

**QuePython云喇叭 软件设计文档**

**LTE Standard/LPWA模块系列**

版本：1.0.0

日期：2023-02-27

 状态：临时文件

上海移远通信技术股份有限公司（以下简称“移远通信”）始终以为客户提供最及时、最全面的服务为宗旨。如需任何帮助，请随时联系我司上海总部，联系方式如下：

上海移远通信技术股份有限公司

上海市闵行区田林路1016号科技绿洲3期（B区）5号楼 邮编：200233

电话：+86 21 5108 6236 邮箱：[info@quectel.com](mailto:info@quectel.com)

或联系我司当地办事处，详情请登录：<http://www.quectel.com/cn/support/sales.htm>。

如需技术支持或反馈我司技术文档中的问题，请随时登陆网址：

<http://www.quectel.com/cn/support/technical.htm> 或发送邮件至：[support@quectel.com](mailto:support@quectel.com)。

**前言**

移远通信提供该文档内容以支持客户的产品设计。客户须按照文档中提供的规范、参数来设计产品。同时，您理解并同意，移远通信提供的参考设计仅作为示例。您同意在设计您目标产品时使用您独立的分析、评估和判断。在使用本文档所指导的任何硬软件或服务之前，请仔细阅读本声明。您在此承认并同意，尽管移远通信采取了商业范围内的合理努力来提供尽可能好的体验，但本文档和其所涉及服务是在“可用”基础上提供给您的。移远通信可在未事先通知的情况下，自行决定随时增加、修改或重述本文档。

**使用和披露限制**

**许可协议**

除非移远通信特别授权，否则我司所提供硬软件、材料和文档的接收方须对接收的内容保密，不得将其用于除本项目的实施与开展以外的任何其他目的。

**版权声明**

移远通信产品和本协议项下的第三方产品可能包含受移远通信或第三方材料、硬软件和文档版权保护的相关资料。除非事先得到书面同意，否则您不得获取、使用、向第三方披露我司所提供的文档和信息，或对此类受版权保护的资料进行复制、转载、抄袭、出版、展示、翻译、分发、合并、修改，或创造其衍生作品。移远通信或第三方对受版权保护的资料拥有专有权，不授予或转让任何专利、版权、商标或服务商标权的许可。为避免歧义，除了正常的非独家、免版税的产品使用许可，任何形式的购买都不可被视为授予许可。对于任何违反保密义务、未经授权使用或以其他非法形式恶意使用所述文档和信息的违法侵权行为，移远通信有权追究法律责任。

**商标**

除另行规定，本文档中的任何内容均不授予在广告、宣传或其他方面使用移远通信或第三方的任何商标、商号及名称，或其缩略语，或其仿冒品的权利。

**第三方权利**

您理解本文档可能涉及一个或多个属于第三方的硬软件和文档（“第三方材料”）。您对此类第三方材料的使用应受本文档的所有限制和义务约束。

移远通信针对第三方材料不做任何明示或暗示的保证或陈述，包括但不限于任何暗示或法定的适销性或特定用途的适用性、平静受益权、系统集成、信息准确性以及与许可技术或被许可人使用许可技术相关的不侵犯任何第三方知识产权的保证。本协议中的任何内容都不构成移远通信对任何移远通信产品或任何其他硬软件、设备、工具、信息或产品的开发、增强、修改、分销、营销、销售、提供销售或以其他方式维持生产的陈述或保证。此外，移远通信免除因交易过程、使用或贸易而产生的任何和所有保证。

**隐私声明**

为实现移远通信产品功能，特定设备数据将会上传至移远通信或第三方服务器（包括运营商、芯片供应商或您指定的服务器）。移远通信严格遵守相关法律法规，仅为实现产品功能之目的或在适用法律允许的情况下保留、使用、披露或以其他方式处理相关数据。当您与第三方进行数据交互前，请自行了解其隐私保护和数据安全政策。

**免责声明**

1. 移远通信不承担任何因未能遵守有关操作或设计规范而造成损害的责任。
2. 移远通信不承担因本文档中的任何因不准确、遗漏、或使用本文档中的信息而产生的任何责任。
3. 移远通信尽力确保开发中功能的完整性、准确性、及时性，但不排除上述功能错误或遗漏的可能。除非另有协议规定，否则移远通信对开发中功能的使用不做任何暗示或法定的保证。在适用法律允许的最大范围内，移远通信不对任何因使用开发中功能而遭受的损害承担责任，无论此类损害是否可以预见。
4. 移远通信对第三方网站及第三方资源的信息、内容、广告、商业报价、产品、服务和材料的可访问性、安全性、准确性、可用性、合法性和完整性不承担任何法律责任。

版权所有 ©上海移远通信技术股份有限公司 2023，保留一切权利。

***Copyright © Quectel Wireless Solutions Co., Ltd. 2023.***

# 文档历史

**修订记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **日期** | **作者** | **变更表述** |
| - | 2023-02-27 | Pawn ZHOU | 文档创建 |
| 1.0.0 | 2023-02-27 | Pawn ZHOU | 临时版本 |

# 目录

[文档历史 3](#_Toc128236513)

[目录 4](#_Toc128236514)

[表格索引 5](#_Toc128236515)

[图片索引 6](#_Toc128236516)

[1 前言 7](#_Toc128236517)

[1.1. 适用模块 7](#_Toc128236518)

[2 系统框架 8](#_Toc128236519)

[2.1. 硬件系统框架 8](#_Toc128236522)

[2.2. 软件系统框架 8](#_Toc128236523)

[3 关键组件 10](#_Toc128236524)

[3.1. EventMesh 12](#_Toc128236525)

[3.2. AudioManager 13](#_Toc128236528)

[3.3. ConfigStoreManager 15](#_Toc128236529)

[3.4. LteNetManager 16](#_Toc128236530)

[3.5. DeviceInfoManager 19](#_Toc128236531)

[3.6. OtaManager 19](#_Toc128236532)

[3.7. AliYunManage 20](#_Toc128236545)

[3.8. GLight 22](#_Toc128236546)

[3.9. DeviceActionManager 23](#_Toc128236547)

[4 系统初始化流程 25](#_Toc128236548)

[5 业务流程 26](#_Toc128236549)

[6 功能示例 27](#_Toc128236550)

[7 附录 术语缩写 30](#_Toc128236551)

# 表格索引

表1：适用模块 7

表2：术语缩写 30

# 图片索引

图1：硬件系统框架 8

图2：软件系统框架 9

图3：项目软件代码UML类图 11

图4：EventMesh中间件流转原理图 12

图5：云喇叭系统初始化流程图 25

图6：业务流程图 26

# 

1. 前言

本文档主要描述移远通信QuecPython云喇叭软件设计框架，包含软硬件系统框架、关键组件功能描述、系统初始化流程和业务流程的介绍以及功能示例，方便用户快速了解QuecPytho云喇叭的整体架构与功能。

* 1. 适用模块

表1：适用模块

|  |  |
| --- | --- |
| **模块系列** | **模块** |
| LTE Standard | EC21-AUX |
| EC800G-CN |
| EC200U系列 |
| EC600U系列 |
| EG915U系列 |
| EC200A系列 |
| EC600M-CN |
| EC800M-CN |
| EC800N-CN |
| EC600N-CN |
| NB-IoT | BC25系列 |
| LPWA | BG95系列 |
| BG77 |

1. 系统框架

* 1. 硬件系统框架

硬件系统框架介绍如下：

模块支持SIM卡、GPIO和UART等硬件接口。



图1：硬件系统框架

* 1. 软件系统框架

软件系统框架介绍如下：

1. 应用层处理云喇叭核心业务，解析上下行数据。
2. EventMesh为事件处理器，通过支持订阅和发布事件的机制来完成业务功能的触发与执行。
3. 模块接收外部事件或数据后，通过EventMesh调用上层注册的事件或数据的处理器来完成事件或 数据的处理。



图2：软件系统框架

1. 关键组件

下图为UML类图，描绘了项目软件代码中各组件对象之间的依赖关系与继承关系。图中以云喇叭作为顶层对象，将其与所依赖的组件对象建立了关联，具体如下图所示：



图3：项目软件代码UML类图

* 1. EventMesh
* **功能描述**

EventMesh是一种动态基础中间件。在事件驱动架构中，事件指的是系统中的变更、操作或观察，然后生成通知，再响应到各个对事件做出响应的处理器函数中。

* **实现原理**

EventMesh中间件流转原理图如下，详细说明请咨询移远通信技术支持。



图4：EventMesh中间件流转原理图

**备注**

|  |
| --- |
| 上层业务功能与实际的执行函数通过EventMesh进行数据通信，系统中的所有事件均来自EventStore（事件池）。需注意：EventStore是EventMesh框架中用来存储用户注册事件的，代码对外的接口中并未体现EventStore，因此上图中EventStore仅用于向用户展示EventMesh的基本实现原理。 |

* **订阅事件**

|  |
| --- |
| import EventMesh  def test(event, msg):  return msg  # 订阅函数  EventMesh.subscribe("test\_event", test) |

* **发布事件**

|  |
| --- |
| import EventMesh  def test(event, msg):  return msg  # 订阅函数  EventMesh.subscribe("test\_event", test)  # 发布事件  EventMesh.publish("test\_event", "OK") |

* 1. AudioManager
* **功能描述**

控制设备音频输出，包含TTS语音播报、音频文件播放和音频音量设置等功能。

* **实现原理**

1. 通过*\_\_init\_\_()*创建类方法并完成音频功能初始化，将原始音量等级简化为5个等级，方便使用。

|  |
| --- |
| class AudioManager(Abstract):  """  audio 初始化  音频文件播放  TTS 播报管理  """  def \_\_init\_\_(self):  self.\_\_audio = audio.Audio(0)  self.\_\_tts = audio.TTS(0)  self.\_\_audio\_volume = 3  self.\_\_tts\_priority = 2  self.\_\_tts\_breakin = 0  self.\_\_tts\_mode = 2  self.\_\_volume\_level = {  1: 1,  2: 3,  3: 6,  4: 9,  5: 11  }  self.log = get\_logger(\_\_name\_\_ + "." + self.\_\_class\_\_.\_\_name\_\_) |

1. 初始化类方法后，通过EventMesh将音频功能初始化注册为事件，完成音量初始化设置。

|  |
| --- |
| def post\_processor\_after\_initialization(self):  self.\_\_set\_audio\_pa()  vol\_num = EventMesh.publish("persistent\_config\_get", "vol\_num")  self.\_\_audio\_volume = vol\_num  # 设置TTS音量  self.\_\_audio.setVolume(self.\_\_volume\_level.get(self.\_\_audio\_volume))  EventMesh.subscribe("audio\_play", self.audio\_play)  EventMesh.subscribe("tts\_play", self.tts\_play)  EventMesh.subscribe("get\_audio\_state", self.get\_audio\_state)  EventMesh.subscribe("get\_tts\_state", self.get\_tts\_state)  EventMesh.subscribe("add\_audio\_volume", self.add\_audio\_volume)  EventMesh.subscribe("reduce\_audio\_volume", self.reduce\_audio\_volume) |

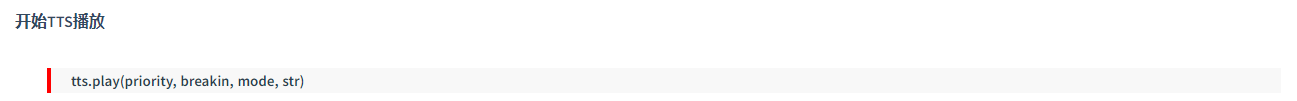
1. 通过QuecPython提供的音频API播放音频文件，API详细描述请参考wiki文档<https://python.quectel.com/wiki/#/>。

图形用户界面, 应用程序

描述已自动生成

|  |
| --- |
| def audio\_play(self, topic=None, filename=None):  """Play audio"""  if filename is None:  return  state = self.\_\_audio.play(self.\_\_tts\_priority, self.\_\_tts\_breakin, filename)  return True if state == 0 else False |

1. 通过QuecPython提供的TTS API播放输入内容，API详细描述请参考wiki文档https://python.quectel.com/wiki/#/。



图形用户界面, 应用程序

描述已自动生成

|  |
| --- |
| def tts\_play(self, topic=None, content=None):  """Play tts  \_\_tts\_priority 播放优先级，支持优先级0 ~ 4，数值越大优先级越高  \_\_tts\_breakin 打断模式，0表示不允许被打断，1表示允许被打断  \_\_tts\_mode 编码模式 2 - UTF-8  content 播报内容  """  if content is None:  return  state = self.\_\_tts.play(self.\_\_tts\_priority, self.\_\_tts\_breakin, self.\_\_tts\_mode, content)  return True if state == 0 else False |

* 1. ConfigStoreManager
* **功能描述**

用于持久化保存设备参数，提供读写文件数据的方法

* **实现原理**

1. 初始化类方法时会自动判断配置文件是否存在。若配置文件不存在，则自动创建一个JSON文件，文件创建成功后会自动将默认参数写入配置文件中；若配置文件已存在，则自动对比默认参数有无新增，若有新增参数则同步更新到配置文件。
2. 该类方法通过EventMesh注册文件读取和文件更新两个事件。

|  |
| --- |
| def post\_processor\_after\_initialization(self):  if ql\_fs.path\_exists(self.file\_name):  file\_map = ql\_fs.read\_json(self.file\_name)  for k in self.map.keys():  if k not in file\_map:  file\_map.update({k: self.map.get(k)})  self.\_\_store(msg=file\_map)  self.map = file\_map  else:  self.\_\_store()  EventMesh.subscribe("persistent\_config\_get", self.\_\_read)  EventMesh.subscribe("persistent\_config\_store", self.\_\_store) |

* 1. LteNetManager
* **功能描述**

初始化模块网络以及管理网络状态。设备默认为自动拨号注网，如需调整为手动拨号或者需要使用设置的APN进行拨号，请参考wiki文档<https://python.quectel.com/wiki>进行修改。

* **实现原理**

1. 该类方法初始化时会等待设备注网完成，且注册网络变化回调函数，通过checkNet API返回值判断设备注网状态，注网成功后会通过事件发布的方式启动TCP连接，若注网失败则会尝试重新注网。初始化类方法时会等待设备注网完成，并注册网络变化回调函数，通过checkNet API的返回值判断设备注网状态。注网成功后会通过事件发布的方式启动TCP连接，若注网失败，则会尝试重新注网。checkNet API的详细描述请参考wiki中心：<https://python.quectel.com/wiki>。

|  |
| --- |
| def wait\_connect(self, timeout):  """等待设备注网"""  self.log.info("wait net -----------")  stagecode, subcode = self.check\_net.wait\_network\_connected(timeout)  if stagecode == 3 and subcode == 1:  # 注网成功  EventMesh.publish("tts\_play", TTS\_CONTENT.STR\_CONNECT\_NET\_OK)  self.log.info("module net success, run mqtt connect")  EventMesh.publish('ota\_check')  EventMesh.publish('mqtt\_connect')  self.net\_error\_audio\_stop()  else:  # 注网失败  self.\_\_net\_error\_mode = 1  self.log.error("module net fail, wait try again")  EventMesh.publish("tts\_play", TTS\_CONTENT.SSTR\_CONNECT\_NET\_FAILED)  self.net\_fail\_process()  self.\_\_data\_call.setCallback(self.net\_state\_cb) |

1. 切换到手动拨号。

图形用户界面, 应用程序

描述已自动生成

1. 配置用户APN。



1. 网络状态异常处理

当设备网络状态发生变化时，可通过注册回调函数的方式通知应用层，如下所示：

|  |
| --- |
| def wait\_connect(self, timeout):  """等待设备注网"""  self.log.info("wait net -----------")  stagecode, subcode = self.check\_net.wait\_network\_connected(timeout)  if stagecode == 3 and subcode == 1:  # 注网成功  EventMesh.publish("tts\_play", TTS\_CONTENT.STR\_CONNECT\_NET\_OK)  self.log.info("module net success, run mqtt connect")  EventMesh.publish('ota\_check')  EventMesh.publish('mqtt\_connect')  self.net\_error\_audio\_stop()  else:  # 注网失败  self.\_\_net\_error\_mode = 1  self.log.error("module net fail, wait try again")  EventMesh.publish("tts\_play", TTS\_CONTENT.SSTR\_CONNECT\_NET\_FAILED)  self.net\_fail\_process()  self.\_\_data\_call.setCallback(self.net\_state\_cb) |

|  |
| --- |
| def net\_state\_cb(self, args):  """网络状态变化，会触发该回调函数"""  nw\_sta = args[1]  if nw\_sta == 1:  EventMesh.publish("tts\_play", TTS\_CONTENT.STR\_CONNECT\_NET\_OK)  self.log.info("network connected!")  self.net\_error\_audio\_stop()  else:  self.net\_error\_audio\_start()  EventMesh.publish("tts\_play", TTS\_CONTENT.SSTR\_CONNECT\_NET\_FAILED)  self.log.info("network not connected!") |

网络状态出现异常时，网络重连示例如下：

网络出现异常后，首先尝试使用cfun先关闭后启用网络功能实现重新注网，若通过cfun重新注网失败，会自动重启模块。若需要限制重连尝试次数，可在配置文件中设置取值进行控制。

|  |
| --- |
| def net\_fail\_process(self):  # 注网失败，尝试Cfun后重新找网，若Cfun失败则模块重启  state = net.setModemFun(0)  if state == -1:  self.log.error("cfun net mode error, device will restart.")  utime.sleep(5)  # Power.powerRestart()  state = net.setModemFun(1)  if state == -1:  self.log.error("cfun net mode error, device will restart.")  utime.sleep(5)  # Power.powerRestart()  self.log.info("cfun net mode success, note the net again")  self.wait\_connect(30) |

* 1. DeviceInfoManager
* **功能描述**

获取设备的一些基础信息，例如设备IMEI、SIM卡的ICCID等。

* **实现原理**

1. 初始化类函数时会将获取设备信息的函数并通过EventMesh注册成事件，用户可直接通过发布事件的方式使用。

|  |
| --- |
| def post\_processor\_after\_instantiation(self):  # 注册事件  EventMesh.subscribe("get\_sim\_iccid", self.get\_iccid)  EventMesh.subscribe("get\_device\_imei", self.get\_imei)  EventMesh.subscribe("get\_fw\_version", self.get\_device\_fw\_version)  EventMesh.subscribe("get\_csq", self.get\_csq) |

1. 查询SIM卡的ICCID时，使用SIM API获取当前SIM卡的ICCID。

|  |
| --- |
| def get\_iccid(self, event=None, msg=None):  """查询 ICCID"""  if self.\_\_iccid == "":  msg = sim.getIccid()  if msg != -1:  self.\_\_iccid = msg  else:  self.log.warn("get sim iccid fail, please check sim")  return self.\_\_iccid |

1. 设备信息API的详细描述可参考wiki文档：<https://python.quectel.com/wiki>。
   1. OtaManager

* **功能描述**

远程升级应用程序代码。

* **实现原理**

1. 使用移远云物联网平台进行升级。
2. 在移远云物联网平台创建升级任务后，将设备上电，待设备注网成功后，通过API检测是否有升级任务。
3. 用户可直接通过发布事件的方式使用OTA升级检测接口函数。

|  |
| --- |
| def post\_processor\_after\_initialization(self):  # 注册事件  EventMesh.subscribe("ota\_check", self.check\_ota\_event) |

1. 检测到升级任务后调用下载接口下载升级文件，待升级文件下载完成后，模块会自动重启完成升级。

|  |
| --- |
| def upgrade\_fota\_sh(self, upgrade\_path):  try:  action = self.upgrade\_info["action"]  url = self.upgrade\_info["url"]  except Exception as e:  return self.status\_code\_dict["ERROR"][4]  try:  if action:  self.report("DOWNLOADING\_FIRMWARE")  fota = app\_fota.new()  fota.download(url, upgrade\_path)  self.report("DOWNLOADED\_NOTIFY\_UPDATE")  fota.set\_update\_flag()  self.report("UPDATE\_START")  return self.status\_code\_dict["UPGRADE"][0]  else:  return self.status\_code\_dict["UPGRADE"][1]  except Exception as e:  self.report(PROCESS\_CODE[5])  return self.status\_code\_dict["ERROR"][2] |

* 1. AliYunManage
* **功能描述**

模块通过MQTT协议连接至阿里云IoT平台，实现连接平台、断开与平台连接、以及进行数据上 下行传输等功能。

* **实现原理**

1. 通过*\_\_init\_\_*()创建类方法并完成类属性的初始化。类属性包含连接阿里云IoT平台的三元组（ProductKey、DeviceName和DeviceSecret）、保活时间等。

|  |
| --- |
| class AliYunManage(Abstract):  '''  MQTT interface  '''  def \_\_init\_\_(self):  self.\_\_mqtt\_client = None  self.product\_key = '' # 产品识别码  self.product\_secret = None # 产品密钥  self.device\_name = '' # 设备（备注）名称  self.device\_secret = '' # 设备密钥  self.client\_id = '' # 自定义  self.clean\_session = True # 客户端类型 (False: 持久客户端，True: 临时的)  self.keep\_alive = 300 # 允许最长通讯时间（s）  self.sub\_topic = '' # 订阅地址  self.qos = 1 # 消息服务质量 0：发送者只发送一次消息，不进行重试 1：发送者最少发送一次消息，确保消息到达Broker  self.conn\_flag = False  self.start\_mqtt\_flag = False  self.log = get\_logger(\_\_name\_\_ + "." + self.\_\_class\_\_.\_\_name\_\_) |

1. 初始化类方法后将需要对外提供的方法通过事件注册的方式注册到EventMesh中。

|  |
| --- |
| def post\_processor\_after\_initialization(self):  EventMesh.subscribe("mqtt\_connect", self.start\_mqtt\_connect) |

1. 发起连接请求。

|  |
| --- |
| def connect(self, topic=None, data=None):  if not self.conn\_flag:  self.conn\_flag = True  self.\_\_mqtt\_client = aLiYun(self.product\_key,self.product\_secret,self.device\_name,self.device\_secret)  con\_state = self.\_\_mqtt\_client.setMqtt(self.client\_id,clean\_session=self.clean\_session,reconn=True)  self.log.info("connect con\_state --{}".format(con\_state))  if con\_state != 0:  self.log.warn("mqtt connect failed!")  EventMesh.publish("tts\_play", TTS\_CONTENT.SERVER\_CONN\_FAIL)  return False  self.\_\_mqtt\_client.start()  self.\_\_mqtt\_client.setCallback(self.callback)  sub\_sta = self.\_\_mqtt\_client.subscribe(self.sub\_topic, qos=self.qos)  if sub\_sta != 0:  self.log.warn("mqtt subscribe topic failed!")  return False  EventMesh.publish("tts\_play", TTS\_CONTENT.SERVER\_CONN\_SUCCESS)  self.log.info("mqtt connect success!")  return True |

1. 下行数据回调函数。

|  |
| --- |
| def callback(self, topic, msg):  '''  mqtt 消息回调  '''  json\_data = ujson.loads(msg)  params = json\_data.get('params', False)  self.log.info("json\_data {}".format(json\_data))  EventMesh.publish("put\_msg\_queue", params) |

* 1. GLight
* **功能描述**

管理设备LED灯状态，通过GPIO使能控制LED的开关和闪烁。

* **实现原理**

1. 通过*\_\_init\_\_()*创建类方法并完成类属性的初始化和LED灯GPIO引脚的初始化。

|  |
| --- |
| class RLight(Light):  def \_\_init\_\_(self):  super(RLight, self).\_\_init\_\_(Pin(Pin.GPIO13, Pin.OUT, Pin.PULL\_DISABLE, 0)) |

1. 初始化类方法后将需要对外提供的方法通过事件注册的方式注册到EventMesh中。

|  |
| --- |
| def post\_processor\_after\_initialization(self):  """订阅此类所有的事件到 EventMesh中"""  EventMesh.subscribe("red\_on", self.on)  EventMesh.subscribe("red\_off", self.off)  EventMesh.subscribe("red\_blink", self.blink\_O)  EventMesh.subscribe("red\_read", self.read)  EventMesh.subscribe("red\_set\_timeout", self.set\_timeout) |

1. 使能GPIO引脚控制LED灯。

|  |
| --- |
| def on(self, topic=None, data=None):  self.l.write(1)  def off(self, topic=None, data=None):  self.l.write(0) |

* 1. DeviceActionManager
* **功能描述**

控制设备状态，包括设备开关机、重启及待机状态，控制外设。

* **实现原理**

1. 初始化类方法后将需要对外提供的方法通过事件注册的方式注册到EventMesh中。

|  |
| --- |
| class DeviceActionManager(Abstract):  """  设备行为  """  def \_\_init\_\_(self):  self.\_\_led\_flag = 1  self.\_\_await\_start\_time = 0  self.\_\_lock = Lock()  self.log = get\_logger(\_\_name\_\_ + "." + self.\_\_class\_\_.\_\_name\_\_)  def post\_processor\_after\_initialization(self):  # 注册事件  EventMesh.subscribe("device\_start", self.device\_start)  EventMesh.subscribe("device\_shutdown", self.device\_shutdown)  EventMesh.subscribe("device\_restart", self.device\_restart)  EventMesh.subscribe("update\_wait\_time", self.update\_device\_standby\_wait\_time)  EventMesh.subscribe("update\_led\_flag", self.update\_led\_flag)  # LED 灯闪烁  \_thread.start\_new\_thread(self.blink\_thread, ())  # 开启待机闪烁  \_thread.start\_new\_thread(self.device\_standby, ()) |

1. 设备待机状态。

|  |
| --- |
| def device\_standby (self):  # 设备待机状态  while True:  # self.log.info("device\_standby led\_flag {}".format(self.\_\_led\_flag))  if self.\_\_led\_flag == 1:  if utime.time() - self.\_\_await\_start\_time > 30:  EventMesh.publish("blue\_blink")  utime.sleep(8)  continue  utime.sleep(10)  continue |

1. 设备LED灯控制任务。

|  |
| --- |
| def blink\_thread(self):  # 闪烁线程  while True:  light\_flag = EventMesh.publish("update\_led\_flag")  # self.log.info("blink\_thread -- light\_flag {}".format(light\_flag))  if light\_flag < 4 and light\_flag != 1:  if light\_flag == 3:  # 语音提示低电量  EventMesh.publish("audio\_play", AUDIO\_FILE\_NAME.BATT\_LOW)  if light\_flag == 1:  EventMesh.publish("blue\_blink")  elif light\_flag == 2:  EventMesh.publish("green\_blink")  elif light\_flag == 3:  EventMesh.publish("red\_blink")  utime.sleep(8)  else:  utime.sleep(8) |

1. 设备开关机动作。

|  |
| --- |
| def device\_shutdown(self, topic=None, data=None):  # 设备关机  EventMesh.publish("audio\_play", AUDIO\_FILE\_NAME.DEVICE\_SHUTDOWN)  utime.sleep(5)  Power.powerDown()  def device\_start(self, topic=None, data=None):  # 设备开机  EventMesh.publish("audio\_play", AUDIO\_FILE\_NAME.DEVICE\_START)  def device\_restart(self, topic=None, data=None):  # 设备重启  Power.powerRestart() |

1. 系统初始化流程



图5：云喇叭系统初始化流程图

系统初始化流程说明：

1. 云喇叭所有功能类进行初始化。
2. 所有的类方法都会约定必须有初始化前后或实例化前后要完成的事件注册或功能处理，因此通过一个应用程序类将在装载和启动所有的类方法时将每个类方法的初始化前后动作执行完成。
3. 业务流程



图6：业务流程图

1. 功能示例

本章节介绍如何基于项目代码进行二次开发，例如新增功能、修改代码功能逻辑等。用户可以参照下文示例快速上手，熟悉代码流程。

在准备调整代码前需先了解当前代码中功能是如何流转运行的。首先整个项目中实现的功能函数均采取的是通过EventMesh进行事件驱动，即实现了一个功能功函数后，为该函数定义一个事件名称，例如查询设备IMEI，并且将事件名称与对应的事件函数以键值对的方式进行映射（例如“Get\_Imei”: Function)，这种创建映射关系的动作称之为订阅事件。EventMesh会保存这些订阅事件，当用户需要查询设备IMEI时，仅需发布Get\_Imei事件即可获取返回结果。因此，每个功能都可以独立开发，独立调试，达到解耦合的效果。

二次开发的步骤如下：

1. 新增一个功能类函数，需先实现该功能函数，如下所示：

|  |
| --- |
| class TestAddFunction(Abstract):  """  测试新增一个功能函数  """  def \_\_init\_\_(self):  pass  def post\_processor\_after\_initialization(self):  # 注册事件  EventMesh.subscribe("test\_function", self.test\_function)  def test\_function(self, event, msg):  """该函数下实现具体功能"""  print("TestAddFunction --test\_function()") |

如上所示，首先创建一个功能类方法，该类方法继承了一个公共基类Abstract，该基类用于约束开发者在功能实现上保持一致性。然后在类方法中重新实现基类Abstract的*post\_processor\_after\_instantiation()*方法，由于新增的功能类函数在实现之后需要加入到应用程序类中注册并启动，应用程序类会调用执行每一个继承于Abstract类的功能类中的初始化方法，即*post\_processor\_after\_instantiation()*方法，因此需要在该方法中将提供调用的方法订阅至EventMesh中。

Abstract基类：

|  |
| --- |
| class Abstract(object):  def post\_processor\_after\_instantiation(self, \*args, \*\*kwargs):  """实例化后调用"""  pass  def post\_processor\_before\_initialization(self, \*args, \*\*kwargs):  """初始化之前调用"""  pass  def initialization(self, \*args, \*\*kwargs):  """初始化load"""  pass  def post\_processor\_after\_initialization(self, \*args, \*\*kwargs):  """初始化之后调用"""  pass |

应用程序类：

|  |
| --- |
| class App(object):  def \_\_init\_\_(self):  self.managers = []  self.log = get\_logger(\_\_name\_\_ + "." + self.\_\_class\_\_.\_\_name\_\_)  def append\_manager(self, manager):  if isinstance(manager, Abstract):  manager.post\_processor\_after\_instantiation()  self.managers.append(manager)  return self  def start(self):  for manager in self.managers:  manager.post\_processor\_before\_initialization()  manager.initialization()  manager.post\_processor\_after\_initialization() |

1. 将新增的类方法加入到*main.py*入口函数中：
2. 导入新增的类方法。*usr.msg*表示usr（用户）分区下面的*mgr.py*文件，新增的类方法编写在*mgr.py*文件中。

|  |
| --- |
| from usr.audio\_control import AudioManager  from usr.cloud import AliYunManage  from usr.common import ConfigStoreManager, Abstract  from usr.led\_control import RLight, GLight, BLight  from usr.ota\_control import OTAManager  from usr.mgr import FactoryManager, DeviceInfoManager, UartManager, KeypadManager,\  DeviceActionManager,OrderManager,ChargeManager,LteNetManager,CloudHornManager  from usr.app\_ota import TestAddFunction |

1. 将类方法添加到应用程序类中：

|  |
| --- |
| app = App()  # app 注册  app.append\_manager(ConfigStoreManager())  app.append\_manager(AudioManager())  app.append\_manager(RLight())  app.append\_manager(GLight())  app.append\_manager(BLight())  app.append\_manager(OTAManager())  app.append\_manager(KeypadManager())  app.append\_manager(DeviceInfoManager())  app.append\_manager(UartManager())  app.append\_manager(DeviceActionManager())  app.append\_manager(OrderManager())  app.append\_manager(ChargeManager())  app.append\_manager(AliYunManage())  app.append\_manager(LteNetManager())  app.append\_manager(CloudHornManager())  app.append\_manager(FactoryManager())  # 启动  app.start() |

1. 使用新增的类方法：

|  |
| --- |
| class PlayTestFuntion():  """  使用新增类方法  """  def execute\_test\_function(self):  # 通过事件发布的方式找到对应的方法执行  EventMesh.publish("test\_function") |

通过事件发布的方式找到已经订阅的事件，执行对应的事件函数。

1. 附录 术语缩写

表2：术语缩写

|  |  |  |
| --- | --- | --- |
| **缩写** | **英文全称** | **中文全称** |
| API | Application Programming Interface | 应用程序编程接口 |
| APN | Access Point Name | 接入点名称 |
| APP | Application | 应用 |
| GPIO | General Purpose Input/Output Port | 通用输入输出端口 |
| ICCID | Integrate Circuit Card Identity | 集成电路卡识别码 |
| IMEI | International Mobile Equipment Identity | 国际移动设备识别码 |
| IoT | Internet of Things | 物联网 |
| LED | Light Emitting Diode | 发光二极管 |
| MQTT | Message Queuing Telemetry Transport | 消息队列遥测传输 |
| OTA | Over-the-air programming | 空中编程 |
| PA | Power Amplifier | 功率放大器 |
| SIM | Subscriber Identity Module | 用户身份识别模块 |
| TCP | Transmission Control Protocol | 传输控制协议 |
| TTS | Text To Speech | 文字转语音 |
| UART | Universal Asynchronous Receiver/Transmitter | 通用异步收发传输器 |
| UML | Unified Modeling Language | 统一建模语言 |
| USB | Universal Serial Bus | 通用串行总线 |