

**QuecPython 电表方案设计指导**

**LTE Standard/LPWA模块系列**

版本：1.0.0

日期：2023-03-24

 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** |
| - | 2023-03-24 | Pawn ZHOU | 文档创建 |
| 1.0.0 | 2022-03-24 | Pawn ZHOU | 临时版本 |

# 目录

[文档历史 3](#_Toc127025846)

[目录 4](#_Toc127025847)

[表格索引 5](#_Toc127025848)

[图片索引 6](#_Toc127025849)

[1 引言 7](#_Toc127025850)

[1.1. 适用模块 7](#_Toc127025851)

[2 系统框架 8](#_Toc127025852)

[2.1. 硬件系统框架 8](#_Toc127025853)

[2.2. 软件系统框架 9](#_Toc127025854)

[3 关键组件 10](#_Toc127025855)

[3.1. 事件处理器（EventMesh） 11](#_Toc127025857)

[3.2. RFC1662协议数据解析器（RFC1662ProtocolResolver） 12](#_Toc127025858)

[3.3. 配置存储管理程序（ConfigStoreManager） 15](#_Toc127025859)

[3.4. 网络状态管理程序（NetStateManager） 16](#_Toc127025860)

[3.5. TCP模式管理程序（TcpModeManager） 17](#_Toc127025861)

[3.6. 主串口管理程序（MainUartManager） 18](#_Toc127025862)

[3.7. 设备信息管理程序（DeviceInfoManager） 20](#_Toc127025863)

[3.8. FTP升级管理程序（FtpOtaManager） 20](#_Toc127025864)

[4 系统初始化流程 23](#_Toc127025865)

[5 业务流程 25](#_Toc127025866)

[6 功能示例 26](#_Toc127025867)

[7 附录 术语缩写 29](#_Toc127025868)

# 表格索引

表1：适用模块 7

表2：术语缩写 29

# 图片索引

[图1：硬件系统框架 8](#_Toc127025809)

[图2：软件系统框架 9](#_Toc127025810)

[图3：项目软件代码UML类图 10](#_Toc127025811)

[图4：EventMesh中间件流转原理图 11](#_Toc127025812)

[图5：电表系统初始化流程图 23](#_Toc127025813)

[图6：业务流程图 25](#_Toc127025814)

# 

1. 引言

本文档主要描述移远通信QuecPython电表的设计框架，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QuecPython电表的整体架构与功能。

* 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
2. 硬件系统框架

硬件系统框架介绍如下：

1. 模块支持SIM卡、GPIO和UART等硬件接口。
2. 模块通过UART口与MCU进行通信以完成数据交互。



**图1：硬件系统框架**

2. 软件系统框架

软件系统框架介绍如下：

1. 应用层用于处理电表核心业务，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。图中以电表作为顶层对象，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. 事件处理器（EventMesh）

* **功能描述**

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

* **实现原理**

EventMesh中间件流转原理如下图所示，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. RFC1662协议数据解析器（RFC1662ProtocolResolver）

* **功能描述**

RFC1662协议数据解析器用于解析业务中传输的RFC1662协议数据并对该类数据进行解包和组包。

* **实现原理**

1. 实例化创建RFC1662协议数据解析器，如下所示。

|  |
| --- |
| rfc1662\_protocol\_resolver = RFC1662ProtocolResolver()  rfc1662\_protocol\_resolver.add\_support\_protocol(ModuleToELEMeterProtocol()) \  .add\_support\_protocol(ELEMeterToModuleProtocol()) \  .add\_support\_protocol(EventReporterProtocol()) \ |

将协议指令事件添加到List数组中，如下所示。

|  |
| --- |
| def add\_support\_protocol(self, prot):  self.support\_col\_list.append(prot)  return self |

1. RFC1662协议解析类函数初始化时，会将添加的协议指令事件MODE与对应的handler处理函数注册到EventMesh组件中。若设备接收到MODE指令，则通过EventMesh发布注册的协议指令事件。

|  |
| --- |
| def post\_processor\_after\_instantiation(self):  for prot in self.support\_col\_list:  EventMesh.subscribe(prot.MODE, prot.handler)  for proc in self.support\_cmd\_list:  EventMesh.subscribe(proc.MODE, proc.handler)  EventMesh.subscribe("RFC1662-PROTOCOL-EXECUTE", self.execute) |

MODE指令函数示例如下：

|  |
| --- |
| class ELEMeterToModuleProtocol(ProtoCol):  """表计to模块"""  MODE = 0x2300  def handler(self, topic, msg: RFC1662Protocol):  print("ELEMeterToModuleProtocol {} {}".format(hex(topic), msg))  if msg.info().param\_id() in [CONSEM\_COMMON\_RFC1662\_PARAM\_ID.LQI]:  EventMesh.publish(msg.info().param\_id(), msg)  else:  EventMesh.publish(msg.info().request\_id(), msg) |

1. 收到表计发送给模块的数据后检查匹配协议帧的格式。

RFC1662协议帧固定格式如下：

|  |
| --- |
| class RFC1662ProtocolResolver(Abstract):  HEADER = 0x7e  ADDRESS = 0xFF  CONTROL = 0x03 |

判断协议帧是否满足格式：

|  |
| --- |
| def check(self, data):  return data[0] == self.HEADER and data[1] == self.ADDRESS and data[2] == self.CONTROL |

1. 判断数据是否能解析，返回的解析状态以及解析到的截止下标。

|  |
| --- |
| def parse(self, data):  # HEADER/1b ADDRESS/1b CONTROL/1b RFC\_PROTO/2b INFO\_LEN/2b  \_i = 0  if len(data) < 7:  # 数据长度小于7字节，报文不满足固定帧头长度  return False, \_i  if data[0] == self.HEADER and data[1] == self.ADDRESS and data[2] == self.CONTROL:  # 这里直接跳过协议 占2字节  \_i += 7  info\_len = struct.unpack(">H", data[5:7])[0]  # 包大小等于头部7字节 + 长度 + (fcs/2b + 0x7e)  \_i = \_i + info\_len + 3  # 判断包数据大小小于整包的大小,包不完整  if len(data) < \_i:  return False, 0  else:  return True, \_i  else:  return False, \_i |

1. 数据解析由RFC1662Protocol类方法中的build函数执行，build函数按照协议帧格式使用struct依次读取数据，并将解析出的数据分别保存在RFC1662Protocol类属性中。数据解析完成后RFC1662Protocol类将作为对象返回。

build函数调用：

|  |
| --- |
| @classmethod  def resolver(cls, data):  rfs: RFC1662Protocol = RFC1662Protocol.build(data)  rfs.print()  EventMesh.publish(rfs.protocol(), rfs)  return None |

数据解析方法：

|  |
| --- |
| @classmethod  def build(cls, data):  rfc\_proto = RFC1662Protocol()  if data[0] == cls.HEADER and data[1] == cls.ADDRESS and data[2] == cls.CONTROL:  # 读取protocol，从第三个字节往后读两个字节  rfc\_proto.increment(3)  rfc\_proto.set\_protocol(struct.unpack\_from("<H", data[rfc\_proto.position():rfc\_proto.position() + 2])[0])  rfc\_proto.increment(2)  # 继续读取info区里面的长度  info\_len = struct.unpack(">H", data[rfc\_proto.position():rfc\_proto.position() + 2])[0]  rfc\_proto.set\_info\_len(info\_len)  rfc\_proto.increment(2)  # 继续读取info区里面的命令  info\_cmd = struct.unpack("B", data[rfc\_proto.position():rfc\_proto.position() + 1])[0]  rfc\_proto.set\_info\_cmd(info\_cmd)  rfc\_proto.increment(1) |

RFS对象作为build函数的结果返回。数据解析完成后RFS对象中将包含所有数据信息，通过EventMesh发布如下指令即可找到对应的处理函数执行。

|  |
| --- |
| //该指令在类函数在初始化时已被注册，详见***步骤2***中描述。  EventMesh.publish(rfs.protocol(), rfs) = EventMesh.publish(MODE指令函数, rfs) |

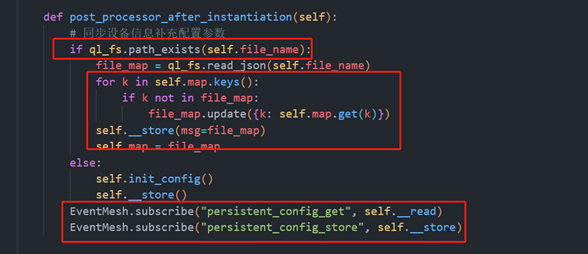
1. 配置存储管理程序（ConfigStoreManager）

* **功能描述**

该组件主要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。



注册文件读取和文件更新两个事件方法

确认是否有新增默认配置参数；若有新增，则同步到配置文件中

判断配置文件是否存在



1. 网络状态管理程序（NetStateManager）

* **功能描述**

该组件主要用于模块网络初始化以及网络状态管理。

* **实现原理**

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\_net\_connect(self, timeout):  """等待设备找网"""    apn\_info\_list = EventMesh.publish("persistent\_config\_get","apn")  self.log.info("apn:{},user:{},password:{}".format(apn\_info\_list[0], apn\_info\_list[1], apn\_info\_list[2]))  ret = self.\_\_net.setApn(1,0,apn\_info\_list[0],apn\_info\_list[1],apn\_info\_list[2],0,0)  if ret == -1:  self.log.info("set apn failed.")  else:  self.log.info("set apn success.")  self.log.info("wait net connect")  EventMesh.publish("net\_led", [600, 600, 0, 0])  self.check\_net.poweron\_print\_once()  stagecode, subcode = self.check\_net.wait\_network\_connected(timeout)  if stagecode == 3:  if subcode == 1:  # 注网成功/拨号成功  self.log.info("network register success")  self.log.info("datacall connect success.")  EventMesh.publish('get\_sim\_iccid')  EventMesh.publish('get\_csq')  EventMesh.publish('get\_ip')  EventMesh.publish('tcp\_start')  EventMesh.publish("net\_led", [75, 3000, 0, 1])  else:  # 拨号失败，启动手动拨号  self.log.info("check net connect:network register success")  self.log.info("check net connect:data call failed.")  self.net\_data\_call() |

1. TCP模式管理程序（TcpModeManager）
   * + - **功能描述**

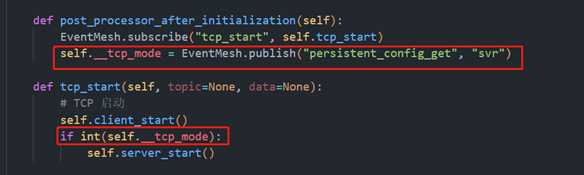
该组件用于管理设备支持的两种TCP连接模式，客户端和服务端，并根据对应的模式在设备注网成功后启动TCP连接。

* + - * **实现原理**

1. 该类方法通过EventMesh注册启动TCP功能事件，在设备注网成功后通过发布该事件来启动TCP连接。

|  |
| --- |
| def post\_processor\_after\_initialization(self):  EventMesh.subscribe("tcp\_start", self.tcp\_start)  self.\_\_tcp\_mode = EventMesh.publish("persistent\_config\_get", "svr") |

1. TCP启动方法会根据配置文件中的TCP模式来选择启动客户端还是服务端。



根据TCP模式选择启动客户端还是服务端

读取配置文件中的TCP模式



1. 主串口管理程序（MainUartManager）

* **功能描述**

该组件主要用于初始化设备串口，发送、接收和处理串口消息数据。

* **实现原理**

1. 模块和表计使用RFC1662协议通过串口进行通信，因此串口对象需要装载协议解析器以用于数据解析。

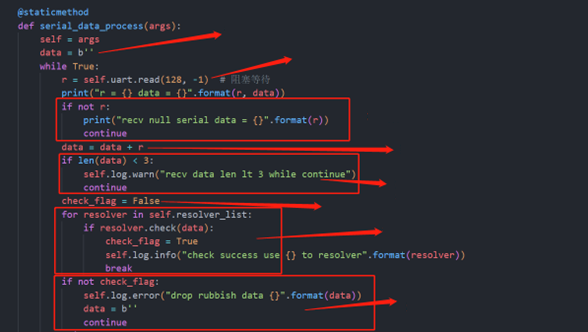
|  |
| --- |
| main\_uart\_manager = MainUartManager()  main\_uart\_manager.add\_resolver(at\_cmd\_resolver)\  .add\_resolver(rfc1662\_protocol\_resolver) |

|  |
| --- |
| def add\_resolver(self, rsl):  self.resolver\_list.append(rsl)  return self |

1. 初始化类函数时会创建一个串口对象，通过该对象可以对串口进行读写操作，且提供向该串口写入数据的功能。

|  |
| --- |
| def post\_processor\_after\_instantiation(self):  for rsl in self.resolver\_list:  rsl.post\_processor\_after\_instantiation()  self.uart = Serial(self.no, self.bate, self.data\_bits, self.parity, self.stop\_bits, self.flow\_control)  EventMesh.subscribe("uart\_write", self.write) |

1. 启动一个串口数据监听任务，当串口收到表计发来的数据后会自动匹配解析器来判断该条数据能否被解析。若数据能够被解析，则数据将通过解析器被返回。收到返回的数据后，数据解析结果会重新按照协议打包好返回给表计。



若未匹配到解析器，则认为是无效数据

判断数据长度是否大于3字节，上一步已经保存了数据

遍历支持的解析器，匹配到解析器后继续执行

检查标志

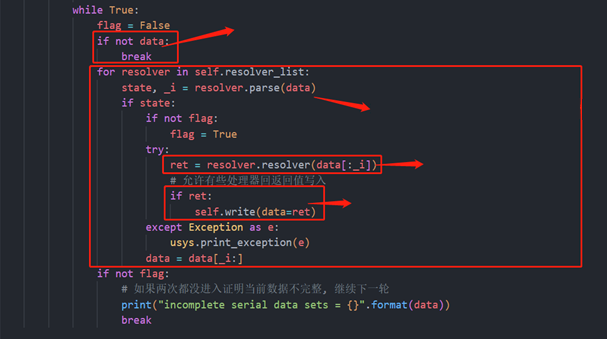
判断上一包数据接收是否完整，若不完整，则接收到第二包数据后与第一包数据相加

没有数据，继续

阻塞，等待串口回调数据

初始数据，可能保留半包的数据





若有返回值，则通过串口返回

处理和解析数据

判断数据能否被解析，返回解析状态和解析到的截止下标

判断是否无数据，若无数据则结束继续等待

遍历匹配的解析器



1. 设备信息管理程序（DeviceInfoManager）

**功能描述**

该组件用于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)  EventMesh.subscribe("get\_ip", self.get\_ip) |

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  self.log.info("ccid:{}".format(self.\_\_iccid))  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中心：<https://python.quectel.com/wiki>。
2. FTP升级管理程序（FtpOtaManager）

* **功能描述**

该组件用于将应用程序代码通过FTP远程升级更新。

* **实现原理**

1. FTP远程升级需要提供服务器的地址信息以及用户名和密码。
2. 初始化类方法时通过EventMesh注册FTP远程升级检测事件，并通过发布事件的方式调用该方法。

|  |
| --- |
| def post\_processor\_before\_initialization(self):  EventMesh.subscribe("ftp\_upgrade", self.download) |

1. 与FTP服务端建立连接，连接建立成功后即可登录FTP服务器，获取升级包路径。

|  |
| --- |
| def download(self, topic, msg):  from ftplib import FTP  self.ftp = FTP()  connect\_res = self.ftp.connect(host=self.host, port=self.port)  print("ftp.connect(): %s" % connect\_res)  login\_res = self.ftp.login(user=self.user, passwd=self.passwd)  print("ftp.login(): %s" % login\_res)  fp = open(self.parent\_dir + self.file\_name, "wb")  server\_path = self.ftp.pwd() |

1. 将升级包下载至模块，下载完成后需要将升级包进行解压。解压完成后设置升级标志，设备重启后完成升级。

|  |
| --- |
| def download(self, topic, msg):  from ftplib import FTP  self.ftp = FTP()  connect\_res = self.ftp.connect(host=self.host, port=self.port)  print("ftp.connect(): %s" % connect\_res)  login\_res = self.ftp.login(user=self.user, passwd=self.passwd)  print("ftp.login(): %s" % login\_res)  fp = open(self.parent\_dir + self.file\_name, "wb")  server\_path = self.ftp.pwd()  res = self.ftp.retrbinary("RETR " + server\_path + "/" + self.file\_name, fp.write)  msg = "Download %s to device %s."  if res.startswith('226 Transfer complete'):  print(msg % (self.file\_name, "success"))  app\_fota\_download.app\_fota\_pkg\_mount.mount\_disk()  fd = FileDecode(self.parent\_dir + self.file\_name, parent\_dir=app\_fota\_download.get\_updater\_dir())  fd.unzip()  stat = fd.unpack()  if stat:  fd.update\_stat()  fd.set\_flag()  # 重启设备升级  print("success")  else:  print("failed")  pass  else:  print(msg % (self.file\_name, "falied"))  fp.close() |

1. 系统初始化流程



**图5：电表系统初始化流程图**

系统初始化流程说明：

1. 创建RFC1662解析器对象，将需要使用到该解析器的类方法都注册到解析器对象中，此处的类方法表示研发人员定义的指令功能实现。
2. 创建串口对象，用于和表计进行通信。由于串口会一直监听表计发送过来的数据，因此需要将解析器注册到串口对象中，在接收数据时可使用解析器对象进行解析并返回结果。
3. 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，因此移研发人员通过一个APP类将所有的类方法，在装载和启动时，将每个类方法的初始化前后动作执行完。

1. 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。用户可以参照下文的示例快速上手，熟悉代码流程。

在调整代码前，需要先了解当前代码中功能是如何流转运行的。整个项目中实现的功能函数采取的都是通过EventMesh进行事件驱动，即实现了一个功能函数之后，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做映射（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这个事件即可得到返回结果。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。

二次开发的步骤描述如下：

**步骤一：**新增一个功能类函数前，需要先实现该功能类函数，如下所示：





如上图所示，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公共基类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APP类中注册并启动，APP类会将每一个继承于Abstract类的功能类中的初始化方法调用并执行，也就是*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到EventMesh中。

Abstract基类：

文本

描述已自动生成



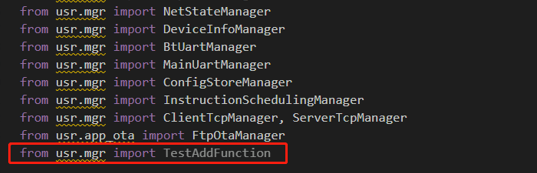
APP类：





**步骤二：**将新增的类方法加入到*main.py*入口函数中：

* + - 1. 导入新增的类方法。usr.msg表示usr（用户）分区下的*mgr.py*文件，新增的类方法编写在mgr.py文件中。





* + - 1. 将类方法添加到APP类中：



需注意：此处类方法加()完成初始化后即表示添加到APP类中



**步骤三：**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 |
| APP | Application | 应用 |
| FTP | File Transfer Protocol | 文件传输协议 |
| 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 |
| LED | Light Emitting Diode | 发光二极管 |
| MCU | Microcontroller Unit | 微控制单元 |
| RFC | Request For Comments | 一系列以编号排定的文件 |
| SIM | Subscriber Identity Module | 用户身份识别模块 |
| SMS | Short Message Service | 短消息服务 |
| TCP | Transmission Control Protocol | 传输控制协议 |
| 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 |