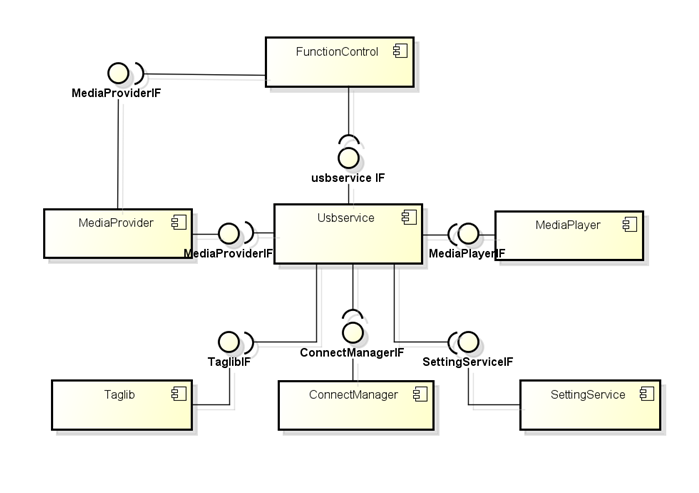
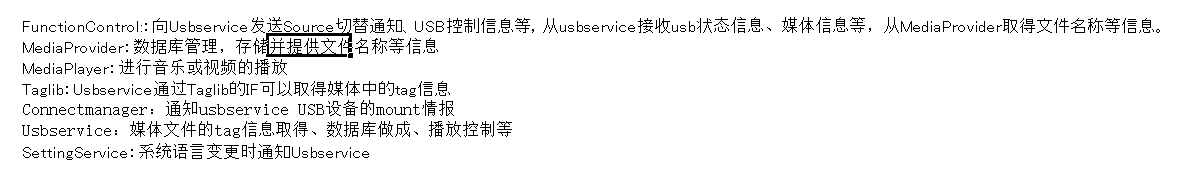
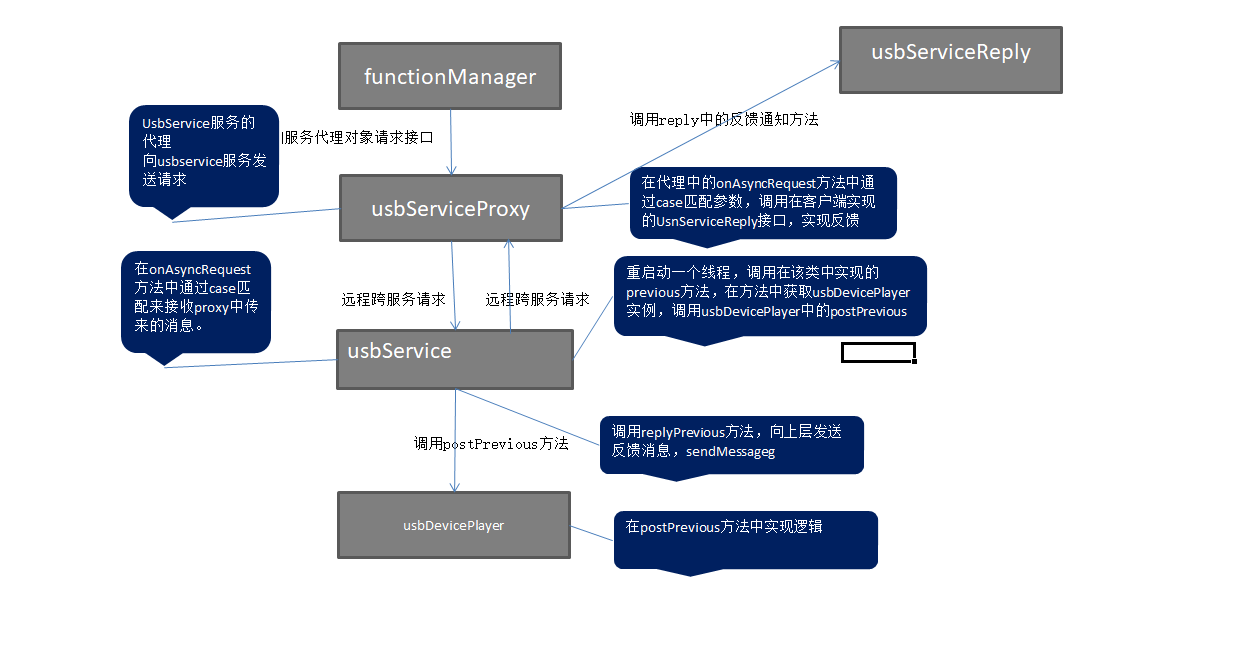
## 1 Usb与各个模块之间的调用关系





在usbservice中，上层与functionManager交互，下层与MediaService服务、DevicePlayerService服务、ConnectService服务交互。



## 2 Usbservice与各个模块交互的方式

### Usbservice与functionManager模块交互

向usbservice发送数据，和接受数据

在usbservice中，通过proxy与functionManager交互。

Proxy作为usbservice的代理接口，在proxy中的方法中sendMessage，该message会被GMUsbserviceBase中的onAsyncResponse方法中的case 接收。

而在UsbserviceBase中从serviceBasea中继承过来的sendAsyncRequest方法发送的消息，会被proxy中的onAsyncResponse方法中的case接收。

在usbService调用底层的逻辑处理请求前会将replyer应答反馈给上层。具体是，在Usbservice中发送sendMessage请求，在代理的onAsyncResponse方法中，以case匹配形式对应方法，实现将结果反馈给functionManager层。

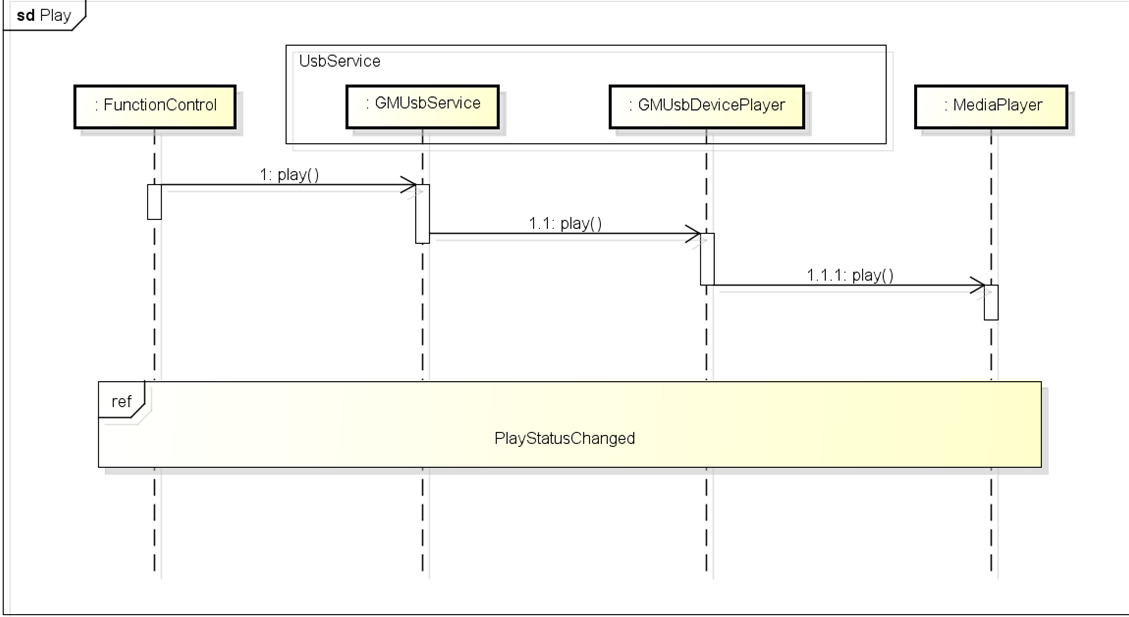
### UsbService与DevicePlayer模块的交互

在USbService中实现了plantform层的DevicePlayer的服务代理类，也叫客户端中的ClassicPlayer.cpp文件中的NMMPClassicPlayerDelegation类。在usbService中接受到message请求在方法onAsyncResponse中被case匹配处理。处理时，调用DevicePlayer中的方法，实现usbService服务与DevicePlayer服务的交互。

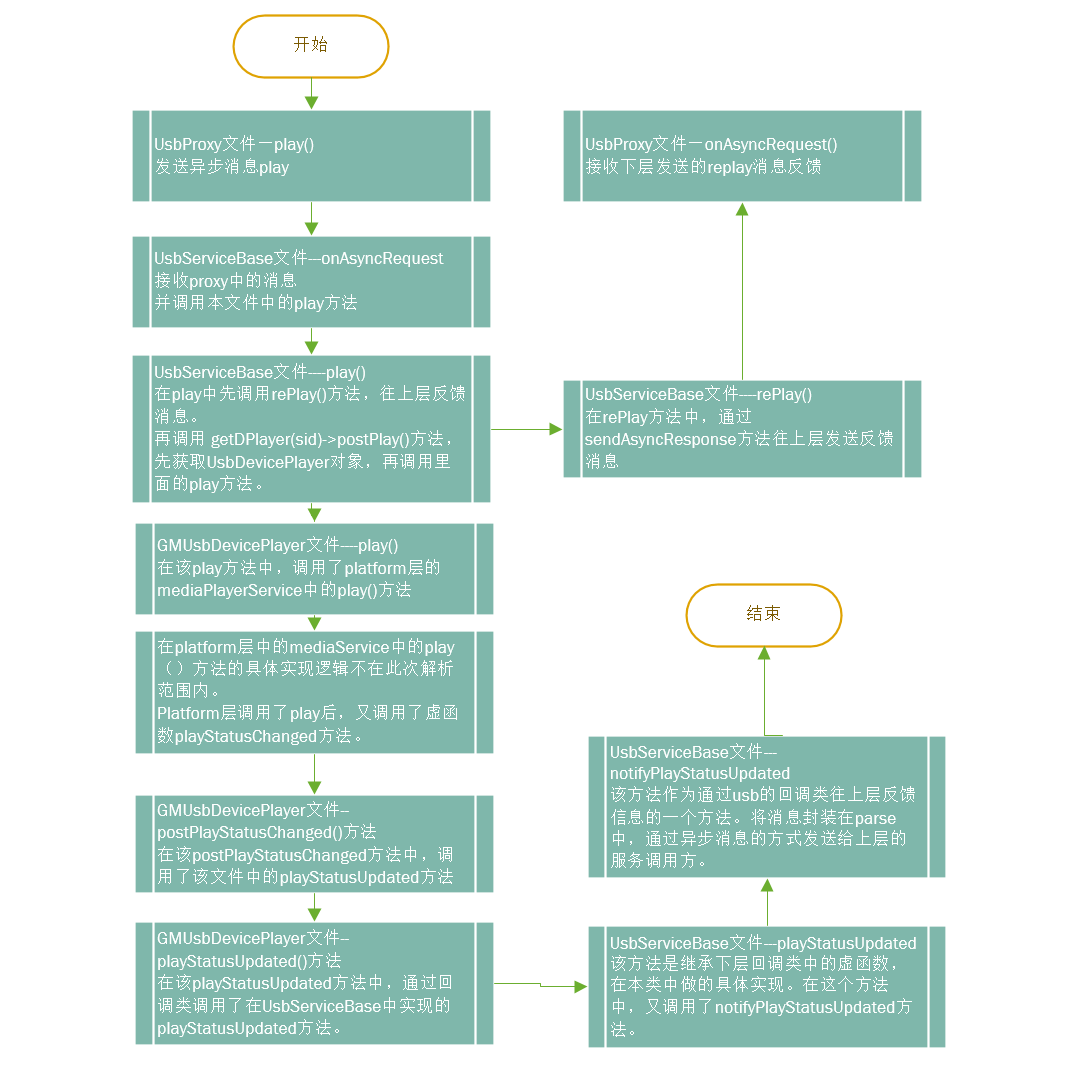
## 3 核心方法实现

### Play

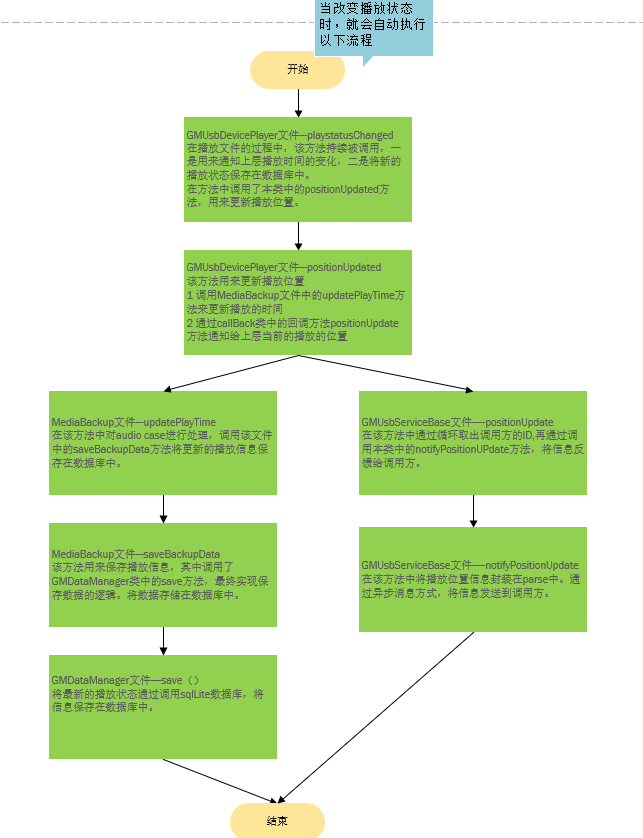
#### 时序



#### 执行流程



在执行完上图的流程后，这时车机的播放状态变为了playing状态，这时，需要把改变后的状态通知到上层和保存到数据库。具体流程如下。



#### 涉及的模块

FunctionManager

GMUsbService

Mediaplayerservice

#### 实现

functionManager 调用proxy代理中play方法

|  |
| --- |
| uint64\_t GMUsbProxy::play()  {  USBLOGD\_FUNCALL; //打log  //定义调用id，静态，unsigned long 型  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_Play; //方法id,通过枚举类型定义  //准备要发送的数据包  android::Parcel data;  prepareAsyncData(data);  // request service  //向usbservice发送异步请求,通过返回值判断请求是否成功。  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  //调用完该方法后，返回call\_id+1  return call\_id++;  } |

proxy代理服务中发出的play请求，在usbService的onAsyncRequest方法中被接收，通过case匹配请求，选择对应的case进行处理。

|  |
| --- |
| int GMUsbServiceBase::onAsyncRequest(SenderId &id, unsigned int code, const android::Parcel &data) {  case USBMethod\_Play:  {  //定义一个task  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_Play");  //启动主线程的looper方法，将task作为参数传入，调用该类中的play方法  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des,this, &GMUsbServiceBase::play, id));  break;  } |

在UsbServiceBase类中play方法被调用

|  |
| --- |
| void GMUsbServiceBase::play(SenderId sid) /\* \_\_0xB02013\_DTFUNCID\_\_ \*/  {  //通过该类中的replyPlay方法，向上层以消息的形式发送一个反馈  replyPlay(sid);  //判断usb的状态  if (GMUsbSourceState\_SourceOn != m\_sourceState) {  USBLOGD("the source is not on, %d", m\_sourceState);  DTLOG\_INFO(0xB02013, "tthe source is not on, %d", m\_sourceState);  return ;  }  //调用该类中的getDPlayer方法，获取DevicePlayer对象，并调用其postPlay方法  if (NULL != getDPlayer(sid)) {  getDPlayer(sid)->postPlay();  }  } |

在DevicePlayer类中，postPlay方法被调用

|  |
| --- |
| void GMUsbDevicePlayer::postPlay() /\* \_\_0x500017\_DTFUNCID\_\_ \*/  {  //打log  USBLOGD\_FUNCALL;  //判断线程类指针是否为空  if (m\_pMPCbThread) {  //不为空时启动新的线程，并调用本类中的play方法  m\_pMPCbThread->looper().postRunnable(MRunnableTask(this, &GMUsbDevicePlayer::play));  }  //为空时，直接通过函数调用的方式调用play方法  else {  play();  }  } |

DevicePlayer类中的play方法

在该方法中实现了上层发出的play指令的具体逻辑，分析代码结构发现，最终还是调用的DevicePlayer的父类，也就是在plantform层中NMMpClassciPlayer中的定义的play方法

|  |
| --- |
| void GMUsbDevicePlayer::play()  {  USBLOGD\_FUNCALL;  if (NULL == m\_pMediaPlayer) {  USBLOGD("player not exist");  DTLOG\_INFO(0x500039, "player not exist");  return;  }  //宏  #ifdef GMUSB\_PROPERTY\_FAKESEEK  if (GIUsbItemType\_APE == m\_mediaStatus.itemInfo.eItemType) {  m\_ForwardRewind = PlaySpeed\_NormalPlay;  m\_isSeeking = true;  NMMP\_PropertyMap muteProperty;  NMMPPropSetAudioMute(NMMP\_DEVICE\_ID\_AUDIO\_FRONT, true, muteProperty);  m\_pMediaPlayer->setProperty(muteProperty);  if (GIUsbPlayStatus\_Playing != m\_mediaStatus.ePlayStatus) {  m\_pMediaPlayer->play();  }  MediaBackup::instance()->updatePlayTime(m\_deviceId, m\_mediaStatus.itemInfo.position, m\_mediaStatus.ePlayMode);  if (NULL != m\_pTimerHolder) {  m\_pTimerHolder->stopTimer();  }  m\_pMediaPlayer->seekTo(m\_mediaStatus.itemInfo.position, m\_ForwardRewind, false);  NMMP\_PropertyMap soundProperty;  NMMPPropSetAudioMute(NMMP\_DEVICE\_ID\_AUDIO\_FRONT, false, soundProperty);  m\_pMediaPlayer->setProperty(soundProperty);  }  #ifdef GMUSB\_PROPERTY\_FAKEOTHERSEEK  else if (PlaySpeed\_NormalPlay != m\_ForwardRewind) {  m\_ForwardRewind = PlaySpeed\_NormalPlay;  m\_isSeeking = true;  MediaBackup::instance()->updatePlayTime(m\_deviceId, m\_mediaStatus.itemInfo.position, m\_mediaStatus.ePlayMode);  if (NULL != m\_pTimerHolder) {  m\_pTimerHolder->stopTimer();  }  m\_pMediaPlayer->seekTo(m\_mediaStatus.itemInfo.position, m\_ForwardRewind, false);  }  #endif  else {  m\_ForwardRewind = PlaySpeed\_NormalPlay;  m\_pMediaPlayer->play();  }  #else  m\_ForwardRewind = PlaySpeed\_NormalPlay;  m\_pMediaPlayer->play();  #endif  } |

Platform层的mediaPlayerService中的ClassicMediaPlayer文件中的play方法

具体逻辑没看懂，等待下次详细解析。

|  |
| --- |
| INT64 NMMPClassicPlayer::play(BOOL sync, const NMMP\_PropertyMap \*pProperty)  {  NMMPAutoSync autoSync(m\_syncExec);  NMMP\_CHECKPTR\_IF\_FAILED\_RETURN(m\_pRequestMgr);  return m\_pRequestMgr->reqPlay(pProperty, sync ? NMMP\_REQ\_TYPE\_SYNC : NMMP\_REQ\_TYPE\_ASYNC);  } |

在platform层的mediaPlayerService层中的ClassicMediaPlayer文件中的play方法的执行逻辑中，最终调用了mediaPlayerService中的NMMPClassicPlayerDelegation类中playStatusChanged虚函数。该函数在usbService中的GMUsbDevicePlayer文件中被继承。因此最终play动作执行完成后，通过playStatusChanged方法。完成一系列的状态设置动作和对上层调用方的通知。

在GmUsbServicePlayer文件中的playStatusChanged方法中，主要是设置当前的播放状态信息。同时调用UsbServiceBase文件中的继承自回调类中的playStatusUpdated方法

|  |
| --- |
| // update info to proxy  void GMUsbDevicePlayer::playStatusUpdated(NMMP\_PlaybackState state, int32\_t speed)  {  USBLOGD\_FUNCALL;  if (NULL == m\_pDPCB) {  USBLOGW("m\_pDPCB not exist, return");  DTLOG\_INFO(0x500045, "m\_pDPCB not exist, return");  return;  }  GIUsbPlayStatus status = GIUsbPlayStatus\_Unknown;  switch (state)  {  case NMMP\_PlaybackState\_Stopped:  m\_mediaStatus.ePlayStatus = status;  USBLOGD("playStatusChanged, state is stopped");  return ;  case NMMP\_PlaybackState\_Prepared:  status = GIUsbPlayStatus\_Ready;  USBLOGD("playStatusChanged, state is prepared");  break;  case NMMP\_PlaybackState\_Paused:  status = GIUsbPlayStatus\_Paused;  USBLOGD("playStatusChanged, state is paused");  break;  case NMMP\_PlaybackState\_Playing:  {  // clear the flag :the first file can play  m\_ignoreError = false;  if (speed == PlaySpeed\_NormalPlay) {  status = GIUsbPlayStatus\_Playing;  USBLOGD("playStatusChanged, state is playing");  }  else if (speed == PlaySpeed\_SlowForward) {  status = GIUsbPlayStatus\_SForward;  USBLOGD("playStatusChanged, state is SF");  }  else if (speed > PlaySpeed\_NormalPlay) {  status = GIUsbPlayStatus\_FForward;  USBLOGD("playStatusChanged, state is FF");  }  else if (speed < PlaySpeed\_SlowForward) {  status = GIUsbPlayStatus\_FRewind;  USBLOGD("playStatusChanged, state is FR");  }  break;  }  default:  break;  }  m\_mediaStatus.ePlayStatus = status;  #ifdef GMUSB\_PROPERTY\_FAKEOTHERSEEK  if (PlaySpeed\_NormalPlay != m\_ForwardRewind  && NMMP\_PlaybackState\_Paused == state) {  USBLOGD("ignore the pause state of when FF or FR");  }  else {  m\_pDPCB->playStatusUpdated(status);  }  #else  m\_pDPCB->playStatusUpdated(status);  #endif  } |

在UsbServiceBase文件中的playStatusUpdated方法中。该方法作为基类中的虚函数在此类中被重写。主要的逻辑是调用了UsbServiceBase文件中的notifyPlayStatusUpdated方法。实现通知上层调用方当前播放的状态信息。

|  |
| --- |
| void GMUsbDevicePlayerNotify::playStatusUpdated(GIUsbPlayStatus playStatus) /\* \_\_0xB02051\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  android::sp<GMUsbServiceBase> sp = m\_wpServiceBase.promote();  // notify to proxy  if (NULL != sp.get()) {  std::vector<int> sdlist = sp->getSenderList(m\_deviceId);  for (unsigned int i = 0; i < sdlist.size(); ++i) {  sp->notifyPlayStatusUpdated(sdlist[i], playStatus);  }  }  } |

在notifyPlayStatusUpdate方法中通过发送异步消息的方式通知给上层播放状态更新。

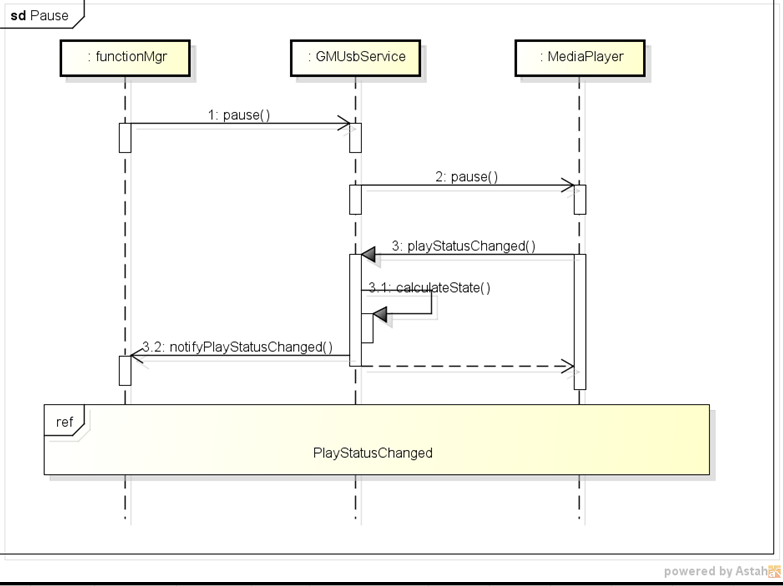
|  |
| --- |
| void GMUsbServiceBase::notifyPlayStatusUpdated(SenderId sid, const GIUsbPlayStatus& ePlayStatus  {  USBLOGD\_FUNCALL;  android::Parcel data;  unsigned int code = USBNotify\_PlayStatusUpdated;  data.writeInt32(ePlayStatus);  data.setDataPosition(0);  sendAsyncResponse(sid, code, data);  } |

#### 总结

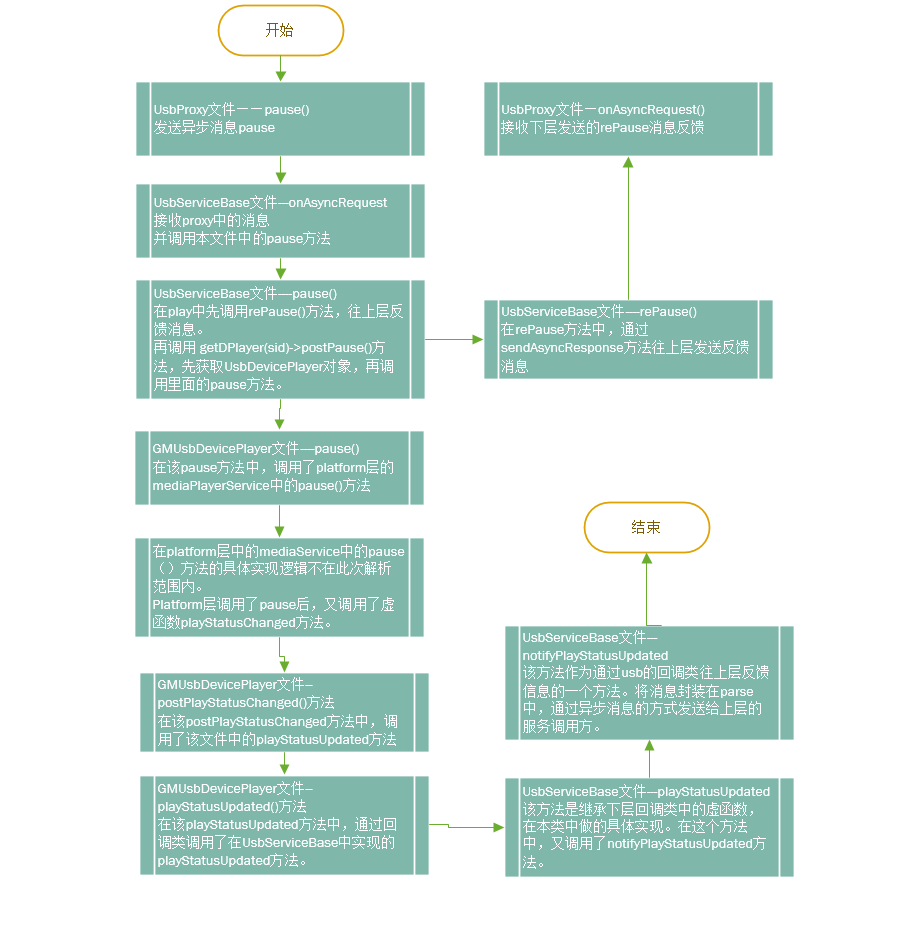
点击车机中的播放按钮时，触发该方法。经过上述的层层调用，最终实现了播放文件的功能。在此次解析过程中，涉及到UsbService模块的代码可以理解。但是因为最终该方法的实际逻辑还是在platform层的MedaiPlayerService中的client---Classicplaer中的play方法中实现。因不熟悉MedaiPlayerService模块代码，无法真正理解play的实现原理。

### Pause

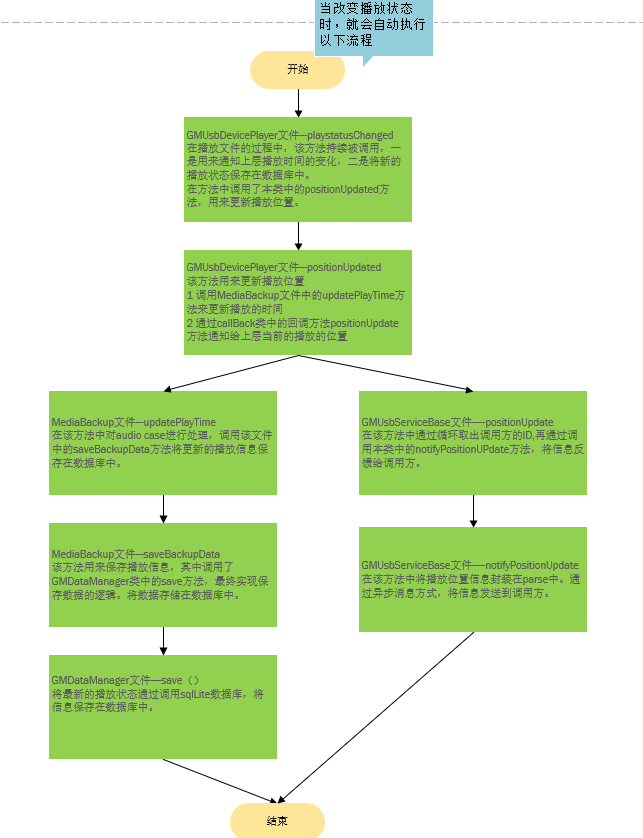
#### 时序



#### 执行流程



在执行完上图的流程后，这时车机的播放状态变为了pause状态，这时，需要把改变后的状态通知到上层和保存到数据库。具体流程如下。



#### 涉及模块

FunctionManager

GMUsbService

DevicePlayer

#### 实现

部分代码

代码的实现逻辑和play一致，都是从functionManager发出请求的message,在service中做中转同时给上层发反馈，最终在GMUSbDevicePlayer（MediaPlayService代理）中实现逻辑。

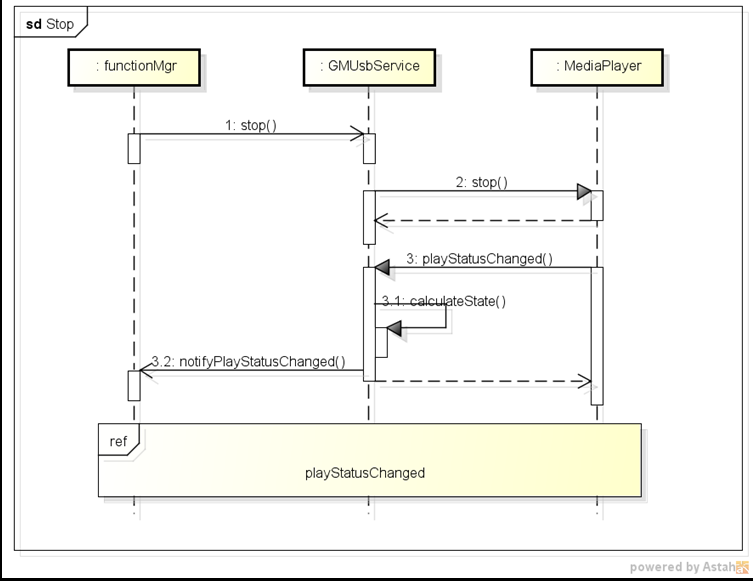
这里列举在GMUsbDevicePlayer中的pause方法，在该类中的pause中，又调用了plantform层的NMMpClassicPlayer中的pause方法。

|  |
| --- |
| void GMUsbDevicePlayer::pause()  {  USBLOGD\_FUNCALL;  if (NULL == m\_pMediaPlayer) {  USBLOGD("player not exist");  DTLOG\_INFO(0x50003A, "player not exist");  return;  }  m\_pMediaPlayer->pause();  } |

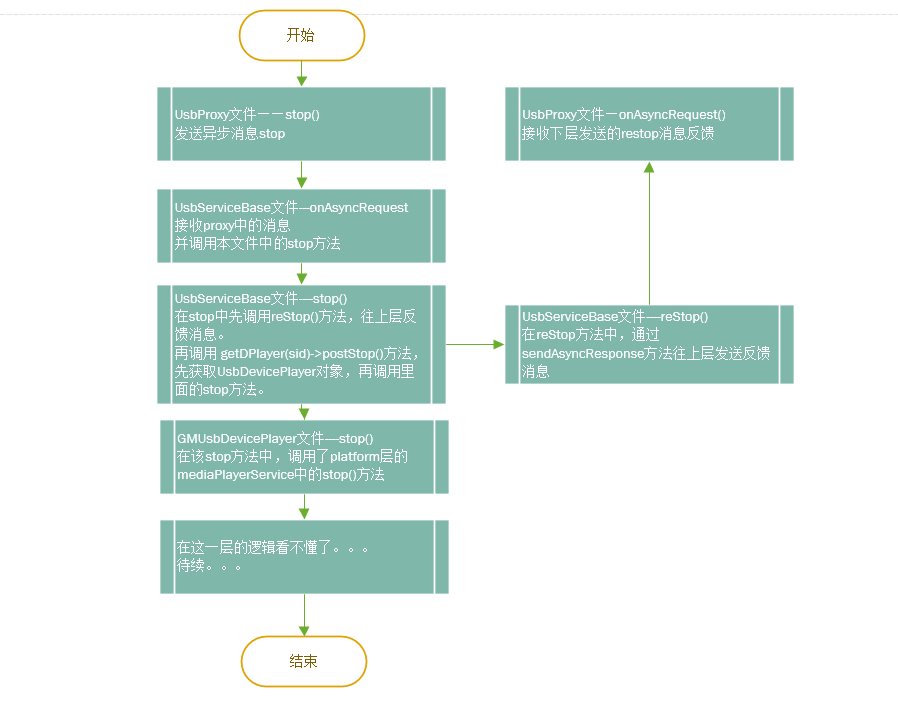
在执行完下层的pause方法后，会调用GMUsbDevicePlayer类中的playStatusUpdate方法，一是将pause状态反馈给上层，二是将最新的播放状态信息保存在数据库中。

### Stop

#### 时序



#### 执行流程



#### 涉及模块

FunctionManager

GMUsbService

DevicePlayer

#### 实现

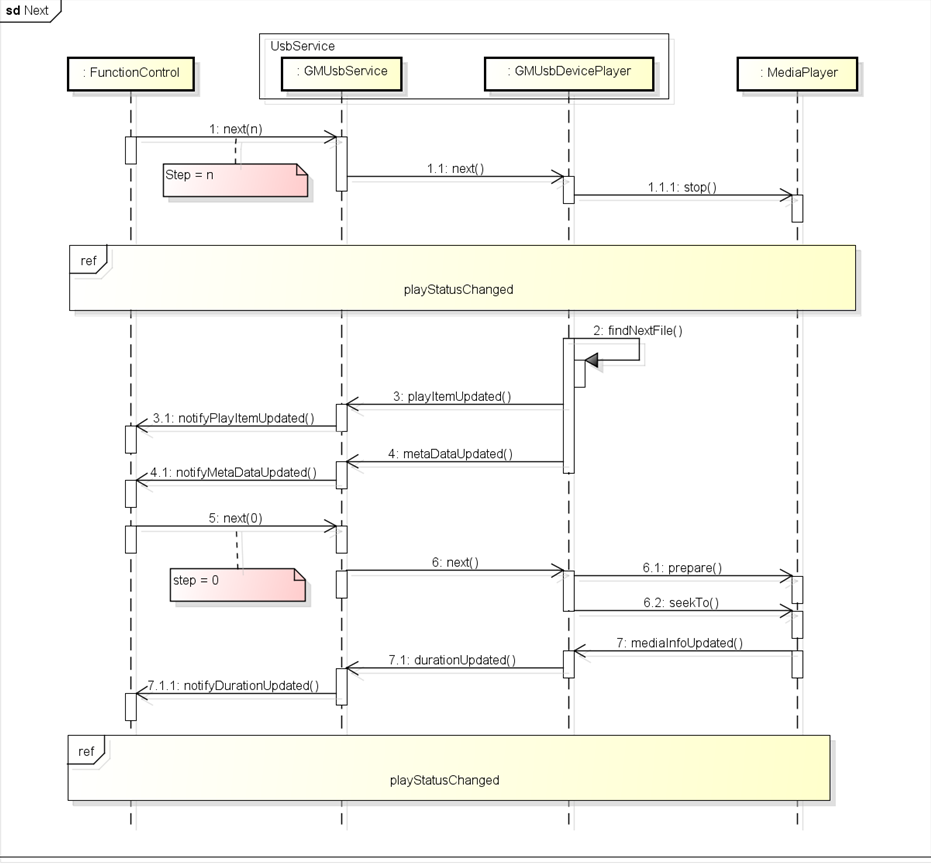
代码的实现逻辑和play一致，都是从functionManager发出请求的message,在service中做中转同时给上层发反馈，最终在GMUSbDevicePlayer（MediaPlayService代理）中实现逻辑。

这里列举在GMUsbDevicePlayer中的stop方法，在该类中的stop中，又调用了plantform层的NMMpClassicPlayer中的stop方法。

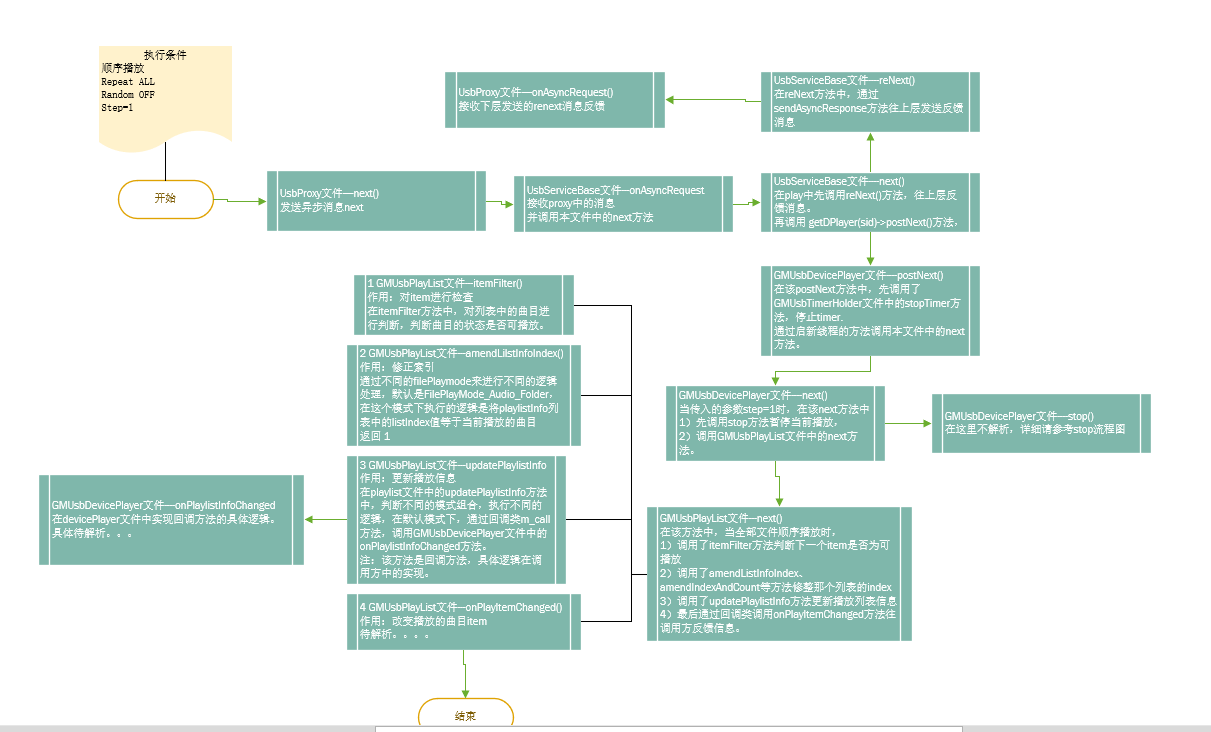
|  |
| --- |
| void GMUsbDevicePlayer::stop() /\* \_\_0x50003B\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  //判断mediaPlayer对象是否为空。  if (NULL == m\_pMediaPlayer) {  USBLOGD("player not exist");  DTLOG\_INFO(0x50003B, "player not exist");  return;  }  //判断曲目的id是否为无穷大，不为则设置为无穷大。再调用下层的stop方法，同时设置playStatus为stoped状态。  if (m\_mediaStatus.itemInfo.uid != INFINITE) {  m\_mediaStatus.itemInfo.uid = INFINITE;  m\_pMediaPlayer->stop();  m\_mediaStatus.ePlayStatus = GIUsbPlayStatus\_Stopped;  }  } |

### Next

#### 时序



#### 执行流程



注：上图如有看不清之处，请参考usbService\_Next.vsdx文件

#### 涉及模块

FunctionManager

GMUsbService

MediaPlayer

#### 实现

在functionMaanger中调用usbService代理proxy中的next方法，向usbService发送异步请求。

|  |
| --- |
| uint64\_t GMUsbProxy::next(const uint32\_t step)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_Next;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(step);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

在usbService中通过case匹配的方式接受处理请求，并通过新启线程的方式，调用usbservice中的next方法。

|  |
| --- |
| 代码和以上模块类似。。。 |

在next方法中，向上层通知通知，同时调用GMUsbDevicePlayer类中的postNext方法。

|  |
| --- |
| 代码和以上模块类似。。。 |

在postNext方法中调用GMUsbDevicePlayer中next方法。通过主线程是否存在判断是通过线程调用还是函数调用。

|  |
| --- |
| 代码和以上模块类似。。。 |

在next方法中实现了具体的逻辑。在next中又调用了playingList中的next.。其实真正的next的逻辑实在playingList中实现的

|  |
| --- |
| void GMUsbDevicePlayer::next(int step, bool forceSequence, bool autoPlay) /\* \_\_0x500028\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  //判断播放列表是否为空。m\_pNowPlayingList在initialize方法中被初始化。  if (NULL == m\_pNowPlayingList) {  USBLOGD("m\_pNowPlayingList not exist");  DTLOG\_INFO(0x500028, "m\_pNowPlayingList not exist");  return;  }  //判断是否连接了mediaplayer  if (m\_isMpBlinkReset) {  USBLOGD("ignor this command, wait for mediaplayer connect");  DTLOG\_INFO(0x500028, "ignor this command, wait for mediaplayer connect");  return;  }  //当步数大于0时  if (0 < step) {  USBLOGD("continue to next");  m\_readyToPlay = false;  //调用stop先停止当前播放  stop();  //调用playingList中的next  m\_pNowPlayingList->next(step, forceSequence);  }  //当步数为0时且自动播放时  if (0 == step || true == autoPlay) {  USBLOGD("stop next and start to play, autoPlay = %d", autoPlay);  m\_readyToPlay = true;  //当前播放item的id不等于INFINITE（无限大）  if (INFINITE != m\_mediaStatus.itemInfo.uid) {  //设置播放准备，在setPrepare中，又调用了prepare方法，而在prepare中又调用了  //底层的prepare方法，真正的prepare逻辑是在plantform中实现。  setPrepare();  //设置播放，调用该类中的setPlay f方法。  setPlay();  }  else {  USBLOGE("ItemInfo uid is not available, trigger failed!!");  DTLOG\_INFO(0x500028, "ItemInfo uid is not available, trigger failed!!");  }  }  else {  USBLOGD("the step is invalid");  DTLOG\_INFO(0x500028, "the step is invalid");  }  } |

在GMUsbDevicePlayer中又调用了GMMediaPlayList类中的next方法。

|  |
| --- |
| bool GMMediaPlaylist::next(unsigned int step, bool forceSequence)  {  PLLOG\_FUNCALL;  if (NULL == m\_cb || DeviceType\_None >= m\_deviceInfo.type() || DeviceType\_Count <= m\_deviceInfo.type()) {  PLLOGW("next : param error, m\_cb = [%p], type = [%d]", m\_cb, m\_deviceInfo.type());  return false;  }  PLLOGD("next step = [%d], forceSequence = [%d]", step, forceSequence);  if (m\_totalError) {  PLLOGD("all error >> m\_errorCount = [%d]", m\_errorCount);  PLLOGD("[debug] : all error!!!,please choose another playlist");  m\_cb->onPlaylistEnded(EndedType\_AllError);  return false;  }  if (!forceSequence && (RandomMode\_All == m\_randomMode || RandomMode\_List == m\_randomMode)) {  if (m\_randomIndex >= m\_random.size()) {  PLLOGD("randomIndex error = [%d]", m\_randomIndex);  return false;  }  for (unsigned int i = 0, startIndex = 0; i < step; ++i) {  startIndex = m\_randomIndex;  do {  ++m\_randomIndex;  if (m\_randomIndex >= m\_random.size()) {  m\_randomIndex = 0;  }  if (startIndex == m\_randomIndex) {  // all file in this list has been played once,reset the list  random\_shuffle(m\_random.begin(), m\_random.end());  for (unsigned int j = 0; j < m\_random.size(); ++j) {  if (NULL != m\_random[j].get()) {  m\_random[j]->setPlayed(false);  }  }  m\_errorCountInList = 0;  }  } while (itemFilter());  m\_errorCountInList = 0;  m\_random[m\_randomIndex]->setPlayed(true);  }  // keep normal index point to the currentItem  m\_originIndex = m\_random[m\_randomIndex]->index();  if (RandomMode\_All == m\_randomMode) {  if (1 == amendListInfoIndex()) {  amendIndexAndCount();  }  }  updatePlaylistInfo();  if (m\_listError) {  PLLOGD("[debug] : list error!!!,please choose another playlist");  m\_listError = false;  m\_cb->onPlaylistEnded(EndedType\_ListError);  }  else {  m\_cb->onPlayItemChanged(m\_random[m\_randomIndex]);  }  }  else { // randomMode\_Off or forceSequence  if (m\_originIndex >= m\_origin.size()) {  PLLOGD("originIndex error = [%d]", m\_originIndex);  return false;  }  for (unsigned int i = 0; i < step; ++i) {  do {  ++m\_originIndex;  if (m\_originIndex >= m\_origin.size()) {  m\_originIndex = 0;  m\_errorCountInList = 0;  }  } while (itemFilter(forceSequence));  }  // keep the randomIndex point to the current item  if (forceSequence) {  for (unsigned int i = 0; i < m\_random.size(); ++i) {  if (m\_random[i]->index() == m\_originIndex) {  m\_randomIndex = i;  break;  }  }  }  if (RepeatMode\_All == m\_repeatMode || RepeatMode\_One == m\_repeatMode) {  if (1 == amendListInfoIndex()) {  amendIndexAndCount();  }  }  updatePlaylistInfo();  if (m\_listError) {  PLLOGD("[debug] : list error!!!,please choose another playlist");  m\_listError = false;  m\_cb->onPlaylistEnded(EndedType\_ListError);  }  else {  m\_cb->onPlayItemChanged(m\_origin[m\_originIndex]);  }  }  PLLOGD("operation next succeed");  return true;  } |

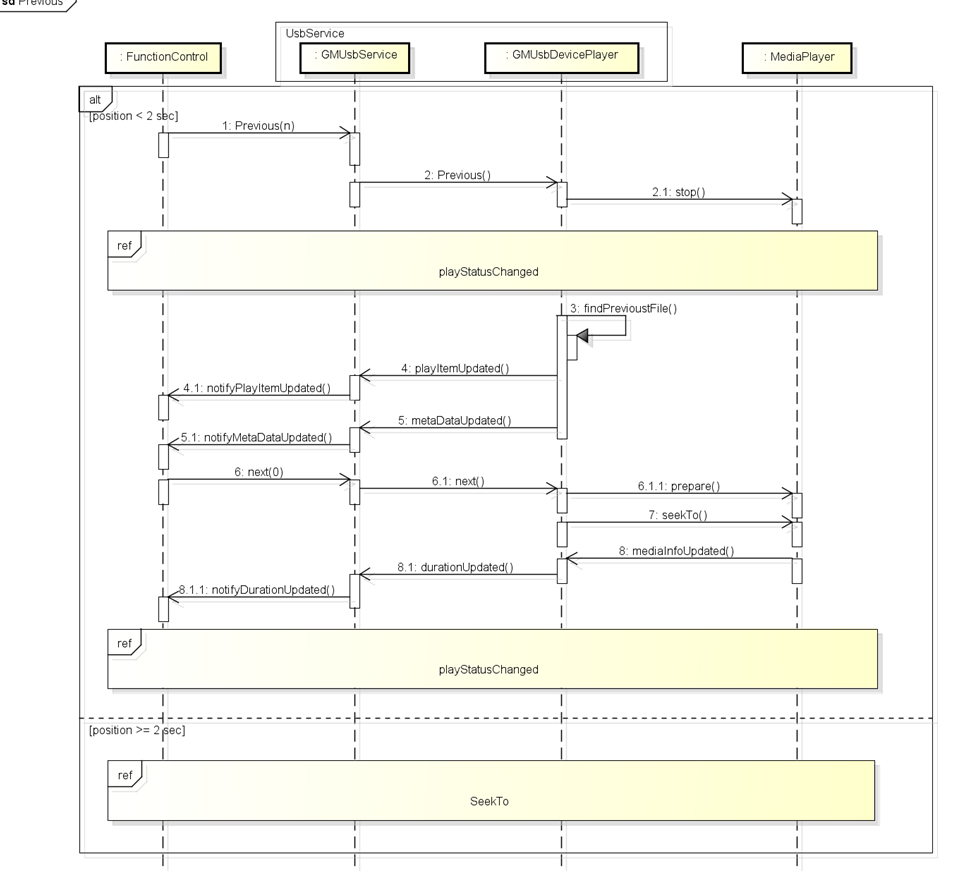
在playingList中的next方法中实现下一曲的真正逻辑

|  |
| --- |
| //该方法在GMUsbDevicePlayer文件中的next方法中被调用  //forceSequence参数，判断是否为顺序执行。为0时随机执行。不为0时顺序执行。  bool GMMediaPlaylist::next(unsigned int step, bool forceSequence)  {  PLLOG\_FUNCALL;  //对一些前提条件做下判断。  if (NULL == m\_cb || DeviceType\_None >= m\_deviceInfo.type() || DeviceType\_Count <= m\_deviceInfo.type()) {  PLLOGW("next : param error, m\_cb = [%p], type = [%d]", m\_cb, m\_deviceInfo.type());  return false;  }  //打印传入参数  PLLOGD("next step = [%d], forceSequence = [%d]", step, forceSequence);  //判断m\_totalError是否为true  if (m\_totalError) {  PLLOGD("all error >> m\_errorCount = [%d]", m\_errorCount);  PLLOGD("[debug] : all error!!!,please choose another playlist");  //在这里，回调函数被devicePlayer继承重写。这里调用的是devicePlayer中的onPlaylistEnded  m\_cb->onPlaylistEnded(EndedType\_AllError);  return false;  }    //在forceSequence为false时，并且随机模式是all或者list下的逻辑  if (!forceSequence && (RandomMode\_All == m\_randomMode || RandomMode\_List == m\_randomMode)) {  //循环索引和列表的容量比较，当randomIndex的数值大于random列表的size时，会报错。  if (m\_randomIndex >= m\_random.size()) {  PLLOGD("randomIndex error = [%d]", m\_randomIndex);  return false;  }  for (unsigned int i = 0, startIndex = 0; i < step; ++i) {  startIndex = m\_randomIndex;  //下面这个do...while循环的作用是：通过两个索引循环判断random列表中的item是否符合播放条件。如果全部都播放过，则将random表重置。  do {  ++m\_randomIndex;  //当循环的索引大于最大文件数时，索引置零。从头开始，  if (m\_randomIndex >= m\_random.size()) {  m\_randomIndex = 0;  }  //重置random表  if (startIndex == m\_randomIndex) {  // all file in this list has been played once,reset the list  //调用函数random\_shuffle重置列表.  random\_shuffle(m\_random.begin(), m\_random.end());  //将random列表中对应的item的播放状态设置false.  for (unsigned int j = 0; j < m\_random.size(); ++j) {  if (NULL != m\_random[j].get()) {  m\_random[j]->setPlayed(false);  }  }  m\_errorCountInList = 0;  }  //调用itemzFilterr方法。对item进行属性判断  } while (itemFilter());  m\_errorCountInList = 0;  //对于满足条件的item。将其setPlayed属性设置为true  m\_random[m\_randomIndex]->setPlayed(true);  }    // keep normal index point to the currentItem  m\_originIndex = m\_random[m\_randomIndex]->index();  //全部随机播放模式下，需要更新index  if (RandomMode\_All == m\_randomMode) {  if (1 == amendListInfoIndex()) {  //调用amendIndexAndCount方法修正index和cout  amendIndexAndCount();  }  }  updatePlaylistInfo();  if (m\_listError) {  PLLOGD("[debug] : list error!!!,please choose another playlist");  m\_listError = false;  m\_cb->onPlaylistEnded(EndedType\_ListError);  }  else {  m\_cb->onPlayItemChanged(m\_random[m\_randomIndex]);  }  }  else { // randomMode\_Off or forceSequence---随机模式关闭，或者强制顺序播放。  if (m\_originIndex >= m\_origin.size()) {  PLLOGD("originIndex error = [%d]", m\_originIndex);  return false;  }  //当顺序播放时，将originIndex的指针+step，超出范围则置零  for (unsigned int i = 0; i < step; ++i) {  do {  ++m\_originIndex;  if (m\_originIndex >= m\_origin.size()) {  m\_originIndex = 0;  m\_errorCountInList = 0;  }  } while (itemFilter(forceSequence));  }  // keep the randomIndex point to the current item  if (forceSequence) {  for (unsigned int i = 0; i < m\_random.size(); ++i) {  if (m\_random[i]->index() == m\_originIndex) {  m\_randomIndex = i;  break;  }  }  }  //重复模式下顺序播放的逻辑  if (RepeatMode\_All == m\_repeatMode || RepeatMode\_One == m\_repeatMode) {  //修正listInfo 的索引，amendListInfoIndex方法，将playlistIndex变量的索引等于listinfo列表中的播放曲目的index  if (1 == amendListInfoIndex()) {  //在默认播放模式的log中显示没有执行到该逻辑中，但是个人分析应该执行。  amendIndexAndCount();  }  }  updatePlaylistInfo();  if (m\_listError) {  PLLOGD("[debug] : list error!!!,please choose another playlist");  m\_listError = false;  m\_cb->onPlaylistEnded(EndedType\_ListError);  }  else {  m\_cb->onPlayItemChanged(m\_origin[m\_originIndex]);  }  }  PLLOGD("operation next succeed");  return true;  } |

待续。。。。

### Previous

#### 时序



#### 执行流程

#### 涉及模块

FunctionManager

GMUsbService

MediaPlayer

#### 实现

这里只列举在DMUsbDevicePlayer中的被调用的previous方法。Previous的逻辑和next的逻辑一致。

|  |
| --- |
| //在devicePlayer中，往线程队列中发送调用该方法的请求  void GMUsbDevicePlayer::previous(int step, bool autoPlay)  {  USBLOGD\_FUNCALL;  if (NULL == m\_pNowPlayingList) {  USBLOGD("m\_pNowPlayingList not exist");  DTLOG\_INFO(0x500029, "m\_pNowPlayingList not exist");  return;  }  if (m\_isMpBlinkReset) {  USBLOGD("ignor this command, wait for mediaplayer connect");  DTLOG\_INFO(0x500029, "ignor this command, wait for mediaplayer connect");  return;  }  if (0 < step) {  USBLOGD("continue to previous");  m\_readyToPlay = false;  //调用该类中的stop方法，停止播放  stop();  //调用下一曲方法，传入step  m\_pNowPlayingList->previous(step);  }  if (0 == step || true == autoPlay) {  USBLOGD("stop previous and start to play, autoPlay = %d", autoPlay);  m\_readyToPlay = true;  if (INFINITE != m\_mediaStatus.itemInfo.uid) {  setPrepare();  setPlay();  }  else {  USBLOGE("ItemInfo uid is not available, trigger failed!!");  DTLOG\_INFO(0x500029, "ItemInfo uid is not available, trigger failed!!");  }  }  else {  USBLOGD("the step is invalid");  DTLOG\_INFO(0x500029, "the step is invalid");  }  } |

在GMUsbDevicePlayer中调用了GMMediaPlayList的previous方法，实现真正的逻辑

|  |
| --- |
| bool GMMediaPlaylist::previous(unsigned int step)  {  PLLOG\_FUNCALL;  if (NULL == m\_cb || DeviceType\_None >= m\_deviceInfo.type() || DeviceType\_Count <= m\_deviceInfo.type()) {  PLLOGW("previous : param error, m\_cb = [%p], type = [%d]", m\_cb, m\_deviceInfo.type());  return false;  }  if (m\_totalError) {  PLLOGD("all error >> m\_errorCount = [%d]", m\_errorCount);  PLLOGD("[debug] : all error!!!,please choose another playlist");  m\_cb->onPlaylistEnded(EndedType\_AllError);  return false;  }  m\_errorCountInList = 0;  if (RandomMode\_All == m\_randomMode || RandomMode\_List == m\_randomMode) {  if (m\_randomIndex >= m\_random.size()) {  PLLOGD("randomIndex error = [%d]", m\_randomIndex);  return false;  }  m\_originIndex = m\_random[m\_randomIndex]->index();  for (unsigned int i = 0; i < step; ++i) {  do {  if (m\_originIndex == 0 && 0 != m\_origin.size()) {  m\_originIndex = m\_origin.size() - 1;  m\_errorCountInList = 0;  }  else {  --m\_originIndex;  }  } while (itemFilter(true));  m\_origin[m\_originIndex]->setPlayed(true);  }  // keep the randomIndex point to the current item  for (unsigned int i = 0; i < m\_random.size(); ++i) {  if (m\_random[i]->index() == m\_originIndex) {  m\_randomIndex = i;  break;  }  }  // mark played in randomArray  m\_random[m\_randomIndex]->setPlayed(true);  if (RandomMode\_All == m\_randomMode) {  if (1 == amendListInfoIndex()) {  amendIndexAndCount();  }  }  updatePlaylistInfo();  if (m\_listError) {  PLLOGD("[debug] : list error!!!,please choose another playlist");  m\_listError = false;  m\_cb->onPlaylistEnded(EndedType\_ListError);  }  else {  m\_cb->onPlayItemChanged(m\_random[m\_randomIndex]);  }  }  else { // randomMode\_Off  if (m\_originIndex >= m\_origin.size()) {  PLLOGD("originIndex error = [%d]", m\_originIndex);  return false;  }  for (unsigned int i = 0; i < step; ++i) {  do {  if (m\_originIndex == 0 && 0 != m\_origin.size()) {  m\_originIndex = m\_origin.size() - 1;  m\_errorCountInList = 0;  }  else {  --m\_originIndex;  }  } while (itemFilter());  }  if (RepeatMode\_All == m\_repeatMode || RepeatMode\_One == m\_repeatMode) {  if (1 == amendListInfoIndex()) {  amendIndexAndCount();  }  }  updatePlaylistInfo();  if (m\_listError) {  PLLOGD("[debug] : all error!!!,please choose another playlist");  m\_listError = false;  m\_cb->onPlaylistEnded(EndedType\_ListError);  }  else {  m\_cb->onPlayItemChanged(m\_origin[m\_originIndex]);  }  }  PLLOGD("operation previous succeed");  return true;  } |

### PlayAll

#### 时序

#### 执行流程

#### 涉及模块

FunctionManager

GMUsbService

DevicePlayer

#### 实现

在GMUsbDevicePlayer中的playAll方法。

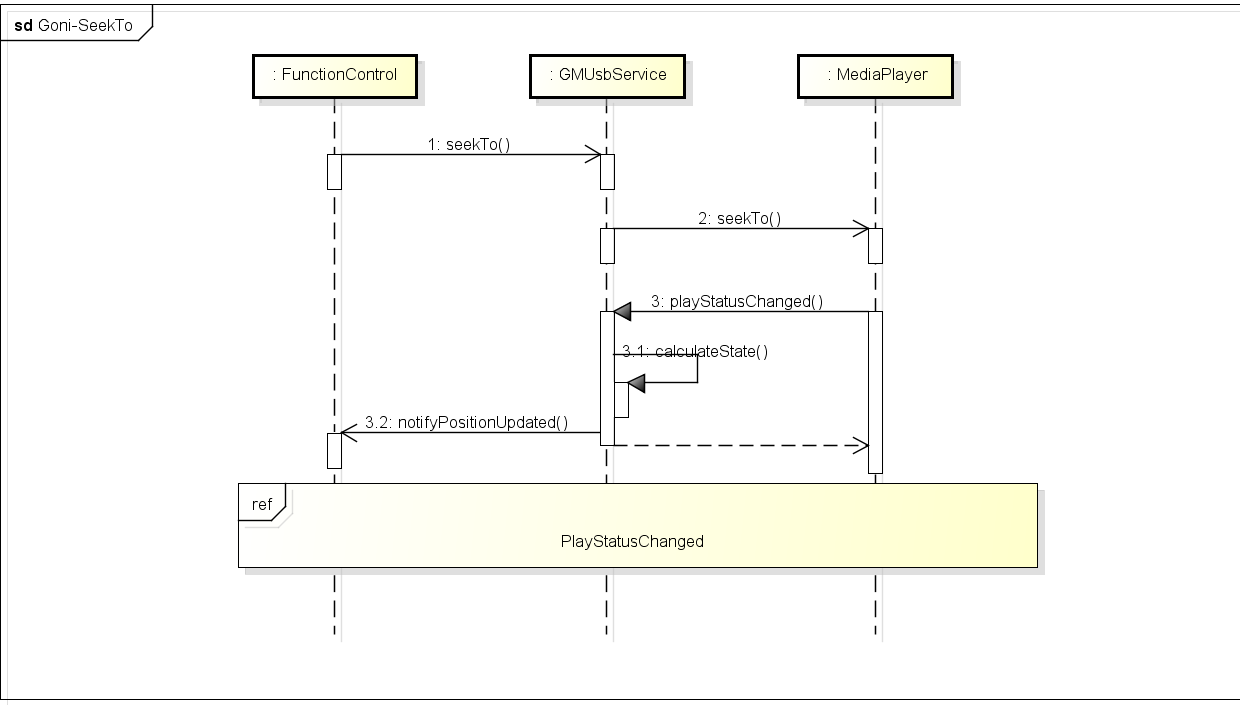
|  |
| --- |
| void GMUsbDevicePlayer::playAll(GMUsbAvMode mode, unsigned int itemid) /\* \_\_0x500022\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  //判断playingList是否为空。  if (NULL == m\_pNowPlayingList) {  USBLOGD("m\_pNowPlayingList not exist");  DTLOG\_INFO(0x500022, "m\_pNowPlayingList not exis");  return;  }  //设置当前速度。  m\_ForwardRewind = PlaySpeed\_NormalPlay;  //调用暂停方法stop  stop();  //设置播放模式  GIUsbPlayMode playMode = static\_cast<GIUsbPlayMode>(mode);  //更新播放模式。  playModeUpdated(playMode);  //调用GMMediaPlaylist中的playAll方法  m\_pNowPlayingList->playAll(static\_cast<GMMediaPlaylist::AvMode>(mode), static\_cast<int>(itemid));  //调用GMMediaPlayList中save方法保存  m\_pNowPlayingList->save();  } |

在GMMediaDevicePlayer中调用了GMMediaPlayList类中的playAll方法，在其中实现 了真正的playAll逻辑

|  |
| --- |
| //AVMode mode参数：  bool GMMediaPlaylist::playAll(AvMode mode, int index)  {  PLLOG\_FUNCALL;  PLLOGD("playall mode = [%d], index = [%d]", mode, index);  int fileType = 0;  m\_playMode = mode;  if (AvMode\_Audio == mode) {  m\_playlistInfo.filePlayMode = FilePlayMode\_Audio\_All;  fileType = 2;  }  else if (AvMode\_Video == mode) {  m\_playlistInfo.filePlayMode = FilePlayMode\_Video\_All;  fileType = 4;  }  else {  return false;  }  m\_playlistInfo.folderId = USB\_INVALID\_ID;  m\_playlistInfo.genreId = USB\_INVALID\_ID;  m\_playlistInfo.artistId = USB\_INVALID\_ID;  m\_playlistInfo.composerId = USB\_INVALID\_ID;  m\_playlistInfo.albumId = USB\_INVALID\_ID;  MediaSrcType type = srcType();  if (MediaSrcType\_Any == type) {  PLLOGW("srcType error");  return false;  }  NCUri uri;  NCList<NCString> select;  NCString where("");  NCString mkey("");  NCList<NCString> whereArgs;  NCString order("SORTKEY");  int dbStatus = deviceDbInfo(DeviceDbInfo\_Status);  if (AvMode\_Video == mode) {  uri = GIMediaProviderUri::FileBase::getContentUri(type);  select<< new NCString("ID")<< new NCString("FOLDER\_ID")<< new NCString("NAME")<< new NCString("PATH");  where = "TYPE=?";  mkey.format("%d", fileType); // 2 refs audio , 4 refs video  whereArgs.append(&mkey);  }  else if (4 <= dbStatus) {  uri = GIMediaProviderUri::AudioBase::Songs::getContentUri(type);  select<< new NCString("FID")<< new NCString("FOLDER\_ID")<< new NCString("FILE\_NAME")<< new NCString("FILE\_PATH")  << new NCString("COMPOSER\_ID")<< new NCString("GENRE\_ID")<< new NCString("ARTIST\_ID")<< new NCString("ALBUM\_ID");  }  else {  uri = GIMediaProviderUri::FileBase::getContentUri(type);  select<< new NCString("ID")<< new NCString("FOLDER\_ID")<< new NCString("NAME")<< new NCString("PATH");  where = "TYPE=?";  mkey.format("%d", fileType); // 2 refs audio , 4 refs video  whereArgs.append(&mkey);  }  #ifndef \_\_LOCAL  NCCursor\* cursor = NULL;  cursor = m\_pProviderResolver->query(uri, select, where, whereArgs, order);  if (NULL == cursor) {  PLLOGD("query failed, no data received");  select.clearData();  return false;  }  int listCountTmp = cursor->getCount();  PLLOGD("list Count = [%d]", listCountTmp);  cursor->moveToFirst();  ncsp<NMDataTable>::sp listSp = new NMDataTable(listCountTmp, PlayItem::Column\_Count);  if (NULL != listSp.get()) {  INIT\_TBL(listSp);  for (int i = 0; i < listCountTmp; ++i) {  listSp->putLong(i, PlayItem::Column\_Fid, cursor->getInt64(PlayItem::Column\_Fid));  listSp->putLong(i, PlayItem::Column\_FolderId, cursor->getInt64(PlayItem::Column\_FolderId));  listSp->putString(i, PlayItem::Column\_Name, cursor->getString(PlayItem::Column\_Name).getString(),  cursor->getString(PlayItem::Column\_Name).getLength());  listSp->putString(i, PlayItem::Column\_Path, cursor->getString(PlayItem::Column\_Path).getString(),  cursor->getString(PlayItem::Column\_Path).getLength());  if (AvMode\_Video == mode) {  listSp->putLong(i, PlayItem::Column\_ComposerId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_GenreId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_ArtistId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_AlbumId, USB\_INVALID\_ID);  }  else {  listSp->putLong(i, PlayItem::Column\_ComposerId, cursor->getInt64(PlayItem::Column\_ComposerId));  listSp->putLong(i, PlayItem::Column\_GenreId, cursor->getInt64(PlayItem::Column\_GenreId));  listSp->putLong(i, PlayItem::Column\_ArtistId, cursor->getInt64(PlayItem::Column\_ArtistId));  listSp->putLong(i, PlayItem::Column\_AlbumId, cursor->getInt64(PlayItem::Column\_AlbumId));  }  listSp->putLong(i, PlayItem::Column\_FileType, fileType);  cursor->moveToNext();  }  loadInfoList(listSp);  selectPlaylist(listSp, index);  // saveLastList();  }  FREEIF(cursor);  #else  NMDataTable\* tmp = GMUsbProvider::instance()->query(uri, select, where, whereArgs, order);  if (NULL != tmp) {  int listCountTmp = tmp->getRowCount();  ncsp<NMDataTable>::sp listSp = new NMDataTable(listCountTmp, PlayItem::Column\_Count);  INIT\_TBL(listSp);  for (int i = 0; i < listCountTmp; ++i) {  listSp->putLong(i, PlayItem::Column\_Fid, tmp->getLong(i, PlayItem::Column\_Fid));  listSp->putLong(i, PlayItem::Column\_FolderId, tmp->getLong(i, PlayItem::Column\_FolderId));  size\_t len;  listSp->putString(i, PlayItem::Column\_Name, tmp->getString(i, PlayItem::Column\_Name, len),  256);  listSp->putString(i, PlayItem::Column\_Path, tmp->getString(i, PlayItem::Column\_Path, len),  256);  if (AvMode\_Video == mode) {  listSp->putLong(i, PlayItem::Column\_ComposerId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_GenreId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_ArtistId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_AlbumId, USB\_INVALID\_ID);  }  else {  listSp->putLong(i, PlayItem::Column\_ComposerId, tmp->getLong(i, PlayItem::Column\_ComposerId));  listSp->putLong(i, PlayItem::Column\_GenreId, tmp->getLong(i, PlayItem::Column\_GenreId));  listSp->putLong(i, PlayItem::Column\_ArtistId, tmp->getLong(i, PlayItem::Column\_ArtistId));  listSp->putLong(i, PlayItem::Column\_AlbumId, tmp->getLong(i, PlayItem::Column\_AlbumId));  }  listSp->putLong(i, PlayItem::Column\_FileType, fileType);  }  loadInfoList(listSp);  selectPlaylist(listSp, index);  }  FREEIF(tmp);  #endif  select.clearData();  return true;  } |

### SeekTo

#### 时序



#### 执行**流程**

#### 涉及模块

FunctionManager

GMUsbService

DevicePlayer

#### 实现

在GMUsbDevicePlayer中seekTo被调用。在seekTo中又调用了NMMPClasssicPlayer中的seekTo方法和PlaylistManager中的autoNext方法。

|  |
| --- |
| void GMUsbDevicePlayer::seekTo(uint32\_t secTimeIn) /\* \_\_0x50003C\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  if (NULL == m\_pMediaPlayer) {  USBLOGD("player not exist");  DTLOG\_INFO(0x50003C, "player not exist");  return;  }  USBLOGD("the position need to seekto is %d", secTimeIn);  USBLOGD("the duration is %d", m\_mediaStatus.itemInfo.duration);  // because the operation of seekto is not finish,so the data is not be saved.  MediaBackup::instance()->updatePlayTime(m\_deviceId, secTimeIn, m\_mediaStatus.ePlayMode);  m\_mediaStatus.itemInfo.position = secTimeIn;  uint32\_t totalTime = m\_mediaStatus.itemInfo.duration;  if (-1 == static\_cast<int>(secTimeIn)) {  USBLOGD("play from tail of the item %d", secTimeIn);  m\_pMediaPlayer->seekTo(0, PlaySpeed\_NormalPlay, true);  }  else if ((totalTime > 0) && (((totalTime - secTimeIn) / 1000) < 1)) {  USBLOGD("stop seekto and next");  playCompletedNotify(0);  stop();  if (NULL != m\_pNowPlayingList) {  m\_pNowPlayingList->autoNext();  }  }  else {  if (!m\_isSeeking) {  #ifdef GMUSB\_PROPERTY\_FAKESEEK  if (GIUsbItemType\_APE == m\_mediaStatus.itemInfo.eItemType) {  m\_mediaStatus.itemInfo.position = secTimeIn;  play();  }  else {  m\_isSeeking = true;  m\_pMediaPlayer->seekTo(secTimeIn, PlaySpeed\_NormalPlay, false);  }  #else  m\_isSeeking = true;  m\_pMediaPlayer->seekTo(secTimeIn, PlaySpeed\_NormalPlay, false);  #endif  }  else {  USBLOGD("seekto is not completed ignore the command");  m\_ignorePosition = secTimeIn;  }  }  } |

PlaylistManager中的autoNext方法

|  |
| --- |
| bool GMMediaPlaylist::autoNext()  {  PLLOGD("autoNext");  if (NULL == m\_cb || DeviceType\_None >= m\_deviceInfo.type() || DeviceType\_Count <= m\_deviceInfo.type()) {  PLLOGW("autoNext : param error, m\_cb = [%p], type = [%d]", m\_cb, m\_deviceInfo.type());  return false;  }  if (m\_totalError) {  PLLOGW("warning : autoNext !!totalError!!");  m\_cb->onPlaylistEnded(EndedType\_AllError);  return true;  }  NCAutoSync autoSync(m\_syncObj);  if (RepeatMode\_One == m\_repeatMode) {  // need not switch to next playitem, play this again  if (NULL != currentPlayItem().get()  && currentPlayItem()->playable()) {  m\_cb->onNeedPlayAgain(currentPlayItem());  return true;  }  else {  return next(1);  }  }  else if (RepeatMode\_List == m\_repeatMode || RandomMode\_List == m\_randomMode) {  if (m\_infoList[m\_listIndex].isListError) {  return listUp(m\_playlistInfo.filePlayMode);  }  else {  return next(1);  }  }  else {  return next(1);  }  } |

### BindSource

#### 时序

无

#### 涉及模块

FunctionManager

GMUsbService

DevicePlayer

#### 实现

该方法主要是将usb与设备绑定

Proxy代理

|  |
| --- |
| // binding source usb1/usb2  uint64\_t GMUsbProxy::bindSource(const GIUsbSource& eSourceId)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_BindSource;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(eSourceId);  data.setDataPosition(0);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

UsbService中接收消息，启动线程，传入sourceId和senderId，调用该类中的bindSource方法

|  |
| --- |
| case USBMethod\_BindSource:  {  // define request param  GIUsbSource eSourceId;  // unpack request param  //从parse中获取sourceID  eSourceId = static\_cast<GIUsbSource>(data.readInt32());  // call request function  // bindSource(id, eSourceId);  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_BindSource");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::bindSource  , id, eSourceId));  break;  } |

UsbService 中的BindSource方法，在这个方法中，又调用了DevicePlayer中的initialize方法，在初始化时进行绑定操作。这也说明了Usbservice只是功能的中转站，最终的处理逻辑还是放在了各个其他服务的代理类中。

|  |
| --- |
| void GMUsbServiceBase::bindSource(SenderId sid, const GIUsbSource& eSourceId) /\* \_\_0xB0200B\_DTFUNCID\_\_ \*/  {  //往上层反馈信息  replyBindSource(sid, eSourceId);  //将source纳入到clientManager的管理中。  m\_spClientMgr->setLink(sid, static\_cast<int>(eSourceId));  if (NULL != getDPlayer(sid)) {  getDPlayer(sid)->initialize();  }  // TODO sync service status with proxy  // GIUsbServiceStatus statusInfo;  // if (NULL != getDPlayer(sid)) {  // statusInfo = getDPlayer(sid)->mediaStatus();  // }  // int deviceId = m\_spClientMgr->getDeviceId(sid);  // m\_usbExtInfo[deviceId].infoMask = mask;  // notifyUsbExtInfo(sid, m\_usbExtInfo[deviceId]);  } |

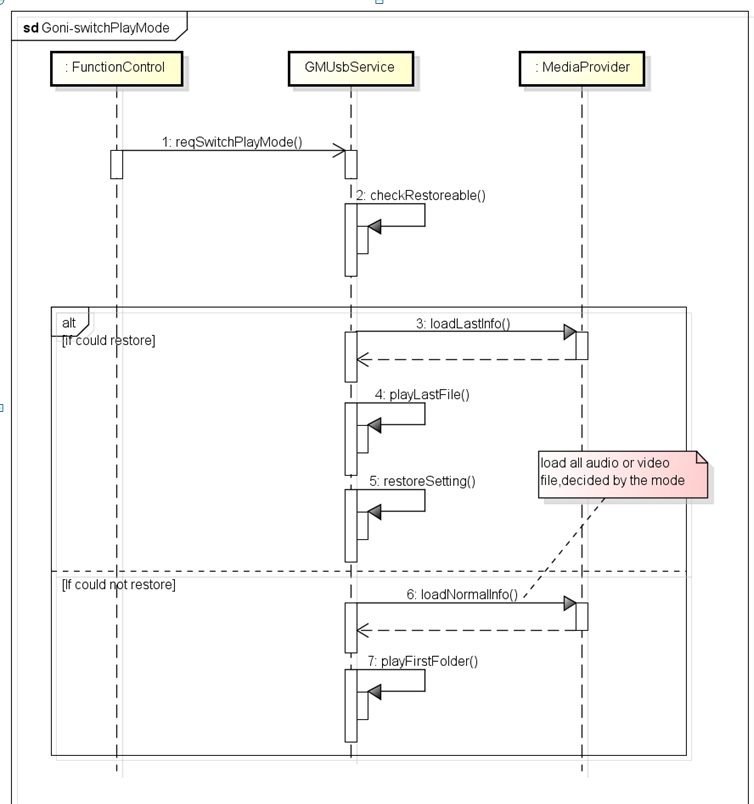
DevicePlayer中的initialize方法。

|  |
| --- |
| bool GMUsbDevicePlayer::initialize() /\* \_\_0x500007\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  //创建新的线程类  if (NULL == m\_pMPCbThread) {  m\_pMPCbThread = new GMUsbRunnableThread();  if (NULL == m\_pMPCbThread) {  return false;  }  }  //创建mediaPlayer对象  //启动线程  if (NULL == m\_pMediaPlayer) {  m\_pMPCbThread->startThread(GONI\_USBSERVICE\_THREAD\_PLAYER\_CALLBACK);  NCRunnableLooper workLooper(m\_pMPCbThread->looper());  //创建mediaPlayer对象  m\_pMediaPlayer = new NMMPClassicPlayer(workLooper, this);  if (NULL == m\_pMediaPlayer) {  return false;  }  // initialize classic player // move to onConnected  // m\_pMediaPlayer->initialize();  // m\_pMediaPlayer->registerNotify(GM\_USB\_MP\_STATUS\_MASK, GM\_USB\_MP\_INFO\_MASK);  // Do not need to connect in this PF Framework  // m\_pMediaPlayer->connectToService();  }  //初始化playlistCallback  if (NULL == m\_pPlaylistCallback) {  m\_pPlaylistCallback = new GMUsbDevicePlayer::PlaylistCallback(\*this);  }  //初始化playlist  if (NULL == m\_pNowPlayingList && NULL != m\_pPlaylistCallback) {  m\_pNowPlayingList = new GMMediaPlaylist(m\_pPlaylistCallback);  //设备信息  GMMediaPlaylist::DeviceInfo devInfo;  devInfo.setRootPath(m\_rootPath);  if (GMUsbDeviceId\_USB1 == m\_deviceId) {  devInfo.setType(GMMediaPlaylist::DeviceType\_Usb1);  }  else if (GMUsbDeviceId\_USB2 == m\_deviceId) {  devInfo.setType(GMMediaPlaylist::DeviceType\_Usb2);  }  else {  devInfo.setType(GMMediaPlaylist::DeviceType\_None);  }  //初始化一系列相关对象后，调用playList中的bindDevice方法  //将devInfo来实例化DeviceInfo  m\_pNowPlayingList->bindDevice(devInfo);  }  #ifdef GMUSB\_PROPERTY\_FAKESEEK  m\_spTimer = new LocalTimer(this, 100);  m\_pTimerHolder = new GMUsbTimerHolder(m\_spTimer.get(), "deviceplayer timer");  #endif  return true;  } |

### SwitchPlayMode

这个方法的主要功能就是Audio和Vedio之间的切换。

#### 时序



#### 涉及模块

FunctionManager

GMUsbService

DevicePlayer

#### 涉及文件

GMUsbProxy.h/cpp

SwitchPlayMode（）

GMUsbServiceBase.h/cpp

case USBMethod\_SwitchPlayMode

SwitchPlayMode（）

GMUSbDevicePlayer.h/cpp

PostSwitchPlayMode()

SwitchPlayMode()

RestorePlay()

seekTo()

playFirstFolder()

playFolder

#### 实现

Proxy中SwitchPlayMode方法

|  |
| --- |
| /// switch play mode  uint64\_t GMUsbProxy::switchPlayMode(const GIUsbPlayMode& ePlayMode)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_SwitchPlayMode;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(ePlayMode);  data.setDataPosition(0);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

UsbService中接收并响应proxy中的请求的case

|  |
| --- |
| case USBMethod\_SwitchPlayMode:  {  // define request param  GIUsbPlayMode ePlayMode;  // unpack request param  ePlayMode = static\_cast<GIUsbPlayMode>(data.readInt32());  // call request function  // switchPlayMode(id, ePlayMode);  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_SwitchPlayMode");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::switchPlayMode  , id, ePlayMode));  break;  } |

UsbService中在case中调用的该类中的SwitchPlayMode方法

|  |
| --- |
| void GMUsbServiceBase::switchPlayMode(SenderId sid, const GIUsbPlayMode& ePlayMode)  {  // reserved  replySwitchPlayMode(sid, ePlayMode);  //判断资源状态是否可用。  if (GMUsbSourceState\_SourceOn != m\_sourceState) {  USBLOGD("the source is not on, %d", m\_sourceState);  if (NULL != getDPlayer(sid)) {  USBLOGD("MARK DOWN THE PLAYMODE FOR MP BLINKRESET AND STUBBORN UI");  MediaBackup::instance()->updateLastAvMode(pathToDeviceId(getDPlayer(sid)->getRootPath()), static\_cast<int>(ePlayMode));  }  return ;  }  //调用DevicePlayer中的postSwitchPlayMode方法。  if (NULL != getDPlayer(sid)) {  getDPlayer(sid)->postSwitchPlayMode(static\_cast<GMUsbAvMode>(ePlayMode));  }  } |

调DevicePlayer中的postSwitchPlayMode方法。

|  |
| --- |
| void GMUsbDevicePlayer::postSwitchPlayMode(GMUsbAvMode mode)  {  USBLOGD\_FUNCALL;  if (m\_pMPCbThread) {  m\_pMPCbThread->looper().postRunnable(MRunnableTask(this, &GMUsbDevicePlayer::switchPlayMode, mode));  }  else {  switchPlayMode(mode);  }  } |

在devicePlayer类中，postSwitchPlayMode调用该类中的switchPlayMode方法

在该方法的逻辑实现过程中，调用了restorePlay方法，在切换播放模式后恢复播放，恢复失败，则再调用playFirstFolder方法，播放第一个文件夹中的文件

|  |
| --- |
| void GMUsbDevicePlayer::switchPlayMode(GMUsbAvMode mode)  {  USBLOGD\_FUNCALL;  USBLOGD("mode = [%d]", mode);  //调用恢复方法恢复播放。  if (!restorePlay(mode)) {  USBLOGD("restore failed, play first file if exist");  DTLOG\_INFO(0x50002D, "restore failed, play first file if exist");  // playAll(mode, 0);  //恢复播放失败，调用playFirstFolder播放第一个文件夹  playFirstFolder(mode);  }  } |

在switchPlayMode中调用该类中的的恢复播放的方法。目的是在切换模式后还可以恢复切换前的播放状态。

思路：

停止播放，从ＤＢ中取出存储的uid,pos等信息，重新赋给新的模式，然后调用seekTo方法，跳转到指定的位置。

|  |
| --- |
| bool GMUsbDevicePlayer::restorePlay(GMUsbAvMode mode) /\* \_\_0x500051\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  m\_isSeeking = false;  m\_readyToPlay = true;  //判断playinglist是否为空  if (NULL == m\_pNowPlayingList) {  USBLOGD("now play list is null, return");  DTLOG\_INFO(0x500051, "now play list is null, return");  return false;  }  // Gypsophila  //打印DB中存储的备份数据  MediaBackup::instance()->printBackupData();  //设置播放速度为常规速度  m\_ForwardRewind = PlaySpeed\_NormalPlay;  //调用stop方法，停止播放  stop();  //mode为Audio  if (GMUsbAvMode\_Audio == mode) {  m\_mediaStatus.ePlayMode = static\_cast<GIUsbPlayMode>(mode);  //从DB中getPlayUid和播放时间  int *uid* = MediaBackup::instance()->getPlayUid(m\_deviceId, static\_cast<int>(GMUsbAvMode\_Audio));  int pos = MediaBackup::instance()->getPlayTime(m\_deviceId, static\_cast<int>(GMUsbAvMode\_Audio));  USBLOGD("get last audio uid [%d]", uid);  if (m\_pNowPlayingList->load(GMMediaPlaylist::AvMode\_Audio, true)) {  m\_restorePlayFlag = true;  GIUsbPlayMode playMode = static\_cast<GIUsbPlayMode>(mode);  //调用播放模式更新方法，一是在数据库中更新数据，二是通知上层播放模式的改变  playModeUpdated(playMode);  // int repeatMode = MediaBackup::instance()->getRepeatMode(m\_deviceId, static\_cast<int>(mode));  // int randomMode = MediaBackup::instance()->getRandomMode(m\_deviceId, static\_cast<int>(mode));  // m\_pNowPlayingList->setRepeatMode(static\_cast<GMMediaPlaylist::RepeatMode>(repeatMode));  // m\_pNowPlayingList->setRandomMode(static\_cast<GMMediaPlaylist::RandomMode>(randomMode));  从数据库中获取原来的播放模式。  int trickplayMode = MediaBackup::instance()->getTrickPlayMode(m\_deviceId, static\_cast<int>(mode));  //恢复设置  restoreSetting(static\_cast<TrickPlayMode>(trickplayMode));  m\_mediaStatus.itemInfo.position = pos;  m\_pNowPlayingList->selectUid(uid);  #ifdef GMUSB\_PROPERTY\_FAKESEEK  if (GIUsbItemType\_APE == m\_mediaStatus.itemInfo.eItemType) {  play();  }  else {  seekTo(pos);  }  #else  seekTo(pos);  #endif  }  else {  USBLOGW("load audio last info failed");  DTLOG\_INFO(0x500051, "load audio last info failed");  return false;  }  }  else if (GMUsbAvMode\_Video == mode) {  m\_mediaStatus.ePlayMode = static\_cast<GIUsbPlayMode>(mode);  int uid = MediaBackup::instance()->getPlayUid(m\_deviceId, static\_cast<int>(GMUsbAvMode\_Video));  int pos = MediaBackup::instance()->getPlayTime(m\_deviceId, static\_cast<int>(GMUsbAvMode\_Video));  USBLOGD("get last video uid [%d]", uid);  if (m\_pNowPlayingList->load(GMMediaPlaylist::AvMode\_Video, true)) {  m\_restorePlayFlag = true;  GIUsbPlayMode playMode = static\_cast<GIUsbPlayMode>(mode);  playModeUpdated(playMode);  // video do not have repeat&random mode  // int repeatMode = MediaBackup::instance()->getRepeatMode(m\_deviceId, static\_cast<int>(mode));  // int randomMode = MediaBackup::instance()->getRandomMode(m\_deviceId, static\_cast<int>(mode));  // m\_pNowPlayingList->setRepeatMode(static\_cast<GMMediaPlaylist::RepeatMode>(repeatMode));  // m\_pNowPlayingList->setRandomMode(static\_cast<GMMediaPlaylist::RandomMode>(randomMode));  int trickplayMode = MediaBackup::instance()->getTrickPlayMode(m\_deviceId, static\_cast<int>(mode));  restoreSetting(static\_cast<TrickPlayMode>(trickplayMode));  m\_mediaStatus.itemInfo.position = pos;  m\_pNowPlayingList->selectUid(uid);  seekTo(pos);  }  else {  USBLOGW("load video last info failed");  DTLOG\_INFO(0x500051, "load video last info failed");  return false;  }  }  else {  USBLOGD("restore mode error");  DTLOG\_INFO(0x500051, "restore mode error");  return false;  }  return true;  } |

playFirstFolder，先通过getFirstFolderId方法获取第一个folder的id,再通过这个ID调用该类中的playFolder方法，从第一个文件夹开始播放

|  |
| --- |
| void GMUsbDevicePlayer::playFirstFolder(GMUsbAvMode mode)  {  USBLOGD\_FUNCALL;  //得到第一个文件夹的id  unsigned int folderId = getFirstFolderId(mode);  if (0 != folderId) {  //调用该类中的playFolder方法,将第一个folder的ID作为参数传入即可  playFolder(mode, folderId, 0);  }  else {  USBLOGW("get first folder id failed");  DTLOG\_INFO(0x500023, "get first folder id failed");  }  } |

DevicePlayer 中的playFolder方法，在这里调用了PlayingList中的playFolder和save方法。

|  |
| --- |
| void GMUsbDevicePlayer::playFolder(GMUsbAvMode mode, unsigned int folderId,  unsigned int fileIndex) /\* \_\_0x500024\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  //判断m\_pNowPlayingList对象是否创建成功  if (NULL == m\_pNowPlayingList) {  USBLOGD("m\_pNowPlayingList not exist");  DTLOG\_INFO(0x500024, "m\_pNowPlayingList not exis");  return;  }  //设置播放速度为正常。  m\_ForwardRewind = PlaySpeed\_NormalPlay;  //停止  stop();  GIUsbPlayMode playMode = static\_cast<GIUsbPlayMode>(mode);  playModeUpdated(playMode);  //调用m\_pNowPlayingList中的playFoleder方法，实现逻辑  m\_pNowPlayingList->playFolder(static\_cast<GMMediaPlaylist::AvMode>(mode), static\_cast<int>(folderId), static\_cast<int>(fileIndex));  //恢复设置  restoreSetting(TrickPlayMode\_RepAll\_RanOff); // restore default setting  //调用m\_pNowPlayingList保存save  m\_pNowPlayingList->save();  } |

### playFolder

#### 时序

#### 涉及模块

FunctionManager

GMUsbService

DevicePlayer

PlayingList

#### 实现

Proxy

folderId 文件夹编号（数据库中folderID）

fileIndex 起始文件编号（数据库中fileID）

|  |
| --- |
| /// play audio/video of a folder  uint64\_t GMUsbProxy::playFolder(const GIUsbPlayMode& eMode  , const uint32\_t folderId, const uint32\_t fileIndex)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_PlayFolder;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(eMode);  data.writeInt32(folderId);  data.writeInt32(fileIndex);  data.setDataPosition(0);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

UsbServiceBase中case

|  |
| --- |
| case USBMethod\_PlayFolder:  {  // define request param  GIUsbPlayMode eMode;  uint32\_t folderId;  uint32\_t fileIndex;  // unpack request param  eMode = static\_cast<GIUsbPlayMode>(data.readInt32());  folderId = data.readInt32();  fileIndex = data.readInt32();  // call request function  // playFolder(id, eMode, folderId, fileIndex);  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_PlayFolder");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::playFolder  , id, eMode, folderId, fileIndex));  break;  } |

UsbServiceBase中playFolder

|  |
| --- |
| void GMUsbServiceBase::playFolder(SenderId sid, const GIUsbPlayMode& eMode, const uint32\_t folderId , const uint32\_t fileIndex)  {  //通知上层  replyPlayFolder(sid, eMode, folderId, fileIndex);  //判断资源是否打开  if (GMUsbSourceState\_SourceOn != m\_sourceState) {  USBLOGD("the source is not on, %d", m\_sourceState);  DTLOG\_INFO(0xB02010, "tthe source is not on, %d", m\_sourceState);  return ;  }  if (NULL != getDPlayer(sid)) {  getDPlayer(sid)->postPlayFolder(static\_cast<GMUsbAvMode>(eMode), folderId, fileIndex);  }  } |

在DevicePlayer中的postPlayFolder

|  |
| --- |
| void GMUsbDevicePlayer::postPlayFolder(GMUsbAvMode mode, unsigned int folderId, unsigned int fileIndex)  {  USBLOGD\_FUNCALL;  if (m\_pMPCbThread) {  m\_pMPCbThread->looper().postRunnable(MRunnableTask(this, &GMUsbDevicePlayer::playFolder, mode, folderId, fileIndex));  }  else {  playFolder(mode, folderId, fileIndex);  }  } |

在DevicePlayer中的PlayFolder

|  |
| --- |
| void GMUsbDevicePlayer::playFolder(GMUsbAvMode mode, unsigned int folderId, unsigned int fileIndex)  {  USBLOGD\_FUNCALL;  //判断m\_pNowPlayingList对象是否创建成功  if (NULL == m\_pNowPlayingList) {  USBLOGD("m\_pNowPlayingList not exist");  DTLOG\_INFO(0x500024, "m\_pNowPlayingList not exis");  return;  }  //设置播放速度为正常。  m\_ForwardRewind = PlaySpeed\_NormalPlay;  //停止  stop();  GIUsbPlayMode playMode = static\_cast<GIUsbPlayMode>(mode);  playModeUpdated(playMode);  //调用m\_pNowPlayingList中的playFoleder方法，实现逻辑  m\_pNowPlayingList->playFolder(static\_cast<GMMediaPlaylist::AvMode>(mode), static\_cast<int>(folderId), static\_cast<int>(fileIndex));  //恢复设置  restoreSetting(TrickPlayMode\_RepAll\_RanOff); // restore default setting  //调用m\_pNowPlayingList保存save  m\_pNowPlayingList->save();  } |

PlayingList中的playFolder，在这里通过DB查询处结果，再将结果存在list中，从list的第一个文件开始播放。

|  |
| --- |
| bool GMMediaPlaylist::playFolder(AvMode mode, int folderId, int index, bool load)  {  PLLOG\_FUNCALL;  PLLOGD("GMMediaPlaylist::playFolder mode = [%d], folderId = [%d], index = [%d], load [%d]"  , mode, folderId, index, load);  int fileType = 0;  m\_playMode = mode;  if (AvMode\_Audio == mode) {  m\_playlistInfo.filePlayMode = FilePlayMode\_Audio\_Folder;  fileType = 2;  }  else if (AvMode\_Video == mode) {  m\_playlistInfo.filePlayMode = FilePlayMode\_Video\_Folder;  fileType = 4;  }  else {  return false;  }  m\_playlistInfo.folderId = folderId;  m\_playlistInfo.genreId = USB\_INVALID\_ID;  m\_playlistInfo.artistId = USB\_INVALID\_ID;  m\_playlistInfo.composerId = USB\_INVALID\_ID;  m\_playlistInfo.albumId = USB\_INVALID\_ID;  //媒体资源类型，通过srcType方法，获取媒体资源类型  MediaSrcType type = srcType();  if (MediaSrcType\_Any == type) {  PLLOGW("serType error");  return false;  }  //构造查询的参数  NCUri uri;  NCList<NCString> select;  NCString where("");  NCString mkey("");  NCList<NCString> whereArgs;  NCString order("");  int dbStatus = deviceDbInfo(DeviceDbInfo\_Status);  if (AvMode\_Video == mode) {  uri = GIMediaProviderUri::FileBase::getContentUri(type);  select<< new NCString("ID")<< new NCString("FOLDER\_ID")<< new NCString("NAME")<< new NCString("PATH");  where = "TYPE=?";  mkey.format("%d", fileType); // 2 refs audio , 4 refs video  whereArgs.append(&mkey);  order = "FOLDER\_SORTKEY, SORTKEY";  }  else if (4 <= dbStatus) {  PLLOGD("play folder, get file from song table");  uri = GIMediaProviderUri::AudioBase::Songs::getContentUri(type);  select<< new NCString("FID")<< new NCString("FOLDER\_ID")<< new NCString("FILE\_NAME")<< new NCString("FILE\_PATH")  << new NCString("COMPOSER\_ID")<< new NCString("GENRE\_ID")<< new NCString("ARTIST\_ID")<< new NCString("ALBUM\_ID");  order = "FOLDER\_SORTKEY, FILE\_SORTKEY";  }  else {  PLLOGD("play folder, get file from file table");  uri = GIMediaProviderUri::FileBase::getContentUri(type);  select<< new NCString("ID")<< new NCString("FOLDER\_ID")<< new NCString("NAME")<< new NCString("PATH");  where = "TYPE=?";  mkey.format("%d", fileType); // 2 refs audio , 4 refs video  whereArgs.append(&mkey);  order = "FOLDER\_SORTKEY, SORTKEY";  }  #ifndef \_\_LOCAL  //定义结果指针  NCCursor\* cursor = NULL;  cursor = m\_pProviderResolver->query(uri, select, where, whereArgs, order);  if (NULL == cursor) {  PLLOGD("query failed, no data received");  select.clearData();  return false;  }  //获取查询数量  int listCountTmp = cursor->getCount();  PLLOGD("list Count = [%d]", listCountTmp);  //将指针移动到第一个位置  cursor->moveToFirst();  ncsp<NMDataTable>::sp listSp = new NMDataTable(listCountTmp, PlayItem::Column\_Count);  if (NULL != listSp.get()) {  INIT\_TBL(listSp);  for (int i = 0; i < listCountTmp; ++i) {  listSp->putLong(i, PlayItem::Column\_Fid, cursor->getInt64(PlayItem::Column\_Fid));  listSp->putLong(i, PlayItem::Column\_FolderId, cursor->getInt64(PlayItem::Column\_FolderId));  listSp->putString(i, PlayItem::Column\_Name, cursor->getString(PlayItem::Column\_Name).getString(),  cursor->getString(PlayItem::Column\_Name).getLength());  listSp->putString(i, PlayItem::Column\_Path, cursor->getString(PlayItem::Column\_Path).getString(),  cursor->getString(PlayItem::Column\_Path).getLength());  if (AvMode\_Video == mode || (4 > dbStatus)) {  listSp->putLong(i, PlayItem::Column\_ComposerId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_GenreId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_ArtistId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_AlbumId, USB\_INVALID\_ID);  }  else {  listSp->putLong(i, PlayItem::Column\_ComposerId, cursor->getInt64(PlayItem::Column\_ComposerId));  listSp->putLong(i, PlayItem::Column\_GenreId, cursor->getInt64(PlayItem::Column\_GenreId));  listSp->putLong(i, PlayItem::Column\_ArtistId, cursor->getInt64(PlayItem::Column\_ArtistId));  listSp->putLong(i, PlayItem::Column\_AlbumId, cursor->getInt64(PlayItem::Column\_AlbumId));  }  listSp->putLong(i, PlayItem::Column\_FileType, fileType);  cursor->moveToNext();  }  if (load) {  loadInfoList(listSp);  }  selectPlaylist(listSp, index);  // saveLastList();  }  FREEIF(cursor);  #else  NMDataTable\* tmp = GMUsbProvider::instance()->query(uri, select, where, whereArgs, order);  if (NULL != tmp) {  int listCountTmp = tmp->getRowCount();  ncsp<NMDataTable>::sp listSp = new NMDataTable(listCountTmp, PlayItem::Column\_Count);  INIT\_TBL(listSp);  for (int i = 0; i < listCountTmp; ++i) {  listSp->putLong(i, PlayItem::Column\_Fid, tmp->getLong(i, PlayItem::Column\_Fid));  listSp->putLong(i, PlayItem::Column\_FolderId, tmp->getLong(i, PlayItem::Column\_FolderId));  size\_t len;  listSp->putString(i, PlayItem::Column\_Name, tmp->getString(i, PlayItem::Column\_Name, len),  256);  listSp->putString(i, PlayItem::Column\_Path, tmp->getString(i, PlayItem::Column\_Path, len),  256);  if (AvMode\_Video == mode) {  listSp->putLong(i, PlayItem::Column\_ComposerId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_GenreId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_ArtistId, USB\_INVALID\_ID);  listSp->putLong(i, PlayItem::Column\_AlbumId, USB\_INVALID\_ID);  }  else {  listSp->putLong(i, PlayItem::Column\_ComposerId, tmp->getLong(i, PlayItem::Column\_ComposerId));  listSp->putLong(i, PlayItem::Column\_GenreId, tmp->getLong(i, PlayItem::Column\_GenreId));  listSp->putLong(i, PlayItem::Column\_ArtistId, tmp->getLong(i, PlayItem::Column\_ArtistId));  listSp->putLong(i, PlayItem::Column\_AlbumId, tmp->getLong(i, PlayItem::Column\_AlbumId));  }  listSp->putLong(i, PlayItem::Column\_FileType, fileType);  }  if (load) {  loadInfoList(listSp);  }  selectPlaylist(listSp, index);  }  FREEIF(tmp);  #endif  select.clearData();  return true;  } |

#### 常用方法总结

##### 获取指定文件的uri

在usbService中获取指定文件的uri

|  |
| --- |
| uri = GIMediaProviderUri::FileBase::getContentUri(type);  type：MediaSrcType 媒体资源类型 |

在UseService的GIMediaProviderUri接口中的内部类FileBase中定义的方法getContentUri方法，将媒体资源类型类型作为参数传入，就可以得到该资源的uri

##### 查询Query

根据指定的条件查询

|  |
| --- |
| NCCursor\* cursor = NULL;  cursor = m\_pProviderResolver->query(uri, select, where, whereArgs, order); |

|  |
| --- |
| //查询参数定义  NCUri uri;  NCList<NCString> select;  NCString where("");  NCString mkey("");  NCList<NCString> whereArgs;  NCString order("");  //为查询参数赋值  //通过在GIMediaProviderUri::FileBase中封装的getContentUri方法，传入资源类型。返回这个资源的uri  uri = GIMediaProviderUri::FileBase::getContentUri(type);  select<< new NCString("ID")<< new NCString("FOLDER\_ID")<< new NCString("NAME")<< new NCString("PATH");  where = "TYPE=?";  mkey.format("%d", fileType); // 2 refs audio , 4 refs video  whereArgs.append(&mkey);  order = "FOLDER\_SORTKEY, SORTKEY"; |

参数：

Uri：资源的uri，NCUri类型

Select： NCList类型。List，存放查询哪些字段。

Where：NCString类型 。查询条件

whereArgs：NCString类型。查询条件的参数

order：NCString类型。结果的顺序

返回值：

NCCursor:query返回的是一个指向结果的指针cursor，指向查询结果的第一个对象。

### setPlaySpeed

#### 时序

无

#### 涉及模块

FunctionManager

GMUsbService

DevicePlayer

PlayingList

#### 实现

Proxy

|  |
| --- |
| /// set play speed  uint64\_t GMUsbProxy::setPlaySpeed(const int32\_t speed)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_SetPlaySpeed;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(speed);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

UsbService中的case USBMethod\_SetPlaySpeed

|  |
| --- |
| case USBMethod\_SetPlaySpeed:  {  // define request param  int32\_t speed;  // unpack request param  speed = data.readInt32();  // call request function  // stopSlowForward(id);  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_SetPlaySpeed");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::setPlaySpeed  , id, speed));  break;  } |

在usbservice中调用setPlaySpeed方法。

|  |
| --- |
| void GMUsbServiceBase::setPlaySpeed(SenderId sid, const int32\_t speed)  {  replySetPlaySpeed(sid);  if (GMUsbSourceState\_SourceOn != m\_sourceState) {  USBLOGD("the source is not on, %d", m\_sourceState);  DTLOG\_INFO(0xB0201B, "tthe source is not on, %d", m\_sourceState);  return ;  }  if (NULL != getDPlayer(sid)) {  getDPlayer(sid)->postSetPlaySpeed(speed);  }  } |

调用DevicePlayer中的postSetPlaySpeed方法。

|  |
| --- |
| void GMUsbDevicePlayer::postSetPlaySpeed(int speed) /\* \_\_0x50001D\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  if (m\_pMPCbThread) {  m\_pMPCbThread->looper().postRunnable(MRunnableTask(this, &GMUsbDevicePlayer::setPlaySpeed, speed));  }  else {  setPlaySpeed(speed);  }  } |

在postsetPlaySpeed中，又启动了一个线程，调用了给类中的setPlaySpedd方法。

|  |
| --- |
| void GMUsbDevicePlayer::setPlaySpeed(int speed) /\* \_\_0x50003D\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  USBLOGD("setPlaySpeed: current speed is %d", speed);  if (NULL == m\_pMediaPlayer) {  USBLOGD("player not exist");  DTLOG\_INFO(0x50003D, "player not exist");  return;  }  //将从上层传过来的speed赋给全局变量m\_ForwardRewind    m\_ForwardRewind = speed;  #ifdef GMUSB\_PROPERTY\_FAKEOTHERSEEK  pause();  if (PlaySpeed\_NormalPlay != m\_ForwardRewind) {  playStatusUpdated(NMMP\_PlaybackState\_Playing, speed);  }  if (NULL != m\_pTimerHolder) {  if (GIUsbPlayMode\_Video == m\_mediaStatus.ePlayMode) {  m\_pTimerHolder->restartTimer(125);  }  else {  m\_pTimerHolder->restartTimer(100);  }  }  #else  m\_pMediaPlayer->setSpeed(speed);  #endif  } |

在devicePlayer中setPlaySpeed,在这个方法中，调用了plantform层中的setPlaySpeed方法

|  |
| --- |
| void GMUsbDevicePlayer::setPlaySpeed(int speed) /\* \_\_0x50003D\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  USBLOGD("setPlaySpeed: current speed is %d", speed);  if (NULL == m\_pMediaPlayer) {  USBLOGD("player not exist");  DTLOG\_INFO(0x50003D, "player not exist");  return;  }  //将从上层传过来的speed赋给全局变量m\_ForwardRewind    m\_ForwardRewind = speed;  #ifdef GMUSB\_PROPERTY\_FAKEOTHERSEEK  pause();  if (PlaySpeed\_NormalPlay != m\_ForwardRewind) {  playStatusUpdated(NMMP\_PlaybackState\_Playing, speed);  }  if (NULL != m\_pTimerHolder) {  if (GIUsbPlayMode\_Video == m\_mediaStatus.ePlayMode) {  m\_pTimerHolder->restartTimer(125);  }  else {  m\_pTimerHolder->restartTimer(100);  }  }  #else  m\_pMediaPlayer->setSpeed(speed);  #endif  } |

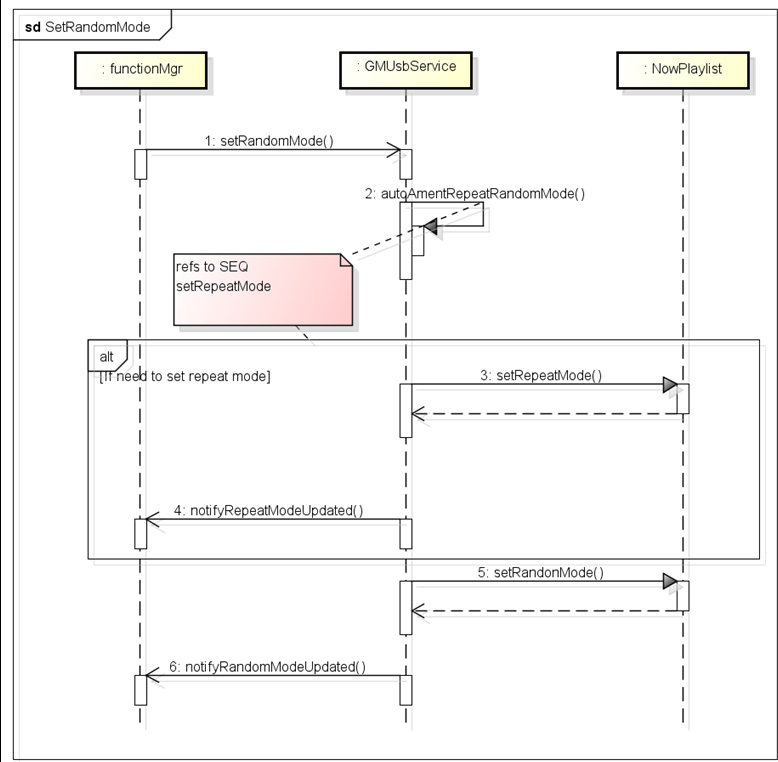
#### 常用方法总结

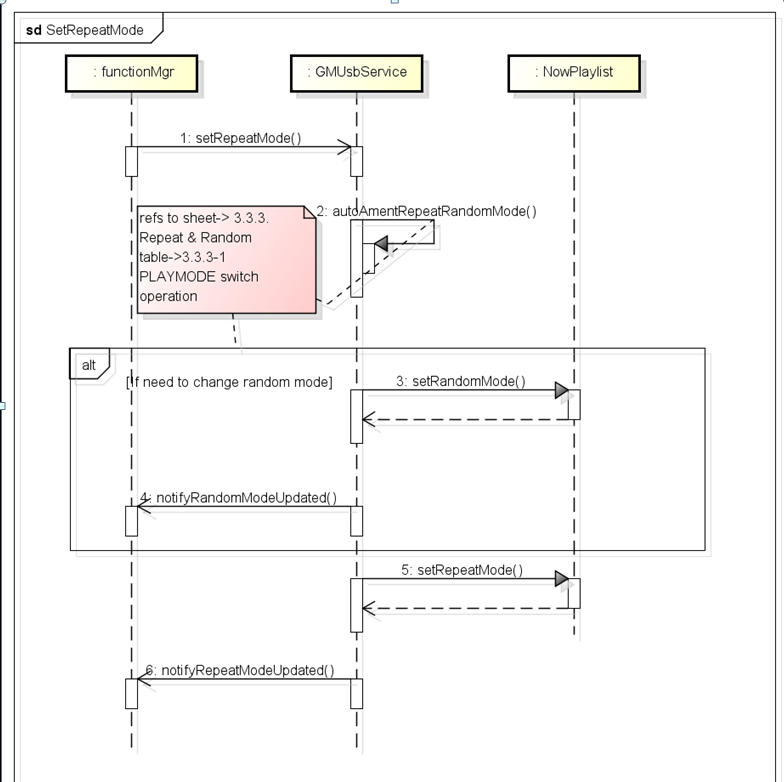
无

### SetRandomMode/SetRandomMode

SetRandomMode和SetRandomMode实现流程一致。

#### 时序





#### 涉及模块

functionManager

UsbService

DevicePlayer

#### 实现

Proxy

|  |
| --- |
| /// set random mode  uint64\_t GMUsbProxy::setRandomMode(const GIUsbRandomMode& eRdmMode)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_SetRandomMode;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(eRdmMode);  data.setDataPosition(0);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

UsbService中的case USBMethod\_SetRandomMode:

在case中启动新的线程，执行setRandomMode方法

|  |
| --- |
| case USBMethod\_SetRandomMode:  {  // define request param  GIUsbRandomMode eRdmMode;  // unpack request param  eRdmMode = static\_cast<GIUsbRandomMode>(data.readInt32());  // call request function  // setRandomMode(id, eRdmMode);  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_SetRandomMode");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::setRandomMode  , id, eRdmMode));  break;  } |

SetRandomMode方法

在该方法中，往上层反馈结果，同时调用DevicePlayer中的postSetRandomMode方法

|  |
| --- |
| void GMUsbServiceBase::setRandomMode(SenderId sid, const GIUsbRandomMode& eRdmMode)  {  replySetRandomMode(sid, eRdmMode);  if (GMUsbSourceState\_SourceOn != m\_sourceState) {  USBLOGD("the source is not on, %d", m\_sourceState);  DTLOG\_INFO(0xB0201D, "tthe source is not on, %d", m\_sourceState);  return ;  }  if (NULL != getDPlayer(sid)) {  getDPlayer(sid)->postSetRandomMode(eRdmMode);  }  } |

postSetRandomMod方法

|  |
| --- |
| void GMUsbDevicePlayer::postSetRandomMode(const GIUsbRandomMode &modeIn) /\* \_\_0x500014\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  if (m\_pMPCbThread) {  m\_pMPCbThread->looper().postRunnable(MRunnableTask(this, &GMUsbDevicePlayer::setRandomMode, modeIn));  }  else {  setRandomMode(modeIn);  }  } |

DevicePlayer中的setRandomMode方法

|  |
| --- |
| void GMUsbDevicePlayer::setRandomMode(const GIUsbRandomMode &modeIn) /\* \_\_0x50002B\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  USBLOGD("setRandomMode: modeIn is %d", modeIn);  // if (NULL == m\_pNowPlayingList) {  // USBLOGD("m\_pNowPlayingList not exist");  // return;  // }  m\_pNowPlayingList->setRandomMode(static\_cast<GMMediaPlaylist::RandomMode>(modeIn));  // randomModeUpdated(modeIn);  autoAmentRepeatRandomMode(RepRanToggle\_Ran);  } |

在DevicePlayer的setRandowMode方法中，调用了autoAmentRandomMode方法。

|  |
| --- |
| // m\_trickPlayMode默认是TrickPlayMode\_RepAll\_RanOff，重复，但不循环  switch (m\_trickPlayMode) {  //默认的播放模式，repeatall，不循环  case TrickPlayMode\_RepAll\_RanOff: // state 1  {  if (RepRanToggle\_Rep == repOrRan) {  m\_trickPlayMode = TrickPlayMode\_RepFld\_RanOff; // state 2  m\_pNowPlayingList->setRepeatMode(GMMediaPlaylist::RepeatMode\_List);  repeatModeUpdated(GIUsbRepeatMode\_List);  }  else if (RepRanToggle\_Ran == repOrRan){  m\_trickPlayMode = TrickPlayMode\_RepAll\_RanAll; // state 4  m\_pNowPlayingList->setRandomMode(GMMediaPlaylist::RandomMode\_All);  randomModeUpdated(GIUsbRandomMode\_All);  }  else {  }  break;  } |

在上面的方法中，调用了PlayingList中的setRandomMode方法

|  |
| --- |
| void GMMediaPlaylist::setRandomMode(RandomMode randomMode)  {  PLLOGD("setRandomMode = [%d]", randomMode);  m\_randomMode = randomMode;  m\_errorCountInList = 0;  m\_listError = false;  //真正的逻辑  if (RandomMode\_All == randomMode || RandomMode\_List == randomMode) {  //随机打乱播放顺序  random\_shuffle(m\_random.begin(), m\_random.end());  //循环list  for (unsigned int i = 0; i < m\_random.size(); ++i) {  if (NULL != m\_random[i].get()) {  if (m\_originIndex == m\_random[i]->index()) { // index in all list  m\_randomIndex = i;  m\_random[i]->setPlayed(true);  }  else {  m\_random[i]->setPlayed(false);  }  }  }  }  amendIndexAndCount();  updatePlaylistInfo();  } |

PlayList 中的setRepeatMode方法

|  |
| --- |
| void GMMediaPlaylist::setRepeatMode(RepeatMode repeatMode)  {  PLLOGD("setRepeatMode = [%d]", repeatMode);  m\_repeatMode = repeatMode;  m\_errorCountInList = 0;  m\_listError = false;  amendIndexAndCount();  updatePlaylistInfo();  } |

#### 常用方法总结

### setWideMode

设置全屏显示

#### 时序

无

#### 涉及模块

FunctionManager

UsbService

#### 实现

Proxy

|  |
| --- |
| /// set wide mode  uint64\_t GMUsbProxy::setWideMode(const GIUsbWideMode& eWideMode)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_SetWideMode;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(eWideMode);  data.setDataPosition(0);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

UsbService中case中接收请求

|  |
| --- |
| case USBMethod\_SetWideMode:  {  // define request param  GIUsbWideMode eWideMode;  // unpack request param  eWideMode = static\_cast<GIUsbWideMode>(data.readInt32());  // call request function  // setWideMode(id, eWideMode);  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_SetWideMode");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::setWideMode  , id, eWideMode));  break;  } |

起一个新的线程，调用本类中的setWideMode方法。

在这个方法中，只是往上层反馈消息，并没有继续往下层调用处理

|  |
| --- |
| void GMUsbServiceBase::setWideMode(SenderId sid, const GIUsbWideMode& eWideMode) /\* \_\_0xB0201E\_DTFUNCID\_\_ \*/  {  // reserved  replySetWideMode(sid, eWideMode);  } |

replySetWideMode方法

往上层发送请求，反馈service端的处理结果

|  |
| --- |
| void GMUsbServiceBase::replySetWideMode(SenderId sid, const GIUsbWideMode& eWideMode)  {  USBLOGD\_FUNCALL;  android::Parcel data;  unsigned int code = USBMethod\_SetWideMode;  data.writeInt32(eWideMode);  data.setDataPosition(0);  sendAsyncResponse(sid, code, data);  } |

在proxy的onAsyncResponse方法的case中接受请求

调用functionManager中的反馈接口中的replySetWideMode方法。

|  |
| --- |
| case USBMethod\_SetWideMode:  {  // define reply param  uint64\_t call\_id = 0;  GIUsbWideMode eWideMode = static\_cast<GIUsbWideMode>(reply.readInt32());  m\_spReply->replySetWideMode(eWideMode, call\_id);  }  break |

#### 常用方法总结

无

### RequestStatus

#### 时序

#### 涉及模块

FunctionManager

UsbService

DevicePlayer

#### 实现

Proxy

|  |
| --- |
| /// request binded source status  uint64\_t GMUsbProxy::requestStatus(const uint32\_t notificationMask)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_RequestStatus;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(notificationMask);  data.setDataPosition(0);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

在USBservice中的接受请求

在这里调用给类中的RequestStatus方法，处理逻辑

|  |
| --- |
| }  case USBMethod\_RequestStatus:  {  // define request param  uint32\_t notificationMask;  // unpack request param  notificationMask = data.readInt32();  // call request function  // requestStatus(id, notificationMask);  //通过传过来的notificationMask参数和0x702、0x703参数比较。确定是否销毁或者清空线程。  if (0x702 == notificationMask) {  m\_pMainThread->looper().dumpRunnable();  break;  }  else if (0x703 == notificationMask) {  m\_pMainThread->looper().clearRunnable();  break;  }  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_RequestStatus");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::requestStatus  , id, notificationMask));  break;  } |

RequestStatus方法

|  |
| --- |
| void GMUsbServiceBase::requestStatus(SenderId sid, const uint32\_t notificationMask)  {  UNUSED(notificationMask);  //定义GIUsbServiceStatus,usbservice的所有相关状态  GIUsbServiceStatus statusInfo;  //调用getDPlayer方法，获得DevicePlayer对象，再调用其中的mediaStatus方法。  if (NULL != getDPlayer(sid)) {  statusInfo = getDPlayer(sid)->mediaStatus();  }  //往上层反馈，将GIUsbServiceStatus 这个包含了usbService中所有状态的结构体返回给调用方  replyRequestStatus(sid, statusInfo);  } |

DevicePlayer对象中的mediaStatus方法

|  |
| --- |
| //将 GIUsbServiceStatus返回给调用方。  const GIUsbServiceStatus& mediaStatus()  {  return m\_mediaStatus;  } |

在functionManager中接收到service端返回给他的GIUsbServiceStatus数据，根据需求进行解析。并将解析出的结果封装在message中发送给上层。

#### 常用方法总结

### SelectUID

方法说明

selectUid，通过播放想要播放的曲目的id。选择 播放的音屏

#### 时序

#### 涉及模块

functionManager

UsbService

DevicePlayer

PlayingList

MediaProvider

#### 实现

Proxy

|  |
| --- |
| /// select file play by UID  //选择UId选择需要播放的文件  uint64\_t GMUsbProxy::selectUID(const uint32\_t uid, const bool isFastFileChange)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_SelectUID;  android::Parcel data;  prepareAsyncData(data);  data.writeInt32(uid);  data.writeInt32(isFastFileChange);  data.setDataPosition(0);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

Usb Service中的Case

|  |
| --- |
| case USBMethod\_SelectUID:  {  // define request param  uint32\_t uid;  bool isFastFileChange;  // unpack request param  uid = data.readInt32();  isFastFileChange = data.readInt32();  // call request function  // selectUID(id, uid);  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_SelectUID");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::selectUID  , id, uid, isFastFileChange));  break;  } |

经过层层调用，先是调用了DevicePalyer中的selectUID方法

|  |
| --- |
| void GMUsbDevicePlayer::selectUID(uint32\_t uid, bool isFastFileChange) /\* \_\_0x500021\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  m\_readyToPlay = true;  if (NULL == m\_pNowPlayingList) {  DTLOG\_INFO(0x500021, "m\_pNowPlayingList not exis");  return;  }  //通过传入的参数判断。为false使，调用autoAmentRepeatRandomMode方法。  if(!isFastFileChange) {  autoAmentRepeatRandomMode(RepRanToggle\_Clear);  }  stop();  m\_pNowPlayingList->selectUid(uid);  } |

最终调用到了PlayingList中的selectUID方法

|  |
| --- |
| bool GMMediaPlaylist::selectUid(unsigned int uid)  {  PLLOGD("selectUid : uid = [%d]", uid);  //判断原始文件list是否为空，为空返回false  if (m\_origin.empty()) {  PLLOGW("list is empty, return false");  return false;  }  if (NULL == m\_cb || DeviceType\_None >= m\_deviceInfo.type() || DeviceType\_Count <= m\_deviceInfo.type()) {  PLLOGW("selectUid : param error, m\_cb = [%p], type = [%d]", m\_cb, m\_deviceInfo.type());  return false;  }  if (m\_totalError) {  PLLOGW("warning : selectUid !!totalError!!");  m\_cb->onPlaylistEnded(EndedType\_AllError);  return true;  }  //通过循环的方式，设置originIndex和传过来的uid相同。同时设置index文件为可播放状态。  unsigned int count = m\_origin.size();  for (m\_originIndex = 0; m\_originIndex < count; ++m\_originIndex) {  if (NULL != m\_origin[m\_originIndex].get() && m\_origin[m\_originIndex]->fid() == static\_cast<int>(uid)) {  setPlayable(m\_origin[m\_originIndex]->index(), true);  break;  }  }  PLLOGD("debug : m\_listIndex = [%d], m\_originIndex = [%d]", m\_listIndex, m\_originIndex);  if (m\_originIndex >= count) {  PLLOGD("song not found , play the first one");  m\_originIndex = 0;  }  // keep the m\_randomIndex point to the current item  //设置循环的索引指针和当前Item相同。目的是维持两个表指针同步  count = m\_random.size();  for (unsigned int i = 0; i < count; ++i) {  if (NULL != m\_random[i].get() && m\_random[i]->index() == m\_originIndex) {  m\_randomIndex = i;  break;  }  }  //将随机表中的现在的index对应的item设置为可播放  if (NULL != m\_random[m\_randomIndex].get()) {  m\_random[m\_randomIndex]->setPlayed(true);  }  if (NULL != m\_origin[m\_originIndex].get()) {  amendListInfoIndex();  updatePlaylistInfo();  m\_cb->onPlayItemChanged(m\_origin[m\_originIndex]);  }  m\_errorCountInList = 0;  return true;  } |

在PlayingList中的selectUID方法中，最终将通过UID查询出的结果，通过callBack返回给调用方DevicePlayer。在DevicePlayer中，根据返回的结果对一系列的值进行处理。

#### 常用方法总结

### ReqSourceChange

#### 方法说明

ReqSourceChange方法，请求改变资源。资源的状态有

GIUsbSourceChangeMode\_On----打开播放器

GIUsbSourceChangeMod\_Off----关闭播放器

GIUsbSourceChangeMode\_DKStop-----暂停播放

GIUsbSourceChangeMode\_DKStopOff----结束暂停，继续播放

GIUsbSourceChangeMode\_OnWithMiracast----通过Miracast的方式打开播放器

等多种方式。根据上层传过来的请求改变的状态，进行对应的逻辑处理。

该方法的主要作用是，当播放音视频时，打开、关闭播放器，点击暂停按钮，播放按钮等改变状态的情景时，会调用该方法。

#### 时序

暂无

#### 涉及模块

functionManger

UsbService

DevicePlayer

#### 实现

在proxy中的reqSourceChange

|  |
| --- |
| uint64\_t GMUsbProxy::reqSourceChange(const GIUsbSourceChangeInfo& info)  {  USBLOGD\_FUNCALL;  static uint64\_t call\_id = 1;  unsigned int code = USBMethod\_ReqSourceChange;  android::Parcel data;  prepareAsyncData(data);  info.writeToParcel(data);  data.setDataPosition(0);  // request service  if (BS\_NO\_CONNECTION == sendAsyncRequest(code, data)) {  return 0; // invalid call id '0' to indicate error  }  return call\_id++;  } |

在service中接收请求

|  |
| --- |
| case USBMethod\_ReqSourceChange:  {  // define request param  GIUsbSourceChangeInfo info;  // unpack request param  info.readFromParcel(data);  // call request function  // selectUID(id, uid);  GMTaskDes des(Runnable\_Pri\_N, 0, "USBMethod\_ReqSourceChange");  (m\_pMainThread->looper()).postRunnable(GMTaskBind(des, this, &GMUsbServiceBase::reqSourceChange  , id, info));  break;  } |

调用该类中的reqSourceChange方法.。在这个方法中，通过if else的方式，对所有的可能改变的资源进行了判断。

|  |
| --- |
| void GMUsbServiceBase::reqSourceChange(SenderId sid, const GIUsbSourceChangeInfo& info)  {  //通过ClientMgr获取设备的ID  int deviceId = m\_spClientMgr->getDeviceId(sid);  if (GIUsbSourceChangeMode\_On == info.eMode) {  USBLOGD("the value of info.isLastSource is %d",info.isLastSource);  //尝试打开资源 。在baseService中 没有实现，在service中做了具体实现。  trySourceOn(deviceId, info.isLastSource, info.screen);  }  else if (GIUsbSourceChangeMode\_Off == info.eMode) {  trySourceOff(deviceId, info.screen);  }  //暂停  else if (GIUsbSourceChangeMode\_DKStop == info.eMode) {  USBLOGD("TODO:");  if (NULL != getDPlayer(sid)) {  当状态为DKStop，先调用DevicePlayer中的方法暂停播放。  getDPlayer(sid)->pause();  }  GIUsbSourceChangeInfo result;  result.isLastSource = false;  result.eMode = GIUsbSourceChangeMode\_DKStop;  result.eResult = GIUsbSourceChangeRlt\_Succeed;  result.screen = 0;  notifySourceChanged(sid, result);  }  //暂停结束。开始播放  else if (GIUsbSourceChangeMode\_DKStopOff == info.eMode) {  USBLOGD("TODO : maybe need to record last player state");  GMUsbDevicePlayer\* DKdevicePlayer = getDPlayer(sid);  if (NULL != DKdevicePlayer) {  GIUsbServiceStatus status = DKdevicePlayer->mediaStatus();  DKdevicePlayer->play();  DKdevicePlayer->playItemUpdated(status.itemInfo);  DKdevicePlayer->metaDataUpdated(status.metaInfo);  }  GIUsbSourceChangeInfo result;  result.isLastSource = false;  result.eMode = GIUsbSourceChangeMode\_DKStopOff;  result.eResult = GIUsbSourceChangeRlt\_Succeed;  result.screen = 0;  //往上层发送通知  notifySourceChanged(sid, result);  }  //the follow case is designed for the miracast source change to usb  else if (GIUsbSourceChangeMode\_OnWithMiracast == info.eMode) {  USBLOGD("TODO : the source from miracast to usb");  if (NULL != getDPlayer(sid)) {  getDPlayer(sid)->setMiracastSourceValue(true);  }  trySourceOn(deviceId, info.isLastSource, info.screen);  }  } |

U盘插入

#### 时序

#### 涉及模块

connectManager

UsbService

ContentProvider

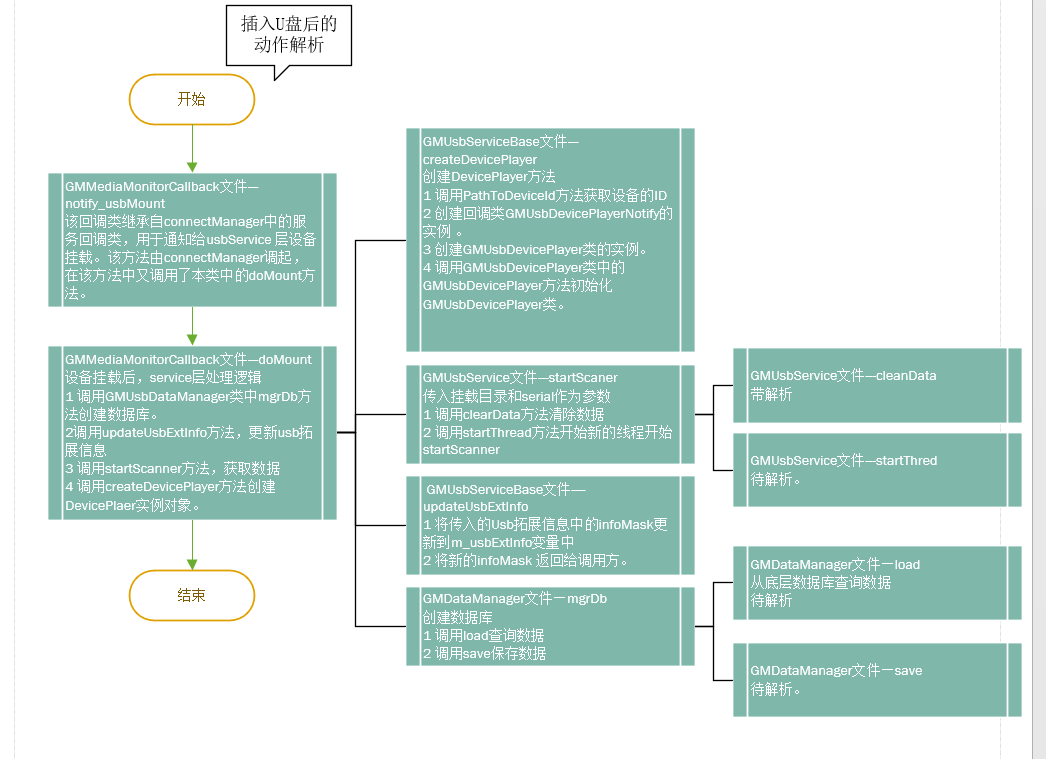
MediaPlayer

#### 实现

### U盘插入

#### 时序

#### 执行流程



#### 涉及模块

UsbService

connectManager

DevicePlayer

DataManager

#### 实现

1 在UsbService.cpp文件的GMMediaMonitorCallback类调用notify\_usbMount方法，该方法由connectManager调起。

在该方法中，将下层传来的设备信息传递给doMount方法处理。

|  |
| --- |
| android::status\_t GMMediaMonitorCallback::notify\_usbMount(const StorageInfo\_Sp & info)  {  USBLOGD\_FUNCALL;  DTLOG\_INFO(0xB01002, "notify\_usbMount");  GMTaskDes des(Runnable\_Pri\_N, 0, "notify\_usbMount");  //doOnMout方法是在usbservicec层中的继承下层的connectManager中的代理replier的基础上新增加的类  //目的是对下层通过回调函数反馈上来的信息。  m\_mainLoop.postRunnable(GMTaskBind(des, this, &GMMediaMonitorCallback::doOnMount, info));  return 0;  } |

2 在doMount方法中处理设备挂载后在service层的逻辑

主要是：

1 创建数据库

2 更新usb拓展信息，并反馈给上层

3 startScanner

4 创建DevicePlayer

|  |
| --- |
| void GMMediaMonitorCallback::doOnMount(const StorageInfo\_Sp & info) /\* \_\_0xB0100A\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  m\_state = "mounted";  StorageInfo\_Sp srgInfo = info;  //serial：设备的序列号，也是设备的唯一标识。  std::string serial = "";  std::string volumeLabel = "";  if (NULL != srgInfo.get()) {  //获取从下层传上来的存储设备的信息。  //获取挂载路径  m\_rootPath = info->getMountPath().string();  //获取卷标  volumeLabel = info->getVolumeLabel().string();  USBLOGD("usb mount path is [%s]", m\_rootPath.c\_str());  USBLOGD("usb volumeLabel is [%s]", volumeLabel.c\_str());  //获取Usb设备的信息。  UsbInfo\_Sp usbInfo = srgInfo->getUsbInfo();  if (NULL != usbInfo.get()) {  //先从参数StorageInfo类中获取成员类UsbInfo,再从UsbInfo中获取u盘序列号  serial = usbInfo->getSerialNum().string();  //打印U盘信息  /\*  device port is [0]  device id is [0]  product id is [6387]  vendor id is [058f]  serial number is [BE0DEE3A]  \*/  USBLOGD("is unsupportDevice ? [%d]", usbInfo->getUnsupportDevice());  USBLOGD("device port is [%d]", usbInfo->getPort());  USBLOGD("device id is [%d]", usbInfo->getDeviceId());  USBLOGD("product id is [%s]", usbInfo->getPid().string());  USBLOGD("vendor id is [%s]", usbInfo->getVid().string());  USBLOGD("serial number is [%s]", usbInfo->getSerialNum().string());  }  }  else {  return;  }  // create uuid and dbpath  if (serial.empty()) {  serial = GMUSB\_DEFAULT\_UUID;  }  else {  //如果serial不为空，则取前16位  serial = serial.substr(0, GMUSB\_UUID\_LENGTH);  }  char dbpath[64];  //将dbpath的64位全部置零。  memset(dbpath, 0, 64);  //c内部函数将GM\_USB\_DBPATH"%s"GM\_USB\_DBSUFFIX代表字符串存入dbpath中。  snprintf(dbpath, 64, GM\_USB\_DBPATH"%s"GM\_USB\_DBSUFFIX, serial.c\_str());  // create uuid = [BE0DEE3A], dbpath = [/var/user/media/usb/BE0DEE3A.db 打印的log  USBLOGD("create uuid = [%s], dbpath = [%-64s]", serial.c\_str(), dbpath);  //通过下层传递过来的信息，调用UsbDataManager中的方法管理数据库。  GMUsbDataManager::ins()->mgrDb(serial, dbpath, false);  //============================================================  //上面的目的是通过传过来的设备信息，先生成调用mgrDb方法生成数据库。  android::sp<GMUsbService> sp = m\_wpService.promote();  if (NULL != sp.get()) {  ///< notify extend info  GIUsbExtendInfo extendInfo;  extendInfo.infoMask = 0x0a;  extendInfo.isUsbMounted = 1;  // 1 is source on with usb  USBLOGD("the value of the islaston is %d", info->getIsLastOn());  extendInfo.islaston = 1;  if (!info->getIsLastOn()) {  extendInfo.islaston = 0;  }  //更新usb拓展信息  /\*updateUsbExtInfo函数参数：  const int &deviceId:设备id  const GIUsbExtendInfo &info：usb拓展信息。  \*/  sp->updateUsbExtInfo(0, extendInfo);  //usb挂载目录 [/mnt/udisk]  //serial :BE0DEE3A  //根据传入的参数值，可以scanner  sp->startScanner(m\_rootPath, serial);  //创建DevicePlayer，传入设备挂载路径和卷标  sp->createDevicePlayer(m\_rootPath, volumeLabel);  DTLOG\_INFO(0xB0100A, "usb mountPath is %s, volumeLabel is %s, serial is %s, dbpath is %s",  m\_rootPath.c\_str(), volumeLabel.c\_str(), serial.c\_str(), dbpath);  }  return;  } |

2.1 创建数据库

在doMount方法中，调用了GMDataManager类中的mgrDb方法创建数据库。

参入参数：mgrDb(std::string uuid, std::string dbPath, bool isEject)

Uuid：设备的序列号

DbPath：数据库的存放位置

isEject:不明

在该方法中调用了load方法从底层数据库中加载数据

调用了save方法保存数据

|  |
| --- |
| bool GMUsbDataManager::mgrDb(std::string uuid, std::string dbPath, bool isEject)  {  MLLOG\_FUNCALL;  //这啥？  NCAutoSync autoSync(m\_innerSync);  MLLOGD("uuid = [%s], dbPath = [%s], isEject = [%d]", uuid.c\_str(), dbPath.c\_str(), isEject);  //GMUsbInfo\_DbMgr ：在dataManager文件中封装的一个存放创建数据库所需usb设备的部分信息。  GMUsbInfo\_DbMgr dbStatus[GMUSB\_MAXDBCOUNT];  memset(dbStatus, 0x00, sizeof(dbStatus));  //DBMGR\_KEY,在执行恢复出厂设置时。  //调用load方法。执行查询操作。将数据查询出来，成功则返回1  bool ret = load(DBMGR\_KEY, dbStatus, sizeof(dbStatus));    MLLOGD("load db status ret = [%d]", ret);  if (!ret) {  for (unsigned int i = 0; i < GMUSB\_MAXDBCOUNT; ++i) {  dbStatus[i].ejectTime = i + 1;  }  }  for (int i = 0; i < GMUSB\_MAXDBCOUNT; ++i) {  std::string uidTmp = dbStatus[i].uuid;  MLLOGD("dbStatus[%d].uuid = [%s], isEject = [%d], ejectTime = [%u]",  i, uidTmp.c\_str(), dbStatus[i].isEject, dbStatus[i].ejectTime);  }  static bool firstCall = false;  if (!firstCall) {  firstCall = true;  unsigned int maxTime = 1;  for (int i = 0; i < GMUSB\_MAXDBCOUNT; ++i) {  if (maxTime < dbStatus[i].ejectTime) {  maxTime = dbStatus[i].ejectTime;  }  }  for (int i = 0; i < GMUSB\_MAXDBCOUNT; ++i) {  if (0 == dbStatus[i].ejectTime) {  dbStatus[i].ejectTime = ++maxTime;  }  }  }  if (isEject) {  int ejectIdx = -1;  unsigned int maxTime = 1;  for (int i = 0; i < GMUSB\_MAXDBCOUNT; ++i) {  std::string tmp = dbStatus[i].uuid;  if ((-1 == ejectIdx) && (tmp == uuid)) {  ejectIdx = i;  }  if (maxTime < dbStatus[i].ejectTime) {  maxTime = dbStatus[i].ejectTime;  }  }  if (ejectIdx != -1) {  dbStatus[ejectIdx].ejectTime = maxTime + 1;  }  else {  MLLOGE("eject device not exist");  return false;  }  }  else {  int i = 0;  int replace = 0;  unsigned int minTime = 0xffffffff;  for (i = 0; i < GMUSB\_MAXDBCOUNT; ++i) {  std::string tmp = dbStatus[i].uuid;  if (tmp == uuid) {  dbStatus[i].ejectTime = 0;  break;  }  if (dbStatus[i].ejectTime != 0 && dbStatus[i].ejectTime < minTime) {  replace = i;  minTime = dbStatus[i].ejectTime;  }  }  if (i == GMUSB\_MAXDBCOUNT) {  ::remove(dbStatus[replace].path);  MLLOGD("remove database situation [%s]", dbStatus[replace].path);  strncpy(dbStatus[replace].path, dbPath.c\_str(), 64);  strncpy(dbStatus[replace].uuid, uuid.c\_str(), 32);  dbStatus[replace].ejectTime = 0;  }  }  ret = save(DBMGR\_KEY, dbStatus, sizeof(dbStatus));  MLLOGD("save db status ret = [%d]", ret);  return ret;  } |

2．2 更新usb拓展信息upateUsbExtInfo

该方法主要

一 更新usb拓展信息

二 将信息反馈给上层

|  |
| --- |
| void GMUsbServiceBase::updateUsbExtInfo(const int &deviceId, const GIUsbExtendInfo &info) /\* \_\_0xB02025\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  // save updated mask for FC, restore after notify current infomask.  uint32\_t infoMaskTmp = m\_usbExtInfo[deviceId].infoMask | info.infoMask;  m\_usbExtInfo[deviceId].infoMask = info.infoMask;  if (0x01 & info.infoMask) {  m\_usbExtInfo[deviceId].isHubInserted = info.isHubInserted;  }  if (0x02 & info.infoMask) {  m\_usbExtInfo[deviceId].isUsbMounted = info.isUsbMounted;  }  if (0x04 & info.infoMask) {  m\_usbExtInfo[deviceId].isSourceValid = info.isSourceValid;  }  if (0x08 & info.infoMask) {  m\_usbExtInfo[deviceId].islaston = info.islaston;  }  if (0x10 & info.infoMask) {  m\_usbExtInfo[deviceId].isSameUSB = info.isSameUSB;  }  if (0x20 & info.infoMask) {  m\_usbExtInfo[deviceId].usbStatus = info.usbStatus;  }  std::vector<int> sdlist = m\_spClientMgr->getSenderId(deviceId);  for (unsigned int i = 0; i < sdlist.size(); ++i) {  //通知给上层拓展信息。  notifyUsbExtInfo(sdlist[i], m\_usbExtInfo[deviceId]);  }  m\_usbExtInfo[deviceId].infoMask = infoMaskTmp;  } |

2.3 startScaner 开始浏览数据

|  |
| --- |
| void GMUsbService::startScanner(const std::string& root, const std::string& serial) /\* \_\_0xB01023\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  //m\_pScanner在GMUsbService类的构造函数中实例化。  m\_pScanner->startScan(root, serial);  }  } |

2.4 创建GMDevicePlayer

主要是创建两个类：GMDevicePlayerNotify和GMUsbDevicePlayer

|  |
| --- |
| bool GMUsbServiceBase::createDevicePlayer(const std::string& rootPath, std::string volumeLabel) /\* \_\_0xB02006\_DTFUNCID\_\_ \*/  {  USBLOGD\_FUNCALL;  USBLOGD("rootPath = [%s]", rootPath.c\_str());  //调用pathToDeviceId方法，通过挂载路径获取DeviceID  GMUsbDeviceId deviceId = pathToDeviceId(rootPath);  if (NULL == m\_pDevicePlayer[deviceId]) {  if (NULL == m\_pDPCB[deviceId]) {  //创建GMDevicePlayerNotify，回调类。  m\_pDPCB[deviceId] = new GMUsbDevicePlayerNotify(this, deviceId);  }  USBLOGD("create device player [%d]", deviceId);  //创建GMDevicePlayer对象。传入构造参数。  m\_pDevicePlayer[deviceId] = new GMUsbDevicePlayer(deviceId, rootPath, m\_pDPCB[deviceId], volumeLabel);  //返回初始化后的GMDevicePlayer实例对象。  return m\_pDevicePlayer[deviceId]->initialize();  }  return true;  } |

#### 常用方法总结

### U盘拔出

#### 时序

#### 涉及模块

#### 实现

#### 常用方法总结

### 切到UsbSource

#### 时序

#### 执行流程

#### 涉及模块

#### 实现

#### 常用方法总结

========================================

### 模版

#### 时序

#### 涉及模块

#### 实现

#### 常用方法总结