M4 脚本开发文档

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，如果使用 JavaScript 的话，推荐使用 TypeScript 进行开发，再编译成 JavaScript。 entity.updateMany 系统启动时，会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，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，再执行 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。

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。
OfzybaKcjo9s1oxzoBncO8zJnog.png

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。
LzzFbqjxSoSkEvxm3V0cmzAIn6g.png

空白的 “猎鹰任务编辑” 界面如下图所示。 LsCPbg74wo5YnOxfivhc8U1QnNg.png

设置新建的猎鹰任务的名称为 “货物转运”。 UxHObUDscofAyRxtD5HcCojtnle.png

添加 “任务起点” 作为 “任务输入参数”。 BRX1bxsy1ohpIzxzPXPcKlAznhf.png

继续添加 “任务终点” 作为 “任务输入参数”。 SxLobSY2yogNgAxodp6c6hTenGh.png

为了方便操作，将 “插件列表” 固定在界面上。 JCj4bwgWRoWnUaxvvJ2cmmnInLg.png

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。 JUrgbEPJaoZ2mcxsCVUcqmfinWf.png

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。 K7a1bcm9OoWE8OxYeAMcZXQknPd.png

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。
RKR0bo1YNoM4ISxZOnAcqrfInvb.png

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。 GtB0bMQgaoW7VDx7unFcU7lBnRd.png

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。

MngsbAqEiob5xsxHYbIc27PKnzh.png

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。 QelzbGh5GopQzFx3o7WcrvWRnUg.png

猎鹰任务的执行记录如下图所示：

QA7ibVeoEomBlgxt6M1cFeJenQg.png

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。

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。 E4YBbOiVCoyRZXxEPkac6cNOn7c.png

重启脚本之后，M4 的日志中会记录 “脚本启动成功”。 BbNIbiJZbopDUCxg1eectDs7ncd.png

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) as {from: string; to: string};
    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";

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);
// 拦截器，当指定的业务对象（猎鹰任务记录:FalconTaskRecord）发生改变时，就调用指
定的方法
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：${reqStr}); // req 的数据示例为： { "taskId": "xxxxx" } const req = JSON.parse(reqStr); base.logDebug(tc, 任务结束了：${req["taskId"]}); 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：
XUmRbtQHDoVxvgxswlhcsN4rnee.png

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 nullutils.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 1utils.anyToInt("100"); // 结果为 100utils.anyToInt("+100"); // 结果为 100utils.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 1utils.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. 0utils.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. 0utils.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。

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 1base.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)

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t

```
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

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})
```

}

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}`});
```

}

}

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

Postman 效果图：

Mx0obfvoLoJKMIxXAyucdc0Enfd.png

2.7 猎鹰任务

异步创建（运行）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

ODYpbu1BloqbKDxPouNc6ufLnff.png

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])

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：

MMBNbcH5PoF7jnxv0AOcsU6Tnge.png

2.9 机器人

2.10 其他

单行搜索、条码解析

条码解析基础

TODO

解析添加多单行

返回 newLines 字段。

示例代码：

// 获取输入框中的文本信息

const sn = ctx.keyword

console.log("sn:", sn)

if (!sn) return null

// 前缀是 host:port/api/；需要注意跨域的问题，在脚本中通过接口转发规避此问题。

// 通过调用 HTTP 接口，获取单行信息

const data = await __jk.getHttpClient().post("mock/MES/GetPalletizingInfo", { sn }).data if (data["success"] !== true) { __jk.popup.alert("异常", 获取条码[${sn}]的单行信息失败：${data["msg"]})

}

const lines = data["lines"]

if (!lines) {

```
__jk.toast.toastError(`未找到条码[${sn}]对应的单行信息！`)
```

return

}

// 根据业务对象中定义的单行结构，将获取到的单行信息，转换为可以被界面加载的数据

const newLines = [

```
{ "btMaterial": "DC", "qty": 55 },
{ "btMaterial": "XQ", "qty": 50 },
{ "btMaterial": "LS", "qty": 35 }
```

]

return { newLines }