开发手册

接口文档

管理平台中接口文档

前端开发

11 items

后端开发

9 items

接口文档

点我跳转到平台接口文档

搭建平台开发环境

AIRIOT 的搭建平台开发环境

React + Antd

AIRIOT 的React + Antd

在平台中运行你的组件

AIRIOT 的��平台中运行你的组件

React组件示例

AIRIOT React组件示例

项目打包及上传

AIRIOT 的代码打包及上传

使用系统API

AIRIOT 的使用系统API

属性扩展

AIRIOT 的属性扩展

前端驱动表单配置

AIRIOT 的前端驱动表单配置

xadmin

AIRIOT 的xadmin

基于Vue框架扩展平台组件

airiot-vue 项目用来实现 vue 扩展，包括：扩展 vue 组件作为画面组件，设置路由直接访问 vue 组件页面

使用系统ws订阅报警数据

使用系统ws订阅报警数据

搭建平台开发环境

示例项目目上传至 https://github.com/Yayahannn/airiot-demo 以下是具体说明

构建平台开发环境

接下来，我们将学习如何在本地构建开发环境。这将涉及一些前端开发的基础知识，包括 npm、webpack 和 babel 等。但你也无需过分关注这些细节，因为平台的开发工具包已经为你处理了一切。

安装工具

建议使用 Visual Studio Code Visual Studio Code（以下简称 VS Code）是一款广泛使用的集成开发环境（IDE）。请根据你的环境下载相应版本并进行安装。

创建项目

首先需要安装 Node.js 和 npm，然后使用 `npm` 来初始化一个项目，在项目文件夹右键打开 `终端` ，然后执行以下命令创建项目：

```
npm init
```

执行后将会逐步看见以下内容：

```
package name:    //项目的名称，用于标识您的 Node.js 项目
version:         //项目的初始版本号，默认为 1.0.0
description:     //对项目的简要描述，可以用来介绍项目的主要功能或目的。
entry point:     //项目的主要入口文件，默认为 index.js，即 Node.js 在引入模块时首先查找的文件。
test command:    //用于定义运行项目测试的命令。
git repository: //项目的 Git 仓库地址，如果有的话。
keywords:        //与项目相关的关键词，有助于其他人更容易地找到您的项目。
author:          //项目的作者或贡献者。
license: (ISC)   //项目所采用的许可证类型，默认为 ISC。
```

如果以上内容均用默认值即可，也可以执行以下命令来代替前面的 `npm init`

```
npm init -y
```

> 有关 npm 的使用，请参考 npm 的使用文档
> node 下载地址：https://nodejs.org/en/

创建好的项目目录中应该包含一个 `package.json` 文件，这是 JavaScript 项目的描述文件，我们将在后面修改其中的一些配置。

安装平台依赖包

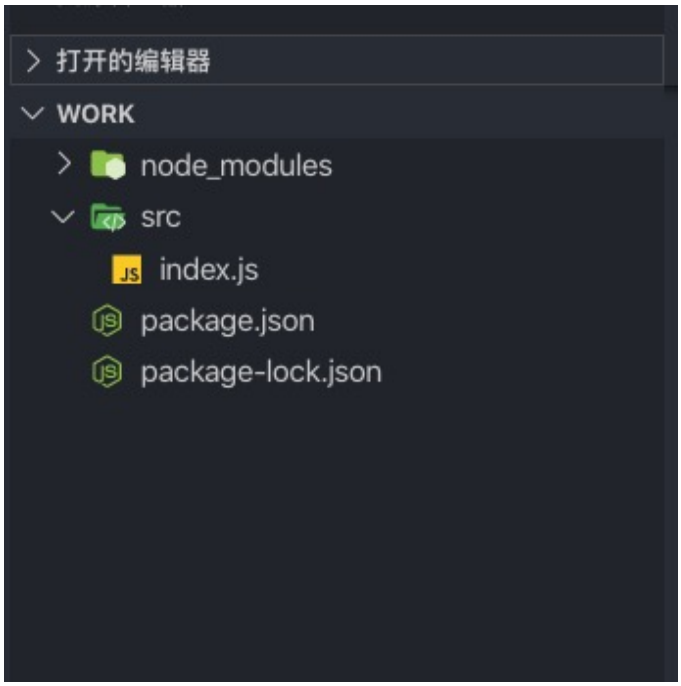
接下来，我们需要安装平台的依赖包。在你的项目目录中执行以下命令来安装平台包：

```
# 安装平台开发工具包
npm i -D @airiot/devtool@1.0.7
# 安装运行时的基础依赖包，你的项目中依赖其他组件可以自行安装
npm i -S react@18.2.0 xadmin@3.0.1 antd@4.24.12
# 如果 npm 安装速度较慢，你可以切换至淘宝镜像
npm config set registry http://registry.npm.taobao.org/
```

创建index.js

项目默认的入口文件为 src/index.js，因此你需要创建一个 src 目录，并在其中创建 index.js 文件，这个文件包含你要实现的代码。

到这一步，项目的目录结构应该如下：



修改index.js

```
import React from 'react';
import { app } from 'xadmin';

app.use({
  name: 'iot.test',
})
```

修改Package.json

修改`package.json`文件，`main` 代表平台后台入口文件，`iotFront` 代表程序前台入口文件，`iot_module` 表示是否是 IoT 扩展模块，`files` 代表需要上传到服务器的文件目录，`iotDependencies` 代表运行时需要加载的内部模块，默认只加载 `core` 模块。

```
{
  "name": "@airiot/projectname",
  "version": "1.0.0",
  "description": "",
  "main": "dist/index.js",
  "iotFront": "dist/front.js",
  "iot_module": true,
  "scripts": {
    "start": "airiot start",
    "build": "airiot build",
    "upload": "airiot upload",
    "test": "airiot test",
    "deploy": "airiot install"
  },
  "files": [
      "dist",
      "package.json"
    ],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@airiot/devtool": "^1.0.7"
  },
  "dependencies": {
    "antd": "^4.24.12",
    "react": "^18.2.0",
    "xadmin": "^3.0.1"
  },
   "iotDependencies":[
    "@airiot/i18n",
```

```
    "@airiot/auth",
    "@airiot/components",
    "@airiot/dashboard"
  ]
}
```

配置平台端口

在src同级创建`.dev.js`,内容如下:

```
module.exports = {
  host: 'http://xxx.xx.xxx.xx:xxxx/',  //前端访问地址
}
```

运行

在终端中执行以下命令来运行项目:

```
npm run start
```

访问平台

在浏览器中输入下面地址来访问刚刚创建的项目： projectID为项目id

```
http://localhost:3000/_p_projectID
```

React + Antd

IOT平台的前端采用`React`框架，因此首先需要学习如何使用`React`。已经掌握`React`的同学可以跳过本课程。

官方资料

React

React 是一个用于构建用户界面的 JavaScript 库。你可以在以下资料文档中系统地学习 React。React 版本需要是 `>=16.8.0`。

React官方文档

这是官方提供的React文档，是学习React的正规途径。从体验React的理念到深入使用API和Hooks等特性，初学者可以仅学习核心概念章节。

React Hook官方文档

我们重点强调了React的Hook特性，请您着重学习React的Hook功能特性。使用Hook来开发React组件是IOT平台建议的组件开发方式。(PS: 未来React官方可能会剔除其他的组件开发方式，也就是基于Class的组件开发方式)

我们为什么建议使用 Hooks 来开发组件?

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。它是React未来最主流的开发组件方式，同时Vue3也借鉴了Hook的实现方式。有关Hook的优势及给我们带来的好处，可以参考这篇文章我们为什么要拥抱React Hook。

Antd

antd 是基于 Ant Design 设计体系的 React UI 组件库，主要用于研发企业级中后台产品。

我们可以使用html和css来制作组件，但是现代浏览器应用中有很多常见的组件使用html+css无法直接开发出来，因此我们需要用到已经开发好的UI组件库，方便我们开发复杂的应用组件。React的UI组件库有很多，其中使用最广泛的是蚂蚁金服开发的Antd组件库。

Antd官方组件文档(v4版本)

学习antd的途径非常简单，就是查看其官方组件文档。你不必熟记每个组件的使用方法和属性，只需要对组件库有整体的了解就行。在实际开发中，我们可以随时查看这个组件文档学习其组件的使用方法。

其他资料

浅析React&Vue两大流行框架优缺点

目前主流的UI框架只有Vue和React，该文章分析了两个框架的优缺点，可以通过框架的对比更加深入地理解React的设计思想。如果你有Vue的开发经验，强烈建议学习一下该文章，能够更顺利地转向React开发。

听说你还不懂React Hook?

从Vue和React的对比中不难看出，Vue3的设计思路更多地借鉴了React Hook特性。可以说Hook是未来前端UI框架的主流设计方法，建议任何从事前端开发的人员都能够更深入地理解和使用Hook。这篇文章通俗易懂地讲解了Hook的由来和使用，是学习Hook的必读文章。

React Hook + TS 购物车实战

这是一个实战型的Hook使用实例，可以帮助你更深入地理解和使用Hook，避免踩坑。

深入进阶

如果你想了解React更深入的知识，建议学习以下文档

React技术揭秘

这个链接深入解析了React的开发和演进原理，带领读者一点点分析React的源码。

深入 React Hook 系统的原理

创建自己的React

这是一个英文文档，以实例代码的方式一步步地打造一个自己的React框架，是深入理解React的最佳途径。

课程任务

通过本章节的学习，你应该能够完成一个包含简单逻辑的React组件。请完成一个包含以下交互逻辑的组件，并在线运行你的组件。

任务目标

- 创建一个表单，包含一个数字输入框
- 创建两个按钮：增加和减少
- 一个数字显示的文本组件，初始显示数字可以由props传入，默认为0
- 每隔一秒钟数字加1
- 点击增加按钮：增加数字输入框中的数字，例如输入框中填写5，数字显示10，点击增加变为15；点击减少按钮为减少逻辑
- 使用Hook方式实现
- 使用ES6语法编写

在平台中运行你的组件

欢迎来到本教程，如果你已经掌握了React的组件开发并成功安装了平台的开发环境，那么下一步我们将帮助你将你开发的组件放到平台中运行，从而完成基于平台的组件开发。

创建新页面

首先，让我们来学习如何在指定路由下创建自定义页面。你需要修改 `index.js` 文件，内容如下：

```
import React from 'react';
import { app } from 'xadmin';

// 定义一个简单的React组件
const TestComponent = () => {
    return <a>hello world</a>
}

// 在路由中注册自定义页面
app.use({
  name: 'iot.test',
  routers: {
    '/app/': {
      path: 'myTest', // 指定路径
      breadcrumbName: '我的测试组件页', // 面包屑名称
      component: TestComponent // 注册组件
    },
  }
})
```

在这段代码中，我们导出了一个对象，这个对象被称为一个扩展模块。该模块包含了多个可配置的属性，其中包括 name、routers 等。稍后我们将专门讲解所有的扩展属性。

通过上述操作，你应该可以在 `/app/myTest` 页面下看到 `TestComponent` 组件，页面将显示 "hello world"。这样你就成功创建了一个新页面，其中的 `TestComponent` 可以替换成你自己编写的任何React组件。

创建画面组件

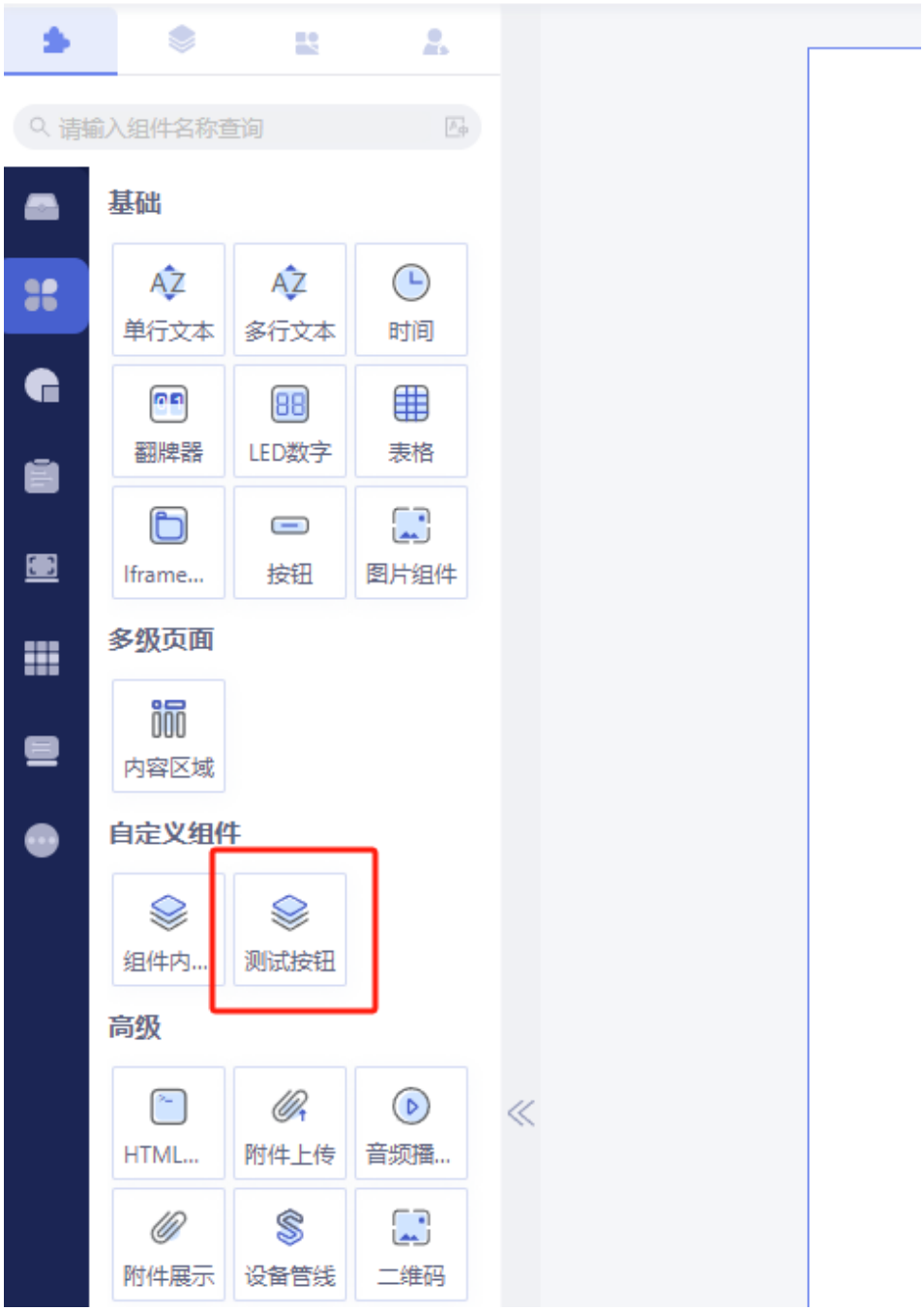
画面编辑器是平台的核心功能之一，编辑器中的小组件我们称之为 `widget`。`widget` 同样也是一个React组件。下面的代码演示了如何将一个React组件注册为一个新的 `widget`，就像下图所展示的 `单行文本`、`LED数字` 等。

通过 `app.use` 方法传入对象中添加 `dashboardWidgets` 属性进行扩展。 修改 `index.js` 文件，内容如下：

```
import React from 'react';
import { app } from 'xadmin';

// 定义一个简单的React组件
const TestComponent = () => {
    return <a>hello world</a>
}

// 定义一个简单的按钮组件
const TestButton = props => {
  const { text, fontSize=14, fontColor } = props
  return <button style={{ fontSize, color: fontColor }}>{text || 'Hello'}</button>
}

// 定义组件的可配置属性格式，使用json Schema格式编写
const paramSchema = {
  // 参数模式，这里定义了一个对象类型的参数模式
  type: 'object',
  // 包含的属性
  properties: {
    // 文字属性，用于定义文字内容
    text: {
      // 标题为“文字”
      title: '文字',
      // 类型为字符串
      type: 'string'
    },
    // 文字大小属性，用于定义文字的大小
    fontSize: {
```

```
      // 标题为“文字大小”
      title: '文字大小',
      // 类型为数值
      type: 'number'
    },
    fontColor: {
      title: '文字颜色',
      type: 'string',
      fieldType: 'color'
    }
  }
}


// 定义一个新的widget，注册为'测试按钮'，并添加到组件中
const TestWidget = {
  title: '测试按钮', // 标题
  category: ['通用组件','自定义组件'], // 分类
  component: TestButton, // 组件
  initLayout: { width: 40, height: 20 }, // 初始化布局
  paramSchema // 可配置属性格式
}

// 在dashboardWidgets中注册新的widget
app.use({
  name: 'airiot.test',
  routers: {
    '/app/': {
      path: 'myTest', // 指定路径
      breadcrumbName: '我的测试组件页', // 面包屑名称
      component: TestComponent // 注册组件
    },
  },
  dashboardWidgets: {
    'test.widget': TestWidget // 注册widget
  }
})
```

注册成功后，我们会在画面编辑页的`组件库`的`通用组件-自定义组件`中看到刚刚注册的`测试按钮`组件，如图所示：



自定义fieldtype

我们可以创建自定义的`fieldtype`来满足特定组件配置的需求。这些自定义的`fieldtype`需要在`form_fields`中注册后方可使用。

```
// 在这里，我们展示了如何创建自定义fieldtype以满足特定组件配置的需求
app.use({
  name: 'airiot.test',
  routers: {
    '/app/': {
```

```
      path: 'myTest', // 指定路径
      breadcrumbName: '我的测试组件页', // 面包屑名称
      component: TestComponent // 注册组件
    },
  },
  dashboardWidgets: {
    'test.widget': TestWidget // 注册widget
  },
  form_fields: {
    // 自定义数字输入框
    customNumber: {
      component: ({ input, field }) => {
        return <Input {...input} {...field.attrs} placeholder='自定义数字输入框'/>
      },
      // 解析函数
      parse: (value) => {
        return value === '' || value == null ? null : parseFloat(value)
      },
      // 属性配置
      attrs: {
        type: 'number',
        style: {
          maxWidth: 200
        }
      }
    }
  },
  // 自定义schema转换器
  schema_converter: [(f, schema, options) => {
    if (schema.type === 'number' && schema.fieldType == 'customNumber') {
      f.type = 'customNumber'
    }
    return f
  }],
})
```

使用自定义Fieldtype

在组件的paramSchema中使用自定义的fieldtype：

```
const paramSchema = {
  // 参数模式，这里定义了一个对象类型的参数模式
  type: 'object',
  // 包含的属性
  properties: {
    // 文字属性，用于定义文字内容
    text: {
```

```
      // 标题为“文字”
      title: '文字',
      // 类型为字符串
      type: 'string'
    },
    // 文字大小属性，用于定义文字的大小
    fontSize: {
      // 标题为“文字大小”
      title: '文字大小',
      // 类型为数值
      type: 'number',
      fieldType: 'customNumber' // 使用自定义的数字输入框
    },
    fontColor: {
      title: '文字颜色',
      type: 'string',
      fieldType: 'color' // 使用自定义的颜色选择器
    }
  }
}
```

在上述代码示例中，我们展示了如何创建自定义`fieldtype`以满足特定组件配置的需求。这些自定义`fieldtype`包括对数字类型和颜色的特殊处理。

注册成功后，当您选择该组件时，您将在组件的`基本属性`中看到刚刚注册的自定义`fieldType`所带来的效果，具体效果如下图所示：

json schema官方文档

官方json schema文档，是学习json schema的正规途径。

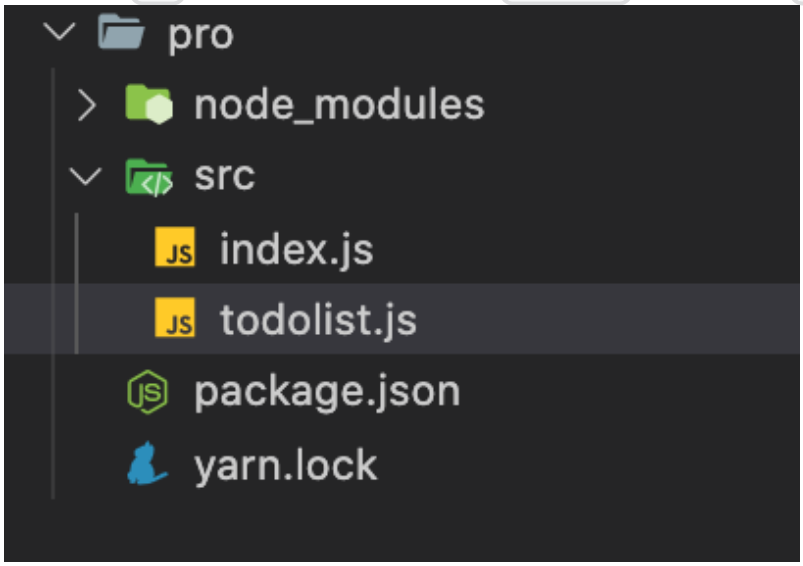
`widget`的属性

属性	说明	类型
title	标题	string
category	在组件列表中每种分类会按组显示	string
icon	编辑器内的图标	string
component	React组件	object
paramSchema	可通过该属性拓展组件可配置属性，该属性会根据自定义的schema在属性配置自动生成表单，表单值会通过props传入component	object

React组件示例

这是一个带有交互操作的React Todolist组件，如果您已经熟练掌握React，可以忽略这部分内容。

首先，在`src`目录下，新建一个名为`todolist`的文件，与`index`文件同级，如下所示：



下面是`Todolist`组件的代码：

```
import React from 'react'
import { Row, Col, Button, Input, Checkbox } from 'antd'
import { DeleteOutlined } from '@ant-design/icons'

// 标题组件
const Title = props => {
  return <h3>todo list</h3>
}

// 列表展示组件，展示items、是否完成状态、删除事件
const List = props => {
  const { items, onDelete, onChange } = props

  const onCheckboxChange = e => {
    const name = e.target.value
    const checked = e.target.checked
    onChange(name, checked)
  }

  const defaultValue = items && items.filter(item => item.status).map(item => item.name) ||
[]

  return (
    <Checkbox.Group style={{ margin: '20px 0' }} defaultValue={defaultValue}>
```

```
      <Row>
        {items && items.length > 0 ? items.map(item => {
          const unfinished = item.name
          const finished = <span style={{ textDecoration: 'line-through' }}>{item.name}
</span>
          const label = item.status ? finished : unfinished
          return (
            <Col span={24}>
              <Checkbox
                onChange={onCheckboxChange}
                value={item.name}>{label}</Checkbox>
              <DeleteOutlined
                onClick={() => onDelete(item.name)}
                style={{ position: 'absolute', right: 0, top: 4, cursor: 'pointer' }}
              />
            </Col>
          )
        }) : null}
      </Row>
    </Checkbox.Group>
  )
}

// 添加item组件
const Add = props => {
  const [state, setState] = React.useState('')

  const onClick = () => {
    props.onAdd(state)
    setState('')
  }

  const onChange = e => {
    const value = e.target.value
    setState(value)
  }

  return (
    <>
      <Input placeholder={'请输入待办事项'} value={state} onChange={onChange} />
      <Button block onClick={onClick}>添加</Button>
    </>
  )
}

// Todolist组件
const TodoList = props => {
  // 初始化数据
  const [items, setItems] = React.useState([
    { name: 'learn english', status: false },
```

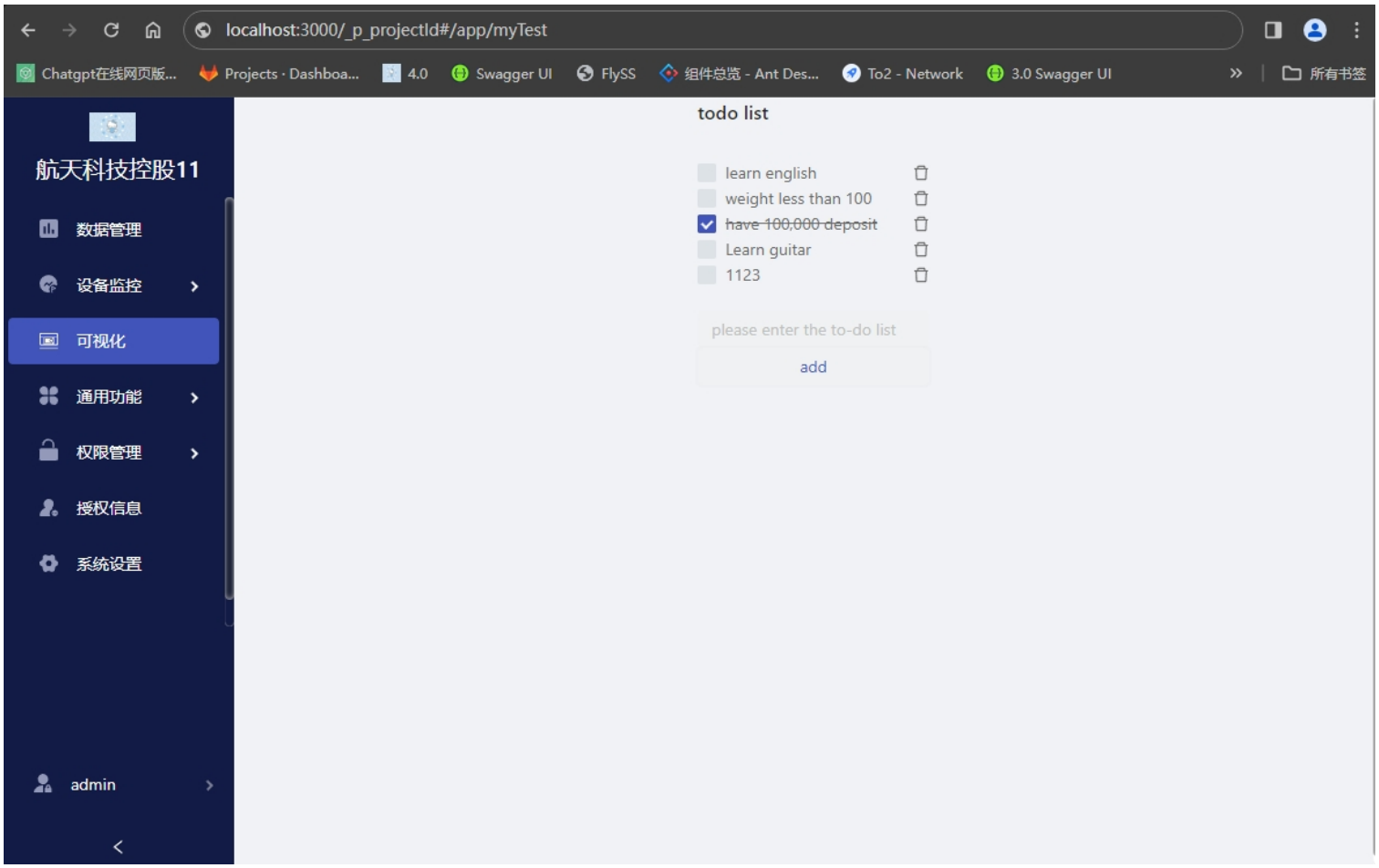
```
    { name: 'weight less than 100', status: false },
    { name: 'have 100,000 deposit', status: true },
    { name: 'Learn guitar', status: false },
  ])

  const onAdd = name => {
    const newItems = [...items, { name, status: 0 }]
    setItems(newItems)
  }

  const onDelete = name => {
    const newItems = items.filter(item => item.name !== name)
    setItems(newItems)
  }

  const onChange = (name, status) => {
    const newItems = items.map(item => {
      if (item.name === name) {
        item.status = status
      }
      return item
    })
    setItems(newItems)
  }

  return (
    <div style={{ width: 200, margin: '0 auto' }}>
      <Title />
      <List items={items} onDelete={onDelete} onChange={onChange} />
      <Add onAdd={onAdd} />
    </div>
  )
}

export default TodoList
```

在`index.js`中引入`Todolist`并且注册到页面,代码如下.

```
import { app } from 'xadmin';
import TodoList from './todolist'

app.use({
  name: 'iot.test',
  routers: {
    '/app/': {
      path: 'myTest',
      breadcrumbName: '我的测试组件页',
      component: TodoList
```

```
    },
  }
})
```

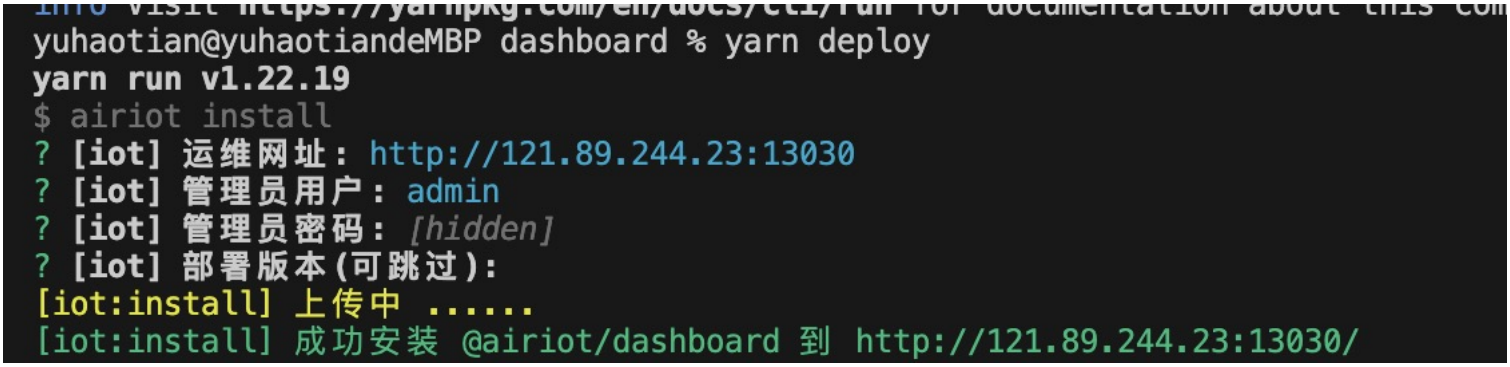
预览页面



这个示例中，我们使用了Ant Design的组件库来构建交互式的Todolist组件。通过这个示例，你可以学习到如何使用React和Ant Design来构建实际的交互组件。

项目打包及上传

1. 编译代码 , 在命令行执行npm run build或yarn build
2. 将编译好的代码上传至指定服务器，执行 npm run deploy 或 yarn deploy
3. 如成功 会显示 输入运维网址, 如网址正确 , 会显示输入管理员姓名 和 管理员密码, 回车, 显示成功安装 , 即上传完毕。如图:



注: 执行yarn命令需要安装yarn包管理工具

使用系统API

在前面的章节中提到，AIRIOT平台的底层框架使用的是xadmin。与Vue类似，xadmin也提供了一些实用的API，便于前端应用的开发。

xadmin的API

xadmin提供的API接口如下：

App.api(options)

参数:

`{Object} options` 参数是一个包含组件选项的对象，参数格式如下：

属性	说明	类型
name	请求地址	string
resource	优先级高于name	string

用法:

调用后端API接口请求或提交数据，相当于`fetch`或`axios`。参数是一个包含组件选项的对象，返回一个`Promise`对象。

```
import { api } from 'xadmin'

api({ name: `core/t/schema` })
  .query({}, {})
  .then(json => console.log(json))
  .catch(err => console.log(err))
```

返回值:

```
[{ name: 'A', id:'A'}, ...]
```

方法:

query

```
import { api } from 'xadmin'
// 查询所有表
api({ name: `core/t/schema` })
 .query({},{})
 .then(json => console.log(json))
 .catch(err => console.log(err))

// 查询表记录
const tableId = ''
const filter = { ship: 1, limit: 20, fields: ['device'] }
const wheres = { }
api({ name: `core/t/${tableId}/d` })
 .query(filter,wheres)
 .then(json => console.log(json))
 .catch(err => console.log(err))
```

filter和wheres参数说明

添加和修改

```
import { api } from 'xadmin'
const tableId = ''
const data = { id, name, position }
const partial = false
api({ name: `core/t/${tableId}/d` })
 .save(data,partial)
 .then(json => console.log(json))
 .catch(err => console.log(err))
```

属性	说明	类型
data	保存的数据,存在id时为修改	string
partial	修改数据时使用PUT或者PATCH方法，默认使用PATCH方法	boolean

删除

```
import { api } from 'xadmin'
const tableId = ''
```

```
const id = ''
api({ name: `core/t/${tableId}/d` })
 .delete(id)
 .then(json => console.log(json))
 .catch(err => console.log(err))
```

属性扩展

什么是json schema?

Json Schema定义了一套词汇和规则，这套词汇和规则用来定义Json元数据，且元数据也是通过Json数据形式表达的。Json元数据定义了Json数据需要满足的规范，规范包括成员、结构、类型、约束等

相关文档链接: JSON Schema官方文档

根据数据结构生成简单schema工具: 在线JSON转Schema工具

schema的扩展配置

```
const CustomField = props => {
  const { input:{ onChange, value } } = props
  retun '我是一个自定义表单项'
}

const mydriver = {
  type: 'object',
  properties: {
    ip: {
      type: 'string',
      format: 'ipv4',
      title: '设备IP',
      fieldType: 'ip',
      field:{
        effect: ({ value }, form) => {
            setTimeout(() => {
              form.setFieldData('port', { display: value === '1' })
            })
          }
      }
    },
    port: {
      type: 'number',
      title: '端口',
    },
    unit: {
      type: 'number',
      title: '站号',
      field:{ component: CustomField },
      description:'生产站的序号',
      field:{
       validate:(value) => {
```

```
        if (value < 0 || value > 1) {
          return '超出取值范围'
        } else {
          return ''
        }
      }
     }
    },
  },
  formEffect: form =>{
      form.useField('port', state => {
      let value = state.value
      //port控制unit站号显示和隐藏
      if (value) {
        form.setFieldData('unit', { display: true })
      } else {
        form.setFieldData('unit', { display: false })
      }
    })
  },
  required:['ip'],
  form:['ip','port'],
}
```

属性	说明
required	必填项
form	显示的表单项
fieldType	使用内置自定义表单项组件
formEffect	表单钩子
description	字段说明

内置的表单项组件可以通过fieldType指定

组件	配置	其他配置 |
文字输入框	type: 'string', title: '文字'	

组件	配置	其他配置 |
文字输入框（高级）	type: 'string', title: '文字', fieldType: 'input'	textType: 'input'(单行), 'textArea'(多行) textContent: 'text', 'password', 'email', 'tel', 'id'
文字输入框（带ip校验）	type: 'string', title: 'IP', fieldType: 'ip'	
数字输入框	type: 'number', title: '数字'	
数字输入框（高级）	type: 'number', title: '数字', fieldType:'inputNumber'	max: 100, min:10, unit: '天', (单位) dbType: 'Int32', 'Int64', 'Double' decimal: 3 (小数位数)
布尔输入	type: 'boolean', title: '布尔值'	
选择器	type: 'string', title: '选择器', enum: ['1', '2', '3'], enum_title:['A','B','C']	
时间选择器	type: 'string', title: '时间', fieldType: 'datePicker'	format: 'date'(年月日), 'datetime'(年月日-时分秒)

field

参数	描述	类型 |
suffix	显示单位	string

参数	描述	类型 |
attrs	当type为'string'时示例 attrs：{ type: 'number', //输入框的类型 type: 'textarea' //文本地域 placeholder：'请输入' // 默认提示 style: { width: '120px' } ...更多属性请参考ant desig组件Input	
}	object	
validate	校验规则可以是正则，return 出的就是错误信息 默认返回null	function

effect	effect表单联动与formEffect类似，effect能直接取到当前输入的值 例如
	effect: ({ value }, form) => {
	setTimeout(() => {
	form.setFieldData('unid', { display: value === '1' })
	})
	}
		function

component	自定义输入组件， 在组件本身内props里可取到字段所有信息 （组件输入值必须与字段的type对应）	ReactNode

formEffect

该属性为函数，传入form参数，常用来处理表单联动，schema 加载时执行 form参数常用方法

- `useField` 监听表单字段修改

```
const fieldKey = 'name'
form.useField(fieldKey, state => {
  console.log('表单字段修改了', state.value)
})
```

- `setFieldData` 设置某字段状态

```
// 隐藏fieldKey
const fieldKey = 'name'
form.setFieldData(fieldKey, { display: false })
```

- `change` 设置某字段值

```
// 修改值
const fieldKey = 'name'
form.change('test', 'abc')
```

- `getState` 获取整个表单状态

```
const formState = form.getState()
```

- `getFieldState` 获取某字段值

```
const fieldKey = 'name'
const formFieldState = form.getFieldState(fieldKey)
```

前端驱动表单配置

如何在前端注册驱动

```
// 表设置
const settings = {
  type: 'object',
  title: '驱动配置',
  properties: {
    ip: {
      type: 'string',
      format: 'ipv4',
      title: '设备IP',
      fieldType: 'ip',
    },
    port: {
      type: 'number',
      title: '端口'
    },
    timeout: {
      type: 'number',
      title: '连接超时时间 (s)',
      description: '默认为10s'
    },
  }
}

// 设备设置
const device = {
  type: 'object',
  properties: {
    ip: {
      type: 'string',
      format: 'ipv4',
      title: '设备IP',
      fieldType: 'ip',
    },
    port: {
      type: 'number',
      title: '端口'
    },
    unit: {
      type: 'number',
      title: '站号'
    },
  },
}
```

```
// 参数
const tags = {
  title: '数据点',
  type: 'array',
  items: {
    type: 'object',
    properties: {
      name: {
        type: 'string',
        title: '名称'
      },
      id: {
        type: 'string',
        title: '标识'
      },
      offset: {
        type: 'number',
        title: '偏移地址'
      },
    },
    required: ['name', 'id', 'offset'],
    form: ['*']
  }
}

// 指令
const commands = {
  title: '指令',
  type: 'array',
  items: {
    type: 'object',
    properties: {
      name: {
        type: 'string',
        title: '名称'
      },
      form,
      ops: {
        type: 'array',
        title: '指令',
        items: {
          type: 'object',
          properties: {
            offset: {
              type: 'number',
              title: '偏移地址'
            },
            index: {
              type: 'number',
```

```
              title: '下标',
              description: '适用于寄存器区控制多开关情况'
            },
            value: {
              type: 'string',
              title: '默认写入值'
            },
            single: {
              type: 'boolean',
              title: '单字节'
            }
          }
        }
      }
    }
  }
}

export default app.use({
  name: 'iot.custom_driver',
  drivers:{
    cutomDriver:{
      title: '自定义XXX协议',
      key: 'cutomDriverName',
      table: {   // 表配置
        properties: {
          settings,
          tags,
          commands
        }
      },
      device: {  //  设备配置
        properties: {
          settings:device,
          tags,
          commands
        }
      }
    }
  }
})
```

xadmin

xadmin是平台的核心框架，平台的扩展功能都是使用xadmin来实现的，同时xadmin也实现了一些常用的扩展模块，包含表单和CRUD等模块。

xadmin的设计思想

xadmin的设计思想非常简单。用最简单的办法实现了现在流行的微前端架构，让我们一步步的来看一下xadmin是如何实现。

目的

首先xadmin核心的目的是创造一个可插拔式的 `模块` 管理框架，模块与模块之间实现`数据共享`、`逻辑复用`、`组件共享` 以及 `逻辑重写`、`组件重写` 等等。

模块创建与加载

调用`app.use()`方法加载模块，例如：

```
// 注册一个名称为custom.models的模块
app.use({
  name:'custom.models'
})
```

该方法传入一个对象，对象内包含`内置属性` (参考...) 以及可以扩展自定义属性。

如何使用`内置属性`

```
// 获取系统内以注册的所有模块
app.get('models')
// 获取系统内注册所有模块的组件
app.get('components')
```

如何扩展自定义属性，实现模块共享

```
app.use({
  // items内注册自定义属性的类型，该方法定义的类型会直接影响调用app.get的返回结果
  items:{
    customA:{ type:'map' }
  },
  // 定义自定义属性
```

```
    customA:{ a: 1 }
})
```

基于Vue框架扩展平台组件

airiot-vue 项目用来实现 vue 扩展，包括：扩展 vue 组件作为画面组件，设置路由直接访问 vue 组件页面

开发步骤

1. 获取airiot-vue项目

示例项目目上传至 https://github.com/Yayahannn/airiot-vue/tree/main

2. 控制台执行安装三方包

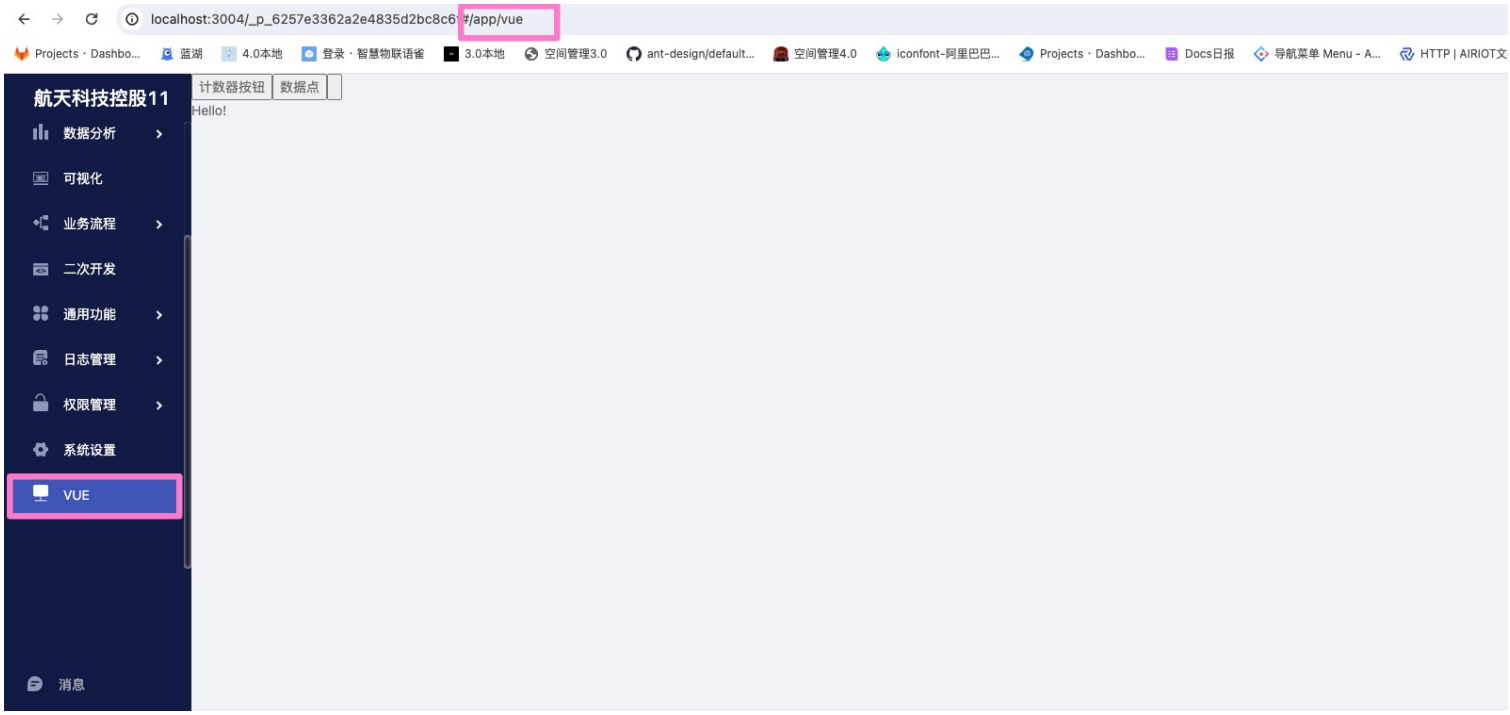
```
npm install
```

3. 启动项目

```
npm start
```

4. 启动成功访问页面

(1) 路由访问 vue 页面



(2) 画面组件编辑页

能够正确访问上面两个截图，说明项目成功启动

5. 项目文件说明

文件	功能
src --> TestWidget.vue	vue 组件实现
src --> VuePage.js	用 vue 组件生成的页面

文件	功能
src --> VueWidget.js	把 vue 组件扩展成画面组件
src --> index.js	入口文件，把 vue 页面及 画面组件 加到路由中
src --> front.js	前台入口文件，把 vue 页面及 画面组件 加到路由中
src --> hooks.js	提供了一个 vue 的 hook

使用系统ws订阅报警数据

订阅报警数据

```
import React from 'react';
import { app, use } from 'xadmin';

// 定义一个简单的React组件
const TestComponent = () => {
 return <a>hello world</a >
}

// 定义一个简单的按钮组件
const TestButton = () => {
const [state, setState] = React.useState('')
// 当前登陆用户信息
const { user } = use('auth.user')
// 当前登陆用户token
const userToken = user?.token
const userName = user?.username

const { onData, subscribe } = use('ws')
// 订阅数据ws推送的返回数据
onData(data => {
   setState(`${data.tableData?.name}:${data.desc}`)
})

React.useEffect(() => {
 const where = {
   "tableDataSetting": [
      { "table": { "id": "sbb1" }, "selectRecord": [{ "id": "bjsb" }] },
     { "table": { "id": "cj" }, "selectRecord": [{ "id": "33" }] },
   ]
}
   /**
    *  订阅数据,
    *  第一个参数指定订阅数据类型
    *  第二个参数可指定订阅哪些表和记录的报警数据,格式为where所示
   */
   return subscribe('warning', {})
 }, [])

 return (
   <>
     <p>当前用户:{userName}</p >
     <p>{state}</p >
   </>
```

```
  )
}

// 定义一个新的widget，注册为'测试按钮'，并添加到组件中
const TestWidget = {
  title: '测试按钮', // 标题
  category: ['通用组件', '自定义组件'], // 分类
  component: TestButton, // 组件
  initLayout: { width: 40, height: 20 }, // 初始化布局
}

// 在dashboardWidgets中注册新的widget
app.use({
  name: 'airiot.test',
  routers: {
    '/app/': {
      path: 'myTest', // 指定路径
      breadcrumbName: '我的测试组件页', // 面包屑名称
      component: TestComponent // 注册组件
    },
  },
  dashboardWidgets: {
    'test.widget': TestWidget // 注册widget
  }
})
```

平台后端服务开发

本文将介绍如何开发一个自己的后端服务, 并打包和部署到平台中.

Java

7 items

Node

7 items

Python

7 items

Go

7 items

Dotnet

7 items

驱动schema说明

本文将详细介绍 数据接入驱动 的 schema 的格式, 以及如何根据自己的需求定义 schema.

算法schema说明

本文将详细介绍 算法集成 的 schema 的格式, 以及如何根据自己的需求定义 schema.

流程扩展节点接入schema说明

本文将详细介绍 流程扩展节点接入 的 schema 的格式, 以及如何根据自己的需求定义 schema.

如何开发一个平台后端服务

本文将介绍如何开发一个自己的后端服务, 并打包和部署到平台中.

介绍

当需要扩展平台功能，或者需要实现系统集成时，可以开发一个独立的服务程序，然后将该服务打包并部署到平台中.

后端服务大概分为两类:

- 一类是运行一些后台任务, 例如: 定时任务, 事件处理等. 该类服务通常不对外提供接口
- 另一类是提供一些接口供前端调用

这两类服务在开发上是相同的, 在打包方面稍有不同. 本文将以 Java 语言为例介绍如何开发第二类服务.

开发步骤

接下来将会详细介绍如何开发一个 Java 后端服务, 以及如何将该服务打包并部署到平台中.

1. 创建项目

与 `Java SDK` 中的数据接入驱动开发类似, 创建一个标准的 `springboot` 项目即可. 项目创建完成后, 根据自己业务的需求向项目中添加相应的依赖包. 可以参考 数据接入驱动开发-创建项目.

> **INFO**
>
> 如果业务中需要访问平台接口, 需要引入 `Java SDK`, 详情见 平台接口客户端

2. 根据自身业务需求实现代码

这一步骤与普通的 `springboot` 项目开发没有区别, 根据自己的业务需求实现相应的代码即可. 如果涉及平台接口的调用, 请查看 平台接口客户端 内容.

> **INFO**

对于开发第二类服务来说, 需要服务内的所有接口提供统一的请求前缀, 例如: `/my-service`, 这样平台才能正确的代理该服务. 该前缀可以在 `application.yml` 文件中配置, 例如:

```yaml
server:
  servlet:
    context-path: /my-service
```

3. 打包

这一步骤介绍如何将一个开发好的 Java 程序打包为可在平台部署的安装包.

平台中使用 `traefik` 组件作为平台网关, 可以为平台中的后端服务提供统一的访问入口、负载均衡、统一认证等功能. 所以, 当开发第二类后端服务时, 需要在程序之外额外提供一些配置信息, 以便 `traefik` 能够正确识别和代理后端服务, 这些配置信息保存在 `service.yml` 文件中, 需要和程序一起打包.

1. 打包程序

使用 `springboot` 自带的打包插件将程序代码打包为 `jar` 文件. 该过程与普通的 `springboot` 项目打包没有区别.

```shell
# 使用 maven 打包
mvn clean package
```

2. 程序配置说明

`springboot` 程序的配置文件通常保存在 `application.yml` 文件中. 如果部署在 `windows` 平台时, 要求配置文件 `application.yml` 必须放在 `jar` 包外面, 因为平台在安装服务时, 需要读取该配置文件的内容.

> **INFO**
>
> `springboot` 默认从多个路径读取配置文件, 可以将 `application.yml` 和 `jar` 文件放在同一个目录下, 或者放在 `jar` 文件同级目录 `config` 中.

3. 编写 service.yml

该文件内容描述了服务的一些信息, 平台通过该配置文件的内容来识别服务以及配置反向代理等功能. `windows` 平台和 `linux` 平台配置文件内容有所不同, 请根据自己的平台选择相应的配置文件.

linux 平台文件内容介绍如下

```yaml
# 必填项. 服务名称
Name: my-service
# 必填项. 如果驱动对外提供 rest 服务, 则需要填写 rest 接口的统一路径前缀.
# 当开发第二类服务时, 该配置项内容需要与 server.servlet.context-path 的值保持一致.
Path: /my-service
# 必填项. 服务的版本号. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 服务的描述信息
# 必填项. 固定为 server
GroupName: server
# 服务端口映射类型, 必填项.
# 可选项有 None Internal External
#
# None: 不暴露端口
# Internal: 只在平台内部暴露端口. 一般为驱动对外提供 rest 服务时, 将端口映射到网关上, 填写为
Internal 即可.
# External: 同时将端口映射到网关和宿主机上.
Service: Internal
# 非必填项. 暴露的端口列表. 对于第二类服务该配置项必填.
Ports:
  - Host: "8558"            # 映射到宿主机的端口号, 如果不填写, 则会随机分配一个端口号
    Container: "8558"       # 服务监听的端口号, 即配置文件中 server.port 的值
    Protocol: ""            # 协议类型, 可选项有 TCP UDP, 如果不填写, 则默认为 TCP
```

windows 平台文件内容介绍如下

```yaml
# 必填项. 服务名称
Name: my-service
# 必填项. 如果驱动对外提供 rest 服务, 则需要填写 rest 接口的统一路径前缀.
# 当开发第二类服务时, 该配置项内容需要与 server.servlet.context-path 的值保持一致.
Path: /my-service
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 服务的描述信息
# 服务配置文件名称. 平台安装服务时会查找并读取该文件. 一般填写 application.yml
ConfigType: application.yml
# 必填项. 固定为 server
GroupName: server
# 必填项. 启动命令. 还可以添加一些启动参数, 例如: -Xms512m -Xmx1024m
Command: java -jar my-service.jar
```

4. 打包

该步骤是将前面已经打包好的 `jar` 文件和 `service.yml` 文件打包为一个安装包, 以便平台安装服务时能够正确识别服务.

linux 平台打包

在 `linux` 平台上, 平台服务是以 `docker` 容器的形式运行的, 需要将服务程序打包为一个 `docker` 镜像. 然后再将镜像和`service.yml` 文件打包为压缩文件. 以下是简单的 `Dockerfile` 示例:

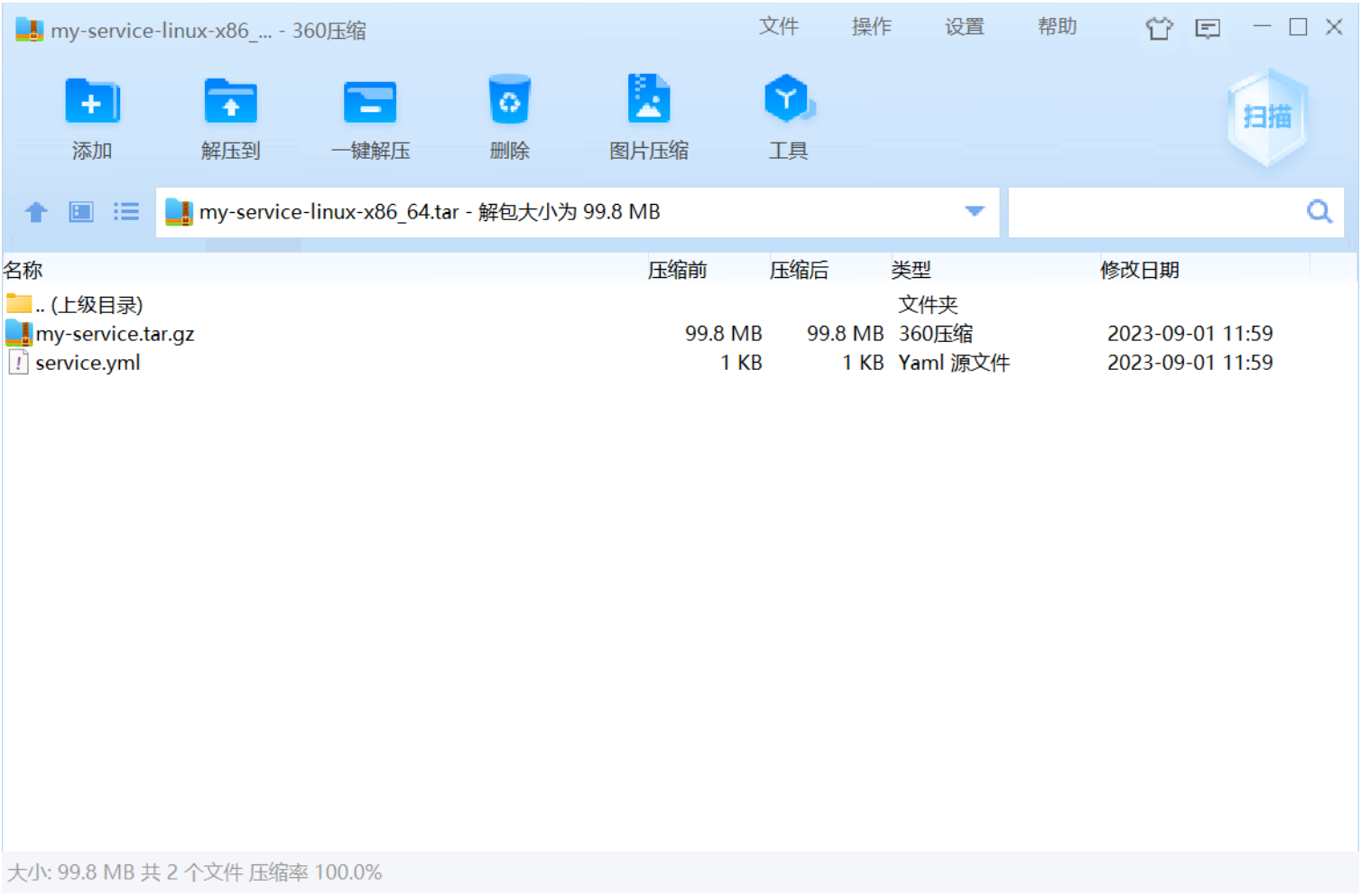
```
FROM openjdk:11.0.13-jre-slim-bullseye

WORKDIR /app

# 将之前打包好的 jar 文件复制到镜像中
COPY target/my-service-*.jar /app/my-service.jar

# 容器的启动命令
CMD ["java", "-jar", "my-service.jar"]
```

```
# 打包镜像
docker build -t my-service:v1.0.0 .
```

将构建好的镜像和 `service.yml` 文件打包为一个压缩文件即可. 例如:

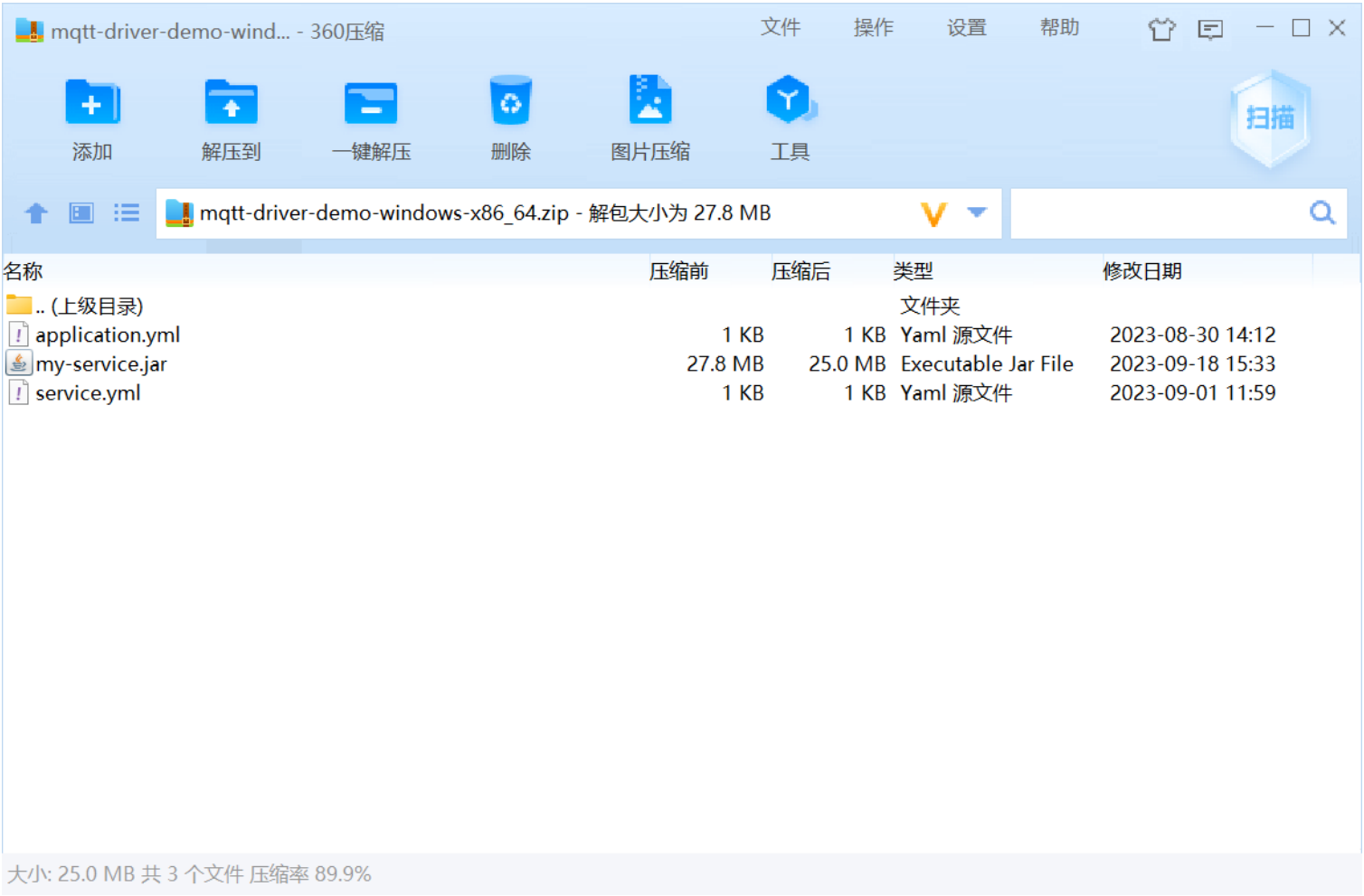
```
# 导出镜像并压缩
docker save my-service:v1.0.0 | gzip > my-service.tar.gz

# 将 service.yml 文件和镜像打包为一个压缩文件
tar czvf my-service-linux-x86-64.tar.gz service.yml my-service.tar.gz
```

最后得到的 `my-service-linux-x86-64.tar.gz` 文件即为平台可识别的安装包. 压缩包内部结构如下图所示:



windows 平台打包

在 `windows` 平台上, 平台服务是直接运行的, 需要将 `jar` 文件、`application.yml` 和 `service.yml` 文件打包为 `zip` 格式压缩文件. 压缩包内部结构如下图所示:



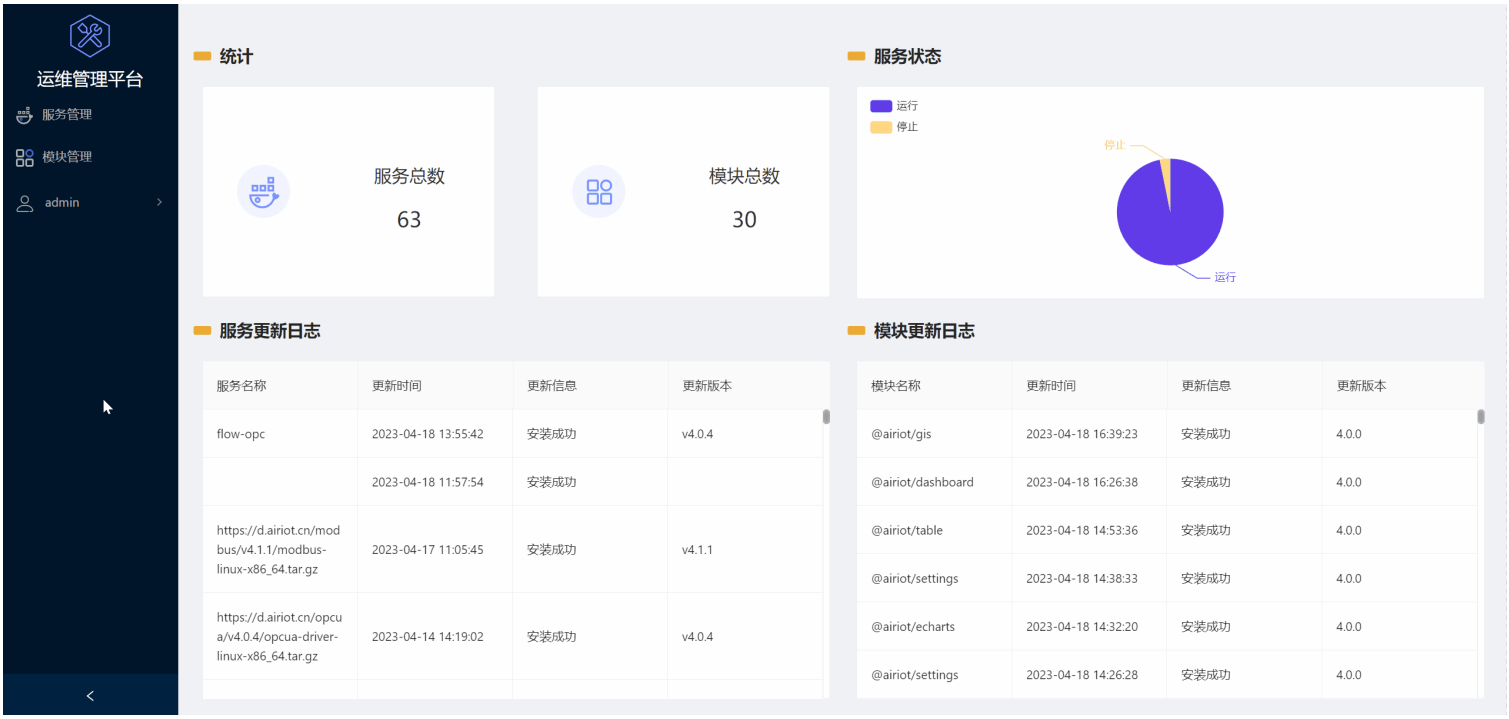
> **INFO**
>
> 平台中默认安装了 `jdk17` 的环境, 如果使用的版本高于该版本, 需要自己提供额外的 `jdk` 环境, 并且修改 `service.yml` 中的 `Command` 配置项.

5. 部署

该步骤介绍如何将上一步骤中打包好的安装包部署到平台中.

1. 打开平台运维管理系统并登录. 默认地址为: `http://ip:13030`.
2. 登录成功后, 点击左侧导航栏中的 `服务管理` 菜单, 进入服务管理页面.
3. 点击右上角的 `添加服务` 按扭, 然后选择 `离线添加` 标签页.
4. 点击 `上传` 按扭, 选择上一步骤中打包好的安装包, 然后点击 `提交` 按钮, 等待平台的安装过程完成即可.
5. 如果安装成功, 则可以在服务管理页面中看到刚刚安装的服务. 如果安装失败, 可以回到首页查看安装日志, 并根据日志信息进行排查.

安装过程序如下图所示:



4. 访问

当服务安装成功后, 如果是第二类服务, 那么该服务内的接口就可以通过平台网关进行访问了. 例如: 如果服务内有一个接口地址为: `GET /student/{id}`, 那么可以通过 `http://ip:31000/my-service/student/1` 来访问该接口.

```
# 使用 curl 访问
curl -X GET -H "Authorization:Bearer
eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2OTYyMjU5ADQsImlhdCI6MTY5NTAxNjM4NCwibmJmIjoxNj
Jo1KOOskbUhiBi4fS4riyEcSWZQMGuXGu8og" http://ip:31000/my-service/student/1
```

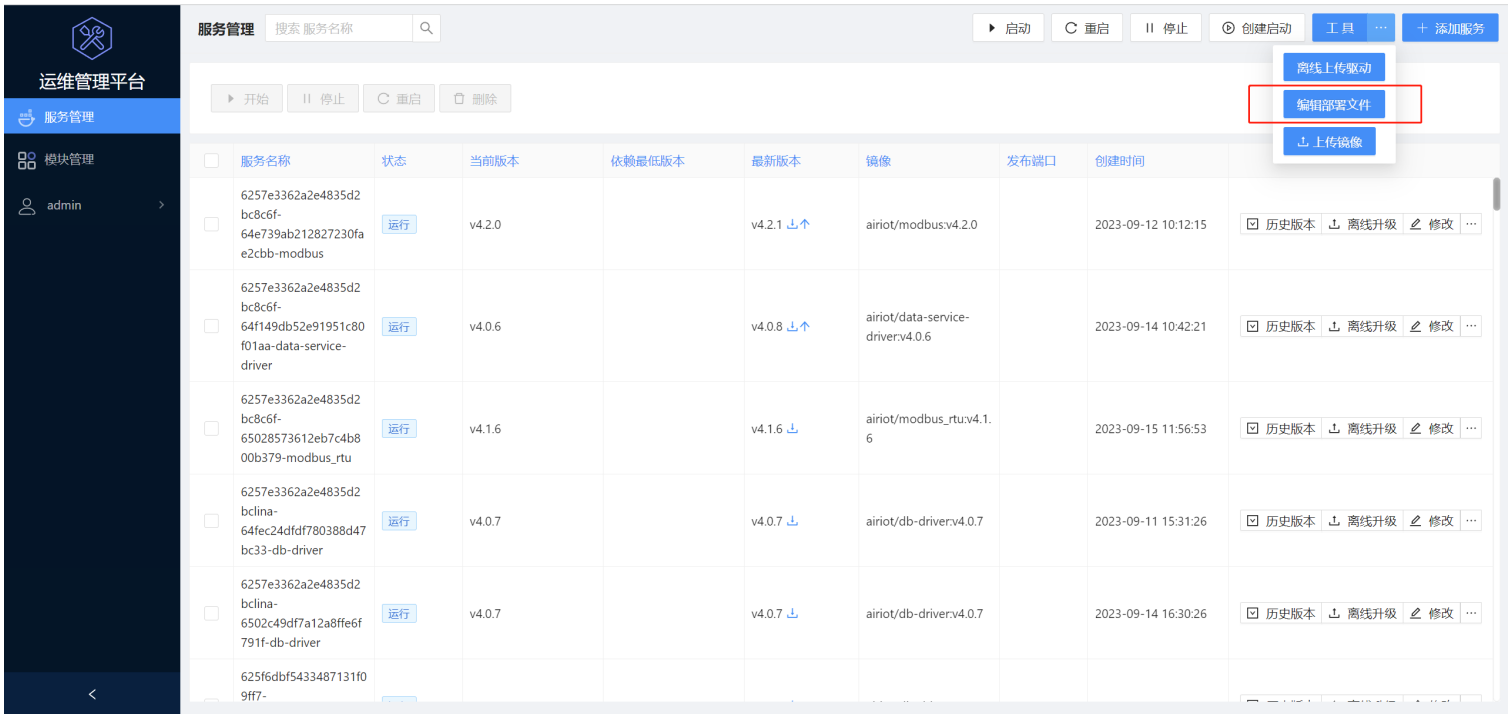
> **INFO**
>
> 注意事项:
>
> 1. 访问安装到的平台的服务时, 需要在请求头中携带有效的平台 `token`, 请求头为 `Authorization`. 如果想禁用到平台的认证功能, 需要在运维管理系统中, 修改该服务的配置并重启. 修改方法见 如何禁用服务认证功能.
> 2. 通过网关访问接口时, 必须添加统一的路径前缀, 即 `service.yml` 中的 `Path` 配置项.

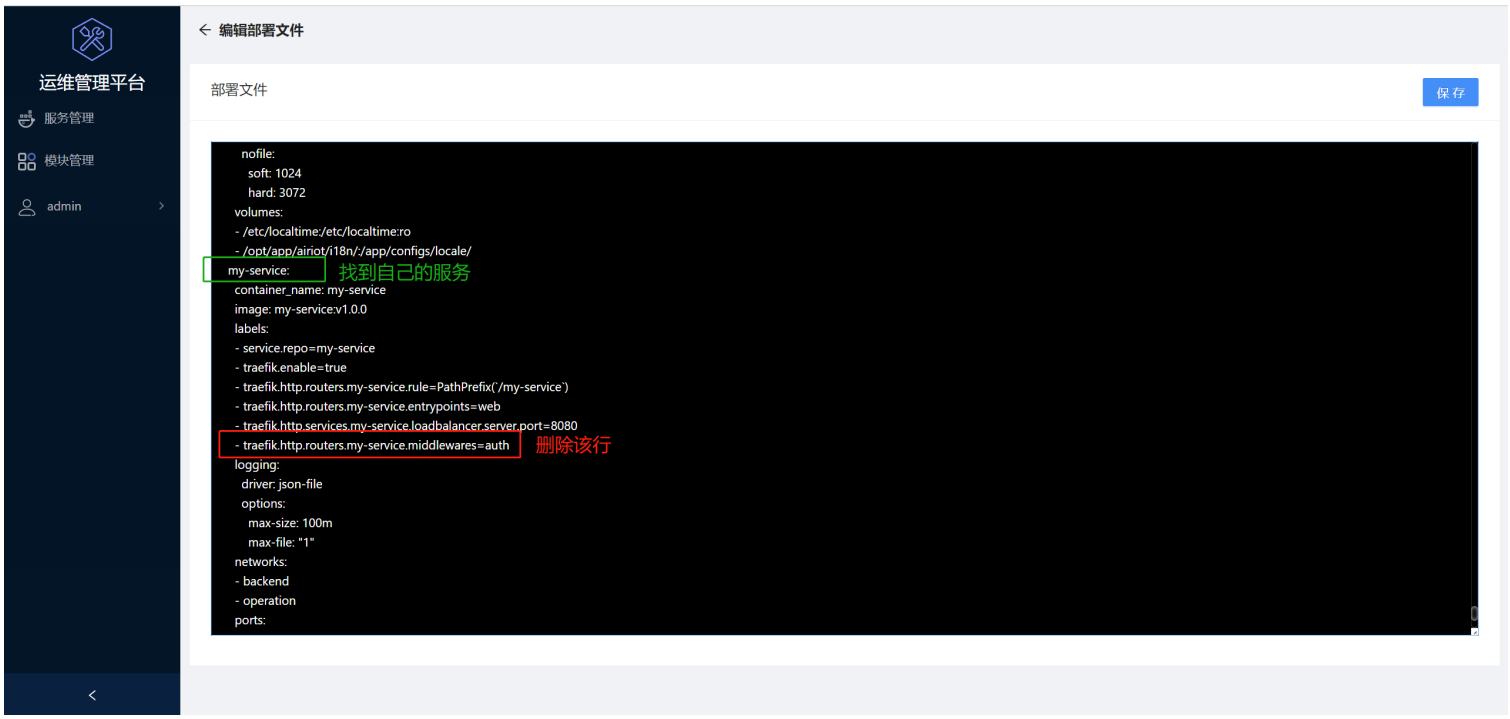
其它

如何禁用服务认证功能

平台网关代理后端服务时, 默认会开启统一的身份认证功能. 如果请求头中没有携带有效的 `token`, 则会返回 `401` 错误. 可按以下步骤进行操作, 然后重启服务即可.

1. 打开平台运维管理系统并登录. 默认地址为: `http://ip:13030`.
2. 登录成功后, 点击左侧导航栏中的 `服务管理` 菜单, 进入服务管理页面.
3. 点击右上角的 `工具 -> 编辑部署文件` 按扭.
4. 找到自己的服务.
5. 删除 `traefik.http.routers.my-service.middlewares=auth`.
6. 保存, 然后重启该服务即可.





Java SDK 介绍

Java SDK 二次开发说明

数据接入驱动开发

介绍如何使用 Java SDK 开发自定义数据接入驱动

流程插件开发

介绍如何使用 Java SDK 开发自定义流程插件

流程扩展节点开发

介绍如何使用 Java SDK 扩展流程节点

平台接口客户端

Java SDK 平台接口客户端

常见问题

Java SDK 使用过程中的常见问题

算法服务开发

介绍如何使用 Java SDK 将算法集成到平台

Java SDK 介绍

`Java SDK` 是 AIRIOT 物联网平台提供的 `Java` 语言的二次开发工具包, 可使用 Java SDK 调用平台开放的接口和实现对平台功能的扩展。

接下来会分别介绍如何使用 `Java SDK` 开发自定义数据接入驱动、开发自定义流程插件、算法集成和通过平台接口客户端实现系统集成。

内容说明

内容	用途
自定义数据接入驱动	实现从设备或其它平台系统采集数据功能
自定义流程插件	扩展流程引擎中的节点
算法集成	扩展或集成已有算法到平台
平台接口客户端	调用平台对外提供的数据接口
日志收集	使用 SDK 中的日志组件输出的日志, 可以被平台收集并在平台的 `系统日志` 模块中查看

SDK 内容列表

包名	描述
sdk-dependencies	统一依赖管理
sdk-driver-starter	提供自定义数据接入驱动开发的相关内容. 包括驱动的接口定义, 以及与平台交互功能
sdk-flow-plugin-starter	提供自定义流程引擎插件开发的相关内容. 包括与流程引擎交互的具体实现和数据交互格式的定义等

包名	描述
sdk-algorithm-starter	提供算法接入的相关内容. 可以扩展平台算法或将已经算法集成到平台
sdk-client-api	平台接口客户端定义. 包括: 空间管理、项目中的用户、部门、工作表、数据接口等内容的操作接口
sdk-client-http-starter	平台接口客户端 `http` 实现
sdk-logger-starter	平台系统日志组件库. 该组件库中提供了平台定义的日志输出格式, 由该组件库输出的日志可以被平台收集和并在 `系统日志` 模块中查看

> **INFO**
>
> `sdk-client-api` 包无须手动导入, 在引入平台客户端实现包(例如: `sdk-client-http-starter`) 时会自动导入.

使用方式

您可以通过 `Maven` 或 `Gradle` 管理配置 SDK 的版本，Maven 的配置示例如下:

```
<dependencies>
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-driver-starter</artifactId>
        <version>4.x.x</version>
    </dependency>
</dependencies>
```

在同时使用 SDK 中的多个模块时, 可以使用 `sdk-dependencies` 统一管理版本, 方式如下:

```
<dependencies>
    <!-- 数据接入驱动开发 SDK -->
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-driver-starter</artifactId>
```

```
    </dependency>
    <!-- 流程插件开发 SDK -->
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-flow-plugin-starter</artifactId>
    </dependency>
    <!-- 流程扩展节点开发 SDK -->
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-flow-extension-starter</artifactId>
    </dependency>
    <!-- 算法集成 SDK -->
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-algorithm-starter</artifactId>
    </dependency>
    <!-- 平台接口 http 客户端 -->
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-client-http-starter</artifactId>
    </dependency>
    <!-- 系统日志组件库 -->
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-logger-starter</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <!-- 统一依赖版本管理 -->
        <dependency>
            <groupId>io.github.air-iot</groupId>
            <artifactId>sdk-dependencies</artifactId>
            <version>4.x.x</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

INFO

注: `SNAPSHOT` 版本的 SDK 为开发版本, 可能会带有一些不稳定的新功能, 请谨慎使用.

为了确保依赖版本的一致性, 建议使用 `sdk-dependencies` 统一管理版本.

版本说明

SDK 版本	Java 版本	SpringBoot 版本
4.1.x	1.8+	2.7.8

数据接入驱动开发

本文将会详细介绍如何使用 `Java SDK` 开发自定义数据接入驱动. 示例项目目上传至 https://github.com/air-iot/sdk-java-examples/tree/master/mqtt-driver-demo.

介绍

`数据接入驱动` 是为实现从不同的协议、设备或其它平台系统采集数据而开发的特定程序. 每个 `数据接入驱动` 程序需要根据协议的特点实现数据采集功能，然后将采集到的数据通过 `Java SDK` 提供的接口发送到平台.

`Java SDK` 提供了 `数据接入驱动` 开发的相关内容. 包括驱动的接口定义, 以及与平台交互功能. 开发者只需要在 `pom.xml` 中引入 `sdk-driver-starter` 依赖, 定义配置信息(schema及配置类)，然后实现 `DriverApp` 接口中的方法即可.

开发步骤

1. 创建项目

由于 `Java SDK` 使用了 `SpringBoot` 框架, 所以需要创建一个 `SpringBoot` 项目. 使用的 `Java` 和 `SpringBoot` 版本见 版本说明.

> **INFO**
>
> 可以使用 `idea` 的 `spring initializr` 功能创建项目. 也可以在 https://start.spring.io 或 https://start.aliyun.com 网站上在线创建项目.

2. 引入SDK

以 `maven` 为例, 在 `pom.xml` 中引入 `sdk-driver-starter` 依赖.

```
<dependencies>
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-driver-starter</artifactId>
        <version>4.x.x</version>
    </dependency>
</dependencies>
```

> **INFO**
>
> 如果 `数据接入驱动` 需要调用平台的接口, 可以另外引入 `sdk-client-http-starter` 依赖.

3. 定义schema

`数据接入驱动` 需要定义一个 `schema` 用于描述驱动的配置信息. `schema` 是一个类似于 `json` 格式的对象, 详细格式说明见 数据接入驱动schema说明. 以下是一个简单的示例:

```
({
    "driver": {
        "properties": {
            "settings": {
                "title": "模型配置",
                "type": "object",
                "properties": {
                    "server": {
                        "type": "string",
                        "title": "MQTT Broker",
                        "descripption": "MQTT 服务器地址. 例如: tcp://127.0.0.1:1883"
                    },
                    "username": {
                        "type": "string",
                        "title": "用户名",
                    },
                    "password": {
                        "type": "string",
                        "title": "密码",
                        "fieldType": "password"
                    },
                    "network": {
                        "type": "object",
                        "title": "通讯监控参数",
                        "properties": {
                            "timeout": {
                                "title": "通讯超时时间(s)",
                                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故
障",
                                "type": "number"
                            }
                        },
                        "form": [{
                            "key": "timeout",
                        }]
                    }
                },
                "required": ["server", "username", "password"]
```

```
            },
            "tags": {
                "title": "数据点",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "key": {
                            "type": "string",
                            "title": "Key",
                            "description": "数据点在 JSON 对象中的 Key. 例如: 'temperature'",
                        },
                    },
                    "required": ["key"]
                }
            },
            "commands": {
                "title": "命令",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "title": "名称"
                        },
                        "ops": {
                            "type": "array",
                            "title": "指令",
                            "items": {
                                "type": "object",
                                "properties": {
                                    "name": {
                                        "type": "string",
                                        "title": "主题",
                                        "description": "发送消息的主题. 例如: /cmd/control",
                                    },
                                    "message": {
                                        "type": "string",
                                        "title": "消息",
                                        "description": "发送的消息. 例如:
{\"cmd\":\"start\"}",
                                    },
                                    "qos": {
                                        "type": "number",
                                        "title": "QoS",
                                        "description": "消息质量. 0,1,2",
                                        "enum": ["0", "1", "2"],
                                        "enum_title": ["QoS0", "QoS1", "QoS2"]
                                    },
```

```
                                },
                                "required": ["name", "message"]
                            }
                        }
                    }
                }
            }
        },
        "model": {
            "properties": {
                "settings": {
                    "title": "模型配置",
                    "type": "object",
                    "properties": {
                        "topic": {
                            "type": "string",
                            "title": "主题",
                            "descripption": "接收数据的主题. 例如: /data/#"
                        },
                        "parseScript": {
                            "type": "string",
                            "title": "数据处理脚本",
                            "fieldType": "deviceScriptEdit",
                            "description": "消息处理脚本. 函数名必须为 'handler'",
                            "defaultScript": "/**\n" +
                                " * 数据处理脚本, 处理从 mqtt 接收到的数据.\n" +
                                " *\n" +
                                " * @param {string} topic 消息主题\n" +
                                " * @param {string} message 消息内容\n" +
                                " * @return 消息解析结果\n" +
                                " */\n" +
                                "function handler(topic, message) {\n" +
                                "\t\n" +
                                "\t// 脚本返回值必须为对象数组\n" +
                                "\t// \tid: 资产编号\n" +
                                "\t//\ttime: 时间戳(毫秒)\n" +
                                "\t//  fields: 数据点数据. 该字段为 JSON 对象, key 为数据点标识,
value 为数据点的值\n" +
                                "\treturn [\n" +
                                "\t\t{\"id\": \"SN10001\", \"time\": new Date().getTime(),
\"fields\": {\"key1\": \"this is a string value\", \"key2\": true, \"key3\": 123.456}}\n" +
                                "\t];\n" +
                                "}"
                        },
                        "commandScript": {
                            "type": "string",
                            "title": "指令处理脚本",
                            "fieldType": "deviceScriptEdit",
                            "description": "指令处理脚本. 函数名必须为 'handler'",
```

```
                        "defaultScript": "/**\n" +
                            " * 指令处理脚本. 发送指令时会将指令内容传递给脚本，然后由指定返回最终
要发送的信息.\n" +
                            " *\n" +
                            " * @param {string} 工作表标识\n" +
                            " * @param {string} 资产编号\n" +
                            " * @param {object} 命令内容\n" +
                            " * @return {object} 最终要发送的消息，及目标 topic\n" +
                            " */\n" +
                            "function handler(tableId, deviceId, command) {\n" +
                            "\t\n" +
                            "\t// 脚本返回值必须为下面对象结构\n" +
                            "\t//\t\ttopic: 消息发送的目标 topic\n" +
                            "\t//\t\tpayload: 消息内容\n" +
                            "\treturn {\n" +
                            "\t\t\"topic\": \"cmd/\" + deviceId,\n" +
                            "\t\t\"payload\": \"发送内容\"\n" +
                            "\t};\n" +
                            "}"
                    },
                    "network": {
                        "type": "object",
                        "title": "通讯监控参数",
                        "properties": {
                            "timeout": {
                                "title": "通讯超时时间(s)",
                                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故
障",
                                "type": "number"
                            }
                        },
                        "form": [{
                            "key": "timeout",
                        }]
                    }
                },
                "required": ["server", "username", "password", "topic"]
            },
            "tags": {
                "title": "数据点",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "key": {
                            "type": "string",
                            "title": "Key",
                            "description": "数据点在 JSON 对象中的 Key. 例如: 'temperature'",
                        },
                    },
```

```
                    "required": ["key"]
                }
            },
            "commands": {
                "title": "命令",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "title": "名称"
                        },
                        "ops": {
                            "type": "array",
                            "title": "指令",
                            "items": {
                                "type": "object",
                                "properties": {
                                    "name": {
                                        "type": "string",
                                        "title": "主题",
                                        "description": "发送消息的主题. 例如: /cmd/control",
                                    },
                                    "message": {
                                        "type": "string",
                                        "title": "消息",
                                        "description": "发送的消息. 例如:
{\"cmd\":\"start\"}",
                                    },
                                    "qos": {
                                        "type": "number",
                                        "title": "QoS",
                                        "description": "消息质量. 0,1,2",
                                        "enum": ["0", "1", "2"],
                                        "enum_title": ["QoS0", "QoS1", "QoS2"]
                                    },
                                },
                                "required": ["name", "message"]
                            }
                        }
                    }
                }
            }
        },
        "device": {
            "properties": {
                "settings": {
                    "title": "设备配置",
```

```
                "type": "object",
                "properties": {
                    "customDeviceId": {
                        "type": "string",
                        "title": "设备编号",
                        "descripption": "自定义设备编号. 如果未定义则使用平台中的资产编号"
                    },
                    "network": {
                        "type": "object",
                        "title": "通讯监控参数",
                        "properties": {
                            "timeout": {
                                "title": "通讯超时时间(s)",
                                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故
障",
                                "type": "number"
                            }
                        }
                    }
                },
                "required": []
            }
        }
    }
})
```

4. 根据schema创建配置类

在上一步骤中, 通过 `schema` 定义了驱动的相关配置, 包括 `驱动配置`、`数据点配置` 和 `指令配置`. 接下来需要根据 `schema` 创建相应的配置类.

整体格式说明

驱动从平台接收到的配置信息整体格式如下:

```
{
    "id": "mqtt-demo",
    "name": "MQTT示例驱动",
    "driverType": "mqtt-demo",
    "device": {
        "settings": {
            "server": "tcp://127.0.0.1:1883",
            "username": "admin",
            "password": "public",
        },
        "tags": [
            {
```

```
                "id": "temperature",
                "name": "温度",
                "unit": "℃",
                "key": "temperature"
            }
        ],
        "commands": [
            {
                "name": "启动",
                "ops": [
                    {
                        "name": "启动",
                        "message": "{\"cmd\":\"start\"}",
                        "qos": 0
                    }
                ]
            }
        ]
    },
    "tables": [{
        "id": "mqtt-devices",
        "device": {
            "settings": {
                "topic": "/data/#",
                "parseScript": "/**\n * 数据处理脚本，处理从 mqtt 接收到的数据.\n *\n * @param {string} topic 消息主题\n * @param {string} message 消息内容\n * @return 消息解析结果\n */\nfunction handler(topic, message) {\n    // 发送的 topic 为 data/javasdk/javasdk001\n    // 其中 javasdk001 为设备编号\n    var fields = topic.split(\"/\");\n\n    // 脚本返回值必须为对象数组\n    // \tid: 资产编号\n    //\tttime: 时间戳(毫秒)\n    //  fields: 数据点数据. 该字段为 JSON 对象，key 为数据点标识，value 为数据点的值\n    return [\n        {\"id\": fields[2], \"time\": new Date().getTime(), \"fields\": JSON.parse(message)}\n    ];\n}",
                "commandScript": "/**\n * 指令处理脚本. 发送指令时会将指令内容传递给脚本，然后由指定返回最终要发送的信息.\n *\n * @param {string} 工作表标识\n * @param {string} 资产编号\n * @param {object} 命令内容\n * @return {object} 最终要发送的消息，及目标 topic\n */\nfunction handler(tableId, deviceId, command) {\n    var op = command.ops[0];\n    var params = command.params;\n\n    // 脚本返回值必须为下面对象结构\n    //\t\ttopic: 消息发送的目标 topic\n    //\t\tpayload: 消息内容\n    return {\n        \"topic\": op.topic + deviceId,\n        \"payload\": JSON.stringify({\"cmd\": params[command.name]})\n    };\n}",
            },
            "tags": [],
            "commands": []
        },
        "devices": [{
            "id": "sensor001",
            "name": "",
            "device": {
                "settings": {
                    "customDeviceId": "sn1002003"
                },
                "tags": [],
```

```
                "commands": []
            }
        }]
    }]
}
```

上述格式中的 `device`、`tables.device` 和 `tables.devices.device` 分别为 `驱动实例配置`、`模型配置(工作表的设备配置)` 和 `资产配置(设备的设备配置)`.

其中 `settings` 为 `驱动配置信息`, 与 `schema` 中的 `settings` 对应. `tags` 为 `数据点配置信息`, 与 `schema` 中的 `tags` 对应. `commands` 为 `指令配置信息`, 与 `schema` 中的 `commands` 对应.

> **INFO**
>
> `驱动实例`、`模型` 和 `资产` 中的 `settings` 可以不相同, 但 `tags` 和 `commands` 必须相同. 例如, 可以把统一的配置信息放在 `驱动实例` 中, 把不同的配置信息放在 `模型` 或 `资产` 中.

配置类定义

根据前面 `schema` 的定义, `settings`、`tags` 和 `commands` 对应的配置类的基本定义. 示例如下:

```
/**
 * 驱动配置信息, 用来承载 schema 中的 settings 配置信息.
 * <br>
 * 该示例中, 将驱动实例、工作表和设备的驱动实例合并到一个配置类中了, 也可以分开定义.
 */
public class Settings {
    private String server;
    private String username;
    private String password;
    private String topic;
    private String parseScript;
    private String commandScript;
    private String customDeviceId;
    // get and set
}

/**
 * 数据点配置信息. 除了 schema 中的 tags 配置信息外, 平台也在数据点中自定义了一些信息, 如: id、name、 有效范围、数值转换等,
 * 这部分信息会被 SDK 和平台使用, 平台自有的信息已经定义在 SDK 中的 {@link io.github.airiot.sdk.driver.model.Tag} 类中,
 * 所以这里只需要定义 schema 中 tags 的配置信息然后继承平台的 Tag 类即可.
 */
```

```java
public class Tag extends io.github.airiot.sdk.driver.model.Tag {

    private String key;

    // get and set
}

/**
 * 指令信息.
 *
 * 其中: id, name 是平台自带信息, 必须添加到指令配置类中.
 */
public class Command {
    /**
     * 指令ID
     */
    private String id;
    /**
     * 指令名称
     */
    private String name;
    /**
     * 驱动指令配置
     * <br>
     * 该字段固定为数组, 但目前只有一个元素. {@code Op} 为指令的配置, 与 scheama 中的 commands 配置对应.
     * 所以, 需要创建一个 Op 类用于接收指令的配置信息
     */
    private List<Op> ops;
    /**
     * 数据写入
     * <br>
     * 该信息在数据点配置页面中的 '数据写入' 选项中配置, 类型固定为 Map<String, Object>, 用于接收通过 '数据写入' 方式填写的数据.
     */
    private Map<String, Object> params;

    // getter and setter
}

/**
 * 指令的配置信息, 与 scheama 中的 commands 配置对应.
 */
public class Op {
    private String topic;
    private String message;
    private String qos;

    // getter and setter
}
```

```java
/**
 * 最终完整的配置类. 该类中包含了驱动配置信息、数据点配置信息和指令配置信息.
 */
public class DriverConfig {
    /**
     * 驱动配置信息
     */
    private Settings settings;
    /**
     * 数据点列表
     */
    private List<Tag> tags;
    /**
     * 指令列表
     */
    private List<Command> commands;

    // getter and setter
}
```

> **INFO**
>
> 1. 配置类中的所有字段都必须提供 `getter` 和 `setter` 方法, 否则无法正常解析配置信息.
> 2. 自定义数据点配置类必须继承 `SDK` 中的 `io.github.airiot.sdk.driver.model.Tag` 类, 否则会导致报错.

5. 实现驱动接口

数据接入驱动二次开发SDK中, 有两个重要接口: `数据接入驱动接口` 和 `驱动与平台交互接口`, 分别为 **DriverApp** 和 **DataSender**.

- **数据接入驱动接口(DriverApp)** 该接口中定义平台控制驱动程序运行的相关方法, 是平台控制驱动程序的入口. `SDK` 在驱动程序启动后, 会监听平台下发的控制指令, 然后调用 `数据接入驱动接口` 中的方法并将执行结果返回给平台. 开发者需要实现这个接口, 并且将实现类注入到 `spring` 容器中. 接口定义及详细说明见 数据接入驱动接口说明.

- **驱动与平台交互接口(DataSender)** 是驱动程序与平台进行交互的接口. 驱动程序在运行过程中, 除了接收平台的控制之外, 也需要与平台进行各种交互. 例如: 向平台发送采集的到数据, 向平台发送指令执行结果、调试日志等. `SDK` 中定义了 `驱动与平台交互接口` **DataSender** 以及实现类, 并且实现类已经注入到

`spring` 容器中. 在驱动中可直接注入 `驱动与平台交互接口` 即可. 接口定义及详细说明见 驱动与平台交互接口说明.

> **INFO**
>
> 每个驱动程序必须向 `spring` 容器中注入一个 `数据接入驱动接口` 的实现类, 并且只能注入一个. 在程序启动时, `SDK` 会在 `spring` 容器中查找 `数据接入驱动接口` 的实现类. 如果没有找到或找到多个实现类时, 会抛出异常.
>
> **注: 向平台上报采集到的数据时, 必须通过 `驱动与平台交互接口` 中的 `writePoint` 方法发送, 不能直接调用 `MQTT` 客户端发送. 因为 `SDK` 会对发送的数据进行一些处理, 包括有效范围处理、数值映射、缩放比例、小数位等处理.**

6. 配置驱动

这里所说 `驱动配置` 主要是一些静态配置信息（**与 schema 中定义的配置无关**）, 其中包括 `平台配置信息`, `驱动配置信息` 和 `自定义配置信息`. 这些信息一般通过配置文件(application.yml)、环境变量、命令行参数等方式传入. 其中一些配置信息由平台启动驱动时通过命令行参数传入.

这些配置信息, 在开发过程中可以根据实际情况进行调整. 但是在打包时必须按照平台的要求进行配置. 打包时的配置信息见 驱动配置说明.

> **INFO**
>
> `自定义配置信息` 是指驱动本身的一些配置信息, 对驱动使用者不可见. 例如: 连接池大小. 这些信息一般通过配置文件(application.yml)、环境变量、命令行参数等方式传入.

平台配置信息

`平台配置信息` 主要为平台的连接信息. 包括: `MQTT` 连接信息, `驱动管理服务` 连接信息. 内容如下:

```yaml
# 驱动管理服务配置信息
driver-grpc:
  # 在开发时, 需要配置为平台的地址
  host: 192.168.11.101
  # 端口默认为 9224, 一般无须修改. 如果有修改, 可在运维管理系统中查看 `driver` 服务的端口号
  port: 9224
# 平台 MQTT 配置信息
mq:
  mqtt:
    # 在开发时, 需要配置为平台的地址
```

```
    host: 192.168.11.101
    port: 1883
    username: admin
    password: public
```

> **INFO**
>
> 相关服务的端口号可在运维管理系统中查看. 平台中的 `driver` 服务的端口对应 `driver-grpc.port` 配置项, `mqtt` 服务的端口对应 `mq.mqtt.port` 配置项.

驱动配置信息

`驱动配置信息` 主要包括 `驱动ID`, `驱动名称`, `驱动实例ID`, `所属项目ID`.

- `驱动ID` 为驱动的唯一标识, 必须在平台中唯一. 需要和 `service.yml` 文件中的 `Name` 字段的值保持一致.
- `驱动名称` 为该驱动在平台中的显示名称.
- `驱动实例ID` 为该驱动实例的唯一标识. 同一个驱动可以创建多个实例, 每个实例的 `驱动ID` 相同但 `驱动实例ID` 唯一. 该信息由平台在 `驱动管理` 中创建驱动实例时生成.
- `所属项目ID` 每个驱动实例都属于一个项目, 该驱动实例只会拿到该项目中的模型和设备信息.

```
airiot:
  driver:
    id: 驱动ID
    name: 驱动名称
    instance-id: 驱动实例ID
    project-id: 项目ID
```

> **INFO**
>
> 1. `驱动ID` 和 `驱动名称` 需要在 `application.yml` 中手动定义.
> 2. `驱动实例ID` 和 `所属项目ID` 在开发过程中, 需要将这些信息手动配置到 `application.yml` 中. 在打包时无须定义, 在平台中安装驱动时这些信息会由平台通过命令行参数传入.

7. 打包

驱动打包就是将开发完成的程序打包为可以在平台部署的驱动. 平台自身支持运行在 `windows`、`linux` 和 `macOS` 系统中, 并且支持 `x86` 和 `arm` 平台. 在 `windows` 系统中平台服务和驱动程序都是直接运行在操作系统

中, 而在 `linux` 系统中是以 `容器` 的方式运行, 平台中的每个服务和驱动程序都是一个独立的容器, 所以针对不同的操作系统打包方式也不相同. 下面分别介绍在 `windows` 和 `linux` 系统中如何打包驱动， 对于不同平台只需要保证使用软件和库支持即可.

windows系统打包

在 `windows` 系统中, 驱动程序是直接运行在操作系统中, 所以需要将驱动程序打包为 `jar` 文件. **打包时需要注意的是, 不要将 `application.yml` 文件打包在 `jar` 文件内, 因为平台在安装驱动时可能会 `application.yml` 文件进行修改, 如果打包在 `jar` 内就无法修改可能会导致一些问题**. 具体打包步骤如下:

1. 打程序程序和相关资源打包为 `jar` 文件. 由于使用 `springboot` 框架开发, 所以可以直接使用 `springboot` 提供的插件进行打包. 在 `pom.xml` 中配置好相应插件后, 执行下面的命令进行打包:

```
mvn clean package
```

2. 准备驱动配置文件 `application.yml`. 可以将 `application.yml` 放在与 `jar` 文件相同目录或 `config` 目录下, 这样驱动在启动时会自动加载该配置文件.

> **INFO**
>
> 注: `application.yml` 中需要填写好 `驱动ID` 和 `驱动名称` 两个配置项.

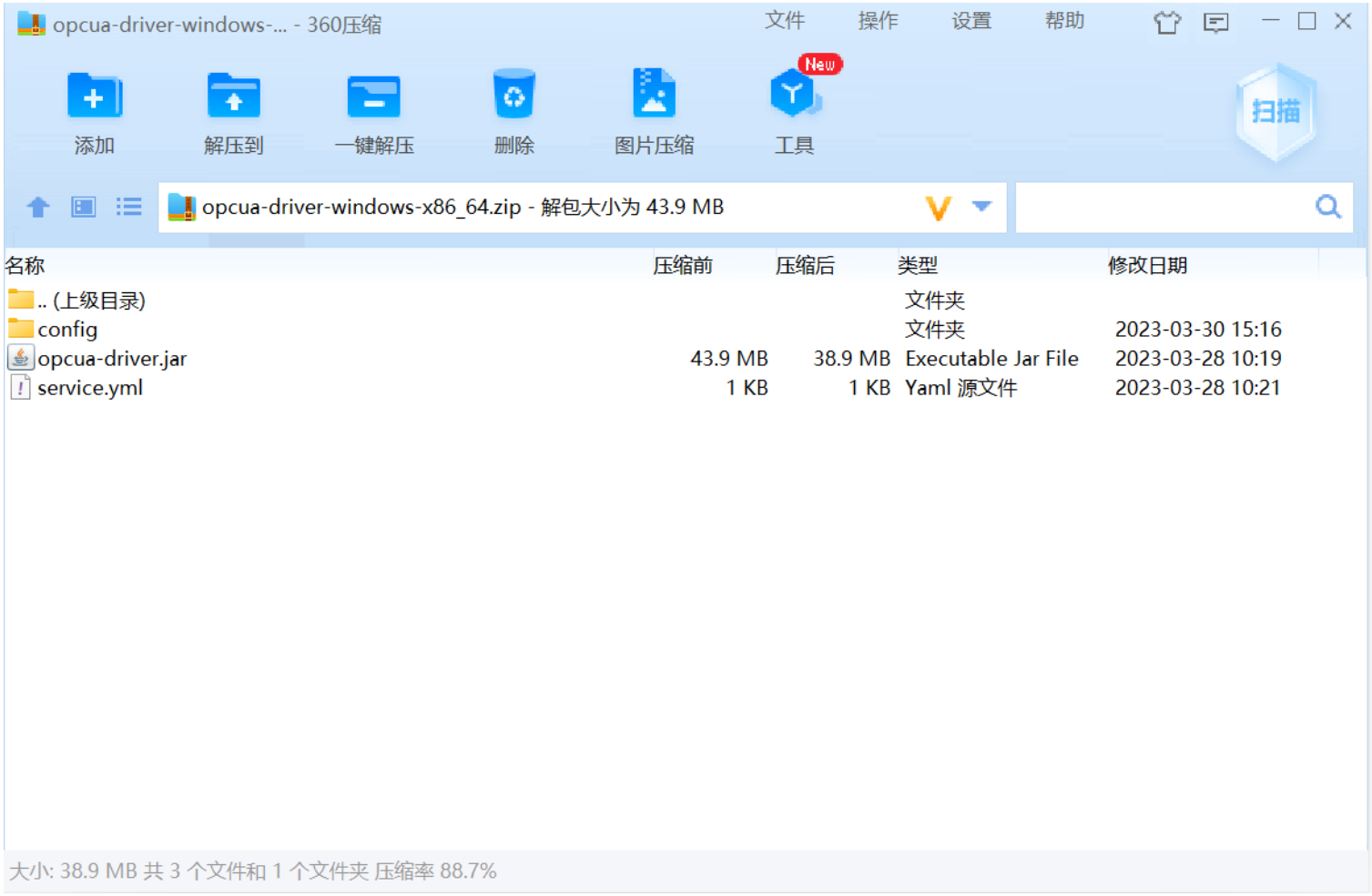
3. 准备驱动安装配置文件 `service.yml` . 在平台中安装驱动时, 需要提供一些驱动的基本信息, 例如: 版本号、驱动描述、端口号等. 这些信息需要在 `service.yml` 中定义, 平台会根据该文件中的配置信息进行安装. `service.yml` 的具体格式如下:

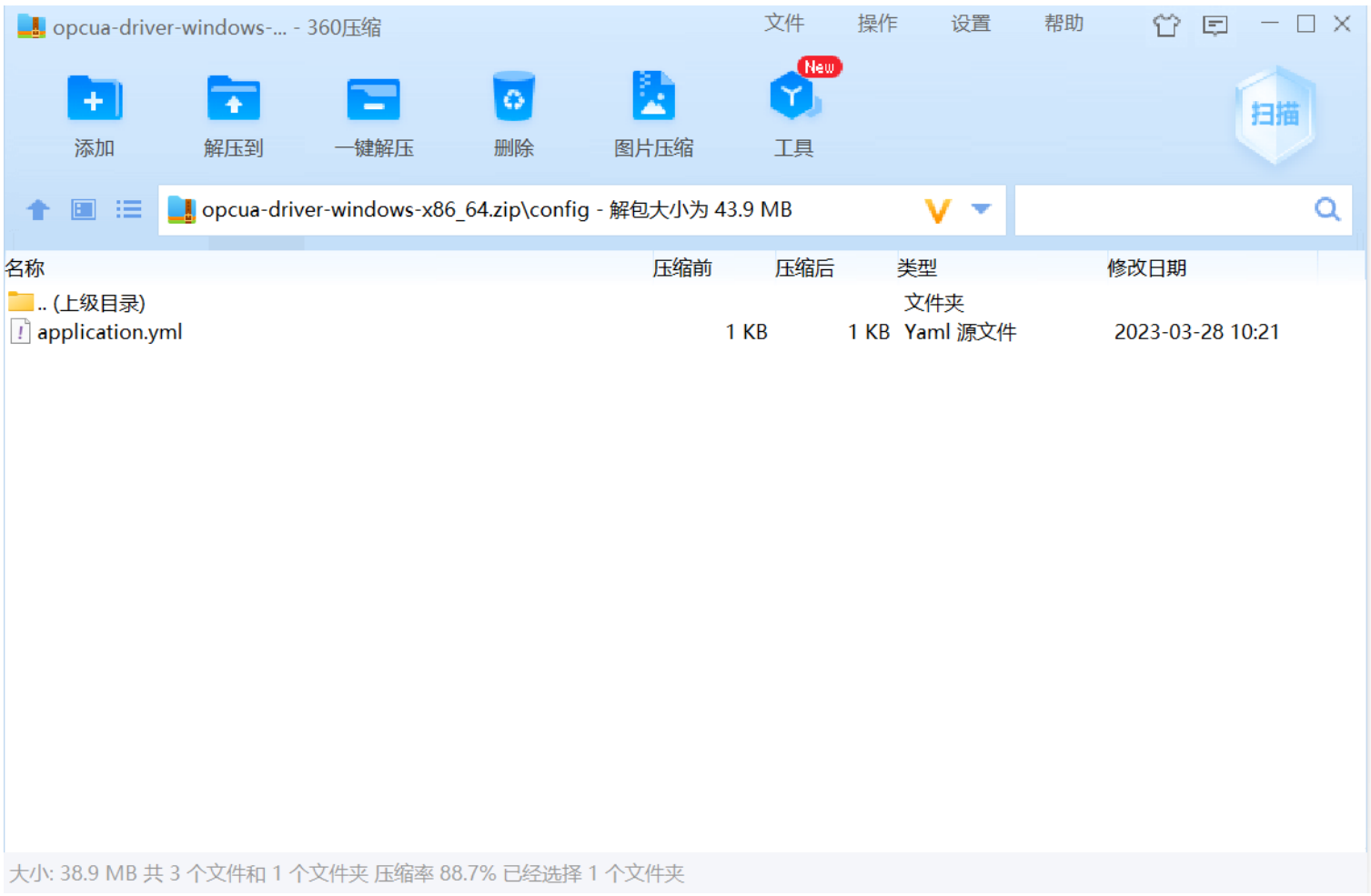
```
# 必填项. 驱动名称
Name: myDriver
# 非必填项. 如果驱动对外提供 rest 服务, 则需要填写 rest 接口的统一路径前缀.
# 当填写该配置项时, 平台会自动在网关中添加该路径的路由, 并将请求转发到该驱动, 代理端口为
application.yml 文件中的 server.port 配置项.
Path: /myDriver
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 驱动描述信息
# 驱动的配置文件名称, 平台在创建驱动时会查找驱动打包文件中查找该文件. 一般固定填写 application.yml
ConfigType: application.yml
# 必填项. 固定为 driver
GroupName: driver
# 必填项. 驱动启动命令. 还可以添加一些启动参数, 例如: -Xms512m -Xmx1024m
Command: java -jar myDriver.jar
```

> **INFO**
>
> 注: `service.yml` 中的 `Name` 字段的值必须与驱动配置文件中的 `airiot.driver.id` 的值保持一致.

4. 将所有资源打包为 `zip` 文件.

将 `jar` 文件、`application.yml`、`service.yml` 和其它资源打包为 `zip` 文件, 平台会根据该文件进行安装. 建议打包后的 `zip` 文件结构如下:


opcua-driver-windows-... - 360压缩
文件
操作
设置
帮助
添加
解压到
一键解压
删除
图片压缩
工具
New
扫描
opcua-driver-windows-x86_64.zip - 解包大小为 43.9 MB
名称
压缩前
压缩后
类型
修改日期
.. (上级目录)
文件夹
config
文件夹
2023-03-30 15:16
opcua-driver.jar
43.9 MB
38.9 MB
Executable Jar File
2023-03-28 10:19
service.yml
1 KB
1 KB
Yaml 源文件
2023-03-28 10:21
大小: 38.9 MB 共 3 个文件和 1 个文件夹 压缩率 88.7%


opcua-driver-windows-... - 360压缩
文件
操作
设置
帮助
添加
解压到
一键解压
删除
图片压缩
工具
New
扫描
opcua-driver-windows-x86_64.zip\config - 解包大小为 43.9 MB
名称
压缩前
压缩后
类型
修改日期
.. (上级目录)
文件夹
application.yml
1 KB
1 KB
Yaml 源文件
2023-03-28 10:21
大小: 38.9 MB 共 3 个文件和 1 个文件夹 压缩率 88.7% 已经选择 1 个文件夹

linux系统打包

由于在 `linux` 系统中, 驱动程序是以 `容器` 的方式运行, 所以打包时需要先将驱动程序打包为 `docker` 镜像. 然后再将镜像文件和 `service.yml` 打包为 `.tar.gz` 压缩包. 具体打包步骤如下:

1. 打包程序程序打包为 `jar` 文件. 具体打包步骤参考 windows系统打包 中的第一步. 需要注意的是, 在 `linux` 系统打包中, 不要求将驱动配置文件 `application.yml` 放在在 `jar` 文件外面.
2. 准备 `Dockerfile` 文件. 以下是一个简单的 `Dockerfile` 文件示例, 具体内容根据自身的需求进行修改:

```
# 根据自身的需求选择合适的基础镜像
FROM openjdk:11.0.13-jre-slim-bullseye
WORKDIR /app
# 如果驱动配置文件在 jar 文件外面
COPY application.yml /app/config/
# 复制 jar 文件
COPY myDriver-*.jar /app/myDriver.jar
# 启动命令，根据自身的需求添加启动参数
ENTRYPOINT ["java", "-Xms512m", "-Xmx512m", "-jar", "myDriver.jar"]
```

3. 构建 `docker` 镜像.

使用上一步中的 `Dockerfile` 文件构建 `docker` 镜像, 具体命令如下:

```
docker build -t myDriver:1.0.0 .
```

4. 导出 `docker` 镜像并压缩.

```
docker save myDriver:1.0.0 | gzip > myDriver.tar.gz
```

5. 准备驱动安装配置文件 `service.yml` . 该文件的格式与 windows系统打包 中的第三步中的 `service.yml` 文件格式相似但又有区别. 具体格式如下:

```
# 必填项. 驱动名称
Name: myDriver
# 非必填项. 如果驱动对外提供 rest 服务，则需要填写 rest 接口的统一路径前缀.
# 当填写该配置项时，平台会自动在网关中添加该路径的路由，并将请求转发到该驱动，代理端口为
application.yml 文件中的 server.port 配置项.
Path: /myDriver
# 必填项. 例如: 1.0.0, 通常用镜像版本号一致
Version: 1.0.0
# 非必填项.
```

```
Description: 驱动描述信息
# 必填项. 固定为 driver
GroupName: driver
# 容器端口映射类型, 非必填项. 如果驱动需要对外提供 rest 服务, 或暴露端口时, 需要填写该配置项.
# 可选项有 None Internal External
#
# None: 不暴露端口
# Internal: 只在平台内部暴露端口. 一般为驱动对外提供 rest 服务时, 将端口映射到网关上, 填写为
Internal 即可.
# External: 对外暴露端口. 一般为驱动为作 server 端, 需要对外暴露端口以供设备连接, 此时该端口会暴露
在宿主机上, 填写为 External 即可.
Service: Internal
# 非必填项. 暴露的端口列表
Ports:
  - Host: "8558"            # 映射到宿主机的端口号, 如果不填写, 则会随机分配一个端口号
    Container: "8558"       # 容器内部的端口号, 即驱动服务监听的端口号
    Protocol: ""            # 协议类型, 可选项有 TCP UDP, 如果不填写, 则默认为 TCP
```

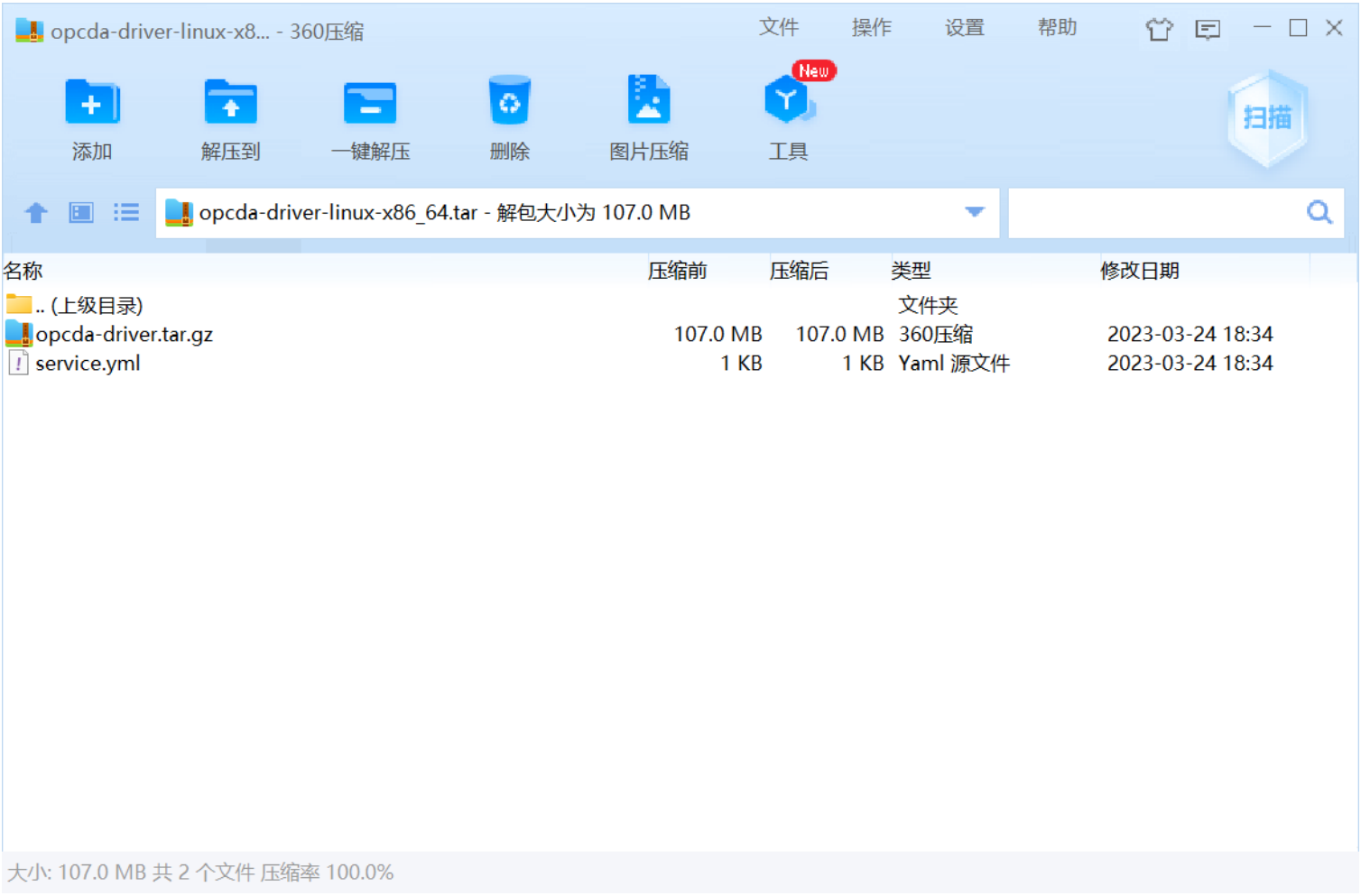
INFO

当 `service.yml` 文件中定义了 `Path` 和 `Ports` 时, 驱动部署到平台后, 会自动将驱动中的服务添加到网关代理, 驱动启动后, 就可以通过 `http[s]://IP:3030/rest/Path/otherPaths` 访问平台中的服务. 其中 `IP` 为平台的地址, `Path` 为 `service.yml` 中定义的 `Path` 字段的值, `otherPaths` 为驱动中的服务路径. 例如: `http://192.168.2.100:3030/rest/myDriver/users`.

6. 将所有资源打包为 `gzip` 文件. 将 `docker镜像` 和 `service.yml` 文件打包为 `gzip` 文件. 打包命令如下:

```
tar cvf myDriver-linux.tar myDriver.tar service.yml
gzip myDriver-linux.tar
```

打包后的 `gzip` 文件结构如下:



INFO

`linux` 系统整个打包过程对应的命令如下所示:

```
# 将驱动打包为镜像
docker build -t myDriver:1.0.0 .

# 导出镜像并压缩
docker save myDriver:1.0.0 | gzip > myDriver.tar.gz

# 将镜像文件和 service.yml 打包
tar cvf myDriver-linux.tar  myDriver.tar.gz service.yml

# 对整个驱动包进行压缩
gzip myDriver-linux.tar

# 最后得到 myDriver-linux.tar.gz 压缩文件
```

8. 部署

将上一步骤中得到的驱动安装包通过 `运维管理系统` 上传到平台, 平台会自动解析并安装驱动. 安装成功后, 就可以在项目中使用该驱动了.

安装驱动

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `离线上传驱动` 按钮, 选择上一步中得到的 `myDriver-linux.tar.gz` 文件, 点击 `确定` 按钮, 平台会自动解析并安装驱动.

如果驱动安装失败, 可以在 `运维管理系统` 的 `首页` 中查看详细的日志信息.

> **INFO**
>
> 不同版本的平台, `离线上传驱动` 按扭的位置可能不同.

使用驱动

当驱动成功安装到平台后, 就可以在项目中使用该驱动了.

具体使用方法请参考 驱动管理.

数据接入驱动接口说明

```java
/**
 * 驱动管理接口，主要实现对驱动运行状态的管理
 *
 * @param <DriverConfig> 驱动配置信息类型，需要与 schema 中的 {@code settings} 定义一致
 * @param <Command>      指令信息类型，需要与 schema 中的 {@code commands} 定义一致
 * @param <Tag>          数据点信息类型，需要与 schema 中的 {@code tags} 定义一致
 */
public interface DriverApp<DriverConfig, Command, Tag> {

    /**
     * 启动或重启驱动
     * <br>
     * 当驱动程序启动或与平台重新连接成功后，会自动调用方法.
     * <br>
     * 注：当驱动与平台连接断开并重连成功后也会调用该方法，所以在实现该方法时，需要处理好之前已经创建的资源. 例如：已经建立的 TCP 连接等
     *
     * @param config 驱动实例，模型及资产信息
     */
    void start(DriverConfig config);

    /**
     * 停止驱动
     * <br>
     * 当驱动进程退出时，会调用该方法，可以在该法内完成对相关资产的清理工作
     */
    void stop();

    /**
     * 执行平台对设备下发指令
     * <br>
     * 如果有额外需要保存的指令执行信息时，可通过 {@link DataSender#writeRunLog(RunLog)} 方法上报到平台.
     *
     * @param request 指令信息
     * @return 指令下发结果
     */
    Object run(Cmd<Command> request);

    /**
     * 批量下发指令
     * <br>
     * 如果有额外需要保存的指令执行信息时，可通过 {@link DataSender#writeRunLog(RunLog)} 方法上报到平台.
     *
     * @param request 指令信息
     * @return 指令下发结果
```

```
     */
    Object batchRun(BatchCmd<Command> request);

    /**
     * 向数据点写入数据
     * <br>
     * 有些驱动支持向数据点写入数据, 例如: OPC UA, 即可以从数据点读取数据, 也可以向数据点写入数据.
     * <br>
     * 如果驱动支持该功能, 需要在 schema 的 tags 定义中添加 rw 属性, 并且在数据点中设置为 true.
     * <br>
     * 详情请参考官方档中的 '数据接入驱动配置说明'
     *
     * @param request 指令信息
     * @return 指令下发结果
     */
    Object writeTag(Cmd<Tag> request);

    /**
     * 驱动调试
     * <br>
     * 该功能未实现
     */
    default Debug debug(Debug config) {
        return config;
    }

    /**
     * 获取驱动配置 schema 定义
     * <br>
     * 通常情况下, 可以将 schema 定义写在驱动程序的 resources 目录下, 然后通过 {@link
ClassLoader#getResourceAsStream(String)} 方法获取. 示例如下:
     *
     * <pre>
     * try (InputStream stream = this.getClass().getResourceAsStream("/schema.js")) {
     *     if (stream == null) {
     *         throw new IOException("未找到驱动配置文件");
     *     }
     *
     *     byte[] data = new byte[stream.available()];
     *     int n = stream.read(data);
     *     if (n != data.length) {
     *         throw new IllegalStateException("读取驱动配置文件异常, 数据不完整");
     *     }
     *     return new String(data, StandardCharsets.UTF_8);
     * } catch (IOException e) {
     *     throw new IllegalStateException("读取驱动配置文件异常", e);
     * }
     * </pre>
     *
     * @return 表单 schema
```

```
     */
    String schema();
}
```

泛型说明

`DriverApp` 接口要求提供 3 个泛型信息, 分别为 `驱动实例配置类型`、`指令配置类型` 和 `数据点配置类型`.

- **驱动实例配置类型** `SDK` 中提供了 2 个封装类 **DriverSingleConfig** 和 **DriverConfig**. `DriverSingleConfig` 类适用于 `驱动实例`、`模型` 和 `设备` 的驱动配置信息一致的场景. 此时, 只需要编写一个驱动配置类即可. 然后使用 `DriverSingleConfig<自定义驱动配置类>` 作为第 1 个泛型参数使用. 而 `DriverConfig` 适用于 `驱动实例`、`模型` 和 `设备`的配置不相同的场景, 需要分别编写相应的配置类. **注: `自定义驱动配置类` 的定义请参考 驱动配置类型 中的 `DriverConfig` 类.**
- **指令配置类型** 参考 配置类定义 中的 `Command` 类.
- **数据点配置类型** 参考 配置类定义 中的 `Tag` 类.

`DriverApp` 实现类的定义示例如下:

```
/**
 * 该示例中使用了 SDK 中提供的 DriverSingleConfig 类作为驱动实例配置类。
 *
 * 注: 需要将该类注入到 spring 容器中.
 */
@Component
public class MyDriver implements DriverApp<DriverSingleConfig<DriverConfig>, Command, Tag> {
    // ..
}
```

驱动与平台交互接口说明

```
/**
 * 驱动与平台交互接口
 * <br>
 * 主要用于驱动向平台上报采集到的数据, 运行过程中的重要事件, 日志等信息
 */
public interface DataSender {

    /**
     * 上报驱动采集到的数据
     *
     * @param point 数据点
```

```
     * @throws IllegalStateException 如果连接未建立或已断开
     * @throws DataSenderException   如果上报数据时发生异常
     */
    void writePoint(Point point) throws DataSenderException;

    /**
     * 上报资产采集到的数据. 部分信息会自动填充
     *
     * @param tableId   设备所属工作表标识
     * @param deviceId  设备编号
     * @param time      数据产生的时间. unix时间戳(ms)
     * @param tagValues 数据点的值. value 为 {@code null}的数据点不会上报
     * @throws IllegalStateException    如果连接未建立或已断开
     * @throws IllegalArgumentException 如果设备不存在或者数据点信息不正确
     * @throws DataSenderException      如果上报数据时发生异常
     */
    void writePoint(String tableId, String deviceId, long time, Map<String, Object>
tagValues) throws DataSenderException;

    /**
     * 上报资产采集到的数据. 部分信息会自动填充. 使用服务器接收到数据的时间作为数据产生时间
     *
     * @param tableId   设备所属工作表标识
     * @param deviceId  设备编号
     * @param tagValues 数据点的值. value 为 {@code null}的数据点不会上报
     * @throws IllegalStateException    如果连接未建立或已断开
     * @throws IllegalArgumentException 如果设备不存在或者数据点信息不正确
     * @throws DataSenderException      如果上报数据时发生异常
     */
    default void writePoint(String tableId, String deviceId, Map<String, Object> tagValues)
throws DataSenderException {
        this.writePoint(tableId, deviceId, 0L, tagValues);
    }

    /**
     * 发送事件
     *
     * @param event 事件信息
     * @return 事件发送结果
     * @throws IllegalStateException 如果连接未建立或已断开
     * @throws EventSenderException  如果发送事件时发生异常
     */
    Response writeEvent(Event event) throws EventSenderException;

    /**
     * 上报指令执行结果日志.
     * <br>
     * 一条指令可以产生多条日志
     *
     * @param runLog 日志
```

```
     * @return 上报结果
     * @throws IllegalStateException 如果连接未建立或已断开
     * @throws RunLogSenderException 如果发送运行日志时发生异常
     */
    Response writeRunLog(RunLog runLog) throws RunLogSenderException;

    /**
     * 更新设备信息
     *
     * @param tableDTO 设备信息
     * @return 更新结果
     * @throws IllegalArgumentException 如果参数不正确
     * @throws UpdateTableDataException 如果接口调用失败
     */
    Response updateTableData(UpdateTableDTO tableDTO) throws UpdateTableDataException;

    /**
     * 更新设备信息
     *
     * @param tableId 设备所在工作表标识
     * @param rowId   设备编号
     * @param fields  更新的字段信息
     * @return 更新结果
     */
    default Response updateTableData(String tableId, String rowId, Map<String, Object>
fields) {
        return this.updateTableData(new UpdateTableDTO(tableId, rowId, fields));
    }

    /**
     * 写入 {@code debug } 级别日志
     * <br>
     * 该日志可以在设备调试窗口中看到
     *
     * @param tableId  设备所属工作表标识
     * @param deviceId 设备编号
     * @param msg      日志内容
     * @throws IllegalStateException 如果连接未建立或已断开
     * @throws LogSenderException    如果写日志时发生异常
     */
    void logDebug(String tableId, String deviceId, String msg) throws LogSenderException;

    /**
     * 写入 {@code info } 级别日志
     * <br>
     * 该日志可以在设备调试窗口中看到
     *
     * @param tableId  设备所属工作表标识
     * @param deviceId 设备编号
     * @param msg      日志内容
```

```
     * @throws IllegalStateException 如果连接未建立或已断开
     * @throws LogSenderException    如果写日志时发生异常
     */
    void logInfo(String tableId, String deviceId, String msg) throws LogSenderException;

    /**
     * 写入 {@code warn} 级别日志
     * <br>
     * 该日志可以在设备调试窗口中看到
     *
     * @param tableId  设备所属工作表标识
     * @param deviceId 设备编号
     * @param msg      日志内容
     * @throws IllegalStateException 如果连接未建立或已断开
     * @throws LogSenderException    如果写日志时发生异常
     */
    void logWarn(String tableId, String deviceId, String msg) throws LogSenderException;

    /**
     * 写入 {@code error} 级别日志
     * <br>
     * 该日志可以在设备调试窗口中看到
     *
     * @param tableId  设备所属工作表标识
     * @param deviceId 设备编号
     * @param msg      日志内容
     * @throws IllegalStateException 如果连接未建立或已断开
     * @throws LogSenderException    如果写日志时发生异常
     */
    void logError(String tableId, String deviceId, String msg) throws LogSenderException;
}
```

在 `SDK` 中已经有上面的接口的实现类并且已经注入到 `spring` 容器中, 可以直接注入. 使用方式如下所示:

```
/**
 * 自定义驱动实现, 使用 {@link Component} 注解将该类注入到 {@code spring} 容器中.
 * 并使用 {@link Autowired} 注解注入 {@link DataSender} 接口的实现类
 */
@Component
public class MyDriver implements DriverApp<DriverConfig, Command, Tag> {

    /**
     * 使用 {@link @Autowired} 注解注入
     */
    @Autowired
    private DataSender dataSender;
```

```
    // 实现接口中的方法
}
```

驱动配置说明

以下是完整的驱动配置文件, 请参考该配置文件进行配置.

```
# 如果驱动提供 rest 服务时, 需要配置该项. 给所有接口添加统一前缀以方便网关进行代理
server:
  port: 8080
  servlet:
    context-path: /driver-mydriver
# 驱动配置
airiot:
  driver:
    id: mydriver                                    # 驱动ID
    name: 定制开发驱动                                # 驱动名称
    project-id: 6438a01ec25a859defb4b98c            # 项目ID, 在开发过程中需要填写, 但是发布后不需要
填写
    instance-id: 6238a0aec45a859defb4b6c7           # 驱动实例ID, 在开发过程中需要填写, 但是发布后不
需要填写
driver-grpc:
  host: 192.168.50.112                              # 驱动服务(driver)地址, 在开发过程中需要填写, 但
是发布后不需要填写
  port: 9224                                        # 驱动服务(driver)端口
mq:
  mqtt:
    host: 192.168.50.112                            # MQTT 地址
    port: 1883                                      # MQTT 端口
    username: admin                                 # MQTT 用户名
    password: public                                # MQTT 密码
    protocol-version: 0                             # MQTT 协议版本, 支持 0, 3, 4. 默认为 0, 优先使
用 3.1.1 如果不支持则使用 3.1
    publish-timeout: 1s                             # MQTT 发送消息超时时间, 默认为 1s
```

windows系统打包发布时的驱动配置

```
# 如果驱动提供 rest 服务时, 需要配置该项. 给所有接口添加统一前缀以方便网关进行代理
server:
  port: 8080
  servlet:
    context-path: /driver-mydriver
# 驱动配置
```

```yaml
airiot:
  driver:
    id: mydriver                                # 驱动ID
    name: 定制开发驱动                           # 驱动名称
driver-grpc:
  host: 127.0.0.1                               # 驱动服务(driver)地址，在开发过程中需要填写，但是发布后不需要填写
  port: 9224                                    # 驱动服务(driver)端口
mq:
  mqtt:
    host: 127.0.0.1                             # MQTT 地址
    port: 1883                                  # MQTT 端口
    username: admin                             # MQTT 用户名
    password: public                            # MQTT 密码
```

linux系统打包发布时的驱动配置

```yaml
# 如果驱动提供 rest 服务时，需要配置该项. 给所有接口添加统一前缀以方便网关进行代理
server:
  port: 8080
  servlet:
    context-path: /driver-mydriver
# 驱动配置
airiot:
  driver:
    id: mydriver                                # 驱动ID
    name: 定制开发驱动                           # 驱动名称
driver-grpc:
  host: driver                                  # 驱动服务(driver)地址，在开发过程中需要填写，但是发布后不需要填写
  port: 9224                                    # 驱动服务(driver)端口
mq:
  mqtt:
    host: mqtt                                  # MQTT 地址
    port: 1883                                  # MQTT 端口
    username: admin                             # MQTT 用户名
    password: public                            # MQTT 密码
```

流程插件开发

本文将会详细介绍如何使用 `Java SDK` 开发流程插件. 示例项目目上传至 https://github.com/air-iot/sdk-java-examples/tree/master/flow-plugin-example.

介绍

`流程插件` 是扩展 `流程引擎` 中的节点的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程插件来实现自定义的功能.

> **INFO**
>
> `流程插件` 只是扩展流程功能的方式之一. 除了 `流程插件` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

该过程同 数据接入驱动开发-创建项目 中的创建项目过程一致.

2. 引入SDK

以 `maven` 为例, 在 `pom.xml` 中引入 `sdk-flow-plugin-starter` 依赖.

```
<dependencies>
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-flow-plugin-starter</artifactId>
        <version>4.x.x</version>
    </dependency>
</dependencies>
```

3. 实现流程插件接口

`SDK` 中定义了 `流程插件接口`, 该接口是平台与插件交互的桥梁. 开发者需要实现这个接口, 并且将实现类注入到 `spring` 容器中. 接口定义及详细说明见 流程插件接口说明.

流程插件启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程执行到该插件对应的节点时, 会发送请求给该插件对应的程序. `SDK` 接收到请求后会调用对应的插件实现, 并将插件处理结果返回给流程引擎.

> **INFO**
>
> 一个程序中可以包含多个流程插件. 但是每个流程插件的名称必须唯一.

4. 配置插件

插件配置主要是插件与平台的连接配置.

```yaml
flow-engine:
  host: 192.168.11.101              # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
  connect-timeout: 15s              # 连接超时
  retry-interval: 30s               # 重连间隔
  heartbeat-interval: 30s           # 心跳间隔
```

windows系统打包发布时的插件配置

```yaml
flow-engine:
  host: 127.0.0.1                   # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
```

linux系统打包发布时的插件配置

```yaml
flow-engine:
  host: flow-engine                 # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
```

> **INFO**
>
> `connect-timeout`、`retry-interval` 和 `heartbeat-interval` 通常情况下不需要修改.

5. 打包

流程插件打包就是将开发完成的程序打包为可以在平台部署的服务. 打包方式与 数据接入驱动开发-打包 中的打包方式基本一致， 但 `service.yml` 文件的内容稍有不同.

windows系统插件配置文件

```yaml
# 必填项. 服务名称
Name: myPlugin
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 插件描述信息
# 插件的配置文件名称, 平台在安装插件服务时会查找打包文件中查找该文件. 一般固定填写 application.yml
ConfigType: application.yml
# 必填项. 固定为 server
GroupName: server
# 必填项. 启动命令. 还可以添加一些启动参数, 例如: -Xms512m -Xmx1024m
Command: java -jar myPlugin.jar
```

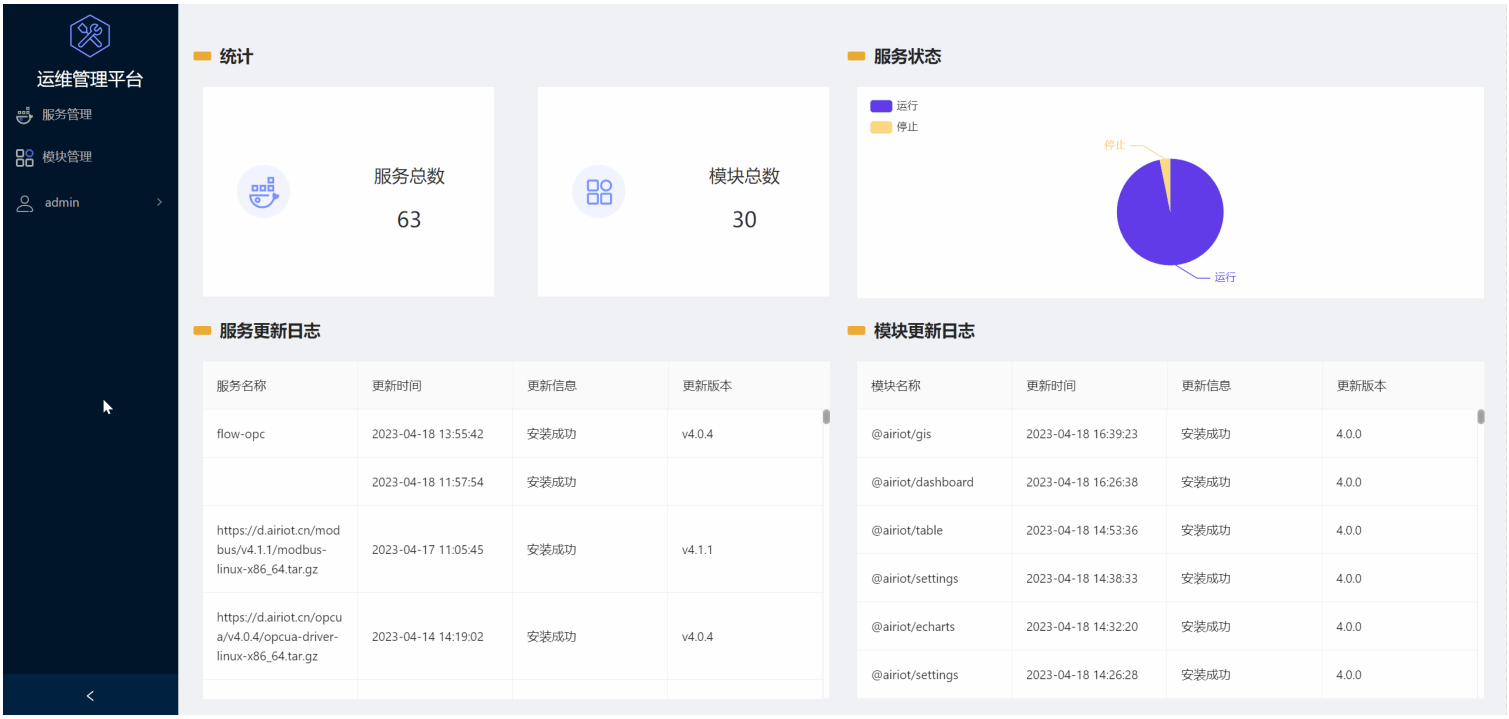
linux系统插件配置文件

```yaml
# 必填项. 服务名称
Name: myPlugin
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 插件描述信息
# 必填项. 固定为 server
GroupName: server
# 固定为 None
Service: None
```

6. 部署

`流程插件` 的部署方式与 `数据接入驱动` 不同, 每个流程插件都是一个独立的服务, 在整个平台中只有一个实例. 部署过程如下:

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `添加服务` 按钮, 然后选择 `离线添加`.
4. 点击 `上传` 按扭, 选择刚刚打包好的流程插件压缩包, 然后点击 `提交` 按钮.



> **INFO**
>
> 如果 流程插件 部署失败, 可以在 运维管理系统 的首页中查看日志.

流程插件接口说明

```java
/**
 * 流程插件接口
 */
public interface FlowPlugin<Request> {

    /**
     * 获取插件名称
     * <br>
     * 插件名称在整个平台中必须唯一，流程引擎根据插件名称来区分不同的插件.
     *
     * @return 插件名称
     */
    String getName();

    /**
     * 插件类型
     *
     * @return 插件类型
     */
    FlowPluginType getPluginType();
```

```
    /**
     * 与流程引擎连接状态发生变化时触发
     * <br>
     * 注: 该方法在每次连接状态发生变化时都会被调用
     *
     * @param connected 当前与流程引擎的连接状态. {@code true} 连接已建立, {@code false} 连接已
断开
     */
    default void onConnectionStateChange(boolean connected) {

    }

    /**
     * 当流程插件服务启动时执行的操作
     * <br>
     * 可以在此方法中执行一些初始化操作, 如建立连接, 初始化对象等.
     * <br>
     * 注: 该方法只会调用一次, 并且在 {@link #onConnectionStateChange(boolean)} 之前执行
     */
    default void onStart() {

    }

    /**
     * 当流程插件服务停止时执行的操作
     * <br>
     * 可以在此方法中执行一些清理操作, 如关闭连接等.
     * <br>
     * 注: 该方法只会调用一次.
     */
    default void onStop() {

    }

    /**
     * 执行流程引擎的请求并返回处理结果.
     * <br>
     * 如果抛出异常则视为请求执行失败. 否则视为请求执行成功.
     *
     * @param task 任务信息
     * @return 请求处理结果.
     * @throws FlowPluginException 如果请求执行失败
     */
    FlowTaskResult execute(FlowTask<Request> task) throws FlowPluginException;
}
```

流程扩展节点开发

本文将会详细介绍如何使用 `Java SDK` 扩展流程节点. 示例项目目上传至 https://github.com/air-iot/sdk-java-examples/tree/master/flow-extension-example.

介绍

`流程扩展节点` 是扩展 `流程引擎` 中的节点的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程扩展节点来实现自定义的功能.

> **INFO**
>
> `流程扩展节点` 只是扩展流程功能的方式之一. 除了 `流程扩展节点` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

该过程同 流程插件开发-创建项目 中的创建项目过程一致.

2. 引入SDK

以 `maven` 为例, 在 `pom.xml` 中引入 `sdk-flow-extension-starter` 依赖.

```
<dependencies>
    <dependency>
        <groupId>io.github.air-iot</groupId>
        <artifactId>sdk-flow-extension-starter</artifactId>
        <version>4.x.x</version>
    </dependency>
</dependencies>
```

3. Schema 定义

每个流程扩展节点需要提供一个 `schema` 定义, 该 `schema` 描述了当前扩展节点的输入参数定义. 详细说明点击查看. 示例如下:

```
{
  "type": "object",
  "properties": {
    "num1": {
      "title": "参数1",
      "type": "number"
    },
    "num2": {
      "title": "参数2",
      "type": "number"
    }
  },
  "required": [
    "num1",
    "num2"
  ]
}
```

4. 实现流程扩展节点接口

`SDK` 中定义了 `流程扩展节点接口`, 该接口是平台与扩展节点交互的桥梁. 开发者需要实现这个接口, 并且将实现类注入到 `spring` 容器中. 接口定义及详细说明见 流程扩展节点接口说明.

流程扩展节点启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程调用该扩展节点时, 会发送请求给该扩展节点对应的程序. `SDK` 接收到请求后会调用对应的流程扩展节点实现, 并将处理结果返回给流程引擎.

> **INFO**
>
> 一个程序中可以包含多个流程扩展节点. 但是每个流程扩展节点的ID和名称必须唯一.

5. 配置流程扩展节点

流程扩展节点配置主要是扩展节点与平台的连接配置.

```
flow-engine:
  host: 192.168.11.101              # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
  connect-timeout: 15s              # 连接超时, 可选. 默认: 15s
```

```
  retry-interval: 30s                # 重连间隔，可选。默认：30s
  heartbeat-interval: 30s            # 心跳间隔，可选。默认：30s
  max-threads: 10                    # 最大线程数，可选。默认：0，即取当前主机的CPU核数
```

windows系统打包发布时的流程扩展节点配置

```
flow-engine:
  host: 127.0.0.1                    # 流程引擎服务地址
  port: 2333                         # 流程引擎服务端口
```

linux系统打包发布时的流程扩展节点配置

```
flow-grpc:
  host: flow-engine                  # 流程引擎服务地址
  port: 2333                         # 流程引擎服务端口
```

> **INFO**
>
> - `connect-timeout`、`retry-interval` 和 `heartbeat-interval` 通常情况下不需要修改.
> - 每次请求都是异步执行的, `max-threads` 用来设置异步执行的最大线程数. 如果不设置, 则取当前主机的CPU核数.

6. 打包

流程扩展节点打包就是将开发完成的程序打包为可以在平台部署的服务. 打包方式与 数据接入驱动开发-打包 中的打包方式基本一致.

windows系统流程扩展节点配置文件

```
# 必填项。服务名称
Name: myFlowExtNode
# 必填项。例如：1.0.0
Version: 1.0.0
# 非必填项。
Description: 扩展节点描述信息
# 流程扩展节点的配置文件名称，平台在安装流程扩展节点服务时会查找打包文件中查找该文件。一般固定填写application.yml
ConfigType: application.yml
# 必填项。固定为 server
GroupName: server
```

```
# 必填项. 启动命令. 还可以添加一些启动参数, 例如: -Xms512m -Xmx1024m
Command: java -jar myFlowExtNode.jar
```

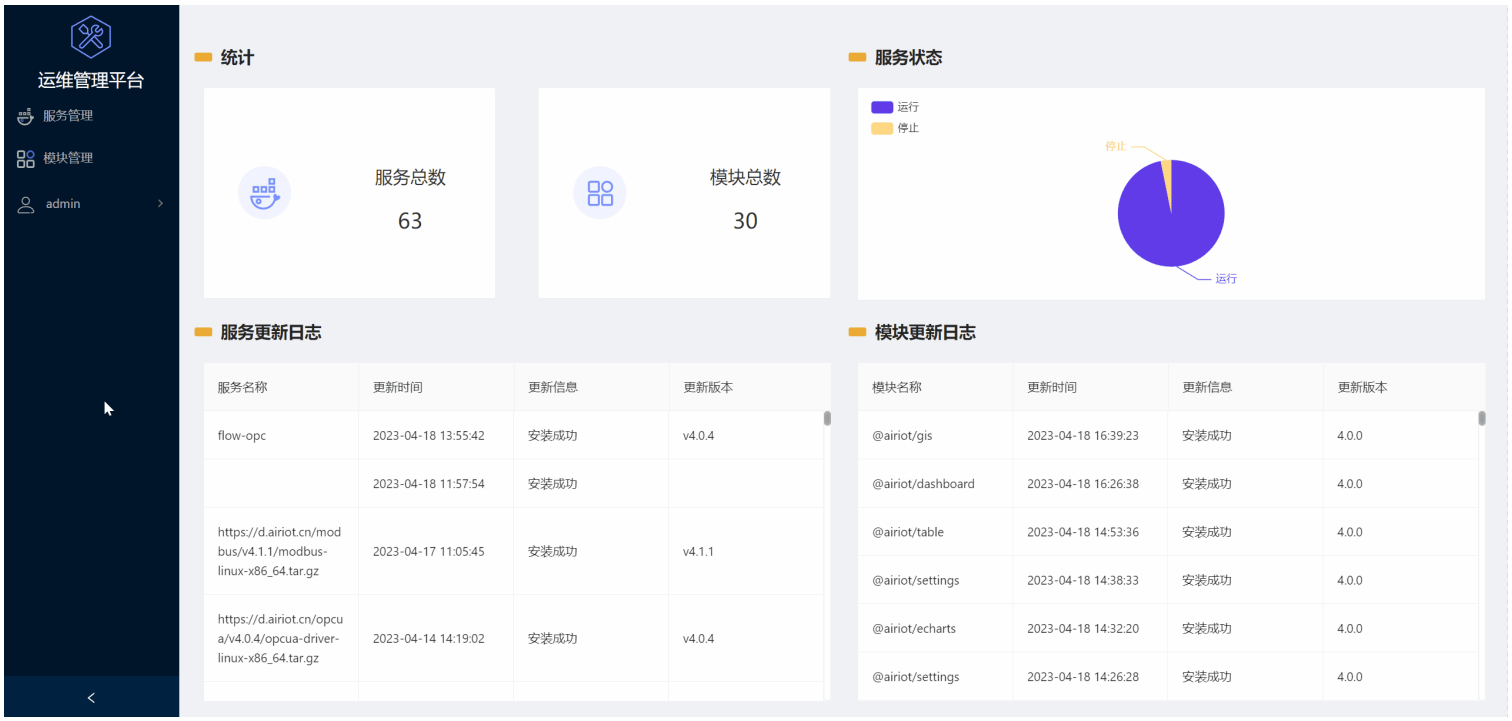
linux系统流程扩展节点配置文件

```
# 必填项. 服务名称
Name: myFlowExtNode
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 扩展节点描述信息
# 必填项. 固定为 server
GroupName: server
# 固定为 None
Service: None
```

7. 部署

每个 `流程扩展节点` 都是一个独立的服务, 在整个平台中只有一个实例, 其部署方式与 `流程插件` 一致, 部署过程如下:

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `添加服务` 按钮, 然后选择 `离线添加`.
4. 点击 `上传` 按扭, 选择刚刚打包好的流程扩展节点压缩包, 然后点击 `提交` 按钮.



> **INFO**
>
> 如果 `流程扩展节点` 部署失败, 可以在 `运维管理系统` 的首页中查看日志.

流程扩展节点接口说明

```
/**
 * 流程扩展节点接口
 */
public interface FlowExtension<Request> {

    /**
     * 获取扩展节点标识
     * <br>
     * 节点标识在整个平台中必须唯一, 流程引擎根据流程扩展节点标识来区分不同的节点.
     *
     * @return 节点标识
     */
    String getId();

    /**
     * 扩展节点名称
     *
     * @return 节点名称
     */
    String getName();

    /**
     * 与流程引擎连接状态发生变化时触发
     * <br>
     * 注: 该方法在每次连接状态发生变化时都会被调用
     *
     * @param connected 当前与流程引擎的连接状态. {@code true} 连接已建立, {@code false} 连接已
断开
     */
    default void onConnectionStateChange(boolean connected) {

    }

    /**
     * 当流程流程扩展节点服务启动时执行的操作
     * <br>
     * 可以在此方法中执行一些初始化操作, 如建立连接, 初始化对象等.
     * <br>
     * 注: 该方法只会调用一次, 并且在 {@link #onConnectionStateChange(boolean)} 之前执行
     */
```

```
    default void onStart() {

    }

    /**
     * 当流程流程扩展节点服务停止时执行的操作
     * <br>
     * 可以在此方法中执行一些清理操作，如关闭连接等.
     * <br>
     * 注：该方法只会调用一次.
     */
    default void onStop() {

    }

    /**
     * 获取扩展节点的 schema 定义
     * <br>
     * 如果抛出异常则视为请求执行失败. 否则视为请求执行成功.
     *
     * @return schema 定义信息
     * @throws FlowExtensionException 如果请求执行失败
     */
    String schema() throws FlowExtensionException;

    /**
     * 执行请求
     *
     * @param request 请求参数
     * @return 请求执行结果，必须为可以序列化为 JSON 的对象，例如：Map，自定义 Class 等.
     * @throws FlowExtensionException 如果请求执行异常
     */
    Object run(Request request) throws FlowExtensionException;
}
```

Request 泛型说明

`FlowExtension` 接口中的泛型 `Request` 为请求参数类型, 用于接收流程引擎发送的请求参数. 该泛型的类型由 `schema` 定义决定. 该泛型可以为 `Map` 或自定义类型, 示例如下:

```
/**
 * 使用 Map 作为请求参数类型，注意：Map 的 key 必须为 String.
 */
public class MyExtension1 implements FlowExtension<Map<String, Object>> {
  // ...
}
```

```java
public class MyRequest {
  private int base;
  private int pow;

  // ...
}

/**
 * 使用自定义类型作为请求参数类型
 */
public class MyExtension1 implements FlowExtension<MyRequest> {
  // ...
}
```

平台接口客户端

介绍

平台客户端 SDK 用于访问平台接口，提供了平台接口的 Java 实现, 可以很方便实现第三方系统与平台的集成。平台客户端 SDK 中包括常用的 `租户管理`、`项目管理`、`用户管理`、`角色管理`、`工作表及数据管理`、`系统变量(数据字典)`、`报警管理` 等接口. 示例项目目上传至 https://github.com/air-iot/sdk-java-examples/tree/master/sdk-client-http-example.

使用方式

1. 在项目中引入依赖

在 `pom.xml` 中引入依赖:

```
<dependency>
    <groupId>io.github.air-iot</groupId>
    <artifactId>sdk-client-http-starter</artifactId>
    <version>4.x.x</version>
</dependency>
```

2. 配置平台访问信息

在使用平台客户端之前, 需要在 `application.yml` 中配置平台的访问信息, 包括平台的 `host`、`app-key`、`app-secret` 等信息, 配置示例如下:

```
airiot:
  client:
    authorization:                                          # 授权信息
      type: project                                         # 授权类型, 可选值: project, tenant
      project-id: 1438f01ec22a859dedb3b98c                  # 授权项目ID, 当授权类型为 project 时必填. 如果平台为单项目版本时, 可以填 `default` 或不配置该项
      app-key: e2233587-a2c9-ecf4-ed6e-c4a6122d2308         # 授权应用ID
      app-secret: 3b74042c-b3ea-e548-a18c-cc78a20ffd0b      # 授权应用密钥
    http:                                                   # http 协议客户端配置
      host: http://192.168.1.100:31000                      # 平台网关地址. 平台网关默认端口为 31000
      default-config:                                       # 请求默认配置, 当具体服务未配置时,
```

```
使用默认配置
        connect-timeout: 3000                                    # 连接超时，可以使用 `3s`, `1m` 等
格式
        read-timeout: 5000                                       # 请求超时
      services:                                                  # 具体服务的配置
        core:                                                    # 服务名称，可选值：core，data-
service，spm，warning
          connect-timeout: 5000
          read-timeout: 15000
```

> **INFO**
>
> 当授权类型为 `project` 时，只能访问该项目中的相关资源。 当授权类型为 `tenant` 时，可以访问该租户下的所有项目内的资源，但需要在请求项目内的资源时需要指定目标项目ID. 点击查看。
>
> **关于如何创建二次开发应用授权，请查看 二次开发-第三方应用添加。**

3. 在需要调用平台接口的类中注入客户端对象

在引入客户端 SDK 后，各客户端对象会自动注入到 Spring 容器中，可以直接在需要调用平台接口的类中注入客户端对象，并调用相应的接口方法。示例代码如下：

```java
@Component
public class Demo {

    /**
     * 注入用户客户端接口
     */
    @Autowired
    private UserClient userClient;

    /**
     * 注入角色客户端接口
     */
    @Autowired
    private RoleClient roleClient;

    /**
     * 创建用户
     */
    public void createUser() {
        User user = new User();
        user.setName("admin");
        user.setPassword("123456");
        user.setPhone("187xxxx1234");
```

```
        // 调用平台接口并获取返回结果
        ResponseDTO<InsertResult> responseDTO = this.userClient.create(user);

        // 处理返回结果
    }
}
```

4. 统一响应格式说明

所有客户端接口方法统一返回 `ResponseDTO<T>` 结构, 泛型 `T` 代表请求返回的数据类型, 该类型可能为 `Void`。 `ResponseDTO<T>` 结构如下:

```
public class ResponseDTO<T> {
/**
     * 请求是否成功标识. 如果为 {@code true} 表明请求成功, 否则请求失败
     */
    private boolean success;
    /**
     * 请求状态码
     */
    private int code;
    /**
     * 响应信息
     */
    private String message;
    /**
     * 详细信息
     */
    private String detail;
    /**
     * 字段级别的错误信息
     */
    private String field;
    /**
     * 总记录数
     * <br>
     * 当查询请求设置了 {@code cn.airiot.sdk.client.builder.Query#withCount} 为 {@code true}
时, 该字段为匹配的记录数量.
     */
    private long count;
    /**
     * 响应数据
     */
    private T data;
}
```

客户端列表

空间管理

空间管理服务(spm)客户端主要包括 `项目管理` 等接口。在需要为该服务内的接口配置独立的超时时间时, 可以在 `application.yml` 中配置 `airiot.client.http.services.spm` 节点。 包含的客户端接口如下:

资源对象	客户端类	内容
项目信息	io.github.airiot.sdk.client.service.spm.ProjectClient	包括对项目信息的增删改查以及授权等操作

项目管理客户端

```
/**
 * 项目信息管理客户端接口
 */
public interface ProjectClient {

    /**
     * 创建项目
     *
     * @param project 项目信息
     * @return 项目信息或错误信息
     */
    ResponseDTO<ProjectCreateResult> create(@Nonnull Project project);

    /**
     * 查询项目信息
     *
     * @param query 查询条件
     * @return 项目信息
     */
    ResponseDTO<List<Project>> query(@Nonnull Query query);

    /**
     * 查询全部项目信息
     *
     * @return 项目信息
     */
    ResponseDTO<List<Project>> queryAll();

    /**
```

```
     * 根据项目ID查询项目信息
     *
     * @param projectId 项目ID
     * @return 项目信息
     */
    ResponseDTO<Project> queryById(@Nonnull String projectId);

    /**
     * 更新项目信息
     * <br>
     * 如果字段的值为 {@code null} 则不更新
     *
     * @param project 更新后的项目信息
     * @return 更新结果
     */
    ResponseDTO<Void> update(@Nonnull Project project);

    /**
     * 替换项目信息
     *
     * @param project 替换后的项目信息
     * @return 替换结果
     */
    ResponseDTO<Void> replace(@Nonnull Project project);

    /**
     * 删除项目
     *
     * @param projectId 项目ID
     * @return 删除结果
     */
    ResponseDTO<Void> deleteById(@Nonnull String projectId);
}
```

核心服务

核心服务(`core`)客户端主要包括 `用户管理`、`角色管理`、`工作表及数据管理`、`系统变量(数据字典)` 等接口，并且不断在丰富和完善中。 在需要为该服务内的接口配置独立的超时时间时, 可以在 `application.yml` 中配置 `airiot.client.http.services.core` 节点。包含的客户端接口如下:

资源对象	客户端类	内容
用户信息	io.github.airiot.sdk.client.service.core.UserClient	包括对用户信息的增删改查操作

资源对象	客户端类	内容
角色信息	io.github.airiot.sdk.client.service.core.RoleClient	包括对角色信息的增删改查, 以及对象角色权限信息的修改操作
系统变量(数据字典)	io.github.airiot.sdk.client.service.core.SystemVariableClient	包括对系统变量(数据字典)的增删改查操作
工作表定义	io.github.airiot.sdk.client.service.core.TableSchemaClient	包括对工作表的定义的查询操作
工作表数据	io.github.airiot.sdk.client.service.core.SpecificTableDataClient	包括对工作表内的数据进行增删除改查等操作
时序数据	io.github.airiot.sdk.client.service.core.TimingDataClient	包括对数据点历史数据和最新数据的查询操作

用户管理客户端

```
/**
 * 用户管理客户端
 */
public interface UserClient extends PlatformClient {

    /**
     * 创建用户
     *
     * @param user 用户信息
     * @return 用户ID或错误信息
     */
    ResponseDTO<User> create(@Nonnull User user);

    /**
     * 更新用户信息
     *
     * @param userId 要更新的用户ID
     * @param user   要更新的用户信息
```

```
     * @return 更新结果
     */
    ResponseDTO<Void> update(@Nonnull String userId, @Nonnull User user);

    /**
     * 替换用户全部信息
     *
     * @param userId 要替换的用户ID
     * @param user   替换后的用户信息
     * @return 替换结果
     */
    ResponseDTO<Void> replace(@Nonnull String userId, @Nonnull User user);

    /**
     * 根据用户ID删除用户
     *
     * @param userId 用户ID
     * @return 删除结果
     */
    ResponseDTO<Void> deleteById(@Nonnull String userId);

    /**
     * 根据条件查询用户信息
     *
     * @param query 查询条件
     * @return 用户信息或错误信息
     */
    ResponseDTO<List<User>> query(@Nonnull Query query);

    /**
     * 根据用户ID查询用户信息
     *
     * @param userId 用户ID
     * @return 用户信息或错误信息
     */
    ResponseDTO<User> queryById(@Nonnull String userId);

    /**
     * 根据用户名查询用户信息
     *
     * @param name 用户名
     * @return 用户信息或错误信息
     */
    default ResponseDTO<List<User>> queryByName(@Nonnull String name) {
        return query(Query.newBuilder().select(User.class).filter().eq(User::getName,
name).end().build());
    }
}
```

角色管理客户端

```
/**
 * 角色客户端
 */
public interface RoleClient {

    /**
     * 创建角色
     *
     * @param role 角色信息
     * @return 创建结果. 如果创建成功, 则返回角色ID
     */
    ResponseDTO<InsertResult> create(@Nonnull Role role);

    /**
     * 查询角色信息
     *
     * @param query 查询条件
     * @return 角色信息
     */
    ResponseDTO<List<Role>> query(@Nonnull Query query);

    /**
     * 根据角色ID查询角色信息
     *
     * @param roleId 角色ID
     * @return 角色信息
     */
    ResponseDTO<Role> queryById(@Nonnull String roleId);

    /**
     * 根据角色名称查询角色信息
     *
     * @param roleName 角色名称
     * @return 角色信息列表
     */
    default ResponseDTO<List<Role>> queryByName(@Nonnull String roleName) {
        return query(Query.newBuilder().select(Role.class).filter().eq(Role::getName,
roleName).end().build());
    }

    /**
     * 查询全部角色信息
     *
     * @return 角色信息列表
     */
    default ResponseDTO<List<Role>> queryAll() {
        return query(Query.newBuilder().select(Role.class).build());
```

```
    }

    /**
     * 替换角色全部信息
     *
     * @param roleId 被替换角色ID
     * @param role   替换后的角色信息
     * @return 替换结果
     */
    ResponseDTO<Void> replace(@Nonnull String roleId, @Nonnull Role role);

    /**
     * 更新角色信息
     *
     * @param roleId 被更新角色ID
     * @param role   要替换的角色信息(值为 null 的字段不会被更新)
     * @return 更新结果
     */
    ResponseDTO<Void> update(@Nonnull String roleId, @Nonnull Role role);

    /**
     * 删除角色信息
     *
     * @param roleId 角色ID
     * @return 删除结果
     */
    ResponseDTO<Void> deleteById(@Nonnull String roleId);
}
```

系统变量客户端

```
/**
 * 系统变量(数据字典)客户端
 */
public interface SystemVariableClient {

    /**
     * 查询系统变量信息
     *
     * @param query 查询条件
     * @return 系统变量信息
     */
    ResponseDTO<List<SystemVariable>> query(@Nonnull Query query);

    /**
     * 查询全部系统变量信息
     *
     * @return 系统变量信息
```

```
     */
    default ResponseDTO<List<SystemVariable>> queryAll() {
        return query(Query.newBuilder().select(SystemVariable.class).build());
    }

    /**
     * 根据系统变量编号查询系统变量信息
     *
     * @param uid 系统变量编号
     * @return 系统变量信息
     */
    default ResponseDTO<SystemVariable> queryByUId(@Nonnull String uid) {
        ResponseDTO<List<SystemVariable>> response = query(Query.newBuilder()
                .select(SystemVariable.class)
                .filter()
                .eq(SystemVariable::getUid, uid).end()
                .build());
        if (!response.isSuccess()) {
            return new ResponseDTO<>(response.isSuccess(), response.getCode(),
response.getMessage(), response.getDetail(), null);
        }

        List<SystemVariable> systemVariables = response.getData();
        if (systemVariables != null && systemVariables.size() > 1) {
            throw new IllegalStateException("根据 uid '" + uid + "' 查询到多个系统变量, " +
systemVariables);
        }

        return new ResponseDTO<>(response.isSuccess(), response.getCode(),
                response.getMessage(), response.getDetail(),
                systemVariables == null || systemVariables.isEmpty() ? null :
systemVariables.get(0));
    }

    /**
     * 根据系统变量ID查询系统变量信息
     *
     * @param id 系统变量ID
     * @return 系统变量信息
     */
    ResponseDTO<SystemVariable> queryById(@Nonnull String id);

    /**
     * 根据系统变量名称查询系统变量信息
     *
     * @param name 系统变量名称
     * @return 系统变量信息
     */
    default ResponseDTO<List<SystemVariable>> queryByName(@Nonnull String name) {
        return query(Query.newBuilder().select(SystemVariable.class)
```

```
            .filter().eq(SystemVariable::getName, name).end()
            .build());
    }

    /**
     * 创建系统变量信息
     *
     * @param systemVariable 系统变量信息
     * @return 创建结果
     */
    ResponseDTO<InsertResult> create(@Nonnull SystemVariable systemVariable);

    /**
     * 替换系统变量信息
     *
     * @param id             系统变量ID
     * @param systemVariable 系统变量信息
     * @return 替换结果
     */
    ResponseDTO<Void> replace(@Nonnull String id, @Nonnull SystemVariable systemVariable);

    /**
     * 更新系统变量信息
     *
     * @param id             系统变量ID
     * @param systemVariable 系统变量信息
     * @return 更新结果
     */
    ResponseDTO<Void> update(@Nonnull String id, @Nonnull SystemVariable systemVariable);

    /**
     * 更新系统变量的值
     *
     * @param id    系统变量ID
     * @param value 系统变量值
     * @return 更新结果
     */
    default ResponseDTO<Void> updateValue(@Nonnull String id, @Nonnull Object value) {
        SystemVariable variable = new SystemVariable();
        variable.setId(id);
        variable.setValue(value);
        return this.update(id, variable);
    }

    /**
     * 删除系统变量信息
     *
     * @param id 系统变量ID
     * @return 删除结果
     */
```

```
    ResponseDTO<Void> deleteById(@Nonnull String id);
}
```

工作表定义客户端

```
/**
 * 工作表定义客户端
 */
public interface TableSchemaClient {

    /**
     * 查询工作表定义
     *
     * @return 工作表定义信息
     */
    ResponseDTO<List<TableSchema>> query(@Nonnull Query query);

    /**
     * 查询全部工作表定义
     *
     * @return 工作表定义信息
     */
    default ResponseDTO<List<TableSchema>> queryAll() {
        return query(Query.newBuilder().select(TableSchema.class).build());
    }

    /**
     * 查询工作表定义
     *
     * @param tableId 表标识
     * @return 工作表定义信息
     */
    ResponseDTO<TableSchema> queryById(@Nonnull String tableId);

    /**
     * 根据工作表标题查询工作表定义
     *
     * @param tableTitle 工作表标题
     * @return 工作表定义信息
     */
    default ResponseDTO<List<TableSchema>> queryByTitle(@Nonnull String tableTitle) {
        return
query(Query.newBuilder().select(TableSchema.class).filter().eq(TableSchema::getTitle,
tableTitle).end().build());
    }
}
```

工作表数据客户端

由于每个工作表的结构都不相同, 因此工作表数据的客户端接口是动态生成的, 通过 `TableDataClientFactory` 工厂类获取, 在操作具体的工作表数据时, 需要先获取工作表数据客户端, 然后再进行操作。
`TableDataClientFactory` 对象已经注入到 Spring 容器中, 可以直接注入使用。

```
/**
 * 工作表记录客户端工厂, 可用于创建指定工作表记录客户端
 */
public abstract class TableDataClientFactory {

    /**
     * 根据工作表记录类创建指定工作表记录客户端. 要求该类必须标注 {@link WorkTable} 注解
     *
     * @param clazz 工作表记录类
     * @param <T>   工作表记录类型
     * @return 工作表记录客户端
     */
    public <T> SpecificTableDataClient<T> newClient(Class<T> clazz) {
        // 创建工作表记录客户端
    }

    /**
     * 根据工作表记录类和表标识创建指定工作表记录客户端
     *
     * @param tableId 表标识
     * @param clazz   工作表记录类
     * @param <T>     工作表记录类型
     * @return 工作表记录客户端
     */
    public <T> SpecificTableDataClient<T> newClient(String tableId, Class<T> clazz) {
        // 创建工作表记录客户端
    }
}
```

创建具体工作表数据客户端对象示例如下:

```

/**
 * 学生信息表, 使用 @WorkTable 注解标注工作表标识
 */
@WorkTable("student")
public class Student {
  /**
   * 学号
   */
  private String id;
```

```java
    /**
     * 姓名
     */
    private String name;
    /**
     * 年龄
     */
    private String age;
    /**
     * 性别.
     * 如果工作表定义中的字段名与类中的属性名不一致，可以使用 @Field 注解标注.
     * <br>
     * 例如: 性别在工作表定义中的名称为 "sex"，可以按下面方式标注.
     */
    @Field("sex")
    private String gender;
}


@Service
public class DemoService {

    /**
     * 学生信息表的数据客户端
     */
    private final SpecificTableDataClient<Student> studentClient;

    public DemoService(TableDataClientFactory factory) {
        // 创建学生信息表的数据客户端
        // 方式一: 要求工作表记录类必须标注 @WorkTable 注解
        this.studentClient = factory.newClient(Student.class);

        // 方式二: 手动指定工作表标识
        // this.studentClient = factory.newClient("student", Student.class);
    }

    public void createStudent() {
        Student student = new Student();
        student.setId("1001");
        student.setName("张三");
        student.setAge("18");
        student.setGender("男");

        // 调用学生信息表的数据客户端，向学生信息表中添加记录
        ResponseDTO<InsertResult> response = this.studentClient.create(student);

        // 处理响应结果
    }
}
```

> **INFO**
>
> 注: `TableDataClientFactory` 对会对已创建的工作表数据客户端缓存(工作表标识作为缓存的 key), 因此也可以在需要使用的时候创建. 在使用 `newClient(Class<T> clazz)` 方法创建工作表数据客户端时, 要求类型上必须添加 `@WorkTable` 注解.

```
/**
 * 指定工作表的数据客户端
 * @param <T> 承载工作表记录的类型的泛型
 */
public abstract class SpecificTableDataClient<T> {

  /**
   * 向工作表中添加记录
   *
   * @param row 记录
   * @return 数据添加结果
   */
  public abstract ResponseDTO<InsertResult> create(@Nonnull T row);

  /**
   * 向工作表中批量添加记录
   *
   * @param rows 记录列表
   * @return 数据添加结果
   */
  public abstract ResponseDTO<BatchInsertResult> create(@Nonnull List<T> rows);

  /**
   * 更新工作表记录
   *
   * @param rowId 记录ID
   * @param data  要更新的记录
   * @return 数据更新结果
   */
  public abstract ResponseDTO<UpdateOrDeleteResult> update(@Nonnull String rowId, @Nonnull T
data);

  /**
   * 批量更新工作表记录.
   * <br>
   * 更新所有与 {@code query} 匹配的记录
   *
   * @param query 更新条件
   * @param data  要更新的数据
   * @return 数据更新结果
   */
```

```
    public abstract ResponseDTO<UpdateOrDeleteResult> update(@Nonnull Query query, @Nonnull T
data);

    /**
     * 替换记录全部信息
     *
     * @param rowId 记录ID
     * @param data  替换后的记录信息
     */
    public abstract ResponseDTO<Void> replace(@Nonnull String rowId, @Nonnull T data);

    /**
     * 根据记录ID删除数据
     *
     * @param rowId 记录ID
     * @return 数据删除结果
     */
    public abstract ResponseDTO<Void> deleteById(@Nonnull String rowId);

    /**
     * 批量删除工作表记录
     *
     * @param query 删除条件
     * @return 数据删除结果
     */
    public abstract ResponseDTO<UpdateOrDeleteResult> deleteByQuery(@Nonnull Query query);

    /**
     * 根据条件查询用户信息
     *
     * @param query 查询条件
     * @return 用户信息或错误信息
     */
    public abstract ResponseDTO<List<T>> query(@Nonnull Query query);

    /**
     * 根据ID查询记录信息
     *
     * @param rowId 记录ID
     * @return 记录信息或错误信息
     */
    public abstract ResponseDTO<T> queryById(@Nonnull String rowId);
}
```

时序数据客户端

```
/**
 * 时序数据客户端
```

```
 */
public interface TimingDataClient {

    /**
     * 查询数据点的历史数据
     *
     * @param queries 查询信息，可以对数据进行过滤、分组、汇总和排序等操作
     * @return 查询结果
     */
    List<TimingData> query(List<TimingDataQuery> queries);

    /**
     * 查询指定数据点的最新数据. 可同时查询多个设备的数据点最新数据.
     *
     * @param query 要查询的数据点列表
     * @return 数据点的最新数据
     */
    List<LatestData> queryLatest(LatestDataQuery query);

}
```

> **INFO**
>
> `LatestDataQuery` 的构建方式与 Query 类似.

报警服务

报警服务(`warning`)客户端主要包括 `报警规则`、`报警记录` 等接口。 在需要为该服务内的接口配置独立的超时时间时, 可以在 `application.yml` 中配置 `airiot.client.http.services.warning` 节点。

资源对象	客户端类	内容
报警规则	io.github.airiot.sdk.client.service.warning.RuleClient	包括对报警规则操作
报警记录	io.github.airiot.sdk.client.service.warning.WarnClient	包括对报警记录的增删改查等操作

报警规则客户端

```java
/**
 * 告警规则客户端
 */
public interface RuleClient {

    /**
     * 查询报警规则
     *
     * @param query 查询条件
     * @return 报警规则信息
     */
    ResponseDTO<List<Rule>> query(@Nonnull Query query);
}
```

报警记录客户端

```java
/**
 * 告警信息客户端
 */
public interface WarnClient {

    /**
     * 查询告警信息
     *
     * @param query   查询条件
     * @param archive 是否在归档数据中查询
     * @return 告警信息
     */
    ResponseDTO<List<Warning>> query(@Nonnull Query query, String archive);

    /**
     * 根据告警信息ID查询告警信息
     *
     * @param warningId 告警信息ID
     * @param archive
     * @return 告警信息
     */
    ResponseDTO<Warning> queryById(@Nonnull String warningId, String archive);

    /**
     * 创建告警
     *
     * @param warning 告警信息
     * @return 创建结果
     */
    ResponseDTO<InsertResult> create(@Nonnull Warning warning);
}
```

数据源服务

数据源服务(`data-service`)客户端主要为调用项目中已添加的数据源。 在需要为该服务内的接口配置独立的超时时间时, 可以在 `application.yml` 中配置 `airiot.client.http.services.data-service` 节点。

资源对象	客户端类	内容
数据源	io.github.airiot.sdk.client.service.ds.DataServiceClient	调用项目中已添加的数据接口

数据源客户端

```
/**
 * 数据接口客户端，用于调用平台已创建的数据接口
 */
public interface DataServiceClient {

    /**
     * 调用数据接口
     *
     * @param tClass 接口返回值类型
     * @param dsId   接口标识
     * @param params 参数列表，即数据接口中添加的参数，如果没有定义参数则传 {@code null}. key: 参数名，value: 参数值.
     * @return 请求结果
     */
    <T> ResponseDTO<T> call(@Nonnull Class<T> tClass, @Nonnull String dsId, @Nullable Map<String, Object> params);

}
```

其它

客户端 SDK 中提供了一些工具类, 用于简化和辅助开发。

请求上下文

当二次开发的服务需要同时操作多个项目内的资源时, 可以使用请求上下文 (`RequestContext`) 来切换当前请求的项目, 该上下文是线程安全的, 所有数据是通过 `ThreadLocal` 存储的, 不同请求之间相互隔离.
`RequestContext` 主要包括以下方法:

```
```

```
/**
 * 请求上下文
 */
public class RequestContext {

    /**
     * 设置当前请求的项目ID
     * @param projectId 项目ID
     */
    public static void setProjectId(String projectId) {
        // ...
    }

    /**
     * 获取当前已设置的项目ID
     * @return 项目ID
     */
    public static String getProjectId() {
        // ...
    }

    /**
     * 清空当前请求的项目ID
     */
    public static void clearProjectId() {
        // ...
    }
}
```

使用示例:

```
@RestController
public class MyController {

    @GetMapping("/doSomething")
    public void doSomething(HttpServletRequest request) {
        // 获取请求头中的项目ID
        String projectId = request.getHeader("X-Project-Id");
        // 设置当前请求的项目ID
        RequestContext.setProjectId(projectId);
        try {
          // ...
        } finally {
          // 清空当前请求的项目ID
          RequestContext.clearProjectId();
        }
```

```
    }
}
```

查询构造器

客户端接口中的很多查询接口, 其结构比较复杂, 不易构造且容易出错, 为此 SDK 中提供了一个查询构造器 `Query` 来简化查询条件的构造.

查询参数的整体结构如下所示:

```
{
  "project": {},
  "filter": {},
  "sort": {},
  "limit": 30,
  "skip": 20,
  "withCount": true
}
```

字段说明如下:

- **`project`** 查询请求需要返回的字段列表. 例如: `{"id": 1, "name": 1, "address": {"city": 1}}`. `key` 为字段名, `value` 为 `1` 或 `Map`. 如果为为一级字段需要设置为 `1` 例如: `{"id": 1, "name": 1}`, 如果要返回嵌套对象内的字段, 则需要设置为 `Map`, 例如: `{"address": {"city": 1}}`
- **`filter`** 查询条件, 如果没有添加任何条件则查询全部数据. `key` 为字段名, `value` 为过滤的值或逻辑运算符, 例如: `{"name": "Tom", "age": {"$gt": 20, "$lt": 30}}`.
- **`sort`** 排序条件, `key` 为字段名, `value` 为 `1` 表示升序, `-1` 表示降序, 例如: `{"age": 1, "name": -1}`.
- **`limit`** 查询结果的最大数量, 可用于分页查询或限制返回的记录数量.
- **`skip`** 查询结果的偏移量, 即忽略前 N 记录, 可用于分页查询.
- **`withCount`** 是否返回符合条件的记录总数, 如果为 `true` 则会在查询结果记录数量会保存在响应对象 `ResponseDTO<T>` 中的 `count` 字段.

> **INFO**
>
> **注意事项**
>
> 1. 如果查询条件需要使用逻辑或, 可以在 `filter` 中添加 `$or` 字段, 其值为 `Map<String, Object>` 结构与 `filter` 一致, 任一条件成立时表示记录匹配.
> 2. 如果同一字段存在多个逻辑条件, 则需要将多个条件放在一个 `Map` 中, 例如: `{"age": {"$gt": 20, "$lt": 30}}`, 表示查询 `20 < age < 30` 的记录.

逻辑运算符

符号	说明	示例
$not	不相等, 与 `SQL` 中的 `<>` 作用相同	{"age": {"$not": 18}}
$in	在指定列表内, 与 `SQL` 中的 `in` 作用相同	{"id": {"$in": [1,3,4]}}
$nin	不在指定列表内, 与 `SQL` 中的 `not in` 作用相同	{"id": {"$nin": [1,3,4]}}
$gt	大于指定的值, 与 `SQL` 中的 `>` 作用相同	{"age": {"$gt": 18}}
$gte	大于等于指定的值, 与 `SQL` 中的 `>=` 作用相同	{"age": {"$gte": 18}}
$lt	小于指定的值, 与 `SQL` 中的 `<` 作用相同	{"age": {"$lt": 18}}
$lte	小于等于指定的值, 与 `SQL` 中的 `<=` 作用相同	{"age": {"$lte": 18}}
$regex	正则匹配, 与 `SQL` 中的 `like` 相似	{"name": {"$regex": "张"}}

```
/**
 * 查询构造器
 */
public class Query {

    public static class Builder {

        /**
         * 构造过滤条件
         */
        public FilterBuilder filter() {
            // ...
        }

        /**
         * 构造逻辑或过滤条件
         */
        public FilterBuilder or() {
            // ...
        }

        /**
```

```
         * 添加要返回的字段列表
         *
         * @see #select(Collection)
         */
        public Builder select(String... fields) {
            // ...
        }

        /**
         * 添加要返回的字段列表
         *
         * @see #select(Collection)
         */
        public <T> Builder select(SFunction<T, ?>... columns) {
            // ...
        }

        /**
         * 获取指定类型中定义的全部字段
         * <br>
         * 如果字段被 static 和 transient 修饰则会跳过. 如果字段上带有 {@link Field} 则使用该注解定义的名称
         *
         * @param tClass 类型
         */
        public <T> Builder select(Class<T> tClass) {
            // ...
        }

        /**
         * 设置查询返回的字段列表
         *
         * @param fields 字段名称列表
         */
        public Builder select(Collection<String> fields) {
            // ...
        }

        /**
         * 设置查询返回的嵌套对象内的字段列表
         *
         * @see #selectSubFields(String, Map)
         */
        public <T> Builder selectSubFields(SFunction<T, ?> column, String... subFields) {
            // ...
        }

        /**
         * 设置查询返回的嵌套对象内的字段列表
         *
```

```
         * @see #selectSubFields(String, Map)
         */
        public <T> Builder selectSubFields(String field, String... subFields) {
            // ...
        }

        /**
         * 设置查询返回的嵌套对象内的字段列表
         *
         * @see #selectSubFields(String, Map)
         */
        public <T> Builder selectSubFields(SFunction<T, ?> column, Map<String, Object>
subFields) {
            // ...
        }

        /**
         * 设置查询返回的嵌套对象内的字段列表
         *
         * @param field     字段名称
         * @param subFields 嵌套对象内的字段列表
         */
        public Builder selectSubFields(String field, Map<String, Object> subFields) {
            // ...
        }

        /**
         * 排除要查询的字段列表，即查询结果中不返回该字段
         * <br>
         * 通常情况下，在使用 {@link #select(Class)} 添加所有字段后，再使用该方法排除不需要的字段
         *
         * @param columns 要排除的字段名称列表
         */
        public Builder exclude(Collection<String> columns) {
            // ...
        }

        /**
         * 排除要查询的字段列表
         *
         * @see #exclude(Collection)
         */
        public Builder exclude(String... columns) {
            // ...
        }

        /**
         * 排除要查询的字段列表
         *
         * @see #exclude(Collection)
```

```
     */
    public <T> Builder exclude(SFunction<T, ?>... columns) {
        // ...
    }

    /**
     * 汇总查询字段
     *
     * @param field 字段名
     */
    public AggregateBuilder summary(String field) {
        // ...
    }

    /**
     * 汇总查询字段
     *
     * @param column 列
     */
    public <T> AggregateBuilder summary(SFunction<T, ?> column) {
        // ...
    }

    /**
     * 分组
     *
     * @param field 字段名
     */
    public GroupByBuilder groupBy(String field) {
        // ...
    }

    /**
     * 分组汇总
     */
    protected Builder groupBy(Map<String, Object> field) {
        // ...
    }

    /**
     * 分组汇总
     *
     * @param column 列
     */
    public <T> GroupByBuilder groupBy(SFunction<T, ?> column) {
       // ...
    }


    /**
```

```
     * 添加升序字段
     *
     * @param fields 字段名称列表
     */
    public Builder orderAsc(Collection<String> fields) {
        // ...
    }

    /**
     * 添加升序字段
     *
     * @see #orderAsc(Collection)
     */
    public Builder orderAsc(String... fields) {
        // ...
    }

    /**
     * 添加升序字段
     *
     * @see #orderAsc(Collection)
     */
    public <T> Builder orderAsc(SFunction<T, ?>... fields) {
        // ...
    }


    /**
     * 添加降序字段
     *
     * @see #orderDesc(Collection)
     */
    public Builder orderDesc(String... fields) {
        // ...
    }

    /**
     * 添加降序字段
     *
     * @see #orderDesc(Collection)
     */
    public <T> Builder orderDesc(SFunction<T, ?>... fields) {
        // ...
    }

    /**
     * 添加降序字段
     *
     * @param fields 字段名称列表
     */
```

```
        public Builder orderDesc(Collection<String> fields) {
            // ...
        }

        /**
         * 设置跳过的记录数，用于分页
         *
         * @param skip 跳过数量
         */
        public Builder skip(int skip) {
            // ...
        }

        /**
         * 设置返回的记录数
         *
         * @param limit 返回的记录数量
         */
        public Builder limit(int limit) {
            // ...
        }

        /**
         * 返回总记录数
         */
        public Builder withCount() {
            // ...
        }

        /**
         * 完成构建并返回查询以象
         */
        public Query build() {
            // ...
        }
    }
}
```

通过 `filter()` 方法创建过滤条件构造器, 或通过 `or()` 方法创建逻辑或过滤条件, 在过滤条件构造完成后需要调用 `end()` 方法完成构造. 构造器如下所示:

```
public class FilterBuilder {
    /**
     * 结束查询条件构建，并返回查询构建器
     */
    public Query.Builder end() {
        // ...
    }
```

```java
    /**
     * 等于
     *
     * @param field 字段名
     * @param value 字段值
     */
    public <T> FilterBuilder eq(String field, T value) {
        // ...
    }

    public <Type, T> FilterBuilder eq(SFunction<Type, ?> column, T value) {
        // ...
    }

    /**
     * 不等于
     *
     * @param field 字段名
     * @param value 字段值
     */
    public <T> FilterBuilder ne(String field, T value) {
        // ...
    }

    /**
     * 不等于
     *
     * @see #ne(String, Object)
     */
    public <Type, T> FilterBuilder ne(SFunction<Type, ?> column, T value) {
        // ...
    }

    /**
     * 在指定列表之内
     *
     * @see #in(String, Object[])
     */
    public <T> FilterBuilder in(String field, T... value) {
        // ...
    }

    /**
     * 在指定列表之内
     *
     * @see #in(String, Object[])
     */
    public <Type, T> FilterBuilder in(SFunction<Type, ?> column, T value) {
        // ...
```

```
    }

    /**
     * 在指定列表之内
     *
     * @param field 字段名
     * @param value 字段值列表
     */
    public <T> FilterBuilder in(String field, Collection<T> value) {
        // ...
    }

    /**
     * 不在指定列表之内
     *
     * @see #notIn(String, Object...)
     */
    public <T> FilterBuilder notIn(String field, T... values) {
        // ...
    }

    /**
     * 不在指定列表之内
     *
     * @see #notIn(String, Object...)
     */
    public <Type, T> FilterBuilder notIn(SFunction<Type, ?> column, T... values) {
        // ...
    }

    /**
     * 不在指定列表之内
     *
     * @param field  字段名
     * @param values 字段值列表
     */
    public <T> FilterBuilder notIn(String field, Collection<T> values) {
        // ...
    }

    /**
     * 正则表达式匹配
     *
     * @see #regex(String, String)
     */
    public <Type> FilterBuilder regex(SFunction<Type, ?> column, String regex) {
        // ...
    }

    /**
```

```java
     * 正则表达式匹配
     *
     * @param field 字段名
     * @param regex 正则表达式
     */
    public FilterBuilder regex(String field, String regex) {
        // ...
    }

    /**
     * 小于
     *
     * @param field 字段名
     * @param value 字段值
     */
    public FilterBuilder lt(String field, Object value) {
        // ...
    }

    /**
     * 小于
     *
     * @see #lt(String, Object)
     */
    public <T> FilterBuilder lt(SFunction<T, ?> column, Object value) {
        // ...
    }

    /**
     * 小于或等于
     *
     * @param field 字段名
     * @param value 字段值
     */
    public FilterBuilder lte(String field, Object value) {
        // ...
    }

    /**
     * 小于或等于
     *
     * @see #lte(String, Object)
     */
    public <T> FilterBuilder lte(SFunction<T, ?> column, Object value) {
        // ...
    }

    /**
     * 大于
     *
```

```
     * @param field 字段名
     * @param value 字段值
     */
    public FilterBuilder gt(String field, Object value) {
        // ...
    }

    /**
     * 大于
     *
     * @see #gt(String, Object)
     */
    public <T> FilterBuilder gt(SFunction<T, ?> column, Object value) {
        // ...
    }

    /**
     * 大于或等于
     *
     * @param field 字段名
     * @param value 字段值
     */
    public FilterBuilder gte(String field, Object value) {
        // ...
    }

    /**
     * 大于或等于
     *
     * @see #gte(String, Object)
     */
    public <T> FilterBuilder gte(SFunction<T, ?> column, Object value) {
        // ...
    }

    /**
     * 在指定取值范围之内，左闭右闭。即：[minValue, maxValue].
     *
     * <pre>
     *      例如:
     *      // 查询年龄在 15 - 30 岁之间的用户。包括 30 岁
     *      Query query = Query.newBuilder()
     *            .select(User.class)
     *            .between("age", 15, 30)
     *            .build();
     *
     *      // 查询 2020 年新注册的用户数
     *      Query query = Query.newBuilder()
     *            .groupField("count(id) as count")
     *            .between("createTime", "2020-01-01 00:00:00", "2020-12-31 23:59:59")
```

```
     *          .build();
     * </pre>
     *
     * @param field    字段名
     * @param minValue 最小值. 包含最小值
     * @param maxValue 最大值. 包含最大值
     */
    public FilterBuilder between(String field, Object minValue, Object maxValue) {
        // ...
    }

    /**
     * 在指定取值范围之内, 包含最小值和最大值. 即: [minValue, maxValue].
     *
     * @see #between(String, Object, Object)
     */
    public <T> FilterBuilder between(SFunction<T, ?> column, Object minValue, Object
maxValue) {
        // ...
    }

    /**
     * 在指定取值范围之内, 左开右闭. 即: (minValue, maxValue].
     *
     * <pre>
     *     例如:
     *     // 查询年龄在 15 - 30 岁之间的用户. 不包括 15 岁
     *     Query query = Query.newBuilder()
     *          .select(User.class)
     *          .between("age", 15, 30)
     *          .build();
     *
     *     // 查询 2020 年新注册的用户数
     *     Query query = Query.newBuilder()
     *          .groupField("count(id) as count")
     *          .between("createTime", "2020-01-01 00:00:00", "2020-12-31 23:59:59")
     *          .build();
     * </pre>
     *
     * @param field    字段名
     * @param minValue 最小值. 不包含最小值
     * @param maxValue 最大值. 包含最大值
     */
    public FilterBuilder betweenExcludeLeft(String field, Object minValue, Object maxValue)
{
        // ...
    }

    /**
     * 在指定取值范围之内, 左开右闭. 即: (minValue, maxValue].
```

```
     *
     * @see #betweenExcludeLeft(String, Object, Object)
     */
    public <T> FilterBuilder betweenExcludeLeft(SFunction<T, ?> column, Object minValue,
Object maxValue) {
        // ...
    }

    /**
     * 在指定取值范围之内, 左闭右开. 即: [minValue, maxValue).
     *
     * <pre>
     *     例如:
     *     // 查询年龄在 15 - 30 岁之间的用户. 不包括 30 岁
     *     Query query = Query.newBuilder()
     *          .select(User.class)
     *          .between("age", 15, 30)
     *          .build();
     *
     *     // 查询 2020 年新注册的用户数
     *     Query query = Query.newBuilder()
     *          .groupField("count(id) as count")
     *          .between("createTime", "2020-01-01 00:00:00", "2020-12-31 23:59:59")
     *          .build();
     * </pre>
     *
     * @param field    字段名
     * @param minValue 最小值. 包含最小值
     * @param maxValue 最大值. 不包含最大值
     */
    public FilterBuilder betweenExcludeRight(String field, Object minValue, Object maxValue)
{
        // ...
    }

    /**
     * 在指定取值范围之内, 左闭右开. 即: [minValue, maxValue).
     *
     * @see #betweenExcludeRight(String, Object, Object)
     */
    public <T> FilterBuilder betweenExcludeRight(SFunction<T, ?> column, Object minValue,
Object maxValue) {
        // ...
    }


    /**
     * 在指定取值范围之内, 左开右开. 即: (minValue, maxValue).
     *
     * <pre>
```

```
     *     例如:
     *     // 查询年龄在 15 - 30 岁之间的用户. 不包括 15 和 30 岁
     *     Query query = Query.newBuilder()
     *          .select(User.class)
     *          .between("age", 15, 30)
     *          .build();
     *
     *     // 查询 2020 年新注册的用户数
     *     Query query = Query.newBuilder()
     *          .groupField("count(id) as count")
     *          .between("createTime", "2020-01-01 00:00:00", "2020-12-31 23:59:59")
     *          .build();
     * </pre>
     *
     * @param field    字段名
     * @param minValue 最小值. 不包含最小值
     * @param maxValue 最大值. 不包含最大值
     */
    public FilterBuilder betweenExcludeAll(String field, Object minValue, Object maxValue) {
        // ...
    }

    /**
     * 在指定取值范围之内, 左开右开. 即: (minValue, maxValue).
     *
     * @see #betweenExcludeAll(String, Object, Object)
     */
    public <T> FilterBuilder betweenExcludeAll(SFunction<T, ?> column, Object minValue,
Object maxValue) {
        // ...
    }
}
```

> **INFO**
>
> `select(Class)` 方法, 会通过反射获取该类型中所有的字段并作为查询字段, 如果字段被 `static` 和 `transient` 修饰则会跳过. 如果字段上带有 `@Field` 则使用该注解定义的名称. 也可以在 `select(Class)` 的基础上再使用 `exclude` 方法排除不需要的字段.

示例

示例数据如下所示:

```
{
 "project": {
  "name": 1,
```

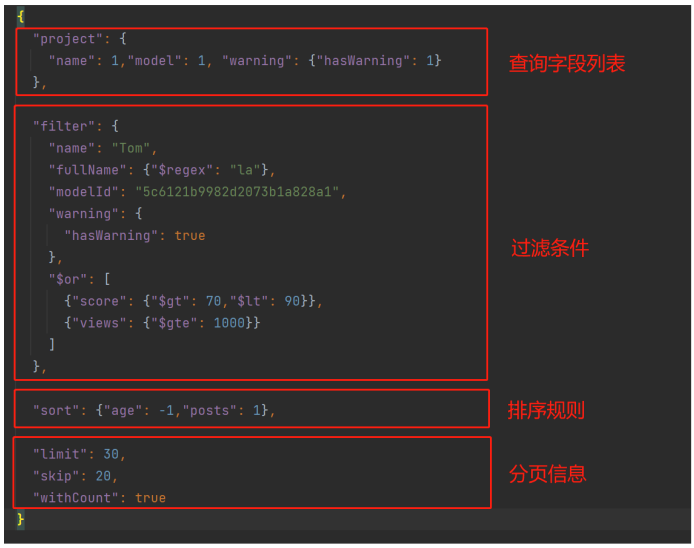
```
  "model": 1,
  "warning": {
   "hasWarning": 1
  }
 },
 "filter": {
  "name": "Tom",
  "fullname": {
   "$regex": "la"
  },
  "modelId": "5c6121b9982d2073b1a828a1",
  "warning": {
   "hasWarning": true
  },
  "$or": [{
   "score": {
    "$gt": 70,
    "$lt": 90
   }
  }, {
   "views": {
    "$gte": 1000
   }
  }]
 },
 "sort": {
  "age": -1,
  "posts": 1
 },
 "limit": 30,
 "skip": 20,
 "withCount": true
}
```

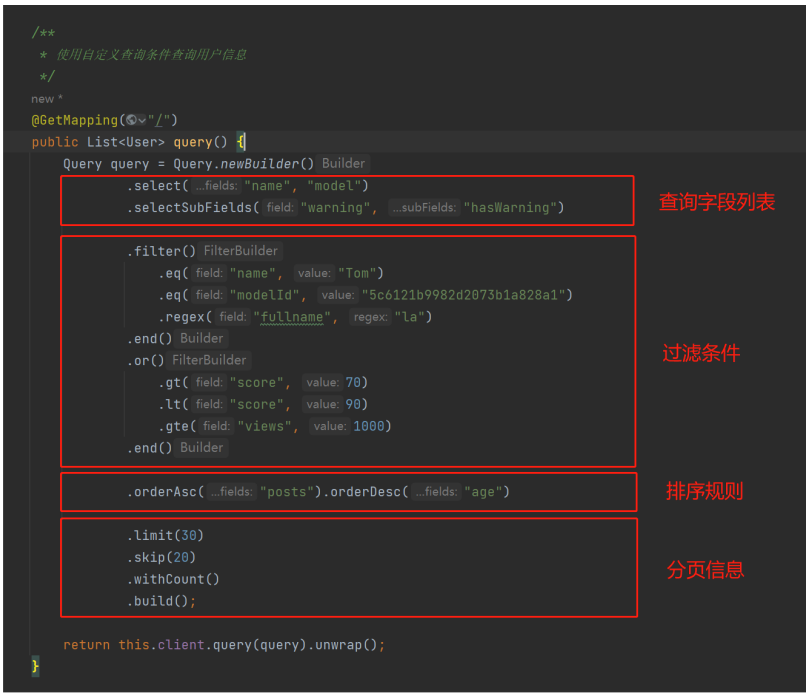
使用构造器创建上述查询条件的代码如下所示:

```
public class Demo {

    public static void main(String[] args) {
        Query query = Query.newBuilder()
                .select("name", "model")
                .selectSubFields("warning", "hasWarning")
                .filter()
                .eq("name", "Tom")
                .eq("modelId", "5c6121b9982d2073b1a828a1")
                .regex("fullname", "la")
                .end()
                .or()
```

```
                .gt("score", 70)
                .lt("score", 90)
                .gte("views", 1000)
                .end()
                .orderAsc("posts").orderDesc("age")
                .limit(30)
                .skip(20)
                .withCount()
                .build();
    }
}
```

对照图





常见问题

数据接入驱动开发常见问题

接收到配置信息为 null

现象描述

在 `DriverApp.start(Config)` 方法中, 接收到的驱动配置存在部分字段的值为 `null`.

解决办法

1. 检查驱动配置中是否该字段是否提供了有效值.
2. 检查驱动配置类(例如: Settings) 中对应的字段的名称与 schema 中的字段名称是否一致.
3. 检查驱动配置类(例如: Settings) 中对应的字段是否提供了 `setter` 方法.

MQTT不断的连接成功后立即断开

现象描述

在日志中不断的输出 `MQTTDataSender: 已连接` 和 `MQTTDataSender: 已断开` 的内容.

原因

有多个驱动程序使用了相同的驱动实例ID.

解决办法

更换驱动的实例ID, 保证每个驱动实例ID对应一个程序即可.

驱动调用错误

现象描述

在编辑驱动实例或表的配置时, 提示 `驱动调用错误`. 通过 `F12` 看到详细错误信息为 `未找到运行的驱动服务`.

原因

1. 该项目没有运行该类型驱动或驱动被停止.

2. 驱动配置中的 `airiot.driver.id` 与 `service.yml` 的 `Name` 字段的值不一致.

解决办法

原因1: 启动或重新安装该驱动. 原因2: 修改驱动配置中的 `airiot.driver.id` 或 `service.yml` 的 `Name` 字段的值, 使两者保持一致.

算法服务开发

本文将会详细介绍如何使用 `Java SDK` 开发算法服务, 实现算法集成. 示例项目目上传至 https://github.com/air-iot/sdk-java-examples/tree/master/algorithm-example.

介绍

`算法服务` 是扩展或将已有 `算法` 集成到平台的一种方式. 在平台现有的功能不满足需求时, 可以通过开发算法服务来实现自定义的功能.

开发步骤

1. 创建项目

该过程同 数据接入驱动开发-创建项目 中的创建项目过程一致.

2. 引入SDK

在 `pom.xml` 中引入依赖:

```
<dependency>
    <groupId>io.github.air-iot</groupId>
    <artifactId>sdk-algorithm-starter</artifactId>
    <version>4.x.x</version>
</dependency>
```

3. 定义 schema

`schema` 定义描述了该算法程序支持哪些算法函数, 以及每个算法函数的输入和输出参数定义. 每个算法程序可以包含多个算法函数, 每个算法函数的名称必须唯一, 详细说明参考算法schema说明. 示例如下:

```
[
    {
        "title": "函数1-加法",
        "function": "add",
        "input": {
            "type": "object",
            "properties": {
```

```
                "num1": {
                    "title": "参数1",
                    "type": "number"
                },
                "num2": {
                    "title": "参数2",
                    "type": "number"
                }
            },
            "required": [
                "num1",
                "num2"
            ]
        },
        "output": {
            "type": "object",
            "properties": {
                "num1": {
                    "title": "结果",
                    "type": "number"
                }
            }
        }
    },
    {
        "title": "函数2-绝对值",
        "function": "abs",
        "input": {
            "type": "object",
            "properties": {
                "num1": {
                    "title": "参数1",
                    "type": "number"
                }
            },
            "required": [
                "num1"
            ]
        },
        "output": {
            "type": "object",
            "properties": {
                "res": {
                    "title": "结果",
                    "type": "number"
                }
            }
        }
    },
    {
```

```
        "title": "函数3-获取当前系统时间",
        "function": "now",
        "input": {
            "type": "object",
            "properties": {},
            "required": []
        },
        "output": {
            "type": "object",
            "properties": {
                "sysdate": {
                    "title": "当前系统时间",
                    "type": "string"
                }
            }
        }
    },
    {
        "title": "函数4-接收Map",
        "function": "recvMap",
        "input": {
            "type": "object",
            "properties": {
                "name": {
                    "title": "姓名",
                    "type": "string"
                }
            },
            "required": ["name"]
        },
        "output": {
            "type": "object",
            "properties": {
                "result": {
                    "title": "输出结果",
                    "type": "string"
                }
            }
        }
    },
    {
        "title": "函数4-接收字符串",
        "function": "recvString",
        "input": {
            "type": "object",
            "properties": {
                "name": {
                    "title": "姓名",
                    "type": "string"
                }
```

```
            },
            "required": ["name"]
        },
        "output": {
            "type": "object",
            "properties": {
                "result": {
                    "title": "输出结果",
                    "type": "string"
                }
            }
        }
    }
]
```

4. 实现算法接口

`SDK` 中定义了 `算法服务接口`, 开发者需要实现这个接口.

```java
/**
 * 算法服务接口定义. 该接口定义了算法应用的生命周期方法, 以及算法函数的执行方法. <br>
 * <b>注: 该接口的实现必须注入到 Spring IoC 容器中, 否则 SDK 无法识别.</b>
 * <br>
 * 可以在该接口的实现类中, 使用 {@link AlgorithmFunction} 注解定义算法函数.
 * 也可以不使用该注解, 而是在 {@link #run(String, String, Map)} 方法中根据 {@code function} 参数的值, 执行对应的算法函数.
 */
public interface AlgorithmApp {

    /**
     * 当算法服务启动时, 会调用此方法.
     * <br>
     * 如果程序需要在算法服务启动时执行一些初始化操作, 可以重写该方法.
     */
    default void start() {

    }

    /**
     * 当算法服务停止时, 会调用此方法
     * <br>
     * 如果程序需要在算法服务停止时执行一些清理工作, 可以重写该方法.
     */
    default void stop() {

    }
```

```
    /**
     * 获取算法的 schema 定义信息
     */
    String schema();

    /**
     * 执行算法. 如果在当前类型中没有找到使用 {@link AlgorithmFunction} 定义方法, 则会调用此方法
     *
     * @param projectId 发起请求的项目ID
     * @param function  函数名
     * @param params    请求参数
     * @return 算法执行结果
     * @throws AlgorithmException 算法执行异常, 该异常消息会作为错误信息返回给调用方
     */
    default Object run(String projectId, String function, Map<String, Object> params) throws
AlgorithmException {
        throw new IllegalArgumentException("未实现的算法 '" + function + "'");
    }
}
```

AlgorithmFunction 注解

`SDK` 提供了 `@AlgorithmFunction` 注解, 用于标识实现类中的方法对应 schema 的算法函数. 该注解只能用在方法上. 使用该注解修饰的方法, 必须满足以下条件:

- 必须为 `public` 方法.
- 必须带有 1 或 2 个参数, 且第 1 个参数必须为 `String projectId` 用于接收发起请求的项目ID.
- 如果该算法函数有参数定义, 必须放在第 2 个参数位置, 且类型只为 `String`, `Map<String, Object>` 或自定义类型.
- 该函数的返回值. 不能是基本类型, 只能是 `Map<String, Object>` 或自定义类型.

算法调用请求执行流程

1. 当平台调用该算法服务时, `SDK` 会解析请求中的 `function` 信息.
2. 根据 `function` 的值, 在实现类中查找是否有使用 `@AlgorithmFunction("算法函数名")` 注解修饰的方法. 如果找到了, 则会调用该方法, 并将方法的返回值作为算法执行结果返回给平台.
3. 如果没有找到使用 `@AlgorithmFunction("算法函数名")` 修饰的方法时, 则会调用 `run` 方法, 并将 `function` 参数的值作为 `run` 方法的第二个参数传入, 由开发者根据 `function` 参数值执行对应的算法逻辑. 最后将 `run` 方法的返回值作为算法执行结果返回给平台.

示例

```java
public class MyAlgorithm implements AlgorithmApp {

    /**
     * 算法函数1，与 schema 中的 algorithm1 对应.
     */
    @AlgorithmFunction("algorithm1")
    public Map<String, Object> algorithm1(String projectId, Map<String, Object> params) {
        // 自定义算法函数1
    }

    /**
     * 算法函数2，与 schema 中的 algorithm2 对应. 该方法接收一个自定义类型的参数，并且返回自定义类型对象.
     */
    @AlgorithmFunction("algorithm2")
    public MyResult algorithm2(String projectId, MyParams params) {
        // 自定义算法函数2
    }

    /**
     * 算法函数3，与 schema 中的 algorithm3 对应. 该方法不接收任何参数.
     */
    @AlgorithmFunction("algorithm3")
    public Map<String, Object> algorithm3(String projectId) {
        // 自定义算法函数3
    }

    /**
     * 算法函数4，与 schema 中的 algorithm4 对应. 该方法接收一个 String 类型的参数.
     */
    @AlgorithmFunction("algorithm4")
    public Map<String, Object> algorithm4(String projectId, String params) {
        // 自定义算法函数4
    }

    /**
     * 默认执行函数. 例如: 当 function 为 algorithm5 时，会调用此方法.
     */
    @Override
    public Object run(String projectId, String function, Map<String, Object> params) throws AlgorithmException {
        if("algorithm5".equals(function)) {
            // 自定义算法函数5
        } else if("algorithm6".equals(function)) {
            // 自定义算法函数6
        } else {
            throw new IllegalArgumentException("未实现的算法 '" + function + "'");
        }
```

```
    }
}
```

5. 算法配置信息

算法配置信息定义了算法的基本信息以及平台算法服务的连接信息. 整体配置如下:

```yaml
algorithm:
  id: 自定义算法标识              # 必填
  name: 自定义算法名称            # 必填
  max-threads: 10                # 可选. 最大线程数, 默认: 0, 表示取 CPU 核心数
algorithm-grpc:      # 平台算法服务连接配置
  host: localhost    # 必填. 算法服务地址
  port: 9236         # 必填. 算法服务端口, 默认: 9236
```

6. 打包

具体打包步骤请参考 流程插件开发-打包 中的步骤.

windows系统打包发布时的算法配置

```yaml
algorithm:
  id: 自定义算法标识
  name: 自定义算法名称
  max-threads: 10
algorithm-grpc:
  host: 127.0.0.1
  port: 9236
```

linux系统打包发布时的算法配置

```yaml
algorithm:
  id: 自定义算法标识
  name: 自定义算法名称
  max-threads: 10
algorithm-grpc:
  host: algorithm
  port: 9236
```

7. 部署

具体部署步骤请参考 流程插件开发-部署 中的步骤.

Node SDK 介绍

Node SDK 二次开发说明

数据接入驱动开发

介绍如何使用 Node SDK 开发自定义数据接入驱动

流程插件开发

介绍如何使用 Node SDK 开发自定义流程插件

流程扩展节点接入

介绍如何使用 Node SDK 开发自定义流程扩展节点接入服务

算法服务开发

介绍如何使用 Node SDK 开发算法服务

平台接口客户端

Node SDK 平台接口客户端

打包部署

本文将详细介绍平台打包及部署.

Node SDK 介绍

`Node SDK` 是 AIRIOT 物联网平台提供的 `Node` 语言的二次开发工具包, 可使用 Node SDK 调用平台开放的接口和实现对平台功能的扩展。

接下来会分别介绍如何使用 `Node SDK` 数据接入驱动、流程插件、算法集成和通过平台接口客户端实现系统集成。

内容说明

内容	用途
数据接入驱动	实现从设备或其它平台系统采集数据
流程插件	扩展流程引擎中的节点
流程扩展节点接入	开发流程扩展节点接入服务
算法集成	扩展自定义算法
平台接口客户端	调用平台开放的API接口

使用方式

`Node SDK` 包已经上传npm管理器，直接使用npm安装SDK的开发包：

```
npm install @airiot/sdk-nodejs
```

版本说明

SDK 版本	Node 版本
4.1.x	18+

示例仓库

示例代码: https://github.com/air-iot/sdk-node-examples

数据接入驱动开发

本文将会详细介绍如何使用 `Node SDK` 开发自定义数据接入驱动.

介绍

`数据接入驱动` 是为实现从不同的协议、设备或其它平台系统采集数据而开发的特定程序. 每个 `数据接入驱动` 程序需要根据协议的特点实现数据采集功能，然后将采集到的数据通过 `Node SDK` 提供的接口发送到平台.

`Node SDK` 提供了 `数据接入驱动` 开发的相关内容. 包括驱动的接口定义, 以及与平台交互功能. 开发者只需要实现接口中的方法.

开发步骤

1. 创建项目

1. 创建目录
2. 进入项目执行初始化

```
npm init
```

3. 创建index.js文件

2. 引入SDK

```
const {App, Driver} = require('@airiot/sdk-nodejs/driver')
```

3. 定义schema

`数据接入驱动` 需要定义一个 `schema` 用于描述驱动的配置信息. `schema` 是一个类似于 `json` 格式的对象, 详细格式说明见 数据接入驱动schema说明. 以下是一个简单的示例:

```
({
  "driver": {
    "properties": {
      "settings": {
```

```
        "title": "实例配置",
        "type": "object",
        "properties": {
          "server": {
            "type": "string",
            "title": "服务器",
            "descripption": "MQTT 服务器地址. 例如: tcp://127.0.0.1:1883"
          },
          "username": {
            "type": "string",
            "title": "用户名",
          },
          "password": {
            "type": "string",
            "title": "密码",
            "fieldType": "password"
          },
          "clientId": {
            "type": "string",
            "title": "客户端ID"
          },
          "topic": {
            "type": "string",
            "title": "主题",
            "descripption": "接收数据的主题. 例如: /data/#"
          },
          "parseScript": {
            "type": "string",
            "title": "数据处理脚本",
            "fieldType": "deviceScriptEdit",
            "description": "消息处理脚本. 函数名必须为 'handler'",
            "defaultScript": "/**\n" +
              " * 数据处理脚本, 处理从 mqtt 接收到的数据.\n" +
              " *\n" +
              " * @param {string} topic 消息主题\n" +
              " * @param {string} message 消息内容\n" +
              " * @return 消息解析结果\n" +
              " */\n" +
              "function handler(topic, message) {\n" +
              "\t\n" +
              "\t// 脚本返回值必须为对象数组\n" +
              "\t// \tid: 资产编号\n" +
              "\t//\ttime: 时间戳(毫秒)\n" +
              "\t//  fields: 数据点数据. 该字段为 JSON 对象, key 为数据点标识, value 为数据点的值
\n" +
              "\treturn [\n" +
              "\t\t{\"table\": \"T10001\", \"id\": \"SN10001\", \"time\": new
Date().getTime(), \"fields\": {\"key1\": \"this is a string value\", \"key2\": true,
\"key3\": 123.456}}\n" +
              "\t];\n" +
```

```
              "}"
          },
          "commandScript": {
            "type": "string",
            "title": "指令处理脚本",
            "fieldType": "deviceScriptEdit",
            "description": "指令处理脚本. 函数名必须为 'handler'",
            "defaultScript": "/**\n" +
              " * 指令处理脚本. 发送指令时会将指令内容传递给脚本，然后由指定返回最终要发送的信
息.\n" +
              " *\n" +
              " * @param {string} 工作表标识\n" +
              " * @param {string} 资产编号\n" +
              " * @param {object} 命令内容\n" +
              " * @return {object} 最终要发送的消息，及目标 topic\n" +
              " */\n" +
              "function handler(tableId, deviceId, command) {\n" +
              "\t\n" +
              "\t// 脚本返回值必须为下面对象结构\n" +
              "\t//\t\ttopic: 消息发送的目标 topic\n" +
              "\t//\t\tpayload: 消息内容\n" +
              "\treturn {\n" +
              "\t\t\"topic\": \"cmd/\" + deviceId,\n" +
              "\t\t\"payload\": \"发送内容\"\n" +
              "\t};\n" +
              "}"
          },
          "network": {
            "type": "object",
            "title": "通讯监控参数",
            "properties": {
              "timeout": {
                "title": "通讯超时时间(s)",
                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故障",
                "type": "number"
              }
            }
          }
        },
        "required": ["server", "username", "password", "topic", "parseScript",
"commandScript"]
      }
    }
  },
  "model": {
    "properties": {
      "settings": {
        "title": "模型配置",
        "type": "object",
        "properties": {
```

```
      "network": {
        "type": "object",
        "title": "通讯监控参数",
        "properties": {
          "timeout": {
            "title": "通讯超时时间(s)",
            "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故障",
            "type": "number"
          }
        }
      }
    }
  },
  "tags": {
    "title": "数据点",
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "id": {
          "type": "string",
          "title": "标识",
          "description": "数据点的标识，用于在数据点列表中唯一标识数据点"
        },
        "name": {
          "type": "string",
          "title": "名称",
          "description": "数据点的名称"
        }
      },
      "required": ["id", "name"]
    }
  },
  "commands": {
    "title": "命令",
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "title": "名称"
        },
        "ops": {
          "type": "array",
          "title": "指令",
          "items": {
            "type": "object",
            "properties": {
              "topic": {
```

```
                  "type": "string",
                  "title": "主题",
                  "description": "发送消息的主题. 例如: /cmd/control",
                },
                "message": {
                  "type": "string",
                  "title": "消息",
                  "description": "发送的消息. 例如: {\"cmd\":\"start\"}",
                },
                "qos": {
                  "type": "number",
                  "title": "QoS",
                  "description": "消息质量. 0,1,2",
                  "enum": [0, 1, 2],
                  "enum_title": ["QoS0", "QoS1", "QoS2"]
                },
              },
              "required": ["name", "message"]
            }
          }
        }
      }
    }
  },
  "device": {
    "properties": {
      "settings": {
        "title": "设备配置",
        "type": "object",
        "properties": {
          "customDeviceId": {
            "type": "string",
            "title": "设备编号",
            "descripption": "自定义设备编号. 如果未定义则使用平台中的资产编号"
          },
          "network": {
            "type": "object",
            "title": "通讯监控参数",
            "properties": {
              "timeout": {
                "title": "通讯超时时间(s)",
                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故障",
                "type": "number"
              }
            }
          }
        },
        "required": []
      }
```

```
    }
  }
})
```

4. 根据schema返回对应配置

在上一步骤中, 通过 `schema` 定义了驱动的相关配置, 包括 `驱动配置`、`数据点配置` 和 `指令配置`.

整体格式说明

驱动从平台接收到的配置信息整体格式如下:

```
{
  "id": "nodedemo",
  "name": "NodeSDKDemo",
  "driverType": "node-demo",
  "device": {
    "commands": [],
    "settings": {
      "clientId": "node-demo",
      "commandScript": "function handler(tableId, deviceId, command) {\n    return
{\"topic\": \"cmd/\" + tableId + \"/\" + deviceId, \"payload\":
JSON.stringify(command.params)}\n  }",
      "parseScript": "function handler(topic, message) {\n    let arr =
JSON.parse(message.toString())\n    let topics = topic.split(\"/\");\n    let field = {}\n
arr.forEach(ele => {\n      field[ele.key] = ele.value\n    })\n    // 脚本返回值必须为对象数组
\n    // \tid: 资产编号\n    //\ttime: 时间戳(毫秒)\n    //  fields: 数据点数据. 该字段为 JSON
对象, key 为数据点标识, value 为数据点的值\n    return [\n      {\"table\": topics[1], \"id\":
topics[2], \"time\": new Date().getTime(), \"fields\": field}\n    ]\n}",
      "password": "public",
      "server": "mqtt://localhost:1883",
      "topic": "test/#",
      "username": "admin"
    }
  },
  "tables": [{
    "id": "nodedriverdemo",
    "device": {
      "driver": "node-demo",
      "groupId": "nodedemo",
      "settings": {
        "clientId": "node-deemo",
        "commandScript": "/**\n * 指令处理脚本. 发送指令时会将指令内容传递给脚本, 然后由指定返回最
终要发送的信息.\n *\n * @param {string} 工作表标识\n * @param {string} 资产编号\n * @param
{object} 命令内容\n * @return {object} 最终要发送的消息, 及目标 topic\n */\nfunction
handler(tableId, deviceId, command) {\n\t// 脚本返回值必须为下面对象结构\n\t//\t\ttopic: 消息发
送的目标 topic\n\t//\t\tpayload: 消息内容\n\treturn {\n\t\t\"topic\": \"cmd/\" +
```

```
deviceId,\n\t\t\"payload\": \"发送内容\"\n\t};\n}",
        "parseScript": "/**\n * 数据处理脚本，处理从 mqtt 接收到的数据.\n *\n * @param {string} topic 消息主题\n * @param {string} message 消息内容\n * @return 消息解析结果\n */\nfunction handler(topic, message) {\n\t// 脚本返回值必须为对象数组\n\t// \tid: 资产编号\n\t//\ttime: 时间戳(毫秒)\n\t//  fields: 数据点数据. 该字段为 JSON 对象，key 为数据点标识，value 为数据点的值\n\treturn [\n\t\t{\"id\": \"SN10001\", \"time\": new Date().getTime(), \"fields\": {\"key1\": \"this is a string value\", \"key2\": true, \"key3\": 123.456}}\n\t];\n}",
        "password": "public",
        "server": "mqtt://localhost:1883",
        "topic": "test/#",
        "username": "admin"
      },
      "tags": [{
        "id": "p1",
        "name": "数据点1"
      }],
      "commands": [{
        "defaultValue": {
          "c1": 2
        },
        "form": [{
          "arrayValue": null,
          "defaultValue": {
            "default": 2
          },
          "ifRepeat": null,
          "ioway": "默认写入",
          "mod": null,
          "name": "c1",
          "objectValue": null,
          "objectValue2": null,
          "select": null,
          "select2": null,
          "tableValue": null,
          "tableValue2": null,
          "tag": null,
          "tagValue": null,
          "type": "number"
        }],
        "name": "c1",
        "ops": [{
          "param": "c1"
        }],
        "showName": "指令1",
        "tag": null,
        "writeOut": {
          "arrayValue": null,
          "defaultValue": null,
          "ifRepeat": null,
          "mod": null,
```

```
          "objectValue": null,
          "objectValue2": null,
          "select": null,
          "select2": null,
          "tableValue": null,
          "tableValue2": null,
          "tag": null,
          "tagValue": null
        }
      }],
      "events": null
    },
    "devices": [{
      "id": "nodesdk1",
      "name": "nodesdk1",
      "device": {
        "driver": "",
        "groupId": "",
        "settings": null,
        "tags": null,
        "commands": null,
        "events": null
      },
      "disable": false,
      "off": false
    }]
  }]
}
```

上述格式中的 `device`、`tables.device` 和 `tables.devices.device` 分别为 `驱动实例配置`、`模型配置(工作表的设备配置)` 和 `资产配置(设备的设备配置)`.

其中 `settings` 为 `驱动配置信息`, 与 `schema` 中的 `settings` 对应. `tags` 为 `数据点配置信息`, 与 `schema` 中的 `tags` 对应. `commands` 为 `指令配置信息`, 与 `schema` 中的 `commands` 对应.

> **INFO**
>
> `驱动实例`、`模型` 和 `资产` 中的 `settings` 可以不相同, 但 `tags` 和 `commands` 必须相同. 例如, 可以把统一的配置信息放在 `驱动实例` 中, 把不同的配置信息放在 `模型` 或 `资产` 中.

5. 实现驱动接口

`SDK` 中定义了 `数据接入驱动接口`, 该接口是平台控制驱动的桥梁. 开发者需要实现这个接口.

```
```

```
class Driver {
  /**
   * @name: schema
   * @msg: 查询返回驱动配置schema js内容
   * @param app
   * @param callback (err,"string") 驱动配置schema,返回字符串
   */
  schema(app, callback) {}

  /**
   * @name: start
   * @msg: 驱动启动
   * @param app
   * @param driverConfig 包含实例、模型及设备数据
   * @param callback (err)
   */
  start(app, driverConfig, callback) {}

  /**
   * @name: run
   * @msg: 运行指令,向设备写入数据
   * @param app
   * @param command 指令参数 {"table":"表标识","id":"设备编号","serialNo":"流水号","command":
{}} command 指令内容
   * @param callback (err,result) result自定义返回的格式或者空
   */
  run(app, command, callback) {}

  /**
   * @name: batchRun
   * @msg: 批量运行指令,向多设备写入数据
   * @param app
   * @param command 指令参数 {"table":"表标识", "ids": ["设备编号"], "serialNo": "流水号",
'command': {}}  command 指令内容
   * @param callback (err,result) result自定义返回的格式或者空
   */
  batchRun(app, command, callback) {}

  /**
   * @name: writeTag
   * @msg: 数据点写入
   * @param app
   * @param command 指令参数 {"table":"表标识","id":"设备编号","serialNo":"流水号","command":
{}} command 指令内容
   * @param callback (err,result) result自定义返回的格式或者空
   */
  writeTag(app, command, callback) {}

  /**
```

```
    * @name: debug
    * @msg: 调试驱动
    * @param app
    * @param debugConfig object 调试参数
    * @param callback (err,result) result调试结果,自定义返回的格式
    */
  debug(app, debugConfig, callback) {}

  /**
    * @name: debug
    * @msg: 代理接口
    * @param app
    * @param type 请求接口标识
    * @param header 请求头
    * @param data 请求数据
    * @param callback (err,result) result) result响应结果,自定义返回的格式
    */
  httpProxy(app, type, header, data, callback) {}

  /**
    * @name: stop
    * @msg: 驱动停止处理
    * @param app
    * @param callback (err)
    */
  stop(app, callback) {}

  /**
    * @name: getVersion
    * @msg: 驱动版本
    * @return: string 驱动版本号
    */
  getVersion() {}
}
```

> **INFO**
>
> **注: 向平台上报采集到的数据时, 必须通过 `驱动与平台交互接口` 中的 `writePoint` 方法发送, 不能直接调用 `MQTT` 客户端发送. 因为 `SDK` 会对发送的数据进行一些处理, 包括有效范围处理、数值映射、缩放比例、小数位等处理.**

6. 配置驱动

这里所说 `驱动配置` 主要是一些静态配置信息（**与 schema 中定义的配置无关**）, 其中包括 `平台配置信息`, `驱动配置信息` 和 `自定义配置信息`. 这些信息一般通过配置文件(rc)、环境变量、命令行参数等方式传入. 其中一些配置信息由平台启动驱动时通过命令行参数传入.

这些配置信息, 在开发过程中可以根据实际情况进行调整. 但是在打包时必须按照平台的要求进行配置. 打包时的配置信息见 驱动配置说明.

平台配置信息

`平台配置信息` 主要为平台的连接信息. 包括: `MQTT` 连接信息, `驱动管理服务` 连接信息. 内容如下:

```
{
  "project": "所属项目ID",
  "serviceId": "驱动实例ID",
  "mq": {
    "type": "mqtt",
    "mqtt": {
      "host": "平台mqtt服务器ip地址",
      "port": "平台mqtt服务器端口"
    }
  },
  "driver-grpc": {
    "host": "驱动管理服务ip地址",
    "port": "驱动管理服务端口"
  },
  "logger": {
    "level": "debug"
  },
  "driver": {
    "id": "驱动ID",
    "name": "驱动名称"
  }
}
```

> **INFO**
>
> 相关服务的端口号可在运维管理系统中查看.

驱动配置信息

`驱动配置信息` 主要包括 `驱动ID`, `驱动名称`, `驱动实例ID`, `所属项目ID`.

- `驱动ID` 为驱动的唯一标识, 必须在平台中唯一.
- `驱动名称` 为该驱动在平台中的显示名称.
- `驱动实例ID` 为该驱动实例的唯一标识. 同一个驱动可以创建多个实例, 每个实例的 `驱动ID` 相同但 `驱动实例ID` 唯一. 该信息由平台在 `驱动管理` 中创建驱动实例时生成.
- `所属项目ID` 每个驱动实例都属于一个项目, 该驱动实例只会拿到该项目中的模型和设备信息.

```
{
  "project": "所属项目ID",
  "serviceId": "驱动实例ID",
  "driver": {
    "id": "驱动ID",
    "name": "驱动名称"
  }
}
```

> **INFO**
>
> 1. `驱动ID` 和 `驱动名称` 需要在配置中手动定义.
> 2. `驱动实例ID` 和 `所属项目ID` 在开发过程中, 需要将这些信息手动配置. 在打包时无须定义, 在平台中安装驱动时这些信息会由平台通过命令行参数传入.

驱动配置说明

以下是完整的驱动配置文件, 请参考该配置文件进行配置.

```
{
  "project": "所属项目ID",
  "serviceId": "驱动实例ID",
  "mq": {
    "type": "mqtt",
    "mqtt": {
      "host": "127.0.0.1",
      "port": 1883
    }
  },
  "driver-grpc": {
    "host": "127.0.0.1",
    "port": 9224
  },
  "logger": {
    "level": "debug"
  },
  "driver": {
    "id": "驱动ID",
    "name": "驱动名称"
  }
}
```

windows系统打包发布时的驱动配置

```
{
  "mq": {
    "type": "mqtt",
    "mqtt": {
      "host": "localhost",
      "port": 1883
    }
  },
  "driver-grpc": {
    "host": "localhost",
    "port": 9224
  },
  "driver": {
    "id": "node-driver-mqtt-demo",
    "name": "Node驱动例子"
  }
}
```

linux系统打包发布时的驱动配置

```
{
  "driver": {
    "id": "node-driver-mqtt-demo",
    "name": "Node驱动例子"
  }
}
```

流程插件开发

本文将会详细介绍如何使用 `Node SDK` 开发流程插件.

介绍

`流程插件` 是扩展 `流程引擎` 中的节点的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程插件来实现自定义的功能.

> **INFO**
>
> `流程插件` 只是扩展流程功能的方式之一. 除了 `流程插件` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

该过程同 数据接入驱动开发-创建项目 中的创建项目过程一致.

2. 引入SDK

```
const {App, Flow} = require('@airiot/sdk-nodejs/flow')
```

3. 实现流程插件接口

`SDK` 中定义了 `流程插件接口`, 该接口是平台与插件交互的桥梁. 开发者需要实现这个接口.

```
class Flow {
  /**
   * @name: handler
   * @msg: 执行流程插件
   * @param app
   * @param req 执行参数 {"projectId":"项目id","flowId":"流程id","job":"流程实例
id","elementId":"节点id","elementJob":"节点的实例id","config":{}} config 节点配置
   * @param callback (err,result) result 自定义格式,节点执行结果
```

```
   */
  handler(app, req, callback) {}

  /**
   * @name: stop
   * @msg: 停止流程服务
   * @param app
   * @param callback (err)
   */
  stop(app, callback) {}
}
```

流程插件启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程执行到该插件对应的节点时, 会发送请求给该插件对应的程序. `SDK` 接收到请求后会调用对应的插件实现, 并将插件处理结果返回给流程引擎.

4. 配置插件

插件配置主要是插件与平台的连接配置.

```
{
  "flow-engine": {
    "host": "流程引擎服务地址",
    "port": "流程引擎服务端口"
  },
  "logger": {
    "level": "debug"
  },
  "flow": {
    "name": "插件名称",
    "mode": "service"
  }
}
```

windows系统打包发布时的插件配置

```
{
  "flow-engine": {
    "host": "127.0.0.1",
    "port": 2333
  },
  "flow": {
    "name": "flowTestNode",
    "mode": "service"
```

```
  }
}
```

linux系统打包发布时的插件配置

```
{
  "flow": {
    "name": "flowTestNode",
    "mode": "service"
  }
}
```

流程扩展节点接入

本文将会详细介绍如何使用 `Node SDK` 开发流程扩展节点接入服务.

介绍

`流程扩展节点接入` 是扩展 `流程引擎` 中的 `扩展节点` 的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程扩展节点接入服务来实现自定义的功能.

> **INFO**
>
> `流程扩展节点接入` 只是扩展流程功能的方式之一. 除了 `流程插件` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

该过程同 数据接入驱动开发-创建项目 中的创建项目过程一致.

2. 引入SDK

```
const {App, Extension} = require('@airiot/sdk-nodejs/flow_extension')
```

3. 实现流程扩展节点接入接口

`SDK` 中定义了 `流程扩展节点接入`, 该接口是平台扩展节点与该服务交互的桥梁. 开发者需要实现这个接口.

```
class Extension {
  /**
   * @name: schema
   * @msg: 查询schema
   * @param app
   * @param callback (err,'string') 配置schema,返回字符串
   */
  schema(app, callback) {}
```

```
  /**
   * @name: run
   * @msg: 执行服务
   * @param app
   * @param cfg 执行参数 {} input 执行参数,应与输出的schema格式相同
   * @param callback (err,result)  result自定义返回的格式,应与输出的schema格式相同
   */
  run(app, cfg, callback) {}


  /**
   * @name: stop
   * @msg: 停止服务
   * @param app
   * @param callback (err)
   */
  stop(app, callback) {}
}
```

流程扩展节点服务启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程执行到该扩展节点时, 会发送请求给该服务程序. `SDK` 接收到请求后会调用对应的服务实现, 并将服务处理结果返回给流程引擎扩展节点.

4. 配置流程扩展节点接入服务

扩展节点接入服务配置主要是服务与平台的连接配置.

```
{
  "flow-engine": {
    "host": "流程引擎服务地址",
    "port": "流程引擎服务端口"
  },
  "logger": {
    "level": "debug"
  },
  "extension": {
    "id": "扩展服务唯一标识",
    "name": "扩展服务显示名称"
  }
}
```

windows系统打包发布时的插件配置

```
```

```
{
  "flow-engine": {
    "host": "127.0.0.1",
    "port": 2333
  },
  "extension": {
    "id": "testNodeFlowExt",
    "name": "测试node扩展"
  }
}
```

linux系统打包发布时的插件配置

```
{
  "extension": {
    "id": "testNodeFlowExt",
    "name": "测试node扩展"
  }
}
```

算法服务开发

本文将会详细介绍如何使用 `Node SDK` 开发算法服务开发.

介绍

`算法服务` 是扩展 `算法` 中的一种方式. 在平台现有的功能不满足需求时, 可以通过开发算法服务来实现自定义的功能.

开发步骤

1. 创建项目

该过程同 数据接入驱动开发-创建项目 中的创建项目过程一致.

2. 引入SDK

```
const {App, Algorithm} = require('@airiot/sdk-nodejs/algorithm')
```

3. 定义schema

`算法服务` 需要定义一个 `schema` 用于描述算法的配置信息. `schema` 是一个类似于 `json` 格式的对象, 详细格式说明见 算法schema说明.

4. 实现算法接口

`SDK` 中定义了 `算法服务接口`, 该接口是平台与插件交互的桥梁. 开发者需要实现这个接口.

```
class Algorithm {
  /**
   * @name: schema
   * @msg: 查询schema
   * @param app
   * @param callback (err,'string') 算法配置schema,返回字符串
   */
  schema(app, callback) {}

  /**
```

```
    * @name: run
    * @msg: 执行算法服务
    * @param app
    * @param cfg 执行参数 {"function":"算法名","input":{}} input 算法执行参数,应与输出的schema格式相同
    * @param callback (err,result)  result自定义返回的格式,应与输出的schema格式相同
    */
  run(app, cfg, callback) {}

  /**
    * @name: stop
    * @msg: 停止算法服务
    * @param app
    * @param callback (err)
    */
  stop(app, callback) {}
}
```

算法服务启动时, `SDK` 会连接平台的 `算法服务` 服务, 并接收算法服务发送的请求. 当执行该服务算法时, 会发送请求给该服务程序. `SDK` 接收到请求后会调用对应的方法, 并将方法的处理结果返回给算法管理.

5. 配置算法

算法配置主要是算法与平台的连接配置.

```
{
  "algorithm-grpc": {
    "host": "算法管理服务地址",
    "port": "算法管理服务端口"
  },
  "logger": {
    "level": "debug"
  },
  "algorithm": {
    "id": "算法唯一标识",
    "name": "算法显示名称"
  }
}
```

windows系统打包发布时的算法配置

```
{
  "algorithm-grpc": {
    "host": "127.0.0.1",
```

```
    "port": 9236
  },
  "algorithm": {
    "id": "test",
    "name": "测试算法"
  }
}
```

linux系统打包发布时的算法配置

```
{
  "algorithm": {
    "id": "test",
    "name": "测试算法"
  }
}
```

平台接口客户端

介绍

平台客户端 SDK 用于访问平台接口，提供了平台接口的 Node 实现, 可以很方便实现第三方系统与平台的集成。平台客户端 SDK 中包括常用的 `租户管理`、`项目管理`、`用户管理`、`角色管理`、`工作表及数据管理`、`系统变量(数据字典)`、`报警管理` 等接口, 并且不断在丰富和完善中。

使用方式

1. 在项目中安装SDK

```
npm install @airiot/sdk-nodejs
```

2. 在需要调用平台接口创建客户端

在引入客户端 SDK 后。示例代码如下:

```
let ApiClient = require("@airiot/sdk-nodejs/api")
const config = {
  endpoint: "http://localhost:31000",
  type: 'tenant', // tenant,project
  // projectId: '625f6dbf5433487131f09ff8',
  ak: "ak",          //您的AccessKey
  sk: "sk"      //您的SecretAccessKey
}
let cli = new ApiClient(config)
```

> **INFO**
>
> 关于如何创建ak、sk

查询构造器

客户端接口中的很多查询接口, 其结构比较复杂.

查询参数的整体结构如下所示:

```
{
  "project": {},
  "filter": {},
  "sort": {},
  "limit": 30,
  "skip": 20,
  "withCount": true
}
```

字段说明如下:

- **`project`** 查询请求需要返回的字段列表. 例如: `{"id": 1, "name": 1, "address": {"city": 1}}`. `key` 为字段名, `value` 为 `1` 或 `对象`. 如果为为一级字段需要设置为 `1` 例如: `{"id": 1, "name": 1}`, 如果要返回嵌套对象内的字段, 则需要设置为 `Map`, 例如: `{"address": {"city": 1}}`
- **`filter`** 查询条件, 如果没有添加任何条件则查询全部数据. `key` 为字段名, `value` 为过滤的值或逻辑运算符, 例如: `{"name": "Tom", "age": {"$gt": 20, "$lt": 30}}`.
- **`sort`** 排序条件, `key` 为字段名, `value` 为 `1` 表示升序, `-1` 表示降序, 例如: `{"age": 1, "name": -1}`.
- **`limit`** 查询结果的最大数量, 可用于分页查询或限制返回的记录数量.
- **`skip`** 查询结果的偏移量, 即忽略前 N 记录, 可用于分页查询.
- **`withCount`** 是否返回符合条件的记录总数, 如果为 `true` 则会在查询结果记录数量会保存在响应对象 `ResponseDTO<T>` 中的 `count` 字段.

> **INFO**
>
> **注意事项**
>
> 1. 如果查询条件需要使用逻辑或, 可以在 `filter` 中添加 `$or` 字段, 其值为 `{k:v}` 结构与 `filter` 一致, 任一条件成立时表示记录匹配.
> 2. 如果同一字段存在多个逻辑条件, 则需要将多个条件放在一个 `对象` 中, 例如: `{"age": {"$gt": 20, "$lt": 30}}`, 表示查询 `20 < age < 30` 的记录.

逻辑运算符

符号	说明	示例
$not	不相等, 与 `SQL` 中的 `<>` 作用相同	{"age": {"$not": 18}}
$in	在指定列表内, 与 `SQL` 中的 `in` 作用相同	{"id": {"$in": [1,3,4]}}

符号	说明	示例
$nin	不在指定列表内, 与 `SQL` 中的 `not in` 作用相同	{"id": {"$nin": [1,3,4]}}
$gt	大于指定的值, 与 `SQL` 中的 `>` 作用相同	{"age": {"$gt": 18}}
$gte	大于等于指定的值, 与 `SQL` 中的 `>=` 作用相同	{"age": {"$gte": 18}}
$lt	小于指定的值, 与 `SQL` 中的 `<` 作用相同	{"age": {"$lt": 18}}
$lte	小于等于指定的值, 与 `SQL` 中的 `<=` 作用相同	{"age": {"$lte": 18}}
$regex	正则匹配, 与 `SQL` 中的 `like` 相似	{"name": {"$regex": "张"}}

打包部署

本文将详细介绍平台打包及部署.

打包

驱动打包就是将开发完成的程序打包为可以在平台部署的驱动. 平台自身支持运行在 `windows`、`linux` 和 `macOS` 系统中, 并且支持 `x86` 和 `arm` 平台. 在 `windows` 系统中平台服务和驱动程序都是直接运行在操作系统中, 而在 `linux` 系统中是以 `容器` 的方式运行, 平台中的每个服务和驱动程序都是一个独立的容器, 所以针对不同的操作系统打包方式也不相同. 下面分别介绍在 `windows` 和 `linux` 系统中如何打包驱动， 对于不同平台只需要保证使用软件和库支持即可.

windows系统打包

在 `windows` 系统中, 驱动程序是直接运行在操作系统中, 所以需要将驱动程序打包. 具体打包步骤如下:

1. 以 `Node` 驱动为例打程序程序和相关资源打包为 `.js` 文件. 由于包含很多依赖库,所以首先安装 `@vercel/ncc` 编译工具,执行下面的命令进行安装:

```
npm i -g @vercel/ncc
```

项目执行下面的命令进行编译:

```
ncc build index.js -o dist -m
```

2. 准备驱动配置文件 `config.json`. 可以将 `config.json` 放在与 `编译后的文件index.js` 文件相同目录下, 这样驱动在启动时会自动加载该配置文件.

> **INFO**
>
> 注: `config.json` 中需要填写好 `驱动ID` 和 `驱动名称` 两个配置项, 参考驱动文档.

3. 准备驱动安装配置文件 `service.yml` . 在平台中安装驱动时, 需要提供一些驱动的基本信息, 例如: 版本号、驱动描述、端口号等. 这些信息需要在 `service.yml` 中定义, 平台会根据该文件中的配置信息进行安装. `service.yml` 的具体格式如下:

```
```

```
Name: driver-mqtt-demo
Description: 测试程序
Version: 4.0.0
ConfigType: config.json
GroupName: driver
Command: node index.js --config config.json
```

4. 将所有资源打包为 `zip` 文件.

将 `index.js`、`config.json`、`service.yml`、`driver.proto`、`schema.js` 和其它资源打包为 `zip` 文件, 平台会根据该文件进行安装. 建议打包后的 `zip` 文件结构如下:

linux系统打包

由于在 `linux` 系统中, 驱动程序是以 `容器` 的方式运行, 所以打包时需要先将驱动程序打包为 `docker` 镜像. 然后再将镜像文件和 `service.yml` 打包为 `.tar.gz` 压缩包. 具体打包步骤如下:

1. 打包程序程序打包为 `index.js` 文件. 具体打包步骤参考 windows系统打包 中的第一步. 需要注意的是, 在 `linux` 系统打包中, 不要求包含驱动配置文件 config.json`.

2. 准备 `Dockerfile` 文件. 以下是一个简单的 `Dockerfile` 文件示例, 具体内容根据自身的需求进行修改:

```
FROM node:18.17.1-alpine3.17
WORKDIR /app
ADD dist/* /app/
ENTRYPOINT [ "node", "index.js" ]
```

3. 构建 `docker` 镜像.

使用上一步中的 `Dockerfile` 文件构建 `docker` 镜像, 具体命令如下:

```
docker build -t airiot/node-driver-example:v4.0.0 .
```

4. 导出 `docker` 镜像并压缩.

```
docker save airiot/node-driver-example:v4.0.0 | gzip > node-driver-example.tar.gz
```

5. 准备驱动安装配置文件 `service.yml` . 该文件的格式与 windows系统打包 中的第三步中的 `service.yml` 文件格式相似但又有区别. 具体格式如下:

```
# 必填项. 驱动名称
Name: node-driver-example
# 必填项. 例如: 1.0.0, 通常用镜像版本号一致
Version: 4.0.0
# 非必填项.
Description: 描述信息
# 必填项. 驱动固定为 driver、流程插件、算法服务为 server
GroupName: driver
# 容器端口映射类型, 非必填项. 如果驱动需要对外提供 rest 服务, 或暴露端口时, 需要填写该配置项.
# 可选项有 None Internal External
#
# None: 不暴露端口
# Internal: 只在平台内部暴露端口. 一般为驱动对外提供 rest 服务时, 将端口映射到网关上, 填写为
Internal 即可.
# External: 对外暴露端口. 一般为驱动为作 server 端, 需要对外暴露端口以供设备连接, 此时该端口会暴露
在宿主机上, 填写为 External 即可.
Service: Node
# 非必填项. 暴露的端口列表
#Ports:
#  - Host: "8558"            # 映射到宿主机的端口号, 如果不填写, 则会随机分配一个端口号
#    Container: "8558"       # 容器内部的端口号, 即驱动服务监听的端口号
#    Protocol: ""            # 协议类型, 可选项有 TCP UDP, 如果不填写, 则默认为 TCP
```

6. 将所有资源打包为 `gzip` 文件. 将 `docker镜像` 和 `service.yml` 文件打包为 `gzip` 文件. 打包命令如下:

```
tar cvf node-driver-example.tar node-driver-example.tar.gz service.yml
```

```
gzip node-driver-example.tar
```

打包后的 `gzip` 文件结构如下:

> **INFO**

`linux` 系统整个打包过程对应的命令如下所示:

```
# 将驱动打包为镜像
docker build -t airiot/node-driver-example:v4.0.0 .

# 导出镜像并压缩
docker save airiot/node-driver-example:v4.0.0 | gzip > node-driver-example.tar.gz

# 将镜像文件和 service.yml 打包
tar cvf node-driver-example.tar node-driver-example.tar.gz service.yml

# 对整个驱动包进行压缩
gzip node-driver-example.tar

# 最后得到 node-driver-example.tar.gz 压缩文件
```

部署

将上一步骤中得到的驱动安装包通过 `运维管理系统` 上传到平台, 平台会自动解析并安装驱动. 安装成功后, 就可以在项目中使用该驱动了.

安装驱动

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `离线上传驱动` 按钮, 选择上一步中得到的 `node-driver-example.tar.gz` 文件, 点击 `确定` 按钮, 平台会自动解析并安装驱动.

如果驱动安装失败, 可以在 `运维管理系统` 的 `首页` 中查看详细的日志信息.

> **INFO**
>
> 不同版本的平台, `离线上传驱动` 按扭的位置可能不同.

使用驱动

当驱动成功安装到平台后, 就可以在项目中使用该驱动了.

具体使用方法请参考 驱动管理.

> **INFO**
>
> 注: 需要将运维服务的 InternetAccess 改为false, 才能读取本地仓库

Python SDK 介绍

Python SDK 二次开发说明

数据接入驱动开发

介绍如何使用 Python SDK 开发自定义数据接入驱动

流程插件开发

介绍如何使用 Python SDK 开发自定义流程插件

平台接口客户端

Python SDK 平台接口客户端

流程扩展节点开发

介绍如何使用 Python SDK 扩展流程节点

算法服务开发

介绍如何使用 Python SDK 将算法集成到平台

常见问题

Python SDK 使用过程中的常见问题

Python SDK 介绍

`Python SDK` 是 AIRIOT 物联网平台提供的 `Python` 语言的二次开发工具包, 可使用 Python SDK 调用平台开放的接口和实现对平台功能的扩展。

接下来会分别介绍如何使用 `Python SDK` 开发自定义数据接入驱动、开发自定义流程插件、算法集成和通过平台接口客户端实现系统集成。

内容说明

内容	用途
自定义数据接入驱动	实现从设备或其它平台系统采集数据功能
自定义流程插件	扩展流程引擎中的插件
扩展流程节点	扩展流程中的节点
算法集成	扩展或集成已有算法到平台
平台接口客户端	调用平台对外提供的数据接口

SDK 内容列表

包名	描述
driver	提供自定义数据接入驱动开发的相关内容. 包括驱动的接口定义, 以及与平台交互功能
flow_plugin	提供自定义流程引擎插件开发的相关内容. 包括与流程引擎交互的具体实现和数据交互格式的定义等
flow_extension	提供自定义流程引擎节点开发的相关内容. 包括与流程引擎交互的具体实现和数据交互格式的定义等
algorithm	提供算法接入的相关内容. 可以扩展平台算法或将已经算法集成到平台

包名	描述
client	平台接口客户端， 提供了常用平台接口调用客户端实现

使用方式

`Python SDK` 包是通过 github 发布的, 访问 `github` 仓库, 然后找到已发布的 release，在发版文件中可以找到 `airiot_python_sdk-{VERSION}-py3-none-any.whl` 文件并下载, 然后执行下面的命令进行安装: `

```
# 例如安装 4.0.3 版本
pip install airiot_python_sdk-4.0.3-py3-none-any.whl
```

版本说明

SDK 版本	Python 版本
4.0.x	3.10+

数据接入驱动开发

本文将会详细介绍如何使用 `Python SDK` 开发自定义数据接入驱动. 示例项目目上传至 https://github.com/air-iot/sdk-python-examples/tree/master/driver.

介绍

`数据接入驱动` 是为实现从不同的协议、设备或其它平台系统采集数据而开发的特定程序. 每个 `数据接入驱动` 程序需要根据协议的特点实现数据采集功能，然后将采集到的数据通过 `Python SDK` 提供的接口发送到平台.

`Python SDK` 提供了 `数据接入驱动` 开发的相关内容. 包括驱动的接口定义, 以及与平台交互功能. 开发者只需要在项目中引入 `airiot_python_sdk` 依赖, 定义配置信息(schema及配置类)，然后实现 `DriverApp` 接口中的方法即可.

开发步骤

1. 创建项目

数据接入驱动开发, 对项目结构未做要求, 根据自己的习惯创建项目即可. 使用的 `Python` 版本见 版本说明.

2. 引入SDK

安装使用

> **INFO**
>
> 驱动二次开发相关内容都在 `driver` 子包内.

3. 定义schema

`数据接入驱动` 需要定义一个 `schema` 用于描述驱动的配置信息. `schema` 是一个类似于 `json` 格式的对象, 详细格式说明见 数据接入驱动schema说明. 以下是一个简单的示例:

```
({
    "driver": {
        "properties": {
            "settings": {
```

```
                "title": "模型配置",
                "type": "object",
                "properties": {
                    "server": {
                        "type": "string",
                        "title": "MQTT Broker",
                        "descripption": "MQTT 服务器地址. 例如: tcp://127.0.0.1:1883"
                    },
                    "username": {
                        "type": "string",
                        "title": "用户名",
                    },
                    "password": {
                        "type": "string",
                        "title": "密码",
                        "fieldType": "password"
                    },
                    "network": {
                        "type": "object",
                        "title": "通讯监控参数",
                        "properties": {
                            "timeout": {
                                "title": "通讯超时时间(s)",
                                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故
障",

                                "type": "number"
                            }
                        },
                        "form": [{
                            "key": "timeout",
                        }]
                    }
                },
                "required": ["server", "username", "password"]
            },
            "tags": {
                "title": "数据点",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "key": {
                            "type": "string",
                            "title": "Key",
                            "description": "数据点在 JSON 对象中的 Key. 例如: 'temperature'",
                        },
                    },
                    "required": ["key"]
                }
            },
```

```
            "commands": {
                "title": "命令",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "title": "名称"
                        },
                        "ops": {
                            "type": "array",
                            "title": "指令",
                            "items": {
                                "type": "object",
                                "properties": {
                                    "name": {
                                        "type": "string",
                                        "title": "主题",
                                        "description": "发送消息的主题. 例如: /cmd/control",
                                    },
                                    "message": {
                                        "type": "string",
                                        "title": "消息",
                                        "description": "发送的消息. 例如:
{\"cmd\":\"start\"}",
                                    },
                                    "qos": {
                                        "type": "number",
                                        "title": "QoS",
                                        "description": "消息质量. 0,1,2",
                                        "enum": ["0", "1", "2"],
                                        "enum_title": ["QoS0", "QoS1", "QoS2"]
                                    },
                                },
                                "required": ["name", "message"]
                            }
                        }
                    }
                }
            }
        },
        "model": {
            "properties": {
                "settings": {
                    "title": "模型配置",
                    "type": "object",
                    "properties": {
                        "topic": {
```

```
                        "type": "string",
                        "title": "主题",
                        "descripption": "接收数据的主题. 例如: /data/#"
                    },
                    "parseScript": {
                        "type": "string",
                        "title": "数据处理脚本",
                        "fieldType": "deviceScriptEdit",
                        "description": "消息处理脚本. 函数名必须为 'handler'",
                        "defaultScript": "/**\n" +
                            " * 数据处理脚本, 处理从 mqtt 接收到的数据.\n" +
                            " *\n" +
                            " * @param {string} topic 消息主题\n" +
                            " * @param {string} message 消息内容\n" +
                            " * @return 消息解析结果\n" +
                            " */\n" +
                            "function handler(topic, message) {\n" +
                            "\t\n" +
                            "\t// 脚本返回值必须为对象数组\n" +
                            "\t// \tid: 资产编号\n" +
                            "\t//\ttime: 时间戳(毫秒)\n" +
                            "\t//  fields: 数据点数据. 该字段为 JSON 对象, key 为数据点标识,
value 为数据点的值\n" +
                            "\treturn [\n" +
                            "\t\t{\"id\": \"SN10001\", \"time\": new Date().getTime(),
\"fields\": {\"key1\": \"this is a string value\", \"key2\": true, \"key3\": 123.456}}\n" +
                            "\t];\n" +
                            "}"
                    },
                    "commandScript": {
                        "type": "string",
                        "title": "指令处理脚本",
                        "fieldType": "deviceScriptEdit",
                        "description": "指令处理脚本. 函数名必须为 'handler'",
                        "defaultScript": "/**\n" +
                            " * 指令处理脚本. 发送指令时会将指令内容传递给脚本, 然后由指定返回最终
要发送的信息.\n" +
                            " *\n" +
                            " * @param {string} 工作表标识\n" +
                            " * @param {string} 资产编号\n" +
                            " * @param {object} 命令内容\n" +
                            " * @return {object} 最终要发送的消息, 及目标 topic\n" +
                            " */\n" +
                            "function handler(tableId, deviceId, command) {\n" +
                            "\t\n" +
                            "\t// 脚本返回值必须为下面对象结构\n" +
                            "\t//\t\ttopic: 消息发送的目标 topic\n" +
                            "\t//\t\tpayload: 消息内容\n" +
                            "\treturn {\n" +
                            "\t\t\"topic\": \"cmd/\" + deviceId,\n" +
```

```
                        "\t\t\"payload\": \"发送内容\"\n" +
                        "\t};\n" +
                        "}"
                },
                "network": {
                    "type": "object",
                    "title": "通讯监控参数",
                    "properties": {
                        "timeout": {
                            "title": "通讯超时时间(s)",
                            "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故
障",
                            "type": "number"
                        }
                    },
                    "form": [{
                        "key": "timeout",
                    }]
                }
            },
            "required": ["server", "username", "password", "topic"]
        },
        "tags": {
            "title": "数据点",
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "key": {
                        "type": "string",
                        "title": "Key",
                        "description": "数据点在 JSON 对象中的 Key. 例如: 'temperature'",
                    },
                },
                "required": ["key"]
            }
        },
        "commands": {
            "title": "命令",
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "name": {
                        "type": "string",
                        "title": "名称"
                    },
                    "ops": {
                        "type": "array",
                        "title": "指令",
```

```
                            "items": {
                                "type": "object",
                                "properties": {
                                    "name": {
                                        "type": "string",
                                        "title": "主题",
                                        "description": "发送消息的主题. 例如: /cmd/control",
                                    },
                                    "message": {
                                        "type": "string",
                                        "title": "消息",
                                        "description": "发送的消息. 例如:
{\"cmd\":\"start\"}",
                                    },
                                    "qos": {
                                        "type": "number",
                                        "title": "QoS",
                                        "description": "消息质量. 0,1,2",
                                        "enum": ["0", "1", "2"],
                                        "enum_title": ["QoS0", "QoS1", "QoS2"]
                                    },
                                },
                                "required": ["name", "message"]
                            }
                        }
                    }
                }
            }
        },
        "device": {
            "properties": {
                "settings": {
                    "title": "设备配置",
                    "type": "object",
                    "properties": {
                        "customDeviceId": {
                            "type": "string",
                            "title": "设备编号",
                            "descripption": "自定义设备编号. 如果未定义则使用平台中的资产编号"
                        },
                        "network": {
                            "type": "object",
                            "title": "通讯监控参数",
                            "properties": {
                                "timeout": {
                                    "title": "通讯超时时间(s)",
                                    "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故
障",
                                    "type": "number"
```

```
                        }
                    }
                }
            },
            "required": []
        }
    }
})
```

4. 实现驱动接口

数据接入驱动二次开发SDK中, 有两个重要接口: `数据接入驱动接口` 和 `驱动与平台交互接口`, 分别为 **DriverApp** 和 **DataSender**.

- **数据接入驱动接口(DriverApp)** 该接口中定义平台控制驱动程序运行的相关方法, 是平台控制驱动程序的入口. `SDK` 在驱动程序启动后, 会监听平台下发的控制指令, 然后调用 `数据接入驱动接口` 中的方法并将执行结果返回给平台. 开发者需要实现这个接口, 并提供一个工厂类 `DriverAppFactory`, 并且将工厂类传递给驱动启动器 `Launcher`. 接口定义及详细说明见 数据接入驱动接口说明.

- **驱动与平台交互接口(DataSender)** 是驱动程序与平台进行交互的接口. 驱动程序在运行过程中, 除了接收平台的控制之外, 也需要与平台进行各种交互. 例如: 向平台发送采集的到数据, 向平台发送指令执行结果、调试日志等. `SDK` 中定义了 `驱动与平台交互接口` **DataSender** 以及实现类, 该实现类会通过工厂类方法传入, 可以在工厂类方法中创建 `DriverApp` 时保存该对象. 接口定义及详细说明见 驱动与平台交互接口说明.

注: 向平台上报采集到的数据时, 必须通过 `驱动与平台交互接口` 中的 `writePoint` 方法发送, 不能直接调用 `MQTT` 客户端发送. 因为 `SDK` 会对发送的数据进行一些处理, 包括有效范围处理、数值映射、缩放比例、小数位等处理. :::

5. 配置驱动

这里所说 `驱动配置` 主要是一些静态配置信息（**与 schema 中定义的配置无关**）, 其中包括 `平台配置信息`, `驱动配置信息` 和 `自定义配置信息`. 这些信息一般通过配置文件(application.yml)、环境变量、命令行参数等方式传入. 其中一些配置信息由平台启动驱动时通过命令行参数传入.

这些配置信息, 在开发过程中可以根据实际情况进行调整. 但是在打包时必须按照平台的要求进行配置. 打包时的配置信息见 驱动配置说明.

> **INFO**

`自定义配置信息` 是指驱动本身的一些配置信息, 对驱动使用者不可见. 例如: 连接池大小. 这些信息一般通过配置文件(application.yml)、环境变量、命令行参数等方式传入.

平台配置信息

`平台配置信息` 主要为平台的连接信息. 包括: `MQTT` 连接信息, `驱动管理服务` 连接信息. 内容如下:

```yaml
# 驱动管理服务配置信息
driver-grpc:
  # 在开发时, 需要配置为平台的地址
  host: 192.168.11.101
  # 端口默认为 9224, 一般无须修改. 如果有修改, 可在运维管理系统中查看 `driver` 服务的端口号
  port: 9224
# 平台 MQTT 配置信息
mq:
  mqtt:
    # 在开发时, 需要配置为平台的地址
    host: 192.168.11.101
    port: 1883
    username: admin
    password: public
# 如果驱动对外提供 web 接口服务, 则需要按以下方式配置 web 服务的端口号
server:
  port: 8080
```

> **INFO**
>
> 相关服务的端口号可在运维管理系统中查看. 平台中的 `driver` 服务的端口对应 `driver-grpc.port` 配置项, `mqtt` 服务的端口对应 `mq.mqtt.port` 配置项.

驱动配置信息

`驱动配置信息` 主要包括 `驱动ID`, `驱动名称`, `驱动实例ID`, `所属项目ID`.

- `驱动ID` 为驱动的唯一标识, 必须在平台中唯一.
- `驱动名称` 为该驱动在平台中的显示名称.
- `驱动实例ID` 为该驱动实例的唯一标识. 同一个驱动可以创建多个实例, 每个实例的 `驱动ID` 相同但 `驱动实例ID` 唯一. 该信息由平台在 `驱动管理` 中创建驱动实例时生成.
- `所属项目ID` 每个驱动实例都属于一个项目, 该驱动实例只会拿到该项目中的模型和设备信息.

```
```

```
id: 驱动ID
name: 驱动名称
```

> **INFO**
>
> 1. `驱动ID` 和 `驱动名称` 需要在 `config.yml` 中手动定义.
> 2. `驱动实例ID` 和 `所属项目ID` 在开发过程中, 需要将这些信息手动配置到 `application.yml` 中. 在打包时无须定义, 在平台中安装驱动时这些信息会由平台通过命令行参数传入.

示例配置

```yaml
id: python_sdk_demo_driver
name: PythonSDK示例驱动
# 驱动管理服务配置信息
driver-grpc:
  # 在开发时, 需要配置为平台的地址
  host: 192.168.11.101
  # 端口默认为 9224, 一般无须修改. 如果有修改, 可在运维管理系统中查看 `driver` 服务的端口号
  port: 9224
# 平台 MQTT 配置信息
mq:
  mqtt:
    # 在开发时, 需要配置为平台的地址
    host: 192.168.11.101
    port: 1883
    username: admin
    password: public
# 如果驱动对外提供 web 接口服务, 则需要按以下方式配置 web 服务的端口号
server:
  port: 8080
```

6. 打包

驱动打包就是将开发完成的程序打包为可以在平台部署的驱动. 平台自身支持运行在 `windows`、`linux` 和 `macOS` 系统中, 并且支持 `x86` 和 `arm` 平台. 在 `windows` 系统中平台服务和驱动程序都是直接运行在操作系统中, 而在 `linux` 系统中是以 `容器` 的方式运行, 平台中的每个服务和驱动程序都是一个独立的容器, 所以针对不同的操作系统打包方式也不相同. 下面分别介绍在 `windows` 和 `linux` 系统中如何打包驱动， 对于不同平台只需要保证使用软件和库支持即可.

windows系统打包

1. 将程序和相关资源打包. 可以使用 `PyInstaller` 等工具将程序打包为可执行文件, 或者直接使用 `python main.py` 等方式运行程序. **注: 使用 `python main.py` 的方式运行程序时, 需要将 `python` 运行环境一起打包**

2. 准备驱动配置文件 `config.yml`. 可以将 `config.yml` 文件放在根目录下, 或放在其它目录中, 然后在程序中加载.

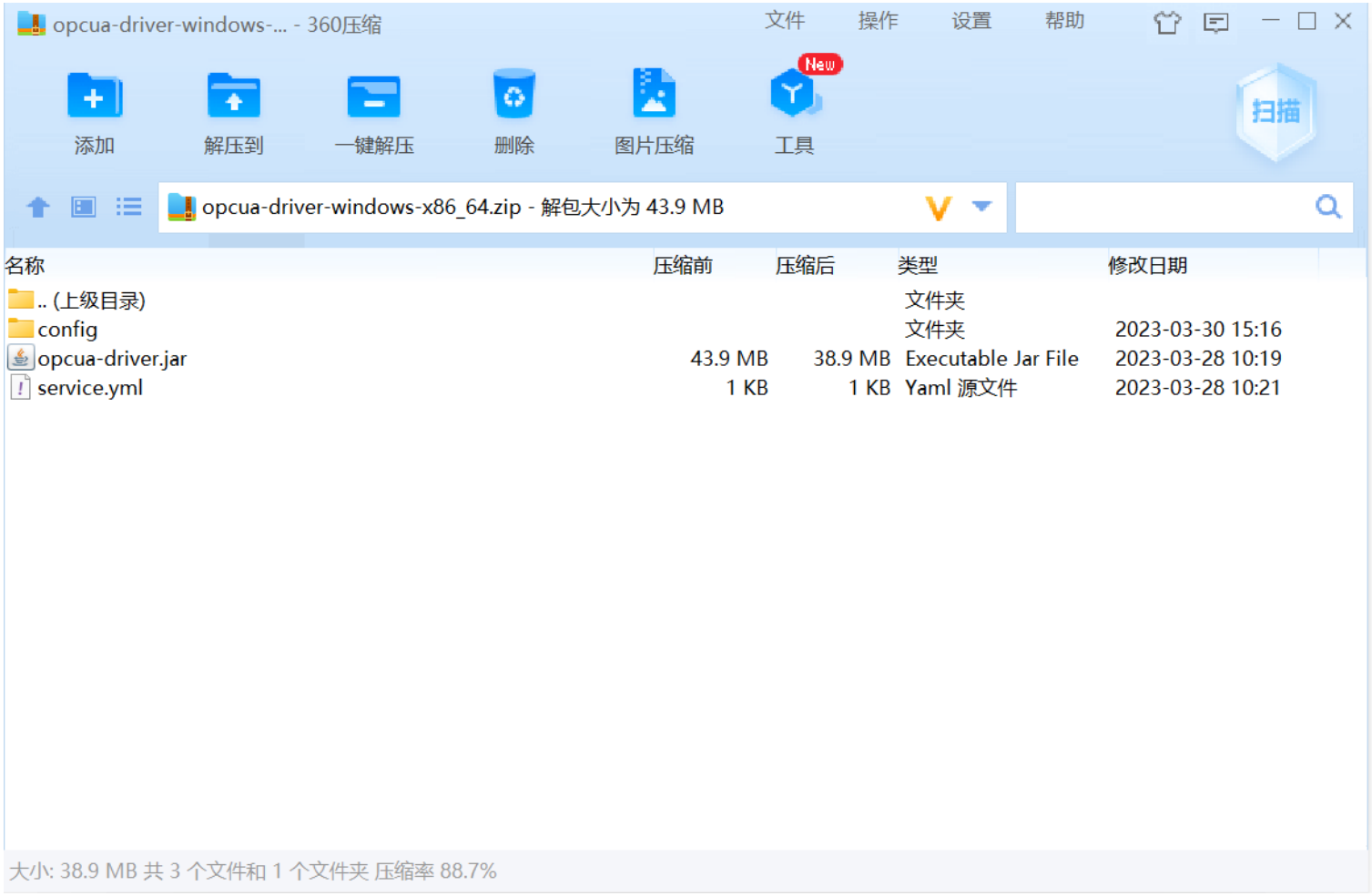
> **INFO**
>
> 注: `config.yml` 中需要填写好 `驱动ID` 和 `驱动名称` 两个配置项.

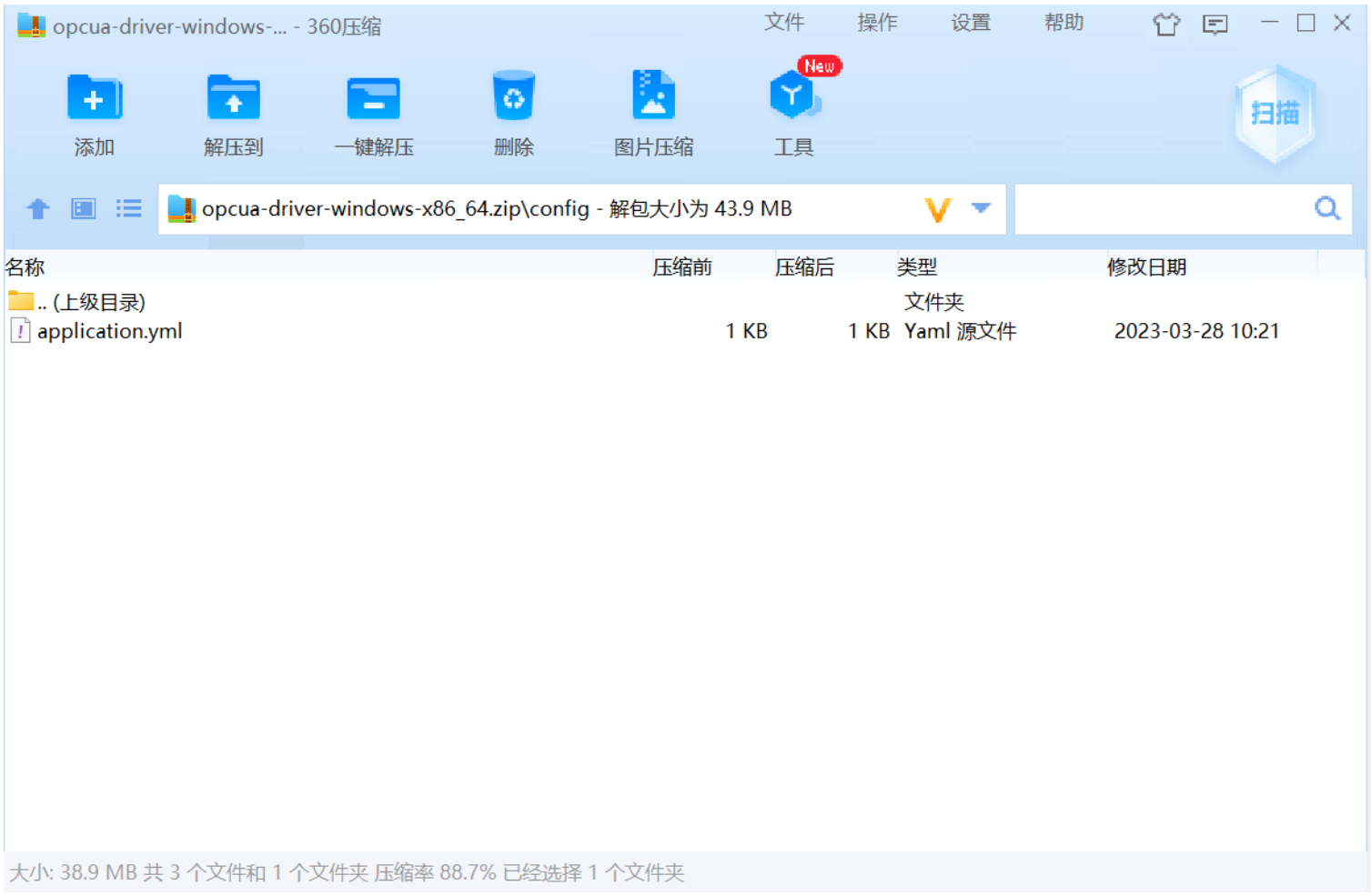
3. 准备驱动安装配置文件 `service.yml` . 在平台中安装驱动时, 需要提供一些驱动的基本信息, 例如: 版本号、驱动描述、端口号等. 这些信息需要在 `service.yml` 中定义, 平台会根据该文件中的配置信息进行安装. `service.yml` 的具体格式如下:

```yaml
# 必填项. 驱动名称
Name: python_sdk_demo_driver
# 非必填项. 如果驱动对外提供 rest 服务, 则需要填写 rest 接口的统一路径前缀.
# 当填写该配置项时, 平台会自动在网关中添加该路径的路由, 并将请求转发到该驱动, 代理端口为 service.yml 文件中的 server.port 配置项.
Path: /python_sdk_demo_driver
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 驱动描述信息
# 驱动的配置文件名称, 平台在创建驱动时会查找驱动打包文件中查找该文件. 一般固定填写 service.yml
ConfigType: service.yml
# 必填项. 固定为 driver
GroupName: driver
# 必填项. 驱动启动命令. 还可以添加一些启动参数
# 如果使用 PyInstaller 工具打包, 直接运行打包后的可执行文件即可.
# 例如, 打包后的可执行文件名为 my-program.exe, 直接填写 Command: my-program.exe [args...]
Command: python main.py
```

4. 将所有资源打包为 `zip` 文件.

将项目中的源码或打包后的文件、`config.yml`、`service.yml` 和其它资源打包为 `zip` 文件, 平台会根据该文件进行安装. 建议打包后的 `zip` 文件结构如下:


opcua-driver-windows-... - 360压缩
文件
操作
设置
帮助
添加
解压到
一键解压
删除
图片压缩
New
工具
扫描
opcua-driver-windows-x86_64.zip - 解包大小为 43.9 MB
名称
压缩前
压缩后
类型
修改日期
.. (上级目录)
文件夹
config
文件夹
2023-03-30 15:16
opcua-driver.jar
43.9 MB
38.9 MB
Executable Jar File
2023-03-28 10:19
service.yml
1 KB
1 KB
Yaml 源文件
2023-03-28 10:21
大小: 38.9 MB 共 3 个文件和 1 个文件夹 压缩率 88.7%


opcua-driver-windows-... - 360压缩
文件
操作
设置
帮助
添加
解压到
一键解压
删除
图片压缩
New
工具
扫描
opcua-driver-windows-x86_64.zip\config - 解包大小为 43.9 MB
名称
压缩前
压缩后
类型
修改日期
.. (上级目录)
文件夹
application.yml
1 KB
1 KB
Yaml 源文件
2023-03-28 10:21
大小: 38.9 MB 共 3 个文件和 1 个文件夹 压缩率 88.7% 已经选择 1 个文件夹

linux系统打包

由于在 `linux` 系统中, 驱动程序是以 `容器` 的方式运行, 所以打包时需要先将驱动程序打包为 `docker` 镜像. 然后再将镜像文件和 `service.yml` 打包为 `.tar.gz` 压缩包. 具体打包步骤如下:

1. 将程序和相关资源打包. 可以使用 `PyInstaller` 等工具将程序打包为可执行文件, 或者直接使用带有 `python` 运行环境的镜像作为基础镜像进行构建自己的镜像. **注: 使用 `python main.py` 的方式运行程序时, 需要将 `python` 运行环境一起打包**

2. 准备 `Dockerfile` 文件. 以下是一个简单的 `Dockerfile` 文件示例, 具体内容根据自身的需求进行修改:

```
# 根据自身的需求选择合适的基础镜像
FROM python:3.10.12-alpine

WORKDIR /app

# 如果驱动配置文件在 jar 文件外面
COPY config.yml /app/
# 复制 jar 文件
COPY main.py /app/main.py

# 启动命令，根据自身的需求添加启动参数
# 注：必须使用 ENTRYPOINT 启动程序
ENTRYPOINT ["python", "main.py"]
```

3. 构建 `docker` 镜像.

使用上一步中的 `Dockerfile` 文件构建 `docker` 镜像, 具体命令如下:

```
docker build -t myDriver:1.0.0 .
```

4. 导出 `docker` 镜像并压缩.

```
docker save myDriver:1.0.0 | gzip > myDriver.tar.gz
```

5. 准备驱动安装配置文件 `service.yml` . 该文件的格式与 windows系统打包 中的第三步中的 `service.yml` 文件格式相似但又有区别. 具体格式如下:

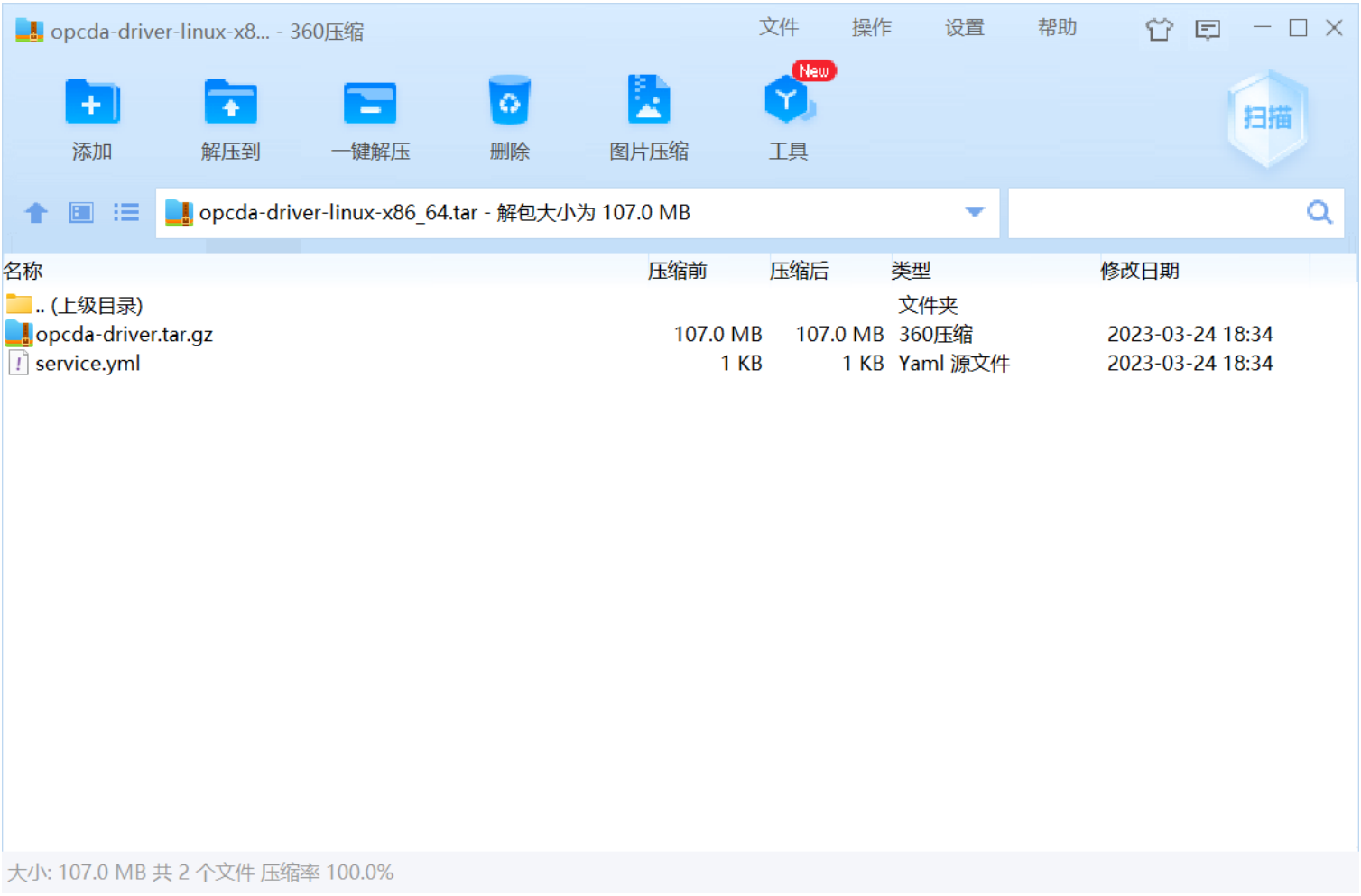
```
# 必填项. 驱动名称
Name: python_sdk_demo_driver
# 非必填项. 如果驱动对外提供 rest 服务, 则需要填写 rest 接口的统一路径前缀.
# 当填写该配置项时, 平台会自动在网关中添加该路径的路由, 并将请求转发到该驱动, 代理端口为
```

```
application.yml 文件中的 server.port 配置项.
Path: /python_sdk_demo_driver
# 必填项. 例如: 1.0.0, 通常用镜像版本号一致
Version: 1.0.0
# 非必填项.
Description: 驱动描述信息
# 必填项. 固定为 driver
GroupName: driver
# 容器端口映射类型, 非必填项. 如果驱动需要对外提供 rest 服务, 或暴露端口时, 需要填写该配置项.
# 可选项有 None Internal External
#
# None: 不暴露端口
# Internal: 只在平台内部暴露端口. 一般为驱动对外提供 rest 服务时, 将端口映射到网关上, 填写为
Internal 即可.
# External: 对外暴露端口. 一般为驱动为作 server 端, 需要对外暴露端口以供设备连接, 此时该端口会暴露
在宿主机上, 填写为 External 即可.
Service: Internal
# 非必填项. 暴露的端口列表
Ports:
  - Host: "8558"            # 映射到宿主机的端口号, 如果不填写, 则会随机分配一个端口号
    Container: "8558"       # 容器内部的端口号, 即驱动服务监听的端口号
    Protocol: ""            # 协议类型, 可选项有 TCP UDP, 如果不填写, 则默认为 TCP
```

6. 将所有资源打包为 `gzip` 文件. 将 `docker镜像` 和 `service.yml` 文件打包为 `gzip` 文件. 打包命令如下:

```
tar czvf myDriver-linux.tar myDriver.tar service.yml
```

打包后的 `gzip` 文件结构如下:



> **INFO**
>
> `linux` 系统整个打包过程对应的命令如下所示:

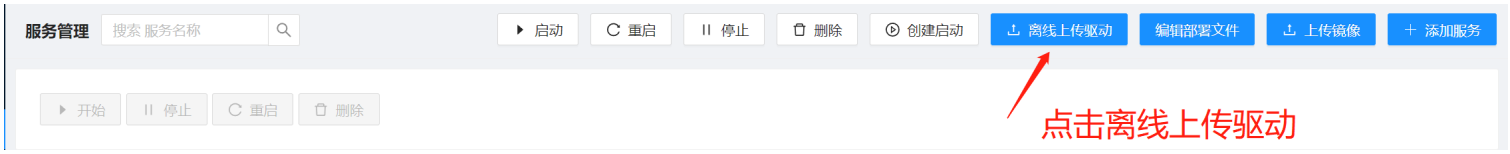
```
# 将驱动打包为镜像
docker build -t myDriver:1.0.0 .

# 导出镜像并压缩
docker save myDriver:1.0.0 | gzip > myDriver.tar.gz

# 将镜像文件和 service.yml 打包
tar czvf myDriver-linux.tar  myDriver.tar.gz service.yml

# 最后得到 myDriver-linux.tar.gz 压缩文件
```

7. 部署

将上一步骤中得到的驱动安装包通过 `运维管理系统` 上传到平台, 平台会自动解析并安装驱动. 安装成功后, 就可以在项目中使用该驱动了.

安装驱动

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `离线上传驱动` 按钮, 选择上一步中得到的 `myDriver-linux.tar.gz` 文件, 点击 `确定` 按钮, 平台会自动解析并安装驱动.



如果驱动安装失败, 可以在 `运维管理系统` 的 `首页` 中查看详细的日志信息.

> **INFO**
>
> 不同版本的平台, `离线上传驱动` 按扭的位置可能不同.

使用驱动

当驱动成功安装到平台后, 就可以在项目中使用该驱动了.

具体使用方法请参考 驱动管理.

数据接入驱动接口说明

驱动接口定义

```python
class DriverApp:
    """
    驱动程序接口. 该接口定义驱动程序与平台交互的方法. 所有驱动程序都需要实现该接口.
    """

    @abstractmethod
    def get_version(self) -> str:
        """
        获取驱动版本号. 例如: v4.0.0
        :return: 驱动版本号
        """
        pass

    @abstractmethod
    def http_proxy_enabled(self) -> bool:
        """
        是否启用 HTTP 请求代理. 用于代理驱动中的 HTTP 请求, 可以将驱动中的一些功能通过 grpc 代理暴露给平台, 由平台代理驱动中的 HTTP 请求.
        :return: 是否启用 HTTP 请求代理. 返回 False 表示不启用, 返回 True 表示启用.
        """
        return False

    @abstractmethod
    async def http_proxy(self, request_type: str, headers: dict[str, List[str]], data: str) -> any:
        """
        HTTP 请求代理. 用于代理驱动中的 HTTP 请求, 例如: 获取设备数据, 获取设备属性, 控制设备等.\r\n
        当前端请求驱动时, 会将请求转发给平台的 driver 服务, 由 driver 服务通过 listener 协议转发到驱动中, 由驱动自行处理请求并返回响应结果.\r\n
        前端 <- http -> driver <- listener -> DriverApp
        :param request_type: 请求类型标识, 每个请求类型标识对应一个请求. 例如: getDevices 表示查询设备列表请求.
        :param headers: 请求头, 即前端请求时传递的 headers.
        :param data: 请求参数.
        :return: 请求响应结果
        """
        pass

    @abstractmethod
    async def start(self, config: str):
        """
        启动驱动. 当驱动程序启动时, SDK 会连接平台 driver 服务, 连接成功后会调用此方法.
        :param config: 驱动配置, 即使用该驱动实例的工作表、设备等信息.
        :return:
        """
        pass
```

```python
    @abstractmethod
    async def stop(self):
        """
        停止驱动. 当驱动程序停止时，SDK 会断开平台 driver 服务的连接，断开连接后会调用此方法.
        :return:
        """
        pass

    @abstractmethod
    async def run(self, serial_no: str, table_id: str, device_id: str, command: str) -> any:
        """
        执行设备命令. 当前端需要控制设备时，会调用此方法.
        :param serial_no: 平台指令序列号，每次发送指令时，平台会生成一个唯一的序列号，用于标识本次指令.
        :param table_id: 设备所属工作表 ID.
        :param device_id: 设备编号.
        :param command: 指令内容.
        :return: 指令执行结果
        """
        pass

    @abstractmethod
    async def batch_run(self, serial_no: str, table_id: str, device_ids: List[str], command: str) -> any:
        """
        批量执行设备命令. 当前端需要控制多个设备时，会调用此方法.
        :param serial_no: 平台指令序列号，每次发送指令时，平台会生成一个唯一的序列号，用于标识本次指令.
        :param table_id: 设备所属工作表 ID.
        :param device_ids: 设备编号列表.
        :param command: 指令内容.
        :return: 指令执行结果
        """
        pass

    @abstractmethod
    async def write_tag(self, serial_no: str, table_id: str, device_id: str, tag: str) -> any:
        """
        向设备的指定数据点写入数据. 只有数据点的 rw 为 true 的数据点，才能被写入数据.
        :param serial_no: 平台指令序列号，每次发送指令时，平台会生成一个唯一的序列号，用于标识本次指令.
        :param table_id: 设备所属工作表 ID.
        :param device_id: 设备编号列表.
        :param tag: 数据点信息及写入的值.
        :return: 指令执行结果
        """
        pass
```

```
    @abstractmethod
    async def debug(self, debug: str) -> str:
        """
        驱动调试. 当前端需要调试驱动时, 会调用此方法.(该方法暂未开放)
        :param debug: 调试内容.
        :return: 调试结果
        """
        pass

    @abstractmethod
    async def schema(self) -> str:
        """
        获取驱动的 schema. 当前端需要获取驱动的 schema 时, 会调用此方法.
        :return: 驱动的 schema 定义
        """
```

工厂类定义

```
class DriverAppFactory:
    """
    驱动程序实例构造工厂接口. 该接口定义驱动程序实例构造工厂的方法. 所有驱动程序实例构造工厂都需要实现该接口.
    """

    def create(self, service_id: str, data_sender: DataSender) -> DriverApp:
        """
        驱动程序实例构造工厂
        :param service_id: 驱动实例ID
        :param data_sender: 数据发送器
        :return: 驱动程序实例
        """
        pass
```

示例

```
class MyDriverApp(DriverApp):
    """
    驱动程序类
    """


class MyDriverAppFactory(DriverAppFactory):
    """
    驱动实例构造工厂
    """
```

```
    def create(self, service_id: str, data_sender: DataSender) -> DriverApp:
        return MyDriverApp()
```

驱动与平台交互接口说明

```
class DataSender:
    """
    数据发送器，用于向平台发送数据
    """

    async def write_point(self, point: Point):
        """
        向平台发送设备采集到的数据
        :param point: 设备采集到的数据
        :return:
        :raise ValueError: point is None
        """
        pass

    def write_event(self, event: Event) -> Response:
        """
        向平台发送事件信息
        :param event: 事件信息
        :return: 事件发送结果
        :raises ValueError: event is None
        """
        pass

    def update_table_data(self, table_id: str, device_id: str, fields: dict[str, any]) ->
Response:
        """
        更新设备数据
        :param table_id: 设备所属工作表标识
        :param device_id: 设备编号
        :param fields: 更新的字段信息. key 为字段名, value 为字段值.
        :return: 设备信息修改结果
        :raise ValueError: the table_id, device_id or fields is None or empty
        """
        pass

    def write_run_log(self, log: RunLog) -> Response:
        """
        向平台发送指令执行日志
        :param log: 执行日志
        :return: 日志发送结果
        :raise ValueError: log is None
        """
```

```
        pass

    def log_debug(self, table_id: str, device_id: str, message: str):
        """
        发送 DEBUG 日志. 该日志信息可以在 "设备配置" 中的 "设备调试" 面板中查看.
        :param table_id: 设备所属工作表标识
        :param device_id: 设备编号
        :param message: 日志内容
        :return:
        """
        pass

    def log_info(self, table_id: str, device_id: str, message: str):
        """
        发送 INFO 日志. 该日志信息可以在 "设备配置" 中的 "设备调试" 面板中查看.
        :param table_id: 设备所属工作表标识
        :param device_id: 设备编号
        :param message: 日志内容
        :return:
        """
        pass

    def log_warn(self, table_id: str, device_id: str, message: str):
        """
        发送 WARN 日志. 该日志信息可以在 "设备配置" 中的 "设备调试" 面板中查看.
        :param table_id: 设备所属工作表标识
        :param device_id: 设备编号
        :param message: 日志内容
        :return:
        """
        pass

    def log_error(self, table_id: str, device_id: str, message: str):
        """
        发送 ERROR 日志. 该日志信息可以在 "设备配置" 中的 "设备调试" 面板中查看.
        :param table_id: 设备所属工作表标识
        :param device_id: 设备编号
        :param message: 日志内容
        :return:
        """
        pass
```

驱动配置说明

以下是完整的驱动配置文件, 请参考该配置文件进行配置.

```
```

```yaml
id: python_sdk_demo_driver      # 驱动ID
name: PythonSDK示例驱动          # 驱动名称
# 驱动管理服务配置信息
driver-grpc:
  # 在开发时，需要配置为平台的地址
  host: 192.168.11.101
  # 端口默认为 9224，一般无须修改. 如果有修改，可在运维管理系统中查看 `driver` 服务的端口号
  port: 9224
# 平台 MQTT 配置信息
mq:
  mqtt:
    # 在开发时，需要配置为平台的地址
    host: 192.168.11.101
    port: 1883
    username: admin
    password: public
# 如果驱动对外提供 web 接口服务，则需要按以下方式配置 web 服务的端口号
server:
  port: 8080
```

windows系统打包发布时的驱动配置

```yaml
id: python_sdk_demo_driver
name: PythonSDK示例驱动
driver-grpc:
  host: 127.0.0.1
  port: 9224
mq:
  mqtt:
    host: 127.0.0.1
    port: 1883
    username: admin
    password: public
server:
  port: 8080
```

linux系统打包发布时的驱动配置

```yaml
id: python_sdk_demo_driver
name: PythonSDK示例驱动
driver-grpc:
  host: driver
  port: 9224
mq:
  mqtt:
```

```
    host: mqtt
    port: 1883
    username: admin
    password: public
server:
  port: 8080
```

流程插件开发

本文将会详细介绍如何使用 `Python SDK` 开发流程插件. 示例项目目上传至 https://github.com/air-iot/sdk-python-examples/tree/master/flow_plugin.

介绍

`流程插件` 是扩展 `流程引擎` 中的节点的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程插件来实现自定义的功能.

> **INFO**
>
> `流程插件` 只是扩展流程功能的方式之一. 除了 `流程插件` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

根据自己的习惯创建项目.

2. 引入SDK

安装使用

> **INFO**
>
> 流程插件二次开发相关内容都在 `flow_plugin` 子包内.

3. 实现流程插件接口

`SDK` 中定义了 `流程插件接口`, 该接口是平台与插件交互的桥梁. 开发者需要实现这个接口. 接口定义及详细说明见 流程插件接口说明.

流程插件启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程执行到该插件对应的节点时, 会发送请求给该插件对应的程序. `SDK` 接收到请求后会调用对应的插件实现, 并将插件处理结果返回给流程引擎.

4. 配置插件

插件配置主要是插件与平台的连接配置.

```
flow-engine:
  host: 192.168.11.101              # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
  connect-timeout: 15s              # 连接超时
  retry-interval: 30s               # 重连间隔
  heartbeat-interval: 30s           # 心跳间隔
```

windows系统打包发布时的插件配置

```
flow-engine:
  host: 127.0.0.1                   # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
```

linux系统打包发布时的插件配置

```
flow-engine:
  host: flow-engine                 # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
```

> **INFO**
>
> `connect-timeout`、`retry-interval` 和 `heartbeat-interval` 通常情况下不需要修改.

5. 打包

流程插件打包就是将开发完成的程序打包为可以在平台部署的服务. 打包方式与 数据接入驱动开发-打包 中的打包方式基本一致， 但 `service.yml` 文件的内容稍有不同.

windows系统插件配置文件

```
```

```yaml
# 必填项. 服务名称
Name: myPlugin
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 插件描述信息
# 插件的配置文件名称, 平台在安装插件服务时会查找打包文件中查找该文件. 一般固定填写 config.yml
ConfigType: config.yml
# 必填项. 固定为 server
GroupName: server
# 必填项. 启动命令.
Command: python main.py
```

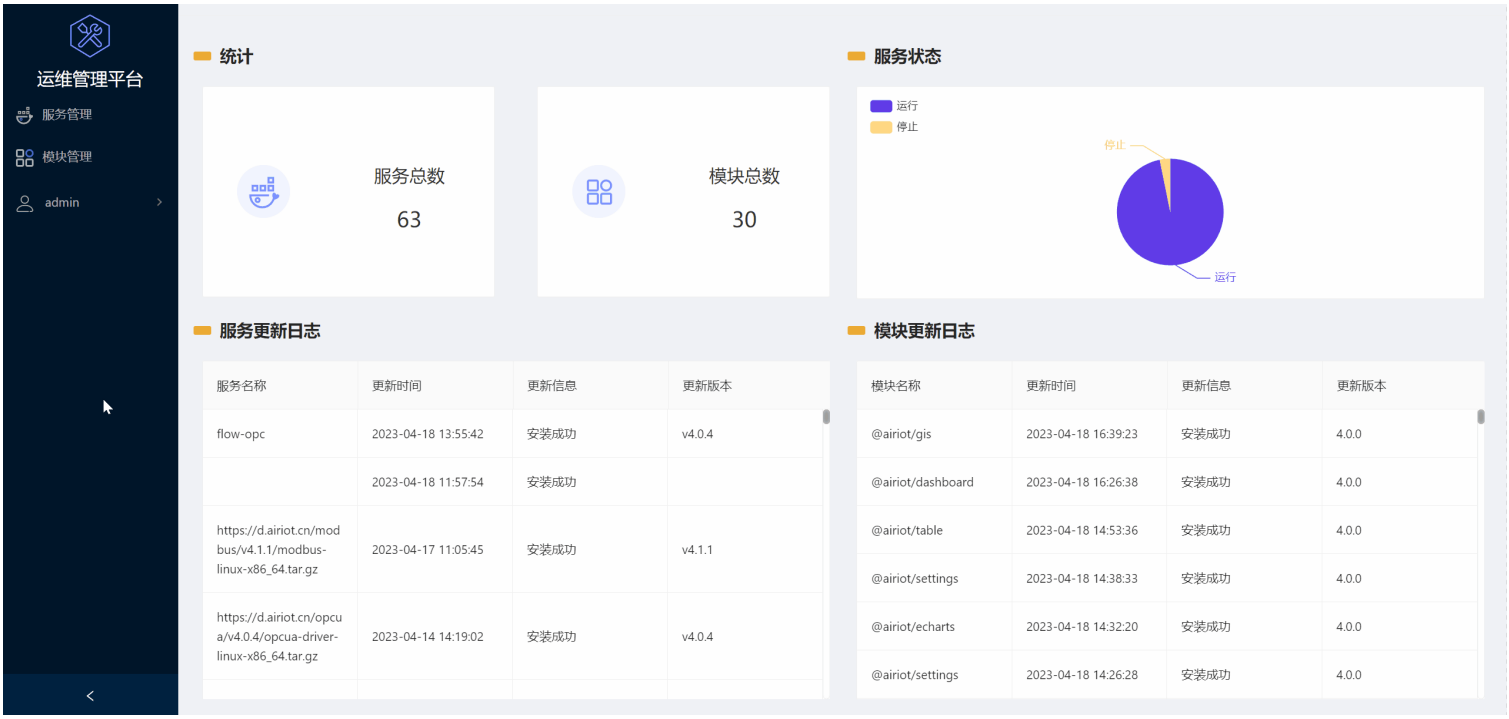
linux系统插件配置文件

```yaml
# 必填项. 服务名称
Name: myPlugin
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 插件描述信息
# 必填项. 固定为 server
GroupName: server
# 固定为 None
Service: None
```

6. 部署

`流程插件` 的部署方式与 `数据接入驱动` 不同, 每个流程插件都是一个独立的服务, 在整个平台中只有一个实例. 部署过程如下:

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `添加服务` 按钮, 然后选择 `离线添加`.
4. 点击 `上传` 按扭, 选择刚刚打包好的流程插件压缩包, 然后点击 `提交` 按钮.



> **INFO**
>
> 如果 流程插件 部署失败, 可以在 运维管理系统 的首页中查看日志.

流程插件接口说明

```python
class FlowPlugin:
    """
    流程插件接口
    """

    @abstractmethod
    def get_name(self) -> str:
        """
        插件名称
        :return:
        """
        pass

    @abstractmethod
    def get_type(self) -> FlowPluginType:
        """
        插件类型
        :return:
        """
        pass
```

```
    @abstractmethod
    def on_connection_state_changed(self, state: bool):
        """
        当与流程引擎的连接状态发生变化时调用
        :param state: 变化后的连接状态
        :return:
        """
        pass

    @abstractmethod
    def start(self):
        """
        当插件启动时调用.
        该方法只会调用一次，并且在 'on_connection_state_changed' 方法之前执行
        :return:
        """
        pass

    @abstractmethod
    def stop(self):
        """
        当流程插件服务停止时执行的操作. 该方法只会调用一次
        :return:
        """
        pass

    @abstractmethod
    async def execute(self, request: FlowTask) -> FlowResult:
        """
        执行流程插件
        :param request: 请求信息
        :return: 执行结果
        :raise Exception: 执行失败时抛出异常
        """
        pass
```

FlowTask 定义

```
class FlowTask:
    """
    流程插件任务请求信息
    """

    # 当前流程所属项目ID
    projectId: str
    # 当前流程ID
    flowId: str
    # 流程实例ID
```

```
    job: str
    # 当前节点ID
    elementId: str
    # 节点实例ID
    elementJob: str
    # 节点配置信息
    config: bytes
```

FlowResult 定义

```
class FlowResult:
    """
    流程插件执行结果
    """
    # 说明信息
    message: str
    # 详细信息
    details: str
    # 执行结果数据
    data: dict
```

平台接口客户端

介绍

平台客户端 SDK 用于访问平台接口，提供了平台接口的 Python 实现, 可以很方便实现第三方系统与平台的集成。平台客户端 SDK 中包括常用的 `工作表数据管理`、`系统变量(数据字典)`、`数据点数据查询` 等接口. 示例项目目上传至 https://github.com/air-iot/sdk-python-examples/tree/master/sdk-client-http-examples.

使用方式

1. 在项目中引入依赖

安装使用

> **INFO**
>
> 平台接口二次开发相关内容都在 `client` 子包内.

2. 初始化平台客户端

配置平台访问信息

使用平台客户端时, 需要提供平台的访问信息, 包括平台网关地址、授权信息等, 这些信息在创建客户端对象时需要. 平台网关地址默认为 `http://IP:31000`, 其中 `IP` 为部署平台服务器的IP地址, 端口默认为 `31000`. 授权信息为平台中提供的二次开发访问信息 `appKey` 和 `appSecret`.

> **INFO**
>
> 当授权类型为 `project` 时, 只能访问该项目中的相关资源。 当授权类型为 `tenant` 时, 可以访问该租户下的所有项目内的资源。
>
> **关于如何创建二次开发应用授权, 请查看 二次开发-第三方应用添加。**

创建客户端对象

在使用平台客户端之前, 需要创建客户端对象, 该对象包含了平台接口的访问地址、授权信息等。创建客户端对象示例如下:

```python
from airiot_python_sdk.client.authentication import create_auth_from_etcd
from airiot_python_sdk.client.sync import SyncClients

# 创建客户端对象
# base_url 为平台网关地址, 例如: http://127.0.0.1:31000
# auth_type 为授权类型, 可选值为: tenant, project
# app_key 为二次开发应用的 appKey
# app_secret 为二次开发应用的 appSecret
# project_id 为项目ID, 当授权类型为 project 时, 该参数必填
clients = SyncClients(base_url = "http://127.0.0.1:31000", auth_type = "project",
    app_key = "3b7f2831-0089-aa20-55b0-4ed1f65d21a9",
    app_secret = "69ea13a9-bfb4-2276-c514-0d4a04c00354", project_id = "default")
```

> **INFO**

3. 使用平台客户端

在上一步聚中创建了客户端对象 `clients`, 该对象是平台接口客户端集合, 通过该对象可以获取不同资源类型的客户端. 该对象包含的客户端接口如下:

```python
class Clients:
    """
    平台客户端列表，用于获取各种资源的客户端
    """

    def get_system_variable_client(self) -> SystemVariableClient:
        """
        获取系统变量(数据字典)客户端. 该客户端用于操作系统变量(数据字典)
        :return:
        """
        pass

    def get_latest_client(self) -> LatestClient:
        """
        获取数据点最新数据客户端. 该客户端用于查询数据点的最新数据
        :return:
        """
        pass

    def get_timing_data_client(self) -> TimingDataClient:
        """
        获取时序数据查询客户端. 该客户端用于查询数据点的历史数据
        :return:
        """
```

```
        pass

    def get_worktable_client(self) -> WorkTableDataClient:
        """
        获取工作表操作客户端. 该客户端用于操作工作表数据
        :return:
        """
        pass
```

4. 统一响应格式说明

所有客户端接口方法统一返回 `Response` 结构, 结构如下:

```
class Response(Generic[T]):
    """
    请求响应
    """

    # 请求是否成功标识
    success: bool = True
    # 请求响应状态码. 请求成功时为200, 请求失败时为相应的业务错误码
    code: int = 200
    # 响应信息. 当请求失败时保存错误信息
    message: str = "OK"
    # 详细信息. 当请求失败时保存详细错误信息
    detail: str = ""
    # 字段名称. 当请求由于字段错误时保存字段名称. 例如: 参数字段校验失败
    field: str = ""
    # 请求响应数据
    data: Optional[T] = None

    @property
    def full_message(self) -> str:
        if self.detail is not None and len(self.detail) != 0:
            return f"{self.message}, {self.detail}"
        return self.message
```

5. 示例

下面以操作系统变量(数据字典)为例, 说明如何使用平台客户端.

```
# 创建客户端对象, 见步骤 2

# 获取系统变量客户端
```

```
sv_client = clients.get_system_variable_client()

# 创建系统变量
response = sv_client.create("default", SystemVariable(uid="maxScore", name="最高得分",
type="number", value=99.5))

if not response.success:
    raise Exception(f"创建系统变量失败, {response.full_message}")

# 查询数据变量的数据
response = sv_client.get_by_uid("default", "maxScore")
if not response.success:
    raise Exception(f"查询系统变量失败, {response.full_message}")
elif not response.data or len(response.data) == 0:
    raise Exception("未查询到系统变量")

print("最高得分:", response.data[0].value)

# 更新系统变量的值
response = sv_client.update_value("default", "maxScore", 99.7)
if not response.success:
    raise Exception(f"更新系统变量的值失败, {response.full_message}")


# 删除系统变量
response = sv_client.delete_by_uid("default", "maxScore")
if not response.success:
    raise Exception(f"删除系统变量失败, {response.full_message}")
```

客户端列表

系统变量(数据字典)

系统变量客户端提供对系统变量的增删改查等操作. 支持的操作如下:

```
class SystemVariableClient:
    """
    系统变量(数据字典)客户端接口
    """

    def get(self, project_id: str, query: Query, headers: Optional[dict[str, str]] = None) -
> Response[
        list[SystemVariable]]:
        """
        自定义查询系统变量(数据字典)列表
        :param headers: 自定义请求头
```

```
        :param project_id: 项目ID
        :param query: 查询参数
        :return:
        """
        pass

    def get_by_id(self, project_id: str, id: str, headers: Optional[dict[str, str]] = None)
-> Response[SystemVariable]:
        """
        根据变量唯一标识获取系统变量信息
        :param headers: 自定义请求头
        :param project_id: 项目ID
        :param id: 唯一标识
        :return:
        """
        pass

    def get_by_uid(self, project_id: str, uid: str, headers: Optional[dict[str, str]] =
None) -> Response[
        list[SystemVariable]]:
        """
        根据自定义变量标识获取系统变量信息
        :param headers: 自定义请求头
        :param project_id: 项目ID
        :param uid: 自定义系统变量标识
        :return:
        """
        pass

    def get_by_uids(self, project_id: str, uids: list[str], headers: Optional[dict[str,
str]] = None) -> Response[
        list[SystemVariable]]:
        """
        根据自定义变量标识批量获取系统变量信息
        :param project_id: 项目ID
        :param uids: 自定义系统变量标识列表
        :param headers: 自定义请求头
        :return:
        """
        pass

    def get_by_name(self, project_id: str, name: str, headers: Optional[dict[str, str]] =
None) -> Response[
        list[SystemVariable]]:
        """
        根据变量名称获取系统变量信息
        :param headers: 自定义请求头
        :param project_id: 项目ID
        :param name: 系统变量名称
        :return:
```

```
        """
        pass

    def get_by_names(self, project_id: str, names: list[str], headers: Optional[dict[str,
str]] = None) -> Response[
        list[SystemVariable]]:
        """
        根据变量名称批量获取系统变量信息
        :param headers: 自定义请求头
        :param project_id: 项目ID
        :param names: 系统变量名称列表
        :return:
        """
        pass

    def create(self, project_id: str, variable: SystemVariable, headers: Optional[dict[str,
str]] = None) -> Response[
        InsertResult]:
        """
        创建系统变量
        :param headers: 自定义请求头
        :param project_id: 项目ID
        :param variable: 系统变量信息
        :return:
        """
        pass

    def update(self, project_id: str, variable: SystemVariable, headers: Optional[dict[str,
str]] = None) -> Response:
        """
        更新系统变量
        :param headers: 自定义请求头
        :param project_id: 项目ID
        :param variable: 系统变量信息
        :return:
        """
        pass

    def update_value(self, project_id: str, id: str, value: any, headers: Optional[dict[str,
str]] = None) -> Response:
        """
        更新系统变量值
        :param headers: 自定义请求头
        :param project_id: 项目ID
        :param id: 系统变量唯一标识
        :param value: 系统变量值
        :return:
        """
        pass
```

```
    def delete(self, project_id: str, id: str, headers: Optional[dict[str, str]] = None) ->
Response:
        """
        删除系统变量
        :param headers: 自定义请求头
        :param project_id: 项目ID
        :param id: 系统变量唯一标识
        :return:
        """
        pass

    def delete_by_uid(self, project_id: str, uid: str, headers: Optional[dict[str, str]] =
None) -> Response:
        """
        根据系统变量 UID 删除系统变量
        :param project_id: 项目ID
        :param uid: 系统变量 UID
        :param headers: 自定义请求头
        :return: 删除结果
        """
        pass
```

工作表记录

工作表记录客户端提供了对工作表内数据的增删改查等操作. 支持的操作如下:

```
class WorkTableDataClient:
    """
    工作表数据操作客户端
    """

    def query(self, project_id: str, table_id: str, query: Query, headers:
Optional[dict[str, str]] = None) -> Response[
        list[dict]]:
        """
        查询工作数据
        :param project_id: 项目ID
        :param table_id: 工作表标识
        :param query: 查询信息
        :param headers: 自定义请求头
        :return:
        """
        pass

    def query_by_id(self, project_id: str, table_id: str, row_id: str, headers:
Optional[dict[str, str]] = None) -> Response[dict]:
        """
```

```
        根据记录ID查询工作表记录
        :param project_id: 项目ID
        :param table_id: 工作表标识
        :param row_id: 记录ID
        :param headers: 自定义请求头
        :return: 如果记录存在，返回记录信息
        """

    def create(self, project_id: str, table_id: str, data, headers: Optional[dict[str, str]]
= None) -> Response[
        InsertResult]:
        """
        向工作表中写入一条记录
        :param project_id: 项目ID
        :param table_id: 工作表标识
        :param data: 记录数据. 可以为自定义对象，dict 或 str
        :param headers: 自定义请求头
        :return: 写入成功，返回新增记录的ID
        """
        pass

    def create_batch(self, project_id: str, table_id: str, data, headers: Optional[dict[str,
str]] = None) -> \
            Response[BatchInsertResult]:
        """
        批量向工作表中写入数据
        :param project_id: 项目ID
        :param table_id: 工作表标识
        :param data: 记录数据列表，必须为 list 或 str
        :param headers: 自定义请求头
        :return: 写入成功，返回新增记录的ID
        """
        pass

    def update(self, project_id: str, table_id: str, row_id: str, data,
               headers: Optional[dict[str, str]] = None) -> Response:
        """
        更新工作表中记录
        :param project_id: 项目ID
        :param table_id: 工作表标识
        :param row_id: 记录ID
        :param data: 更新数据，可以为自定义对象，dict 或 str
        :param headers: 自定义请求头
        :return:
        """
        pass

    def update_many(self, project_id: str, table_id: str, filter_query: Query, data,
                    headers: Optional[dict[str, str]] = None) -> Response:
        """
```

```
        批量更新表中记录. 会更新所有匹配的记录，如果过滤条件为空，则会更新表中所有的数据
        :param project_id: 项目ID
        :param table_id: 工作表标识
        :param filter_query: 更新记录过滤器，只会使用 Query 对象中的 filter 信息
        :param data: 更新的数据，可以为自定义对象，dict 或 str
        :param headers: 自定义请求头
        :return:
        """
        pass

    def delete_by_id(self, project_id: str, table_id: str, row_id: str,
                     headers: Optional[dict[str, str]] = None) -> Response:
        """
        根据记录ID删除表中记录
        :param project_id:  项目ID
        :param table_id: 工作表标识
        :param row_id: 记录ID
        :param headers: 自定义请求头
        :return:
        """

    def delete(self, project_id: str, table_id: str, filter_query: Query,
               headers: Optional[dict[str, str]] = None) -> Response:
        """
        批量删除工作表中记录. 会删除所有匹配的记录，如果过滤条件为空，则会删除表中所有的数据
        :param project_id: 项目ID
        :param table_id: 工作表标识
        :param filter_query: 删除记录过滤器，只会使用 Query 对象中的 filter 信息
        :param headers: 自定义请求头
        :return:
        """
        pass
```

最新数据查询

最新数据查询客户端提供了对数据点最新数据的查询操作, 支持对资产内指定数据点的查询、多个数据点的查询以及不同资产数据点的查询. 支持的操作如下:

```
class LatestClient:
    """
    查询数据点最新数据客户端接口
    """

    def get(self, project_id: str, query: LatestQueries, headers: Optional[dict[str, str]] =
None) -> Response[
        list[LatestData]]:
        """
```

```
        查询数据点最新数据
        :param project_id: 项目ID
        :param query: 查询的设备和数据点信息
        :param headers: 自定义请求头
        :return:
        """
        pass

    def get_by_device_id(self, project_id: str, table_id: str, device_id: str,
                         headers: Optional[dict[str, str]] = None) -> Response[
        list[LatestData]]:
        """
        根据设备编号查询所有数据点最新数据
        :param project_id: 项目ID
        :param table_id: 资产所属工作表标识
        :param device_id: 设备编号
        :param headers: 自定义请求头
        :return:
        """
        pass

    def get_device_tag(self, project_id: str, table_id: str, device_id: str, tag_id: str,
                       headers: Optional[dict[str, str]] = None) -> Response[
        list[LatestData]]:
        """
        根据设备编号和数据点标识查询最新数据
        :param project_id: 项目ID
        :param table_id: 资产所属工作表标识
        :param device_id: 设备编号
        :param tag_id: 数据点标识
        :param headers: 自定义请求头
        :return:
        """
        pass

    def get_device_tags(self, project_id: str, table_id: str, device_id: str, tag_ids:
list[str],
                        headers: Optional[dict[str, str]] = None) -> Response[
        list[LatestData]]:
        """
        查询一个设备下多个数据点的最新数据
        :param project_id: 项目ID
        :param table_id: 资产所属工作表标识
        :param device_id: 设备编号
        :param tag_ids: 数据点标识列表
        :param headers: 自定义请求头
        :return:
        """
        pass
```

```
    def get_every_device(self, project_id: str, table_id: str, device_ids: list[str],
tag_id: str,
                         headers: Optional[dict[str, str]] = None) ->
Response[list[LatestData]]:
        """
        查询多个设备的同一个数据点的最新数据
        :param project_id: 项目ID
        :param table_id: 资产所属工作表标识
        :param device_ids: 设备编号列表
        :param tag_id: 数据点标识
        :param headers: 自定义请求头
        :return:
        """
        pass
```

查询条件说明

```
class LatestQuery:
    """
    查询数据点最新数据请求参数. 查询时，可以查询设备下所有数据点的最新数据，也可以查询指定数据点的最新数据.
    如果查询设备下所有数据点的最新数据，需要设置 allTags 为 true. 如果查询指定数据点的最新数据，需要设置 tagId 为目标数据点标识.
    如果要查询设备的部分数据点的最新数据，可以添加多个 LatestQuery 对象到数组中.
    例如: [LatestQuery("SN001", tag_id: "tag1"), LatestQuery("SN001", tag_id: "tag2")]

    Attributes:
        tableId: 资产所属工作表标识，必填.
        id: 资产编号，必填.
        allTag: 查询所有数据点时设置为 True，选填.
        tagId: 查询指定数据点时设置为目标数据点的标识，选填.
    """

    # 资产所属工作表标识
    tableId: str
    # 资产编号
    id: str
    # 是否查询该设备下所有数据点
    allTag: Optional[bool] = None
    # 数据点标识
    tagId: Optional[str] = None


class LatestQueries:
    queries: list[LatestQuery]

    def __init__(self, queries: list[LatestQuery] = None):
        self.queries = [] if not queries else queries
```

```
    def append(self, query: LatestQuery):
        self.queries.append(query)
```

查询结果说明

每个数据点的数据为一条记录, 查询结果为多条记录组成的列表. 每条记录的结构如下:

```
class LatestData:
    """
    数据点最新数据查询结果. 查询结果中包含了数据点的标识, 最新数据的时间戳和值.

    Attributes:
        tableId: 资产所属工作表标识
        id: 资产编号
        tagId: 数据点标识
        time: 最新数据的时间戳
        value: 最新数据的值. 如果该数据点没有数据, 则为 None
    """

    # 资产所属工作表标识
    tableId: str
    # 资产编号
    id: str
    # 数据点标识
    tagId: str
    # 时间戳(ms)
    time: int
    # 最新数据的值
    value: any
```

历史数据查询

历史数据查询客户端用于查询数据点的历史数据信息, 支持的操作如下:

```
class TimingDataClient:
    """
    时序数据查询客户端
    """

    def query(self, project_id: str, query: TimingDataQueries, headers: Optional[dict[str,
str]] = None) -> Response[
        TimingData]:
        """
```

```
        查询时序数据
        :param project_id: 项目ID
        :param query: 查询信息
        :param headers: 自定义请求头
        :return:
        """
        pass
```

查询条件说明

为了方便构造查询条件, SDK 中提供了 `TimingDataQueries` 类来构造查询条件, 内容如下:

```
class Builder:
    """
    时序数据查询构建器
    """

    def select(self, *fields: str) -> 'Builder':
        """
        查询的字段列表
        :param fields: 字段名称列表
        :return:
        """
        pass

    def select_as(self, field: str, alias: str) -> 'Builder':
        """
        设置查询字段及别名. 例如: select_as("max(\"score\")", "maxScore")
        :param field: 字段名称
        :param alias: 别名
        :return:
        """
        pass

    def table(self, table_id: str) -> 'Builder':
        """
        设置要查询的工作表标识. 该字段为必填项
        :param table_id: 工作表标识
        :return:
        """
        pass

    def device(self, device_id: str) -> 'Builder':
        """
        设置要查询的资产编号
        :param device_id: 资产编号
        :return:
```

```
        """
        pass

    def department(self, *departments: str) -> 'Builder':
        """
        设置查询的资产所属部门
        :param departments: 部门ID列表
        :return:
        """
        pass

    def eq(self, tag: str, value: any) -> 'Builder':
        """
        相等条件
        :param tag: 数据点或标签名称
        :param value: 比较值
        :return:
        """
        pass

    def not_eq(self, tag: str, value: any) -> 'Builder':
        """
        不相等条件
        :param tag: 数据点或标签名称
        :param value: 比较值
        :return:
        """
        pass

    def lt(self, tag: str, value: any) -> 'Builder':
        """
        小于条件
        :param tag: 数据点或标签名称
        :param value: 比较值
        :return:
        """
        pass

    def lte(self, tag: str, value: any) -> 'Builder':
        """
        小于等于条件
        :param tag: 数据点或标签名称
        :param value: 比较值
        :return:
        """
        pass

    def gt(self, tag: str, value: any) -> 'Builder':
        """
        大于条件
```

```
        :param tag: 数据点或标签名称
        :param value: 比较值
        :return:
        """
        pass

    def gte(self, tag: str, value: any) -> 'Builder':
        """
        大于等于条件
        :param tag: 数据点或标签名称
        :param value: 比较值
        :return:
        """
        pass

    def time_between(self, start_time: datetime, end_time: datetime) -> 'Builder':
        """
        设置时间范围
        :param start_time: 开始时间
        :param end_time: 结束时间
        :return:
        """
        pass

    def start_time(self, start_time: datetime) -> 'Builder':
        """
        设置开始时间
        :param start_time: 开始时间
        :return:
        """
        pass

    def end_time(self, end_time: datetime) -> 'Builder':
        """
        设置结束时间
        :param end_time: 结束时间
        :return:
        """
        pass

    def since(self, since: str, contains_start_time: bool = False) -> 'Builder':
        """
        设置查询最近一段时间的数据
        :param since: 时间段. 例如: 1d 表示 1 天, 1h 表示 1 小时, 1m 表示 1 分钟
        :param contains_start_time: 是否包含开始时间
        :return:
        """
        pass

    def group_by(self, column: str) -> 'Builder':
```

```
        """
        设置分组条件
        :param column: 分组字段
        :return:
        """
        pass

    def group_by_device(self) -> 'Builder':
        """
        按资产分组
        :return:
        """
        pass

    def group_by_time(self, interval: str) -> 'Builder':
        """
        按时间段分组
        :param interval: 时间段. 例如: 1h 表示 1 小时, 1d 表示 1 天
        :return:
        """
        pass

    def with_fill(self, value: str) -> 'Builder':
        """
        设置填充方式
        :param value: 填充方式. 可选值: none, null, previous, value 等
        :return:
        """
        pass

    def order_by_time_desc(self) -> 'Builder':
        """
        按时间降序排序
        :return:
        """
        pass

    def order_by_time_asc(self) -> 'Builder':
        """
        按时间升序排序
        :return:
        """
        pass

    def order_by_asc(self, field: str) -> 'Builder':
        """
        按指定字段升序排序
        :param field: 字段名称
        :return:
        """
```

```
        pass

    def order_by_desc(self, field: str) -> 'Builder':
        """
        按指定字段降序排序
        :param field: 字段名称
        :return:
        """
        pass

    def with_limit(self, limit: int) -> 'Builder':
        """
        限制查询的记录数量
        :param limit: 记录数量
        :return:
        """
        pass

    def with_offset(self, offset: int) -> 'Builder':
        """
        设置返回记录的偏移量
        :param offset: 偏移量
        :return:
        """
        pass
```

示例

```
query = (TimingDataQueries.new_builder()
            # 设置查询的工作表，即目标设备数据所在工作表的标识. 该字段为必填项
            .table("opcua130")
            # 查询的字段列表
            .select("id", "key1", "key2", "key3")
            # 查询的时间范围，例如: 查询最近 7 天的数据
            .since("7d")
            # 将数据按设备进行分组
            .group_by_device()
            .finish()
            .build())
```

> **INFO**
>
> 注意事项: `table` 用于设置查询数据所在工作表的标识, 该字段为必填项

其它

客户端 SDK 中提供了一些工具类, 用于简化和辅助开发。

查询构造器

客户端接口中的很多查询接口, 其结构比较复杂, 不易构造且容易出错, 为此 SDK 中提供了一个查询构造器 `Query` 来简化查询条件的构造.

查询参数的整体结构如下所示:

```
{
  "project": {},
  "filter": {},
  "sort": {},
  "limit": 30,
  "skip": 20,
  "withCount": true
}
```

字段说明如下:

- **`project`** 查询请求需要返回的字段列表. 例如: `{"id": 1, "name": 1, "address": {"city": 1}}`. `key` 为字段名, `value` 为 `1` 或 `Map`. 如果为为一级字段需要设置为 `1` 例如: `{"id": 1, "name": 1}`, 如果要返回嵌套对象内的字段, 则需要设置为 `Map`, 例如: `{"address": {"city": 1}}`
- **`filter`** 查询条件, 如果没有添加任何条件则查询全部数据. `key` 为字段名, `value` 为过滤的值或逻辑运算符, 例如: `{"name": "Tom", "age": {"$gt": 20, "$lt": 30}}`.
- **`sort`** 排序条件, `key` 为字段名, `value` 为 `1` 表示升序, `-1` 表示降序, 例如: `{"age": 1, "name": -1}`.
- **`limit`** 查询结果的最大数量, 可用于分页查询或限制返回的记录数量.
- **`skip`** 查询结果的偏移量, 即忽略前 N 记录, 可用于分页查询.
- **`withCount`** 是否返回符合条件的记录总数, 如果为 `true` 则会在查询结果记录数量会保存在响应对象 `ResponseDTO<T>` 中的 `count` 字段.

> **INFO**
>
> **注意事项**
>
> 1. 如果查询条件需要使用逻辑或, 可以在 `filter` 中添加 `$or` 字段, 其值为 `Map<String, Object>` 结构与 `filter` 一致, 任一条件成立时表示记录匹配.

2. 如果同一字段存在多个逻辑条件, 则需要将多个条件放在一个 `Map` 中, 例如: `{"age": {"$gt": 20, "$lt": 30}}`, 表示查询 `20 < age < 30` 的记录.

逻辑运算符

符号	说明	示例
$not	不相等, 与 `SQL` 中的 `<>` 作用相同	{"age": {"$not": 18}}
$in	在指定列表内, 与 `SQL` 中的 `in` 作用相同	{"id": {"$in": [1,3,4]}}
$nin	不在指定列表内, 与 `SQL` 中的 `not in` 作用相同	{"id": {"$nin": [1,3,4]}}
$gt	大于指定的值, 与 `SQL` 中的 `>` 作用相同	{"age": {"$gt": 18}}
$gte	大于等于指定的值, 与 `SQL` 中的 `>=` 作用相同	{"age": {"$gte": 18}}
$lt	小于指定的值, 与 `SQL` 中的 `<` 作用相同	{"age": {"$lt": 18}}
$lte	小于等于指定的值, 与 `SQL` 中的 `<=` 作用相同	{"age": {"$lte": 18}}
$regex	正则匹配, 与 `SQL` 中的 `like` 相似	{"name": {"$regex": "张"}}

示例

示例数据如下所示:

```
{
 "project": {
  "name": 1,
  "model": 1,
  "warning": {
   "hasWarning": 1
  }
 },
 "filter": {
  "name": "Tom",
  "fullname": {
   "$regex": "la"
  },
  "modelId": "5c6121b9982d2073b1a828a1",
```

```
  "warning": {
   "hasWarning": true
  },
  "$or": [{
   "score": {
    "$gt": 70,
    "$lt": 90
   }
  }, {
   "views": {
    "$gte": 1000
   }
  }]
 },
 "sort": {
  "age": -1,
  "posts": 1
 },
 "limit": 30,
 "skip": 20,
 "withCount": true
}
```

使用构造器创建上述查询条件的代码如下所示:

```
```

对照图

流程扩展节点开发

本文将会详细介绍如何使用 `Python SDK` 扩展流程节点. 示例项目目上传至 https://github.com/air-iot/sdk-python-examples/tree/master/flow_extension.

介绍

`流程扩展节点` 是扩展 `流程引擎` 中的节点的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程扩展节点来实现自定义的功能.

> **INFO**
>
> `流程扩展节点` 只是扩展流程功能的方式之一. 除了 `流程扩展节点` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

根据自己的习惯创建项目.

2. 引入SDK

1. 在项目中引入依赖

安装使用

> **INFO**
>
> 流程扩展节点二次开发相关内容都在 `flow_extension` 子包内.

3. Schema 定义

每个流程扩展节点需要提供一个 `schema` 定义, 该 `schema` 描述了当前扩展节点的输入参数定义. 详细说明点击查看. 示例如下:

```
{
  "type": "object",
  "properties": {
    "num1": {
      "title": "参数1",
      "type": "number"
    },
    "num2": {
      "title": "参数2",
      "type": "number"
    }
  },
  "required": [
    "num1",
    "num2"
  ]
}
```

4. 实现流程扩展节点接口

`SDK` 中定义了 `流程扩展节点接口`, 该接口是平台与扩展节点交互的桥梁. 开发者需要实现这个接口.

流程扩展节点启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程调用该扩展节点时, 会发送请求给该扩展节点对应的程序. `SDK` 接收到请求后会调用对应的流程扩展节点实现, 并将处理结果返回给流程引擎.

```
class FlowExtension:
    """
    流程扩展节点接口，该接口定义了流程扩展节点的基本行为
    """

    def id(self) -> str:
        """
        流程扩展节点标识. 要求该标识在平台中唯一
        :return: 节点标识
        """
        pass

    def name(self) -> str:
        """
        流程扩展节点名称
        :return: 节点名称
        """
        pass

    def start(self):
```

```
        """
        流程扩展节点启动时执行的动作，该方法会在节点启动时被调用，可以在该方法中执行一些初始化操作
        :return:
        """
        pass

    def stop(self):
        """
        流程扩展节点停止时执行的动作，该方法会在节点停止时被调用，可以在该方法中执行一些清理操作
        """
        pass

    async def schema(self) -> str:
        """
        获取流程扩展节点的 schema，该方法会在节点被调用时被调用，该方法必须是异步方法
        :return: schema 内容
        """
        pass

    async def run(self, params: dict) -> dict:
        """
        处理流程扩展节点的请求，该方法会在节点被调用时被调用，该方法必须是异步方法
        :param params: 请求参数. 参数内容由  schema 定义
        :return: 执行结果
        """
        pass
```

5. 配置流程扩展节点

流程扩展节点配置主要是扩展节点与平台的连接配置.

```
flow-engine:
  host: 192.168.11.101              # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
  connect-timeout: 15s              # 连接超时, 可选. 默认: 15s
  retry-interval: 30s               # 重连间隔, 可选. 默认: 30s
  heartbeat-interval: 30s           # 心跳间隔, 可选. 默认: 30s
  max-threads: 10                   # 最大线程数, 可选. 默认: 0, 即取当前主机的CPU核数
```

windows系统打包发布时的流程扩展节点配置

```
flow-engine:
  host: 127.0.0.1                   # 流程引擎服务地址
  port: 2333                        # 流程引擎服务端口
```

linux系统打包发布时的流程扩展节点配置

```
flow-grpc:
  host: flow-engine                    # 流程引擎服务地址
  port: 2333                           # 流程引擎服务端口
```

> **INFO**
>
> - `connect-timeout`、`retry-interval` 和 `heartbeat-interval` 通常情况下不需要修改.
> - 每次请求都是异步执行的, `max-threads` 用来设置异步执行的最大线程数. 如果不设置, 则取当前主机的CPU核数.

6. 打包

流程扩展节点打包就是将开发完成的程序打包为可以在平台部署的服务. 打包方式与 数据接入驱动开发-打包 中的打包方式基本一致.

windows系统流程扩展节点配置文件

```
# 必填项. 服务名称
Name: myFlowExtNode
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 扩展节点描述信息
# 流程扩展节点的配置文件名称, 平台在安装流程扩展节点服务时会查找打包文件中查找该文件. 一般固定填写 application.yml
ConfigType: application.yml
# 必填项. 固定为 server
GroupName: server
# 必填项. 启动命令. 还可以添加一些启动参数, 例如: -Xms512m -Xmx1024m
Command: python main.py
```

linux系统流程扩展节点配置文件

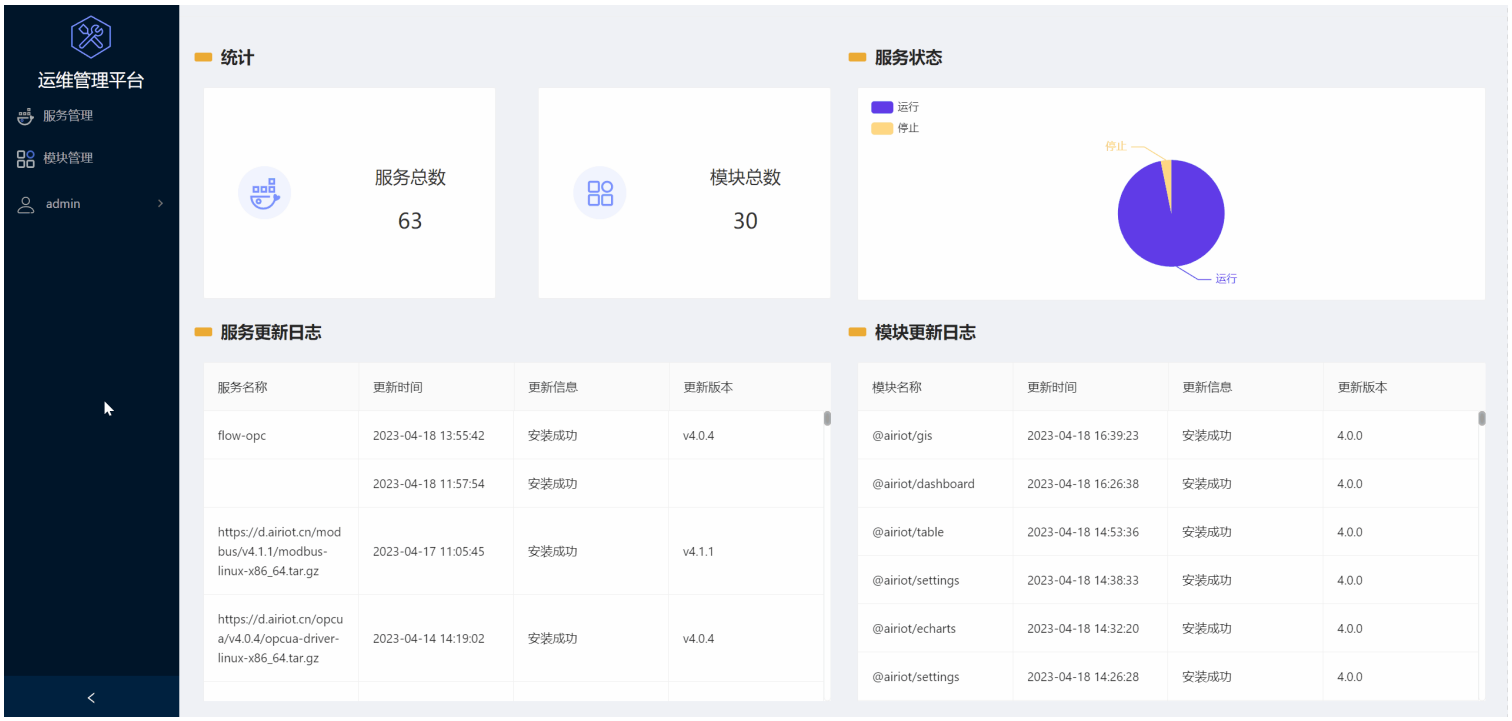
```
# 必填项. 服务名称
Name: myFlowExtNode
# 必填项. 例如: 1.0.0
Version: 1.0.0
# 非必填项.
Description: 扩展节点描述信息
# 必填项. 固定为 server
```

```
GroupName: server
# 固定为 None
Service: None
```

7. 部署

每个 `流程扩展节点` 都是一个独立的服务, 在整个平台中只有一个实例, 其部署方式与 `流程插件` 一致, 部署过程如下:

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `添加服务` 按钮, 然后选择 `离线添加`.
4. 点击 `上传` 按扭, 选择刚刚打包好的流程扩展节点压缩包, 然后点击 `提交` 按钮.



> **INFO**
>
> 如果 `流程扩展节点` 部署失败, 可以在 `运维管理系统` 的首页中查看日志.

算法服务开发

本文将会详细介绍如何使用 `Python SDK` 开发算法服务, 实现算法集成. 示例项目目上传至 https://github.com/air-iot/sdk-python-examples/tree/master/algorithm.

介绍

`算法服务` 是扩展或将已有 `算法` 集成到平台的一种方式. 在平台现有的功能不满足需求时, 可以通过开发算法服务来实现自定义的功能.

开发步骤

1. 创建项目

根据自己的习惯创建项目.

2. 引入SDK

安装使用

> **INFO**
>
> 算法服务二次开发相关内容都在 `algorithm` 子包内.

3. 定义 schema

`schema` 定义描述了该算法程序支持哪些算法函数, 以及每个算法函数的输入和输出参数定义. 每个算法程序可以包含多个算法函数, 每个算法函数的名称必须唯一, 详细说明参考算法schema说明. 示例如下:

```
[
    {
        "title": "函数1-加法",
        "function": "add",
        "input": {
            "type": "object",
            "properties": {
                "num1": {
                    "title": "参数1",
```

```
                    "type": "number"
                },
                "num2": {
                    "title": "参数2",
                    "type": "number"
                }
            },
            "required": [
                "num1",
                "num2"
            ]
        },
        "output": {
            "type": "object",
            "properties": {
                "num1": {
                    "title": "结果",
                    "type": "number"
                }
            }
        }
    },
    {
        "title": "函数2-绝对值",
        "function": "abs",
        "input": {
            "type": "object",
            "properties": {
                "num1": {
                    "title": "参数1",
                    "type": "number"
                }
            },
            "required": [
                "num1"
            ]
        },
        "output": {
            "type": "object",
            "properties": {
                "res": {
                    "title": "结果",
                    "type": "number"
                }
            }
        }
    },
    {
        "title": "函数3-获取当前系统时间",
        "function": "now",
```

```
        "input": {
            "type": "object",
            "properties": {},
            "required": []
        },
        "output": {
            "type": "object",
            "properties": {
                "sysdate": {
                    "title": "当前系统时间",
                    "type": "string"
                }
            }
        }
    },
    {
        "title": "函数4-接收Map",
        "function": "recvMap",
        "input": {
            "type": "object",
            "properties": {
                "name": {
                    "title": "姓名",
                    "type": "string"
                }
            },
            "required": ["name"]
        },
        "output": {
            "type": "object",
            "properties": {
                "result": {
                    "title": "输出结果",
                    "type": "string"
                }
            }
        }
    },
    {
        "title": "函数4-接收字符串",
        "function": "recvString",
        "input": {
            "type": "object",
            "properties": {
                "name": {
                    "title": "姓名",
                    "type": "string"
                }
            },
            "required": ["name"]
```

```
        },
        "output": {
            "type": "object",
            "properties": {
                "result": {
                    "title": "输出结果",
                    "type": "string"
                }
            }
        }
    }
]
```

4. 实现算法接口

`SDK` 中定义了 `算法服务接口`, 开发者需要实现这个接口.

```python
class AlgorithmApp:
    """
    算法应用程序基类，通过继承该类可以实现一个算法应用程序.
    """

    @abstractmethod
    def id(self) -> str:
        """
        算法应用程序ID
        :return:
        """
        pass

    @abstractmethod
    def name(self) -> str:
        """
        算法应用程序名称
        :return:
        """

    @abstractmethod
    def start(self):
        """
        算法应用程序启动时执行的方法. 可以在此方法中进行一些初始化操作.
        :return:
        """
        pass

    @abstractmethod
    def stop(self):
```

```
        """
        算法应用程序停止时执行的方法. 可以在此方法中进行一些清理操作.
        :return:
        """
        pass

    @abstractmethod
    async def schema(self) -> str:
        """
        算法应用程序的 schema 定义.
        :return:
        """
        pass

    @abstractmethod
    async def run(self, project_id: str, function: str, params: dict[str, any]) ->
[dict[str, any] | object]:
        """
        算法执行方法. 该方法会在算法执行请求到达时被调用.
        :param project_id: 请求执行算法的项目ID
        :param function: 请求执行的算法函数名
        :param params: 请求执行的算法参数
        :return: 算法执行结果
        """
        pass
```

@algorithm_function 注解

`SDK` 提供了 `@algorithm_function` 注解, 用于标识实现类中的方法对应 schema 的算法函数. 该注解只能用在方法上. 使用该注解修饰的方法, 必须满足以下条件:

- 必须为 `async` 异步方法.
- 必须带有 2 个参数
- 第 1 个参数必须为 `str` 类型， 用于接收发起请求的项目ID
- 第 2 个参数必须为 `dict` 类型, 用于接收请求参数
- 该函数的返回值的类型必须是 `dict`

算法调用请求执行流程

1. 当平台调用该算法服务时, `SDK` 会解析请求中的 `function` 信息.
2. 根据 `function` 的值, 在实现类中查找是否有使用 `@algorithm_function("算法函数名")` 注解修饰的方法. 如果找到了, 则会调用该方法, 并将方法的返回值作为算法执行结果返回给平台.
3. 如果没有找到使用 `@algorithm_function("算法函数名")` 修饰的方法时, 则会调用 `run` 方法, 并将 `function` 参数的值作为 `run` 方法的第二个参数传入, 由开发者根据 `function` 参数值执行对应的算法逻

辑. 最后将 `run` 方法的返回值作为算法执行结果返回给平台.

示例

```python
import datetime

from airiot_python_sdk.algorithm import AlgorithmApp, algorithm_function

class MyAlgorithmApp(AlgorithmApp):
    """
    自定义算法应用程序
    """

    def id(self) -> str:
        return "PythonAlgorithmSDKExample"

    def name(self) -> str:
        return "Python算法SDK示例程序"

    def start(self):
        logger.info("Python算法示例程序已启动")

    def stop(self):
        logger.info("Python算法示例程序已停止")

    async def schema(self) -> str:
        with open('schema.js', 'r', encoding="utf-8") as f:
            return f.read()

    async def run(self, project_id: str, function: str, params: dict[str, any]) ->
[dict[str, any] | object]:
        if function == "pow":
            return {"result": params["num1"] ** params["num2"]}
        rase Exception(f"不支持的算法函数 '{function}'")

    @algorithm_function(name="add")
    async def add(self, project_id: str, params: dict[str, any]) -> [dict[str, any] |
object]:
        num1 = params["num1"]
        num2 = params["num2"]
        return {"num1": num1 + num2}

    @algorithm_function(name="abs")
    async def abs(self, project_id: str, params: dict[str, any]) -> [dict[str, any] |
object]:
        num = params["num1"]
        return {"num": abs(num)}

    @algorithm_function(name="now")
```

```
    async def now(self, project_id: str, params: dict[str, any]) -> [dict[str, any] |
object]:
        return {"sysdate": datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")}
```

5. 算法配置信息

算法配置信息定义了算法的基本信息以及平台算法服务的连接信息. 整体配置如下:

```
algorithm:
  id: 自定义算法标识            # 必填
  name: 自定义算法名称          # 必填
  max-threads: 10              # 可选. 最大线程数, 默认: 0, 表示取 CPU 核心数
algorithm-grpc:      # 平台算法服务连接配置
  host: localhost    # 必填. 算法服务地址
  port: 9236         # 必填. 算法服务端口, 默认: 9236
```

6. 打包

具体打包步骤请参考 流程插件开发-打包 中的步骤.

windows系统打包发布时的算法配置

```
algorithm:
  id: 自定义算法标识
  name: 自定义算法名称
  max-threads: 10
algorithm-grpc:
  host: 127.0.0.1
  port: 9236
```

linux系统打包发布时的算法配置

```
algorithm:
  id: 自定义算法标识
  name: 自定义算法名称
  max-threads: 10
algorithm-grpc:
  host: algorithm
  port: 9236
```

7. 部署

具体部署步骤请参考 流程插件开发-部署 中的步骤.

常见问题

数据接入驱动开发常见问题

接收到配置信息为 null

现象描述

在 `DriverApp.start(Config)` 方法中, 接收到的驱动配置存在部分字段的值为 `null`.

解决办法

1. 检查驱动配置中是否该字段是否提供了有效值.
2. 检查驱动配置类(例如: Settings) 中对应的字段的名称与 schema 中的字段名称是否一致.
3. 检查驱动配置类(例如: Settings) 中对应的字段是否提供了 `setter` 方法.

MQTT不断的连接成功后立即断开

现象描述

在日志中不断的输出 `MQTTDataSender: 已连接` 和 `MQTTDataSender: 已断开` 的内容.

原因

有多个驱动程序使用了相同的驱动实例ID.

解决办法

更换驱动的实例ID, 保证每个驱动实例ID对应一个程序即可.

Go SDK 介绍

Go SDK 二次开发说明

数据接入驱动开发

介绍如何使用 Go SDK 开发自定义数据接入驱动

流程插件开发

介绍如何使用 Go SDK 开发自定义流程插件

流程扩展节点接入

介绍如何使用 Go SDK 开发自定义流程扩展节点接入服务

算法服务开发

介绍如何使用 Go SDK 开发算法服务

平台接口客户端

Go SDK 平台接口客户端

打包部署

本文将详细介绍平台打包及部署.

Go SDK 介绍

`Go SDK` 是 AIRIOT 物联网平台提供的 `Go` 语言的二次开发工具包, 可使用 Go SDK 调用平台开放的接口和实现对平台功能的扩展。

接下来会分别介绍如何使用 `Go SDK` 数据接入驱动、流程插件、流程扩展节点接入、算法集成 和通过平台接口客户端实现系统集成。

内容说明

内容	用途
数据接入驱动	实现从设备或其它平台系统采集数据
流程插件	扩展流程引擎中的节点
流程扩展节点接入	开发流程扩展节点接入服务
算法集成	扩展自定义算法
平台接口客户端	调用平台开放的API接口

使用方式

`Go SDK` 包已经上传github，直接使用npm安装SDK的开发包：

```
go get github.com/air-iot/sdk-go/v4
```

版本说明

SDK 版本	Go 版本
4.1.x	1.18+

示例仓库

示例代码: https://github.com/air-iot/sdk-go/tree/v4/example

数据接入驱动开发

本文将会详细介绍如何使用 `Go SDK` 开发自定义数据接入驱动.

介绍

`数据接入驱动` 是为实现从不同的协议、设备或其它平台系统采集数据而开发的特定程序. 每个 `数据接入驱动` 程序需要根据协议的特点实现数据采集功能，然后将采集到的数据通过 `Go SDK` 提供的接口发送到平台.

`Go SDK` 提供了 `数据接入驱动` 开发的相关内容. 包括驱动的接口定义, 以及与平台交互功能. 开发者只需要实现接口中的方法.

开发步骤

1. 创建项目

1. 创建目录
2. 进入项目执行初始化

```
go mod init driver-mqtt-demo
```

3. 创建 `main.go` 文件

2. 引入SDK

```
import (
    "github.com/air-iot/sdk-go/v4/driver"
)
```

3. 定义schema

`数据接入驱动` 需要定义一个 `schema` 用于描述驱动的配置信息. `schema` 是一个类似于 `json` 格式的对象, 详细格式说明见 数据接入驱动schema说明. 以下是一个简单的示例:

```
({
  "driver": {
```

```
    "properties": {
      "settings": {
        "title": "实例配置",
        "type": "object",
        "properties": {
          "server": {
            "type": "string",
            "title": "服务器",
            "descripption": "MQTT 服务器地址. 例如: tcp://127.0.0.1:1883"
          },
          "username": {
            "type": "string",
            "title": "用户名",
          },
          "password": {
            "type": "string",
            "title": "密码",
            "fieldType": "password"
          },
          "clientId": {
            "type": "string",
            "title": "客户端ID"
          },
          "topic": {
            "type": "string",
            "title": "主题",
            "descripption": "接收数据的主题. 例如: /data/#"
          },
          "parseScript": {
            "type": "string",
            "title": "数据处理脚本",
            "fieldType": "deviceScriptEdit",
            "description": "消息处理脚本. 函数名必须为 'handler'",
            "defaultScript": "/**\n" +
              " * 数据处理脚本, 处理从 mqtt 接收到的数据.\n" +
              " *\n" +
              " * @param {string} topic 消息主题\n" +
              " * @param {string} message 消息内容\n" +
              " * @return 消息解析结果\n" +
              " */\n" +
              "function handler(topic, message) {\n" +
              "\t\n" +
              "\t// 脚本返回值必须为对象数组\n" +
              "\t// \tid: 资产编号\n" +
              "\t//\ttime: 时间戳(毫秒)\n" +
              "\t//  fields: 数据点数据. 该字段为 JSON 对象, key 为数据点标识, value 为数据点的值
\n" +
              "\treturn [\n" +
              "\t\t{\"table\": \"T10001\", \"id\": \"SN10001\", \"time\": new
Date().getTime(), \"fields\": {\"key1\": \"this is a string value\", \"key2\": true,
```

```
\"key3\": 123.456}}\n" +
              "\t];\n" +
              "}"
          },
          "commandScript": {
            "type": "string",
            "title": "指令处理脚本",
            "fieldType": "deviceScriptEdit",
            "description": "指令处理脚本. 函数名必须为 'handler'",
            "defaultScript": "/**\n" +
              " * 指令处理脚本. 发送指令时会将指令内容传递给脚本，然后由指定返回最终要发送的信
息.\n" +
              " *\n" +
              " * @param {string} 工作表标识\n" +
              " * @param {string} 资产编号\n" +
              " * @param {object} 命令内容\n" +
              " * @return {object} 最终要发送的消息，及目标 topic\n" +
              " */\n" +
              "function handler(tableId, deviceId, command) {\n" +
              "\t\n" +
              "\t// 脚本返回值必须为下面对象结构\n" +
              "\t//\t\ttopic: 消息发送的目标 topic\n" +
              "\t//\t\tpayload: 消息内容\n" +
              "\treturn {\n" +
              "\t\t\"topic\": \"cmd/\" + deviceId,\n" +
              "\t\t\"payload\": \"发送内容\"\n" +
              "\t};\n" +
              "}"
          },
          "network": {
            "type": "object",
            "title": "通讯监控参数",
            "properties": {
              "timeout": {
                "title": "通讯超时时间(s)",
                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故障",
                "type": "number"
              }
            }
          }
        },
        "required": ["server", "username", "password", "topic", "parseScript",
"commandScript"]
      }
    }
  },
  "model": {
    "properties": {
      "settings": {
        "title": "模型配置",
```

```
      "type": "object",
      "properties": {
        "network": {
          "type": "object",
          "title": "通讯监控参数",
          "properties": {
            "timeout": {
              "title": "通讯超时时间(s)",
              "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故障",
              "type": "number"
            }
          }
        }
      }
    },
    "tags": {
      "title": "数据点",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": {
            "type": "string",
            "title": "标识",
            "description": "数据点的标识，用于在数据点列表中唯一标识数据点"
          },
          "name": {
            "type": "string",
            "title": "名称",
            "description": "数据点的名称"
          }
        },
        "required": ["id", "name"]
      }
    },
    "commands": {
      "title": "命令",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string",
            "title": "名称"
          },
          "ops": {
            "type": "array",
            "title": "指令",
            "items": {
              "type": "object",
```

```
              "properties": {
                "topic": {
                  "type": "string",
                  "title": "主题",
                  "description": "发送消息的主题. 例如: /cmd/control",
                },
                "message": {
                  "type": "string",
                  "title": "消息",
                  "description": "发送的消息. 例如: {\"cmd\":\"start\"}",
                },
                "qos": {
                  "type": "number",
                  "title": "QoS",
                  "description": "消息质量. 0,1,2",
                  "enum": [0, 1, 2],
                  "enum_title": ["QoS0", "QoS1", "QoS2"]
                },
              },
              "required": ["name", "message"]
            }
          }
        }
      }
    }
  },
  "device": {
    "properties": {
      "settings": {
        "title": "设备配置",
        "type": "object",
        "properties": {
          "customDeviceId": {
            "type": "string",
            "title": "设备编号",
            "descripption": "自定义设备编号. 如果未定义则使用平台中的资产编号"
          },
          "network": {
            "type": "object",
            "title": "通讯监控参数",
            "properties": {
              "timeout": {
                "title": "通讯超时时间(s)",
                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故障",
                "type": "number"
              }
            }
          }
        },
```

```
        "required": []
      }
    }
  }
})
```

4. 根据schema返回对应配置

在上一步骤中, 通过 `schema` 定义了驱动的相关配置, 包括 `驱动配置`、`数据点配置` 和 `指令配置`.

整体格式说明

驱动从平台接收到的配置信息整体格式如下:

```
{
  "id": "go-driver-mqtt-demo",
  "name": "go-driver-mqtt-demo",
  "driverType": "go-driver-mqtt-demo",
  "device": {
    "commands": [],
    "settings": {
      "clientId": "Go-demo",
      "commandScript": "function handler(tableId, deviceId, command) {\n    return
{\"topic\": \"cmd/\" + tableId + \"/\" + deviceId, \"payload\":
JSON.stringify(command.params)}\n  }",
      "parseScript": "function handler(topic, message) {\n    let arr =
JSON.parse(message.toString())\n    let topics = topic.split(\"/\");\n    let field = {}\n
arr.forEach(ele => {\n      field[ele.key] = ele.value\n    })\n    // 脚本返回值必须为对象数组
\n    // \tid: 资产编号\n    //\ttime: 时间戳(毫秒)\n    //  fields: 数据点数据. 该字段为 JSON
对象, key 为数据点标识, value 为数据点的值\n    return [\n      {\"table\": topics[1], \"id\":
topics[2], \"time\": new Date().getTime(), \"fields\": field}\n    ]\n}",
      "password": "public",
      "server": "mqtt://localhost:1883",
      "topic": "test/#",
      "username": "admin"
    }
  },
  "tables": [{
    "id": "Godriverdemo",
    "device": {
      "driver": "Go-demo",
      "groupId": "Godemo",
      "settings": {
        "clientId": "Go-deemo",
        "commandScript": "/**\n * 指令处理脚本. 发送指令时会将指令内容传递给脚本，然后由指定返回最
终要发送的信息.\n *\n * @param {string} 工作表标识\n * @param {string} 资产编号\n * @param
{object} 命令内容\n * @return {object} 最终要发送的消息，及目标 topic\n */\nfunction
```

```
handler(tableId, deviceId, command) {\n\t// 脚本返回值必须为下面对象结构\n\t//\t\ttopic: 消息发送的目标 topic\n\t//\t\tpayload: 消息内容\n\treturn {\n\t\t\"topic\": \"cmd/\" + deviceId,\n\t\t\"payload\": \"发送内容\"\n\t};\n}",
        "parseScript": "/**\n * 数据处理脚本，处理从 mqtt 接收到的数据.\n *\n * @param {string} topic 消息主题\n * @param {string} message 消息内容\n * @return 消息解析结果\n */\nfunction handler(topic, message) {\n\t// 脚本返回值必须为对象数组\n\t// \tid: 资产编号\n\t//\ttime: 时间戳(毫秒)\n\t//  fields: 数据点数据. 该字段为 JSON 对象，key 为数据点标识，value 为数据点的值\n\treturn [\n\t\t{\"id\": \"SN10001\", \"time\": new Date().getTime(), \"fields\": {\"key1\": \"this is a string value\", \"key2\": true, \"key3\": 123.456}}\n\t];\n}",
        "password": "public",
        "server": "mqtt://localhost:1883",
        "topic": "test/#",
        "username": "admin"
      },
      "tags": [{
        "id": "p1",
        "name": "数据点1"
      }],
      "commands": [{
        "defaultValue": {
          "c1": 2
        },
        "form": [{
          "arrayValue": null,
          "defaultValue": {
            "default": 2
          },
          "ifRepeat": null,
          "ioway": "默认写入",
          "mod": null,
          "name": "c1",
          "objectValue": null,
          "objectValue2": null,
          "select": null,
          "select2": null,
          "tableValue": null,
          "tableValue2": null,
          "tag": null,
          "tagValue": null,
          "type": "number"
        }],
        "name": "c1",
        "ops": [{
          "param": "c1"
        }],
        "showName": "指令1",
        "tag": null,
        "writeOut": {
          "arrayValue": null,
          "defaultValue": null,
```

```
          "ifRepeat": null,
          "mod": null,
          "objectValue": null,
          "objectValue2": null,
          "select": null,
          "select2": null,
          "tableValue": null,
          "tableValue2": null,
          "tag": null,
          "tagValue": null
        }
      }],
      "events": null
    },
    "devices": [{
      "id": "Gosdk1",
      "name": "Gosdk1",
      "device": {
        "driver": "",
        "groupId": "",
        "settings": null,
        "tags": null,
        "commands": null,
        "events": null
      },
      "disable": false,
      "off": false
    }]
  }]
}
```


上述格式中的 `device`、`tables.device` 和 `tables.devices.device` 分别为 `驱动实例配置`、`模型配置(工作表的设备配置)` 和 `资产配置(设备的设备配置)`.

其中 `settings` 为 `驱动配置信息`, 与 `schema` 中的 `settings` 对应. `tags` 为 `数据点配置信息`, 与 `schema` 中的 `tags` 对应. `commands` 为 `指令配置信息`, 与 `schema` 中的 `commands` 对应.

> **INFO**
>
> `驱动实例`、`模型` 和 `资产` 中的 `settings` 可以不相同, 但 `tags` 和 `commands` 必须相同. 例如, 可以把统一的配置信息放在 `驱动实例` 中, 把不同的配置信息放在 `模型` 或 `资产` 中.

5. 实现驱动接口

`SDK` 中定义了 `数据接入驱动接口`, 该接口是平台控制驱动的桥梁. 开发者需要实现这个接口.

```
type Driver interface {
  // Schema
  // @description 查询返回驱动配置schema内容
  // @return schema "驱动配置schema"
  Schema(app App) (schema string, err error)

  // Start
  // @description 驱动启动
  // @param driverConfig "包含实例、模型及设备数据"
  Start(app App, driverConfig []byte) (err error)

  // Run
  // @description 运行指令,向设备写入数据
  // @param command 指令参数{"table":"表标识","id":"设备编号","serialNo":"流水号","command":{}}
command 指令内容
  // @return result "自定义返回的格式或者空"
  Run(app App, command *entity.Command) (result interface{}, err error)

  // BatchRun
  // @description 批量运行指令,向多设备写入数据
  // @param command 指令参数 {"table":"表标识", "ids": ["设备编号"], "serialNo": "流水号",
'command': {}}  command 指令内容
  // @return result "自定义返回的格式或者空"
  BatchRun(app App, command *entity.BatchCommand) (result interface{}, err error)

  // WriteTag
  // @description 数据点写入
  // @param command {"table":"表标识","id":"设备编号","serialNo":"流水号","command":{}}
command 指令内容
  // @return result "自定义返回的格式或者空"
  WriteTag(app App, command *entity.Command) (result interface{}, err error)

  // Debug
  // @description 调试驱动
  // @param debugConfig object 调试参数
  // @return result "调试结果,自定义返回的格式"
  Debug(app App, debugConfig []byte) (result interface{}, err error)

  // HttpProxy
  // @description 代理接口
  // @param t 请求接口标识
  // @param header 请求头
  // @param data 请求数据
  // @return result "响应结果,自定义返回的格式"
  HttpProxy(app App, t string, header http.Header, data []byte) (result interface{}, err
error)

  // Stop
  // @description 驱动停止处理
```

```
  Stop(app App) (err error)
}
```

> **INFO**
>
> **注: 向平台上报采集到的数据时, 必须通过 `驱动与平台交互接口` 中的 `writePoint` 方法发送, 不能直接调用 `MQTT` 客户端发送. 因为 `SDK` 会对发送的数据进行一些处理, 包括有效范围处理、数值映射、缩放比例、小数位等处理.**

6. 配置驱动

这里所说 `驱动配置` 主要是一些静态配置信息（**与 schema 中定义的配置无关**）, 其中包括 `平台配置信息`, `驱动配置信息` 和 `自定义配置信息`. 这些信息一般通过配置文件(rc)、环境变量、命令行参数等方式传入. 其中一些配置信息由平台启动驱动时通过命令行参数传入.

这些配置信息, 在开发过程中可以根据实际情况进行调整. 但是在打包时必须按照平台的要求进行配置. 打包时的配置信息见 驱动配置说明.

平台配置信息

`平台配置信息` 主要为平台的连接信息. 包括: `MQTT` 连接信息, `驱动管理服务` 连接信息. 内容如下:

```yaml
mq:
  type: mqtt
  mqtt:
    host: 平台mqtt服务器ip地址
    port: 平台mqtt服务器端口

serviceId: 所属项目ID
project: 驱动实例ID

driver:
  id: 驱动ID
  name: 驱动名称

driverGrpc:
  host: 驱动管理服务ip地址
  port: 驱动管理服务端口
  healthRequestTime: 10
  waitTime: 5
```

> **INFO**

相关服务的端口号可在运维管理系统中查看.

驱动配置信息

`驱动配置信息` 主要包括 `驱动ID`, `驱动名称`, `驱动实例ID`, `所属项目ID`.

- `驱动ID` 为驱动的唯一标识, 必须在平台中唯一.
- `驱动名称` 为该驱动在平台中的显示名称.
- `驱动实例ID` 为该驱动实例的唯一标识. 同一个驱动可以创建多个实例, 每个实例的 `驱动ID` 相同但 `驱动实例ID` 唯一. 该信息由平台在 `驱动管理` 中创建驱动实例时生成.
- `所属项目ID` 每个驱动实例都属于一个项目, 该驱动实例只会拿到该项目中的模型和设备信息.

```
serviceId: 所属项目ID
project: 驱动实例ID

driver:
  id: 驱动ID
  name: 驱动名称
```

INFO

1. `驱动ID` 和 `驱动名称` 需要在配置中手动定义.
2. `驱动实例ID` 和 `所属项目ID` 在开发过程中, 需要将这些信息手动配置. 在打包时无须定义, 在平台中安装驱动时这些信息会由平台通过命令行参数传入.

驱动配置说明

以下是完整的驱动配置文件, 请参考该配置文件进行配置.

```
mq:
  type: mqtt
  mqtt:
    host: 127.0.0.1
    port: 1883

serviceId: 所属项目ID
project: 驱动实例ID
```

```yaml
driver:
  id: 驱动ID
  name: 驱动名称

driverGrpc:
  host: 127.0.0.1
  port: 9224
  healthRequestTime: 10
  waitTime: 5
```

windows系统打包发布时的驱动配置

```yaml
driverGrpc:
  host: 127.0.0.1
  port: 9224

mq:
  type: mqtt
  mqtt:
    host: 127.0.0.1
    port: 1883

driver:
  id: go-driver-mqtt-demo
  name: Go驱动例子
```

linux系统打包发布时的驱动配置

```yaml
driver:
  id: go-driver-mqtt-demo
  name: Go驱动例子
```

流程插件开发

本文将会详细介绍如何使用 `Go SDK` 开发流程插件.

介绍

`流程插件` 是扩展 `流程引擎` 中的节点的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程插件来实现自定义的功能.

> **INFO**
>
> `流程插件` 只是扩展流程功能的方式之一. 除了 `流程插件` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

该过程同 数据接入驱动开发-创建项目 中的创建项目过程一致.

2. 引入SDK

```
import (
    "github.com/air-iot/sdk-go/v4/flow"
)
```

3. 实现流程插件接口

`SDK` 中定义了 `流程插件接口`, 该接口是平台与插件交互的桥梁. 开发者需要实现这个接口.

```
type Flow interface {
    // Handler
    // @description 执行流程插件
    // @param request 执行参数 {"projectId":"项目id","flowId":"流程id","job":"流程实例id","elementId":"节点id","elementJob":"节点的实例id","config":{}} config 节点配置
    // @return result "自定义返回的格式或者空"
```

```
    Handler(app App, request *Request) (result map[string]interface{}, err error)
}
```

流程插件启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程执行到该插件对应的节点时, 会发送请求给该插件对应的程序. `SDK` 接收到请求后会调用对应的插件实现, 并将插件处理结果返回给流程引擎.

4. 配置插件

插件配置主要是插件与平台的连接配置.

```
flow:
  name: 插件名称
  mode: service  // service 插件执行方式

flowEngine:
  host: 流程引擎服务地址
  port: 流程引擎服务端口
```

windows系统打包发布时的插件配置

```
flow:
  name: testGoFlow
  mode: service

flowEngine:
  host: 127.0.0.1
  port: 2333
```

linux系统打包发布时的插件配置

```
flow:
  name: testGoFlow
  mode: service
```

流程扩展节点接入

本文将会详细介绍如何使用 `Go SDK` 开发流程扩展节点接入服务.

介绍

`流程扩展节点接入` 是扩展 `流程引擎` 中的 `扩展节点` 的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程扩展节点接入服务来实现自定义的功能.

> **INFO**
>
> `流程扩展节点接入` 只是扩展流程功能的方式之一. 除了 `流程插件` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

该过程同 数据接入驱动开发-创建项目 中的创建项目过程一致.

2. 引入SDK

```
import (
    flowextionsion "github.com/air-iot/sdk-go/v4/flow_extension"
)
```

3. 实现流程扩展节点接入接口

`SDK` 中定义了 `流程扩展节点接入`, 该接口是平台扩展节点与该服务交互的桥梁. 开发者需要实现这个接口.

```
type Extension interface {
    // Schema
    // @description 查询schema
    // @return schema "驱动配置schema"
    Schema(app App) (schema string, err error)
```

```
    // Run
    // @description 执行算法服务
    // @param input 执行参数 {} input 执行参数,应与输出的schema格式相同
    // @return result "自定义返回的格式,应与输出的schema格式相同"
    Run(app App, input []byte) (result map[string]interface{}, err error)
}
```

流程扩展节点服务启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程执行到该扩展节点时, 会发送请求给该服务程序. `SDK` 接收到请求后会调用对应的服务实现, 并将服务处理结果返回给流程引擎扩展节点.

4. 配置流程扩展节点接入服务

扩展节点接入服务配置主要是服务与平台的连接配置.

```
extension:
  id: 扩展服务唯一标识
  name: 扩展服务显示名称

flowEngine:
  host: 流程引擎服务地址
  port: 流程引擎服务端口
```

windows系统打包发布时的插件配置

```
extension:
  id: testGoFlowExt
  name: 测试go扩展

log:
  level: 5

flowEngine:
  host: 127.0.0.1
  port: 2333
```

linux系统打包发布时的插件配置

```
extension:
  id: testGoFlowExt
  name: 测试go扩展
```

算法服务开发

本文将会详细介绍如何使用 `Go SDK` 开发算法服务开发.

介绍

`算法服务` 是扩展 `算法` 中的一种方式. 在平台现有的功能不满足需求时, 可以通过开发算法服务来实现自定义的功能.

开发步骤

1. 创建项目

该过程同 数据接入驱动开发-创建项目 中的创建项目过程一致.

2. 引入SDK

```
import (
    "github.com/air-iot/sdk-go/v4/algorithm"
)
```

3. 定义schema

`算法服务` 需要定义一个 `schema` 用于描述算法的配置信息. `schema` 是一个类似于 `json` 格式的对象, 详细格式说明见 算法schema说明.

4. 实现算法接口

`SDK` 中定义了 `算法服务接口`, 该接口是平台与插件交互的桥梁. 开发者需要实现这个接口.

```
type Service interface {
    // Schema
    // @description 查询schema
    // @return result "算法配置schema,返回字符串"
    Schema(App) (result string, err error)

    // Start
    // @description 启动算法服务
```

```
    Start(App) error

    // Run
    // @description 执行算法服务
    // @param bts 执行参数 {"function":"算法名","input":{}} input 算法执行参数,应与输出的schema
格式相同
    // @return result "自定义返回的格式,应与输出的schema格式相同"
    Run(app App, bts []byte) (result interface{}, err error)

    // Stop
    // @description 停止算法服务
    Stop(App) error
}
```

算法服务启动时, `SDK` 会连接平台的 `算法服务` 服务, 并接收算法服务发送的请求. 当执行该服务算法时, 会发送请求给该服务程序. `SDK` 接收到请求后会调用对应的方法, 并将方法的处理结果返回给算法管理.

5. 配置算法

算法配置主要是算法与平台的连接配置.

```
algorithmGrpc:
  host: 算法管理服务地址
  port: 算法管理服务端口

algorithm:
  id: 算法唯一标识
  name: 算法显示名称
```

windows系统打包发布时的算法配置

```
algorithmGrpc:
  host: 127.0.0.1
  port: 9236

algorithm:
  id: testGoAlgorithm
  name: 测试算法
```

linux系统打包发布时的算法配置

```
algorithm:
  id: testGoAlgorithm
```

```
name: 测试算法
```

平台接口客户端

介绍

平台客户端 SDK 用于访问平台接口，提供了平台接口的 Go 实现, 可以很方便实现第三方系统与平台的集成。平台客户端 SDK 中包括常用的 `租户管理`、`项目管理`、 `用户管理`、`角色管理`、`工作表及数据管理`、`系统变量(数据字典)`、`报警管理` 等接口, 并且不断在丰富和完善中。

使用方式

1. 在项目中安装SDK

```
go get github.com/air-iot/api-client-go/v4
```

2. 在需要调用平台接口创建客户端

在引入客户端 SDK 后. 示例代码如下:

```
clientEtcd, err := clientv3.New(clientv3.Config{
    Endpoints:   []string{"127.0.0.1:2379"},
    DialTimeout: time.Second * time.Duration(60),
    DialOptions: []grpc.DialOption{grpc.WithBlock()},
    Username:    "用户名",
    Password:    "密码",
})
if err != nil {
    log.Fatal(err)
}

cli, clean, err := api_client_go.NewClient(clientEtcd, config.Config{
    EtcdConfig: "/airiot/config/dev.json",
    Metadata:   map[string]string{"env": "aliyun"},
    Services: map[string]config.Service{
    },
})
```

> **INFO**
>
> 关于如何创建ak、sk

查询构造器

客户端接口中的很多查询接口, 其结构比较复杂.

查询参数的整体结构如下所示:

```
{
  "project": {},
  "filter": {},
  "sort": {},
  "limit": 30,
  "skip": 20,
  "withCount": true
}
```

字段说明如下:

- **`project`** 查询请求需要返回的字段列表. 例如: `{"id": 1, "name": 1, "address": {"city": 1}}`. `key` 为字段名, `value` 为 `1` 或 `对象`. 如果为为一级字段需要设置为 `1` 例如: `{"id": 1, "name": 1}`, 如果要返回嵌套对象内的字段, 则需要设置为 `Map`, 例如: `{"address": {"city": 1}}`
- **`filter`** 查询条件, 如果没有添加任何条件则查询全部数据. `key` 为字段名, `value` 为过滤的值或逻辑运算符, 例如: `{"name": "Tom", "age": {"$gt": 20, "$lt": 30}}`.
- **`sort`** 排序条件, `key` 为字段名, `value` 为 `1` 表示升序, `-1` 表示降序, 例如: `{"age": 1, "name": -1}`.
- **`limit`** 查询结果的最大数量, 可用于分页查询或限制返回的记录数量.
- **`skip`** 查询结果的偏移量, 即忽略前 N 记录, 可用于分页查询.
- **`withCount`** 是否返回符合条件的记录总数, 如果为 `true` 则会在查询结果记录数量会保存在响应对象 `ResponseDTO<T>` 中的 `count` 字段.

> **INFO**
>
> **注意事项**
>
> 1. 如果查询条件需要使用逻辑或, 可以在 `filter` 中添加 `$or` 字段, 其值为 `{k:v}` 结构与 `filter` 一致, 任一条件成立时表示记录匹配.
> 2. 如果同一字段存在多个逻辑条件, 则需要将多个条件放在一个 `对象` 中, 例如: `{"age": {"$gt": 20, "$lt": 30}}`, 表示查询 `20 < age < 30` 的记录. :::
>
> **逻辑运算符**

符号	说明	示例
$not	不相等, 与 `SQL` 中的 `<>` 作用相同	{"age": {"$not": 18}}
$in	在指定列表内, 与 `SQL` 中的 `in` 作用相同	{"id": {"$in": [1,3,4]}}
$nin	不在指定列表内, 与 `SQL` 中的 `not in` 作用相同	{"id": {"$nin": [1,3,4]}}
$gt	大于指定的值, 与 `SQL` 中的 `>` 作用相同	{"age": {"$gt": 18}}
$gte	大于等于指定的值, 与 `SQL` 中的 `>=` 作用相同	{"age": {"$gte": 18}}
$lt	小于指定的值, 与 `SQL` 中的 `<` 作用相同	{"age": {"$lt": 18}}
$lte	小于等于指定的值, 与 `SQL` 中的 `<=` 作用相同	{"age": {"$lte": 18}}
$regex	正则匹配, 与 `SQL` 中的 `like` 相似	{"name": {"$regex": "张"}}

打包部署

本文将详细介绍平台打包及部署.

打包

驱动打包就是将开发完成的程序打包为可以在平台部署的驱动. 平台自身支持运行在 `windows`、`linux` 和 `macOS` 系统中, 并且支持 `x86` 和 `arm` 平台. 在 `windows` 系统中平台服务和驱动程序都是直接运行在操作系统中, 而在 `linux` 系统中是以 `容器` 的方式运行, 平台中的每个服务和驱动程序都是一个独立的容器, 所以针对不同的操作系统打包方式也不相同. 下面分别介绍在 `windows` 和 `linux` 系统中如何打包驱动， 对于不同平台只需要保证使用软件和库支持即可.

windows系统打包

在 `windows` 系统中, 驱动程序是直接运行在操作系统中, 所以需要将驱动程序打包. 具体打包步骤如下:

1. 以 `Go` 驱动为例打程序程序和相关资源打包为 `二进制` 文件.项目执行下面的命令进行编译:

```
go build -tags netgo -v -o go-driver-mqtt-demo.exe main.go
```

2. 准备驱动配置文件 `config.yaml`. 可以将 `config.yaml` 放在 `main.exe` 文件相同的 `etc` 目录下, 这样驱动在启动时会自动加载该配置文件.

> **INFO**
>
> 注: `config.yaml` 中需要填写好 `驱动ID` 和 `驱动名称` 两个配置项, 参考驱动文档.

3. 准备驱动安装配置文件 `service.yml` . 在平台中安装驱动时, 需要提供一些驱动的基本信息, 例如: 版本号、驱动描述、端口号等. 这些信息需要在 `service.yml` 中定义, 平台会根据该文件中的配置信息进行安装. `service.yml` 的具体格式如下:

```
Name: go-driver-mqtt-demo
Description: 测试程序
Version: 4.0.0
ConfigType: config.yaml
GroupName: driver
Command: go-driver-mqtt-demo.exe
```

4. 将所有资源打包为 `zip` 文件.

将 `go-driver-mqtt-demo.exe`、`config.yaml`、`service.yml` 和其它资源打包为 `zip` 文件, 平台会根据该文件进行安装. 建议打包后的 `zip` 文件结构如下:

linux系统打包

由于在 `linux` 系统中, 驱动程序是以 `容器` 的方式运行, 所以打包时需要先将驱动程序打包为 `docker` 镜像. 然后再将镜像文件和 `service.yml` 打包为 `.tar.gz` 压缩包. 具体打包步骤如下:

1. 以 `Go` 驱动为例打程序程序和相关资源打包为 `二进制` 文件.项目执行下面的命令进行编译:

```
SET CGO_ENABLED=0
SET GOOS=linux
SET GOARCH=amd64

go build -tags netgo -v -o go-driver-mqtt-demo main.go
```

2. 准备驱动配置文件 `config.yaml`. 可以将 `config.yaml` 放在 `go-driver-mqtt-demo` 文件相同的 `etc` 目录下, 这样驱动在启动时会自动加载该配置文件.

> **INFO**
>
> 注: `config.yaml` 中需要填写好 `驱动ID` 和 `驱动名称` 两个配置项, 参考驱动文档.

3. 准备 `Dockerfile` 文件. 以下是一个简单的 `Dockerfile` 文件示例, 具体内容根据自身的需求进行修改:

```
FROM alpine:latest

ADD go-driver-mqtt-demo /app/
ADD config.yaml /app/etc/config.yaml
WORKDIR /app
ENTRYPOINT ["./go-driver-mqtt-demo"]
```

4. 构建 `docker` 镜像.

使用上一步中的 `Dockerfile` 文件构建 `docker` 镜像, 具体命令如下:

```
docker build -t airiot/go-driver-example:v4.0.0 .
```

6 导出 `docker` 镜像并压缩.

```
docker save airiot/go-driver-example:v4.0.0 | gzip > go-driver-example.tar.gz
```

6. 准备驱动安装配置文件 `service.yml` . 该文件的格式与 windows系统打包 中的第三步中的 `service.yml` 文件格式相似但又有区别. 具体格式如下:

```
# 必填项. 驱动名称
Name: go-driver-mqtt-demo
# 必填项. 例如: 1.0.0, 通常用镜像版本号一致
Version: 4.0.0
# 非必填项.
Description: 描述信息
# 必填项. 驱动固定为 driver、流程插件、算法服务为 server
GroupName: driver
# 容器端口映射类型, 非必填项. 如果驱动需要对外提供 rest 服务, 或暴露端口时, 需要填写该配置项.
# 可选项有 None Internal External
#
# None: 不暴露端口
# Internal: 只在平台内部暴露端口. 一般为驱动对外提供 rest 服务时, 将端口映射到网关上, 填写为
Internal 即可.
# External: 对外暴露端口. 一般为驱动为作 server 端, 需要对外暴露端口以供设备连接, 此时该端口会暴露
在宿主机上, 填写为 External 即可.
Service: None

# 非必填项. 暴露的端口列表
#Ports:
#  - Host: "8558"            # 映射到宿主机的端口号, 如果不填写, 则会随机分配一个端口号
#    Container: "8558"       # 容器内部的端口号, 即驱动服务监听的端口号
#    Protocol: ""            # 协议类型, 可选项有 TCP UDP, 如果不填写, 则默认为 TCP
```

7. 将所有资源打包为 `gzip` 文件. 将 `docker镜像` 和 `service.yml` 文件打包为 `gzip` 文件. 打包命令如下:

```
tar cvf go-driver-example.tar go-driver-example.tar.gz service.yml
```

```
gzip go-driver-example.tar
```

打包后的 `gzip` 文件结构如下:

INFO

`linux` 系统整个打包过程对应的命令如下所示:

```
# 将驱动打包为镜像
docker build -t airiot/go-driver-example:v4.0.0 .

# 导出镜像并压缩
docker save airiot/go-driver-example:v4.0.0 | gzip > go-driver-example.tar.gz

# 将镜像文件和 service.yml 打包
tar cvf go-driver-example.tar go-driver-example.tar.gz service.yml

# 对整个驱动包进行压缩
gzip go-driver-example.tar

# 最后得到 go-driver-example.tar.gz 压缩文件
```

部署

将上一步骤中得到的驱动安装包通过 `运维管理系统` 上传到平台, 平台会自动解析并安装驱动. 安装成功后, 就可以在项目中使用该驱动了.

安装驱动

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `离线上传驱动` 按钮, 选择上一步中得到的 `go-driver-example.tar.gz` 文件, 点击 `确定` 按钮, 平台会自动解析并安装驱动.

如果驱动安装失败, 可以在 运维管理系统 的 首页 中查看详细的日志信息.

> **INFO**
>
> 不同版本的平台, 离线上传驱动 按扭的位置可能不同.

使用驱动

当驱动成功安装到平台后, 就可以在项目中使用该驱动了.

具体使用方法请参考 驱动管理.

> **INFO**
>
> 注: 需要将运维服务的 InternetAccess 改为false, 才能读取本地仓库

Dotnet SDK 介绍

Dotnet SDK 二次开发说明

数据接入驱动开发

介绍如何使用 Dotnet SDK 开发自定义数据接入驱动

流程插件开发

介绍如何使用 Dotnet SDK 开发自定义流程插件

流程扩展节点接入

介绍如何使用 Dotnet SDK 开发自定义流程扩展节点接入服务

算法服务开发

介绍如何使用 Dotnet SDK 开发算法服务

平台接口客户端

Dotnet SDK 平台接口客户端

打包部署

本文将详细介绍平台打包及部署.

Dotnet SDK 介绍

`Dotnet SDK` 是 AIRIOT 物联网平台提供的 `Dotnet` 语言的二次开发工具包, 可使用 Dotnet SDK 调用平台开放的接口和实现对平台功能的扩展。

接下来会分别介绍如何使用 `Dotnet SDK` 数据接入驱动、流程插件、流程扩展节点接入、算法集成 和通过平台接口客户端实现系统集成。

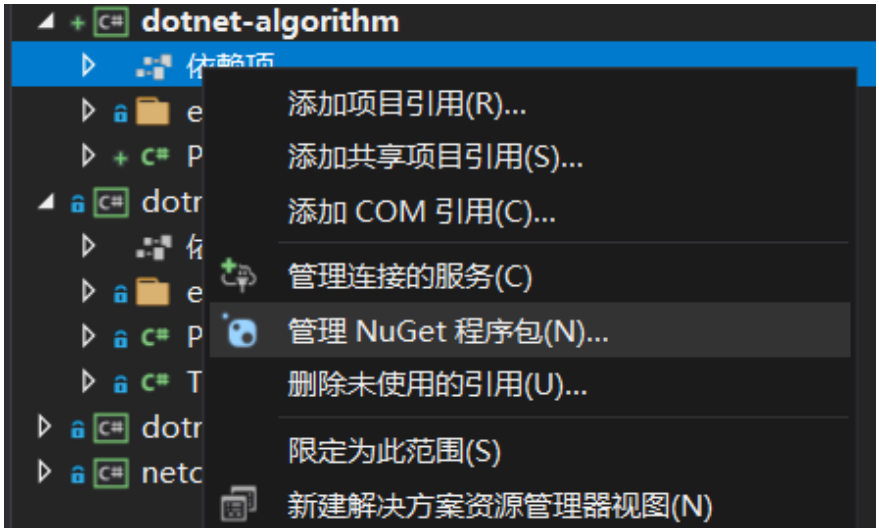
内容说明

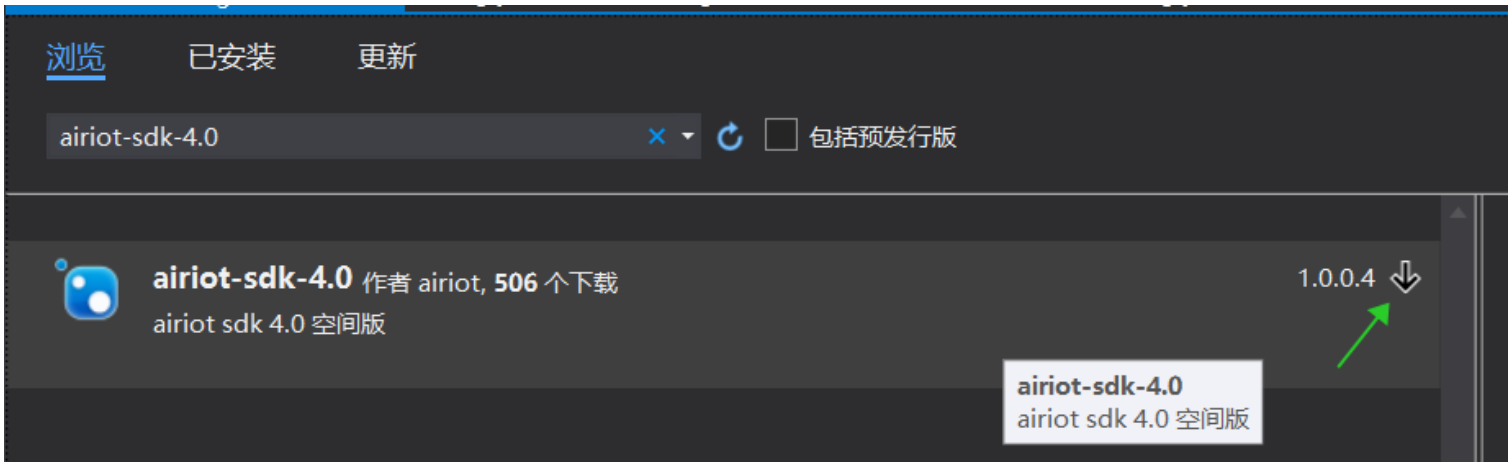
内容	用途
数据接入驱动	实现从设备或其它平台系统采集数据
流程插件	扩展流程引擎中的节点
流程扩展节点接入	开发流程扩展节点接入服务
算法集成	扩展自定义算法
平台接口客户端	调用平台开放的API接口

使用方式

`Dotnet SDK` 包已经上传NuGet，使用NuGet管理工具搜索airiot-sdk-4.0最新稳定版下载即可：

下图为Microsoft Visual Studio工具添加Dotnet sdk示例：





版本说明

.net 环境 .net SDK基于.net Standard 2.0开发，请安装.net Framework 4.6.1或者 .net Core 2.0以上版本开发环境。

示例仓库

示例代码: https://github.com/air-iot/sdk-dotnet-example

数据接入驱动开发

本文将会详细介绍如何使用 `Dotnet SDK` 开发自定义数据接入驱动.

介绍

`数据接入驱动` 是为实现从不同的协议、设备或其它平台系统采集数据而开发的特定程序. 每个 `数据接入驱动` 程序需要根据协议的特点实现数据采集功能，然后将采集到的数据通过 `Dotnet SDK` 提供的接口发送到平台.

`Dotnet SDK` 提供了 `数据接入驱动` 开发的相关内容. 包括驱动的接口定义, 以及与平台交互功能. 开发者只需要实现接口中的方法.

开发步骤

1. 创建项目

1. 创建项目C#控制台程序
2. 创建 `Programs.cs` 文件

2. 引入SDK

1. 添加nuget包，详细步骤见Dotnet sdk介绍-使用方式
2. 引用命名空间

```
    using AiriotSDK.Data;
    using AiriotSDK.Driver;
    using AiriotSDK.Tools;
```

3. 定义schema

`数据接入驱动` 需要定义一个 `schema` 用于描述驱动的配置信息. `schema` 是一个类似于 `json` 格式的对象, 详细格式说明见 数据接入驱动schema说明. 以下是一个简单的示例:

```
({
    "title": "netcore-driver",
    "key": "netcore-driver",
    "model": {
```

```
        "properties": {
            "settings": {
                "title": "设备配置",
                "type": "object",
                "properties": {
                    "interval": {
                        "type": "number",
                        "title": "采集周期"
                    },
                    "port": {
                        "type": "number",
                        "title": "端口"
                    }
                }
            },
            "tags": {
                "title": "数据点",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "title": "名称"
                        },
                        "id": {
                            "type": "string",
                            "title": "标识"
                        },
                        "unit": {
                            "type": "string",
                            "title": "单位"
                        },
                        "fixed": {
                            "type": "number",
                            "title": "小数位数"
                        },
                        "mod": {
                            "type": "number",
                            "title": "缩放比例"
                        }
                    }
                }
            },
            "commands": {
                "title": "命令",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
```

```
                "name": {
                    "type": "string",
                    "title": "名称"
                },
                "form": {
                    "type": "array",
                    "title": "表单项",
                    "items": {
                        "type": "object",
                        "properties": {
                            "name": {
                                "type": "string",
                                "title": "参数名"
                            },
                            "type": {
                                "type": "string",
                                "title": "数据类型",
                                "enum": [
                                    "string",
                                    "number",
                                    "boolean"
                                ],
                                "enum_title": [
                                    "字符串",
                                    "数字",
                                    "布尔型"
                                ]
                            },
                            "format": {
                                "type": "string",
                                "title": "表单类型",
                                "enum": [
                                    "",
                                    "date",
                                    "datetime",
                                    "email",
                                    "upload_file_input"
                                ],
                                "enum_title": [
                                    "默认",
                                    "日期选择器",
                                    "时间选择器",
                                    "电子邮件输入框",
                                    "文件上传"
                                ]
                            },
                            "enum": {
                                "type": "array",
                                "title": "选择项值",
                                "items": {
```

```
                        "type": "string"
                    }
                },
                "enum_title": {
                    "type": "array",
                    "title": "选择项文字",
                    "items": {
                        "type": "string"
                    }
                },
                "required": {
                    "type": "boolean",
                    "title": "是否必填"
                },
                "default": {
                    "type": "string",
                    "title": "默认值"
                },
                "mod": {
                    "type": "number",
                    "title": "缩放比例"
                },
                "tagValue": {
                    "type": "object",
                    "title": "数值定义",
                    "properties": {
                        "minValue": {
                            "title": "最小值",
                            "type": "number"
                        },
                        "maxValue": {
                            "title": "最大值",
                            "type": "number"
                        },
                        "minRaw": {
                            "title": "原始最小值",
                            "type": "number"
                        },
                        "maxRaw": {
                            "title": "原始最大值",
                            "type": "number"
                        }
                    }
                }
            }
        },
        "ops": {
            "title": "指令",
            "type": "array",
```

```
                        "items": {
                            "type": "object",
                            "properties": {
                                "action": {
                                    "type": "string",
                                    "title": "布/撤防",
                                    "enum": [
                                        "arming",
                                        "unArming",
                                        "queryState"
                                    ],
                                    "enum_title": [
                                        "系统全布防",
                                        "系统全撤防",
                                        "查询模块状态"
                                    ]
                                },
                                "modelNo": {
                                    "type": "number",
                                    "title": "模块编号",
                                    "description": "查询模块状态指令填写"
                                }
                            }
                        }
                    }
                }
            }
        }
    },
    "device": {
        "properties": {
            "settings": {
                "title": "设备配置",
                "type": "object",
                "properties": {
                    "interval": {
                        "type": "number",
                        "title": "采集周期"
                    }
                }
            },
            "tags": {
                "title": "数据点",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
```

```
                        "title": "名称"
                    },
                    "id": {
                        "type": "string",
                        "title": "标识"
                    },
                    "unit": {
                        "type": "string",
                        "title": "单位"
                    },
                    "fixed": {
                        "type": "number",
                        "title": "小数位数"
                    },
                    "mod": {
                        "type": "number",
                        "title": "缩放比例"
                    }
                }
            }
        }
    }
})
```

4. 根据schema返回对应配置

在上一步骤中, 通过 `schema` 定义了驱动的相关配置, 包括 `驱动配置`、`数据点配置` 和 `指令配置`.

整体格式说明

驱动从平台接收到的配置信息整体格式如下:

```
{
    "id": "645b4249b4f5349780d205d5",
    "name": "netcore-driver",
    "driverType": "netcore-driver",
    "device": {
        "commands": [],
        "settings": {}
    },
    "tables": [{
        "id": "演示表001",
        "device": {
            "driver": "netcore-driver",
            "groupId": "645b4249b4f5349780d205d5",
```

```
            "settings": {
                "interval": 10
            },
            "tags": [{
                "id": "csdata001",
                "name": "csdata001"
            }, {
                "id": "csdata002",
                "name": "csdata002"
            }],
            "commands": [],
            "events": null
        },
        "devices": [{
            "id": "cstable001",
            "name": "cstable001",
            "device": {
                "driver": "",
                "groupId": "",
                "settings": null,
                "tags": null,
                "commands": null,
                "events": null
            },
            "disable": false,
            "off": false
        }]
    }]
}
```

上述格式中的 `device`、`tables.device` 和 `tables.devices.device` 分别为 `驱动实例配置`、`模型配置(工作表的设备配置)` 和 `资产配置(设备的设备配置)`.

其中 `settings` 为 `驱动配置信息`, 与 `schema` 中的 `settings` 对应. `tags` 为 `数据点配置信息`, 与 `schema` 中的 `tags` 对应. `commands` 为 `指令配置信息`, 与 `schema` 中的 `commands` 对应.

> **INFO**
>
> `驱动实例`、`模型` 和 `资产` 中的 `settings` 可以不相同, 但 `tags` 和 `commands` 必须相同. 例如, 可以把统一的配置信息放在 `驱动实例` 中, 把不同的配置信息放在 `模型` 或 `资产` 中.

5. 实现驱动接口

`SDK` 中定义了 `数据接入驱动接口`, 该接口是平台控制驱动的桥梁. 开发者需要实现这个接口.

```
/// <summary>
/// 驱动实力接口
/// </summary>
public interface IDriver
{
    /// <summary>
    /// 启动方法
    /// </summary>
    ErrorInfo Start(IDriverApp app, byte[] bytes);

    /// <summary>
    /// 驱动重载方法
    /// </summary>
    ErrorInfo Reload(IDriverApp app, byte[] bytes);

    /// <summary>
    /// 执行指令方法
    /// </summary>
    CommResult Run(IDriverApp app, Command cmd);

    /// <summary>
    /// 批量执行指令方法
    /// </summary>
    CommResult BatchRun(IDriverApp app, BatchCommand cmd);

    /// <summary>
    /// 数据点写入
    /// </summary>
    CommResult WriteTag(IDriverApp app, Command cmd);

    /// <summary>
    /// 调试驱动方法
    /// </summary>
    CommResult Debug(IDriverApp app, byte[] bytes);

    /// <summary>
    /// 停止驱动方法
    /// </summary>
    ErrorInfo Stop(IDriverApp app);

    /// <summary>
    /// 获取驱动schema
    /// </summary>
    CommResult Schema(IDriverApp app);
}

/// <summary>
/// 返回结果
/// </summary>
```

```
public class CommResult
{
    public CommResult(bool status, string message, object data = null)
    {
        Data = data;
        if (status)
        {
            Error = null;
        }
        else
        {
            Error = new ErrorInfo()
            {
                Code = -1,
                Message = message
            };
        }
    }
    public ErrorInfo Error;
    public object Data;

    /// <summary>
    /// 失败方法
    /// </summary>
    /// <param name="msg">错误提示信息</param>
    /// <returns></returns>
    public static CommResult Failure(string msg)
    {
        return new CommResult(false, msg);
    }

    /// <summary>
    /// 成功方法
    /// </summary>
    /// <param name="msg">成功提示信息</param>
    /// <param name="data">成功时返回的数据</param>
    /// <returns></returns>
    public static CommResult Success(string msg, object data = null)
    {
        return new CommResult(true, msg, data);
    }
}

/// <summary>
/// 错误消息
/// </summary>
public class ErrorInfo
{
    /// <summary>
    /// 错误码
```

```
        /// </summary>
        public int Code { get; set; }

        /// <summary>
        /// 错误提示信息
        /// </summary>
        public string Message { get; set; }

        /// <summary>
        /// 生成一条新的错误消息
        /// </summary>
        /// <param name="msg">错误提示信息</param>
        /// <returns></returns>
        public static ErrorInfo New(string msg)
        {
            return new ErrorInfo()
            {
                Code = -1,
                Message = msg
            };
        }
    }
```

> **INFO**
>
> **注: 向平台上报采集到的数据时, 必须通过 `驱动与平台交互接口` 中的 `writePoint` 方法发送, 不能直接调用 `MQTT` 客户端发送. 因为 `SDK` 会对发送的数据进行一些处理, 包括有效范围处理、数值映射、缩放比例、小数位等处理.**

6. 配置驱动

这里所说 `驱动配置` 主要是一些静态配置信息（**与 schema 中定义的配置无关**）, 其中包括 `平台配置信息`, `驱动配置信息` 和 `自定义配置信息`. 这些信息一般通过配置文件(rc)、环境变量、命令行参数等方式传入. 其中一些配置信息由平台启动驱动时通过命令行参数传入.

这些配置信息, 在开发过程中可以根据实际情况进行调整. 但是在打包时必须按照平台的要求进行配置. 打包时的配置信息见 驱动配置说明.

平台配置信息

`平台配置信息` 主要为平台的连接信息. 包括: `MQTT` 连接信息, `驱动管理服务` 连接信息. 内容如下:

```
mq:
  type: mqtt
```

```
  mqtt:
    host: 平台mqtt服务器ip地址
    port: 平台mqtt服务器端口

serviceId: 所属项目ID
project: 驱动实例ID

driver:
  id: 驱动ID
  name: 驱动名称

driverGrpc:
  host: 驱动管理服务ip地址
  port: 驱动管理服务端口
```

> **INFO**
>
> 相关服务的端口号可在运维管理系统中查看.

驱动配置信息

`驱动配置信息` 主要包括 `驱动ID`, `驱动名称`, `驱动实例ID`, `所属项目ID`.

- `驱动ID` 为驱动的唯一标识, 必须在平台中唯一.
- `驱动名称` 为该驱动在平台中的显示名称.
- `驱动实例ID` 为该驱动实例的唯一标识. 同一个驱动可以创建多个实例, 每个实例的 `驱动ID` 相同但 `驱动实例ID` 唯一. 该信息由平台在 `驱动管理` 中创建驱动实例时生成.
- `所属项目ID` 每个驱动实例都属于一个项目, 该驱动实例只会拿到该项目中的模型和设备信息.

```
serviceId: 所属项目ID
project: 驱动实例ID

driver:
  id: 驱动ID
  name: 驱动名称
```

> **INFO**
>
> 1. `驱动ID` 和 `驱动名称` 需要在配置中手动定义.
> 2. `驱动实例ID` 和 `所属项目ID` 在开发过程中, 需要将这些信息手动配置. 在打包时无须定义, 在平台中安装驱动时这些信息会由平台通过命令行参数传入.

驱动配置说明

以下是完整的驱动配置文件, 请参考该配置文件进行配置.

```yaml
mq:
  type: mqtt
  mqtt:
    host: 127.0.0.1
    port: 1883

serviceId: 所属项目ID
project: 驱动实例ID

driver:
  id: 驱动ID
  name: 驱动名称

driverGrpc:
  host: 127.0.0.1
  port: 9224
```

windows系统打包发布时的驱动配置

```yaml
driverGrpc:
  host: 127.0.0.1
  port: 9224

mq:
  type: mqtt
  mqtt:
    host: 127.0.0.1
    port: 1883

driver:
  id: netcore-driver
  name: dotnet驱动例子
```

linux系统打包发布时的驱动配置

```
```

```yaml
driver:
  id: netcore-driver
  name: dotnet驱动例子
```

流程插件开发

本文将会详细介绍如何使用 `Dotnet SDK` 开发流程插件.

介绍

`流程插件` 是扩展 `流程引擎` 中的节点的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程插件来实现自定义的功能.

> **INFO**
>
> `流程插件` 只是扩展流程功能的方式之一. 除了 `流程插件` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

1. 创建项目C#控制台程序
2. 创建 `Programs.cs` 文件

2. 引入SDK

1. 添加nuget包，详细步骤见Dotnet sdk介绍-使用方式
2. 引用命名空间

```
using AiriotSDK.Data;
using AiriotSDK.Flow;
using AiriotSDK.Tools;
```

3. 实现流程插件接口

`SDK` 中定义了 `流程插件接口`, 该接口是平台与插件交互的桥梁. 开发者需要实现这个接口.

```
```

```
    public class Request {
        public string ProjectId { get; set; }
        public string FlowId { get; set; }
        public string Job { get; set; }
        public string ElementId { get; set; }
        public string ElementJob { get; set; }
        public byte[] Config { get; set; }
    }

    /// <summary>
    /// Flow interface
    /// </summary>
    public interface IFlow
    {
        /// <summary>
        /// 流程节点处理方法
        /// </summary>
        /// <param name="app"></param>
        /// <param name="req"></param>
        CommResult Handler(IFlowApp app, Request req);
    }
```

流程插件启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程执行到该插件对应的节点时, 会发送请求给该插件对应的程序. `SDK` 接收到请求后会调用对应的插件实现, 并将插件处理结果返回给流程引擎.

4. 配置插件

插件配置主要是插件与平台的连接配置.

```
flow:
  name: 插件名称
  mode: service  // service 插件执行方式

flowEngine:
  host: 流程引擎服务地址
  port: 流程引擎服务端口
```

windows系统打包发布时的插件配置

```
flow:
  name: dotnetFlow
  mode: service
```

```
flowEngine:
  host: 127.0.0.1
  port: 2333
```

linux系统打包发布时的插件配置

```
flow:
  name: dotnetFlow
  mode: service
```

流程扩展节点接入

本文将会详细介绍如何使用 `Dotnet SDK` 开发流程扩展节点接入服务.

介绍

`流程扩展节点接入` 是扩展 `流程引擎` 中的 `扩展节点` 的一种方式. 在流程引擎现有的功能不满足需求时, 可以通过开发流程扩展节点接入服务来实现自定义的功能.

> **INFO**
>
> `流程扩展节点接入` 只是扩展流程功能的方式之一. 除了 `流程插件` 扩展方式之外, 还可以开发一个独立的服务或与现有的服务集成. 例如: 在 `数据接口` 中添加被调用的目标服务, 然后在流程中使用 `数据接口` 节点调用该接口, 也可以实现对流程的扩展. 详细信息参考 HTTP数据接口 和 数据库接口

开发步骤

1. 创建项目

1. 创建项目C#控制台程序
2. 创建 `Programs.cs` 文件

2. 引入SDK

1. 添加nuget包，详细步骤见Dotnet sdk介绍-使用方式
2. 引用命名空间

```
using AiriotSDK.Data;
using AiriotSDK.FlowExtension;
using AiriotSDK.Tools;
```

3. 实现流程扩展节点接入接口

`SDK` 中定义了 `流程扩展节点接入`, 该接口是平台扩展节点与该服务交互的桥梁. 开发者需要实现这个接口.

```
```

```
    /// <summary>
    /// Flow interface
    /// </summary>
    public interface IFlowExtension
    {
        /// <summary>
        /// Schema
        /// </summary>
        /// <param name="app"></param>
        /// <param name="req"></param>
        CommResult Schema(IFlowExtensionApp app);

        /// <summary>
        /// Run
        /// </summary>
        /// <param name="app"></param>
        /// <param name="input"></param>
        /// <returns></returns>
        CommResult Run(IFlowExtensionApp app, byte[] input);
    }
```

流程扩展节点服务启动时, `SDK` 会连接平台的 `流程引擎` 服务, 并接收流程引擎发送的请求. 当流程执行到该扩展节点时, 会发送请求给该服务程序. `SDK` 接收到请求后会调用对应的服务实现, 并将服务处理结果返回给流程引擎扩展节点.

4. 配置流程扩展节点接入服务

扩展节点接入服务配置主要是服务与平台的连接配置.

```
extension:
  id: 扩展服务唯一标识
  name: 扩展服务显示名称

flowEngine:
  host: 流程引擎服务地址
  port: 流程引擎服务端口
```

windows系统打包发布时的插件配置

```
extension:
  id: dotnetFlowExtension
  name: dotnet扩展节点服务

log:
```

```
  level: debug

flowEngine:
  host: 127.0.0.1
  port: 2333
```

linux系统打包发布时的插件配置

```
extension:
  id: dotnetFlowExtension
  name: dotnet扩展节点服务
```

算法服务开发

本文将会详细介绍如何使用 `Dotnet SDK` 开发算法服务开发.

介绍

`算法服务` 是扩展 `算法` 中的一种方式. 在平台现有的功能不满足需求时, 可以通过开发算法服务来实现自定义的功能.

开发步骤

1. 创建项目

1. 创建项目C#控制台程序
2. 创建 `Programs.cs` 文件

2. 引入SDK

1. 添加nuget包，详细步骤见Dotnet sdk介绍-使用方式
2. 引用命名空间

```
    using AiriotSDK.Data;
    using AiriotSDK.Algorithm;
    using AiriotSDK.Tools;
```

3. 定义schema

`算法服务` 需要定义一个 `schema` 用于描述算法的配置信息. `schema` 是一个类似于 `json` 格式的对象, 详细格式说明见 算法schema说明.

4. 实现算法接口

`SDK` 中定义了 `算法服务接口`, 该接口是平台与插件交互的桥梁. 开发者需要实现这个接口.

```
/// <summary>
/// 算法服务接口
/// </summary>
```

```
public interface IService
{
    /// <summary>
    /// 执行方法
    /// </summary>
    /// <param name="app"></param>
    /// <param name="bytes"></param>
    /// <returns></returns>
    CommResult Run(IAlgorithmApp app, byte[] bytes);

    /// <summary>
    /// 获取Schema
    /// </summary>
    /// <param name="app"></param>
    /// <returns></returns>
    CommResult Schema(IAlgorithmApp app);
}
```

算法服务启动时, `SDK` 会连接平台的 `算法服务` 服务, 并接收算法服务发送的请求. 当执行该服务算法时, 会发送请求给该服务程序. `SDK` 接收到请求后会调用对应的方法, 并将方法的处理结果返回给算法管理.

5. 配置算法

算法配置主要是算法与平台的连接配置.

```
algorithmGrpc:
  host: 算法管理服务地址
  port: 算法管理服务端口

algorithm:
  id: 算法唯一标识
  name: 算法显示名称
```

windows系统打包发布时的算法配置

```
algorithmGrpc:
  host: 127.0.0.1
  port: 9236

algorithm:
  id: testDotnetAlgorithm
  name: 测试算法
```

linux系统打包发布时的算法配置

```yaml
algorithm:
  id: testDotnetAlgorithm
  name: 测试算法
```

平台接口客户端

介绍

平台客户端 SDK 用于访问平台接口，提供了平台接口的 Dotnet 实现, 可以很方便实现第三方系统与平台的集成。平台客户端 SDK 中包括常用的 `租户管理`、`项目管理`、 `用户管理`、`角色管理`、`工作表及数据管理`、`系统变量(数据字典)`、`报警管理` 等接口, 并且不断在丰富和完善中。

使用方式

1.创建项目

1. 创建项目C#控制台程序
2. 创建 `Programs.cs` 文件

2. 引入SDK

1. 添加nuget包，详细步骤见Dotnet sdk介绍-使用方式
2. 引用命名空间

```
    using AiriotSDK.Data;
    using AiriotSDK.Api;
    using AiriotSDK.Tools;
```

3. 在需要调用平台接口创建客户端

在引入客户端 SDK 后. 示例代码如下:

```
static void Main(stirng[] args)
{
    Config cfg = new()
    {
        ProjectId = "default",
        Schema = "http",
        Host = "127.0.0.1",
        Port = 31000,
        Credentials = new()
        {
```

```
            AK = "AK",
            SK = "SK"
        }
    };

    Client cli = new(cfg);
    Query qr = new()
    {
        Filter = new Dictionary<string, object>
        {
            { "title", "CoAP" },
        },
        Project = new Dictionary<string, object>
        {
            { "id", 1},
            { "title", 1 }
        }
    };

    string query = JsonConvert.SerializeObject(qr);

    string tables = cli.FindTableQuery(query);

    Console.WriteLine(tables);
}
```

> **INFO**
>
> 关于如何创建ak、sk

查询构造器

客户端接口中的很多查询接口, 其结构比较复杂.

查询参数的整体结构如下所示:

```
{
  "project": {},
  "filter": {},
  "sort": {},
  "limit": 30,
  "skip": 20,
  "withCount": true
}
```

字段说明如下:

- `project` 查询请求需要返回的字段列表. 例如: `{"id": 1, "name": 1, "address": {"city": 1}}`. `key` 为字段名, `value` 为 `1` 或 `对象`. 如果为为一级字段需要设置为 `1` 例如: `{"id": 1, "name": 1}`, 如果要返回嵌套对象内的字段, 则需要设置为 `Map`, 例如: `{"address": {"city": 1}}`
- `filter` 查询条件, 如果没有添加任何条件则查询全部数据. `key` 为字段名, `value` 为过滤的值或逻辑运算符, 例如: `{"name": "Tom", "age": {"$gt": 20, "$lt": 30}}`.
- `sort` 排序条件, `key` 为字段名, `value` 为 `1` 表示升序, `-1` 表示降序, 例如: `{"age": 1, "name": -1}`.
- `limit` 查询结果的最大数量, 可用于分页查询或限制返回的记录数量.
- `skip` 查询结果的偏移量, 即忽略前 N 记录, 可用于分页查询.
- `withCount` 是否返回符合条件的记录总数, 如果为 `true` 则会在查询结果记录数量会保存在响应对象 `ResponseDTO<T>` 中的 `count` 字段.

> **INFO**
>
> **注意事项**
>
> 1. 如果查询条件需要使用逻辑或, 可以在 `filter` 中添加 `$or` 字段, 其值为 `{k:v}` 结构与 `filter` 一致, 任一条件成立时表示记录匹配.
> 2. 如果同一字段存在多个逻辑条件, 则需要将多个条件放在一个 `对象` 中, 例如: `{"age": {"$gt": 20, "$lt": 30}}`, 表示查询 `20 < age < 30` 的记录. :::
>
> **逻辑运算符**

符号	说明	示例
$not	不相等, 与 `SQL` 中的 `<>` 作用相同	{"age": {"$not": 18}}
$in	在指定列表内, 与 `SQL` 中的 `in` 作用相同	{"id": {"$in": [1,3,4]}}
$nin	不在指定列表内, 与 `SQL` 中的 `not in` 作用相同	{"id": {"$nin": [1,3,4]}}
$gt	大于指定的值, 与 `SQL` 中的 `>` 作用相同	{"age": {"$gt": 18}}
$gte	大于等于指定的值, 与 `SQL` 中的 `>=` 作用相同	{"age": {"$gte": 18}}
$lt	小于指定的值, 与 `SQL` 中的 `<` 作用相同	{"age": {"$lt": 18}}
$lte	小于等于指定的值, 与 `SQL` 中的 `<=` 作用相同	{"age": {"$lte": 18}}

符号	说明	示例
$regex	正则匹配, 与 `SQL` 中的 `like` 相似	{"name": {"$regex": "张"}}

打包部署

本文将详细介绍平台打包及部署.

打包

驱动打包就是将开发完成的程序打包为可以在平台部署的驱动. 平台自身支持运行在 `windows`、`linux` 和 `macOS` 系统中, 并且支持 `x86` 和 `arm` 平台. 在 `windows` 系统中平台服务和驱动程序都是直接运行在操作系统中, 而在 `linux` 系统中是以 `容器` 的方式运行, 平台中的每个服务和驱动程序都是一个独立的容器, 所以针对不同的操作系统打包方式也不相同. 下面分别介绍在 `windows` 和 `linux` 系统中如何打包驱动， 对于不同平台只需要保证使用软件和库支持即可.

windows系统打包

在 `windows` 系统中, 驱动程序是直接运行在操作系统中, 所以需要将驱动程序打包. 具体打包步骤如下:

1. 以 `Dotnet` 驱动为例打程序程序和相关资源打包为 `二进制` 文件.项目执行下面的命令进行编译:

```
dotnet build
```

2. 准备驱动配置文件 `config.yaml`. 可以将 `config.yaml` 放在 `netcore-driver.exe` 文件相同的 `etc` 目录下, 这样驱动在启动时会自动加载该配置文件.

> **INFO**
>
> 注: `config.yaml` 中需要填写好 `驱动ID` 和 `驱动名称` 两个配置项, 参考驱动文档.

3. 准备驱动安装配置文件 `service.yml` . 在平台中安装驱动时, 需要提供一些驱动的基本信息, 例如: 版本号、驱动描述、端口号等. 这些信息需要在 `service.yml` 中定义, 平台会根据该文件中的配置信息进行安装. `service.yml` 的具体格式如下:

```
Name: dotnetTestDriver
Description: dotnet测试驱动
Version: 4.0.0
ConfigType: config.yaml
GroupName: driver
Command: netcore-driver.exe
```

4. 将所有资源打包为 `zip` 文件.

将 `netcore-driver.exe`、`config.yaml`、`service.yml` 和其它资源打包为 `zip` 文件, 平台会根据该文件进行安装. 建议打包后的 `zip` 文件结构如下:

名称	修改日期	类型	大小
etc	2023/9/13 9:55	文件夹	
ref	2023/9/13 16:21	文件夹	
runtimes	2023/9/13 9:55	文件夹	
airiot-sdk-4.0.dll	2023/9/13 15:49	应用程序扩展	254 KB
Dockerfile	2023/9/13 10:02	文件	1 KB
Google.Protobuf.dll	2022/9/14 1:06	应用程序扩展	400 KB
Grpc.Core.Api.dll	2022/5/24 8:35	应用程序扩展	58 KB
Grpc.Core.dll	2022/5/24 8:32	应用程序扩展	471 KB
Microsoft.Extensions.Logging.Abstractions.dll	2018/6/7 0:26	应用程序扩展	47 KB
Microsoft.Win32.SystemEvents.dll	2020/10/20 2:40	应用程序扩展	27 KB
MQTTnet.dll	2020/10/24 18:09	应用程序扩展	279 KB
netcore-driver.deps.json	2023/9/13 16:21	JSON File	64 KB
netcore-driver.dll	2023/9/13 16:21	应用程序扩展	11 KB
netcore-driver.exe	2023/9/13 16:21	应用程序	125 KB
netcore-driver.pdb	2023/9/13 16:21	Program Debug Data...	12 KB
netcore-driver.runtimeconfig.dev.json	2023/9/13 16:21	JSON File	1 KB
netcore-driver.runtimeconfig.json	2023/9/13 16:21	JSON File	1 KB
Newtonsoft.Json.dll	2019/11/9 8:56	应用程序扩展	678 KB
Quartz.dll	2021/1/20 1:47	应用程序扩展	914 KB

linux系统打包

由于在 `linux` 系统中, 驱动程序是以 `容器` 的方式运行, 所以打包时需要先将驱动程序打包为 `docker` 镜像. 然后再将镜像文件和 `service.yml` 打包为 `.tar.gz` 压缩包. 具体打包步骤如下:

1. 准备驱动配置文件 `config.yaml`. 可以将 `config.yaml` 放在 `netcore-driver.csproj` 文件相同的 `etc` 目录下, 这样在生成镜像时会根据如下Dockerfile拷贝到对应目录.

   > **INFO**
   >
   > 注: `config.yaml` 中需要填写好 `驱动ID` 和 `驱动名称` 两个配置项, 参考驱动文档.

2. 编写Dockerfile,示例如下:

```
```

```
FROM mcr.microsoft.com/dotnet/runtime:5.0 AS base
WORKDIR /app

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["netcore-driver.csproj", "."]
RUN dotnet restore "./netcore-driver.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "netcore-driver.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "netcore-driver.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
COPY etc /app/etc
ENTRYPOINT ["dotnet", "netcore-driver.dll"]
```

3. 构建 `docker` 镜像.

使用上一步中的 `Dockerfile` 文件构建 `docker` 镜像, 具体命令如下:

```
docker build -t airiot/netcore-driver:v4.0.0 .
```

6 导出 `docker` 镜像并压缩.

```
docker save airiot/netcore-driver:v4.0.0 | gzip > netcore-driver.tar.gz
```

6. 准备驱动安装配置文件 `service.yml` . 该文件的格式与 windows系统打包 中的第三步中的 `service.yml` 文件格式相似但又有区别. 具体格式如下:

```
# 必填项. 驱动名称
Name: netcore-driver
# 必填项. 例如: 1.0.0, 通常用镜像版本号一致
Version: 4.0.0
# 非必填项.
Description: 描述信息
# 必填项. 驱动固定为 driver、流程插件、算法服务为 server
GroupName: driver
# 容器端口映射类型, 非必填项. 如果驱动需要对外提供 rest 服务, 或暴露端口时, 需要填写该配置项.
# 可选项有 None Internal External
```

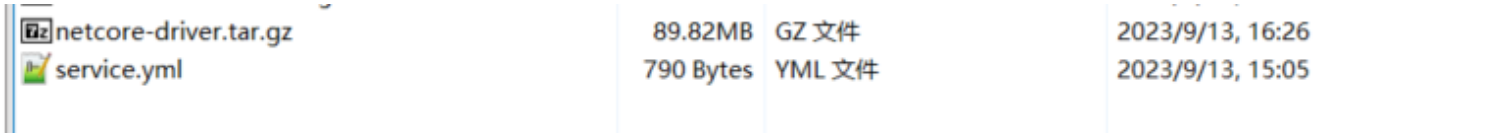
```
#
# None: 不暴露端口
# Internal: 只在平台内部暴露端口. 一般为驱动对外提供 rest 服务时, 将端口映射到网关上, 填写为
Internal 即可.
# External: 对外暴露端口. 一般为驱动为作 server 端, 需要对外暴露端口以供设备连接, 此时该端口会暴露
在宿主机上, 填写为 External 即可.
Service: None

# 非必填项. 暴露的端口列表
#Ports:
#  - Host: "8558"            # 映射到宿主机的端口号, 如果不填写, 则会随机分配一个端口号
#    Container: "8558"       # 容器内部的端口号, 即驱动服务监听的端口号
#    Protocol: ""            # 协议类型, 可选项有 TCP UDP, 如果不填写, 则默认为 TCP
```

7. 将所有资源打包为 `gzip` 文件. 将 `docker镜像` 和 `service.yml` 文件打包为 `gzip` 文件. 打包命令如下:

```
tar cvf netcore-driver-v400.tar netcore-driver.tar.gz service.yml
```

```
gzip netcore-driver-v400.tar
```

打包后的 `gzip` 文件结构如下:



部署

将上一步骤中得到的驱动安装包通过 `运维管理系统` 上传到平台, 平台会自动解析并安装驱动. 安装成功后, 就可以在项目中使用该驱动了.

安装驱动

1. 登录 `运维管理系统`, 运维管理系统的默认登录地址为 `http://IP:13030/`, 将 `IP` 换成平台地址即可.
2. 点击左侧菜单栏中的 `服务管理` 选项, 进入服务管理页面.
3. 点击页面右上角的 `离线上传驱动` 按钮, 选择上一步中得到的 `go-driver-example.tar.gz` 文件, 点击 `确定` 按钮, 平台会自动解析并安装驱动.

如果驱动安装失败, 可以在 运维管理系统 的 首页 中查看详细的日志信息.

> **INFO**
>
> 不同版本的平台, 离线上传驱动 按扭的位置可能不同.

使用驱动

当驱动成功安装到平台后, 就可以在项目中使用该驱动了.

具体使用方法请参考 驱动管理.

> **INFO**
>
> 注: 需要将运维服务的 InternetAccess 改为false, 才能读取本地仓库

驱动schema说明

本文将详细介绍 `数据接入驱动` 的 schema 的格式, 以及如何根据自己的需求定义 `schema`.

介绍

每个 `数据接入驱动` 进程为一个驱动实例, 每个驱动实例可以配置多个 `模型(工作表)`, 每个模型中可以添加多个 `设备`. `驱动实例`、`模型(工作表)` 和 `设备` 都可以有独立的配置信息, 可以相同也可以不同.

`数据接入驱动` 的 `schema` 就是用于定义 `数据接入驱动` 的配置信息, 其中包括 `驱动实例配置`, `模型配置(工作表)` 和 `设备配置`, 分别对应 `驱动实例`、`模型(工作表)` 和 `设备` 的配置信息. 其中每一项中又包含 `驱动配置`、`数据点配置` 和 `命令配置` 3 个部分.

> **INFO**
>
> 通常情况下, `驱动实例配置`, `模型配置(工作表)` 和 `设备配置` 的配置是相同的, 但有些场景中会有所不同. 但一般只有 `驱动配置` 内容不同, 但 `数据点配置` 和 `命令配置` 通常情况是相同的.
>
> 一般在实现 `数据接入驱动` 时, 需要将 `驱动实例配置`, `模型配置(工作表)` 和 `设备配置` 进行合并, 合并后的配置才是设备的最终配置, 配置的优先级从高到低为 `设备 > 模型(工作表) > 驱动实例`.

整体格式介绍

`schema` 内容是 `Json Schema`, 在 `设备配置` 页面中, 会根据 `schema` 的内容动态生成配置表单. 最外层定义了 `driver`、`model` 和 `device` 3 个配置项, 分别对应 `驱动实例配置`, `模型配置(工作表)` 和 `设备配置`. 如下所示:

```
{
    "driver": {...},     // 驱动实例配置
    "model": {...},      // 模型配置(工作表)
    "device": {...}      // 设备配置
}
```

每个配置项中都包含 `settings`, `tags` 和 `commands` 3 个部分, 分别对应 `驱动配置`, `数据点配置` 和 `命令配置`. 以 `driver` 为例, 展开后整体格式如下所示:

```
{
    "driver": {
```

```
        "properties": {
            "settings": {                   // 驱动配置
                "title": "实例配置",
                "type": "object",
                "properties": {},           // 驱动配置项
                "required": []              // 驱动配置项中的必填项列表
            },
            "tags": {                       // 数据点配置
                "title": "数据点",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {},       // 数据点配置项
                    "required": []          // 数据点配置项中的必填项列表
                }
            },
            "commands": {                   // 命令配置
                "title": "命令",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {},       // 命令配置项
                    "required": []          // 命令配置项中的必填项列表
                }
            }
        }
    }
}
```

> **INFO**
>
> 只需要关注 `properties` 和 `required` 的内容即可, 其他内容都是固定的.

详细说明

接下来会分别详细介绍 `驱动配置`, `数据点配置` 和 `命令配置` 的格式.

驱动配置说明

`驱动配置` 为驱动进行数据采集所需要的配置项. 例如想通过 `MQTT` 接收设备发送的数据, 驱动必须要知道 `MQTT` 服务器地址、端口、用户名、密码以及订阅的主题等信息, 这些信息就是 `驱动配置`.

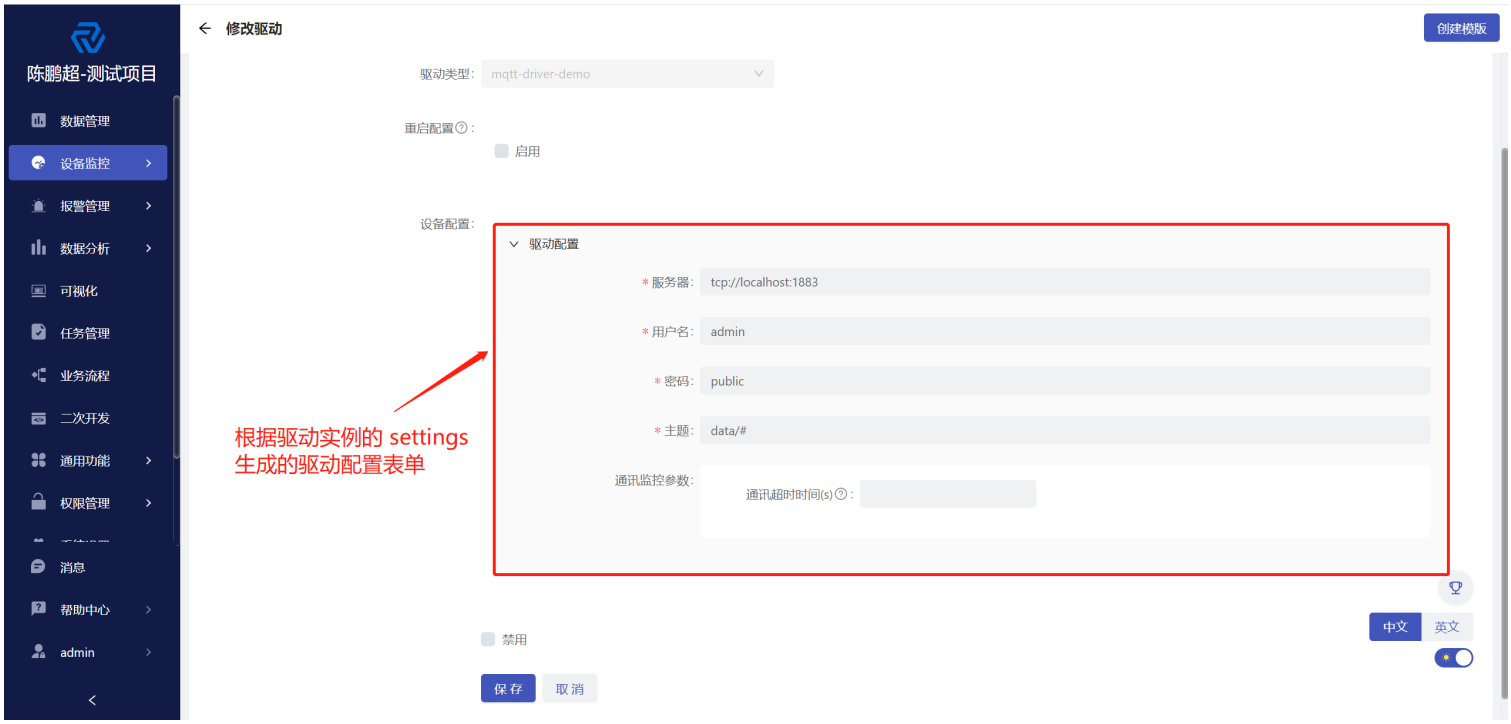
`驱动配置` 的 `properties` 为配置项列表, `required` 为必填项列表. 以 `MQTT` 为例, `驱动配置` 的 `properties` 和 `required` 如下所示:

```
{
    "driver": {
        "properties": {
            "settings": {
                "title": "实例配置",
                "type": "object",
                "properties": {
                    "server": {
                        "type": "string",
                        "title": "服务器地址",
                        "descripption": "MQTT 服务器地址. 例如: tcp://127.0.0.1:1883"
                    },
                    "username": {
                        "type": "string",
                        "title": "用户名",
                    },
                    "password": {
                        "type": "string",
                        "title": "密码",
                        "fieldType": "password"
                    },
                    "topic": {
                        "type": "string",
                        "title": "主题",
                        "descripption": "接收数据的主题. 例如: /data/#"
                    },
                    "network": {
                        "type": "object",
                        "title": "通讯监控参数",
                        "properties": {
                            "timeout": {
                                "title": "通讯超时时间(s)",
                                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故
障",
                                "type": "number"
                            }
                        }
                    }
                },
                "required": ["server", "username", "password", "topic"]
            }
        }
    }
}
```

INFO

`驱动配置` 中的 `network` 为固定字段, 用于配置设备通讯超时参数. 即当该设备下的所有数据点在指定时间内没有数据上传, 则认为该设备通讯故障. 该设备的状态会变为 `离线`.

根据上述配置, 在页面中动态生成的表单如下图所示:



数据点配置说明

`数据点` 为从设备上采集到的数据列表, 例如设备上有 `温度`、`湿度`、`电压` 等, 这些就是 `数据点`. 而 `数据点配置` 就是用来定义和描述数据点信息以及平台在接收到数据后如何处理这些数据.

> **INFO**
>
> **需要注意的是, `数据点配置` 并不是用来定义一个设备有哪些 `数据点`, 而是用来定义每个数据点的特征, 让驱动程序知道如何去采集这个数据点的数据.**
>
> 例如: `OPC-UA` 协议中, 订阅数据时需要提供 `NamespaceIndex`, `Identifier` 和 `IdentifierType` 等信息, 这些信息在服务器中标识唯一的数据点. 此时就需要在 `数据点配置` 中定义这些信息, 以便驱动程序在采集数据时能够正确的订阅到数据点.

例如设备通过 `MQTT` 发送 `温度` 和 `湿度` 两个数据, 数据格式为 `{"T": 25, "H": 50}`, 此时需要在平台中定义 `温度` 和 `湿度` 两个数据点用来接收这些数据, 并且需要告诉平台 `T` 为 `温度` 数据, 单位为 `℃`; `H` 为 `湿度` 数据, 其单位为 `%RH`. 所以在 `数据点配置` 中定义一个 `key` 用于标识该数据点的数据在设备发送的 `json` 数据的 `key`.

而 `数据点配置` 的 `properties` 为配置项列表, `required` 为必填项列表. 如下所示:

```
{
    "driver": {
        "properties": {
            "settings": {},
            "tags": {
                "title": "数据点",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "id": {
                            "type": "string",
                            "title": "标识",
                            "description": "数据点的标识，用于在数据点列表中唯一标识数据点"
                        },
                        "name": {
                            "type": "string",
                            "title": "名称",
                            "description": "数据点的名称"
                        },
                        "key": {
                            "type": "string",
                            "title": "键",
                            "description": "数据点在 JSON 对象中的 key"
                        }
                    },
                    "required": ["id", "name", "key"]
                }
            }
        }
    }
}
```

根据上述配置, 在页面中动态生成的表单如下图所示:

红框内为根据 `tags` 动态生成的内容, 其它内容均为平台自带内容

然后添加两个数据点, 分别为 `温度` 和 `湿度`, 并且将 `key` 设置为 `T` 和 `H`, 如下图所示:

INFO

`id` 和 `name` 为固定配置, 分别为 `数据点标识` 和 `数据点的名称`. 而且需要将 `id` 和 `name` 设置为必填项.

> 通过 `schema` 定义的 `数据点配置` 只是整个数据点配置中的一部分, 还有一些配置是平台定义的. 例如: 标识、名称、单位以及一些其它处理功能等.

命令配置说明

`命令` 为平台向设备发送的指令, 例如: `开灯`、`关灯`、`重启` 等. 而 `命令配置` 就是用来定义和描述如何向设备发送命令以及发送内容. 例如通过平台向设备发送 `重启` 控制命令. 以 `MQTT` 为例, 发送一条命令，需要知道命令发送的 `Topic`, `Qos` 和发送内容等信息, 所以在 `命令配置` 中定义这些信息, 以便驱动程序在发送命令时能够正确的发送命令.

```
{
    "driver": {
        "properties": {
            "settings": {},
            "commands": {
                "title": "命令",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "title": "名称"
                        },
                        "ops": {
                            "type": "array",
                            "title": "指令",
                            "items": {
                                "type": "object",
                                "properties": {
                                    "name": {
                                        "type": "string",
                                        "title": "主题",
                                        "description": "发送消息的主题. 例如: /cmd/control",
                                    },
                                    "message": {
                                        "type": "string",
                                        "title": "消息",
                                        "description": "发送的消息. 例如:
{\"cmd\":\"start\"}",
                                    },
                                    "qos": {
                                        "type": "number",
                                        "title": "QoS",
                                        "description": "消息质量. 0,1,2",
                                        "enum": [0, 1, 2],
```

```
                                    "enum_title": ["QoS0", "QoS1", "QoS2"]
                                },
                            },
                            "required": ["name", "message"]
                        }
                    }
                }
            }
        }
    }
}
```

根据上述配置, 在页面中动态生成的表单如下图所示:

红框内为根据 `commands` 动态生成的内容, 其它内容均为平台自带内容

然后控制设备重启的命令, 目标 Topic 为 `/command`, Qos 为 `1`, 消息内容为 `restart`, 如下图所示:

完整配置示例

```
{
    "driver": {
        "properties": {
            "settings": {
                "title": "实例配置",
                "type": "object",
                "properties": {
                    "server": {
                        "type": "string",
                        "title": "服务器地址",
                        "descripption": "MQTT 服务器地址. 例如: tcp://127.0.0.1:1883"
                    },
                    "username": {
                        "type": "string",
                        "title": "用户名",
                    },
                    "password": {
                        "type": "string",
                        "title": "密码",
                        "fieldType": "password"
                    },
                    "topic": {
                        "type": "string",
                        "title": "主题",
                        "descripption": "接收数据的主题. 例如: /data/#"
                    },
```

```
                    "network": {
                        "type": "object",
                        "title": "通讯监控参数",
                        "properties": {
                            "timeout": {
                                "title": "通讯超时时间(s)",
                                "description": "经过多长时间仪表还没有任何数据上传，认定为通讯故
障",
                                "type": "number"
                            }
                        }
                    }
                },
                "required": ["server", "username", "password", "topic"]
            },
            "tags": {
                "title": "数据点",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "id": {
                            "type": "string",
                            "title": "标识",
                            "description": "数据点的标识，用于在数据点列表中唯一标识数据点"
                        },
                        "name": {
                            "type": "string",
                            "title": "名称",
                            "description": "数据点的名称"
                        },
                        "key": {
                            "type": "string",
                            "title": "键",
                            "description": "数据点在 JSON 对象中的 key"
                        }
                    },
                    "required": ["id", "name", "key"]
                }
            },
            "commands": {
                "title": "命令",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "title": "名称"
                        },
```

```
                        "ops": {
                            "type": "array",
                            "title": "指令",
                            "items": {
                                "type": "object",
                                "properties": {
                                    "name": {
                                        "type": "string",
                                        "title": "主题",
                                        "description": "发送消息的主题. 例如: /cmd/control",
                                    },
                                    "message": {
                                        "type": "string",
                                        "title": "消息",
                                        "description": "发送的消息. 例如:
{\"cmd\":\"start\"}",
                                    },
                                    "qos": {
                                        "type": "number",
                                        "title": "QoS",
                                        "description": "消息质量. 0,1,2",
                                        "enum": [0, 1, 2],
                                        "enum_title": ["QoS0", "QoS1", "QoS2"]
                                    },
                                },
                                "required": ["name", "message"]
                            }
                        }
                    }
                }
            }
        }
    },
    "model": {},
    "device": {},
}
```

算法schema说明

本文将详细介绍 `算法集成` 的 schema 的格式, 以及如何根据自己的需求定义 `schema`.

介绍

每个 `算法集成` 进程为一个算法服务, 每个服务可以配置多个 `算法`.

`算法集成` 的 `schema` 就是用于定义 `算法集成` 的配置信息, 其中包括 `算法名`, `输入参数` 和 `输出参数`,.

> **INFO**
>
> 算法名与算法一一对应.
>
> 输入参数是算法方法执行需要的参数列表.
>
> 输出参数是算法方法执行后的返回值格式.

格式介绍

`schema` 内容是 `Json Schema`, 在 `我的算法` 页面中, 会根据 `schema` 的内容动态生成函数列表、输入参数及输出参数. 最外层为数组可以配置多个算法, 每个元素对应一个算法, 如下所示:

```
[{
    "title": "加法", // 页面显示名称
    "function": "add", // 算法名
    "input": {}, // 输入参数
    "output": {} // 输出参数
}, {
    "title": "绝对值",
    "function": "abs",
    "input": {},
    "output": {}
}]
```

每个元素配置项中都包含 `title`, `function`, `input` 和 `output` 3 个部分, 分别对应 `页面显示名称`, `算法名`, `输入参数` 和 `输出参数`. 展开后整体格式如下所示:

```
```

```
[
  {
    "title": "加法",
    "function": "add",
    "input": {
      "type": "object",
      "properties": {
        "num1": {
          "title": "参数1",
          "type": "number"
        },
        "num2": {
          "title": "参数2",
          "type": "number"
        }
      },
      "required": [
        "num1",
        "num2"
      ]
    },
    "output": {
      "type": "object",
      "properties": {
        "num1": {
          "title": "结果",
          "type": "number"
        }
      }
    }
  },
  {
    "title": "绝对值",
    "function": "abs",
    "input": {
      "type": "object",
      "properties": {
        "num1": {
          "title": "参数1",
          "type": "number"
        }
      },
      "required": [
        "num1"
      ]
    },
    "output": {
      "type": "object",
      "properties": {
```

```
        "res": {
          "title": "结果",
          "type": "number"
        }
      }
    }
  }
]
```

流程扩展节点接入schema说明

本文将详细介绍 `流程扩展节点接入` 的 schema 的格式, 以及如何根据自己的需求定义 `schema`.

介绍

每个 `流程扩展节点接入` 进程为一个扩展服务, 每个服务可以执行节点处理.

`流程扩展节点接入` 的 `schema` 就是用于定义 `流程扩展节点接入` 的配置信息, 其中包括 `输入参数`.

> **INFO**
>
> 输入参数是服务执行需要的参数列表.

格式介绍

`schema` 内容是 `Json Schema`, 在 `扩展节点` 页面中, 会根据 `schema` 的内容动态生成输入参数. 如下所示:

```
{
    "type": "object",
    "properties": {},
    "required": []
}
```

每个元素配置项中都包含 `title`, `input` 和 `output` 3 个部分, 分别对应 `页面显示名称`, `输入参数` 和 `输出参数`. 展开后整体格式如下所示:

```
{
  "type": "object",
  "properties": {
    "num1": {
      "title": "参数1",
      "type": "number"
    },
    "num2": {
      "title": "参数2",
      "type": "number"
    }
  },
  "required": [
```

```
    "num1",
    "num2"
  ]
}
```