M4 开发手册

仙工智能科技有限公司

Version 4.20.5, 2024-06-24

目录

一、M4 HTTP 和 WebSocket 文档

1.1 概述

M4 对外提供多种协议的接口。本文档主要讨论 HTTP 和 WebSocket 接口。

1.1.1 HTTP 风格概述

M4 的 HTTP 接口多数的请求和响应正文是 JSON （application/json）。并且根据 HTTP 规范，一定是 UTF-8 编码的（即 application/json; charset=utf-8）。

HTTP 接口参考了 Restful 风格。但考虑到国内的接受程度，未能全面采用 Restful 风格。比如还是尽量使用了 GET 和 POST 请求，少用 PUT/DELETE 请求。

但在 HTTP 响应码上，一般：

- 200 表示请求成功。
- 201 现在已尽量避免使用 201。规范含义是创建成功。但现在尽量都尽量改用 200。
- 400 表示请求错误（Bad Request）。
- 404 表示请求的 URL 不存在。
- 401 表示身份验证失败（未登录）。
- 403 表示已登录但权限不够。
- 500 表示服务器内部错误，即服务器出 BUG 了。
- 502 表示服务不可用。
- 504 表示请求超时。

NOTE	特别注意：按 Restful 规范，比如查询一个订单但不存在，应返回 404。但按国内习惯，还是返回 200；通过响应正文中的字段给出请求的订单是否存在。

对于 400，即请求错误。响应正文是一个 JSON。有 code 和 message 两个字段，给出错误码和错误消息。code 可能为空，但 message 一定有内容。

```
{ "code": "NoSuchRobot", "message": "找不到机器人 '2032'" }
```

1.1.2 时间

为了方便各语音解析，接口返回的日期时间字段，一般是一个 Unix timestamp 长整数，毫秒精度，如 1699200000000。注意需要一个 8 字节整数才能表示。

请求中的日期时间，我们会“尽力解析”，包括 ISO 8601 等。目前支持以下格式：

```
"yyyy-MM-dd'T'HH:mm:ss.SSSXXX",
"yyyy-MM-dd'T'HH:mm:ss.SSSZ",
"yyyy-MM-dd'T'HH:mm:ssX",
"yyyy-MM-dd HH:mm:ss",
"yyyy-MM-dd HH:mm:ss.SSS",
"yyyy-MM-dd HH:mm",
"yyyy-MM-dd",
```

```
"yyyy/MM/dd",
"HH:mm:ss"
```

1.2 安全相关接口

1.2.1 PING 接口

```
GET /api/ping
```

用来检查是否能与服务器通讯并且处于已登录状态。如果未登录（或过期）返回 401。如果已登录，返回用户 ID、用户名等信息。如

```
{
  "id": "__admin__",
  "username": "admin",
  "roAdmin": true,
  "permissions": null
}
```

1.2.2 登录接口

```
POST /api/sign-in
```

请求正文，携带用户名和密码，如：

```
{
  "username": "admin",
  "password": "admin"
}
```

登录成功响应 200，登录失败响应 400，给出错误原因。

登录成功响应的正文中，给出用户 ID，例如，

```
{
    "userId": "__admin__",
    "userToken": "n6F0HgxprbgTvH1AJ95qmQ5H"
}
```

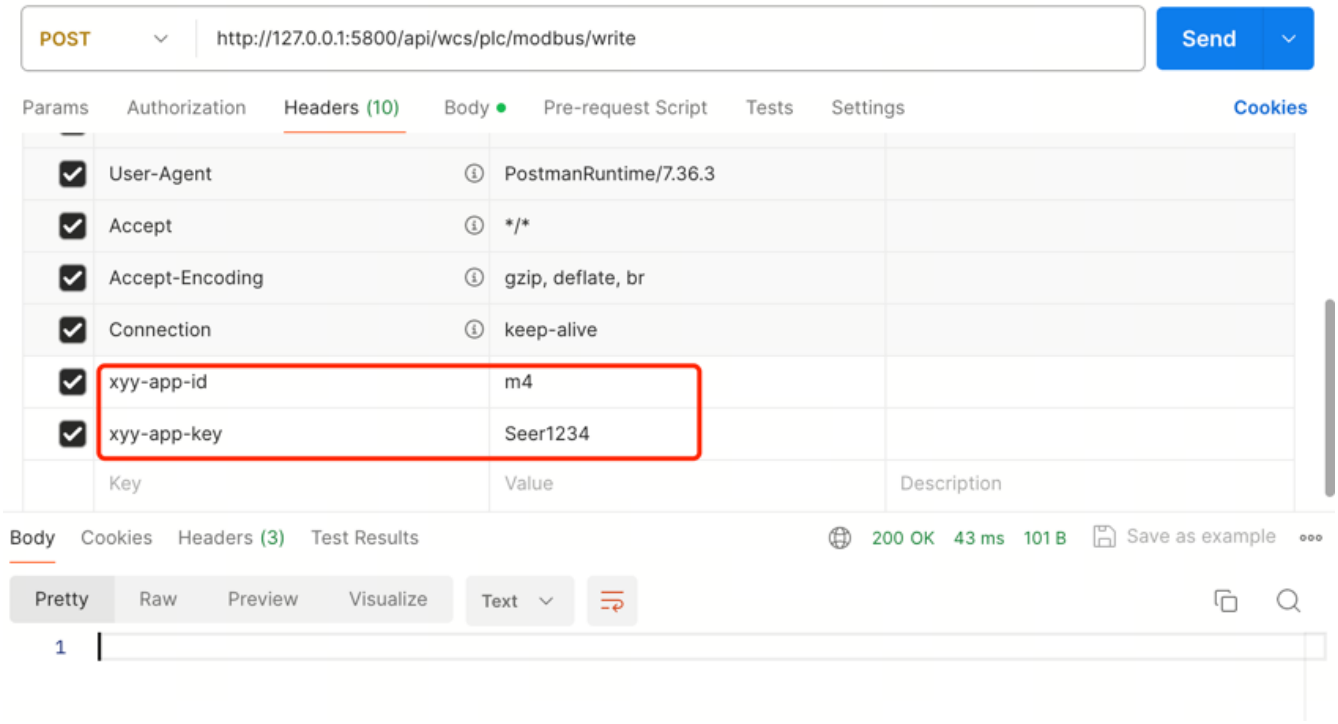
1.2.3 应用安全验证

通过用户名密码登录、获取 token/cookie 适合人类用户使用。对于程序构成的客户端，可以在每次发送请求时携带身份信息，不需要先登录。

身份信息放在 HTTP 请求头上。`xyy-app-id` 给出应用 ID，`xyy-app-key` 给出应用的秘钥。 获取应用 ID 和 密

钥的方式步骤如下图，管理员在 M4 的 代理用户 菜单中添加对应的应用 ID 和密钥。

在请求 M4 的 API 接口中，将其放在请求头中即可，如图：



1.3 业务对象增删改查

M4　系统整体建构在一个低代码系统上。系统内的各种概念，比如用户、库位、机器人、任务、订单等，称为一个业务对象。在国内外软件领域，也称为实体（Entity）或领域对象（Domain Object）。

我们为业务对象提供一批统一的接口，只要接入这套接口，几乎可以查询和修改系统内的所有数据。

1.3.1 查询条件

M4　接口支持构造任意你想表达的查询条件。可以用于分页查询、批量查询，也可以用于更新、删除条件。例如，下面会将批量查询，如果要查询供应商为 **QA0001** 且单据状态为 **Created** 或 **Pending** 的单据：

```
{
        "type": "Compound",
        "items": [
            {
                "type": "General",
```

```
                "field1": "vendor",
                "operator": "Eq",
                "value": "QA0001"
            },
            {
                "type": "General",
                "field1": "status",
                "operator": "In",
                "value": [
                    "Created",
                    "Pending"
                ]
            }
        ]
    }
```

查询条件类似 SQL 查询。有三种类型：全部、复合、通用。

全部，即查询全部数据：

```
{ "type": "All" }
```

复合，可以构造 AND 或 OR 条件。`or` 为 `false` 即是 AND，`or` 为 `true` 即是 OR。

```
{ "type": "Compound", "or": false, "items": [ ... ] }
```

通用，可以将指定字段与特定条件匹配。

```
{
    "type": "General",
    "field1": "vendor",
    "operator": "Eq",
    "value": "QA0001"
}
```

`field1` 即字段名。`operator` 是比较条件。`value` 是比较值。

`operator` 类似 SQL 运算符，可以使用：

- `Eq` 相等
- `Ne` 不等于
- `Gt` 大于
- `Gte` 大于等于
- `Lt` 小于
- `Lte` 小于等于
- `In` 包含多个值，`value` 传一个数组。

- `Between` 用于数字和日期类型，`value` 传一个数组，两个值，闭闭区间。
- `Contain` 包含，用于文本，包含给定文本
- `ContainIgnoreCase` 忽略大小写
- `Start` 用于文本，匹配文本开头
- `End` 用于文本，匹配文本结束
- `Null` 值精确等于 null
- `NotNull` 值精确不等于 null
- `Empty` 用于文本，字段为 null 或是空字符串
- `NotEmpty` 上面的反转
- `CurrentUser` 匹配当前用户 ID，不需要传 `value`
- `CurrentUsername` 匹配当前用户的用户名，不需要传 `value`
- `ThisWeek` 筛选指定日期为本周内

一些列子：

查询我创建的单据：

```
{
    "type": "General",
    "field1": "createdBy",
    "operator": "CurrentUser"
}
```

查询本周创建的单据：

```
{
    "type": "General",
    "field1": "createdOn",
    "operator": "ThisWeek"
}
```

查询 2023-01-01 到 2023-01-31 创建的单据：

```
{
    "type": "General",
    "field1": "createdOn",
    "operator": "Between",
    "value": [ "2023-01-01", "2023-01-31" ]
}
```

1.3.2 分页查询

分页查询，顾名思义，就是分页查询。可以指定查询条件、排序方式、返回部分字段。

```
POST /api/entity/find/page
```

注意，为了携带复杂查询，此接口是 POST 请求不是 GET 请求。

请求正文示例：

```
{
    "entityName": "HumanUser",
    "query": null,
    "pageNo": 1,
    "pageSize": 50,
    "projection": null,
    "sort": [
        "-id"
    ]
}
```

- `entityName` 是业务对象的名字。
- `query` 是查询条件。`null` 表示查询全部。
- `pageNo` 是页码，从 1 开始。
- `pageSize` 是一页几条数据。
- `projection` 可以裁剪业务对象字段。`null` 表示返回全部字段。
- `sort` 是排序方式。

如果想只返回几个字段，可以通过 `projection` 设置：

```
"projection": ["username", "age"]
```

`sort` 支持按多个字段排序。字段名前 `-` 表示倒序。`+` 或空表示正序。例如，先按创建创建倒序排序，再按 id 正序排序。

```
"sort": ["-createdOn", "+id"]
```

响应示例：

```
{
    "pageNo": 1,
    "pageSize": 50,
    "total": 200,
    "page": [
        {
            "username": "test1",
            "email": null,
            "createdBy": "__admin__",
            "createdOn": 1699804800000,
```

```
            "id": "5DB8C406735B40C4A81CB8FCFF8656E8"
        }
    ]
}
```

响应里，`total` 是一共有多条数据。`page` 是当前页数组。每个元素表示业务对象，这里为了文档长度，删除了其他业务对象，只保留了一个。

1.3.3 查询数量

返回符合条件的业务对象数量。

```
POST /api/entity/find/count
```

注意，为了携带复杂查询，此接口是 POST 请求不是 GET 请求。

请求正文示例：

```
{
    "entityName": "HumanUser",
    "query": null
}
```

- `entityName` 是业务对象的名字。
- `query` 是查询条件。`null` 表示查询全部。

响应示例：

```
{"count":10}
```

1.3.4 批量查询

批量返回符合条件的多个业务对象。可以限制最多返回几个。与分页一样，支持排序、筛选字段。

```
POST /api/entity/find/many
```

注意，为了携带复杂查询，此接口是 POST 请求不是 GET 请求。

请求正文示例：

```
{
    "entityName": "HumanUser",
    "query": null,
    "limit": 10,
    "projection": null,
    "sort": [
        "-id"
```

```
    ]
}
```

- `entityName` 是业务对象的名字。
- `query` 是查询条件。`null` 表示查询全部。
- `limit` 是最多返回多少个对象。null 或负数表示不限制。
- `skip` 表示跳过前面的 N 个对象。null 忽略。
- `projection` 可以裁剪业务对象字段。`null` 表示返回全部字段。
- `sort` 是排序方式。

响应示例：

```
[
    {
        "username": "test1",
        "email": null,
        "createdBy": "__admin__",
        "createdOn": 1699804800000,
        "id": "5DB8C406735B40C4A81CB8FCFF8656E8"
    }
]
```

响应是一个数组。

1.3.5 单个查询

单个查询有两种形式，一种是指定 id，一种是指定查询条件。

```
POST /api/entity/find/one
```

注意，为了携带复杂查询，此接口是 POST 请求不是 GET 请求。

请求正文示例：

指定 ID 查询：

```
{
    "entityName": "HumanUser",
    "id": "OKLD2022-01-02-01221",
    "projection": null,
    "sort": [
        "-id"
    ]
}
```

也可以指定查询条件，跟批量查询一样，传 `query` / `skip` 字段。用于实现一些边界问题，如“查询符合条件的第 4 个对象”。

响应正文：

如果有指定的一个对象，返回：

```
{
    "entityValue": {
        "username": "test1",
        "email": null,
        "createdBy": "__admin__",
        "createdOn": 1699804800000,
        "id": "5DB8C406735B40C4A81CB8FCFF8656E8"
    }
}
```

如果找不到，返回：

```
{
    "entityValue": null
}
```

1.3.6 创建单个

```
POST /api/entity/create/one
```

请求正文实例：

```
{
    "entityName": "HumanUser",
    "entityValue": {
        "username": "test2",
        "password": "12345678"
    }
}
```

`entityName` 是业务对象的名字。`entityValue` 是要新增的业务对象本身。

响应是新增业务对象的 ID：

```
{"id":"3524032D91A04700911F55E00BA03294"}
```

1.3.7 创建多个

```
POST /api/entity/create/many
```

请求正文实例：

```
{
    "entityName": "HumanUser",
    "entityValues": [
        {
            "username": "test2",
            "password": "12345678"
        },
        {
            "username": "test3",
            "password": "12345678"
        }
    ]
}
```

entityName 是业务对象的名字。entityValues 是要新增的业务对象数组。

响应是新增业务对象的 ID 列表：

```
{
    "ids": [
        "3524032D91A04700911F55E00BA03294",
        "3524032D91A04700911F55E00BA03295"
    ]
}
```

1.3.8 更新单个

```
POST /api/entity/update/one
```

请求正文实例：

```
{
    "entityName": "HumanUser",
    "id": "3524032D91A04700911F55E00BA03294",
    "update": {
        "roAdmin": false,
        "btDisabled": false,
    }
}
```

entityName 是业务对象的名字。id 是更新对象的 ID。update 是业务对象需要更新的部分字段（对，可以只传需要更新的字段）。

响应是成功更新的对象数量：

```
{"updatedCount":1}
```

1.3.9 更新多个

```
POST /api/entity/update/many
```

请求正文实例：

```
{
    "entityName": "HumanUser",
    "query": {
        "type": "General",
        "field1": "id",
        "operator": "In",
        "value": [
            "5DB8C406735B40C4A81CB8FCFF8656E8",
            "3524032D91A04700911F55E00BA03294"
        ]
    },
    "update": {
        "disabled": true
    }
}
```

`entityName` 是业务对象的名字。`query` 是筛选条件。`update` 是业务对象需要更新的部分字段。此外还可以传 `limit` 字段，限制最多更新几个业务对象。

响应是成功更新的对象数量：

```
{"updatedCount":2}
```

1.3.10 删除单个

```
POST /api/entity/remove/one
```

请求正文实例：

```
{
    "entityName": "HumanUser",
    "id": "3524032D91A04700911F55E00BA03294"
}
```

`entityName` 是业务对象的名字。`id` 是更新对象的 ID。

响应是实际删除的对象数量：

```
{"removedCount":1}
```

1.3.11 删除多个

```
POST /api/entity/remove/many
```

请求正文实例：

```
{
    "entityName": "HumanUser",
    "query": {
        "type": "General",
        "field1": "id",
        "operator": "In",
        "value": [
            "5DB8C406735B40C4A81CB8FCFF8656E8",
            "3524032D91A04700911F55E00BA03294"
        ]
    }
}
```

`entityName` 是业务对象的名字。`query` 是筛选条件。此外还可以传 `limit` 字段，限制最多删除几个业务对象。

响应是实际删除的对象数量：

```
{"removedCount":2}
```

1.3.12 导出 Excel

```
POST /api/entity/export
```

请求正文实例：

```
{
    "entityName": "HumanUser",
    "fields": [
        "id",
        "username",
        "email",
        "truename"
    ],
    "query": null
}
```

`entityName` 是业务对象的名字。`query` 是筛选条件。`fields` 是要导出的字段/列。

响应示例：

```
{"path":"tmp/export1282539A910D42E88AC19FB3B52B8643.xlsx"}
```

path 是文件下载路径。可以用文件下载接口获取。

1.4 文件上传下载

1.4.1 文件上传

```
POST /api/files/upload
```

上传文件采用的格式是 multipart/form-data。文件放在 f0 字段里。一次只能上传一个文件。

响应示例：

```
{
    "originalName": "test.png",
    "size": 7182,
    "path": "upload/202311318/5DAED8FC3BEC498BA42CA36282E1DC0E.png"
}
```

originalName 是文件原始名称。size 是文件大小，单位字节。path 是后续获取此文件的路径。

1.4.2 文件下载/读取

```
GET /api/files/get/<path>
```

path 是之前上传时返回的路径。

1.5 机器人管理相关

1.5.1 获取所有机器人的所有信息

```
GET /api/wcs/mr/fleet/all-all
```

响应是一个数组。每个元素表示一个机器人：

```
{
    "id": "AMB-01", // 机器人 ID

    "systemConfig": {}, // 参见“移动机器人系统配置”业务对象
    "runtimeRecord": {}, // 机器人任务状态
    "selfReport": { // 机器人自身报告，可能为 null，表示没收到任何报文
      "error": false, // true 表示与机器人连接故障
      "errorMsg": "", // 故障详情
```

```
      "main": {}, // 主要信息
      "rawReport": {}, // 机器人报告原始报文
      "timestamp": 1699200000000 // 收到报告的时间
    },
}
```

systemConfig 是机器人系统配置信息。如厂商、停用、不接单、最大载货数、类别、标签等。参见“移动机器人系统配置”业务对象。

runtimeRecord 是机器人任务状态。如机器人身上载货/库位情况、调度执行状态、当前运单等。参见“移动机器人运行记录”业务对象。

main 是机器人报告的主要部分。对所有厂商的机器人做了统一化处理。主要包括：

- battery 电量，从 0 到 1
- x 当前位置 x
- y 当前位置 y
- direction 当前机器人方向
- currentSite 当所在站点/点位（逻辑点位名）
- blocked 被阻挡
- charging 充电中
- selfReport 报警信息
  - level 报警等级
  - code 报警码编号
  - message 报警内容
  - times 出现次数
  - timestamp 报警时间

1.5.2 向系统中添加机器人

```
POST /api/wcs/mr/fleet/add-robots
```

请求正文是一个数组。每个元素表示一个机器人配置。字段参见“移动机器人系统配置”业务对象。

1.5.3 从系统中删除机器人

```
POST /api/wcs/mr/fleet/remove-robots
```

请求正文是一个数组。每个元素是要删除的机器人名（即 ID）。

1.5.4 更新单个机器人配置

```
POST /api/wcs/mr/fleet/update-robot
```

请求正文：

```
{
    "id": "AMB-01", // 机器人 ID
    "systemConfig": { ... }, // 参见“移动机器人系统配置”业务对象
}
```

例如，如果要禁用机器人，只需要：

```
{
    "id": "AMB-01",
    "systemConfig": {
        "disabled": true
     },
}
```

1.6 Modbus HTTP 接口

其他系统调用 M4 的 API 接口的方式，来控制基于 Modbus 协议的外部设备，此类的接口验证可以在 M4 注册 APP ID 来免登录，注册方式请见 1.2.3应用安全验证。

1.6.1 读 Modbus

```
POST /api/wcs/plc/modbus/read
```

请求示例：查询 PLC 中地址位从 35 开始查询 4 位的数值，失败重试 1 次，间隔 1s。

```
{
    "deviceName": "PLC",
    "maxRetry": 1,
    "retryDelay": 1000,
    "commands": [{
         "code": 3,
         "address": 35,
         "qty": 4,
         "slaveId": 0
        }]
```

```
    }
}
```

返回示例：地址位 35、36、37、38 的参数分别为 1、111111、0、0。

```
[
    [1,11111,0,0]
]
```

1.6.2 写 Modbus

```
POST /api/wcs/plc/modbus/write
```

请求示例：PLC 中地址位为 35 写入 1。

```
{
    "deviceName": "PLC",
    "maxRetry": 1,
    "retryDelay": 1000,
    "commands": [{
         "code": 6,
         "address": 35,
         "slaveId": 0,
         "values":[1]
      }]
}
```

1.7 附录 A：如何获取业务对象详情

通过界面，打开菜单《业务对象首页》，在列表中找到想要了解的业务对象。

可以了解业务对象基本详情、字段等信息。

二、M4 脚本开发文档

2.1 基础

通过脚本可以扩展系统功能。例如，定制出库分配库存策略，定制单据保存前的检查逻辑，实现自定义猎鹰任务组件，实现与第三方系统集成，定制一键恢复等功能按钮，等等。

M4 脚本系统目前支持 JavaScript 和 Python 两种语言。 JavaScript 支持大多数最新语言特性（ES2015、ES7）。Python 支持 3.x 语法。

M4 核心系统基于 Java，准确说基于 JVM。不管使用 JavaScript 还是 Python，本质是用脚本语言的语法写 Java / JVM 程序。

2.1.1 脚本目录

一个项目只能采用一种脚本，要么是 JavaScript 要么是 Python。系统启动时，会在在项目的主文件夹下寻找特定文件夹，将文件夹中的文件识别为脚本。

首先会寻找 `scripts-js` 或 `scripts` 文件夹。如果存在，则认为此项目采用 JavaScript 脚本，会把文件夹下的所有 js 文件作为脚本文件聚合加载。

如果找不到，则寻找 `scripts-py` 文件夹。如果存在，如果存在，则认为此项目采用 Python 脚本，会把文件夹下的所有 py 文件作为脚本文件聚合加载。

如果以上文件夹都找不到，则认为此项目不使用脚本。

新建项目时，只需要按上述命名规则先创建文件夹再创建文件即可。

我们推荐使用 JavaScript 语言，因此 `scripts` 文件夹默认为 js 脚本文件夹。

您可以使用您喜欢的任何编辑器开发脚本。如 Visual Studio Code，WebStorm，PyCharm 等。

脚本文件夹里可以有多个脚本文件。它们在执行时会被合并为一个文件再执行。文件名任意。合并时按文件名升序排序合并。由于核心机制限制，不支持 JavaScript 模块系统和 Python 模块系统。

我们推荐有经验的 JavaScript 开发者使用 TypeScript 语言。为此语言，我们提供一个 sdk.ts 文件。里面含有 M4 脚本系统内建变量、函数声明，方便进行代码提示和类型检查。

2.1.2 启动和重启

脚本会在系统启动时自动加载，并执行 `boot()` 函数。

系统关闭时，会执行 `dispose()` 函数。

在界面上，通过《运维中心》，可以在不重启主程序的情况下，重启脚本。会先执行 `dispose()` 函数，在执行 `boot()` 函数。

在脚本里，一些函数名具有特殊意义，如 `boot()` 和 `dispose()`。只要这些函数存在，会在特定时机被系统调用。

2.1.3 示例

目的

- 注册一个下单的接口，让 M4 派遣任意机器人将起点的货物搬运到终点。
- M4 接受到下单请求后，要判断参数的合法性，如果参数异常，则给出对应的提示。

- 下单成功之后，M4 需要将创建的任务的 ID 回传给请求方。
- 机器人将货物放置到终点后，回调接口 `http://localhost:5800/api/ext/mock/taskFinished` ，告知任务结束。

操作思路

- 创建一条猎鹰任务，派遣任意机器人从一个点取货，再去另一个点放货。
- 用 JavaScript 给 M4 注册一个下单接口。
- 用 JavaScript 给 M4 注册一个回调接口。

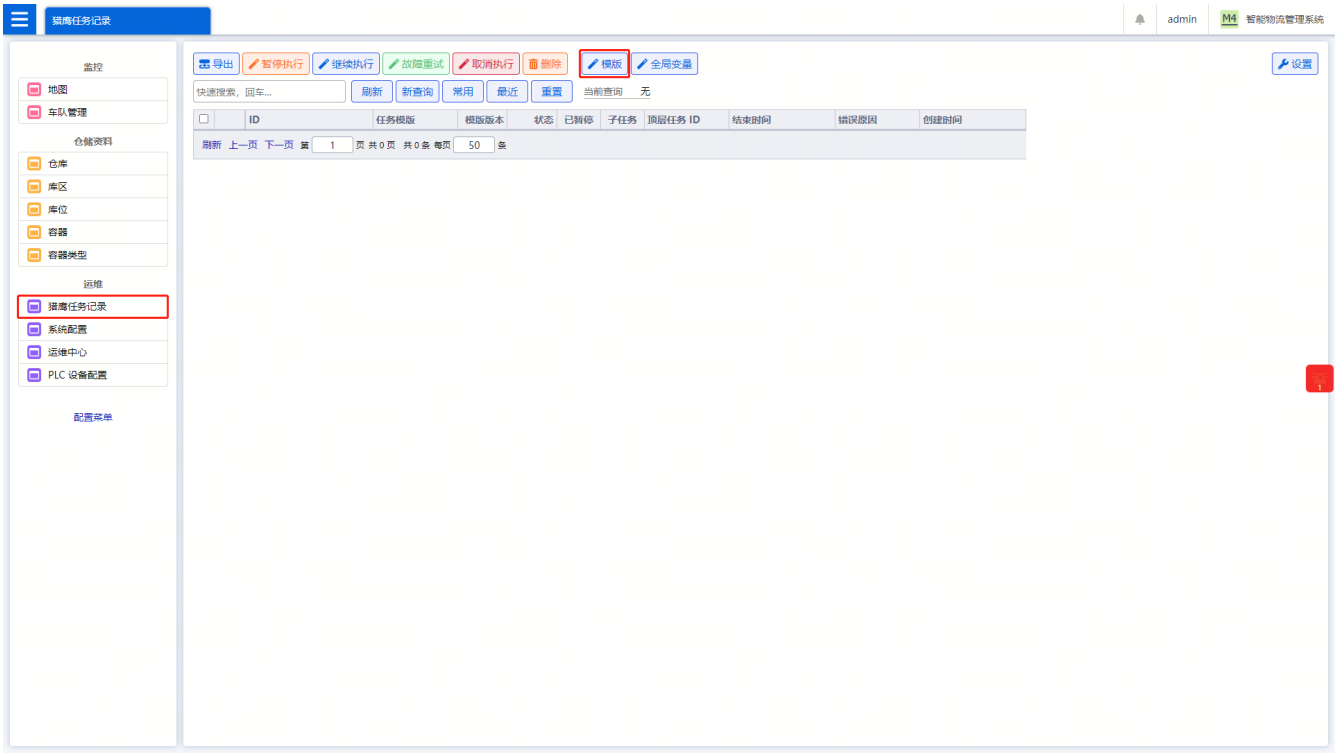
> **NOTE** 本来应该是回调上位机的接口，但是为了更直观地展示，就直接回调 M4 的接口。

- 通过 M4 脚本的 “实体生命周期拦截器” 监听猎鹰任务状态，当上述猎鹰任务结束时，回调指定的接口。

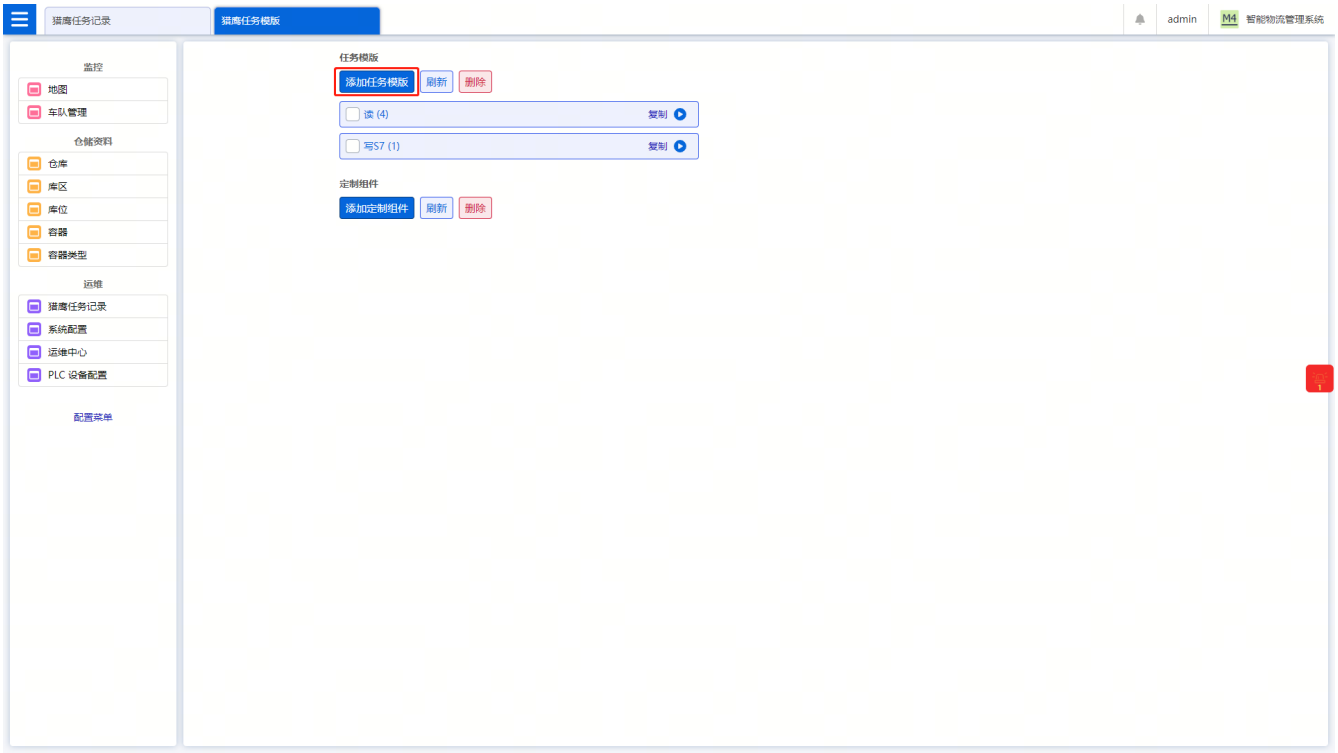
操作步骤

第 1 步：创建猎鹰任务。

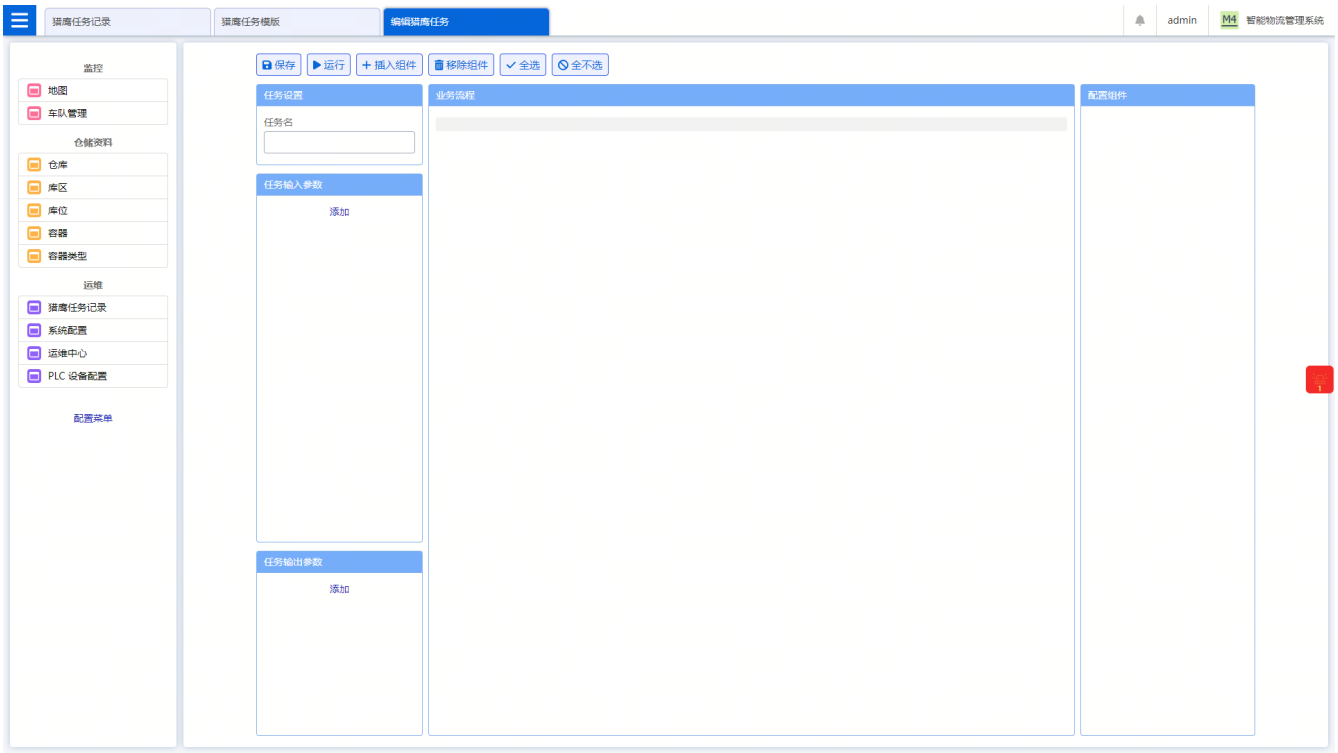
- 登录 M4 的操作界面。
- 然后点击左侧的 “猎鹰任务记录” 菜单。
- 接着点击 “猎鹰任务记录” 界面顶部的 “模板” 按钮，就会跳转到 “猎鹰任务模板” 界面。



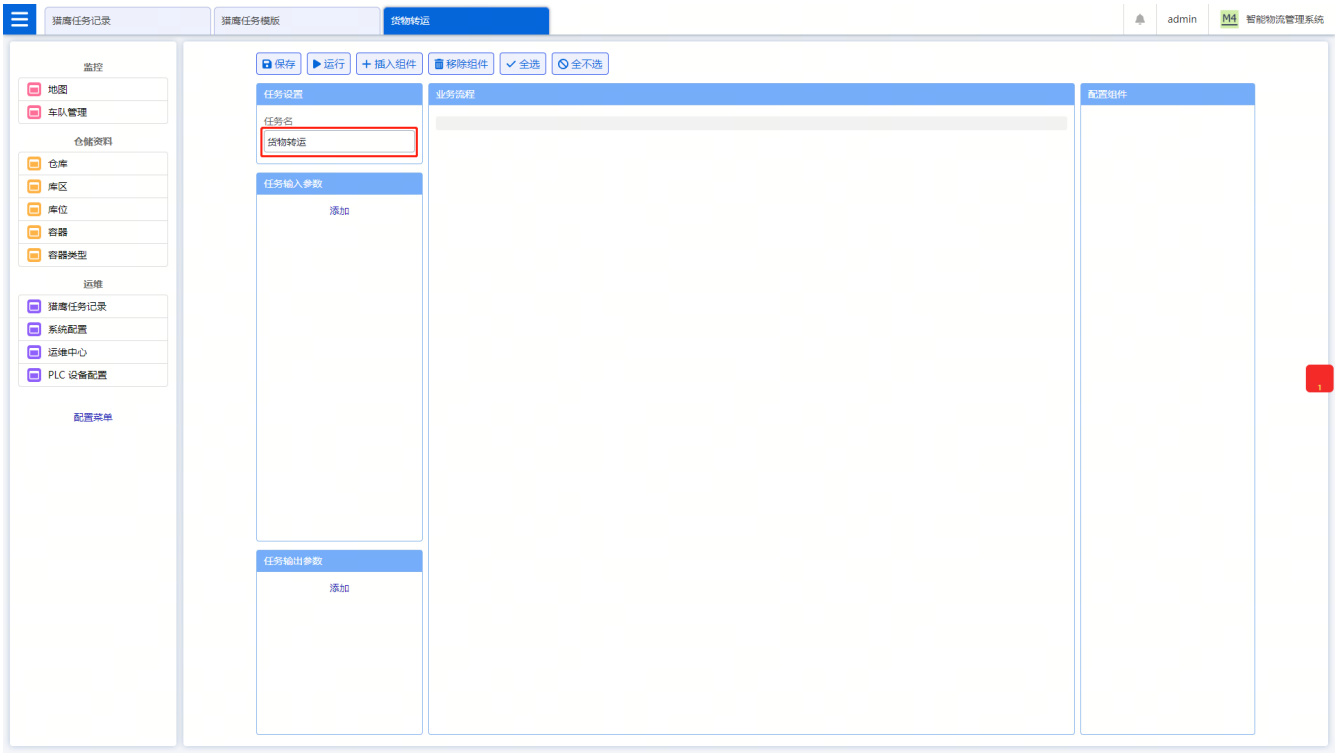
- 点击 “猎鹰任务模板” 界面的 “添加任务模板” 按钮，就会跳转到 “猎鹰任务的编辑” 界面。



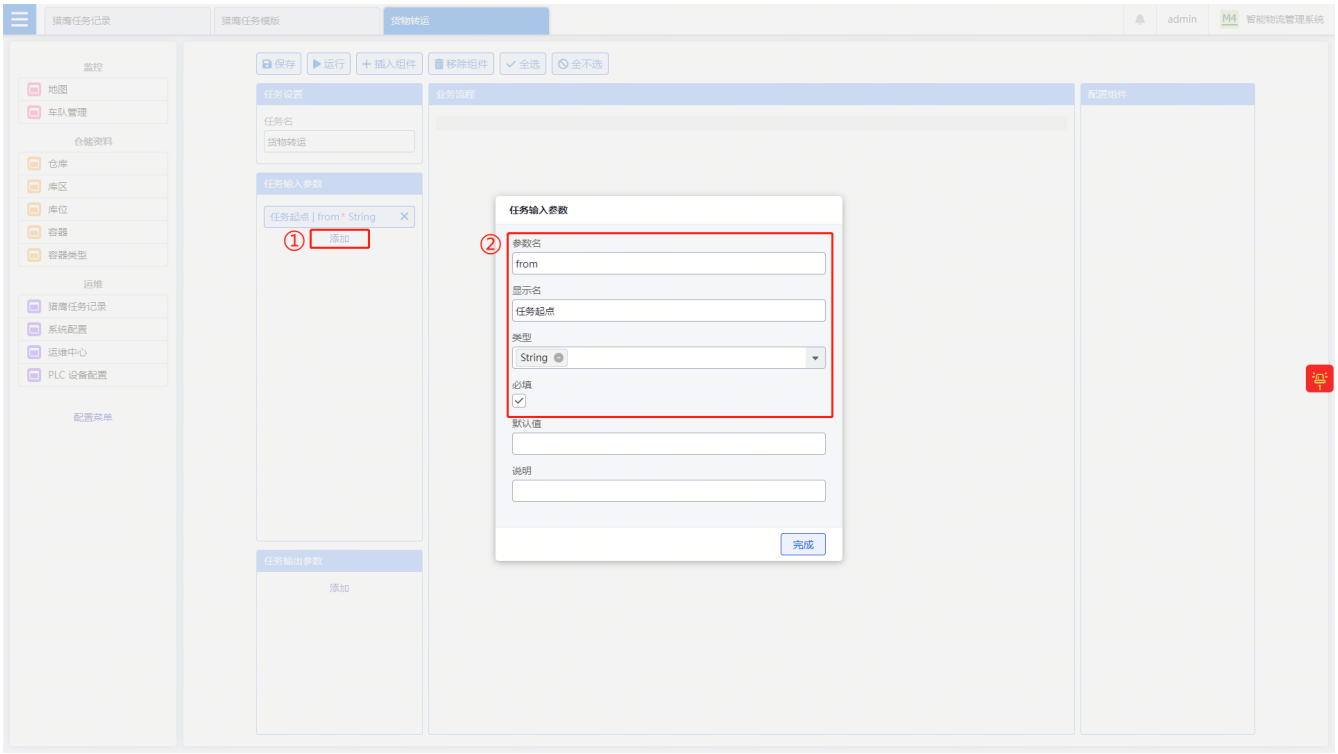
- 空白的 “猎鹰任务编辑” 界面如下图所示。



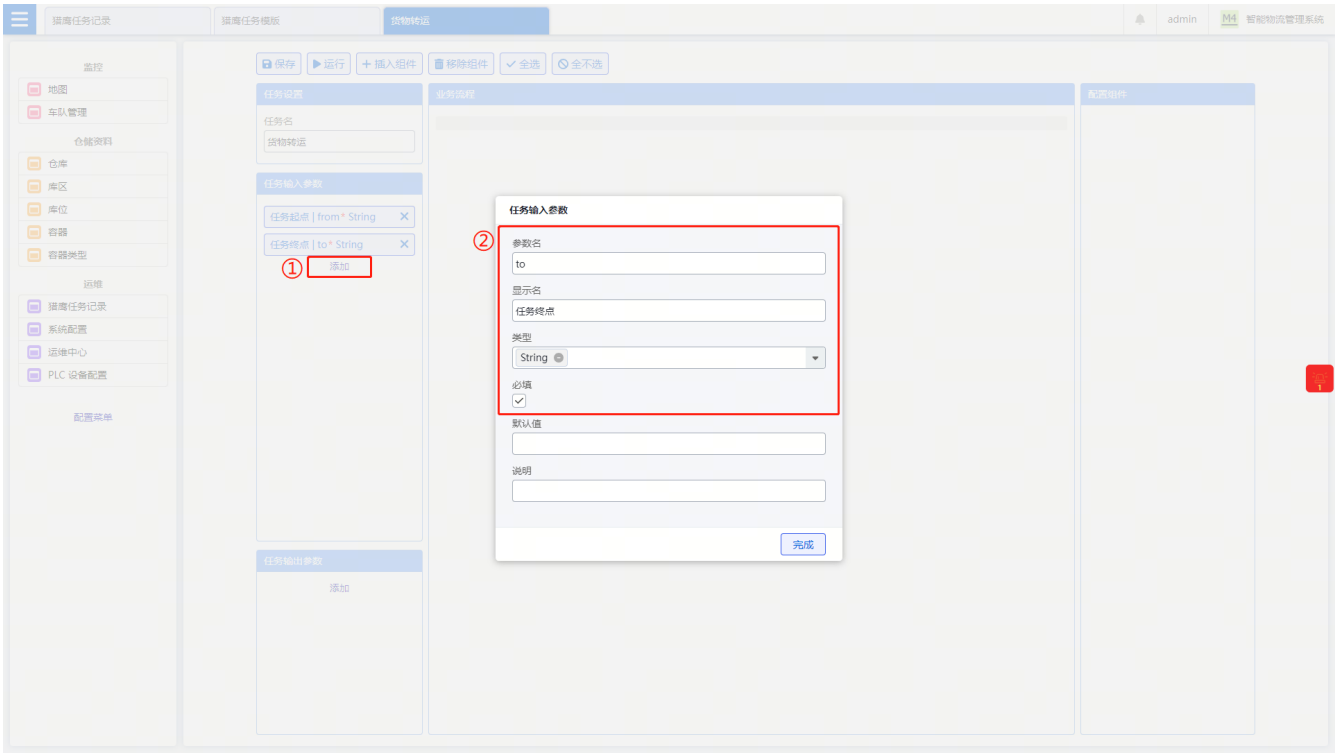
- 设置新建的猎鹰任务的名称为 “货物转运”。



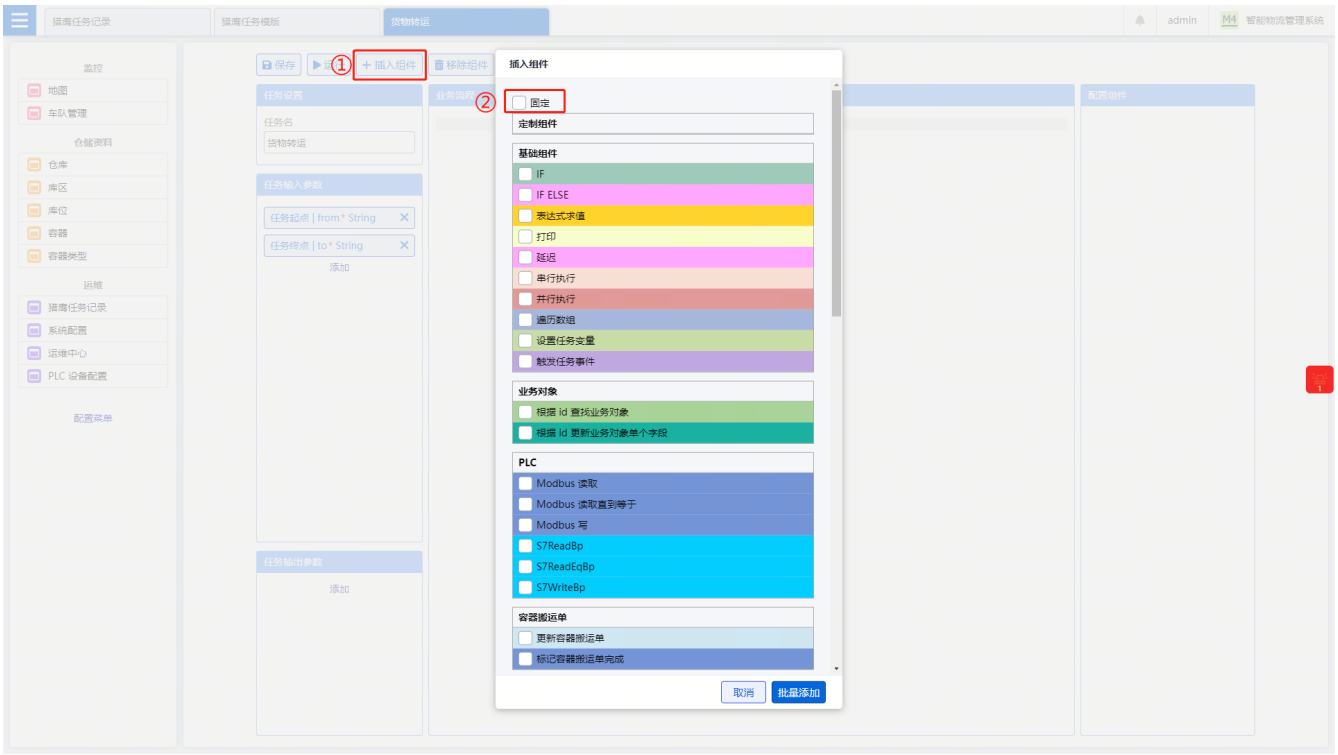
- 添加 “任务起点” 作为 “任务输入参数”。



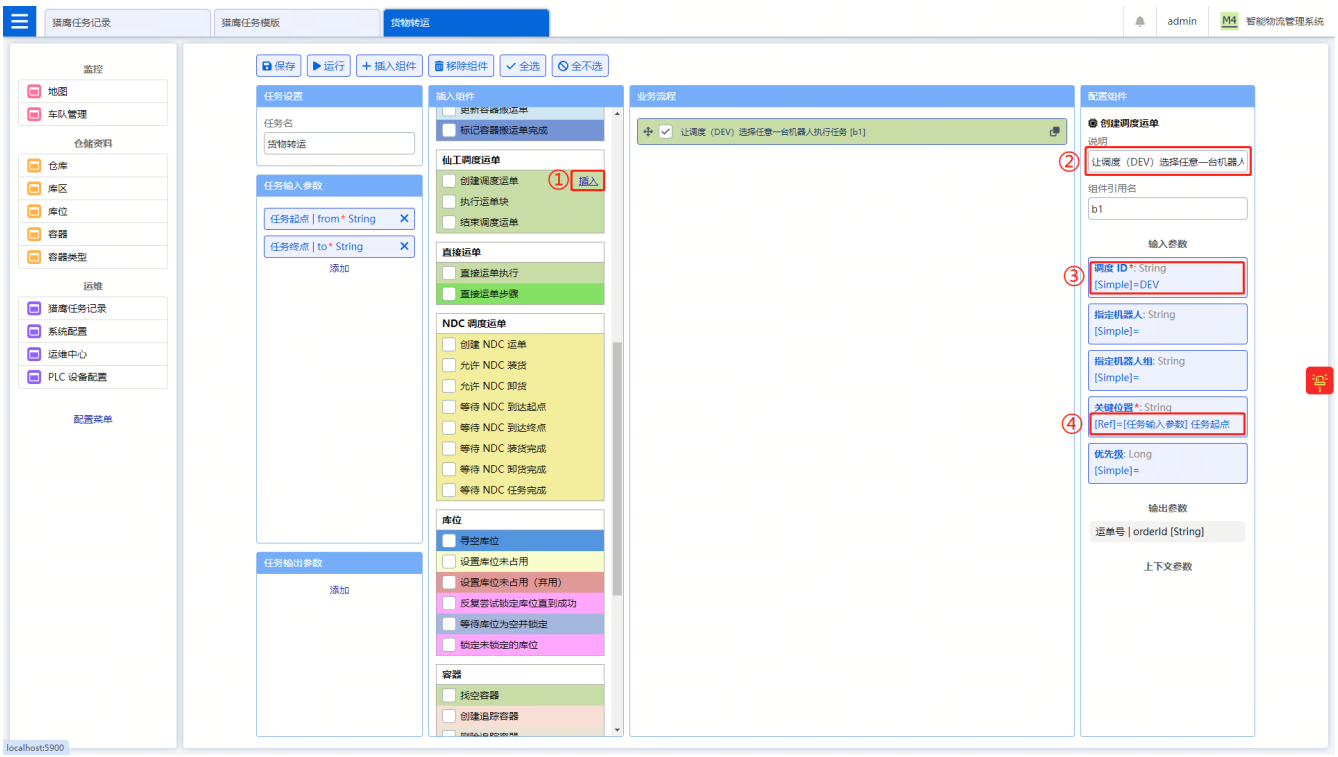
- 继续添加 “任务终点” 作为 “任务输入参数”。



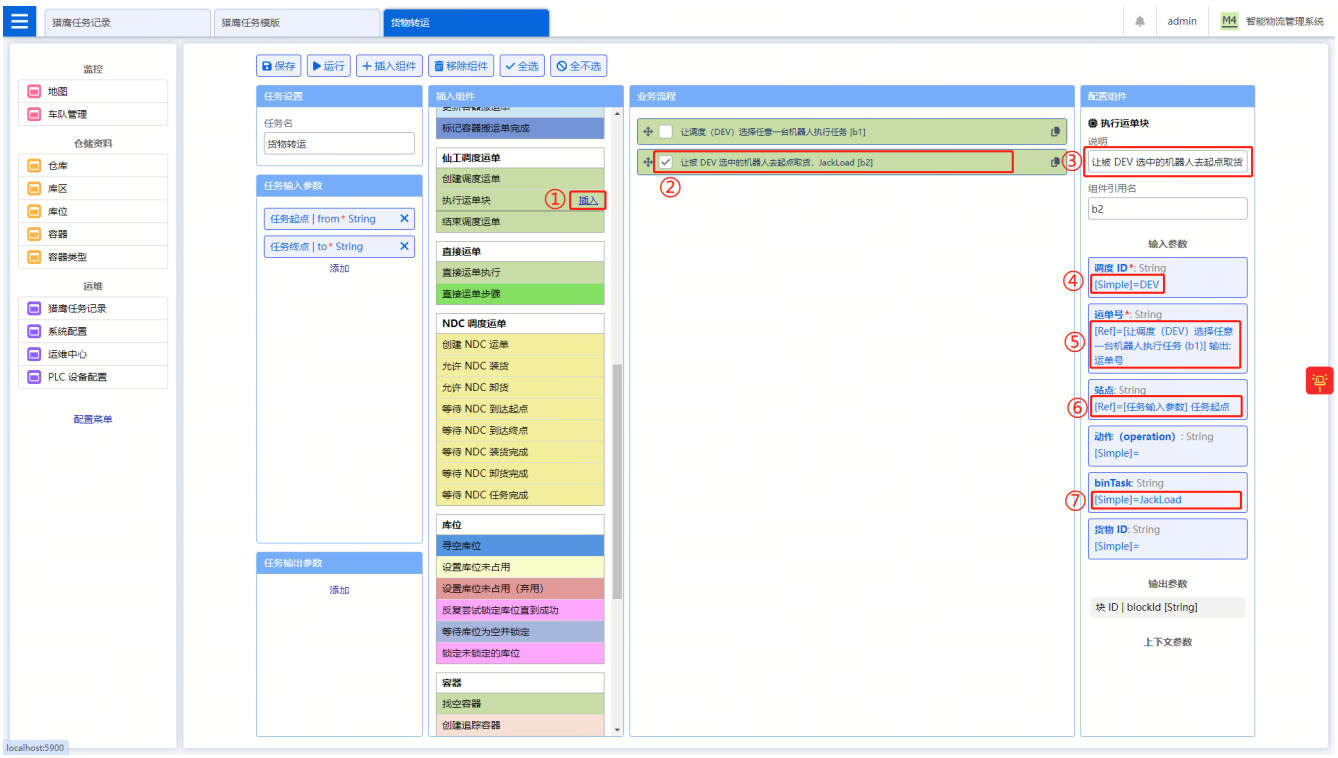
- 为了方便操作，将 “插件列表” 固定在界面上。



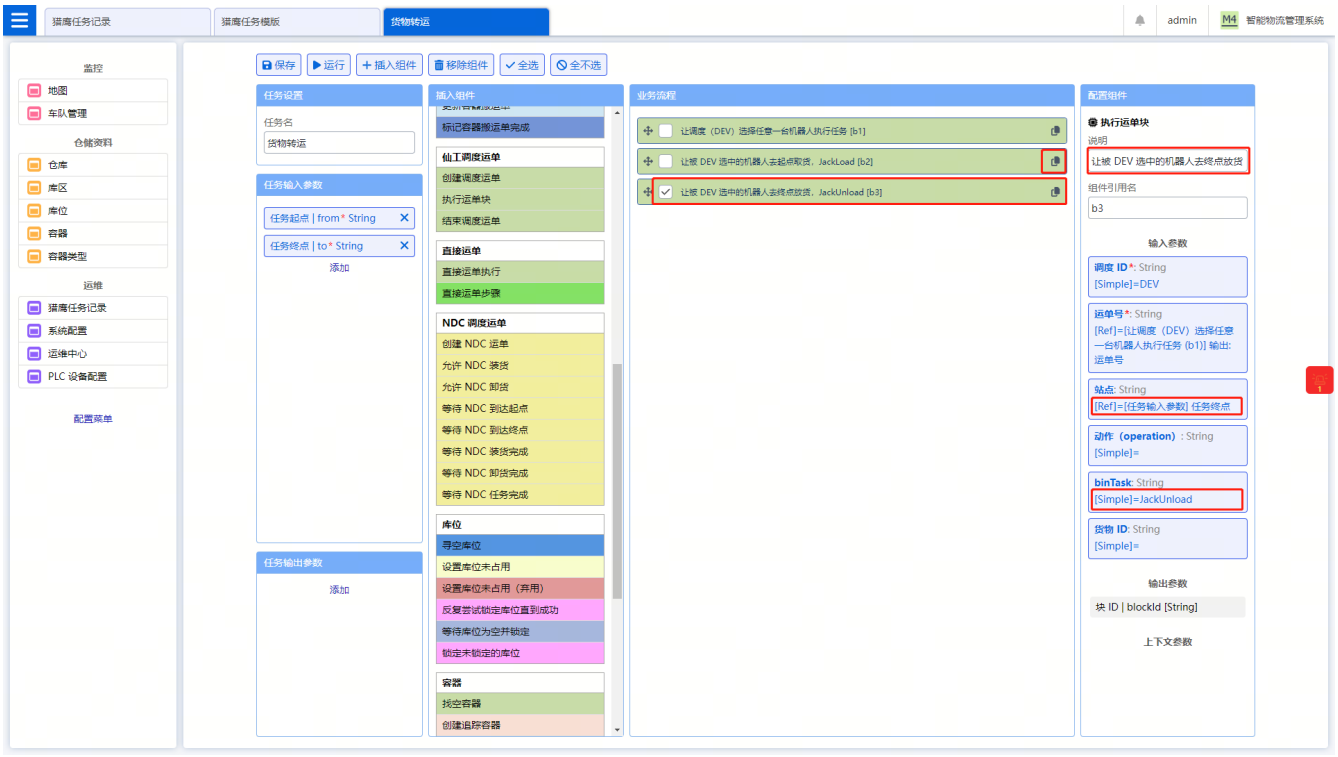
- 在 “业务流程” 中添加 “创建调度运单” 组件。
  - 在组件列表中找到 “创建调度运单” 组件，点击组件右侧的 “插入” 按钮，将其添加到 “业务流程” 中，其 “引用名” 为 b1。不要随意修改组件的 “引用名” ！！！
  - 选中当前组件 b1。
  - 添加必要的 “说明” 为 “让调度（DEV）选择任意一台机器人执行任务”，便于理解业务流程。
  - 设置 “调度 ID” 的值为简单值（Simple） DEV。
  - 引用（Ref） “任务输入参数” 中的 “任务起点”，作为当前组件的 “关键位置” 的值。



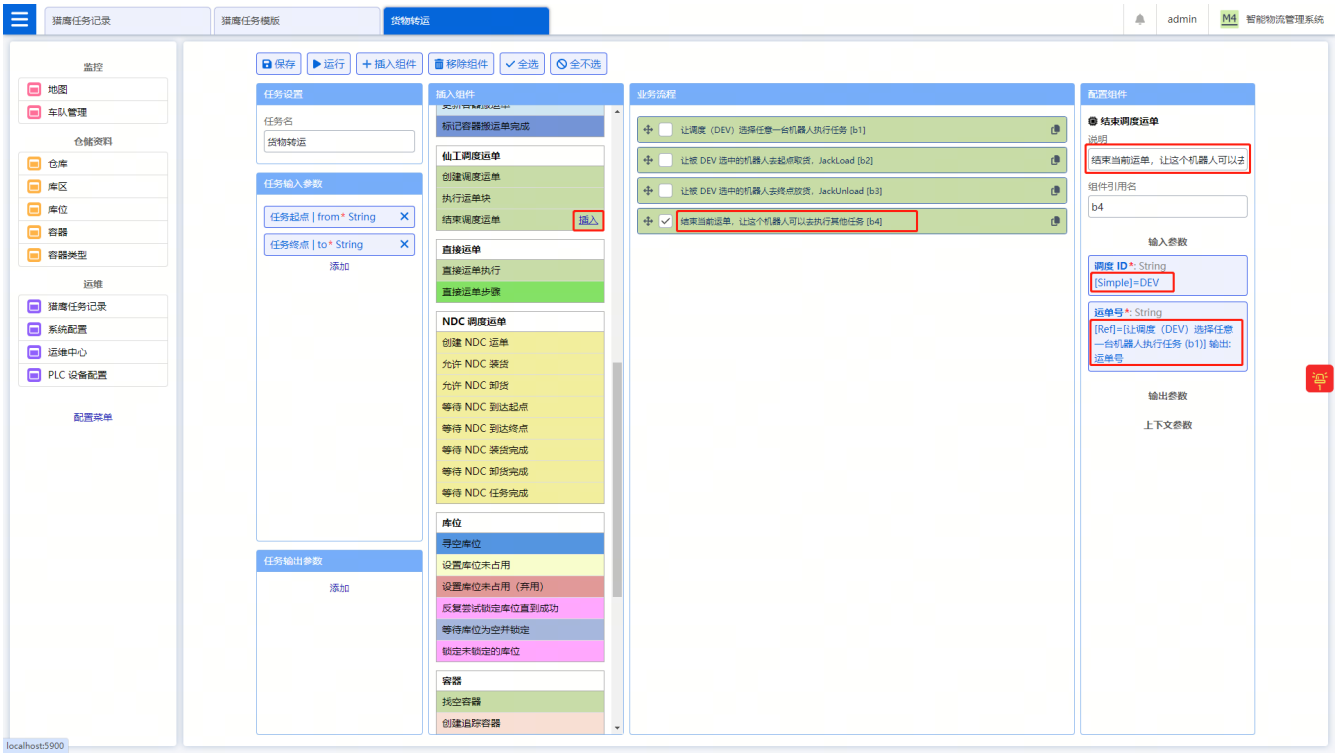
- 在“业务流程”中添加“执行运单块”组件。
  - 在组件列表中找到“执行运单快”组件，点击其右侧的“插入”按钮个，将其添加到“业务流程”中，其“引用名”为 b2。
  - 选中当前组件 b2。
  - 添加必要的“说明”为“让被 DEV 选中的机器人去起点取货，JackLoad”，便于理解业务流程。
  - 设置“调度 ID”的值为简单值（Simple）DEV。
  - 引用（Ref）组件 b1 的“输出参数 —— 运单号”，作为当前组件的“运单号”的值。
  - 引用（Ref）“任务输入参数”中的“任务起点”，作为当前组件的“站点”的值。
  - 设置“binTask”的值为简单值（Simple）JackLoad。



- 在 “业务流程” 中再添加一个 “执行运单块” 组件。
  - 在 “业务流程” 中，点击组件 b2 右侧的 “复制” 按钮，然后会在 “业务流程” 中增加新的组件 b3。
  - 选中当前组件 b3。
  - 添加必要的 “说明” 为 “让被 DEV 选中的机器人去终点放货，JackUnload”，便于理解业务流程。
  - 引用（Ref） “任务输入参数” 中的 “任务终点”，作为当前组件的 “站点” 的值。
  - 设置 “binTask” 的值为简单值（Simple） JackUnload。

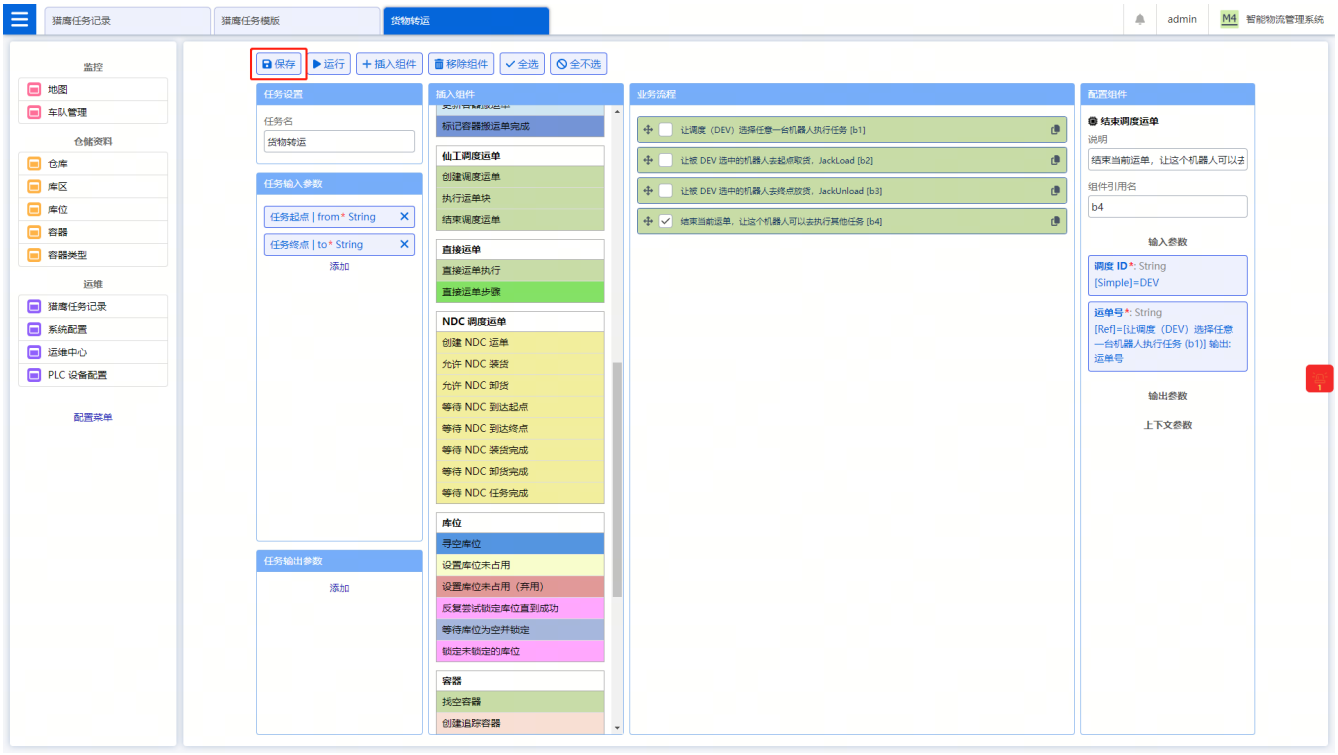


- 在 “业务流程” 中添加 “结束调度运单” 组件。
  - 在组件列表中找到 “结束调度运单” 组件，点击其右侧的 “插入” 按钮，将其添加到 “业务流程” 中，其 “引用名” 为 b4。
  - 选中当前组件 b4。
  - 添加必要的 “说明” 为 “结束当前运单，让这个机器人可以去执行其他任务”，便于理解业务流程。
  - 设置 “调度 ID” 的值为简单值（Simple） DEV。
  - 引用（Ref）组件 b1 的 “输出参数 —— 运单号”，作为当前组件的 “运单号” 的值。

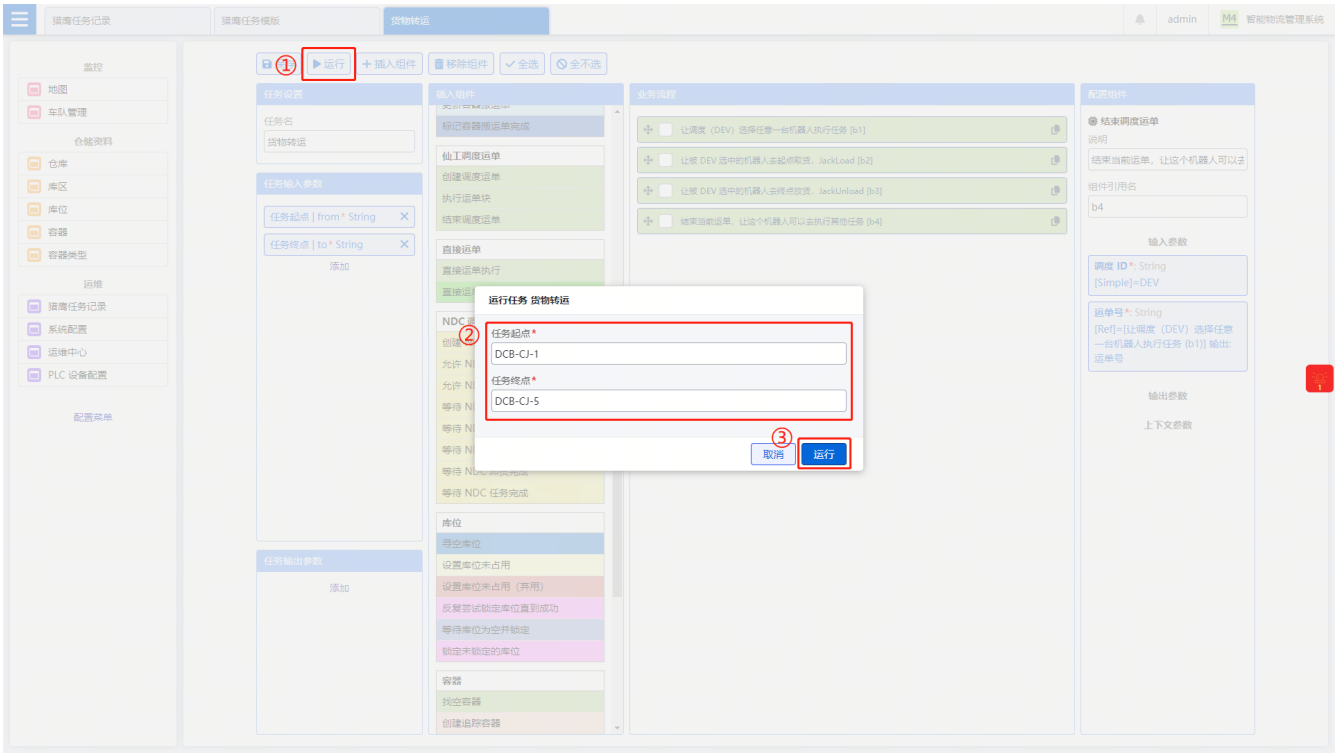


- 至此，猎鹰任务就已经编辑完成了，点击界面左上角的 “保存” 按钮，保存当前任务。

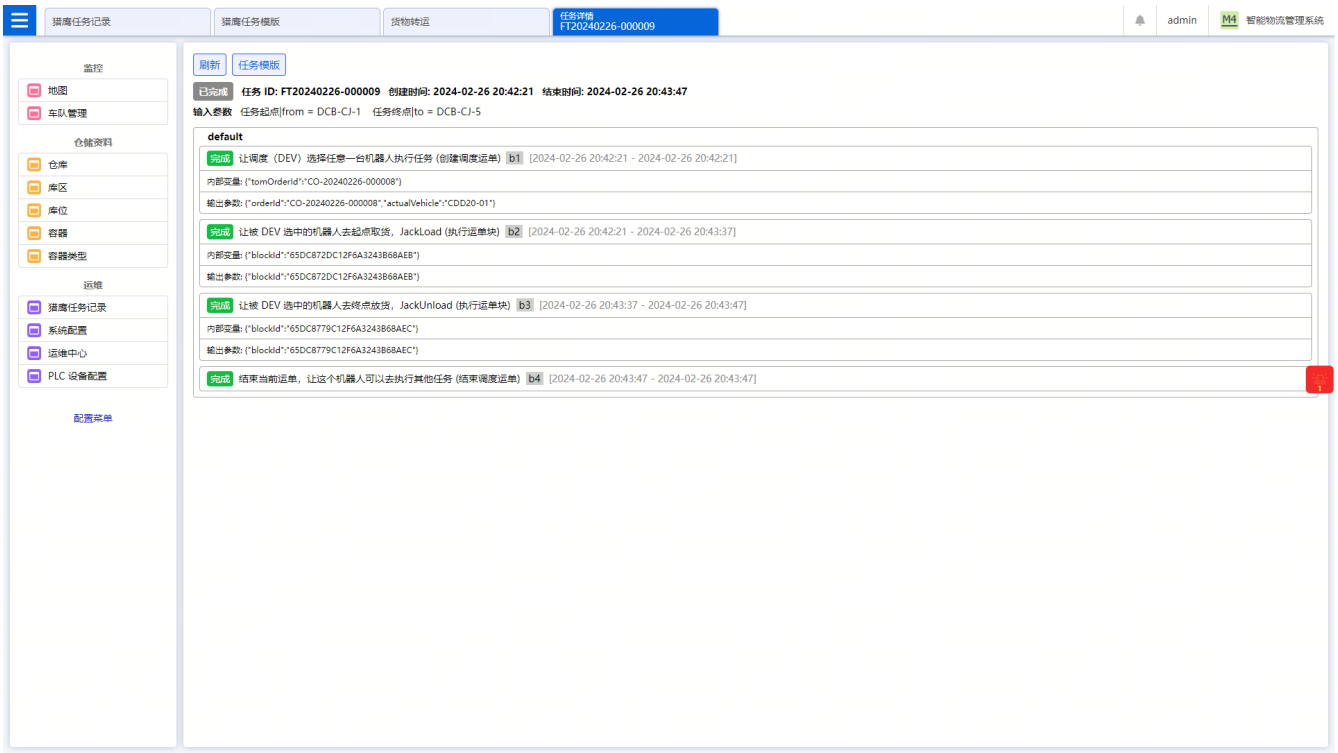
NOTE 注意：保存！保存！一定要保存啊！



- 测试猎鹰任务。
  - 点击界面左上方的 “运行” 按钮。
  - 分别输入 “任务起点” 和 “任务终点”。
  - 点击弹窗右下角的 “运行” 按钮，确认操作。



- 猎鹰任务的执行记录如下图所示：



业务流程验证通过之后，才能进行下一步操作！

第 2 步：用 JavaScript 创建下单接口。

NOTE	需要先在 M4 根目录的 `scripts` 目录下，创建一个 `boot.js` 文件；如果已经存在，则无需创建。

下单接口的参数包括：起点信息、终点信息，例如：

```
{
    "from": "A",
    "to": "B"
}
```

下单接口需要处理的逻辑包括：

- 校验任务 ID 是否重复，如果是，则报错。
- 校验任务起点是否存在，如果不存在，则报错。
- 校验任务终点是否存在，如果不存在，则报错。
- 执行指定名称的猎鹰任务，即名称为 货物转运 的猎鹰任务。

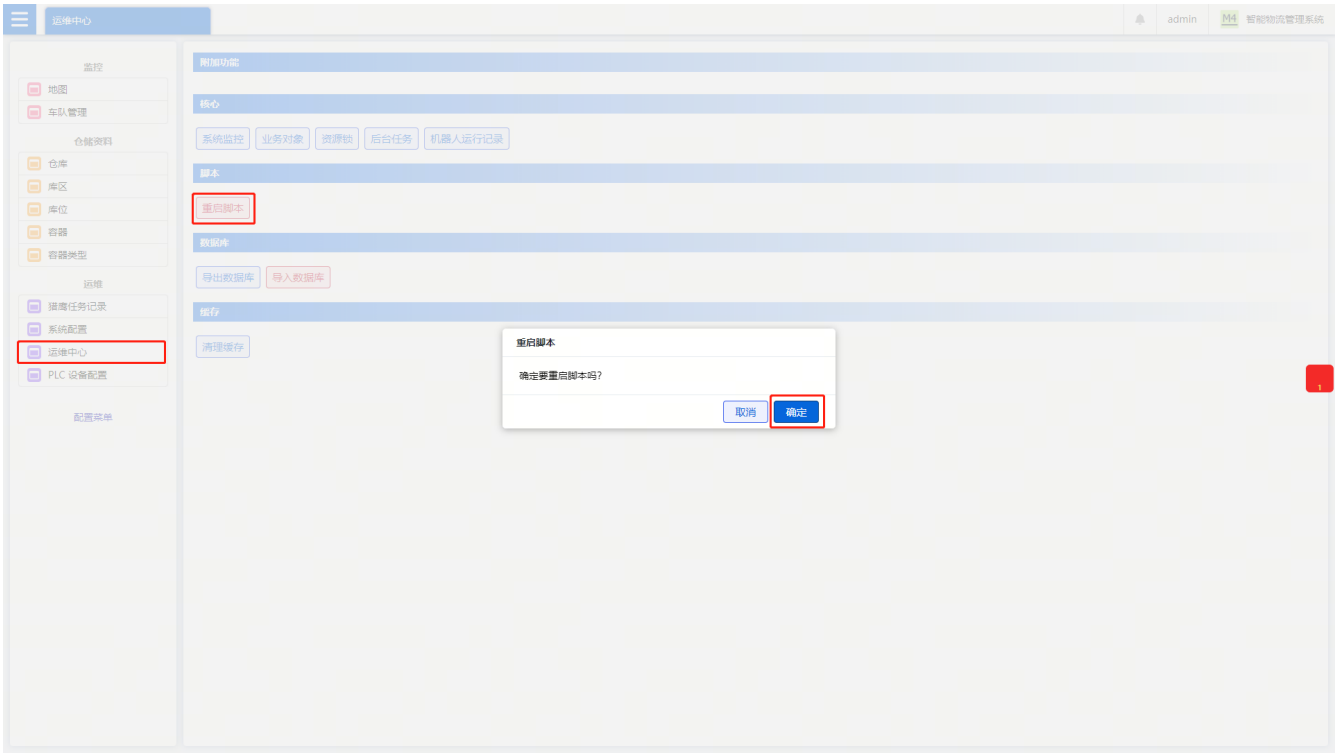
将以下代码复制到 boot.js 文件中。注意：**保存文件！保存文件！保存文件！**

```
const ftDefLabel = "货物转运";

function boot() {
    const tc = base.traceContext();
    base.logInfo(tc, "进入脚本启动函数");

    // 注册下单接口
    httpServer.registerHandler("POST", "api/ext/task/create", "handleCreateTask",
false);
}

// 下单接口对应的接口函数
function handleCreateTask(ctx) {
    try {
        const tc = base.traceContext();
        const reqStr = ctx.getBodyAsString();
        base.logInfo(tc, `收到下单请求：${reqStr}`);

        // req 的数据示例为： { "from": "A", "to": "B" }
        const req = JSON.parse(reqStr);
        const from = req["from"];
        const to = req["to"];

        // 判断 任务起点（from） 的有效性
        if (utils.isNullOrBlank(from))
            throw new Error("任务起点（from）必须是有效的字符串！");
        if (!entity.findOneById(tc, "FbBin", from, null))
            throw new Error(`M4 没有任务起点（from）【${from}】 ！`);

        // 判断 任务终点（to） 的有效性
        if (utils.isNullOrBlank(to))
            throw new Error("任务终点（to）必须是有效的字符串！");
        if (!entity.findOneById(tc, "FbBin", from, null))
            throw new Error(`M4 没有任务终点（to）【${to}】 ！`);
```

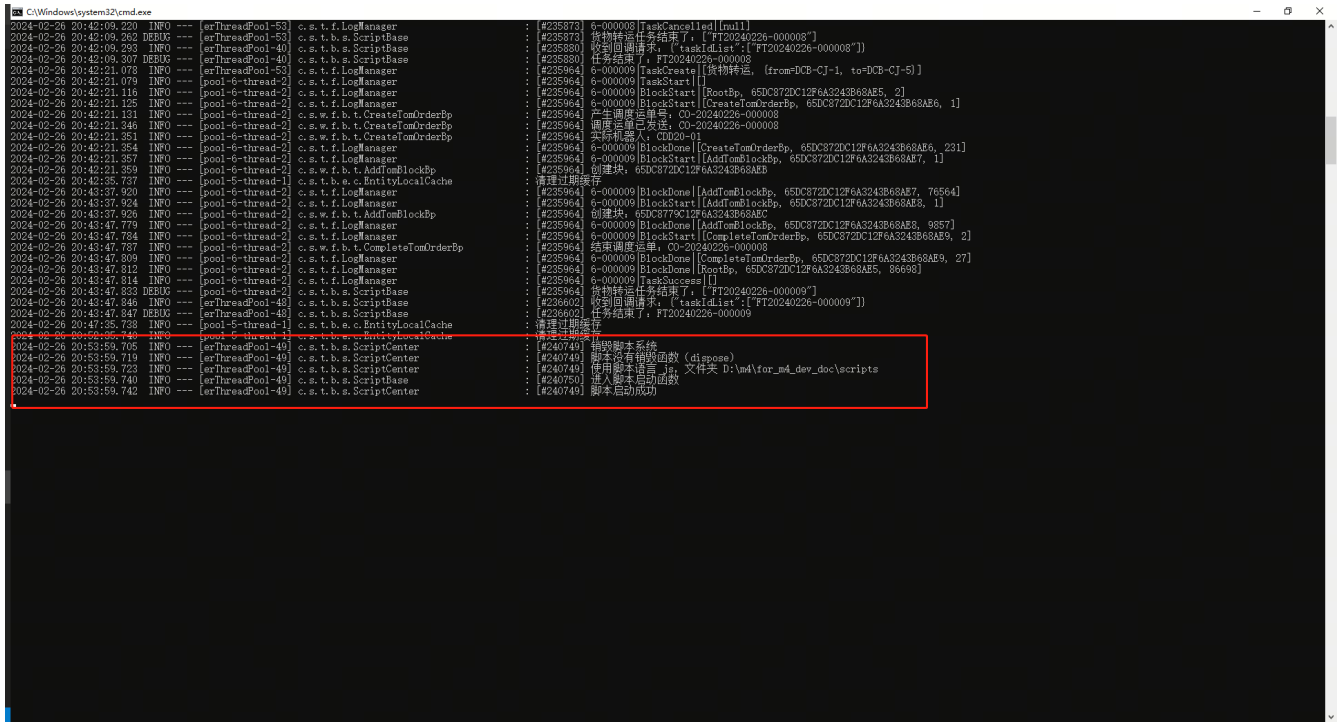
```
        // 执行猎鹰任务
        const ftId = falcon.runTaskByLabelAsync(tc, ftDefLabel, { from, to });
        base.logDebug(tc, `下单成功；请求参数为 ${reqStr}；任务 ID 为 ${ftId}`);

        // 成功之后，将 M4 生成的任务 ID 回传给调用方。
        ctx.setJson({ "code": 200, "message": ftId });
    } catch (err) {
        ctx.setStatus(400);
        ctx.setJson({ "code": 400, "message": `下单失败：${err}` });
    }
}
```

修改并保存 boot.js 文件之后，需要重启脚本。

- 点击左侧菜单栏中的 “运维中心”。
- 在弹出的 “运维中心” 界面中，点击 “重启脚本” 按钮。
- 在弹出的对话框中，点击 “确定” 按钮，然后脚本就会重启了。



重启脚本之后，M4 的日志中会记录 “脚本启动成功”。



接下来就可以通过 HTTP 调试工具，调试此接口了。

第 3 步：用 JavaScript 注册回调接口。

参考上一步注册下单接口的操作，将以下代码复制到 boot.js 文件中，直接覆盖原来的内容即可。

```
const ftDefLabel = "货物转运";
const mockCallBackPath = "api/ext/mock/taskFinished";

function boot() {
    const tc = base.traceContext();
    base.logInfo(tc, "进入脚本启动函数");

    // 注册下单接口
    httpServer.registerHandler("POST", "api/ext/task/create", "handleCreateTask",
false);

    // 注册回调接口
    httpServer.registerHandler("POST", mockCallBackPath, "handleMockTaskFinished",
false);
}

// 下单接口对应的接口函数
function handleCreateTask(ctx) {
    try {
        const tc = base.traceContext();
        const reqStr = ctx.getBodyAsString();
        base.logInfo(tc, `收到下单请求：${reqStr}`);

        // req 的数据示例为： { "from": "A", "to": "B" }
        const req = JSON.parse(reqStr);
        const from = req["from"];
```

```
        const to = req["to"];

        // 判断 任务起点（from） 的有效性
        if (utils.isNullOrBlank(from))
            throw new Error("任务起点（from）必须是有效的字符串！");
        if (!entity.findOneById(tc, "FbBin", from, null))
            throw new Error(`M4 没有任务起点（from）【${from}】 ！`);

        // 判断 任务终点（to） 的有效性
        if (utils.isNullOrBlank(to))
            throw new Error("任务终点（to）必须是有效的字符串！");
        if (!entity.findOneById(tc, "FbBin", from, null))
            throw new Error(`M4 没有任务终点（to）【${to}】 ！`);

        // 执行猎鹰任务
        const ftId = falcon.runTaskByLabelAsync(tc, ftDefLabel, { from, to });
        base.logDebug(tc, `下单成功；请求参数为 ${reqStr}；任务 ID 为 ${ftId}`);

        // 成功之后，将 M4 生成的任务 ID 回传给调用方。
        ctx.setJson({ "code": 200, "message": ftId });
    } catch (err) {
        ctx.setStatus(400);
        ctx.setJson({ "code": 400, "message": `下单失败：${err}` });
    }
}

// 下单接口对应的接口函数
function handleMockTaskFinished(ctx) {
    const tc = base.traceContext();
    const reqStr = ctx.getBodyAsString();
    base.logInfo(tc, `收到回调请求：${reqStr}`);

    // req 的数据示例为： { "taskId": "xxxxx" }
    const req = JSON.parse(reqStr);
    base.logDebug(tc, `任务结束了：${req["taskId"]}`);
    ctx.setJson({ "code": 200, "message": "OK" });
}
```

修改并保存 boot.js 文件之后，需要重启脚本！

第 4 步：通过 “实体生命周期拦截器” 监听任务状态。

在上一步的基础上，我们在 JavaScript 的脚本中，继续增加 “实体生命周期拦截器”，以监听猎鹰任务的变化，并且在任务结束时，回调接口，告知上位机有任务结束了；关键代码如下所示：

```
// 通过 拦截器，监听实体的状态变化，并在任务结束时，回调接口，告知上位机任务结束了。
function trytoCallbackWhenTaskFinished(tc, em, ids) {
    // 如果发生改变的实体不是 猎鹰任务（FalconTaskRecord），则不处理。
    if (em.getName() != ftEntityName) return;

    // 仅筛选已经处于终态的猎鹰任务
```

```
    const ftrList = entity.findMany(tc, ftEntityName, cq.and([cq.include("id", ids),
cq.eq("defLabel", ftDefLabel), cq.gt("status", 140)]), null);

    // 如果没有结束的任务，则直接返回。
    if (ftrList.length === 0) return;

    // 告知上位机已经结束的任务的 ID；
    const ftrIdList = ftrList.map(it => it["id"]);
    base.logDebug(tc, `${ftDefLabel}任务结束了：${base.jsonToString(ftrIdList)}`);
    const resp = httpClient.requestJson("post", `http://127.0.0.1:5900/
${mockCallBackPath}`, { "taskIdList": ftrIdList }, {});
    if (resp["code"] !== 200) {
        base.logDebug(tc, `任务结束后，回调接口失败：${resp["bodyString"]}，稍后请
人工处理！`);
    }
}
```

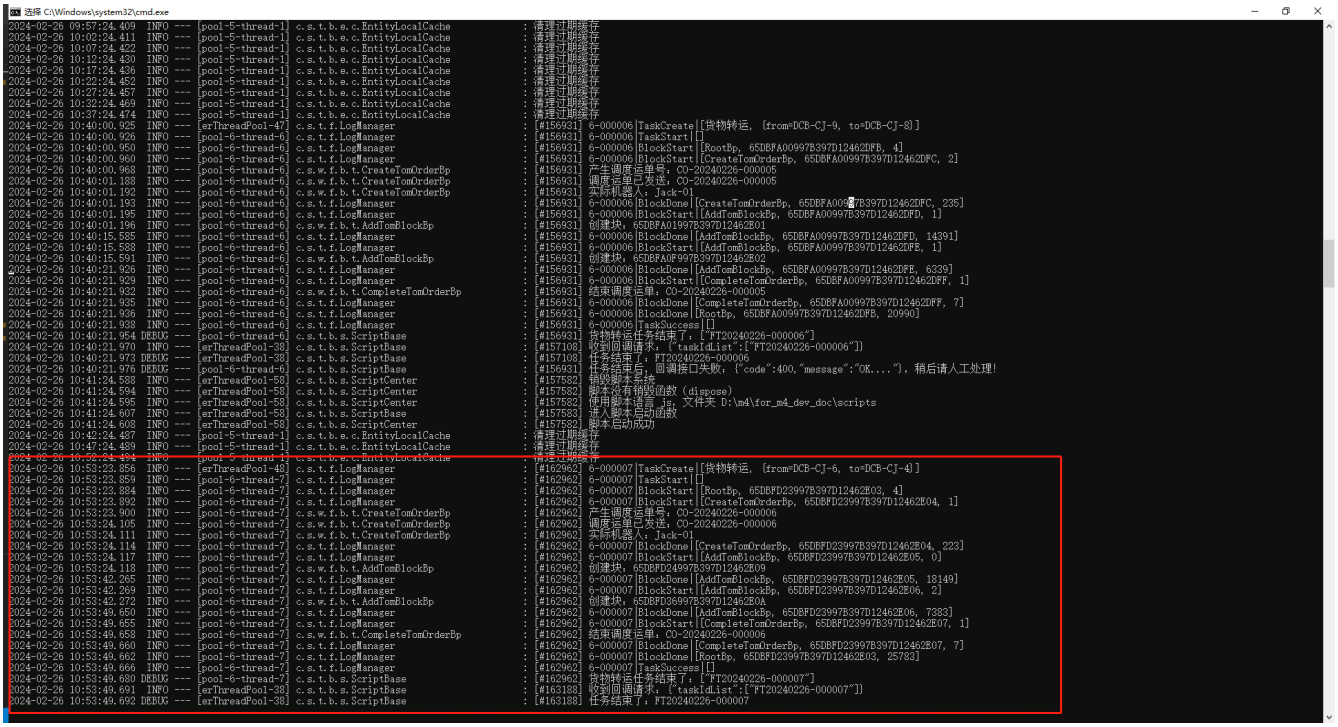
至此，完整的 JavaScript 代码如下所示：

```
const ftDefLabel = "货物转运";
const mockCallBackPath = "api/ext/mock/taskFinished";
const ftEntityName = "FalconTaskRecord";

function boot() {
    const tc = base.traceContext();
    base.logInfo(tc, "进入脚本启动函数");

    // 注册下单接口
    httpServer.registerHandler("POST", "api/ext/task/create", "handleCreateTask",
false);

    // 注册回调接口
    httpServer.registerHandler("POST", mockCallBackPath, "handleMockTaskFinished",
false);

    // 通过拦截器
    entityExt.extAfterUpdating("FalconTaskRecord", "trytoCallbackWhenTaskFinished");
}

// 下单接口对应的接口函数
function handleCreateTask(ctx) {
    try {
        const tc = base.traceContext();
        const reqStr = ctx.getBodyAsString();
        base.logInfo(tc, `收到下单请求：${reqStr}`);

        // req 的数据示例为： { "from": "A", "to": "B" }
        const req = JSON.parse(reqStr);
        const from = req["from"];
        const to = req["to"];
```

```
        // 判断 任务起点（from） 的有效性
        if (utils.isNullOrBlank(from))
            throw new Error("任务起点（from）必须是有效的字符串！");
        if (!entity.findOneById(tc, "FbBin", from, null))
            throw new Error(`M4 没有任务起点（from）【${from}】 ！`);

        // 判断 任务终点（to） 的有效性
        if (utils.isNullOrBlank(to))
            throw new Error("任务终点（to）必须是有效的字符串！");
        if (!entity.findOneById(tc, "FbBin", from, null))
            throw new Error(`M4 没有任务终点（to）【${to}】 ！`);

        // 执行猎鹰任务
        const ftId = falcon.runTaskByLabelAsync(tc, ftDefLabel, { from, to });
        base.logDebug(tc, `下单成功；请求参数为 ${reqStr}；任务 ID 为 ${ftId}`);

        // 成功之后，将 M4 生成的任务 ID 回传给调用方。
        ctx.setJson({ "code": 200, "message": ftId });
    } catch (err) {
        ctx.setStatus(400);
        ctx.setJson({ "code": 400, "message": `下单失败：${err}` });
    }
}

// 下单接口对应的接口函数
function handleMockTaskFinished(ctx) {
    const tc = base.traceContext();
    const reqStr = ctx.getBodyAsString();
    base.logInfo(tc, `收到回调请求：${reqStr}`);

    // req 的数据示例为： { "taskIdList": ["xxxxx", "yyyyy"] }
    const req = JSON.parse(reqStr);
    base.logDebug(tc, `任务结束了：${req["taskIdList"]}`);
    ctx.setJson({ "code": 200, "message": "OK" });
}

// 通过 拦截器，监听实体的状态变化，并在任务结束时，回调接口，告知上位机任务结束了。
function trytoCallbackWhenTaskFinished(tc, em, ids) {
    // 如果发生改变的实体不是 猎鹰任务（FalconTaskRecord），则不处理。
    if (em.getName() != ftEntityName) return;

    // 仅筛选已经处于终态的猎鹰任务
    const ftrList = entity.findMany(tc, ftEntityName, cq.and([cq.include("id", ids),
cq.eq("defLabel", ftDefLabel), cq.gt("status", 140)]), null);

    // 如果没有结束的任务，则直接返回。
    if (ftrList.length === 0) return;

    // 告知上位机已经结束的任务的 ID；
    const ftrIdList = ftrList.map(it => it["id"]);
```

```
    base.logDebug(tc, `${ftDefLabel}任务结束了：${base.jsonToString(ftrIdList)}`);
    const resp = httpClient.requestJson("post", `http://127.0.0.1:5900/
${mockCallBackPath}`, { "taskIdList": ftrIdList }, {});
    if (resp["code"] !== 200) {
        base.logDebug(tc, `任务结束后，回调接口失败：${resp["bodyString"]}，稍后请
人工处理！`);
    }
}
```

至此，此示例的开发工作就结束了，可以尝试调用下单接口下单；后台的执行记录，如下图所示：



2.1.4 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：

js

```
base.traceContext()
```

py

```
base.traceContext()
```

2.1.5 日志

支持 DEBUG / INFO / ERROR 三个级别的日志，分别有三个函数。函数的第一个参数都是 TraceContext 对象，第二参数是一个字符串，即日志的内容。对于错误，传入第三个参数。

js

```
base.logDebug(tc, msg)
base.logInfo(tc, msg)
base.logError(tc, msg, err)
```

py

```
base.logDebug(tc, msg)
base.logInfo(tc, msg)
base.logError(tc, msg, err)
```

例子:

```
base.logInfo(base.traceContext(), "系统已启动")
```

2.1.6 声明变量

```
let value = "你好"; // 声明一个变量 value, 并且设置其初始值为“你好”。
```

2.1.7 声明函数

```
function foo() {
    // do something...
};
```

例子:

```
function add(a, b) {
    return a + b;
};
```

2.1.8 创建数组（List）

```
const myList = [];
```

例子:

```
const myList = [1, 2, 4];
```

2.1.9 创建字典（Map）

```
const myMap = {};
```

例子：

```
const myMap = { "key1": "value1" };
```

2.1.10 将对象转换为 JSON 字符串

```
const jsonObj = { "key1": "value1", "key2": 3, "key3": false };
const jsonStr = base.jsonToString(jsonObj);
```

2.1.11 将 JSON 字符串解析为对象

```
const jsonStr = "{\"key1\":\"value1\",\"key2\":3,\"key3\":false}";
const jsonObj = JSON.parse(jsonStr)
```

2.2 常用

2.2.1 处理字符串

- 判断字符串为 null、undefined，或者空字符串（即 ""）、或者全是空格。

```
utils.isNullOrBlank(str);
```

例子：

```
utils.isNullOrBlank(null); // 返回值为 true ;

utils.isNullOrBlank(undefined); // 返回值为 true ;

utils.isNullOrBlank(""); // 返回值为 true ;

utils.isNullOrBlank("  "); // 字符串全是空格，返回值为 true ;

utils.isNullOrBlank("你好"); // 返回值为 false ;
```

- 在字符串 str 中找到第一次出现分隔符 sep 的位置，并从 str 中获取出现在分隔符 sep 之后的内容。

```
utils.substringAfter(str, sep);
```

例子：

```
utils.substringAfter(null, *);      // 结果为 null

utils.substringAfter("", *) ;       // 结果为  ""

utils.substringAfter(*, null) ;     // 结果为  ""

utils.substringAfter("abc", "a") ;  // 结果为  "bc"

utils.substringAfter("abcba", "b") ; // 结果为  "cba"

utils.substringAfter("abc", "c") ;   // 结果为  ""

utils.substringAfter("abc", "d") ;   // 结果为  ""

utils.substringAfter("abc", "") ;    // 结果为  "abc"
```

- 在字符串 str 中找到第一次出现分隔符 sep 的位置，并从 str 中获取出现在分隔符 sep 之前的内容。

```
utils.substringBefore(str, sep);
```

例子：

```
utils.substringBefore(null, *) ;      // 结果为  null

utils.substringBefore("", *) ;        // 结果为  ""

utils.substringBefore("abc", "a") ;   // 结果为  ""

utils.substringBefore("abcba", "b") ; // 结果为  "a"

utils.substringBefore("abc", "c") ;   // 结果为  "ab"

utils.substringBefore("abc", "d") ;   // 结果为  "abc"

utils.substringBefore("abc", "") ;    // 结果为  ""

utils.substringBefore("abc", null) ;  // 结果为  "abc"
```

2.2.2 处理数字

如果是字符串，字符必须全部为十进制数字，但第一个字符可以是 - 表示负值，或者 + 表示正值。

- 将 v 转换为 Int 类型的数值。

```
utils.anyToInt(v);
```

例子：

```
utils.anyToInt(1); // 结果为 1

utils.anyToInt("100"); // 结果为 100

utils.anyToInt("+100"); // 结果为 100

utils.anyToInt("-100"); // 结果为 -100
```

- 将 v 转换为 Long 类型的数值。

```
utils.anyToLong(v);
```

例子：

```
utils.anyToLong(1); // 结果为 1

utils.anyToLong("100"); // 结果为 100
```

- 将 v 转换为 Float 类型的数值。

```
utils.anyToFloat(v);
```

例子：

```
utils.anyToFloat(1); // 结果为 1.0

utils.anyToFloat("100"); // 结果为 100.0
```

- 将 v 转换为 Double 类型的数值。

```
utils.anyToDouble(v);
```

例子：

```
utils.anyToDouble(1); // 结果为 1.0

utils.anyToDouble("100"); // 结果为 100.0
```

2.2.3 处理时间

- 将 v 转换为 Date 类型的数据。

```
utils.anyToDate(v);
```

例子：

```
utils.anyToDate("2024-03-20");
```

- 以指定的字符串格式，显示时间 d。

```
utils.formatDate(d, format);
```

例子：

```
const d = new Date();
utils.formatDate(d, "yyyy-MM-dd_HH_mm_ss"); // 输出：2024-06-19_13_11_28
```

2.2.4 全局变量

全局变量的作用域是整个程序，整个系统中都可以访问和使用，通常用于数据共享。

> **NOTE** 注意数据未持久化，只存在缓存中，重启系统会清空

- 设置/清空全局缓存

key 是非空字符串，value 是任意类型

```
base.setGlobalValue(key, value)
```

例子：

```
base.setGlobalValue("a", 1)      // 设置缓存变量 a 值是 1

base.setGlobalValue("a", null)   // 移除缓存变量 a
```

- 获取全局缓存

key 是非空字符串

```
base.getGlobalValue(key)
```

例子：

```
base.getGlobalValue("a")    // 获取缓存变量 a 值，如果变量 a 不存在返回 null
```

2.3 业务对象增删改查

2.3.1 构造查询

业务对象的增、删、改、查通常需要根据条件筛选数据。在查询操作中，可能还需要排序、分页和指定字段。因此，首先介绍条件筛选和结果集控制。

条件筛选

- 获取查询所有条件

```
cq.all()      // 得到查询所有的条件
```

- 获取某个字段等于给定值的查询条件

```
cq.eq(field, v)  // field: 字段 v: 给定值
```

- 获取某个字段不等于给定值的查询条件

```
cq.ne(field, v) // field: 字段 v: 给定值
```

- 获取字段大于给定值的查询条件

```
cq.gt(field, v) // field: 字段 v: 给定值
```

- 获取字段值大于等于给定值的查询条件

```
cq.gte(field, v) // field: 字段 v: 给定值
```

- 获取字段值小于给定值的查询条件

```
cq.lt(field, v) // field: 字段 v: 给定值
```

- 获取字段值小于等于给定值的查询条件

```
cq.lte(field, v) // field: 字段 v: 给定值
```

- 获取根据 id 的查询条件

```
cq.idEq(id) // id: id的值
```

- 获取字段值为空的查询条件

```
cq.empty(field) // field: 字段
```

- 获取字段值不为空的查询条件

```
cq.notEmpty(field) // field: 字段
```

- 获取字段值等于多个指定值的查询条件

```
cq.include(field, []) // field: 字段 []: 数组
```

- 获取满足多个字段条件组合的查询条件

```
cq.and([])      // []: 查询条件的数组
```

例子:

```
cq.and([cq.eq(field1, v1), cq.gt(field2, v2)]) // 表示查询 field1 字段值等于
v1,并且 field2 字段值等于 v2
```

- 获取满足任意一个字段条件的查询条件

```
cq.or([])      // []: 查询条件的数组
```

例子

```
cq.or([cq.eq(field1, v1), cq.gt(field2, v2)]) // 表示查询 field1 字段值等于 v1,
或者 field2 字段值等于 v2
```

结果集控制

获取结果集控制

```
// projection 指定查询返回字段，可不指定；指定查询按 age 字段降序排列，可不指定
// skip 指定查询跳过前 10 条记录，可不指定；指定查询只返回 5 条记录，可不指定
entity.buildFindOptions(projection, sort, skip, limit)
```

例子:

```
let option = entity.buildFindOptions(["httpMethod"], ["id DESC"],  10, 5)
```

NOTE 结果集控制是可选的。

参数说明

下面 tc 指 TraceContext 对象, entityName 指业务对象的名称，conditions 指筛选条件，findOptions 指结果集控制条件

2.3.2 读取

- 根据条件查询并返回单条业务对象实体

```
entity.findOne(tc, entityName, conditions, findOptions)
```

例子:

```
 // 需求：查询库位 id 是 LOC-01 的库位
let traceContext = base.traceContext()                                  // 创建上下文对象
let conditons = cq.eq("id", "LOC-01")                                    // 查询条件
let result = entity.findOne(traceContext, "FbBin", conditons, null)  // 调用
查询单条数据方法，如果不存在返回 null，存在返回结果
```

```
 // 需求：查询库位 LOC-01 有货
let traceContext = base.traceContext()
let conditons = cq.and([cq.eq("id", "LOC-01"), cq.eq("occupied", true)])
let result = entity.findOne(traceContext, "FbBin",conditons, null)
```

- 根据 id 查询业务对象实体

```
entity.findOneById(tc, entityName, conditions, findOptions)
```

例子:

```
// 需求：根据库位 id 查询库位
let traceContext = base.traceContext()
let result = entity.findOneById(traceContext, "FbBin", "LOC-01", null) //
如果不存在返回 null，存在返回结果
```

- 根据条件查询所有符合条件的业务对象实体

```
entity.findMany(tc, entityName, conditions, findOptions)
```

例子:

```
// 需求：查询所有未占用的库位，并且只返回库位 id
let traceContext = base.traceContext()
let conditons = cq.eq("occupied", false)                                 // 构建查询条件
let findOptions = entity.buildFindOptions(["id"], null,  null, null) // 构建
```

```
结果集控制条件
let result = entity.findMany(traceContext, "FbBin", conditons, findOptions) // 返回
结果类型是数组
```

- 校验业务对象实体是否存在

```
entity.exists(tc, entityName, conditons)
```

例子：

```
// 需求：查询库位 LOC-01 是否存在
let traceContext = base.traceContext()
let conditions = cq.idEq("LOC-01")
let num = entity.exists(traceContext, "FbBin", conditions) // true: 符合查询条件
false：不符合查询条件
```

- 根据条件查询业务对象实体数量

```
entity.count(tc, entityName, conditions)
```

例子：

```
let traceContext = base.traceContext()
let conditons = cq.all()                          // 构建查询条件
let num = entity.count(traceContext, "FbBin", conditons) // 返回数量
```

2.3.3 创建、新增

- 创建单条业务对象实体

```
// evjson 指要新增列的数据对象 {}
entity.createOne(tc, entityName, evJson, null) // todo keepId 返回id
```

例子：

```
// 需求：新增库位 id 为 LOC-02 的库位
let traceContext = base.traceContext()
let evJson = {
  "id":"LOC-02"
}
let id = entity.createOne(traceContext, "FbBin", evJson, null)  // 返回 id
```

- 创建多条业务对象实体

```
//evJsonList 指要新增多个数据对象 [{}]
entity.createMany(tc, entityName, evJsonList, null)  // todo keepId
```

例子:

```
// 需求：同时新增库位 LOC-03、LOC-04
let traceContext = base.traceContext()
let evJsonList = [{
  "id": "LOC-03"
},
{
  "id": "LOC-04"
}]
let ids = entity.createMany(traceContext, "FbBin", evJsonList, null) // 返回 id 的数组[]
```

2.3.4 修改、更新

- 更新单条业务对象实体

```
// updateJson 指要更新列的数据对象 {}
entity.updateOne(tc, entityName, conditons, updateJson, null)
```

例子:

```
// 需求：更新库位 LOC-01 为占用
let traceContext = base.traceContext()
let conditions = cq.idEq("LOC-01")          // 更新的条件
let updateJson = {                          // 更新的字段
  "loadStatus": "Occupied",
  "occupied": true
}
let num = entity.updateOne(traceContext, "FbBin", conditions, updateJson, null) // 返回更新的条数
```

- 批量更新业务对象实体

```
// updateJson 指要更新列的数据对象 {}  updateOptions 更新条数数量限制
entity.updateMany(tc, entityName, conditions, updateJson, updateOptions)
```

例子:

```
// 需求：更新所有库位为非占用
let traceContext = base.traceContext()
let conditions = cq.all()                   // 更新的条件
```

```
let updateJson = {                          // 更新的字段
  "loadStatus": "Empty",
  "occupied": false
}
let num = entity.updateMany(traceContext, "FbBin", conditions, updateJson, null) //
返回更新的条数
```

```
// 需求：随机更新一个库位为非占用
let traceContext = base.traceContext()
let conditions = cq.all()
let updateJson = {                          // 更新的字段
  "loadStatus": "Empty",
  "occupied": false
}
let updateOptions = {                       // 更新条数限制
  "limit": 1
}
let num = entity.updateMany(traceContext, "FbBin", conditions, updateJson,
updateOptions) // 返回更新的条数
```

- 根据 id 更新业务对象记录

```
// id 业务对象实体的id updateJson 指要更新列的数据对象 {}
entity.updateOneById(tc, entityName, id, updateJson, null)
```

例子：

```
// 根据库位 id 为 LOC-01 将库位更新为占用
let traceContext = base.traceContext()
let updateJson = {                                                          //
更新的字段
  "loadStatus": "Occupied",
  "occupied": true
}
let num = entity.updateOneById(traceContext, "FbBin", "LOC-01", updateJson, null) //
返回更新的条数
```

2.3.5 删除

- 清空所有业务对象记录缓存

```
entity.clearCacheAll()          // 清空所有业务对象记录的缓存，并不会删除持久化的数据
```

- 清空指定业务对象所有实体

```
entity.clearCacheByEntity(entityName) // 清空指定业务对象记录的缓存，并不会
删除持久化的数据
```

- 根据特定条件删除一条业务对象实体

```
entity.removeOne(tc, entityName, conditions, null)
```

例子：

```
// 需求：
let traceContext = base.traceContext()
let conditions = cq.idEq("LOC-03")                                    // 删除的条件
let num = entity.removeOne(traceContext, "FbBin", conditions, null) // 返回删除的条数
```

- 根据特定条件批量删除业务对象实体

```
// removeOptions 删除记录数量限制，可不指定
entity.removeMany(tc, entityName, conditions, removeOptions)
```

例子：

```
// 需求：随机删除一个库位
let traceContext = base.traceContext()
let conditions = cq.all()
let removeOptions = {                        // 限制删除条数
  "limit": 1
}
let num = entity.removeMany(traceContext, "FbBin", conditions, removeOptions) //
返回删除的条数
```

```
// 删除所有未占用的库位
let traceContext = base.traceContext()
let conditions = cq.eq("occupied", false)                              // 删除条件
let num = entity.removeMany(traceContext, "FbBin", conditions, null) // 返回删除的条数
```

2.3.6 拦截

参数说明 以下变量 entityName 指业务对象名称，func 指脚本方法名

- 实体生命周期拦截器：实体新建前触发的事件

```
entityExt.extBeforeCreating(entityName, func)
```

- 实体生命周期拦截器：实体新建后触发的事件

```
entityExt.extAfterCreating(entityName, func)
```

- 实体生命周期拦截器：实体更新前触发的事件

```
entityExt.extBeforeUpdating(entityName, func)
```

- 实体生命周期拦截器：实体更新后触发的事件

```
entityExt.extAfterUpdating(entityName, func)
```

- 实体生命周期拦截器：实体删除前触发的事件

```
entityExt.extBeforeRemoving(entityName, func)
```

- 实体生命周期拦截器：实体删除后触发的事件

```
entityExt.extAfterRemoving(entityName, func)
```

例子

```
// 需求：库位实体修改之前之后打印一条日志，修改之后打印日志
function boot() {
  entityExt.extBeforeUpdating("FbBin", "binBeforeUpdating");      // 注
册实体修改之前事件
  entityExt.extAfterUpdating("FbBin", "binAfterUpdating");        // 注
册实体修改之后事件
}
// tc:TraceContext 上下文对象；em:业务对象 ids 业务对象实体 id 集合
function binAfterUpdating(tc, em, ids) {
  if (em.getName() != "FbBin")            // 判断是库位的业务对象
    return;
  base.logInfo(tc, JSON.stringify(ids));
}
function binBeforeUpdating(tc, em, ids) {
  if (em.getName() != "FbBin")
    return;
  base.logInfo(tc, JSON.stringify(ids));
}
```

2.4 线程和并发

- 异步执行脚本函数

异步执行一个脚本函数方法，不会阻塞当前业务线程

```
// name: 设置执行线程名字，  func：执行脚本函数的名称
thread.createThread(name, func)
```

- 线程休眠等待

```
thread.sleep(time)  // 线程休眠时间 单位：ms
```

例子:

```
thread.sleep(2000)  // 休眠 2s
```

- 检测线程是否中断

```
thread.interrupted() // bool 类型：true 线程被中断
```

例子:

```
// 并行的查询库位占用数量
function boot(){
  // 创建一个线程
  thread.createThread("QueryBin", "queryBin");
}
function queryBin() {
  let traceContext = base.traceContext();
  while (!thread.interrupted()) {
    let conditions = cq.eq("occupied", true);
    let num = entity.count(traceContext, "FbBin", conditions);
    base.logInfo(traceContext, `占用 num ${num}`);
    thread.sleep(5000);
  }
}
```

2.5 HTTP 客户端

脚本中请求第三方(上位)

参数说明

下文变量 tc 表示上下文，req 表示请求第三方系统的接口参数，如下

```
let req = {
  "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
  "traceReqBody": false,    // 是否记录请求正文, 非必填
  "traceResBody": false,    // 是否记录响应正文, 非必填
  "reqOn": new Date()       // 请求时间, 非必填
}
```

okChecker 表示自定义校验规则非必填,接口函数如下:

```
(res: HttpResult) => boolean;
// res 结构如下
let res = {
  "successful": true,    // http请求是否成功
  "ioError": false,      // 是否连接报错
  "ioErrorMsg": "",      // 连接报错信息
  "code": 200,           // http响应码
  "bodyString": "",      // 响应结果
  "checkRes": null,      // 校验结果。后端会自动补上, 不要手动赋值
  "checkMsg": "",        // 自定义校验提示信息可不填
}
```

callRetryOptions 表示请求失败重试参数, 如下 (非必填)

```
let callRetryOptions = {
  "maxRetryNum": 1,   // 最大重试次数
  "retryDelay": 5000, // 重试间隔
}
```

- 同步请求第三方 (阻塞)

```
httpSyncCall(tc, req, okChecker, callRetryOptions)
```

例子:

```
// 需求：定时向第三方上报库位占用数量
function boot(){
  // 模拟上位的接口
  httpServer.registerHandler("POST", "api/ext/mockAccept", "mockAccept", false);
  // 创建一个线程并定时去上报库位占用情况
  thread.createThread("Dispatch", "scheduledFun");
}
let mockUrl = "http://127.0.0.1:5800/api/ext/mockAccept";
function scheduledFun() {
  base.scheduledAtFixedDelay("reportSitesOccupied", 5000, counter => {
    let traceContext = base.traceContext();
```

```
    // 查询被占用库位数量
    let conditions = cq.eq("occupied", true);
    let num = entity.count(traceContext, "FbBin", conditions);
    //请求参数
    let req = {
      "url": mockUrl,
      "method": "Post",
      "contentType": "Json",
      "reqBody": JSON.stringify({ "num": num })
    };
    let callContainerQty = {
      "maxRetryNum": 1,
    };
    //上报
    let httpResult = httpSyncCall(traceContext, req, null, callContainerQty);
    base.logInfo(traceContext, `上报后返回结果${base.jsonToString(httpResult)}`);
  });
}

function mockAccept(ctx) {
  let traceContext = base.traceContext()
  base.logInfo(traceContext, `模拟接受打印结果：${ctx.getBodyAsString()}`)
}
```

自定义校验规则

```
// 需求：定时向第三方上报库位占用数量，并解析返回值判定成功失败
function boot(){
  // 模拟上位的接口
  httpServer.registerHandler("POST", "api/ext/mockAccept", "mockAccept", false);
  // 创建一个线程并定时去上报库位占用情况
  thread.createThread("Dispatch", "scheduledFun");
}
let mockUrl = "http://127.0.0.1:5800/api/ext/mockAccept";
function scheduledFun() {
  base.scheduledAtFixedDelay("reportSitesOccupied", 5000, counter => {
    let traceContext = base.traceContext();
    // 查询被占用库位数量
    let conditions = cq.eq("occupied", true);
    let num = entity.count(traceContext, "FbBin", conditions);
    //请求参数
    let req = {
      "url": mockUrl,
      "method": "Post",
      "contentType": "Json",
      "reqBody": JSON.stringify({ "num": num })
    };
    let callContainerQty = {
      "maxRetryNum": 1,
    };
```

```
    //上报
    let httpResult = httpSyncCall(traceContext, req, (res) => {
      // 自定义校验规则，请求返回结果 code 为 1 就表示成功
      let jsonRes = JSON.parse(res.bodyString);
      if (jsonRes["code"] == 1) {
        res.checkMsg = "成功";
        return true;
      }
      res.checkMsg = "失败";
      return false;
    }, callContainerQty);
    base.logInfo(traceContext, `上报后返回结果${base.jsonToString(httpResult)}`);
  });
}

function mockAccept(ctx) {
  let traceContext = base.traceContext();
  base.logInfo(traceContext, `模拟接受打印结果：${ctx.getBodyAsString()}`);
  ctx.setJson({ "code": 1 });
}
```

- 异步请求第三方(不阻塞)

```
// scriptFun 指脚本方法名，异步请求后执行的自定义校验规则的脚本方法,可选
httpAsyncCallback(tc, req, scriptFun, callRetryOptions)
```

例子：

```
// 定时上报库位占用数量，异步请求上位
function boot(){
  // 模拟上位的接口
  httpServer.registerHandler("POST", "api/ext/mockAccept", "mockAccept", false);
  // 创建一个线程并定时去上报库位占用情况
  thread.createThread("Dispatch", "scheduledFun");
}
let mockUrl = "http://127.0.0.1:5800/api/ext/mockAccept";
function scheduledFun() {
  base.scheduledAtFixedDelay("reportSitesOccupied", 5000, counter => {
    let traceContext = base.traceContext();
    // 查询被占用库位数量
    let conditions = cq.eq("occupied", true);
    let num = entity.count(traceContext, "FbBin", conditions);
    //请求参数
    let req = {
      "url": mockUrl,
      "method": "Post",
      "contentType": "Json",
      "reqBody": JSON.stringify({ "num": num })
    };
```

```
    let callContainerQty = {
      "maxRetryNum": 1,
    };
    //异步上报，执行 checkResult 脚本方法
    let id = httpAsyncCallback(traceContext, req, "checkResult", callContainerQty); //
返回后台任务记录id
    base.logInfo(traceContext, `返回 id ${id}`);
  });
}

function checkResult(res) {
  base.logInfo(base.traceContext(), `返回 res ${res}`);
  // 判定 http 请求成功
  if (!res["successful"]) return JSON.stringify({ "ok": false })
  // 取响应值
  let bodyJosn = JSON.parse(res["bodyString"])
  // 根据响应值自定义校验规则
  if (bodyJosn["code"] == 1) {
    return JSON.stringify({ "ok": true })
  }
  return JSON.stringify({ "ok": false })
}
```

NOTE 上文提到的 checkResult 方法的入参结构为 HttpResult。自定义脚本校验规则的脚本方法必须返回一个值。

2.6 HTTP 服务器

注册接口方法

```
// method: 请求方式 path: 请求路径 func: 执行的方法名 auth: bool 类型是否需要登录
httpServer.registerHandler(method, path, func, auth)
```

例子： 第三方通过 http 协议请求 M4，可通过注册脚本接口来实现，比如第三方系统通过传库位 id 来查询库位占用状态。参数如下：

```
{
  "siteId": "LOC-01"
}
```

例子：

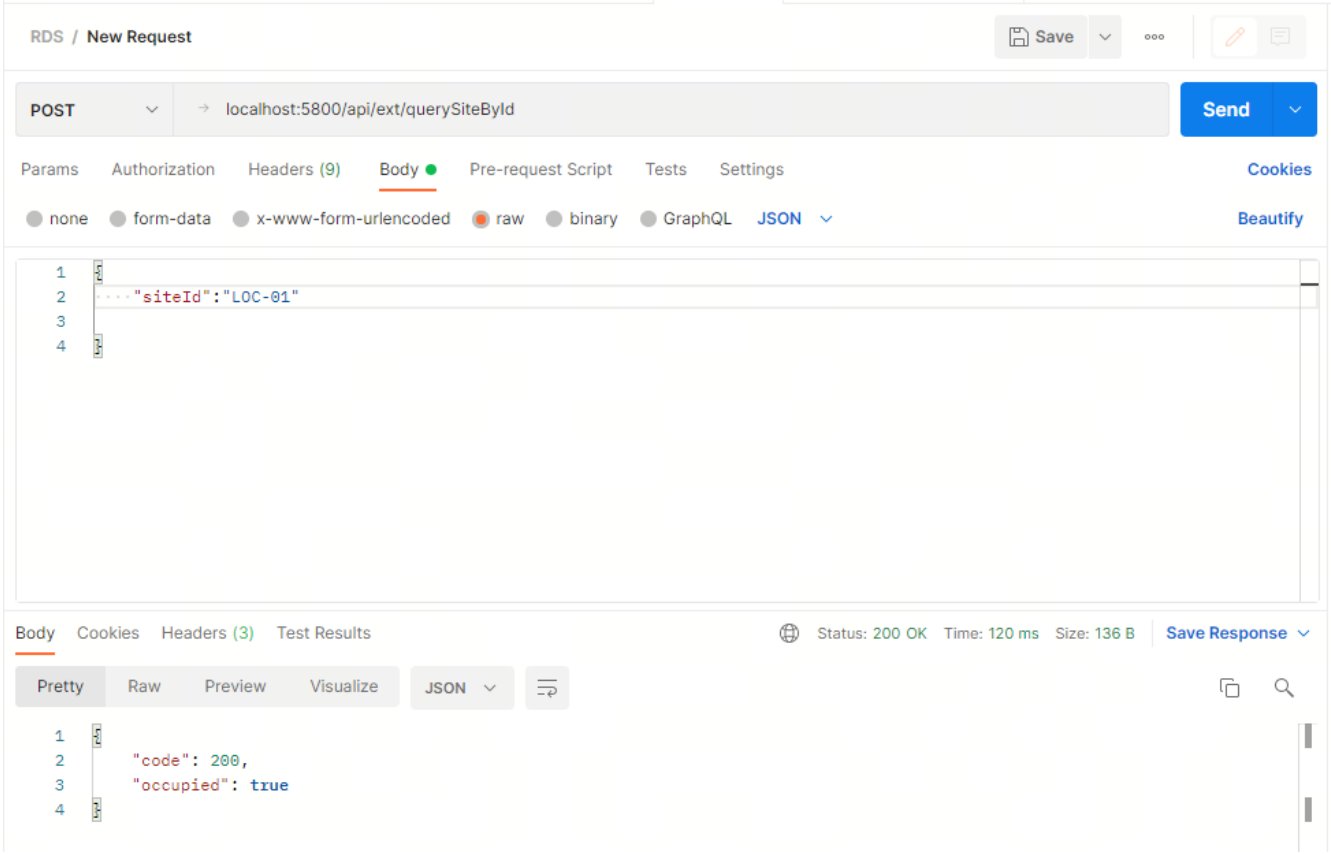
```
function boot() {
  httpServer.registerHandler("POST", "api/ext/querySiteById", "querySiteById", false);
}

function querySiteById(ctx) {
```

```
  try {
    const tc = base.traceContext();
    const reqStr = ctx.getBodyAsString();
    base.logInfo(tc, `收到下单请求：${reqStr}`);
    // req 的数据示例为： { "siteId": "LOC-01"}
    const req = JSON.parse(reqStr);
    const siteId = req["siteId"];
    // 判断 （siteId） 的有效性
    if (utils.isNullOrBlank(siteId))
      throw new Error("库位（siteId）必须是有效的字符串！");
    let site = entity.findOneById(tc, "FbBin", siteId, null);
    if (!site)
      throw new Error(`M4 没有此（siteId）【${siteId}】 ！`);
    ctx.setJson({ "code": 200, "occupied": site.occupied });
  }
  catch (err) {
    ctx.setStatus(400);
    ctx.setJson({ "code": 400, "message": `请求失败：${err}` });
  }
}
```

NOTE 上文中的 `ctx` 结构如下：

```
let ctx = {
  getBodyAsString() // 获取接口请求参数
  setJson({})  // 设置返回值
}
```

Postman 效果图：



2.7 猎鹰任务

异步创建一个猎鹰任务

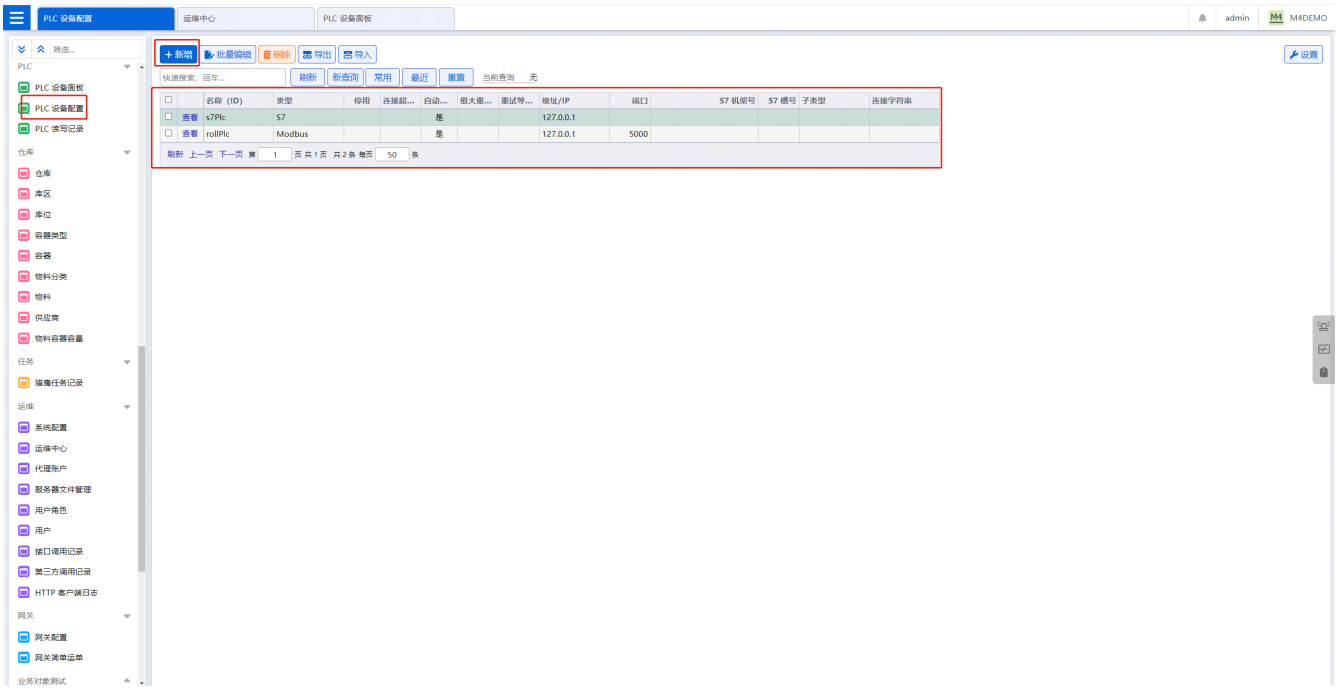
```
// tc:TraceContext 上下文对象； ftDefLabel: 猎鹰任务名称  inputparams: 猎鹰任务的输入参数 类型: {}
falcon.runTaskByLabelAsync(tc, ftDefLabel, inputparams);
```

例子:

详见: 示例

2.8 PLC

首先，如下图在菜单 PLC 配置面板 配置 PLC



2.8.1 ModbusTcp

参数说明

下文 tc 表示上下文， deviceName 表示配置的设备名称（ ID ）；reqMap 表示读取 modbus 地址参数，如下:

```
// 参数意思是：用功能码 03 去读取 0 地址位，从站 1，最大重试次数是2，重试间隔是 5s
let resMap = {
  "code": 3,                  // 功能码
  "address": 0,               // 开始读取的地址位
  "qty": 1,                   // 读取地址位个数
  "slaveId": 1,               // 从站 id
  "maxRetry": 2,              // 最大重试次数，可不指定
  "retryDelay": 5000          // 重试间隔，可不指定
}
```

writeMap 表示写入 modbus 地址参数，如下:

```
// 参数意思是：用功能码 03 去写入 0 地址位，从站 1，最大重试次数是2，重试间隔是 5s
let writeMap = {
  "code": 3,           // 功能码
  "address": 0,          // 写入地址位
  "slaveId": 1,          // 从站 id
  "maxRetry": 2,         // 最大重试次数，可不指定
  "retryDelay": 5000    // 重试间隔，可不指定
}
```

- 只读一次

```
plc.modbusRead(tc, deviceName, reqMap)
```

例子：

```
// 需求：从 0 地址位连续读 3 个地址位
let traceContext = base.traceContext()
let reqMap = {
  "code": 3,
  "address": 0,
  "qty": 3,
  "slaveId": 1,
  "maxRetry": 2,
  "retryDelay": 5000
}
let resPlc = plc.modbusRead(traceContext, "rollPlc", reqMap) // 返回 number[]
```

- 读取直到等于指定值

```
// targetValue: 值目标值, readDelay：读取间隔, 单位 ms
plc.modbusReadUtilEq(tc, deviceName, reqMap, targetValue, readDelay)
```

例子：

```
// 需求：读取 0 地址位值等于1
let traceContext = base.traceContext()
let reqMap = {
  "code": 3,
  "address": 0,
  "qty": 1,
  "slaveId": 1
}
plc.modbusReadUtilEq(traceContext, "rollPlc", reqMap, 1, 2000)
```

- 读取数据，直到等于指定值或达到最大重试次数

```
//readDelay：读取间隔, 单位 ms maxRetry：最大重试次数
plc.modbusReadUtilEqMaxRetry(tc, deviceName, reqMap, targetValue, readDelay, maxRetry)
```

例子：

```
// 需求：读取 0 地址位值等于1, 重试 5 次
let traceContext = base.traceContext()
let reqMap = {
  "code": 3,
  "address": 0,
  "qty": 1,
  "slaveId": 1
}
```

```
let resPlc = plc.modbusReadUtilEqMaxRetry(traceContext, "rollPlc", reqMap, 1, 2000, 5
)// 返回 bool 值, true: 成功, false:
```

- 向某个地址位写入一个值

```
// values：写入值数组：number[]
plc.modbusWrite(tc, deviceName, writeMap, values)
```

例子:

```
// 需求：向地址 0 写入 1
let traceContext = base.traceContext()
let writeMap = {
  "code": 6,
  "address": 0,
  "slaveId": 1,
  "maxRetry": 1,
  "retryDelay": 5000
}
plc.modbusWrite(traceContext, "rollPlc", writeMap, [1])
```

NOTE 如果写入到达最大重试次数还没成功会抛异常

2.8.2 S7

参数说明

tc 表示上下文；deviceName 表示配置的设备名称(ID)；s7ReqMap 表示读取 S7 参数，如下:

```
// 读取 DB1.1.1
let s7ReqMap = {
  "blockType": "DB",    // 可选值有 "DB" | "Q" | "I" | "M" | "V"
  "dataType": "BOOL",   // 可选值有 "BOOL" | "BYTE" | "INT16" | "UINT16" | "INT32" |
"UINT32" | "FLOAT32" | "FLOAT64" | "STRING"
  "dbId": 1,            // db 编号
  "byteOffset": 1,      // 地址位(字节)
  "bitOffset": 1,       // 地址位上的第几位
  "maxRetry": 1,        // 最大重试次数, 可选
  "retryDelay": 2000    // 重试间隔, 单位 ms
}
```

s7WriteMap 表示写入 S7 参数，如下:

```
//  向 DB1.1.1 DB1区 1地址位的第 1 位(从 0 开始)写入
let s7WriteMap = {
  "value": true,        // 写入值
  "blockType": "DB",    // s7区类型 "DB" | "Q" | "I" | "M" | "V"
```

```
  "dataType": "BOOL",   // 数据类型 "BOOL" | "BYTE" | "INT16" | "UINT16" | "INT32" |
"UINT32" | "FLOAT32" | "FLOAT64" | "STRING"
  "dbId": 1,            // db 编号
  "byteOffset": 1,      // 地址位(字节)
  "bitOffset": 1,       // 地址位上的第几位
  "maxRetry": 1,        // 最大重试次数, 可选
  "retryDelay": 2000,   // 重试间隔, 单位 ms, 可选
}
```

- 读一次地址上的一位

```
plc.s7Read(tc, deviceName, s7ReqMap);
```

例子:

```
// 需求：读取 DB1.1.1   DB1 区 1 地址位的第 1 位(从 0 开始)
let traceContext = base.traceContext();
let s7ReqMap = {
  "blockType": "DB",
  "dataType": "BOOL",
  "dbId": 1,
  "byteOffset": 1,
  "bitOffset": 1,
  "maxRetry": 1,
  "retryDelay": 2000 // 重试间隔, 单位 ms
};
let s7Res = plc.s7Read(traceContext, "s7Plc", s7ReqMap); // 返回值类型和
s7ReqMap.dataType 值相关
```

- 读取地址上的一位直到等于指定值

```
// targetValue: 目标值 和 reqMap.dataType 保持一致, readDelay：读取间隔,单位 ms
plc.s7ReadUntilEq(tc, deviceName, reqMap, targetValue, readDelay)
```

例子:

```
// 需求：读取 DB1.1.1   DB1 区 1 地址位的第 1 位(从 0 开始)
let traceContext = base.traceContext();
let s7ReqMap = {
  "blockType": "DB",
  "dataType": "BOOL",
  "dbId": 1,
  "byteOffset": 1,
  "bitOffset": 1,
  "maxRetry": 1,
  "retryDelay": 2000 // 重试间隔, 单位 ms
};
```

```
plc.s7ReadUntilEq(traceContext, "s7Plc", s7ReqMap, true, 2000);
```

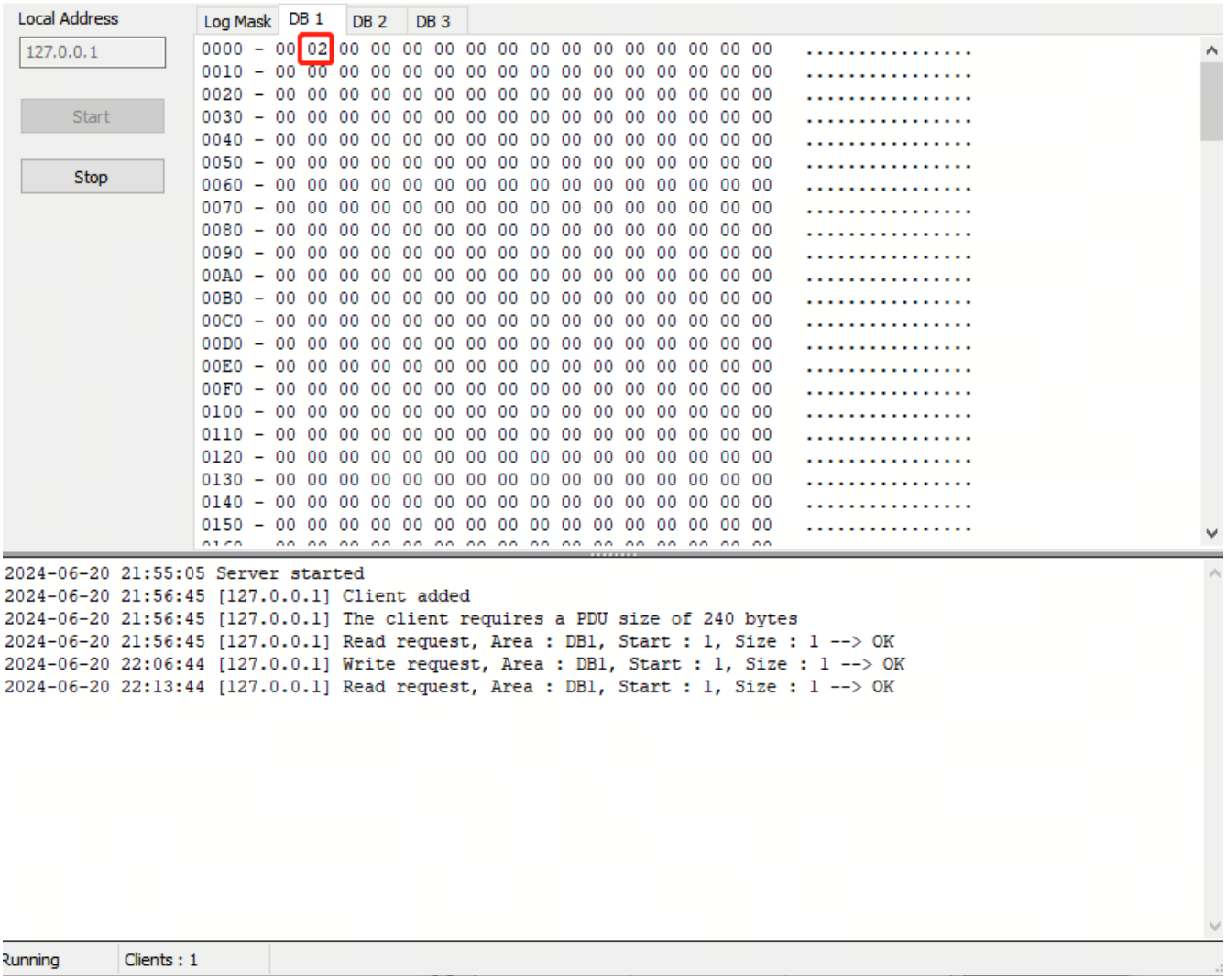
- 向地址位上的指定位写入

```
plc.s7Write(tc, deviceName, reqMap)
```

例子：

```
// 需求：向 DB1.1.1   DB1 区 1 地址位的第 1 位(从 0 开始) 写入 1
let traceContext = base.traceContext()
let s7WriteMap = {
  "value": true,        // 写入值  类型和 dataType 相关
  "blockType": "DB",    // s7区类型 "DB" | "Q" | "I" | "M" | "V"
  "dataType": "BOOL",   // 数据类型 "BOOL" | "BYTE" | "INT16" | "UINT16" | "INT32" |
"UINT32" | "FLOAT32" | "FLOAT64" | "STRING"
  "dbId": 1,            // db 编号
  "byteOffset": 1,      // 地址位(字节)
  "bitOffset": 1,       // 地址位上的第几位
  "maxRetry": 1,        // 最大重试次数, 可选
  "retryDelay": 2000,   // 重试间隔, 单位 ms, 可选
}
plc.s7Write(traceContext, "s7Plc", s7WriteMap)
```

S7 仿真效果：


Local Address
127.0.0.1
Start
Stop
Log Mask
DB 1
DB 2
DB 3
0000 - 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0010 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0020 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0030 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0040 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0050 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0060 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0070 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0080 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0090 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00A0 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00B0 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00C0 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00D0 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00E0 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00F0 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0100 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0110 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0120 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0130 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0140 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0150 - 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
2024-06-20 21:55:05 Server started
2024-06-20 21:56:45 [127.0.0.1] Client added
2024-06-20 21:56:45 [127.0.0.1] The client requires a PDU size of 240 bytes
2024-06-20 21:56:45 [127.0.0.1] Read request, Area : DB1, Start : 1, Size : 1 --> OK
2024-06-20 22:06:44 [127.0.0.1] Write request, Area : DB1, Start : 1, Size : 1 --> OK
2024-06-20 22:13:44 [127.0.0.1] Read request, Area : DB1, Start : 1, Size : 1 --> OK
Running
Clients : 1

2.9 机器人

2.10 其他

三、M4 PLC 和工业设备对接

3.1 Modbus TCP

Modbus TCP 是一种常见的工业对接协议。比较简单。简单说，就是通过对设备变量（地址）的读写，实现功能。

例如一个三色灯设备，暴露一个变量（地址）。写 1 表示亮红色，写 2 表示亮绿色，写 3 表示亮黄色，写 0 表示关灯。再比如一个滚筒线对接口，暴露两个变量（地址）。第一个变量供读取，1 表示可以放货，0 表示不能放货。第二个变量供写入，1 表示放货完成，0 表示未放货。

Modbus 协议其实有两种，TCP 和 RTU（串口）。这里只讨论 TCP 协议，即基于以太网的 Modbus 协议。

一个设备（作为 `Slave`）有一个 IP 和一个端口。M4 连接到这个设备（作为 `Master`）。

Modbus 有两种变量类型。一种是布尔量（也称线圈量）**1 个 bit（比特）**，只有两个值，真或假，或者说 0 和 1。另一种是寄存器变量，是一个整数，短整数，**占用 1 个 word（字）**，能表示 0 ~ 65535。

NOTE	1 word（字） = 2 bytes（字节）

有两种看待 Modbus 设备的角度，从存储看或者从读写看。

从存储看，Modbus 协议规定了 4 个存储区，分别是 0 1 3 4 区，其中 0 区和 4 区是可读可写，1 区和 3 区是只读。

区号	名称	读写	地址范围
0区	输出线圈	可读可写布尔量	00001-09999
1区	输入线圈	只读布尔量	10001-19999
3区	输入寄存器	只读寄存器	30001-39999
4区	保持寄存器	可读可写寄存器	40001-49999

但在与客户或第三方沟通时，更直接且准确的方式是从读写角度看。问：“用什么功能码、读哪个地址。特别是当非 PLC 对 PLC 通讯，如 PLC 对 C# 或 Java 代码，不同系统设备对上面存储定义是有细微差异的，但功能码一定是准的。

协议规定了很多功能码，常用的有（注意此功能码用十六进制表示）：

功能码	含义	英文含义
01	读取线圈/离散量输出值	ReadCoils
02	读取离散量输入值	ReadDiscreteInputs
03	读取保持寄存器	ReadHoldingRegisters
04	读取输入寄存器	ReadInputRegisters
05	写单个线圈/单个离散输出	WriteSingleCoil
06	写单个保持寄存器	WriteSingleRegister
0F	写入多线圈	WriteMultipleCoils
10	写入多寄存器	WriteMultipleRegisters

例如，在对接协议时，可以约定“使用 01 功能码，读取地址 1，表示货物是否就绪”、“当取货完成，使用 05 功能码，写入地址 10”。

3.1.1 字符串

Modbus 原生只支持线圈（bool）和寄存器（word），不支持字符串、浮点型、长整型，为此提几个标准，项目中优先按照此标准。

此标准基于 **寄存器** 设计；对于多字节，暂时只支持大端。

暂时只支持ASCII字符串。即一个字符用一个字节表示。

方案协议中约定字符串的首末位地址、解析模式。

每个地址的解析模式有 2 种（需在方案协议中约定）：

- 一个地址表示一个字符，如 `0x0041` 表示 `A`， `0x0061` 表示 `a`。
  - 优点：方便处理，每个地址视为一个字符，无需拆分。
  - 缺点：不使用左侧字节，只使用地址的右侧字节，会浪费一定空间。
- 一个地址表示两个字符，如 `0x4161` 表示 `Aa`， `0x3132` 表示 `12`。
  - 地址中第一个字节表示第一个字符，第二个字节表示第二个字符。
  - 此模式 **不支持** 只读一个地址中的第一个字节/第二个字节，如：`400001 ~ 400002` 的 2 个地址只读前 3 个字节的值，忽略最后一个字节。
  - 如果字符串实际长度小于定义长度，则右侧无实际值的字节写 `00`。
  - 优点：充分利用空间。
  - 缺点：每个地址要拆开处理，不便于操作；必须要填满所有字节。

暂不支持：

- 一个地址表示字符串长度，然后连续读 N 个地址，每个表示一个字符串。
- 非 ASCII 编码的字符。

例： `400001 ~ 400003` 的 3 个地址表示一个定长的容器号，每个地址表示一个字节。Modbus 中的 `0x0041 0x0042 0x0043` 表示字符串 `ABC`。

3.1.2 长整数

Modbus 的寄存器（word）占 2 个字节，对应 Java 的短整数（Short）类型，原生不支持四字节整数（Integer）、八字节整数（Long）。

此标准基于 **寄存器** 设计；对于多字节，暂时只支持大端。

方案协议中约定长整数的首末位地址、整数占用字节数量。

例：在 `400001 ~ 400002` 写一个四字节的长整数，Modbus 地址中的 `0x1234 0x23EF` 表示十进制的值 `305406959`。

支持：

- 四字节整数
  - 连续两个地址。第一个地址表示高位字节，第二个自制表示低位字节。
- 八字节整数
  - 连续四个地址。第一个地址表示最高位字节，第四个自制表示最低位字节。

3.1.3 浮点型

对于 N 位小数，乘以 10 的 N 次方写入 Modbus。如约定 3 位小数。如果是 3.14，则 Modbus 地址里写 3140；如果是 234.123，写入 234123。如果数值范围两字节整数表示不了，则按上述大整数处理。

方案协议中约定 N 的值、占用字节数、首末位地址。

此标准基于 **寄存器** 设计；对于多字节，暂时只支持大端。

NOTE 需要保证原始值乘 10 的 N 次方之后，一定是整数，不能留小数。

例 1：在 `400001` 写一个 2 字节浮点数，将原始值乘 10 的 2 次方写入。原始值为 1.2，则在 Modbus 地址里写 120。

例 2：在 `400001 ~ 400002` 写一个 4 字节的浮点数，将原始值乘 10 的 5 次方写入。原始值为 3.14159，则在 Modbus 地址里写 314159。

支持：

- 两字节浮点数
- 四字节浮点数
- 八字节浮点数

3.2 S7

3.2.1 基础

S7 是西门子 Smart 系列 PLC 的内部集成的一种通讯协议，封装了更丰富的数据类型，如整型、浮点型、字符型，其本质也是读写地址空间上的字节。

目前只讨论 Client/Server 的单边通信，Client 记录配置并主动访问，Server 准备被访问的数据。PLC 作为 Server，M4 作为 Client。

PLC 可将地址空间上的数个字节打包为一个 S7 节点，用于存储变量值，供 M4/PLC 读写。

链接到 Server 需要的参数

参数	描述	默认值
ip	PLC 的 IP	
port	PLC 的端口号	102
rack	机架号	0
slot	槽号	0

NOTE PLC 工程师不知道 rack、slot 时，尝试用 0。

读写时需要的参数

参数	描述	默认值
dbId	db 编号	1
offset	S7 节点的第一个字节在地址空间的索引	
length	S7 节点的长度	
type	S7 节点的数据类型	

西门子常见 PLC 型号 S1200、S1500。

3.2.2 数据类型

PLC 将地址空间的字节定义为数个 S7 节点。例：将下图的 0000~000F 地址定义为一个 S7 STRING，将 0010～0011 定义为一个 S7 INT16。

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0010	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0020	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0030	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0040	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0050	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0060	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0070	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0080	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
0090	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
00A0	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00
00B0	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00	$00

S7 数据类型取值范围 & 与 Java 数据类型的映射关系

S7 数据类型	Java 数据类型	取值范围	特殊说明
BOOL	Boolean	true/false	一个 BOOL 节点只占用一个 bit，一个字节能拆成 8 个 BOOL 节点
BYTE	Byte	-128～127	
INT16	Short	-32768～32767	
UINT16	Integer	0～65535	
INT32	Integer	-2147483648~2147483647	
UINT32	Long	0~4294967295	
FLOAT32	Float	-3.402823E+38 ~ -1.175495E-38（负数）；+1.175495E-38 ~+3.402823E+38（正数）； 0.0※	
FLOAT64	Double		
STRING	String		String 类型在内存中的结构为：定义长度 + 实际长度 + 实际值。详情见下文

S7 BOOL

S7 将一个字节拆为 8 个 BOOL 节点，偏移量如 123.1、123.2，但不推荐这样用（Java 的库有可能要按照字节来，所以可能需要单独处理，所以不推荐用，推荐用 INT16）。

S7 STRING

S7 的 STRING 分为 3 段：定义长度（1 字节）a + 实际长度（1 字节）b + 实际值（N 字节，占用 a 个字节，字符串存储在前 b 个字节）。

例：字符串 ABCDEF 定义长 10，但实际长 6，则第一个字节为 10，第二个字节为 6，第三个字节开始为 ABCDEF，后面 4 个字节无效，需要忽略。

3.2.3 S7 与 Modbus 的区别

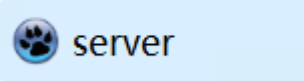
Modbus 只能读某个线圈（1 bit）或者某个寄存器（2 字节），而 S7 一般是读写一个或多个字节；所以 Modbus 的取值范围较小且只能是布尔型 Boolean 或者短整型 Short，S7 额外支持整数 Integer、长整数 Long、浮点型 Float、字符串 String。

使用上，Modbus 只能写 true/false 或者 0～65535 的整数，不能写小数、字符串；S7 支持 true/false、较大范围的整数、小数、字符串。

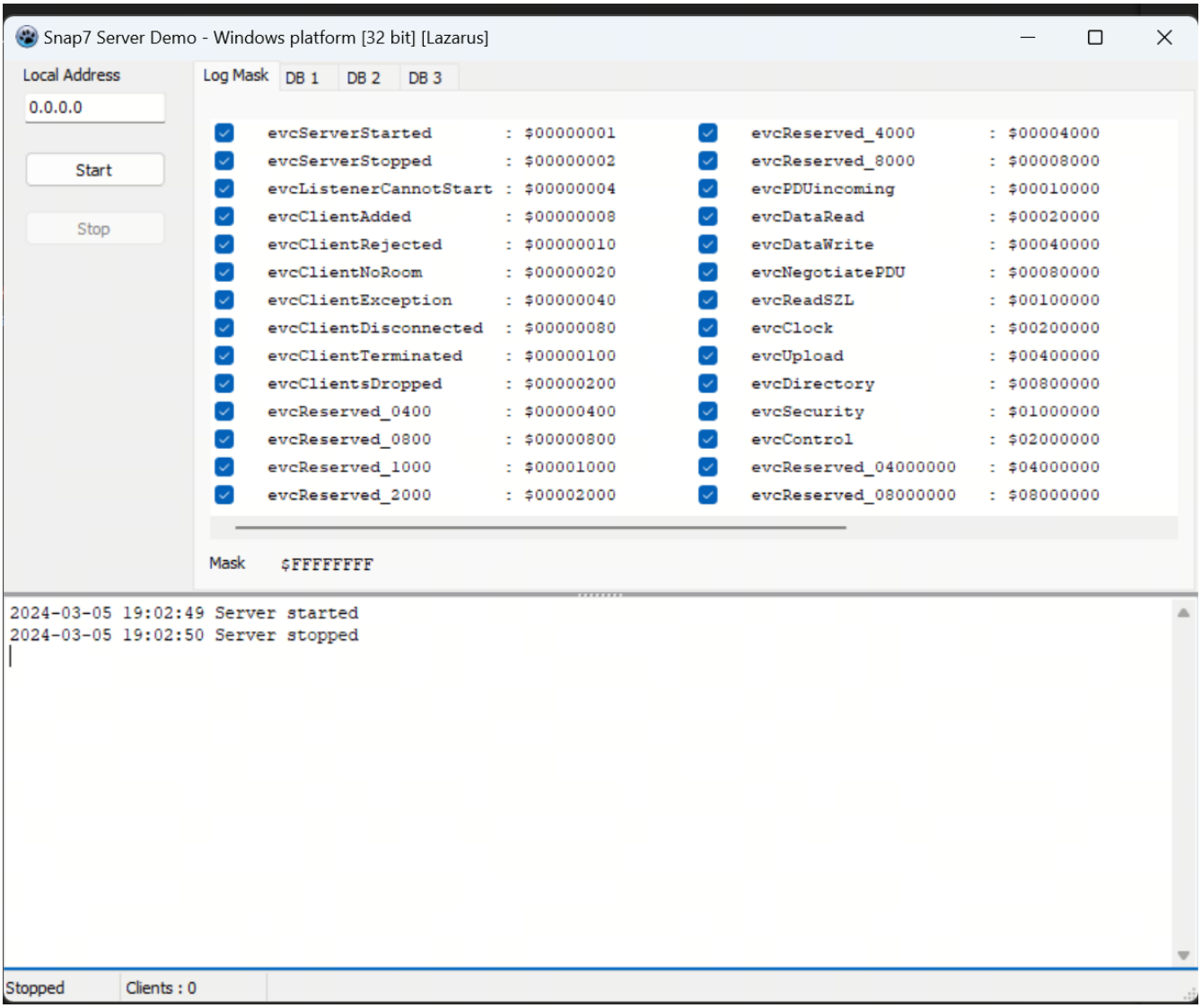
例：要将容器编号 “BOX-A-023” 传给 PLC， S7 可以直接传 “BOX-001”；Modbus 需要与 PLC 定义一套转换规则 “去掉‘BOX-’前缀，并将 A 映射为 1，再去掉前面的 0” ，然后再将处理过的数据 “1”，“23” 传给 PLC。

3.2.4 仿真工具 snap7

Server



双击 server.exe 启动 S7 Server 模拟器，用于仿真 PLC，提供地址空间。



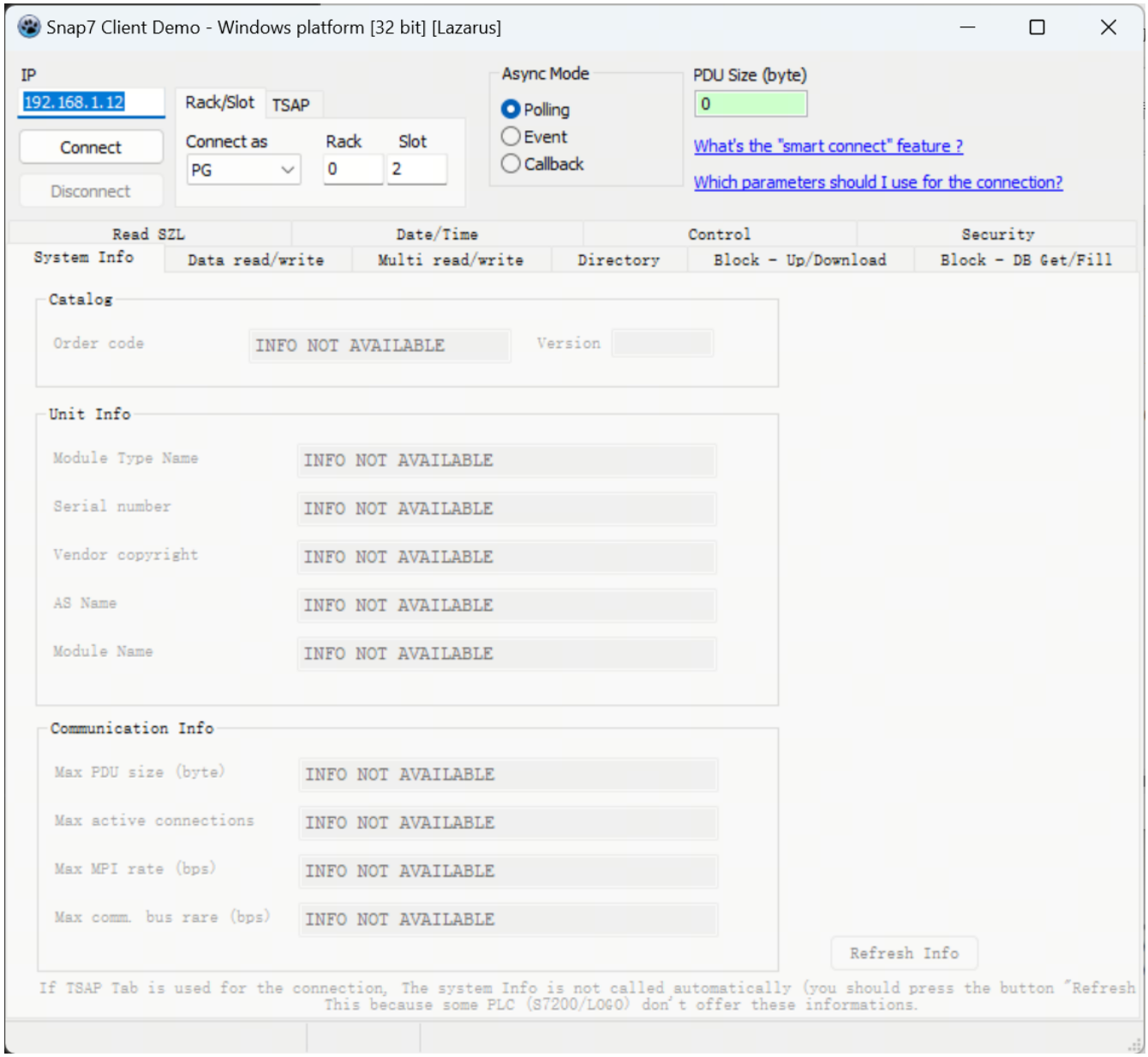
将 “0.0.0.0” 改为 “127.0.0.1”。

点击【start】，即可启动 S7 仿真 Server

Client

clientd

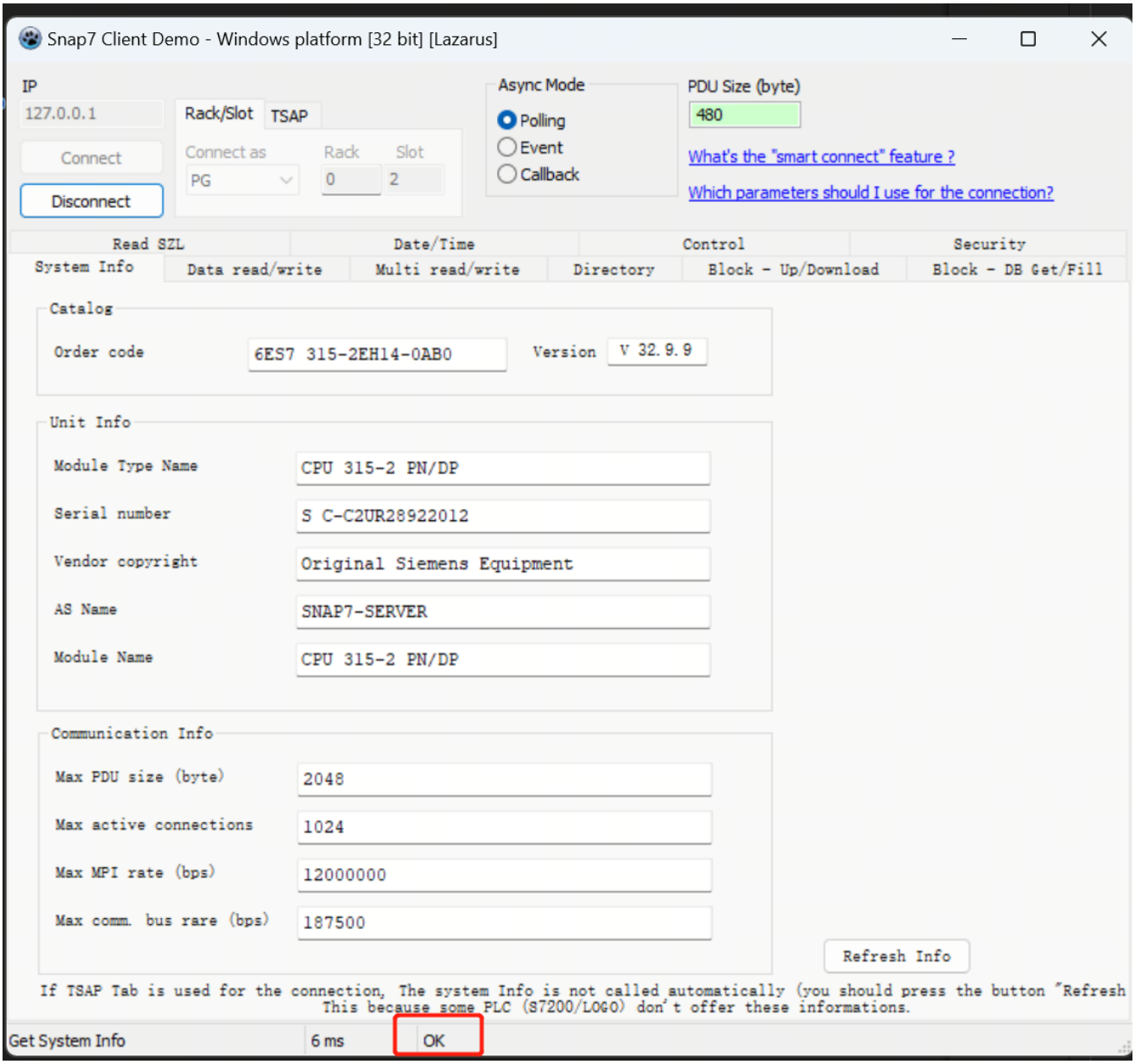
双击 clientd 启动 S7 Client 模拟器，等价于 M4，常用于读取 PLC 地址空间的值。



修改 IP 为 “127.0.0.1”。

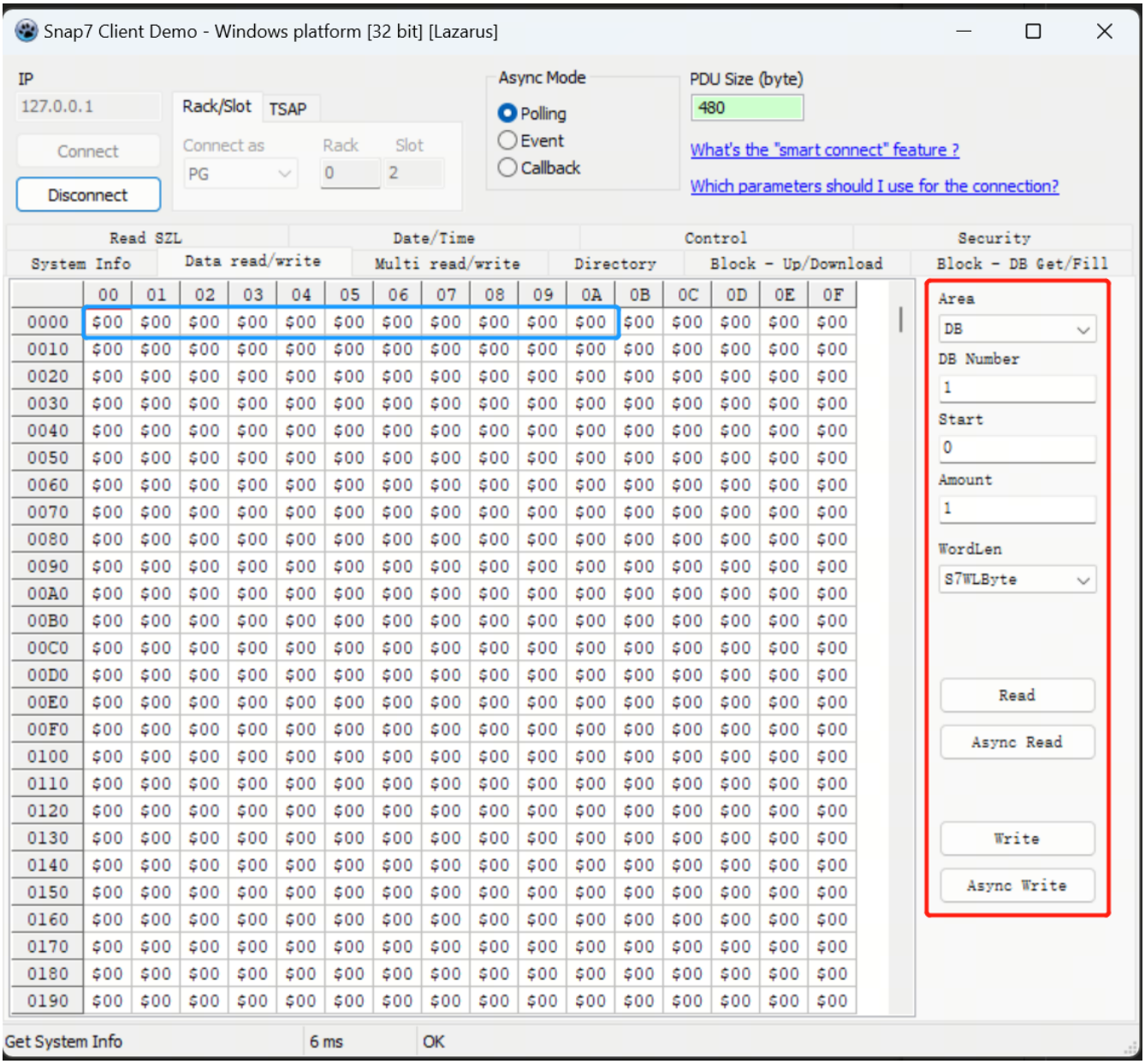
如果链接真实 S7 Server 的话，需将 Rack、Slot 改为 PLC 真实的值。

点击【Connect】即可建立链接



看到红框中的 “OK” 就代表链接成功。

点击 【Data read/write】 切换到读写区域。



读节点的操作步骤

1. 修改【DB Number】为实际的 dbId
2. 修改【Start】为节点的偏移量 offset
3. 修改【Amount】为节点的长度（单位：字节）
4. 点击【Read】按钮
5. 读取蓝框中的值，并将其转为实际的格式

写节点的操作步骤

1. 按照 “读节点” 都出该节点的值
2. 修改上图蓝框中的值
   a. 这里写的是 16 进制，不是字符，也不是 10 进制的数字
   b. 注意是大端还是小端
3. 点击【Write】按钮