M4 脚本 Python 开发使用手册

M4 脚本 Python 开发使用手册

java.py 介绍

该文件给予 使用 Python 语言进行定制开发者使用，主要为 M4 对脚本开放的方法，以及开发过程中提供对应的方法提示，默认有 Python 基础。注：java.py，由于 某些原因，导致该文件最好第一个被加载。 java.py 不定时更新 , 使用前请下载(如无下载权限,可以复制该内容)最新版本 ,

当前版本 0.0.1

java.py

脚本启动

脚本会在系统启动时自动加载，并执行 boot() 函数，所以在 对应 脚本中需要存在一个 boot() 函数。

新增 .py 的文件，请再第一行 引 用 java.py

from java import *

如果在脚本启动过程中，引用其他自定义 xxx.py 文件报错，编译报错：ModuleNotFoundError: No module named 'xxx'，则需要在对应的 from xxx import * 前加上如下代码

import sys

import os

current_dir = os.path.dirname(os.path.abspath(**file**))

sys.path.append(current_dir)

from xxx import *

基础 base : ScriptBase

ScriptBase 类提供 base 方法，其中可以使用 base.xxx 来执行 ScriptBase 类中的方法

class ScriptBase:

```
def traceContext(self) -> TraceContext:
```

pass

```
def logDebug(self, tc: TraceContext, msg: str):
```

pass

```
def logInfo(self, tc: TraceContext, msg: str):
```

pass

```
def logError(self, tc: TraceContext, msg: str, err: object | None):
```

pass

```
def scheduledAtFixedDelay(self, name: str, delay: float, work: callable):
```

pass

```
def robustExecute(self, tc: TraceContext, description: str, func: str,
args):
```

pass

```
def runOnce(self, tc: TraceContext, mainId: str, actionId: str, work:
callable) -> object:
```

pass

```
def withLock(self, name: str, action: callable):
```

pass

```
def withFairnessLock(self, name: str, fair: bool, action: callable):
```

pass

```
def parseJsonString(self, string: str):
```

pass

```
def jsonToString(self, o) -> str | None:
```

pass

```
def xmlToJsonStr(self, o) -> object | None:
```

pass

def now(self) -> Date:

pass

```
def throwBzError(self, code: str, args: []) -> object:
```

pass

```
def assignJavaMapToJsMap(self, jsMap, javaMap: JavaMap):
```

pass

```
def setScriptButtons(self, buttons: str):
```

pass

```
def getGlobalValue(self, key: str) -> object | None:
```

pass

```
def setGlobalValue(self, key: str, value: object | None):
```

pass

```
def recordFailure(self, tc: TraceContext, failure: FailureRecordReq):
```

pass

```
def createUserNotice(self, tc: TraceContext, req: UserNoticeReq):
```

pass

```
def stringToMd5(self, tc: TraceContext, data: str) -> str:
```

pass

base: ScriptBase

traceContext

M4 很多函数的第一个参数是 TraceContext 对象。这个对象表示要处理的一件事情。在处理开始时创建这个对象，后续函数调用，都传入同一个对象，那么在打日志、记录数据库时，就能把处理这件事情的所有调用都串起来。

创建 TraceContext 对象的方式：

tc = base.traceContext()

logDebug

打印 log 日志，日志等级为 debug：

base.logDebug(tc,"logDebug")

logInfo

打印 log 日志，日志等级为 info：

base.logInfo(tc,"logInfo")

logError

打印 log 日志，日志等级为 error：

第三个参数为报错信息，允许为空

base.logError(tc, "logError", None)

scheduledAtFixedDelay

定时器，定时执行定制方法，使用场景举例：

每隔 3s 打印一条 log日志，scheduledAtFixedDelay 需要与 thread.createThread 方法搭配使用

def boot():

'''

参数1: thr1eadA（线程名称）

参数2: 执行的函数名称

'''

thread.createThread("thr1eadA","scheduledTask")

'''

参数1: scheduler_name（自定义定时器名称）

参数2: 3000（间隔时间，单位毫秒）

参数3: log_info（执行的方法）

'''

def scheduledTask():

```
base.scheduledAtFixedDelay("scheduler_name", 3000, log_info)
```

num 当前执行的次数

def log_info(num):

tc = base.traceContext()

```
base.logInfo(tc, f"logInfo={num}")
```

robustExecute

TODO

runOnce

执行一次，同步方法，构建步骤时需要一些上下文环境，为避免改变， 比如某个变量的内容变换很频繁，防止在执行此方法的时候，数据发生改变

'''

参数1: tc

参数2: BOX-01（上游单号id，或者机器人id等，无限制，str 类型，承上启下）

参数3: 执行任务（任务描述）

参数4: locked （执行）

'''

base.runOnce(tc, "BOX-01", "执行任务", log_info)

def log_info():

tc = base.traceContext()

base.logInfo(tc, f"logInfo={num}")

withLock

方法同步锁，如果存在多个地方，多线程同时调用此方法的时候，每次只会允许一个线程执行该方法，将其他线程阻塞在队列中，随机执行，哪个线程抢到资源，则执行。 考虑到可能需要获取上游的参数，可以通过闭包的形式，即 在函数中定义内部函数

def boot():

tc = base.traceContext()

base.logInfo(tc, "准备执行")

def log_info():

base.logInfo(tc, "aaa方法")

'''

参数1: 同步锁（描述）

参数2: log_info （执行函数）

'''

```
base.withLock("同步锁", log_info)
```

base.logInfo(tc, "执行完成")

withFairnessLock

公平同步锁，如果存在多个地方，多线程同时调用此方法的时候，每次只会允许一个线程执行该方法，将其他线程阻塞在队列中，排队按照时间顺序执行。 考虑到可能需要获取上游的参数，可以通过闭包的形式，即 在函数中定义内部函数

def boot():

tc = base.traceContext()

base.logInfo(tc, "**准备执行**")

def log_info():

base.logInfo(tc, "**aaa方法**")

'''

参数1: 同步锁（描述）

```
参数2：是否开启公平锁，选择 False 则与 withLock 无区别
```

参数2: log_info （执行函数）

'''

```
base.withFairnessLock("公平同步锁", True ,log_info)
```

base.logInfo(tc, "**执行完成**")

parseJsonString

Jons 格式的字符串数据转 Json 类型

s = base.parseJsonString('{"A": 1, "B": 2}') base.logInfo(tc, f"parseJsonString={s}")

jsonToString

Json 类型转 字符串,

json 类型的转换 请勿使用 Python 自带的 str(xxx)，

因为它会吧 json 中的 " 双引号 转为 ' 单引号， 导致问题

s = base.jsonToString({"A": 1, "B": 2}) base.logInfo(tc, f"jsonToString={s}")

xmlToJsonStr

xml 格式 转 json 格式的字符串

now

获取系统当前时间

d = base.now()

throwBzError

脚本抛异常，停止运行，如：

查询库位，如果发现没有此库位信息，直接异常报错

bin = None

if bin is None:

```
base.throwBzError("当前库位不存", None)
```

assignJavaMapToJsMap

TODO

setScriptButtons

注册脚本按钮，前端点击按钮后，执行指定的脚本方法

按钮展示在 运维中心 中

image.png

java.py 中已经封装，直接调用 setScriptButtons即可 注意：注册脚本必须写在 boot 中

def boot():

setScriptButtons(

```
    [ScriptButton("输出脚本版本", "show_script_version", "确定要看吗"),
     ScriptButton("库位", "get_bin", "获取库位", None, "FbBin", 1600)])
```

def show_script_version(input_parameter): return base.jsonToString({"message": f"脚本版本1.0}"})

def get_bin(input_parameter):

```
return base.jsonToString({"message": f"{input_parameter}"})
```

setGlobalValue

设置脚本全局变量，作用域是整个程序，整个系统中都可以访问和使用，通常用于数据共享。 注意：重启服务 / 重启脚本后会清空，与猎鹰任务中内置块的全局变量不同（setGlobalVariable）

给 BOX 赋值，为 CK-01

base.setGlobalValue("BOX", "CK-01")

getGlobalValue

获取通过 setGlobalValue 设置脚本全局变量的值

获取 BOX 的值

base.getGlobalValue("BOX")

recordFailure

记录报错信息

createUserNotice

通知用户，消息提醒，在脚本中触发消息提醒，前端消息中提示，指定一个或者多个用户 例子：通知 **admin** 用户，用户点击后弹出 库位 是 CK-01 的信息

'''

参数1: 标题（消息提醒标题，必填）

参数2: ["**admin**"] （提醒人，用户id 必填） 参数3: None | ReadOneEntity （必填，None:点击及确认，ReadOneEntity:点击打开指定的实体的指定编号）

参数4: Fasle （固定值 False，可不填）

参数5: "" （固定值空字符串，可不填）

参数6: FbBin （实体名，如果参数3填写了 ReadOneEntity，则必填，填写 None 可不填） 参数7: CK-01 （编号，如果参数3填写了 ReadOneEntity，则必填，填写 None 可不填）

'''

---

req = UserNoticeReq("标题1", ["**admin**"], "ReadOneEntity", False, "", "FbBin","CK-01")
base.createUserNotice(tc, req)parseJsonString

image.png

image.png

strToMd5

计算字符串的 32 位 MD5 加密哈希值

z代码块

md5 = base.strToMd5(tc, "需要加密的字符串")

线程 thread: ScriptThread

ScriptThread 类提供 thread 方法，其中可以使用 thread.xxx 来执行 ScriptThread 类中的方法

class ScriptThread:

```
def createThread(self, name: str, func: str):
```

pass

def sleep(self, ms: int):

pass

```
def interrupted(self) -> bool:
```

pass

thread: ScriptThread

createThread

创建线程，脚本中主要用作于和 bese的定时器 scheduledAtFixedDelay 方法协同使用，

如: 每隔 10 s，获取系统当前时间

def boot():

'''

参数1: getNow（线程名称）

参数2: get_now （执行方法）

'''

```
thread.createThread("getNow","get_now")
```

def get_now():

```
base.scheduledAtFixedDelay("scheduler_name", 10000, get_time)
```

num 当前执行的次数

def get_time(num):

tc = base.traceContext()

```
base.logInfo(tc, f"now={base.now()}")
```

sleep

暂停，线程暂执行指定的时间后，自动唤醒，继续执行

如：暂停 2 s

thread.sleep(2000)

interrupted

线程是否终断，通常用在循环中，线程在某种情况下循环，通过检查中断状态，线程可以响应外部的中断请求，而不是无限制地继续执行

while(!threadinterrupted):

pass

服务器 httpServer: HttpServer

HttpServer 类提供 httpServer 服务器方法，其中可以使用 httpServer.xxx 来执行 ScriptThread 类中的方法

class HttpServer:

```
def registerHandler(self, method: str, path: str, func: str, auth: bool):
```

pass

class HttpServerContext:

```
def getBodyAsString(self) -> str:
```

pass

def setJson(self, jo):

pass

httpServer: HttpServer

registerHandler

注册接口，当 M4 的标准接口不支持场景的时候，可以该方法，注册定制接口，供外部访问

注：注册类型的方法通常都是在 boot 方法中使用

例：定制一个 Get 和 Post 方法

def boot():

"""

参数1: 请求类型（Get ｜ Post）

参数2: url

参数3: getAB（执行函数）

参数4: False （是否需要认证）

"""

httpServer.registerHandler("Get", "api/ext/log", "getAB", False) httpServer.registerHandler("Post", "api/ext/log", "postAB", False)

"""

ctx: HttpServerContext

setJson : 返回信息

```
getBodyAsString : 请求的正文(会自动转为 字符串 类型)
```

"""

def getAB(ctx):

ctx.setStatus(200)

ctx.setJson({"A": "B"})

def postAB(ctx):

```
req = base.parseJsonString(ctx.getBodyAsString())
```

ctx.setStatus(200)

ctx.setJson(req)

Get

image.png

Post

image.png

客户端 httpClient: HttpClient

httpClient 类提供 HttpClient 方法，其中可以使用 httpClient.xxx 来执行 ScriptThread 类中的方法

class HttpResult:

```
def __init__(self, successful: bool, ioError: bool, ioErrorMsg: str | None,
code: int, bodyString: str | None,
             checkRes: bool | None, checkMsg: str | None):
```

self.successful = successful

self.ioError = ioError

self.ioErrorMsg = ioErrorMsg

self.code = code

self.bodyString = bodyString

self.checkRes = checkRes

self.checkMsg = checkMsg

class HttpClient:

```
def syncCall(self, tc: TraceContext, reqStr: str, okChecker, oStr: str |
None) -> HttpResult:
```

pass

```
def asyncCallback(self, tc: TraceContext, reqStr: str, okChecker: str,
oStr: str | None) -> str:
```

pass

```
# 反复轮训直到成功。使用此方法的好处：不用自己写循环。优化日志，开始结束时打印日
志。中间如果一致是相同结果，不重复打印日志。
# purpose 是目的，写好了方便差错。
# 后面会与 SOC 结合。
def requestJsonUntil(self, method: str, url: str, reqBody: object | None,
headers: object | None,
                     purpose: str, delay: int, check) -> HttpResult:
```

pass

class Method:

Get = 'Get'

Post = 'Post'

Put = 'Put'

Delete = 'Delete'

class ContentType:

Json = 'Json'

Xml = 'Xml'

Plain = 'Plain'

class HttpRequest:

```
def __init__(self, url: str, method: Method, contentType: ContentType,
reqBody: str | None = None,
             headers: object | None = None, basicAuth: object | None =
None, trace: bool = None,
             traceReqBody: bool = None, traceResBody: bool = None, reqOn:
Date = None):
```

self.url = url

self.method = method

```
    self.contentType = contentType
```

self.reqBody = reqBody

self.headers = headers

self.basicAuth = basicAuth

self.trace = trace

```
    self.traceReqBody = traceReqBody
    self.traceResBody = traceResBody
```

self.reqOn = reqOn

class CallRetryOptions:

```
def __init__(self, maxRetryNum: int = None, retryDelay: int = None):
    self.maxRetryNum = maxRetryNum
```

self.retryDelay = retryDelay

class CheckResultResult:

def **init**(self, ok: bool):

self.ok = ok

def httpAsyncCallback(tc: TraceContext, req: HttpRequest, okChecker: str, o: CallRetryOptions | None = None) -> str: return httpClient.asyncCallback(tc, json.dumps(req), okChecker, json.dumps(o) if o is not None else json.dumps({}))

def httpSyncCall(tc: TraceContext, req: HttpRequest, okChecker, o: CallRetryOptions | None = None) -> HttpResult: return httpClient.syncCall(tc, json.dumps(req.**dict**), okChecker, json.dumps(o.**dict**) if o is not None else json.dumps({}))

httpClient: HttpClient

syncCall

同步请求，会等待直到返回或超时，主要用于请求第三方系统接口

例: 获取 M4 库位列表信息,失败后重试 3 次，重试间隔时间 1 s

'''

url: 请求的地址

```
method: 请求类型，可选参数: "Get" | "Post" | "Put" | "Delete"
contentType: 指定响应的 HTTP内容类型 可选参数:  "Json" | "Xml" | "Plain"
reqBody: 请求正文 jsonstring | null，非必填"
```

headers: 请求头，非必填

```
basicAuth:  Basic Auth 验证参数，非必填
```

trace: 是否记录请求 没有默认 None

traceReqBody: 是否记录请求正文，非必填

traceResBody: 是否记录响应正文, 非必填

reqOn : 请求时间，非必填

'''

req = HttpRequest(url="http://127.0.0.1:5800/api/entity/find/page", method=Method.Post, contentType=ContentType.Json, reqBody='{"entityName": "FbBin", "query": null, "pageNo": 1, "pageSize": 50}', headers={"xyy-app-id": "A", "xyy-app-key": "B"}, basicAuth=None, trace=None, traceReqBody=None, traceResBody=None,

reqOn=None)

"""

3: 失败重试次数

1000 : 重试间隔

"""

o = CallRetryOptions(3,1000)

"""

java.py 提供 httpSyncCall 函数封装 同步方法

"""

result = httpSyncCall(tc, req, check, o)

"""

请求结束后，一般需要验证是否成功

"""

if result.successful and result.code == 200: body_string = base.parseJsonString(result.bodyString) if body_string.get("success") == True:

return True

return False

"""

校验函数，返回 False 则需要重试

"""

def check(result: HttpResult): if result.successful and result.code == 200: body_string = base.parseJsonString(result.bodyString) if body_string.get("success") == True:

return True

return False

asyncCallback

异步请求

'''

url: 请求的地址

```
method: 请求类型，可选参数: "Get" | "Post" | "Put" | "Delete"
contentType: 指定响应的 HTTP内容类型 可选参数:  "Json" | "Xml" | "Plain"
reqBody: 请求正文 jsonstring | null，非必填"
```

headers: 请求头，非必填

```
basicAuth:  Basic Auth 验证参数，非必填
```

trace: 是否记录请求 没有默认 None

traceReqBody: 是否记录请求正文，非必填

traceResBody: 是否记录响应正文, 非必填

reqOn : 请求时间，非必填

'''

req = HttpRequest(url="http://127.0.0.1:5800/api/entity/find/page", method=Method.Post, contentType=ContentType.Json, reqBody='{"entityName": "FbBin", "query": null, "pageNo": 1,

"pageSize": 50}', headers={"xyy-app-id": "A", "xyy-app-key": "B"}, basicAuth=None, trace=None, traceReqBody=None, traceResBody=None,

reqOn=None)

"""

3: 失败重试次数

1000 : 重试间隔

"""

o = CallRetryOptions(3,1000)

"""

java.py 提供 httpSyncCall 函数封装 同步方法

check_fuc 校验方法名

"""

result = asyncCallback(tc, req, check, o)

"""

请求结束后，一般需要验证是否成功

"""

if result.successful and result.code == 200: body_string = base.parseJsonString(result.bodyString) if body_string.get("success") == True:

return True

return False

"""

校验函数，返回 False 则需要重试

"""

def check(result: HttpResult): if result.successful and result.code == 200: body_string = base.parseJsonString(result.bodyString) if body_string.get("success") == True:

return True

return False

监控中心 (Old) soc:Soc

Soc 类提供 soc 方法，其中可以使用 soc.xxx 来执行 Soc 类中的方法

class SocAttention(Enum):

NONE = "None"

GREEN = "Green"

YELLOW = "Yellow"

RED = "Red"

class SocNode:

def getGroup(self) -> str:

pass

def getId(self) -> str:

pass

```
def getDescription(self) -> str:
```

pass

def getValue(self) -> object:

pass

```
def getAttention(self) -> SocAttention:
```

pass

```
def getModifiedReason(self) -> str:
```

pass

```
def getModifiedTimestamp(self) -> int:
```

pass

class Soc:

```
def updateStringNode(self, id: str, desc: str, content: str, attention:
SocAttention):
```

pass

```
def updateIntNode(self, id: str, desc: str, content: int, attention:
SocAttention):
```

pass

```
def getNode(self, id: str) -> (SocNode | None):
```

pass

```
def removeNode(self, id: str):
```

pass

soc: Soc

updateStringNode

更新监控中心 Old 节点字符

soc.updateStringNode("test_id","test_dec","一些内容","Yellow")

image.png

updateIntNode

更新监控中心 Old 节点数字

soc.updateIntNode("test_id","test_dec",123,"Red")

image.png

getNode

获取监控中心 Old 节点数据

image.png

group = soc.getNode("test_id").getGroup() id= soc.getNode("test_id").getId() description = soc.getNode("test_id").getDescription() value = soc.getNode("test_id").getValue() attention = soc.getNode("test_id").getAttention() modifiedReason = soc.getNode("test_id").getModifiedReason() modifiedTimestamp = soc.getNode("test_id").getModifiedTimestamp()

打印结果

group = 扩展,id = test_id,description = test_dec,value = 一些内容,attention = Red,modifiedReason = ,modifiedTimestamp = Tue Nov 19 17:55:11 CST 2024

removeNode

移除监控中心 Old 节点

soc.removeNode("test_id")

判断条件 cq : Cq

Cq 类提供 cq 方法，其中可以使用 cq.xxx 来执行 Cq 类中的方法, cq 通常与 entity : ScriptEntity 中使用

class ComplexQuery:

pass

class Cq:

```
def all(self) -> ComplexQuery:
```

pass

```
def cqAnd(self, ls: list) -> ComplexQuery:
```

pass

```
def cqOr(self, ls: list) -> ComplexQuery:
```

pass

```
def eq(self, field1: str, v) -> ComplexQuery:
```

pass

```
def ne(self, field1: str, v) -> ComplexQuery:
```

pass

```
def lt(self, field1: str, v) -> ComplexQuery:
```

pass

```
def lte(self, field1: str, v) -> ComplexQuery:
```

pass

```
def gt(self, field1: str, v) -> ComplexQuery:
```

pass

```
def gte(self, field1: str, v) -> ComplexQuery:
```

pass

```
def idEq(self, id: str) -> ComplexQuery:
```

pass

```
def include(self, field1: str, items: list) -> ComplexQuery:
```

pass

```
def empty(self, field1: str) -> ComplexQuery:
```

pass

```
def notEmpty(self, field1: str) -> ComplexQuery:
```

pass

```
def thisDay(self, field1: str)  -> ComplexQuery:
```

pass

```
def thisWeek(self, field1: str) -> ComplexQuery:
```

pass

```
def currentUser(self, field1: str) -> ComplexQuery:
```

pass

```
def currentUsername(self, field1: str)-> ComplexQuery:
```

pass

```
def containIgnoreCase(self, field1: str, field2: str) -> ComplexQuery:
```

pass

cq: Cq

all

全部，所有

cq.all()

eq

等于，id 等于 A， qty 等于 10

cq.eq("id","A")

cq.eq("qty",10)

ne

不等于，id 不等于 A， qty 不等于 10，locked 不等于 true

cq.ne("id","A")

cq.ne("qty",10)

"""

python 中布尔类型 首字母大写 True False

"""

cq.ne("locked",True)

lt

小于， qty 小于 10

cq.lt("qty",10)

lte

小于等于， qty 小于等于 10

cq.lte("qty",10)

gt

大于， qty 大于 10

cq.gt("qty",10)

gte

大于等于， qty 大于等于 10

cq.gte("qty",10)

idEq

id 等于，id 在 M4 中属于 唯一值，类似于单据编号，物料编码，容器号等，用此方法无需写 id id 等于 LX0001 ， id 等于 A-01-01-01

cq.idEq("LX0001")

cq.idEq("A-01-01-01")

include

包含多值，用于判断数组，qty 包含 10，20

cq.include("qty",[10,20])

cqAnd

组合条件 与， id 等于 A 且 qty 等于 10

cq.cqAnd([cq.eq("id","A"),cq.eq("qty",10)])

cqOr

组合条件 或, ， id 等于 A 或 qty 等于 10

cq.cqOr([cq.eq("id","A"),cq.eq("qty",10)])

empty

空，name 为空, 空包含了 空字符串 ""，和 空 null

cq.empty("name")

notEmpty

不为空，name 不为空

cq.notEmpty("name")

thisDay

日期为今天的，如：查找创建时间为今天

cq.thisDay('createdOn')

thisWeek

日期为这周的

cq.thisWeek('createdOn')

containIgnoreCase

不考虑大小写的查找

cq.containIgnoreCase('name','abc')

增删改查 entity: ScriptEntity

ScriptEntity 类提供 entity 方法查询实体数据，其中可以使用 cq.xxx 来执行 Cq 类中的方法, 与 cq:Cq 配合使用

class CreateOptions:

```
def __init__(self, keepId: bool = None):
```

self.keepId = keepId

class UpdateOptions:

```
def __init__(self, limit: int = None):
```

self.limit = limit

class RemoveOptions:

```
def __init__(self, limit: int = None):
```

self.limit = limit

class FindOptions:

```
def __init__(self, projection: list = None, sort: list = None, skip: int =
None, limit: int = None):
```

self.projection = projection

self.sort = sort

self.skip = skip

self.limit = limit

class ScriptEntity:

```
def createOne(self, tc: TraceContext, entityName: str, evJson: object, o:
CreateOptions | None) -> str:
```

pass

```
def createMany(self, tc: TraceContext, entityName: str, evJsonList: list,
o: CreateOptions | None) -> list:
```

pass

```
def buildCreateOptions(self, keepId: bool) -> CreateOptions:
```

pass

```
def updateOne(self, tc: TraceContext, entityName: str, queryJson:
ComplexQuery, updateJson: object,
              o: UpdateOptions | None) -> int:
```

pass

```
def updateOneById(self, tc: TraceContext, entityName: str, id: str,
updateJson: object,
                  o: UpdateOptions | None) -> int:
```

pass

```
def updateMany(self, tc: TraceContext, entityName: str, queryJson:
ComplexQuery, updateJson: object,
               o: UpdateOptions | None) -> int:
```

pass

```
def buildUpdateOptions(self, limit) -> UpdateOptions:
```

pass

```
def removeOne(self, tc: TraceContext, entityName: str, queryJson:
ComplexQuery):
```

pass

```
def removeMany(self, tc: TraceContext, entityName: str, queryJson:
ComplexQuery, o: RemoveOptions | None) -> object:
```

pass

```
def buildRemoveOptions(self, limit) -> RemoveOptions:
```

pass

```
def count(self, tc: TraceContext, entityName: str, queryJson: ComplexQuery)
-> object:
```

pass

```
def findOne(self, tc: TraceContext, entityName: str, queryJson:
ComplexQuery,
            o: FindOptions | None) -> object | None:
```

pass

```
def findOneById(self, tc: TraceContext, entityName: str, id: str, o:
FindOptions | None) -> object | None:
```

pass

```
def findMany(self, tc: TraceContext, entityName: str, queryJson:
ComplexQuery, o: FindOptions | None) -> list:
```

pass

```
def exists(self, tc: TraceContext, entityName: str, queryJson:
ComplexQuery) -> bool:
```

pass

```
def buildFindOptions(self, projection: list | None, sort: list | None,
skip: int | None,
                     limit: int | None) -> FindOptions:
```

pass

def clearCacheAll(self):

pass

```
def clearCacheByEntity(self, entityName: str):
```

pass

entity: ScriptEntity

createOne

创建单个数据，创建一个类型为 LX 的容器，编号为LX0001，成功返回 id container = {"id":"LX0001","type":"LX"}

"""

参数1:tc

参数2:实体名称

参数3:具体数据

参数4: 默认 None

"""

containerId = entity.createOne(tc, "FbContainer", container, None)

createMany

创建多个数据，创建type为 LX 的容器，id为LX0001和 LX0002,成功返回 id集合 containers = [{"id":"LX0001","type":"LX"},{"id":"LX0002","type":"LX"}]

"""

参数1:tc

参数2:实体名称

参数3:具体数据

参数4: 默认 None

"""

ids = entity.createMany(tc, "FbContainer", containers, None)

updateOne

更新单条数据，更新一个 id 为LX0001 的容器，type 变为 KB

update = {"type":"KB"}

"""

参数1:tc

参数2:FbContainer (实体名称)

参数3: 条件

参数4: 更新内容

参数5: 默认 None

"""

entity.updateOne(tc, "FbContainer", cq.idEq("LX0001"),update, None)

updateOneById

根据id更新，更新 id 为LX0001 的容器，type 变为 KB

update = {"type":"KB"}

"""

参数1:tc

参数2:FbContainer (实体名称)

参数3: id

参数4: 更新内容

参数5: 默认 None

"""

entity.updateOneById(tc, "FbContainer", "LX0001",update, None)

updateMany

更新多条数据，更新 type 为KB 的容器，type 变为 LX

update = {"type":"LX"}

"""

参数1:tc

参数2:FbContainer (实体名称)

参数3: 条件

参数4: 更新内容

参数5: 默认 None

"""

entity.updateMany(tc, "FbContainer", cq.eq("type","KB"),update, None)

removeOne

删除单个数据，删除一个id 为LX0001 的容器

"""

参数1:tc

参数2:实体名称

参数3:条件

参数4: 默认 None

"""

entity.removeOne(tc, "FbContainer", cq.idEq("LX0001"), None)

removeMany

删除多个数据，删除type为 LX 的容器

"""

参数1:tc

参数2:实体名称

参数3:条件

参数4: 默认 None

"""

entity.removeMany(tc, "FbContainer", cq.eq("type","LX"), None)

findOne

查询单条数据，查询成功返回查询的数据信息，没有则返回 None

查询一个 id 为LX0001 的容器

获取一个 type 为 KB且 locked 不为 True 的容器

"""

参数1:tc

参数2:FbContainer (实体名称)

参数3: 条件

参数4: 默认 None

"""

container = entity.findOne(tc, "FbContainer", cq.idEq("LX0001"), None) c = entity.findOne(tc, "FbContainer", cq.cqAnd([cq.eq("type","KB"),cq.ne("locked",True)]), None)

findOneById

根据id查询，查询 id 为LX0001 的容器，查询成功返回查询的数据信息，没有则返回 None

"""

参数1:tc

参数2:FbContainer (实体名称)

参数3: id

参数4: 默认 None

"""

container = entity.findOneById(tc, "FbContainer", "LX0001", None)

findMany

查询多条数据，查询 type 为KB 的容器，查询成功返回查询的数据信息集合，没有则返回空集合

"""

参数1:tc

参数2:FbContainer (实体名称)

参数3: 条件

参数4: 默认 None

"""

containers = entity.findMany(tc, "FbContainer", cq.eq("type","KB"), None)

exists

数据是否存在

查询 id 为 LX0001的容器是否存在，如果存在 isExist 返回 True ， 不存在 返回 False

"""

参数1:tc

参数2:FbContainer (实体名称)

参数3: 条件

参数4: 默认 None

"""

isExist = entity.exists(tc,"FbContainer", cq.idEq("LX0001"), None)

clearCacheAll

清理 M4 全部缓存数据

entity.clearCacheAll()

clearCacheByEntity

清理 M4 指定的实体缓存缓存数据，清理容器的缓存数据

entity.clearCacheByEntity("FbContainer") 实体拦截：entityExt: ScriptEntityExt

ScriptEntityExt 类提供 entityExt 方法查询实体数据，主要用作与 实体对象的 增，删，改 的前后的方法。

注意：注册实体拦截器必须要需要在 boot 方法中

比如：

创建出库单，创建前需要逻辑校验，校验不通过则直接报错，不创建。 创建出库单，出库单创建后，需要触发某些特殊逻辑。

class ScriptEntityExt:

注意实体生命周期拦截器：实体新建前

```
def extBeforeCreating(self, entityName: str, func: str):
```

pass

注意实体生命周期拦截器：实体新建后

```
def extAfterCreating(self, entityName: str, func: str):
```

pass

注意实体生命周期拦截器：实体更新前

```
def extBeforeUpdating(self, entityName: str, func: str):
```

pass

注意实体生命周期拦截器：实体更新后

```
def extAfterUpdating(self, entityName: str, func: str):
```

pass

实体删除前

```
def extBeforeRemoving(self, entityName: str, func: str):
```

pass

实体删除后

```
def extAfterRemoving(self, entityName: str, func: str):
```

pass

entityExt: ScriptEntityExt

extBeforeCreating

实体新建前，创建前，可以校验数据，报错直接结束，不创建实体，以及追加数据 创建实体 FbOutboundOrder，如果 container 字段没有传，则报错，传了则remark字段赋值 A 注意：如果 line 中没有这个字段，或者字段为 null，用 line.get("xxx") is None 会返回 Fasle

def boot():

"""

参数1: 实体名称

参数2: 执行函数

"""

```
entityExt.extBeforeCreating("FbOutboundOrder","checkOrder")
```

"""

固定输入

tc

em : 实体配置

ls : 创建的内容，可能是批量创建所以是 list

如果报错，则需要返回 M4 根据 error 是 False 还是 True 来判断是否报错

"""

---

def checkOrder(tc: TraceContext, em: EntityMeta, ls: list): if em.getName() != "FbOutboundOrder": return base.jsonToString({"error": True, "message": "实体错误"})

for line in ls:

if line.get("container"):

line["remark"] = "A"

else:

```
        return base.jsonToString({"error": True, "message": "容器不能为空"})
```

extAfterCreating

实体创建后,执行此方法定制逻辑。 创建容器 FbContainer创建后，如果 容器类型字段 type，在实体容器类型 FbContainerType中没有，则创建对应的容器类型

def boot():

"""

参数1: 实体名称

参数2: 执行函数

"""

```
entityExt.extAfterCreating("FbContainer","createContainer")
```

"""

固定输入

tc

em : 实体配置

evList : 创建的内容，可能是批量创建所以是 list

"""

def createContainer(tc: TraceContext, em: EntityMeta, evList: list):

for line in evList:

```
    containerType = entity.findOneById(tc, "FbContainerType",
line.get("id"), None)
```

if containerType:

base.logInfo(tc, "类型存在")

else:

base.logInfo(tc, "类型不存在，创建")

t = {"id": line.get("id")}

```
        entity.createOne(tc, "FbContainerType", t, None)
```

extBeforeUpdating

实体更新前，更新前，可以校验数据，报错直接结束，不更新

更新库位 FbBin，把 锁定locked 更新为 True ， 如果更新的库位已经锁了，则报错

def boot():

"""

参数1: 实体名称

参数2: 执行函数

"""

```
entityExt.extBeforeUpdating("FbBin","checkBin")
```

"""

固定输入

tc

em : 实体配置

ids : 更新的 id 集合，可能更新多个所以是 list update : 更新的内容如果报错,则需要返回,M4 根据 error 是 False 还是 True 来判断是否报错

"""

def checkBin(tc: TraceContext, em: EntityMeta, ids: list[str], update): bins = entity.findMany(tc, em.getName(), cq.include("id", ids), None)

for bin in bins:

if bin.get("locked"):

```
        return base.jsonToString({"error": True, "message": f"容器
{bin['id']}被锁,                        不能重复锁定"})
```

extAfterUpdating

实体更新后,执行此方法定制逻辑。 更新容器 FbContainer，字段有货 filled 设置为 False 后，清理容器中的库存，删除库存明细FbInvLayout 中容器字段 leafContainer包含的 ids 的库存

def boot():

"""

参数1: 实体名称

参数2: 执行函数

"""

```
entityExt.extAfterUpdating("FbContainer","updateContainer")
```

"""

固定输入

tc

em : 实体配置

ids : 更新的 id 集合，可能更新多个所以是 list

oldValues : 更新前的数据集合

newValues : 更新后的数据集合

"""

def updateContainer(tc: TraceContext, em: EntityMeta, ids: list, oldValues:list, newValues: list):
entity.removeMany(tc,"FbInvLayout",cq.include("leafContainer",ids),None)

extBeforeRemoving

删除前，实体删除前，可以校验数据，报错直接结束，不删除

删除库存明细 FbInvLayout，把如果库存锁定字段locked 更新为 True ， 则报错，不删除

def boot():

"""

参数1: 实体名称

参数2: 执行函数

"""

```
entityExt.extBeforeRemoving("FbInvLayout","checkInv")
```

"""

固定输入

tc

em : 实体配置

ids : 删除的 id 集合，可能删除多个所以是 list

"""

def checkInv(tc: TraceContext, em: EntityMeta, ids: list[str]): invs = entity.findMany(tc, em.getName(), cq.include("id", ids), None)

for inv in invs:

if inv.get("locked"):

```
        return base.jsonToString({"error": True, "message": f"库存
{inv['id']}已经被锁                    , 不能删除"})
```

extAfterRemoving

删除后，实体删除后，执行此方法定制逻辑。 删除库存明细 FbInvLayout后，如果删除的库存明细的容器字段leftContainer,再库存明细中没有其他的库存，则把容器FbContainer的有货字段filled 设置为False

def boot():

"""

参数1: 实体名称

参数2: 执行函数

"""

```
entityExt.extAfterRemoving("FbInvLayout","moveInvLayout")
```

"""

固定输入

tc

em : 实体配置

oldValues : 删除前的数据集合

"""

def moveInvLayout(tc: TraceContext, em: EntityMeta, oldValues: list[str]): containers = [inv["leafContainer"] for inv in oldValues if "leafContainer" in inv]

for c in containers:

```
    invLayout = entity.findOne(tc, "FbInvLayout", cq.eq("leafContainer",
c), None)
```

if not invLayout:

update = {"filled": False}

```
        entity.updateOneById(tc, "FbContainer", c, update, None)
```

猎鹰任务 falcon: ScriptFalcon

ScriptFalcon 类提供 falcon 方法，其中可以使用 falcon.xxx 来执行 ScriptFalcon 类中的方法

class ScriptFalcon:

```
def runTaskByLabelAsync(self, tc: TraceContext, defLabel: str,
jsInputParams: object) -> str:
```

pass

```
def setGlobalVariable(self, tc: TraceContext, key: str, value: object |
None):
```

pass

```
def getGlobalVariable(self, tc: TraceContext, key: str) -> object | None:
```

pass

falcon: ScriptFalcon

setGlobalVariable

设置猎鹰全局变量，作用域是整个程序，整个系统中都可以访问和使用，功能与【猎鹰：设置全局变量组件】作用相同，通常用于数据共享。 注意：重启服务后会清空，与脚本全局变量 base.setGlobalValue 不同；

给 BOX 赋值，为 CK-01

falcon.setGlobalVariable(tc,"BOX","CK-01")

getGlobalVariable

获取通过 setGlobalVariable 设置的猎鹰全局变量的值

获取 BOX 的值

value = falcon.getGlobalVariable(tc,"BOX")

runTaskByLabelAsync

执行猎鹰任务，在脚本中，可以通过此方式直接创建一条猎鹰任务并执行

inputParams = {"bin":"A"}

"""

tc

测试任务:猎鹰任务模板名

inputParams:猎鹰任务的输入参数

falconId: 创建并执行的猎鹰任务编号

"""

falconId = falcon.runTaskByLabelAsync(tc,"测试任务",inputParams) (旧)机器人信息 wcs: ScriptWcs(4.23.xx 以上版本弃用)

ScriptWcs 类提供 wcs 方法，其中可以使用 wcs.xxx 来执行 ScriptWcs 类中的方法，ScriptWcs 主要作用于 光通许场景下。

class MrRobotAlert:

```
def __init__(self, level: str, code: str | None, message: str | None,
times: int | None, timestamp: object | None):
```

self.level = level

self.code = code

self.message = message

self.times = times

self.timestamp = timestamp

class MrRobotSelfReportMain:

```
def __init__(self, battery: float | None, x: float | None, y: float | None,
direction: float | None,
             currentSite: str | None, blocked: bool | None, charging: bool
| None, currentMap: str | None,
             currentMapMd5: str | None, alerts: list[MrRobotAlert] | None):
```

self.battery = battery

self.x = x

self.y = y

self.direction = direction

```
    self.currentSite = currentSite
```

self.blocked = blocked

self.charging = charging

self.currentMap = currentMap

```
    self.currentMapMd5 = currentMapMd5
```

self.alerts = alerts

class MrRobotSelfReport:

```
def __init__(self, error: bool = None, errorMsg: str | None = None, main:
MrRobotSelfReportMain | None = None,
             selfReport: object | None = None, rawReport: object | None =
None, ):
```

self.error = error

self.errorMsg = errorMsg

self.main = main

self.selfReport = selfReport

self.rawReport = rawReport

class MrRobotInfoAll:

```
def __init__(self, id: str, systemConfig: object, runtimeRecord: object |
None,
             selfReport: MrRobotSelfReport | None,
             online: bool, lockedSiteIds: list[str]):
```

self.id = id

```
    self.systemConfig = systemConfig
    self.runtimeRecord = runtimeRecord
```

self.selfReport = selfReport

self.online = online

```
    self.lockedSiteIds = lockedSiteIds
```

class BinRobotArgs:

```
def __init__(self, bin: str, action: str, site: str, fields: object,
rbkFields: object):
```

self.bin = bin

self.action = action

self.site = site

self.fields = fields

self.rbkFields = rbkFields

class ScriptWcs:

机器人是否在线

```
def isRobotOnline(self, robotName: str) -> bool:
```

pass

```
def mustGetRobotInfoAll(self, id: str) -> MrRobotInfoAll:
```

pass

寻找离机器人最近的通讯点

```
def findClosestRobotConnectedPoint(self, robotName: str) -> object | None:
```

pass

```
def buildHaiMoves(self, tc: TraceContext, robotId: str, rawSteps: str) ->
list:
```

pass

```
def buildSeerMoves(self, tc: TraceContext, robotId: str, rawSteps: str) ->
list:
```

pass

未禁用、可接单、在线的机器人

```
def listWorkableRobots(self) -> list[MrRobotInfoAll]:
```

pass

未禁用、可接单、在线、空闲（扩展任务状态）的机器人

```
def listExtIdleRobots(self) -> list[MrRobotInfoAll]:
```

pass

采用RunOnce 的方式，创建直接运单，并等待其完成

```
def awaitRunOnceDirectRobotOrder(self, tc: TraceContext, robotName: str,
mainId: str, action: str,
```

moves: list) -> str:

pass

```
def unlockByRobot(self, tc: TraceContext, robotId: str):
```

pass

```
def tryLockOneSiteByName(self, tc: TraceContext, robotId: str, siteIds:
list) -> str | None:
```

pass

```
# 重置所有扩展机器人，终止所有后台任务，清楚所有地图资源锁
def resetExtRobots(self, tc: TraceContext):
```

pass

```
def getCost(self,startSiteId:str,endSiteId:str)->float:
```

pass

wcs: ScriptWcs

isRobotOnline

判断机器人是否在线，在线返回 True ， 不在线返回 Fasle

"""

Box-01:机器人编号

"""

onLine = wcs.isRobotOnline("Box-01")

mustGetRobotInfoAll

获取机器人信息

"""

Box-01:机器人编号

"""

robot = mars.mustGetRobotInfoAll("Box-01","Main") findClosestRobotConnectedPoint 获取机器人最近的光通讯点：此方法能在光通讯场景下使用，同时需要在实体 RobotConnectedPoint

中维护光通讯点位

"""

Box-01:机器人编号

"""

onLine = wcs.findClosestRobotConnectedPoint("Box-01")

buildHaiMoves

控制海柔 料箱车执行动作，只适用于 海柔 的料箱车及海柔 的控制器。

动作: Move-移动，Load-取货，Unload-放货

控制海柔 料箱车 Box-01移动到 坐标 x 为 20，y为 20，移动方向 为 0 控制海柔 料箱车 Box-01到 库位 RK-01 取容器 LX0001到机器人背篓位置 0 上，x，y，position 都维护在库位 FbBin中

"""

tc: tc

Box-01:机器人编号

req : operation-动作，position-点位坐标 req1 : operation-动作，bin-M4库位编号

"""

req = base.jsonToString([{"operation": "Move", "position": {"x": 20, y: 20,

"direction": 0.0}}])

req1 = base.jsonToString([{"operation": "Load", "bin": "RK-01", "container": "LX0001", "robotBin": 0]})

tc = base.traceContext()

wcs.buildHaiMoves(tc,"Box-01",req)

buildSeerMoves

控制仙工 机器人执行动作，只适用于 仙工 的控制器。

控制仙工 机器人 Box-01移动到 库位RK-01

"""

---

tc: tc

Box-01:机器人编号

moves : id-点位或者库位，position-动作

"""

---

moves = base.jsonToString([{ "id": "RK-01", "operation": "Wait" }])

tc = base.traceContext()

wcs.buildSeerMoves(tc,"Box-01",moves)

listWorkableRobots

获取未禁用、可接单、在线的机器人

robots = wcs.listWorkableRobots()

listExtIdleRobots

未禁用、可接单、在线、空闲（扩展任务状态）的机器人

robots = wcs.listExtIdleRobots()

awaitRunOnceDirectRobotOrder

RunOnce 的方式，创建直接运单，并等待其完成，简单项目推荐使用猎鹰任务

"""

tc

3066 :

main-1:任务编号

测试:标签

True： 3066， False：3051

"""

robots = wcs.awaitRunOnceDirectRobotOrder(tc, "main-1", "测试", True, [{

id: "CP410"}])

unlockByRobot

释放机器人占用的资源，在光通讯场景下，径路规划后，机器人会占用锁定路径，避免被其他机器人的任务分配导致撞车等，此方法为主动释放机器人当前锁定的资源

"""

tc

Box-01: 机器人编号

"""

wcs.unlockByRobot(tc, "Box-01")

tryLockOneSiteByName

寻找可用的点位，光通讯场景下，机器人完成任务后，需要去停靠点去停靠，通过输入多个点位，得到一个可用的点位

"""

tc

Box-01: 机器人编号

["CP1,CP2,CP3"]:停靠点位集合

"""

site = wcs.tryLockOneSiteByName(tc, "Box-01"， ["CP1,CP2,CP3"])

resetExtRobots（慎用）

重置所有扩展机器人，终止所有后台任务，清除所有地图资源锁

wcs.resetExtRobots(tc)

getCost

从 A 点到 B 点的权重，权重越小距离越近。（不包含旋转，转角等），参数可以传递库位和地图中的点位。 场景举例：从起点A 要去多个库位取货，优先去哪个。

备注：如果不在同一个巷道，可能会有误差

bins = ["B","C","D","E","F","G"]

for bin in bins:

w = wcs.getCost("A",bin)

```
    base.logInfo(tc, f"A 到 {bin} 的库位成本为 {w}")
```

PLC：plc: ScriptPlc

ScriptPlc 类提供 plc 方法，其中可以使用 plc.xxx 来执行 ScriptPlc 类中的方法，目前支持 modbus 和 S7 协议。

class ModbusReadReq:

```
def __init__(self, code: int, address: int, qty: int, slaveId: int | None =
None, maxRetry: int | None = None,retryDelay: int | None = None):
```

self.code = code # 功能码

```
    self.address = address # 开始读取的地址位
```

self.qty = qty # 读取地址位个数

```
    self.slaveId = slaveId # 从站 id
    self.maxRetry = maxRetry # 最大重试次数，可不指定
    self.retryDelay = retryDelay # 重试间隔，可不指定
```

class ModbusWriteReq:

```
def __init__(self, code: int, address: int, qty: int, slaveId: int | None =
None, maxRetry: int | None = None,
             retryDelay: int | None = None):
```

self.code = code # 功能码

```
    self.address = address # 开始写的地址位
```

self.qty = qty # 写地址位个数

```
    self.slaveId = slaveId # 从站 id
    self.maxRetry = maxRetry  # 最大重试次数，可不指定
    self.retryDelay = retryDelay # 重试间隔，可不指定
```

class BlockType(Enum):

DB = 'DB'

Q = 'Q'

I = 'I'

M = 'M'

V = 'V'

class DataType(Enum):

BOOL = 'BOOL'

BYTE = 'BYTE'

INT16 = 'INT16'

UINT16 = 'UINT16'

INT32 = 'INT32'

UINT32 = 'UINT32'

FLOAT32 = 'FLOAT32'

FLOAT64 = 'FLOAT64'

STRING = 'STRING'

class S7ReadReq:

```
def __init__(self, blockType: BlockType, dataType: DataType, dbId: int,
byteOffset: int, bitOffset: int,maxRetry: int | None = None, retryDelay:
int | None = None):
    self.blockType = blockType # 类型 BlockType
    self.address = dataType  # 类型 DataType
```

self.dbId = dbId # db 编号

```
    self.byteOffset = byteOffset # 地址位(字节)
    self.bitOffset = bitOffset # 地址位上的第几位
    self.maxRetry = maxRetry # 最大重试次数，可选
    self.retryDelay = retryDelay # 重试间隔，单位 ms
```

class S7WriteReq:

```
def __init__(self, value: object, blockType: BlockType, dataType: DataType,
dbId: int, byteOffset: int,
             bitOffset: int, maxRetry: int | None = None, retryDelay: int |
None = None):
```

self.value = value # 写入值

```
    self.blockType = blockType # s7区类型 BlockType
```

self.address = dataType # 写入值

self.dbId = dbId # db 编号

```
    self.byteOffset = byteOffset # 地址位(字节)
    self.bitOffset = bitOffset # 地址位上的第几位
    self.maxRetry = maxRetry # 最大重试次数，可选
    self.retryDelay = retryDelay # 重试间隔，单位 ms，可选
```

}

---

class ScriptPlc:

```
def modbusRead(self, tc: TraceContext, deviceName: str, reqMap:
ModbusReadReq) -> list[int]:
```

pass

重复读，直到等于某个值；无限尝试

```
def modbusReadUtilEq(self, tc: TraceContext, deviceName: str, reqMap:
ModbusReadReq, targetValue: int,
```

readDelay: int):

---

pass

 # 重复读，直到等于某个值；最多尝试指定次数

```
def modbusReadUtilEqMaxRetry(self, tc: TraceContext, deviceName: str,
reqMap: ModbusReadReq, targetValue: int, readDelay: int, maxRetry: int):
```

pass

```
def modbusWrite(self, tc: TraceContext, deviceName: str, reqMap:
ModbusWriteReq, values: list[int]):
```

pass

```
def s7Read(self, tc: TraceContext, deviceName: str, reqMap: S7ReadReq) ->
object:
```

pass

```
def s7ReadUntilEq(self, tc: TraceContext, deviceName: str, reqMap:
S7ReadReq, targetValue: any, readDelay: int):
```

pass

```
def s7Write(self, tc: TraceContext, deviceName: str, reqMap: S7WriteReq):
```

pass

plc: ScriptPlc

modbusRead

Mobuds 读，读取 Mobuds 某个或多个地址位的值

"""

tc

PLC，在 M4 的实体 PlcDeviceConfig 中 配置数据 id

address : 开始读取的地址位

qty : 读取长度如下，1 则是读取 40005，如果是 2 的话，则是读取 40005 和 40006 俩个地址的值，

返回数据格式为 list

code: 读取指令

"""

vl = plc.modbusRead(tc, "PLC", { "address": 40005, "qty": 1, "code": 0x03 })

modbusReadUtilEq

Mobuds 重复读，直到等于某个值，无限尝试，读取 Mobuds 某个地址的值，如果一致不是想要的，则一致读取直到是需要的值才结束

"""

tc

PLC，在 M4 的实体 PlcDeviceConfig 中 配置数据 id

code: 读取指令

address : 读取的地址位

qty : 固定值 1， 只能读取 1 位

2 : 期望值

1000 : 读取间隔时间单位毫秒

"""

plc.modbusReadUtilEq(tc, "PLC", { code: 0x03, address: 40005,qty: 1 }, 2 1000)

modbusReadUtilEqMaxRetry

Mobuds 重复读，直到等于某个值，最多尝试指定次数，不是需要的继续执行，超过多少次后，直接结束。

"""

tc

PLC，在 M4 的实体 PlcDeviceConfig 中 配置数据 id

code: 读取指令

address : 读取的地址位

qty : 固定值 1， 只能读取 1 位

2 : 期望值

1000 : 读取间隔时间单位毫秒

10 : 最大次数

"""

plc.modbusReadUtilEqMaxRetry(tc, "PLC", { code: 0x03, address: 40005,qty: 1 }, 1000, 10)

modbusWrite

Mobuds 写，给 Mobuds 某个地址为写值

"""

tc

PLC，在 M4 的实体 PlcDeviceConfig 中 配置数据 id

address : 写的地址位

code: 指令

[1]:写的值，数组里面只能写一个参数

"""

plc.modbusWrite(tc, "PLC", { "address": 40005, code: 0x06 },[1])

s7Read

S7读，读取 DB1.1.1 DB1 区 1 地址位的第 1 位(从 0 开始)

tc = base.traceContext()

s7ReqMap = {"blockType": BlockType.DB,"DataType": DataType.BOOL,"dbId": 1, "byteOffset": 1,"bitOffset": 1,"maxRetry": 1,"retryDelay": 2000}

返回值类型和 s7ReqMap.dataType 值相关

s7Res = plc.s7Read(tc, "s7Plc", s7ReqMap);

s7ReadUntilEq

S7 读直到等于

tc = base.traceContext();

s7ReqMap = {

```
"blockType": "DB",
"dataType": "BOOL",
"dbId": 1,
"byteOffset": 1,
"bitOffset": 1,
"maxRetry": 1,
```

"retryDelay": 2000

}

plc.s7ReadUntilEq(tc, "s7Plc", s7ReqMap, true, 2000)

s7Write

S7 写值，向 DB1.1.1 DB1区 1地址位的第 1 位(从 0 开始)写入

tc = base.traceContext()

s7WriteMap = {

```
        "value": True,
        "blockType": BlockType.DB,
        "dataType": DataType.BOOL,
        "dbId": 1,
        "byteOffset": 1,
        "bitOffset": 1,
        "maxRetry": 1,
        "retryDelay": 2000,
```

}

plc.s7Write(tc, "s7Plc", s7WriteMap)

前端弹窗 ui: ScriptUi

ScriptUi 类提供 ui 方法，其中可以使用 ui.xxx 来执行 ScriptUi 类中的方法。

class Mode(Enum):

Edit = 'Edit'

Read = 'Read'

class ScriptUi:

在指定工位电脑上，打开业务对象的界面

```
def openEntityViewPage(self, workSite: str, entityName: str, id: str, mode:
str):
```

pass

ui: ScriptUi

openEntityViewPage

通过监听数据，主动在前端页面，弹出实体窗口。 举例：出库单，料箱LX1111出库搬运到分拣位的时候，分拣位有 PLC 读取料箱号，当监听 分拣位地址 40005 有料箱，为 1，后读取地址 40006，值为料箱号，根据容器号查询到待分拣的分拣单PickOrder 后，主动弹出分拣单。

def boot():

```
thread.createThread("newThread","new_thread")
```

def new_thread():

```
base.scheduledAtFixedDelay("readPLC", 1000, read_PLC)
```

num 当前执行的次数

def read_PLC(num):

tc = base.traceContext()

```
ls = plc.modbusRead(tc, "PLC", { "address": 40005, "qty": 2, "code": 0x03
})
```

c = ""

if ls[0] == 1:

c = "LX" + str(ls[1])

```
pick = entity.findOne(tc, "FbContainer", cq.cqAnd([
        cq.idEq("container",c),cq.idEq("btOrderState","Todo"]), None)
```

if pick:

"""

分拣位：在 M4 前端设置的，具体看下方截图

PickOrder：分拣单实体

pick["id"]：分拣单单号

Mode：打开类型 Edit-编辑，Read-只读

"""

```
    ui.openEntityViewPage("分拣位", "PickOrder", pick["id"], Mode.Read.name)
```

工位配置

image.png

效果

image.png

bz: Bz

Bz 类提供 bz 方法，其中可以使用 bz.xxx 来执行 Bz 类中的方法。

class Bz:

```
def tryCallContainer(self, tc: TraceContext):
```

pass

```
def finishPutOrderByContainer(self, tc: TraceContext, containerId: str):
```

pass

```
# 修复并创建库存明细。填充物料信息、位置信息、容器等。
def fixCreateInvLayout(self, tc: TraceContext, layouts: list):
```

pass

bz: Bz

tryCallContainer

呼叫空容器，可以通过此方法，执行 M4 标准的叫空容器功能

bz.tryCallContainer(tc)

finishPutOrderByContainer

完成装货单，可以通过此方法，完成 M4 标准的装货单 PutinContainerOrder，装货单通常是通过叫空容器方法生成。

container = "LX0001"

bz.finishPutOrderByContainer(tc，container)

fixCreateInvLayout

创建入库明细，注意：只会创建库存明细，如果有容器和库位的信息，需要用 entity.updateXXX 方法更新容器和库位的信息。

库存明细数据，list 类型

bin = "RK-01"

container = "LX0001"

invs = [{"btMaterial":"AAA","qty":10,"bin":bin,"leafContainer":container}] bz.fixCreateInvLayout(tc, invs)

库位更新容器，设置占用

entity.updateOneById(tc, "FbBin", "RK-01", {"occupied": True, "container": container}, None)

容器更新库位，设置有货

entity.updateOneById(tc, "FbContainer", container, {"filled": True, "bin":bin}, None)

工具类 utils: ScriptUtils

ScriptUtils 类提供 utils 方法，其中可以使用 utils.xxx 来执行 ScriptUtils 类中的方法，提供一些工具方法

class ScriptUtils:

```
def isNullOrBlank(self, string: str | None) -> bool:
```

pass

```
def substringAfter(self, string: str, sep: str) -> str:
```

pass

```
def substringBefore(self, string: str, sep: str) -> str:
```

pass

```
def splitTrim(self, string: str | None, sep: str) -> list:
```

pass

```
# py 中 int 和 long 统一使用 int，在转换的时候可能存在问题，使用的时候先测试一下
def anyToInt(self, v: any) -> int | None:
```

pass

```
def anyToLong(self, v: any) -> int | None:
```

pass

```
# py 中 没有单双号精度  统一使用 float，在转换的时候可能存在问题，使用的时候先测试一下
def anyToFloat(self, v: any) -> float | None:
```

pass

```
def anyToDouble(self, v: any) -> float | None:
```

pass

```
def anyToBool(self, a: any) -> bool:
```

pass

```
def anyToDate(self, input: any) -> Date:
```

pass

format 是 Java 的格式

```
def formatDate(self, d: Date, format: str) -> str:
```

pass

产生 UUID 字符串

def uuidStr(self,) -> str:

pass

产生 ObjectId 字符串

def oidStr(self,) -> str:

pass

utils: ScriptUtils

isNullOrBlank

空判断，判断字符串是否为空

s = None

isNone = utils.isNullOrBlank(s)

substringAfter

字符串截取，截取出第一个指定字符后面的字符串

s = "a1,b2,c3"

```
a = utils.substringAfter(s,",")
```

a 打印结果为 b2,c3

substringBefore

字符串截取，截取出第一个指定字符前面的字符串

s = "a1,b2,c3"

```
a = utils.substringBefore(s,",")
```

a 打印结果为 a1

splitTrim

字符串转换数组，一个有规则的字符串，转为数组

s = "a1,b2,c3"

a = utils.splitTrim(s,",")

 # a 打印结果为 ["a1","b2","c3"]

anyToInt

转数字

s = "1"

i = utils.anyToInt(s)

i 打印结果为 1

anyToLong

转数字，在 Pyhon 中，没有 long 类型，int 只有int 类型，所以与 anyToInt 无区别

s = "1"

i = utils.anyToLong(s)

anyToFloat

转浮点类型

s = "1"

i = utils.anyToFloat(s)

i 打印结果为 1.0

anyToDouble

转浮点类型，在 Pyhon 中 ，没有 double 类型， 只有 float 类型，所以与 anyToFloat 无区别

s = "1"

isNone = utils.isNullOrBlank(s)

anyToBool

转 布尔类型，只要传入的参数，不为None ， 0，fasle-不论大小写，no-不论大小，off-不论大小写，剩余返回都是 True

s = "FALSE"

i = utils.anyToBool(s)

i 打印结果为 True

anyToDate

转日期,根据输入的内容，转为日期，比如外部系统获取的时间为 int ， 我们需要转为时间 ， 毫秒

image.png

s = 1609459200000

t = utils.anyToDate(s)

t 打印结果为 Fri Jan 01 08:00:00 CST 2021

formatDate

转换指定格式，已经获取的日期时间数据格式不是我们想要的，可以通过这个方法转换

s = 1609459200000

t = utils.anyToDate(s)

d = utils.formatDate(t,'yyyy-MM-dd HH:mm:ss.SSS')

d 打印结果为 2021-01-01 08:00:00.000

扩展定制

定制按钮 : CallEntityExtButtonReq

实体的定制按钮输入类，在 M4 中有某些实体信息，需要点击按钮后，触发定制逻辑，比如生成下游单据等，在 M4 的扩展按钮中配置需要执行的函数名 xxx, 输入类型为 CallEntityExtButtonReq。

class CallEntityExtButtonReq:

```
def __init__(self, tc: TraceContext, func: str, entityName: str,
selectedIds: list[str] | None = None, evId: str | None = None,
```

ev: object | None = None):

self.tc = tc

self.func = func # 函数名

```
    self.entityName = entityName # 实体名称
    self.selectedIds = selectedIds # 选择的实体 id 集合，作用于列表页面传参
    self.evId = evId # 实体的 id ，作用于编辑和查看，数据存在且是单条数据
    self.ev = ev # 实体的全部数据，作用于创建和查看
```

函数方法默认需要返回参数，特别注意：

如果返回 modifiedEv 的则直接base.jsonToString("message":"","modifiedEv":req.ev)即可，如果加上了 error则 modifiedEv不生效

'''

error: 是否报错

message : 报错信息，不报错传递 ""

modifiedEv : 只会存在与保存前的按钮里面，不传即可，且不能与 error 一起传

'''

return base.jsonToString({"error":False,"message":"","modifiedEv":req.ev}) 因为 定制按钮 会存在 多种场景，编辑，列表选择，所以提供多个例子： 上游系统调用 M4 标准接口，创建 出库计划单 ，操作员在通过 M4 的出库计划单下推生成 出库单： 点击一个出库计划单进入详情页面，输入下推数量，点击下推按钮，生成已提交的 出库单，如果 输入的下推数量，超过了待下推数量，则提示报错,如果没有则，生成已提交的出库单，且更新待下推数量,如果待下推数量等于出库数，则计划单状态更新为已完成。

实现：

在 M4 中配置编辑页面定制按钮下推，且执行push_plan_order函数。

image.png

push_plan_order实现

def push_plan_order(req):

tc = base.traceContext()

打印数据

```
base.logInfo(req.tc, base.jsonToString(req))
```

lines = []

```
for line in req.ev.get("btLines"):
```

waitQty:待下推数

pushQty:下推数

qty:总数

单行不需要下推，则跳过

if line.get("pushQty") == 0:

continue

待下推数为0，则返回错误

```
    elif line.get("waitQty") == 0:
        return base.jsonToString({"error": True, "message": f'物料
{ev.get("btMaterial")}待下推数为0'})
```

物料下推数大于待下推数，则返回错误

```
    elif line.get("pushQty") > line.get("waitQty"):
        return base.jsonToString({"error": True, "message": f'物料
{line.get("btMaterial")}下推数大于待下推数'})
```

else:

校验无问题

base.logInfo(req.tc, "校验无问题")

lines.append({

```
            "btMaterial": line.get("btMaterial"),
```

"qty": line.get("pushQty")

})

更新待下推数

```
        line["waitQty"] = line.get("waitQty") - line.get("pushQty")
```

line["pushQty"] = 0

创建出库单

if len(lines) > 0:

```
    order = entity.createOne(req.tc, "FbOutboundOrder", {
            "btOrderState": "Committed",
```

"btLines": lines

},None)

is_done = True

检查是否所有物料都已下推

```
    for l in req.ev.get("btLines"):
```

if l.get("waitQty") != 0:

is_done = False

break

如果所有物料都已下推，则更新状态为已完成

if is_done:

```
        req.ev["btOrderState"] = "Done"
    return base.jsonToString({"message": f'下推成功，出库单为
{order}',"modifiedEv": req.ev})
```

else:

```
    return base.jsonToString({"error": True, "message": f'没有需要下推的物
料'})
```

出库计划单列表页面勾选多个单据，点击批量下推按钮，如果选的单据中有已完成的则提示报错，如果没有则根据所有勾选的单据中未完成的单行，生成已提交的 出库单，为计划单剩余的全待下推数，且更新所选单据中单行信息，且计划单状态更新为已完成。 实现：在 M4 中配置列表主页面定制按钮批量下推，且执行push_plan_orders函数。

image.png

push_plan_ordesr实现

def push_plan_orders(req):

打印数据

```
base.logInfo(req.tc, base.jsonToString(req))
```

获取所有待下推的出库单

```
orders = entity.findMany(req.tc, req.entityName,
cq.include("id",req.selectedIds),None)
```

out_lines = []

遍历所有出库计划单

for order in orders:

如果出库计划单已完成，则返回错误

```
    if order.get("btOrderState") == "Done":
        return base.jsonToString({"error": True, "message": f'单据
{order.get("id")}已完成'})
```

else:

lines = order.get("btLines")

for line in lines:

如果物料待下推数大于0，则添加到出库单中

if line.get("waitQty") > 0:

```
                out_lines.append({"btMaterial":
line.get("btMaterial"),"qty": line.get("qty")})
```

line["waitQty"] = 0

```
                line["pushQty"] = line.get("qty")
```

创建出库单

```
o = entity.createOne(req.tc, "FbOutboundOrder", {"btOrderState":
"Committed","btLines": out_lines},None)
```

更新出库计划单状态

for order in orders:

```
    order["btOrderState"] = "Done"
    entity.updateOne(req.tc, req.entityName, cq.idEq(order.get("id")),
{"btOrderState": "Done","btLines": order.get("btLines")},None)
return base.jsonToString({"error": False, "message": f'下推成功，出库单为
{o}'})
```

定制猎鹰任务组件

当猎鹰任务的标准组件无法满足需求上的流程时, 可以使用脚本定制组件的方式来进行开发

例:

顶升车身上料架为一个库区，当顶升车移动的时候，该库区的所有库位不能被其他任务找到，即全部锁定. 定制猎鹰任务组件 M4 支持网页端页面编写方法 和 后台编写.

打开 猎鹰任务模板 菜单, 点击添加定制组件

image.png

配置组件 id (组件名必须唯一), 显示名, 调用的函数名, 脚本语言, 输入输出参数 等,如果在后台编写函数代码, 则 Codes可以不填

image.png

保存后 , 编写猎鹰任务的时候,即可直接使用

image.png

lock_district_all_bin实现:

函数的固定三个输入:

参数1: tc

参数2: params 组件的输入, 如上图输入参数 district 设置为 LX ,则 params 的值为 {"district":"LX"}

参数3: taskInputParams 猎鹰任务的输入

参数4: extExtraCtx 组件的数据集合 , 正常情况下不会用到 , 可以忽略 , 但是不能不加

5.11.2 以上版本新增第四个参数, 如果不加此参数, 则脚本会直接报错,

5.11.2 及以下依旧使用三个参数,如光通讯制定版本 : 4.24.1

函数的固定输出:

如果没有输出, 则可以不 return

如果定制组件设置了输出,正常执行后则都需要返回对应的输出,格式为: return base.jsonToString({"outputParams": {"num": "x"}})

"""

固定三个输入参数

参数1: tc

参数2: 组件的输入, 如上图 district参数的内容为 LX ,则 params 的值为 {"district":"LX"}

参数3: 猎鹰任务的输入

参数4: extExtraCtx 任务块的数据集合 , 正常情况下不会用到 (5.11.2以上版本需要加上)

"""

def lock_district_all_bin(tc, params,taskInputParams):

def lock_district_all_bin(tc, params,taskInputParams, extExtraCtx):

获取要锁定的库区

district = params["district"]

```
num = entity.updateMany(tc, "FbBin", cq.eq("district", district), {
"locked": True }, None)
return base.jsonToString({"outputParams": {"num": num}})
```

执行效果图下:

锁定了 8 个 LX 的空库位

image.png

想要定制组件在不符合条件的时候报错 , 则直接在脚本代码中实现 : return base.jsonToString({"error": True, "errorMsg": "xxx"})

"""

固定三个输入参数

参数1: tc

参数2: 组件的输入, 如上图 district参数的内容为 LX ,则 params 的值为 {"district":"LX"}

参数3: 猎鹰任务的输入

"""

def lock_district_all_bin(tc, params,taskInputParams):

获取要锁定的库区

district = params["district"]

```
num = entity.updateMany(tc, "FbBin", cq.eq("district", district), {
"locked": True }, None)
```

if len(num) <0 :

```
    return base.jsonToString({"error": True, "errorMsg": "没有可以锁定的库
位"})
return base.jsonToString({"outputParams": {"num": num}})
```

(新)机器人信息 mars: MarsScript

在 wcs: ScriptWcs 类中提供有部分函数, 随着 M4 版本迭代已经逐步淘汰 , M4 版本为 4.23.xx 及以上的版本请使用 mars 中的函数. 与 wcs 中同名的函数比较, 表面的体现为多了 sceneName : 场景 字段 , 具体配置在 M4 的 调度场景 菜单中.

class MarsScript:

```
def isRobotOnline(self, robotName: str, sceneName: str | None) -> bool:
```

pass

未禁用、可接单、在线的机器人

```
def listWorkableRobots(self, sceneName: str | None) ->
list[MrRobotInfoAll]:
```

pass

未禁用、可接单、在线、空闲（扩展任务状态）的机器人

```
def listExtIdleRobots(self, sceneName: str | None) -> list[MrRobotInfoAll]:
```

pass

```
def getCost(self, robotName: str, startSiteId: str, endSiteId: str,
sceneName: str | None) -> float:
```

pass

```
def mustGetRobotInfoAll(self, robotName: str, sceneName: str | None) ->
MrRobotInfoAll:
```

pass

```
def findClosestRobotConnectedPoint(self, robotName: str, sceneName: str |
None) -> object | None:
```

pass

```
def buildSeerMoves(self, tc: TraceContext, robotId: str, rawSteps: str,
sceneName: str | None) -> list:
```

pass

```
# 采用 RunOnce 的方式，创建直接运单，并等待其完成。注意如果3066，moves 要填充好路
径
def awaitRunOnceDirectRobotOrder(self, tc: TraceContext, robotName: str,
mainId: str, action: str, seer3066: bool,
```

moves: list) -> str:

pass

```
def unlockByRobot(self, tc: TraceContext, robotId: str, sceneName: str |
None):
```

pass

```
def unlockBySiteIds(self, tc: TraceContext, robotId: str, siteIds:
list[str], sceneName: str | None):
```

pass

```
def tryLockOneSiteByName(self, tc: TraceContext, robotId: str, siteIds:
list[str],
                         sceneName: str | None) -> str | None:
```

pass

```
def resetExtRobots(self, tc: TraceContext, sceneName: str | None):
```

pass

```
def tryBuildSeerMoves(self, tc: TraceContext, robotId: str, rawSteps: str,
sceneName: str | None) -> list:
```

pass

mars: MarsScript

isRobotOnline

判断机器人是否在线，在线返回 True ， 不在线返回 Fasle

"""

Box-01:机器人编号

Main: 场景名

"""

onLine = mars.isRobotOnline("Box-01","Main")

istWorkableRobots （暂不推荐）

获取未禁用、可接单、在线的机器人

robots = mars.listWorkableRobots("Main")

listExtIdleRobots

未禁用、可接单、在线、空闲（扩展任务状态）的机器人

robots = mars.listExtIdleRobots("Main")

awaitRunOnceDirectRobotOrder

RunOnce 的方式，创建直接运单，并等待其完成，简单项目推荐使用猎鹰任务

"""

tc

3066 :

main-1:任务编号

测试:标签

True： 3066， False：3051

"""

robots = mars.awaitRunOnceDirectRobotOrder(tc, "main-1", "测试", True, [{

id: "CP410"}])

unlockByRobot

释放机器人占用的资源，在光通讯场景下，径路规划后，机器人会占用锁定路径，避免被其他机器人的任务分配导致撞车等，此方法为主动释放机器人当前锁定的资源

"""

tc

Box-01: 机器人编号

Main: 场景名

"""

mars.unlockByRobot(tc, "Box-01","Main")

unlockBySiteIds

释放机器人占用的指定点位资源，在光通讯场景下，径路规划后，机器人会占用锁定路径，避免被其他机器人的任务分配导致撞车等，此方法为主动释放机器人当前锁定的指定部分点位资源, 与 unlockByRobot 的区别为, unlockByRobot 释放所有, 而 unlockBySiteIds 只会释放 siteIds 指定的资源

"""

tc

Box-01: 机器人编号

["AP-01,AP-02"]: 指定点位

Main : 场景名

"""

mars.unlockBySiteIds(tc, "Box-01", ["AP-01,AP-02"],"Main")

tryLockOneSiteByName

寻找可用的点位，光通讯场景下，机器人完成任务后，需要去停靠点去停靠，通过输入多个点位，得到一个可用的点位

"""

tc

Box-01: 机器人编号

["CP1,CP2,CP3"]:停靠点位集合

Main: 场景名

"""

site = mars.tryLockOneSiteByName(tc, "Box-01"， ["CP1,CP2,CP3"],"Main")

resetExtRobots（慎用）

重置所有扩展机器人，终止所有后台任务，清除所有地图资源锁

mars.resetExtRobots(tc,"Main")

getCost

从 A 点到 B 点的权重，权重越小距离越近。（不包含旋转，转角等），参数可以传递库位和地图中的点位。 场景举例：从起点A 要去多个库位取货，优先去哪个。

备注：如果不在同一个巷道，可能会有误差

bins = ["B","C","D","E","F","G"]

for bin in bins:

```
    w = mars.getCost("A",bin,"Main")
    base.logInfo(tc, f"A 到 {bin} 的库位成本为 {w}")
```

findClosestRobotConnectedPoint 获取机器人最近的光通讯点：此方法能在光通讯场景下使用，同时需要在实体 RobotConnectedPoint

中维护光通讯点位

"""

Box-01:机器人编号

"""

onLine = mars.findClosestRobotConnectedPoint("Box-01","Main")

buildSeerMoves

控制仙工 机器人执行动作，只适用于 仙工 的控制器。

控制仙工 机器人 Box-01移动到 库位RK-01

"""

tc: tc

Box-01:机器人编号

moves : id-点位或者库位，position-动作

Main: 场景名

"""

moves = base.jsonToString([{ "id": "RK-01", "operation": "Wait" }])

tc = base.traceContext()

mars.buildSeerMoves(tc,"Box-01",moves,"Main")

tryBuildSeerMoves

尝试控制仙工 机器人执行动作一次，只适用于 仙工 的控制器 , 与 buildSeerMoves的区别为 , buildSeerMoves 会尝试规划路径, 直到成功 , 而 tryBuildSeerMoves 只会尝试一次 , 不成功则返回失败

尝试控制仙工 机器人 Box-01移动到 库位RK-01

"""

tc: tc

Box-01:机器人编号

moves : id-点位或者库位，position-动作

Main: 场景名

"""

moves = base.jsonToString([{ "id": "RK-01", "operation": "Wait" }])

tc = base.traceContext()

mars.buildSeerMoves(tc,"Box-01",moves,"Main") (新)三代调度脚本扩展 fleet3: FleetScriptApi

M4 版本需要 >= 5.5.1

class FleetScriptApi:

获取场景下的机器人名

```
def listRobotNames(self, sceneName: str ) -> list[str]:
```

pass

获取机器人上报信息

```
def mustGetRobotRawReport(self, sceneName: str, robotName: str) -> list:
```

pass

更新机器人是否不接单

```
def updateOffDuty(self, sceneName: str, robotName: str, offDuty: bool) ->
None:
```

pass

fleet3: FleetScriptApi

listRobotNames

列出三代调度场景下的机器人名

"""

fleet3Scene: 实际的三代调度场景名

"""

```
robot_names = fleet3.listRobotNames("fleet3Scene")
```

mustGetRobotRawReport

获取机器人上报信息

"""

AMB-01: 机器人名

fleet3Scene: 实际的三代调度场景名

"""

```
robot_raw_report = fleet3.mustGetRobotRawReport("fleet3Scene", "AMB-01")
```

updateOffDuty

更新机器人不接单/接单

"""

AMB-01: 机器人名

fleet3Scene: 实际的三代调度场景名

"""

设置机器人 AMB-01 不接单

```
fleet3.updateOffDuty("fleet3Scene", "AMB-01", True)
```

设置机器人 AMB-01 可接单

```
fleet3.updateOffDuty("fleet3Scene", "AMB-01", False)
```