m\_client);  }  } |

在webLink的开发商提供了供我们使用的和webLink客户端交互的一个开发包。其中在这个方法中， m\_client是他们提供的一个客户端的对象。IWebLinkConnection\* conn也是webLink中提供的webLink连接对象。通过连接 对象的SendCommand方法，就可以实现服务端和客户端的通信。在这里手机端的webLink是服务端，车机中是客户端。

##### 涉及模块

FunctionManager

webLinkservice

external中的webLink的第三方库

##### 常用方法总结

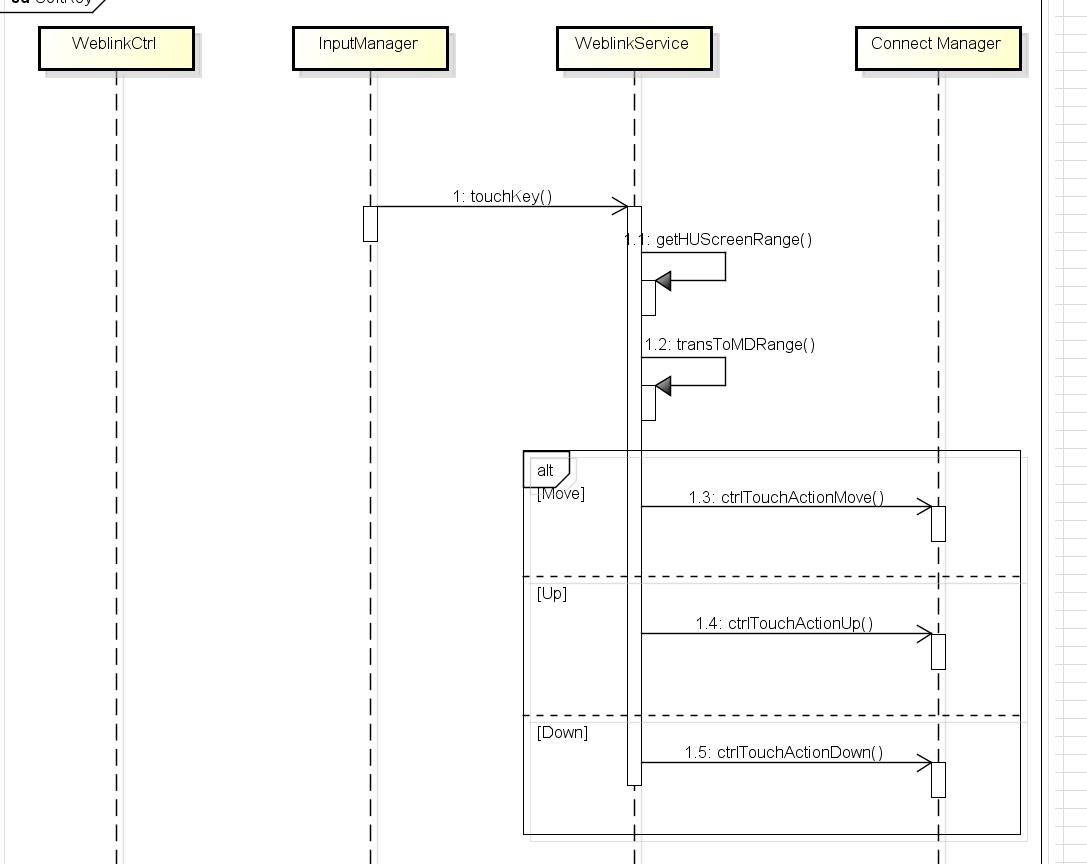
在webLink中通过第三方提供的客户端类获取连接对像。在swich case 中等匹配结果，通过SendCommand方法发送命令。

virtual bool SendCommand(CCommand\* command, bool waitForCompletion = false)

|  |
| --- |
| IWebLinkConnection\* conn = m\_client->GetConnection();  if(conn)  {  switch(key)  {  case WLHardKey\_HOME:  conn->SendCommand(new CBrowserCommand(CBrowserCommand::eHome));  break;  case WLHardKey\_PREV:  conn->SendCommand(new CBrowserCommand(CBrowserCommand::eBack));  break;  default:  WEBLINKLOGD("%s key = %d not recognized.", \_\_FUNC\_\_, key);  break;  }  } |

#### 2 SendTouchEvent

##### 时序图



##### 具体实现

在proxy中将

WLTouchEvent 触摸事件

WLTouchPoints\_SpVec 触摸指针的verctor列表

Count 数量传到stub端。

在stub端通过case匹配读取parcel中值，调用了stubService中的sendTouchEvent方法。

但是在这个方法中，并没有做实际的逻辑处理，不知道为啥。代码如下

|  |
| --- |
| android::status\_t WeblinkServiceStub::sendTouchEvent(SenderId id, const WLTouchEvent& type, const WLTouchPoints\_SpVec& points, const uint32\_t& count)  {  WEBLINKLOGD("[%s]",\_\_FUNC\_\_);  return 0;  } |

##### 涉及模块

FunctionManager

WebLinkService

##### 常用方法总结

对于从functionManagert中传递过来的参数中，有列表类型时的处理方式

列表参数：const WLTouchPoints\_SpVec& points

|  |
| --- |
| {  \_data.writeInt32(points.size());  WLTouchPoints\_SpVec::const\_iterator it = points.begin();  for (; it != points.end(); ++it) {  // writeToParcel()在WLTouchPoints方法中定义的将类中的多个属性加入到parcle的方法。  (\*it)->writeToParcel(\_data);  }  } |

在WLTouchPoints类中，将其各个属性加入到parcel中

|  |
| --- |
| void writeToParcel(android::Parcel& \_data)  {  \_data.writeInt32(m\_id);  \_data.writeInt32(m\_posX);  \_data.writeInt32(m\_posY);  \_data.writeInt32(static\_cast<WLTouchPointState>(m\_state));  \_data.writeFloat(m\_pressure);  } |

从proxy端将请求发送到stub时，在stub中接收含有列表类型的Parcel的方式。

|  |
| --- |
| {  int \_len = data.readInt32();  points.clear();  for (int i = 0; i < \_len; i++) {  WLTouchPoints\_Sp \_tmp(new WLTouchPoints);  //通过在 WLTouchPoints中定义的readFromParcel方法。将parcel中的数据读入到tmp中  \_tmp->readFromParcel(data);  points.push\_back(\_tmp);  }  } |

#### 3 ChangeSourceStatus

##### 具体实现

该方法，携带着需要改变的资源的状态，在proxy中发出请求，在stubBase中接收该请求后调用stub中的changeSourceStatus方法进行处理，而在stub的changeSourceStatus方法中，再次调用了WebLinkserviceManager中的changeSourceStatus方法。

WLSourceStatus 资源状态

WLSourceStatus\_Off= 0,

WLSourceStatus\_On= 1, ///< source off

WLSourceStatus\_DKStopOn= 2, ///< source on

WLSourceStatus\_DKStopOff= 3, ///< DK Stop on

WLSourceStatus\_Invalid= 4, ///< DK Stop off

|  |
| --- |
| android::status\_t WeblinkServiceManager::changeSourceStatus(SenderId id, const WLSourceStatus& status)  {  WEBLINKLOGD("%s status = %d", \_\_FUNC\_\_, status);  DTLOG\_INFO(0x703013, "status = %d", status);  //实例化一个WLServiceMsg对象。将MN\_EVT\_SOURCE\_STATUS\_CMD实例化WLServiceMsg，为WLServiceMsg中的fid变量  android::sp<WLServiceMsg> msg = new WLServiceMsg(MN\_EVT\_SOURCE\_STATUS\_CMD);  设置wlmsg的sourceSts属性  msg->setSourceSts(status);  //将该消息post到消息队列中  if (msg != NULL ) {  //动态类型转换  mThread.postMessage(reinterpret\_cast<Message\*>(msg.get()), 0);  }  else {  DTLOG\_PTRASSERT(msg);  }  return 0;  } |
|  |

这里调用postMessage后，将数据放在了消息队列中，在WLManagerHandler 类中有一个专门处理消息队列中的方法，onReceiveMessage，在该方法中，通过case匹配方式，处理消息对列中的message。

|  |
| --- |
| …….  //从消息队列中取出消息  WLServiceMsg\* req = static\_cast<WLServiceMsg\*>(msg.get());  switch (req->funcId())  …..  //case匹配  case MN\_EVT\_SOURCE\_STATUS\_CMD:  {  int sts = req->getSourceSts();  WEBLINKLOGD("\*\*\*\*\*%s\*\*\*\*\*, MN\_EVT\_SOURCE\_STATUS\_CMD sts = %d", \_\_func\_\_, sts);  if(sts <= WLSourceStatus\_Invalid /\* max \*/)  {  //调用该类中的中的changeSourceStatusPro方法处理逻辑  mOwner->changeSourceStatusPro((WLSourceStatus)sts);  }  break;  } |

在onReceiveMessage方法中，又调用了changeSourceStatusPro方法，处理最终的逻辑

|  |
| --- |
| //最终的处理changeSourceStatus逻辑的方法  void WeblinkServiceManager::changeSourceStatusPro(const WLSourceStatus &status) /\* \_\_0x703046\_DTFUNCID\_\_ \*/  {  WEBLINKLOGD("%s status = %d", \_\_FUNC\_\_, status);  DTLOG\_INFO(0x703046, "SourceStatus -- status = %d", status);  // error status after initializeed.  if(m\_sourceStatus == WLSourceStatus\_Invalid && status != WLSourceStatus\_On )  {  WEBLINKLOGD("%s m\_sourceStatus == Invalid & error status = %d", \_\_FUNC\_\_, status);  if(m\_replier != NULL)  {  WEBLINKLOGD( "notifySourceStatusChanged status = %d", status);  //通知调用方  m\_replier->notifySourceStatusChanged(mId, status, WLNotiFlag\_Completed);  }  else  {  DTLOG\_PTRASSERT(m\_replier);  }  return;  }  //Source on & Video on.  //打开资源，同时打开video  if(status == WLSourceStatus\_On && m\_sourceStatus != WLSourceStatus\_On)  {  m\_sourceStatus = WLSourceStatus\_On;  m\_videoChannelStatus = true;  // when source on, disconnect BTA and Miracast.  onVideoControlCommandPro(true);  setAudioChannelStatus(WLAudioChannel\_Audio, WLAudioStatus\_On);  if(m\_needResetUsb && isAppleDevice())  {  WEBLINKLOGD("%s TryRestUsb", \_\_FUNC\_\_);  GoniAOAManager::instance()->TryRestUsb();  }  else  {  WEBLINKLOGD("%s m\_needResetUsb = %d", \_\_FUNC\_\_, m\_needResetUsb);  if(m\_needLaunch)  {  checkAppLaunch();  }  EapConnectionResume();  CheckConnection();  }  // changeVideoStatus(id, WLVideo\_On);  GoniAOAManager::instance()->SetAppSourceStatus(true);  }  //Source off & Video off.  //资源关闭，video关闭  else if (status == WLSourceStatus\_Off && m\_sourceStatus != WLSourceStatus\_Off)  {  m\_sourceStatus = WLSourceStatus\_Off;  m\_videoChannelStatus = false;  setAudioChannelStatus(WLAudioChannel\_Audio, WLAudioStatus\_Off);  EapConnectionPause();  m\_establishProcess = false;  GoniAOAManager::instance()->SetAppSourceStatus(false);  }  else if(status == WLSourceStatus\_DKStopOn && m\_sourceStatus != WLSourceStatus\_DKStopOn)  {  // do nothing.  // tryPauseVideoPlay();  }  else if(status == WLSourceStatus\_DKStopOff && m\_sourceStatus != WLSourceStatus\_DKStopOff)  {  // do nothing.  // tryResumeVideoPlay();  }  else  {  DTLOG\_ERR(0x703046, "SourceStatus -- error status = %d", status);  WEBLINKLOGD("%s error status = %d", \_\_FUNC\_\_, status);  }  if(m\_replier != NULL)  {  WEBLINKLOGD( "notifySourceStatusChanged status = %d", status);  m\_replier->notifySourceStatusChanged(mId, status, WLNotiFlag\_Completed);  }  else  {  DTLOG\_PTRASSERT(m\_replier);  }  } |
|  |

##### 涉及模块

functionManager

WebLinkService

##### 常用方法总结

暂无

#### 4 changeVideoStatus

##### 具体实现

改变视频播放的状态—开关。。。

该方法用于改变video的状态

WLVideotatus ：vedio状态

WLVideo\_Off= 0,

WLVideo\_On= 1, ///< video off

WLVideo\_Invalid= 2, ///< video on

在proxy中发送消息，在stubBase中调用其中的changeVideoStatus方法，最终调用了WebLinkServiceManager中的方法changeVideoStatus方法，在这个方法中，当传入的状态值为off时，将消息post到消息对列中，在OnRecessive中处理。为on时，返回0

|  |
| --- |
| android::status\_t WeblinkServiceManager::changeVideoStatus(SenderId id, const WLVideotatus& status) /\* \_\_0x703012\_DTFUNCID\_\_ \*/  {  WEBLINKLOGD("%s status = %d", \_\_FUNC\_\_, status);  DTLOG\_INFO(0x703012, "status = %d", status);  if(id != -1) { mId = id; }  //当status为Off时才将消息放在消息队列中  if(status == WLVideo\_Off)  {  android::sp<WLServiceMsg>msg=new WLServiceMsg(MN\_EVT\_VIDEO\_CH\_STATUS\_CMD);  msg->setVideoChSts(false);  if (msg != NULL ) {  mThread.postMessage(reinterpret\_cast<Message\*>(msg.get()), 0);  }  else {  DTLOG\_PTRASSERT(msg);  }  }  return 0;  } |

在OnRecessive方法的case中处理消息，在其中调用changeVideoChannelStatusPro方法，处理逻辑

|  |
| --- |
| case MN\_EVT\_VIDEO\_CH\_STATUS\_CMD:  {  bool sts = req->getVideoChSts();  WEBLINKLOGD("\*\*\*\*\*%s\*\*\*\*\*, MN\_EVT\_VIDEO\_CH\_STATUS\_CMD sts = %d", \_\_func\_\_, sts);  mOwner->changeVideoChannelStatusPro(sts);  break;  } |

在该类中的changeVideoChannelStatusPro方法。主要是对传入的isReqVideOn参数为为false时的逻辑处理，这里又调用了changeVideoStatus方法。

|  |
| --- |
| void WeblinkServiceManager::changeVideoChannelStatusPro(bool isReqVideOn) /\* \_\_0x703045\_DTFUNCID\_\_ \*/  {  WEBLINKLOGD("%s isReqVideOn = %d", \_\_FUNC\_\_, isReqVideOn);  DTLOG\_INFO(0x703045, "ideoChannelStatus -- isReqVideOn = %d", isReqVideOn);  //当isReqVideOn为false时，执行下面的逻辑，但是当为true，直接方法结束。  if(!isReqVideOn)  {  m\_videoChannelStatus = false;  //在这里又调用了changeVideoStatus方法，将false作为参数传入  changeVideoStatus(false);  }  } |

在该类中的changeVideoStatus方法，传入isReqVideOn参数。这里有个逻辑的问题，在关闭video的时候，audio也要关闭。所以当isReqVideOn传入false时，会先暂停audio,再调用VideoStatusPro方法，传入videostatus停止的状态，停止video。

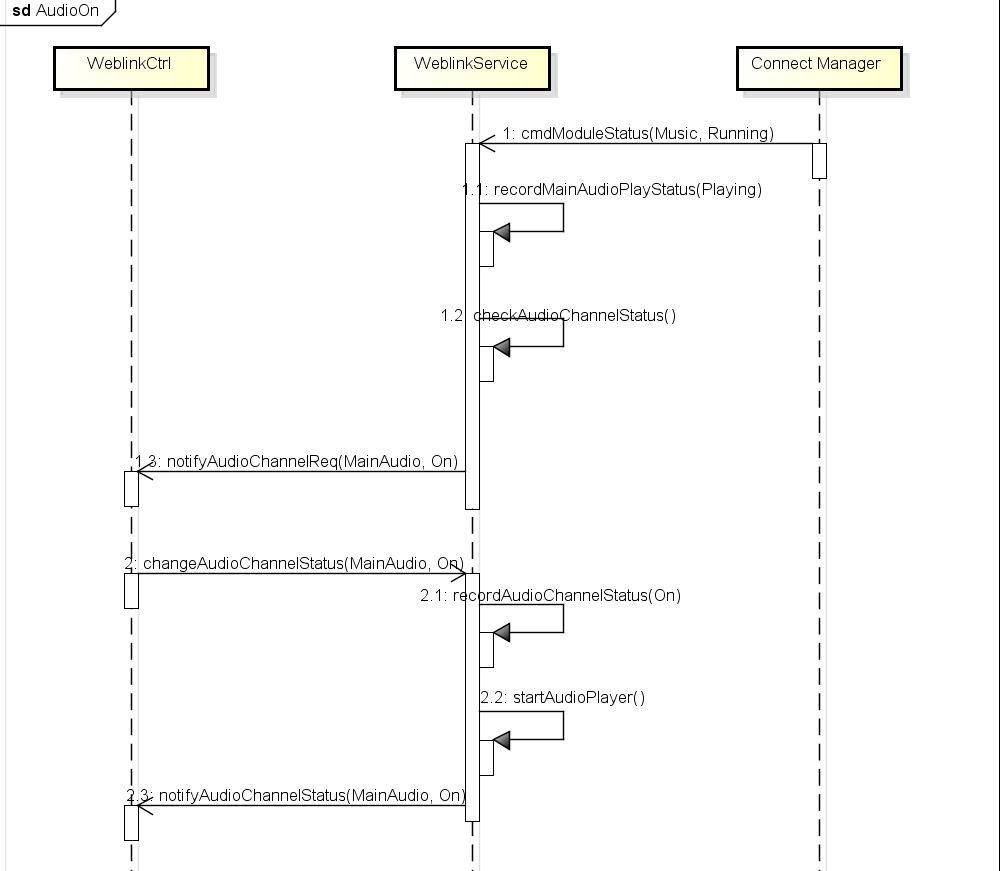
|  |
| --- |
| //changeVideoChannelStatusPro方法中，当isReqVideOn为false时会调用到这个方法。  void WeblinkServiceManager::changeVideoStatus(bool isReqVideOn) /\* \_\_0x70302D\_DTFUNCID\_\_ \*/  {  WEBLINKLOGD("%s isReqVideOn = %d", \_\_FUNC\_\_, isReqVideOn);  DTLOG\_INFO(0x70302D, "VideoStatus -- isReqVideOn = %d", isReqVideOn);  m\_videoChannelStatus = isReqVideOn;  if(isReqVideOn)//当传入true时。。。停止audio ,播放videoAction    {  if(m\_client && isWLConnected() && !m\_isVideoPlayerPrepared)  {  videoAction();  AudioPlayStop();  m\_audioSt = WL\_AUDIO\_ST\_AVAILABLE;  AudioPlayStart();  }  else  {  DTLOG\_INFO(0x70302D,"VideoStatus -- m\_client->IsConnected() = %d m\_isVideoPlayerPrepared = %d", m\_client->IsConnected(), m\_isVideoPlayerPrepared);  WEBLINKLOGD("%s m\_client->IsConnected() = %d", \_\_FUNC\_\_, m\_client->IsConnected());  WEBLINKLOGD("%s m\_isVideoPlayerPrepared = %d", \_\_FUNC\_\_, m\_isVideoPlayerPrepared);  }  }  else  {  //停止Audio播放。释放audio的loopback  AudioPlayStop();  VideoStatusPro(false);  }  } |

#### ****5** changeAudioChannelStatus**

改变audio的状态。开---关

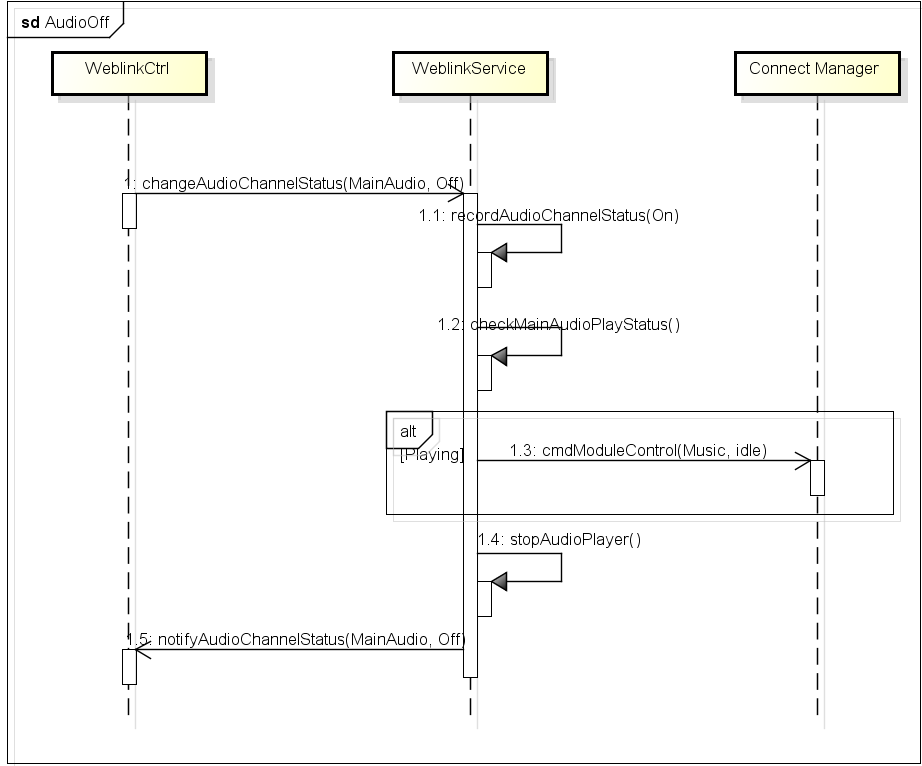
##### 时序图

Audio ON



该方法在webLinkService启动服务时，由connectManager调用，启动audio

Audio OFF



##### 具体实现

当需要关闭audio时，在proxy中发送请求，在stub中接收后，转到WebLInkServiceManager中的setAudioChannelStatus方法执行。在这个方法中，调用audioPlayStop方法，停止audio播放。

|  |
| --- |
| //设置音频状态。  void WeblinkServiceManager::setAudioChannelStatus(const WLAudioChannel channel, const WLAudioStatus status)  {  WEBLINKLOGD("%s", \_\_FUNC\_\_);  m\_audioChannelStatus = status;  if (m\_audioChannelStatus == WLAudioStatus\_Off) {  //当传入的audioStatus是off时，调用stop方法，停止audioPlay播放  AudioPlayStop();  }  } |

在audioPlayStop中调用audioLoo中的freeLoopback方法。

|  |
| --- |
| //audio停止播放  void WeblinkServiceManager::AudioPlayStop() /\* \_\_0x70300B\_DTFUNCID\_\_ \*/  {  WEBLINKLOGD("%s m\_audioSt = %d", \_\_FUNC\_\_, m\_audioSt);  DTLOG\_INFO(0x70300B, "AudioStop -- m\_audioSt = %d", m\_audioSt);  if(m\_audioSt != WL\_AUDIO\_ST\_PLAYING)  {  return;  }  //m\_audioLoop在initiliza方法中被实例化  if(m\_audioLoop != NULL)  {  WEBLINKLOGD("%s freeLoopback", \_\_FUNC\_\_);  //调用freeLoopback方法,释放loopback  m\_audioLoop->freeLoopback();  //将audio的状态置为WL\_AUDIO\_ST\_IDLE  m\_audioSt = WL\_AUDIO\_ST\_IDLE;  }  else  {  DTLOG\_PTRASSERT(m\_audioLoop);  }  } |

##### 涉及模块

functionManager

WebLinkService

WebLinkServiceManager

##### 常用方法总结

暂无

#### ****6 C**hangeActiveStatus**

改变活动状态。

WLActiveStatus\_ForeGround= 0,

WLActiveStatus\_BackGround= 1,

WLActiveStatus\_Other= 2,

##### 具体实现

该方法在proxy中发出请求，在stubBase中接收后转到stub中。但是在stub中的**ChangeActiveStatus方法中，并没有做具体的处理**

|  |
| --- |
| android::status\_t WeblinkServiceStub::changeActiveStatus(SenderId id, const WLActiveStatus& status)  {  WEBLINKLOGD("[%s] status = %d",\_\_FUNC\_\_, status);  //TODO: add active status process.  return 0;  } |

##### 涉及模块

##### 常用方法总结

暂无

#### 7 startSearch

搜索webLink设备， 根据webLink的状态来确定是否重置Usb

##### 具体实现

在proxy中发送消息，传入senderid,在stubBase中接收，转到Stub中的startSearch方法中调用，webLinkserviceManager中的startSearch方法，向消息队列中发送一个message。

在OnRecessive方法中，通过case 匹配，处理这个message。在处逻辑中，调用了resetUsbPro。

weblinkState表示webLink的状态。

WL\_STATE\_NONE,

WL\_STATE\_IDLE,

WL\_STATE\_CONNECTING,

WL\_STATE\_CONNECTED,

WL\_STATE\_PLAYING,

在resetUsbPro方法中通过判断webLink的状态对应响应的逻辑处理

|  |
| --- |
| void WeblinkServiceManager::resetUsbPro() /\* \_\_0x70304A\_DTFUNCID\_\_ \*/  {  WEBLINKLOGD( "%s m\_weblinkState = %d", \_\_FUNC\_\_, m\_weblinkState);  DTLOG\_INFO(0x70304A, "resetUsb -- m\_weblinkState = %d", m\_weblinkState);  if(WL\_STATE\_PLAYING == m\_weblinkState || WL\_STATE\_CONNECTING == m\_weblinkState)  {  DTLOG\_INFO(0x70304A, "ResetUsb");  WEBLINKLOGD( "%s ResetUsb", \_\_FUNC\_\_);  //重置usb,调用底层的方法TryRestUsb方法，重置USB  GoniAOAManager::instance()->TryRestUsb();  }  else if(WL\_STATE\_IDLE == m\_weblinkState && isAppleDevice())  {  //usb reset later.  m\_needResetUsb = true;  }  else  {  WEBLINKLOGD( "%s isAppleDevice : %d ", \_\_FUNC\_\_, isAppleDevice());  //do nothing  }  } |

##### 涉及模块

functionManager

WebLinkService

AOAManager

##### 常用方法总结

暂无

#### 8 stopSercarch

停止搜索

##### 具体实现

经过发送、接收请求，最后在webLinkstub中return 0，并没有处理具体逻辑。

### ****AIDL文件****

#### 定义

AIDL（Android Interface Definition Language）是Android接口定义语言

#### 作用

可以用于让某个Service与多个应用程序组件之间进行跨进程通信，从而可以实现多个应用程序共享同一个Service的功能。

是用于定义服务器和客户端通信接口的一种描述语言，可以拿来生成用于IPC的代码。暴露方法给其他应用进行调用的应用称为服务端，调用其他应用的方法的应用称为客户端，客户端通过绑定服务端的Service来进行交互。

从某种意义上说AIDL其实是一个模板。

设计AIDL这门语言的目的就是为了实现进程间通信。

#### 优势

AIDL的优势在于速度快(系统底层直接是共享内存)，性能稳，效率高，一般进程间通信就用它。