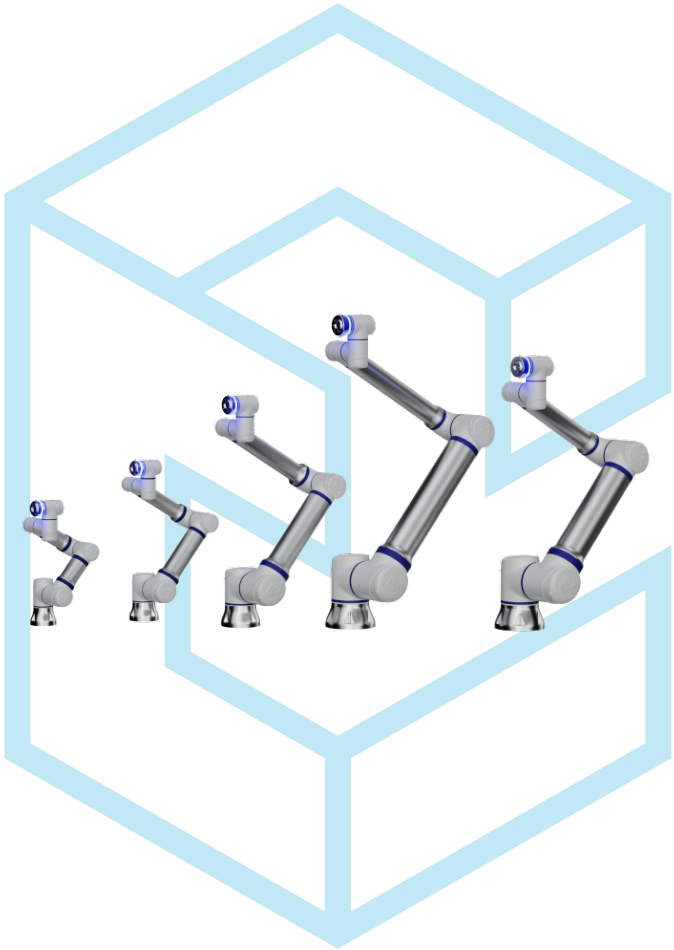
ELITE ROBOTS
艾利特机器人

ELITE ROBOTS
开发手册



Elite 插件开发手册

艾利特智能机器人股份有限公司

2026-01-19

版本：Ver2.15.0

使用前请仔细阅读本手册

此版本手册对应产品版本信息请见用户手册版本信息章节，使用前请仔细核对实际产品版本信息，确保一致。

本手册会定期进行检查和修正，更新后的内容将出现在新版本中。本手册中的内容或信息如有变更, 恕不另行通知。

艾利特智能机器人股份有限公司对本手册中可能出现的任何错误概不负责。

艾利特智能机器人股份有限公司对因使用本手册及其中所述产品而引起的意外或间接伤害概不负责。

安装、使用产品前，请阅读本手册。

请保管好本手册，以便可以随时阅读和参考。

本手册图片仅供参考，请以收到的实物为准。

摘要

ELITECO Plugin 软件开发平台支持第三方开发者通过开放的 SDK 定制 EliRobot 机器人平台功能。通过 ELITECO SDK，第三方开发者可以定制各种功能并接入到 EliRobot 机器人平台，包括但不限于任务节点、配置节点、主页选项卡以及 Daemon 程序等。例如可以定制更具体化的符合终端用户工艺场景的任务节点或模板、也可以为外部软硬件设备提供友好的配置节点等等。为了开发者面更好的使用 ELITECO SDK 进行功能定制，本教程将对此进行详细说明。

目录

1 引言 1

1.1 ELITECO SDK 功能 1

1.2 章节安排 2

1.3 开发环境相关 3

2 ELITECO SDK 5

3 ELITECO Plugin 项目前期 7

3.1 ELITECO Plugin 项目新建 7

3.2 ELITECO Plugin 项目准备 9

3.2.1 LICENSE 文件 10

3.2.2 国际化资源 10

3.2.3 图片资源 12

3.2.4 服务注册绑定 12

3.3 ELITECO Plugin 项目实现 15

3.4 ELITECO Plugin 项目构建与部署 18

3.5 ELITECO Plugin UI 组件 23

3.5.1 手风琴导航组件 23

3.5.2 开关按钮组件 25

3.5.3 高级面板组件 26

3.5.4 文件选择器组件 27

3.5.5 键盘组件 29

3.5.6 消息组件 31

3.5.7 弹出框组件 33

3.5.8 提示组件 34

3.5.9 进度条组件 35

3.5.10 按钮样式 37
3.5.11 字体样式 39

4 定制配置节点 43

4.1 项目新建 43
4.2 定制配置节点贡献 44
4.3 定制配置节点参数视图 48
4.4 定制配置节点服务 54

5 定制任务节点 59

5.1 项目新建 59
5.2 定制任务节点贡献 60
5.3 定制任务节点参数视图 69
5.4 定制任务节点服务 76
5.5 任务模块其他功能特性 79
5.5.1 任务节点创建 79
5.5.2 节点插入删除 83
5.5.3 任务树及节点访问 88
5.5.4 撤销重做 94
5.5.5 任务树 UI 句柄、节点策略及事件 95
5.5.6 变量表达式 99

6 定制导航栏贡献 103

6.1 项目新建 103
6.2 定制工具栏服务 104
6.3 定制导航栏贡献 105

7 定制 Daemon 程序 111

7.1 项目新建 111
7.2 定制 Daemon 程序 113

7.3 定制导航栏贡献 . . . 115
7.4 XmlRpc 客户端 . . . 118
7.5 Daemon 程序 . . . 119

8 其他功能特性 125

8.1 变量 . . . 125
8.1.1 配置变量 . . . 127
8.1.2 任务变量 . . . 131
8.1.3 坐标系变量 . . . 133
8.2 计量类数据 . . . 134
8.3 I/O . . . 135
8.3.1 工具 I/O 锁 . . . 141
8.4 工具 . . . 145
8.5 坐标系 . . . 148
8.6 数据持久化 . . . 149
8.7 脚本生成 . . . 157
8.8 系统 . . . 159
8.8.1 机器人运动服务 . . . 159
8.8.2 机器人仿真服务 . . . 164
8.8.3 Rpc 服务 . . . 167
8.8.4 命名服务 . . . 168
8.8.5 机器人状态 . . . 169
8.8.6 系统设置服务 . . . 170
8.8.7 系统 IP 服务 . . . 172
8.8.8 任务状态服务 . . . 172
8.8.9 软件版本 . . . 173

9 附录 175

9.1 内部任务节点创建及配置示例 . . . 175

9.1.1 Move 节点创建 175

第 1 章 引言

本手册描述的是 ELITECO SDK 1.2.15.0 的功能特性，随 EliRobot 2.15.0 版本一起发布。为后续描述清晰，这里明确若干概念：

ELITECO SDK：指随 EliRobot 主版本同步发布的工具包，内含 API jar 包、依赖 jar 包、文档、工具脚本等文件，用于 ELITECO Plugin 的开发。默认存放在 ELITECO Plugin 虚拟机开发平台的/home 目录下；

ELITECO Plugin：指基于 ELITECO SDK 开发的 EliRobot 机器人平台功能的扩展软件包，后缀名通常为.elico；

同时约定后续 ELITECO SDK 随 EliRobot 同步发布新版本，并同时附带对应版本的 CS_Elite 插件开发手册。

1.1 ELITECO SDK 功能

ELITECO Plugin 软件开发平台许第三方开发人员对 EliRobot 机器人平台多个模块功能进行定制扩展，主要支持以下功能：

定制配置节点

开发人员可以定制配置节点，用于拓展对应的前端视图，使之成为 EliRobot 机器人平台“配置”页签下的一个功能节点，点击该节点可以展示对应的前端视图。

定制的配置节点主要有以下特征：

- 其对应的前端视图可作为外部软硬件设备或其他功能的配置入口；
- 其关键数据可以随配置文件的存储/加载实现持久化；
- 可以通过内部接口在任务脚本前部生成脚本行，供内部或定制任务节点使用相关数据或功能。

定制任务节点

开发人员可以定制任务节点，用于扩展对应的参数视图，使之成为 EliRobot 机器人平台“任务”页签下“指令-插件”下的节点，点击该节点可以在当前树上插入该任务节点对应的任务节点。

定制的任务节点主要有以下特征：

- 其对应的参数视图作为任务树上该类型节点的公共视图，用于对该类型节点的参数配置；

- 任务树上每一个该类型节点的数据都可以随任务文件的存储/加载实现持久化；
- 任务树上每一个该类型节点通过内部接口在任务脚本文件中生成脚本行，完成预期功能。

定制导航栏贡献

开发人员可以定制导航栏，用于拓展对应的导航栏视图，使之成为 EliRobot 机器人平台“插件”选项卡管理的定制导航栏之一，点击其激活条目可以在 EliRobot 主视图中展开该选项卡。

定制的导航栏主要有以下特征：

- 选项卡内容可作为外部软硬件设备或其他功能的配置入口；
- 其数据生命周期仅限本次 EliRobot 启动期间 (EliRobot 不为其提供数据持久化，如需持久化，需开发者自行处理)。

1.2 章节安排

为使开发人员能够快速上手、快速开发出同 EliRobot 机器人平台风格一致的外部贡献，ELITECO SDK 提供了大量的 api 接口、基于 Swing 的基础组件、开发文档、脚本等。因此本教程在接下来的部分将会对这些功能及特性按照由基础到高级，由框架到功能渐进式详尽描述，划分如下：

- 第 2 章：ELITECO SDK，主要介绍了文件结构及相关文件功能；
- 第 3 章：ELITECO Plugin 项目前期，描述了 ELITECO SDK API 提供的 UI 组件使用方法，并借助示例阐述了 ELITECO Plugin 项目的快速创建、构建、部署过程；
- 第 4 章：定制配置节点，以定制配置节点示例工程为例，讲解定制配置节点主要流程及所涉及的接口，同时也就附带一部分内部接口的使用示例；
- 第 5 章：定制任务节点，以定制任务节点示例工程为例，讲解定制任务节点主要流程及所涉及的接口，同时也就附带一部分内部接口的使用示例；
- 第 6 章：定制导航栏贡献，以定制导航栏贡献工程为例，讲解定制导航栏贡献主要流程及所涉及的接口，同时也就附带一部分内部接口的使用示例；
- 第 7 章：定制 Daemon，以定制 Daemon 示例工程为例，讲解定制 Daemon 守护程序主要流程及所涉及的接口；
- 第 8 章：其他功能特性，主要讲解部分内部接口中重要且开发者会常用的模块，并附带示例代码。

1.3 开发环境相关

ELITECO Plugin 开发需要依赖 Java SDK 8 开发环境，同时需要 Maven 3.5 以及 ant 分别作为依赖管理工具和构建流程管理工具。ELITECO Plugin 的前端 UI 框架使用的是 Java Swing，但不局限于 Java Swing，因此开发人员需要具备这些相关知识。

第 2 章 ELITECO SDK

ELITECO SDK 存放在 ELITECO Plugin 虚拟机开发平台的根目录/home/sdk 目录下，它包含了开发人员进行 ELITECO Plugin 开发所需要的接口 Jar 包、接口说明文档、依赖 jar 包、ELITECO Plugin 模板、定制实例工程、功能文档及脚本等。这些内容可使开发人员快速创建 ELITECO Plugin 工程，并较快的掌握如何使用 ELITECO SDK 提供的接口、基于 Swing 的 UI 组件及其他内部接口或功能实现自己的目标 ELITECO Plugin。

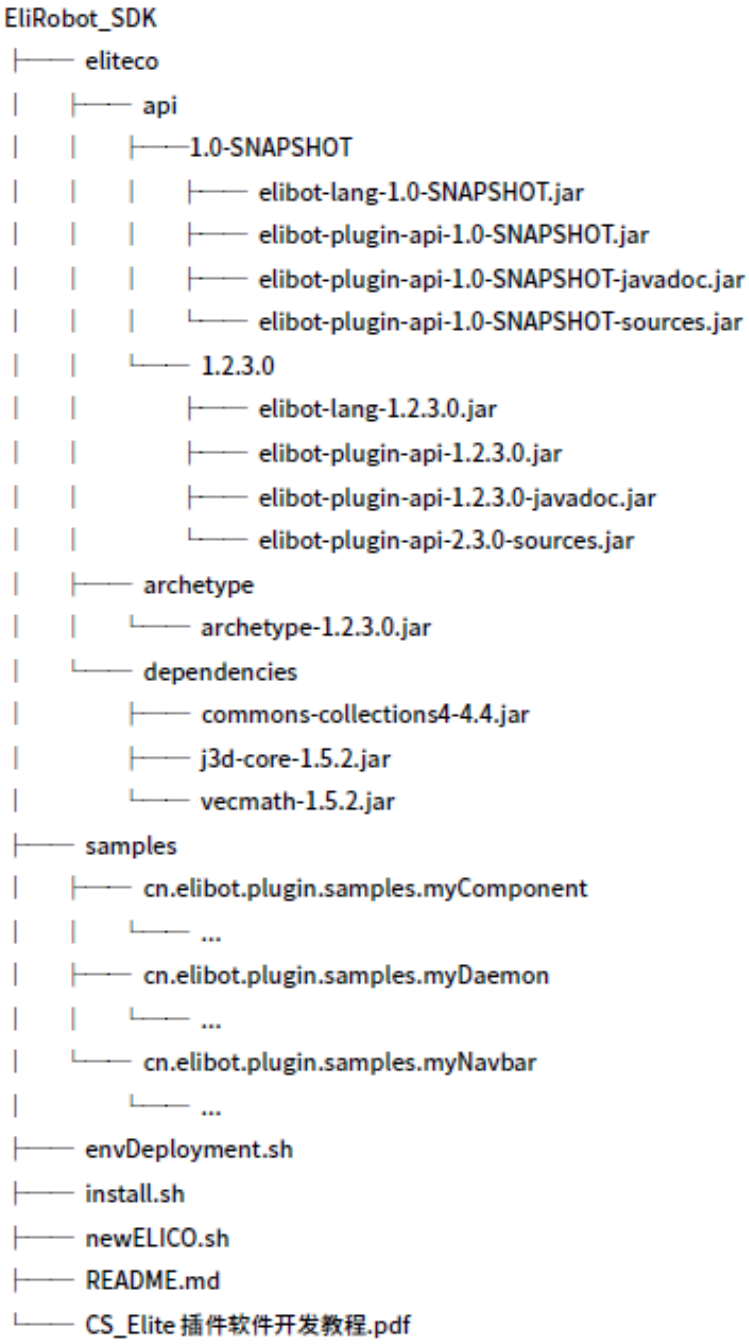


图 2-1: ELITECO SDK 文件结构

如上**图** 2-1所示为 ELITECO SDK 文件结构，下面针对该文件目录下重要文件及目录功能定位进行描述：

- /eliteco/api/: 该目录下包含了当前版本的 ELITECO API，每个版本的 ELITECO SDK API 都对应名称为当前版本号的目录，包含当前版本的 API jar 包、Javadoc 及 API 源码；
- /eliteco/archetype/: 该目录下是一个 ELITECO Plugin 工程模板，用于以此工程为模板使用 Maven 快速创建外部贡献工程，模板工具已经配置好了国际化、图标资源、依赖管理及构建管理等基础功能；
- /eliteco/dependencies/:ELITECO SDK API 的依赖 Jar 包；
- /samples/: 定制实例工程，开发者可以参考其中代码进行外部扩展定制；
- install.sh: 安装模板工程、ELITECO SDK API 及其他的依赖到 Maven 仓库；
- newELITECO.sh: 在当前路径下以 ELITECO Plugin 项目模板为蓝本，引导辅助开发者创建新的 ELITECO Plugin 项目创建；
- Elite 插件开发手册.pdf:ELITECO SDK API 相关功能文档及教程。

第 3 章 ELITECO Plugin 项目前期

在上一章中对 ELITECO SDK 文件结构及相关文件功能进行了描述，本章节开始对 ELITECO Plugin 项目开发的前期工作进行介绍。本章将会结合 demo 示例项目来向开发者展示 ELITECO Plugin 项目新建、项目国际化、GUI 支持组件库、项目构建及部署。目的是让开发者对 ELITECO Plugin 项目通用流程有一个整体认识，后面第 4 章-第 7 章 将主要讲解具体业务模块的定制流程，涉及到项目相关的通用流程将不再赘述。

3.1 ELITECO Plugin 项目新建

如上一章所述，ELITECO SDK 目录下 newELITECO.sh 用于引导开发者进行项目创建，终端运行 newELITECO.sh 即可以看到如**图** 3-1所示的 ELITECO Plugin 项目创建交互视图。

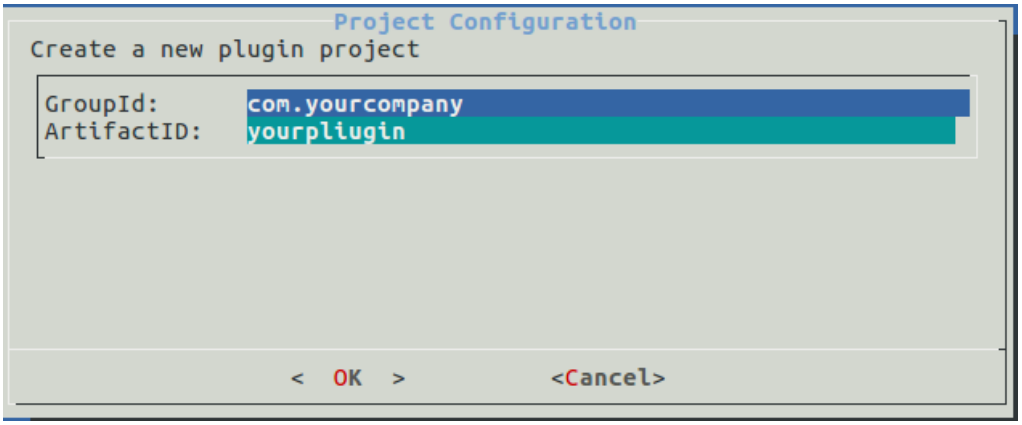


图 3-1: ELITECO Plugin 项目创建交互视图

正确完善 GroupId(组织 Id)、ArtifactID(项目名称) 后，通过 OK 确认提交，接下来会按照 ELITECO SDK 目录下的模板项目为蓝本进行新项目的自动创建，如**图** 3-2所示。创建完成后，屏幕会显示“BUILD SUCCESS”，如**图** 3-3所示。此时在 ELITECO SDK 目录下 (即 newELITECO.sh 同级目录) 已经生成了创建的项目。

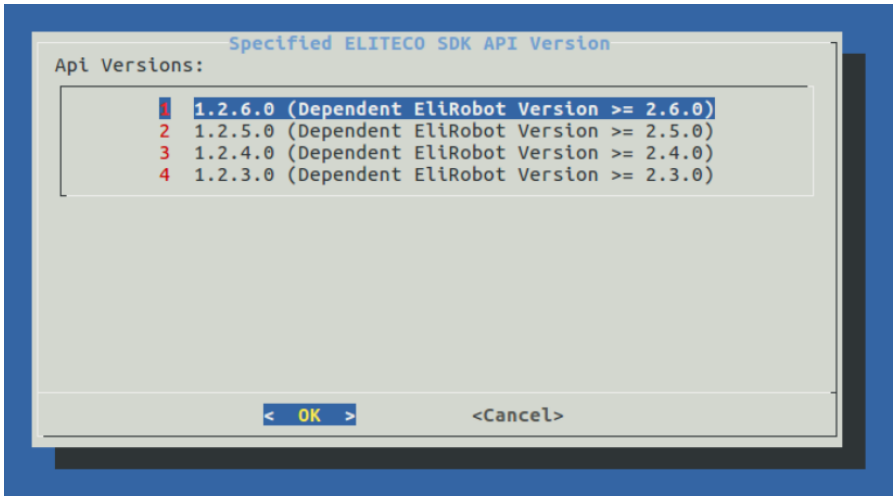


图 3-2: 新项目自动创建中

```
s @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:3.2.0:generate (default-cli) @ standalone-pom
---
[INFO] Generating project in Batch mode
[WARNING] Archetype not found in any catalog. Falling back to central repository
.
[WARNING] Add a repository with id 'archetype' in your settings.xml if archetype
's repository is elsewhere.
[INFO] ------------------------------------------------------------------------
---
[INFO] Using following parameters for creating project from Archetype: archetype
:1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
---
[INFO] Parameter: groupId, Value: cn.elibot.plugin.samples
[INFO] Parameter: artifactId, Value: myNavbar
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: cn.elibot.plugin.samples.myNavbar.impl
[INFO] Parameter: packageInPathFormat, Value: cn/elibot/plugin/samples/myNavbar/
impl
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: cn.elibot.plugin.samples.myNavbar.impl
[INFO] Parameter: groupId, Value: cn.elibot.plugin.samples
[INFO] Parameter: apiversion, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: myNavbar
[INFO] Project created from Archetype in dir: /home/elite/EliRobot_SDK/myNavbar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.896 s
```

图 3-3: 新项目自动创建完毕

启动桌面 IntelliJ IDEA 开发工具，导入创建的项目进行后续开发。编辑 pom.xml 文件，添加 ELITECO Plugin 相关信息。包含指定供应商、联系人地址、版权、描述和许可证信息的元数据。当 ELITECO Plugin 安装在 EliRobot 机器人平台中时，这些信息将显示给用户。有关这些信息内容如下代码块3.1所示：

```
1 <!--**********************************************************************
    -->
2 <!--   Note: Update this section with relevant metadata      -->
3 <!--     that comes along with your ELITECO Plugin          -->
4 <!--**************** BEGINNING OF ELICO PLUGIN META DATA *************--
    >
5 <plugin.symbolicname>${project.groupId}.${project.artifactId}</plugin.
    symbolicname>
```

```
<plugin.bundleActivator>cn.elibot.plugin.samples.navbar.Activator</
    plugin.bundleActivator>
<plugin.licenseType>Sample license</plugin.licenseType>
<plugin.licenseIcon/>
</plugin.iconURL/>
<plugin.smallIconURL/>
<plugin.description>Describe About Navigator Contribution</plugin.
    description>
<plugin.vendor>ELITE ROBOTS</plugin.vendor>
<plugin.contactAddress>Building 18, Lane 36, Xuelin Roas, Pudong
    District, Shanghai</plugin.contactAddress>
<plugin.copyright>Copyright(C)${copyright.year}ELITE ROBOTS. All Rights
    Reserved.</plugin.copyright>
<!--*************** END OF ELICO PLUGIN METADATA *********************--
    >
<!--************************************************************************
    -->
```

代码块 3.1: ELITECO Plugin 基础元数据信息

3.2 ELITECO Plugin 项目准备

在正式的 ELITECO Plugin 项目开始前，还需要开发者了解一些前期基础数据及功能：License、国际化及图片资源等，因此在 ELITECO Plugin 定制项目正式开始之前，有必要对这些通用流程进行了解。下面就以一个简单定制项目-导航栏贡献定制 demo 为示例，讲述这些 ELITECO Plugin 项目通用流程，该定制示例项目主要是创建了导航栏贡献：包含一个激活条目和一个选项卡视图，前者带有图标和文本，用于点击后展示选项卡视图。该项目重点不在于如何定制导航栏贡献（第 6 章定制工具栏按钮中重点讲述），而是细致的讲解 ELITECO Plugin 项目的通用定制流程，在后续定制功能模块贡献章节中，将重点讲解功能模块的定制流程，不再对通用流程进行讲解。

按照第 3.1 节 ELITECO Plugin 项目新建流程新建导航栏贡献定制 demo-myNavbar 文件结构如下**图** 6-1所示：

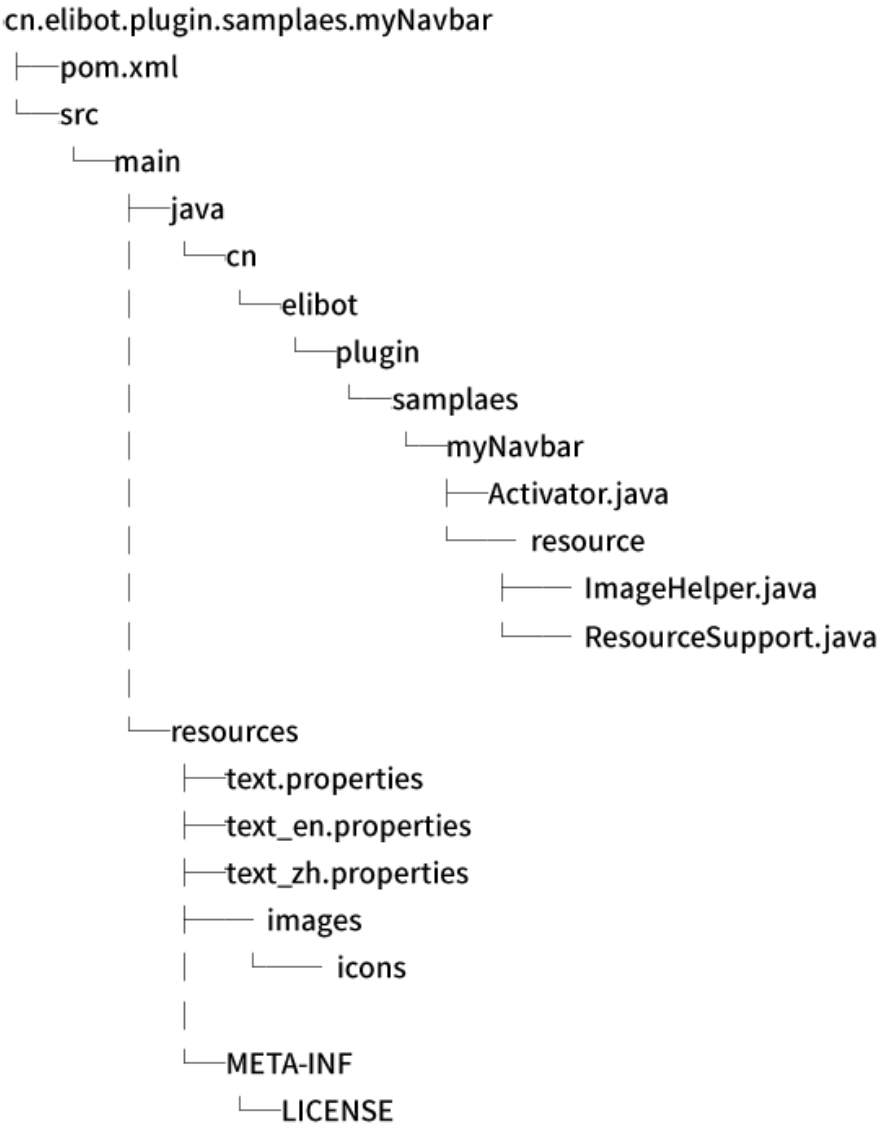


图 3-4: myNavbar 项目文件结构

3.2.1 LICENSE 文件

开发者可以根据实际的需求准备 License 文件放入到项目中。LICENSE 的存放路径为/src/main/resource/META-INF/LICENSE。

3.2.2 国际化资源

EliRobot 机器人平台目前支持中英文双语，如**图** 6-1所示新建 ELITECO Plugin 实例的多语文件路径/src/main/resources/下，当然开发人员也可以直接在该目录下创建所需要的其他语言环境翻译文件。

ELITECO Plugin 支持开发者可以在开发过程中使用各种多语资源,并在模板创建的 ELITECO

Plugin 项目中提供了 ResourceSupport.java 来帮助开发者管理和快捷访问多语资源。如下代码块3.2所示为 ResourceSupport.java 文件内容。具体多语资源的使用方法将在 ELITECO Plugin 进行讲解。

```
1   public class ResourceSupport {
2     private static LocaleProvider provider;
3     private static String properties = "text";
4
5     public static void setLocaleProvider(LocaleProvider provider) {
6       ResourceSupport.provider = provider;
7    }
8
9     public static LocaleProvider getLocaleProvider() {
10      return ResourceSupport.provider;
11    }
12    // 获取指定Locale的多语Bundle
13    public static ResourceBundle getResourceBundle(Locale locale){
14      return ResourceBundle.getBundle(properties, locale);
15    }
16    // 获取默认Locale的多语Bundle
17    public static ResourceBundle getDefaultResourceBundle(){
18      if (provider != null) {
19        return ResourceBundle.getBundle(properties, provider.getLocale
    ());
20      }else{
21        return null;
22      }
23    }
24  }
```

代码块 3.2: ResourceSupport 文件内容

如上代码块3.2所示，其中 property 表示多语资源文件的相对路径 (相对于资源目录 resources) 及多语资源文件名的结合，以便能够准确获取对应的多语资源 Bundle (包含对应语言环境所有已定义多语资源，可通过 getString() 方法获取对应语言的翻译结果)。如上 myNavbar 中 property 内容为“text”。

代码块3.2主要方法中 getLocaleProvider/setLocaleProvider 主要用力 get/set 获取的 EliRobot 机器人平台的语言环境 provider，比较简单，且后文会有涉及，不予过多赘述。

getResourceBundle 方法则用于获取指定 Locale 语言环境的多语资源 bundle，可通过 bundle 的 getString 方法获取对应翻译资源，后续章节中会有所涉及。

getDefaultResourceBundle 方法则用于获取 EliRobot 机器人平台当前语言环境的多语资源 bundle，可通过 bundle 的 getString 方法获取对应翻译资源。后续章节中会有所涉及。

3.2.3 图片资源

ELITECO Plugin 支持开发者可以在开发过程中使用各种图片资源,并在模板创建的 ELITECO Plugin 项目中提供 ImageHelper.java 来帮助开发者管理和快捷访问图片资源。ELITECO Plugin 图标资源的默认路径：/src/main/resources/image/icons/，开发者可以通过 ImageHelper.java 类来获取图标进行使用。如下代码块3.3所示为 ImageHelper.java 的文件内容。

```
1   public class ImageHelper {
2     private ImageHelper() {
3     }
4
5     public static ImageIcon loadImage(String name) {
6       ImageIcon image = null;
7
8       try {
9         URL url = ImageHelper.class.getResource("/images/icons/" +
    name);
10        if (url != null) {
11          Image img = Toolkit.getDefaultToolkit().createImage(url);
12          if (img != null) {
13            image = new ImageIcon(img);
14          }
15        }
16      } catch (Exception e) {
17        e.printStackTrace();
18      }
19
20      return image;
21    }
22  }
```

代码块 3.3: ImageHelper 文件内容

如代码块3.3所示，loadImage 方法图片资源的加载路径是相对于资源路径 resources 而言，因此根据**图** 6-1：myNavbar 项目文件结构所示文件结构，可知 myNavbar 加载图片资源方式如代码块3.3的第 10 行所示，加载路径为“/images/icons/”。因此若实际开发中，开发者调整了图片资源路径，需要相应的调整此处的加载路径。

3.2.4 服务注册绑定

定制的 ELITECO Plugin 输出文件将通过 Activator 方式将服务注册到 EliRobot 机器人平台中。被安装到 EliRobot 机器人平台中后，EliRobot 机器人平台启动时，将会通过 ELITECO

Plugin 项目的 Activator 的 start 方法，进行定制模块服务注册，并会将获取到 EliRobot 机器人平台中的语言环境放入到 ResourceSupport 中，以便用户进行国际化的相关操作 (newELITECO.sh 创建的项目里获取 EliRobot 机器人平台中的语言环境并更新到本地多语管理类 ResourceSupport 中的操作已自动处理)。如下代码块3.4所示为当前 Activator.java 内容：

```
1   public class Activator implements BundleActivator {
2       private ServiceReference<LocaleProvider>
    localeProviderServiceReference;
3
4     @Override
5      public void start(BundleContext bundleContext) throws Exception{
6        localeProviderServiceReference = bundleContext.
    getServiceReference (LocaleProvider.class);
7        if (localeProviderServiceReference != null) {
8       LocaleProvider localeProvider = bundlecontext.getService(
    localeProviderServiceReference);
9        if (localeProvider != null) {
10          ResourceSupport.setLocaleProvider(localeProvider);
11        }
12      }
13      System.out.printIn("cn.elibot.plugin.samples.myNavbar.Activator
    says Hello World!");
14      BundleContext.registerService(NavbarService.class, new
    MyNavbarServiceImpl(), null);
15      }
16
17      @Override
18      public void stop(BundleContext context) {
19          System.out.printIn(“cn.elibot.plugin.samples.myNavbar.
    Activator says Goodbye World!”);
20      }
21  }
```

代码块 3.4: Activator 文件内容

如上代码块3.4中 Activator 中的 start 方法会在该 EliRobot 机器人平台加载该项目所在 bundle 时的入口，其参数 BundleContext 是一个 Bundle 上下文 (context)，该 context 允许开发者通过改接口访问框架相关的一些方法，因此该方法通常用来获取系统默认语言环境以及注册定制模块服务到系统中，而 stop 方法则是该项目被卸载时会执行的方法。

如前 start 方法相关描述中提到，该方法用来注册定制模块服务，ELITECO Plugin 支持的 4 种定制功能类型，都是需要在定制实现由相关服务类管理，并在 Activator 中的 start 方法中完成注册相关语句。在 EliRobot 机器人平台加载该项目所在 bundle 时被注册到 EliRobot

机器人平台。ELITECO Plugin 支持的 4 种定制功能项目与 EliRobot 机器人平台大致关系如**图** 3-5所示。

定制任务节点服务类、定制配置节点服务类、导航栏服务类及 Daemon 服务类都要通过 Activator.java 中 start 方法的参数 BundleContext 注册到框架中，以便在安装到 EliRobot 机器人平台后，能被系统加载，几种功能模块定制服务类注册方法如下代码块3.5所示:

```
1     @Override
2     public void start(BundleContext bundleContext) throws Exception {
3         // 定制任务节点服务 YourTaskNodeService 注册
4     bundleContext.registerService(SwingTaskNodeService.class, new
      YourTaskNodeService(), null);
5         // 定制配置节点服务 YourConfigurationNodeService 注册
6         bundleContext.registerService(SwingConfigurationNodeService.
      class, new YourConfigurationNodeService(), null);
7         // 定制导航栏服务 YourNavbarService 注册
8         bundleContext.registerService(NavbarService.class, new
      YourNavbarService(), null);
9         // 定制Daemon服务 YourDaemonService 注册
10        bundleContext.registerService(DaemonService.class, new
      YourDaemonService(), null);
11    }
```

代码块 3.5: 功能模块定制服务类注册方法

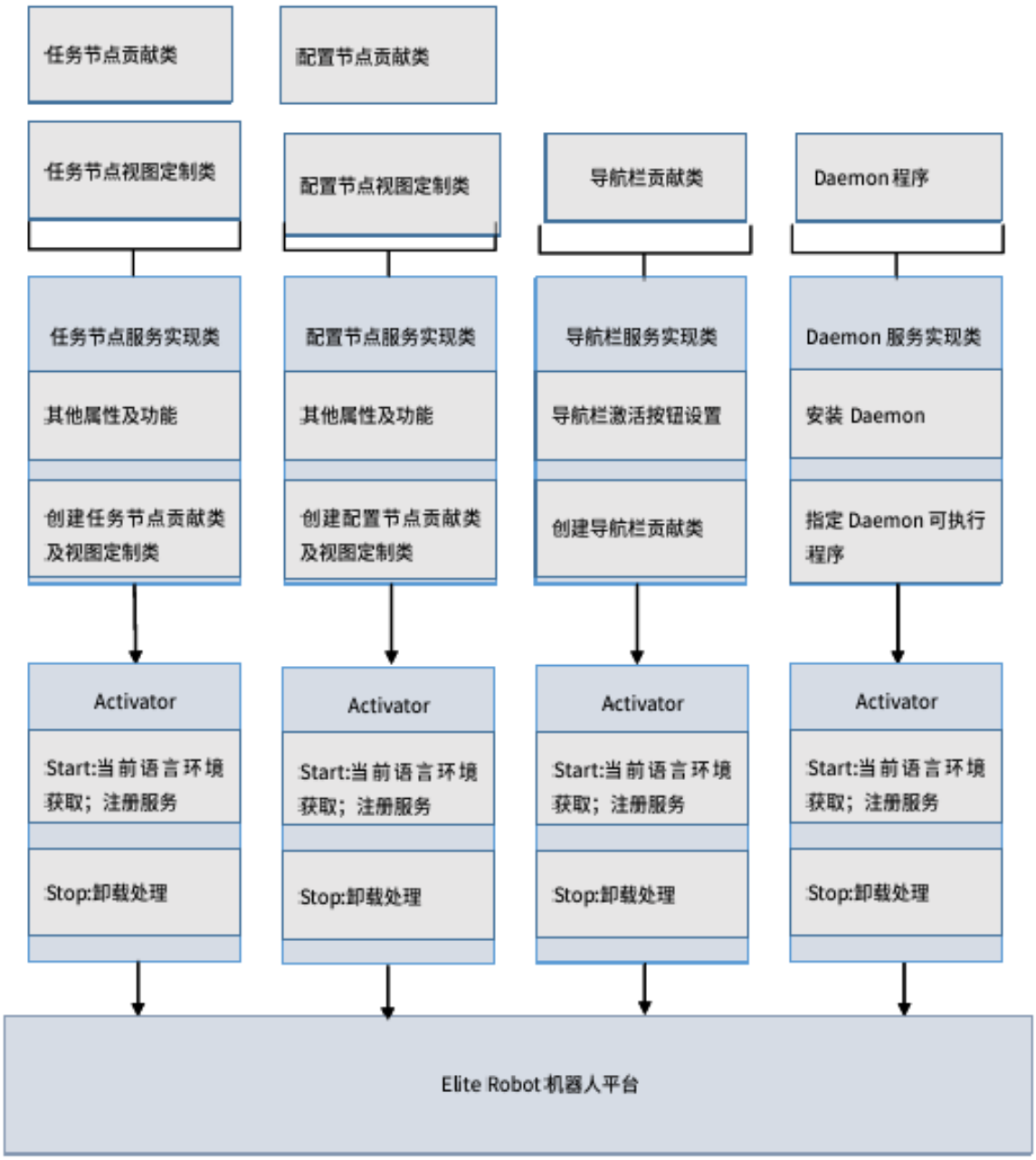


图 3-5: 4 种 ELITECO Plugin 定制项目与 EliRobot 机器人平台关系

3.3 ELITECO Plugin 项目实现

基于第 3.1 节 ELITECO Plugin 项目新建中新建的 myNavbar 项目及第 3.2 节 ELITECO Plugin 项目准备 中相关特性的讲解，实现 myNavbar 项目中业务功能：添加 EliRobot 机器人平台导航栏贡献，该贡献有一个带有图标和名称的激活按钮，点击该激活按钮会在主视图中展开该贡献的视图，其中视图带有一个多语标题和一个显示英文文本的标签。

由**图** 3-5可知定制导航栏贡献需要实现导航栏贡献类及导航栏服务，并在 Activator 中注册导航栏服务，则按照功能设计实现导航栏贡献类及导航栏服务类分别如代码块3.6及代码

块 6.1所示。导航栏贡献定制将会在第 6 章 定制导航栏贡献 中详细讲解，此处仅用于展示 ELITECO Plugin 项目通用流程。

```
 1    public class MyNavbarContributionImpl implements NavbarContribution
     {
 2        private final ResourceBundle defaultLocaleBundle;
 3
 4        public MyNavbarContributionImpl() {
 5            defaultLocaleBundle = ResourceSupport.
     getDefaultResourceBundle();
 6        }
 7
 8        @Override
 9        public void buildUI(JPanel panel) {
10            panel.setLayout(new BorderLayout(5, 5));
11
12            JPanel mainPanel = SwingService.seniorPaneService.
     createSeniorPane(this.defaultLocaleBundle.getString("hello_world"));
13            mainPanel.setLayout(new BorderLayout());
14            panel.add(mainPanel, BorderLayout.CENTER);
15
16            JLabel elitePluginLabel = new JLabel(this.enUsLocaleBundle.
     getString("hello_elite_plugin"));
17            elitePluginLabel.setHorizontalAlignment(SwingConstants.
     CENTER);
18            elitePluginLabel.setPreferredSize(new Dimension(280, 36));
19            elitePluginLabel.setMaximumSize(new Dimension(280, 36));
20            elitePluginLabel.setMinimumSize(new Dimension(280, 36));
21            mainPanel.add(elitePluginLabel, BorderLayout.CENTER);
22        }
23
24        @Override
25        public void openView() {
26            System.out.println("Hello World Navigator Contribution View
     Opened.");
27        }
28
29        @Override
30        public void closeView() {
31            System.out.println("Hello World Navigator Contribution View
     Closed.");
32        }
33    }
```

代码块 3.6: MyNavbarContributionImpl.java 文件内容

```
1   public class MyNavbarServiceImpl implements NavbarService {
2       @Override
3       public void configure(NavbarContext context) {
4           context.setNavbarIcon(ImageHelper.loadImage("demo.png"));
5           context.setNavbarName(ResourceSupport.
    getDefaultResourceBundle().getString("hello_world"));
6       }
7
8       @Override
9       public MyNavbarContributionImpl createContribution() {
10          return new MyNavbarContributionImpl();
11      }
12  }
```

代码块 3.7: MyNavbarServiceImpl.java 文件内容

由代码块3.6及代码块6.1 导航栏贡献类中 buildUI 方法在栏贡献贡献视图中创建了带标题页面并设置与 EliRobot 机器人平台相同语言环境的标题，同时在带标题页面内布局了一个文本标题并设置“Locale.US”语言环境的文本；导航栏贡献类 open/close 则在事件触发时输出了一段文本；导航栏服务类中则 configure 方法则配置了导航栏贡献的图标和名称；导航栏服务类 createContribution 则是创建导航栏贡献类实例。

最后在 Acvivator.java 类的 start 方法中完成导航栏服务类注册后代码如下代码块3.8所示：

```
1   public class Activator implements BundleActivator {
2       private ServiceReference<LocaleProvider>
    localeProviderServiceReference;
3
4       @Override
5       public void start(BundleContext bundleContext) throws Exception
    {
6           localeProviderServiceReference = bundleContext.
    getServiceReference(LocaleProvider.class);
7           if (localeProviderServiceReference != null) {
8               LocaleProvider localeProvider = bundleContext.getService
    (localeProviderServiceReference);
9               if (localeProvider != null) {
10                  ResourceSupport.setLocaleProvider(localeProvider);
11              }
12          }
13          System.out.println("cn.elibot.plugin.myNavbar.Activator says
     Hello World!");
```

```
14              bundleContext.registerService(NavbarService.class, new
    MyNavbarServiceImpl(), null);
15          }
16          @Override
17          public void stop(BundleContext bundleContext) throws Exception {
18              System.out.println("cn.elibot.plugin.samples.myNavbar.
    Activator says Goodbye World!");
19          }
20      }
```

代码块 3.8: 完成导航栏服务类注册的 Acvivator.java 文件内容

至此 MyNavbar 导航栏贡献定制完成，当前其目录结构如**图** 3-6所示。

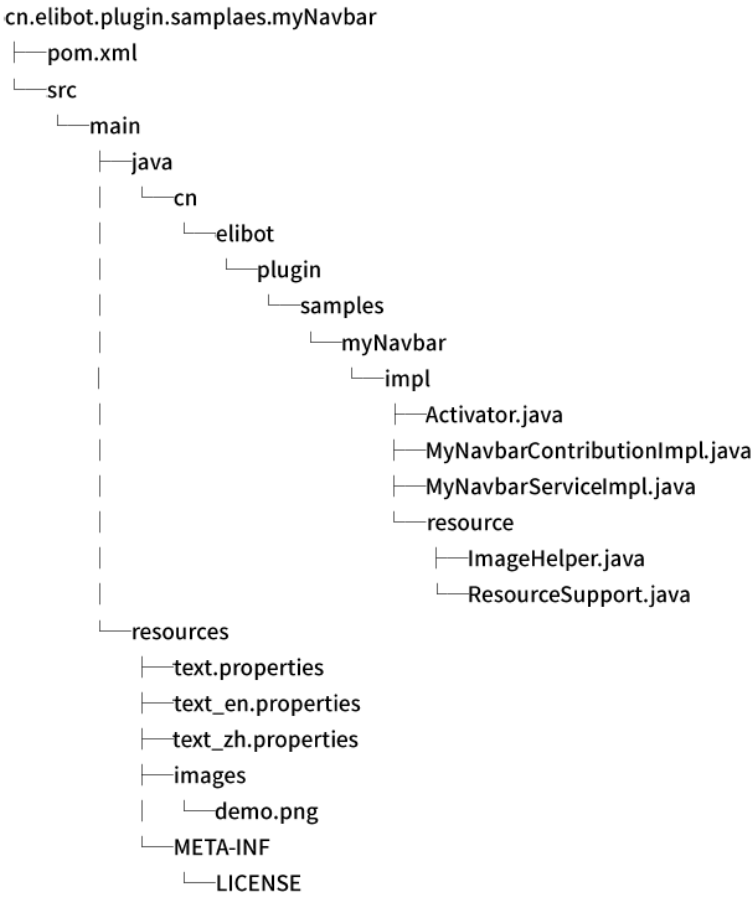


图 3-6: 功能实现完毕的 myNavbar 项目文件结构

3.4 ELITECO Plugin 项目构建与部署

项目构建

开发人员可以直接通过 IDEA 提供的 maven 插件直接进行 package 打包，也可以在终

端中进入项目根目录通过 mvn package 命令进行打包。项目打包之后将会在项目根目录下生成一个 target 目录，该目录下的 elico 类型的文件即为 ELITECO Plugin 插件文件。

插件部署

EliRobot 机器人平台支持在多种平台上运行，因此部署方式也有所不同，主要部署方式如下：

- Linux/Windows PC 平台：将前文生成的插件文件拷贝到 EliRobot 机器人平台安装目录下的 program 目录下，即可被 EliRobot 机器人平台识别用于加载；
- 机器人示教器：将前文生成的插件文件拷贝拷贝到 U 盘中，并将 U 盘插入机器人示教器。待机器人示教器识别并成功挂载 U 盘后可拷贝对应插件文件到 EliRobot 机器人平台安装目录系统下即可用于加载。

将插件文件拷贝到指定目录后 (示教器平台则是插入 U 盘，拷贝对应插件文件到 EliRobot 机器人平台安装目录系统下)，在 EliRobot 机器人平台右上角 Elite Logo“更多”按钮激发后展开的抽屉组件中点击“设置”按钮，并在展开的系统设置页面点击“ELITECOs”进入插件管理页面如**图** 3-7所示。

图 3-7: 插件管理页面

如**图** 3-7所示插件管理页面中有“已激活”、“未激活”两个选项卡，前者用于展示当前处于已激活状态的插件，表示这些插件已经加载到 EliRobot 机器人平台并处于已激活状态；而后者则表示已经加载到 EliRobot 机器人平台但处于未激活状态 (未启动)。

“未激活”选项卡激发按钮右侧的添加按钮“+”则用于加载处于 EliRobot 机器人平台目录系统下 (主要指 program 及其子目录下) 的插件，点击该按钮弹出文件对话框如**图** 3-8所

示，示教器则需要点击文件对话框右上角的 U 盘 logo 打开 U 盘目录，如图 3-9所示。使用文件管理器上方的拷贝按钮将插件从 U 盘目录拷贝到 EliRobot 机器人平台目录系统下，并通过“ELITECOs”管理页面进行加载，但需注意加载后的插件处于“已加载未激活”，在“未激活”选项卡下即可看到该插件，如图 3-10所示该插件处于不可用状态，此时重新启动 EliRobot 机器人平台即可启动该插件使之处于“已激活”状态。

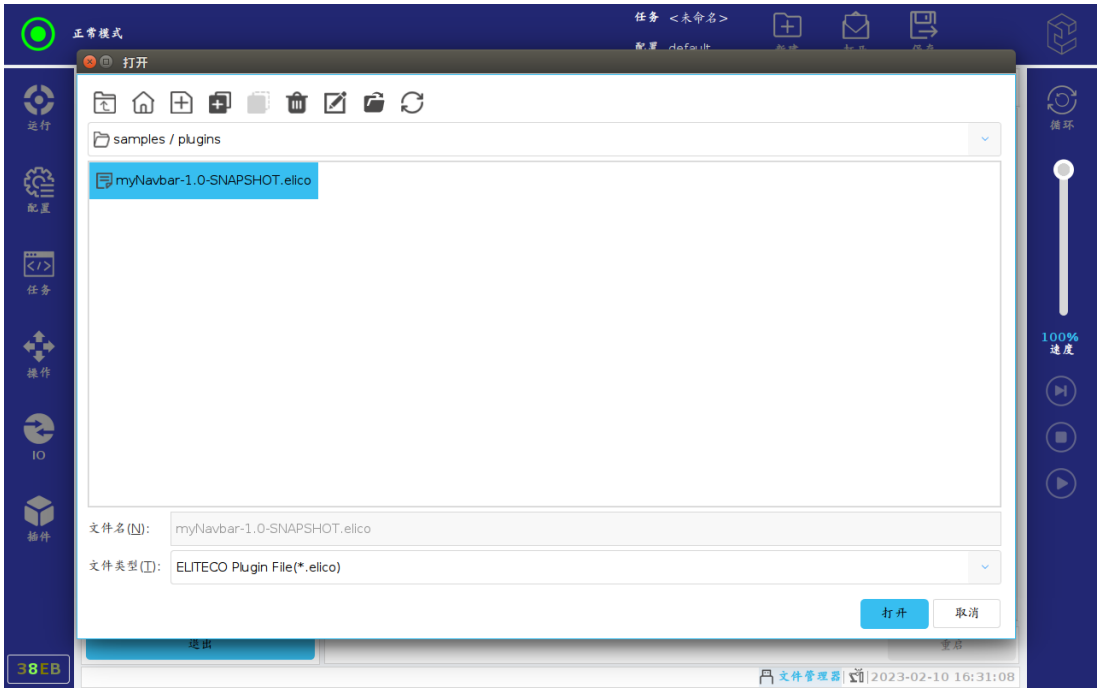


图 3-8: 本地插件添加对话框

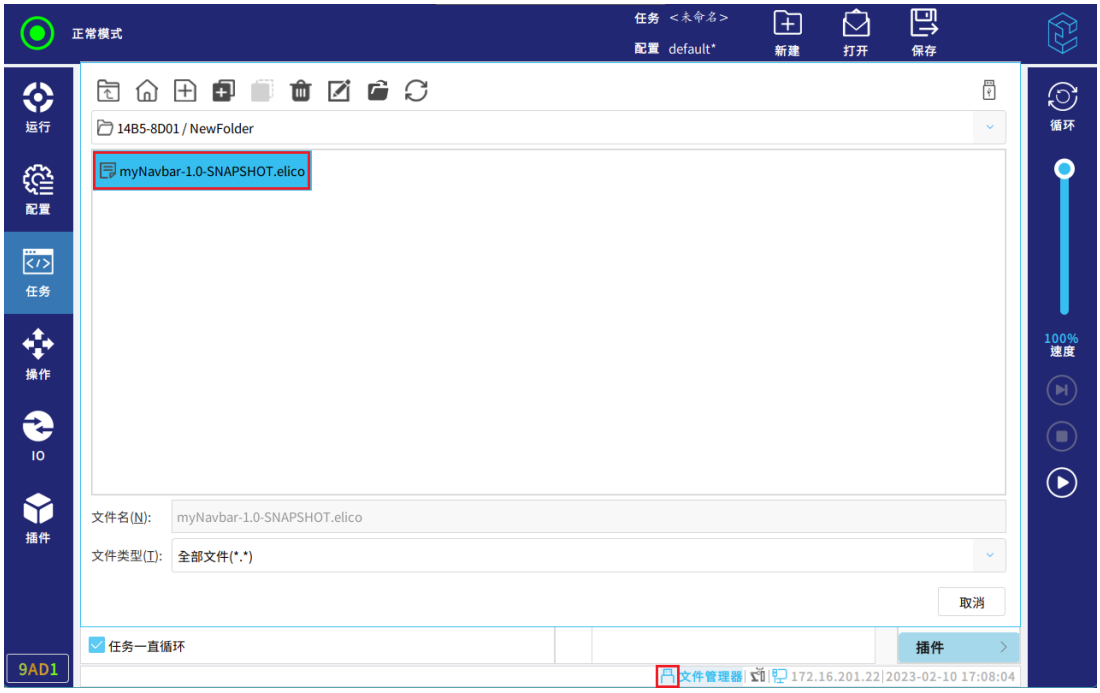


图 3-9: 将 U 盘目录下插件拷贝至机器人平台目录

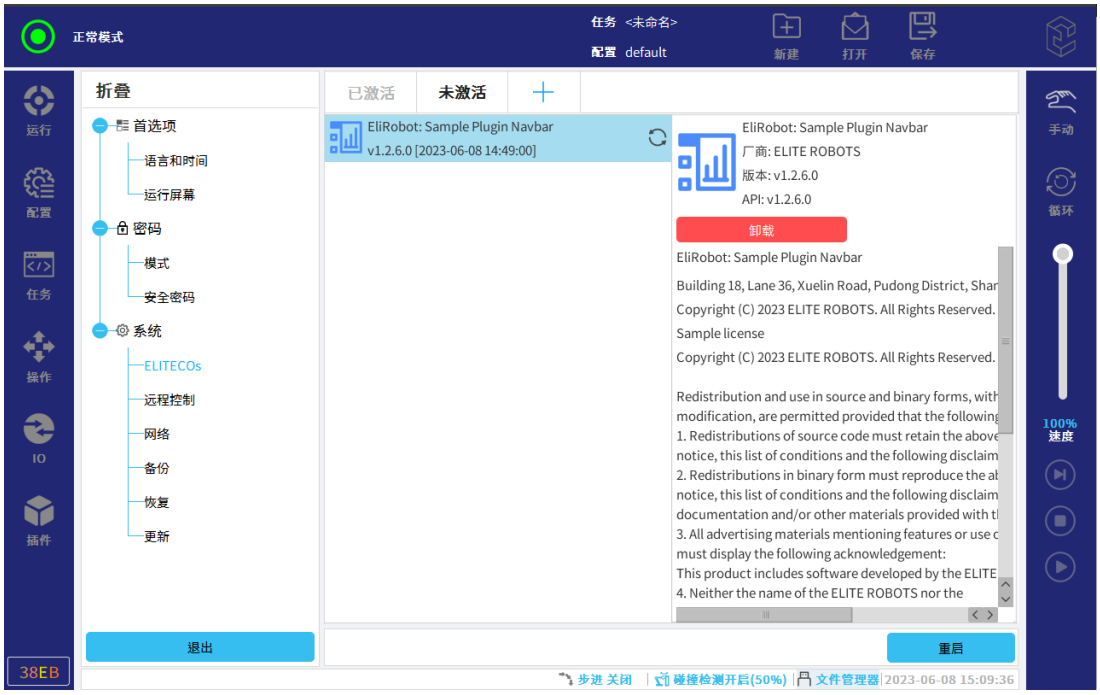


图 3-10: myNavbar 插件不可用状态

由上所述可知，加载插件不会使之处于已激活状态，需要重启 EliRobot 机器人平台才能使之处于已激活状态。同理，卸载插件也只是将该插件从插件管理页面移除，但插件功能依然能够正确使用，需要重启才能使卸载生效，插件在对应模块不再显示。

其中可见**图** 3-10中在选项卡视图右侧的详细信息页面，显示有插件版本、开发者、名称、单位地址、版权信息等在 myNavbar 项目 pom 文件中定义的插件属性。点击 ELITECOs 管理页面右下角的“重启”按钮即可重启 EliRobot 机器人平台，使 myNavbar 插件处于已激活状态，如**图** 3-11所示为 myNavbar 插件中的导航栏贡献按钮，其文本显示已经跟随 EliRobot 机器人平台语言环境显示为中文。点击后，进入 myNavbar 插件导航栏贡献视图，如**图** 3-12所示，其页面标题也跟随 EliRobot 机器人平台语言环境显示为中文，同时页面中部的文本标签显示文本则是按照指定的语言环境显示为英文。

图 3-11: myNavbar 导航栏贡献按钮

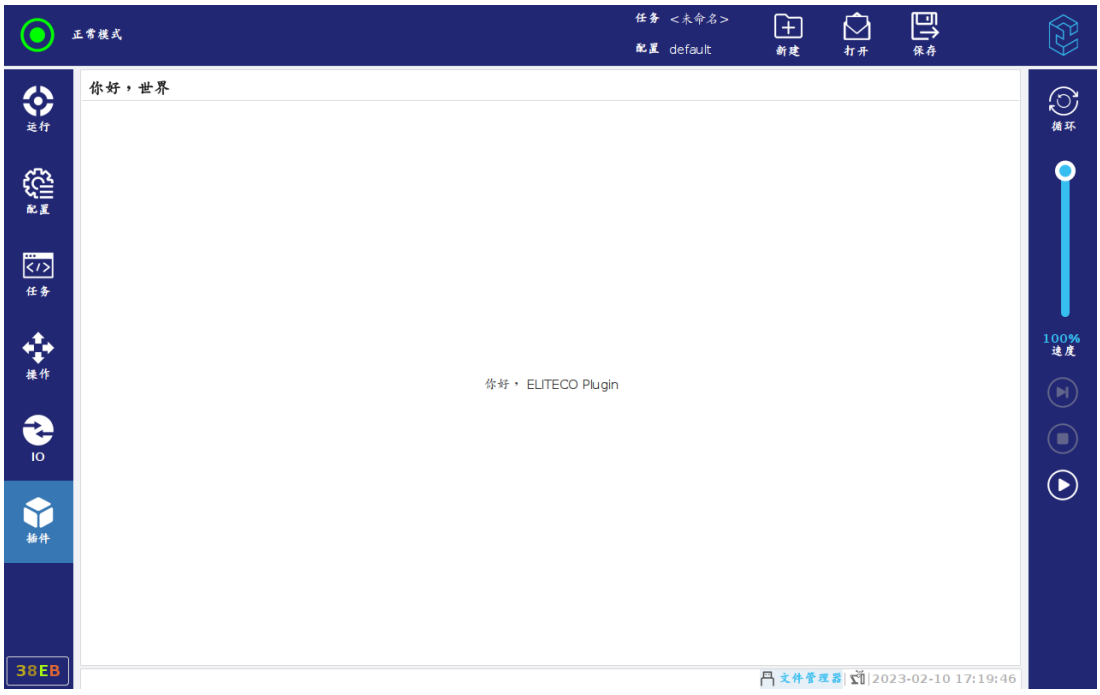


图 3-12: myNavbar 导航栏贡献视图

至此，本章已经通过 myNavbar 项目将 ELITECO Plugin 项目的整个流程及过程中的一些特性进行了详细的讲解。其中涉及到的导航栏贡献定制不是本章重点，仅仅作为示例来为开发者演示 myNavbar 项目全流程，第 6 章 定制导航栏贡献中会有详细描述。

3.5 ELITECO Plugin UI 组件

ELITECO SDK API 提供了大量的基于 Swing 的 UI 组件供开发者在定制相关模块视图时使用，以期开发者定制的 ELITECO Plugin 前端视图能够与 EliRobot 机器人平台保持风格一致。因此作为本章最后一节将重点对这些组件的用法进行讲解。

3.5.1 手风琴导航组件

手风琴导航组件可以通过与 Swing 组件中的 CardLayout 组件结合使用，可以快速的进行页面切换。用户可以通过 SwingService.accordionService 接口来创建 Accordion 组件进行使用。具体结合使用：

```
1     CardLayout cl = (CardLayout) (mainPane.getLayout());
2     DefaultListModel<String> listModel = new DefaultListModel<>();
3     listModel.addElement(Common.change(resourceBundle.getString("
      base_font")));
4     listModel.addElement(Common.change(resourceBundle.getString("
      base_button")));
5     listModel.addElement(Common.change(resourceBundle.getString("
      base_switch")));
6     JList<String> list = new JList<String>(listModel) {
7         @Override
8         public int locationToIndex(Point location) {
9             int index = super.locationToIndex(location);
10            if (index != -1 && !getCellBounds(index, index).contains(
      location)) {
11                return -1;
12            } else {
13                return index;
14            }
15        }
16    };
17    list.addListSelectionListener(e -> cl.show(mainPane, list.
      getSelectedValue()));
18
19    DefaultListModel<String> listModel1 = new DefaultListModel<>();
20    listModel1.addElement(Common.change(resourceBundle.getString("
      senior_senior")));
21    listModel1.addElement(Common.change(resourceBundle.getString("
      senior_keyboard")));
```

```
22      listModel1.addElement(Common.change(resourceBundle.getString("
    senior_message")));
23      listModel1.addElement(Common.change(resourceBundle.getString("
    senior_dialog")));
24      listModel1.addElement(Common.change(resourceBundle.getString("
    senior_tip")));
25      JList<String> list1 = new JList<String>(listModel1) {
26          @Override
27          public int locationToIndex(Point location) {
28              int index = super.locationToIndex(location);
29              if (index != -1 && !getCellBounds(index, index).contains(
    location)) {
30                  return -1;
31              } else {
32                  return index;
33              }
34          }
35      };
36      list1.addListSelectionListener(e -> cl.show(mainPane, list1.
    getSelectedValue()));
37      List<AccordionModel> amList = new ArrayList<>();
38      amList.add(new AccordionModel(Common.change(resourceBundle.getString
    ("main_base")), list));
39      amList.add(new AccordionModel(Common.change(resourceBundle.getString
    ("main_senior")), list1));
40      JComponent accordion = SwingService.accordionService.showAccordion(
    amList);
41      accordion.addPropertyChangeListener("tab", new
    PropertyChangeListener() {
42          @Override
43          public void propertyChange(PropertyChangeEvent evt) {
44              cl.show(mainPane, evt.getNewValue().toString());
45          }
46      });
47
48      JSplitPane splitPane = new JSplitPane(1, true, accordion, mainPane);
49      splitPane.setDividerLocation(180);
50      splitPane.setOneTouchExpandable(true);
51      panel.add(splitPane, BorderLayout.CENTER);
52      // card展示页面
53      cl.show(mainPane, Common.change(resourceBundle.getString("base_font
    ")));
54
```

页面展示效果如**图** 3-13所示：

图 3-13: 手风琴导航页面效果图

3.5.2 开关按钮组件

开发人员可以通过 SwingService.switchButtonService 接口来创建开关按钮进行使用。在创建开关按钮时可以选择设置大小、打开与关闭颜色。通过按钮的 isSelect() 方法来确定按钮的状态，true 是打开，false 是关闭。

创建一个默认的开关按钮，代码如下：

```
1    JToggleButton s1 = SwingService.switchButtonService.
   createSwitchButton();
```

创建一个指定大小的开关按钮，代码如下：

```
1    JToggleButton s2 = SwingService.switchButtonService.
   createSwitchButton(new Dimension(60, 30));
```

创建一个指定大小和颜色的开关按钮，代码如下：

```
1    JToggleButton s3 = SwingService.switchButtonService.
   createSwitchButton(new Dimension(100, 40), Color.GREEN, Color.
   LIGHT_GRAY);
```

通过监听按钮的 ItemListener 来监听按钮的选中状态，代码如下：

```
    s3.addItemListener(e -> {
        if (s3.isSelected()) {
            System.out.println("on");
        } else {
            System.out.println("off");
        }
    });
```

实现的页面效果如**图** 3-14所示，具体使用可以参考 NavbarExtensionImpl.java 中的相关示例。

图 3-14: 开关按钮页面效果图

3.5.3 高级面板组件

为保持与 EliRobot 页面风格一致，ELITECO SDK API 提供了高级面板组件。开发人员可以通过 SwingService.seniorPaneService 接口来创建面板进行使用。在创建面板的时候可以指定圆角弧度大小以及设置面板的标题。

创建一个默认的面板，代码如下：

```
    JPanel s1 = SwingService.seniorPaneService.createSeniorPane();
    s1.setBounds(10, 60, 400, 250);
    s1.add(new JLabel("Default SeniorPanel"));
```

创建一个圆角的面板，代码如下：

```
    JPanel s2 = SwingService.seniorPaneService.createSeniorPane(30);
    s2.setBounds(10, 320, 400, 250);
    s2.add(new JLabel("SeniorPanel with fillet radius 30"));
```

创建一个带标题的面板，代码如下：

```
    JPanel s3 = SwingService.seniorPaneService.createSeniorPane("Title
   Message");
    s3.setBounds(420, 320, 400, 250);
    s3.add(new JLabel("SeniorPanel with title"));
```

具体使用可以参考 NavbarExtensionImpl.java 中的相关示例。页面效果如**图** 3-15：

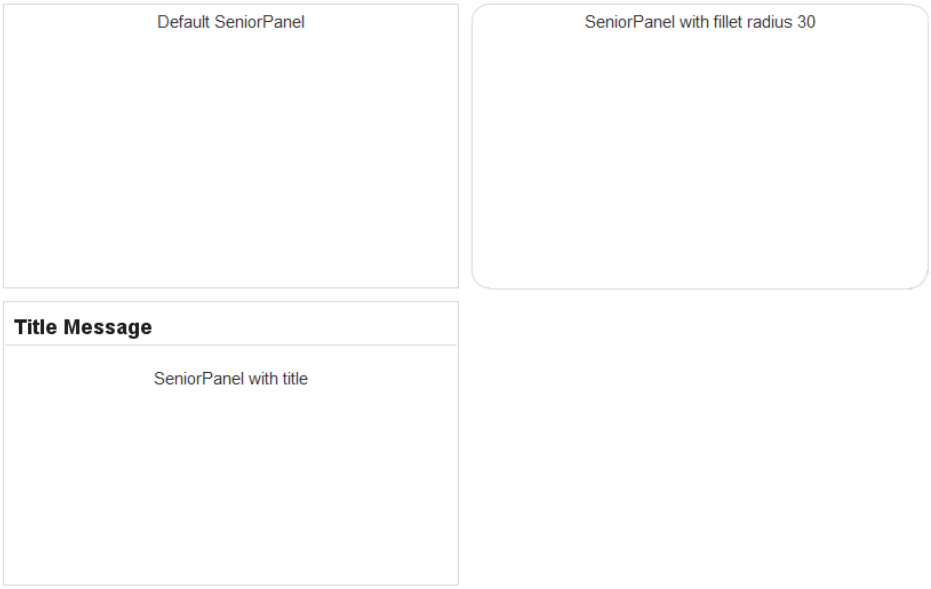


图 3-15: 面板页面效果图

3.5.4 文件选择器组件

文件选择器组件支持自定义路径、自定义过滤器、监听打开文件事件、监听关闭事件等。用户可以通过文件选择器选择所需的任务脚本、插件文件等使用。其代码如下：

```
BaseFileFilter txtFileFilter = new BaseFileFilter() {
    @Override
    public String getFileExtension() {
        return "txt";
    }    @Override
    public String getDescription() {
        return "任务结构文件(.txt)";
    }
};
File file = taskApiProvider.getSystemAppProvider().getSystemSettings().
   getProgramPath();
FileChooserView fileChooserView = taskApiProvider.getComponentHubService
   ().getFileChooserViewService().createFileChooserView();
```

```
fileChooserView.setCurrentDirectory(file);
fileChooserView.setFileFilter(new BaseFileFilter[]{txtFileFilter});
fileChooserView.showOpenChooserView(new FileChooserAction() {
    @Override
    public void onAccept(File selectedFile) {
        String content = null;
        if (selectedFile != null && selectedFile.isFile()) {
            String absPath = selectedFile.getAbsolutePath();
            String encoding = "UTF-8";
            File file = new File(absPath);
            long fileLength = file.length();
            byte[] fileContents = new byte[(int) fileLength];
            try (FileInputStream input = new FileInputStream(file)) {
                input.read(fileContents);
            } catch (IOException exception) {
                exception.printStackTrace();
            }
            try {
                content = new String(fileContents, encoding);
            } catch (UnsupportedEncodingException exception) {
                System.err.println("The OS does not support " + encoding
    );
                exception.printStackTrace();
            }
        }
    }    @Override
    public void onCancel() {
        FileChooserAction.super.onCancel();
    }
});
```

页面展示效果如**图** 3-16所示：

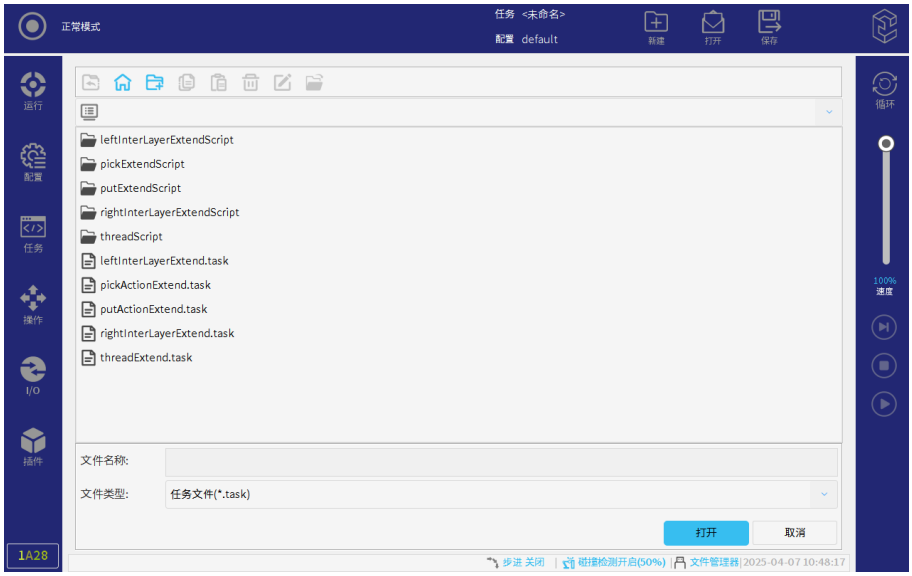


图 3-16: 文件选择器页面效果图

3.5.5 键盘组件

ELITECO SDK API 向开发人员提供了文本、数字、IP 键盘组件。开发人员可以根据实际需要使用不同的键盘组件进行输入。开发人员可以通过 SwingService.keyboardService 接口来创建面板进行使用。

创建一个文本输入键盘，其中第二个参数为校验的正则表达式，第三个参数为 true 时为密码类型，将会以 ● 的形式显示文本内容：

```
1    SwingService.keyboardService.showLetterKeyboard(tx1, null, false,
     new BaseKeyboardCallback() {
2        @Override
3        public void onOk(Object o) {
4
5        }
6    });
```

文本键盘页面展示效果图 3-17所示：

图 3-17: 文本键盘页面效果图

创建一个数字键盘，第二个参数初始显示数据，第三个参数为指定小数位数，超过之后将不能再输入第四个参数为是否是正数：

```
    SwingService.keyboardService.showNumberKeyboard(tx3, 0.0, 2, true,
new BaseKeyboardCallback() {
        @Override
        public void onOk(Object o) {

        }
    });
    还可以创建一个指定输入区间的数字键盘：
    SwingService.keyboardService.showNumberKeyboard(tx4, null, 100.0,
300.0, true, new BaseKeyboardCallback() {
        @Override
        public void onOk(Object o) {

        }
    });
```

数字键盘页面展示效果图 3-18所示：

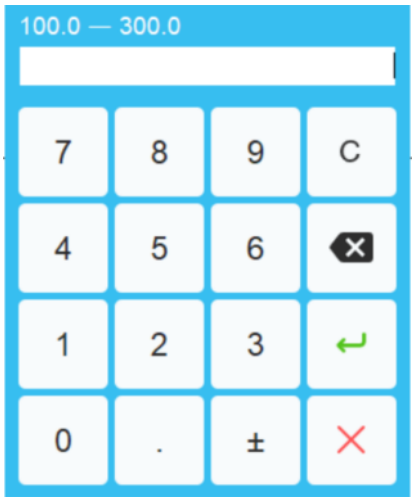


图 3-18: 数字键盘页面效果图

创建一个输入框类型的 IP 键盘：

```
    SwingService.keyboardService.showIpKeyboard(tx5, new
BaseKeyboardCallback() {
        @Override
        public void onOk(Object o) {

        }
    });
```

也可以直接创建一个 IP 面板:

```
1    JPanel tx6 = SwingService.ipPaneService.createIpPane(200, 35);
```

IP 键盘展示效果如**图** 3-19:

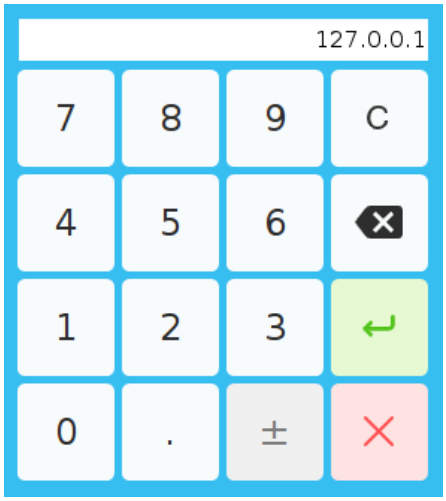


图 3-19: 面板页面效果图

具体使用可以参考 NavbarExtensionImpl.java 中的相关示例。

3.5.6 消息组件

为与 EliRobot 页面风格一致，ELITECO SDK API 提供了消息组件。开发人员可以通过 SwingService.message Service 接口来展示消息。提供 4 种消息类型（普通类型消息、成功类型消息、错误类型消息、警告类型消息）的样式。

展示普通提示消息:

```
1    SwingService.messageService.showMessage(change(resourceBundle.
   getString("message_title")), change(resourceBundle.getString("
   message-content")), MessageType.INFO);
```

普通消息页面效果如**图** 3-20:

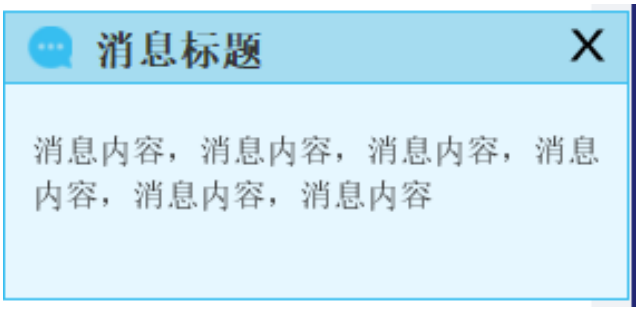


图 3-20: 普通消息页面效果图

展示成功提示消息：

```
1    SwingService.messageService.showMessage(change(resourceBundle.
   getString("message_title")), change(resourceBundle.getString("
   message-content")), MessageType.SUCCESS);
```

成功消息页面效果如图 3-21：

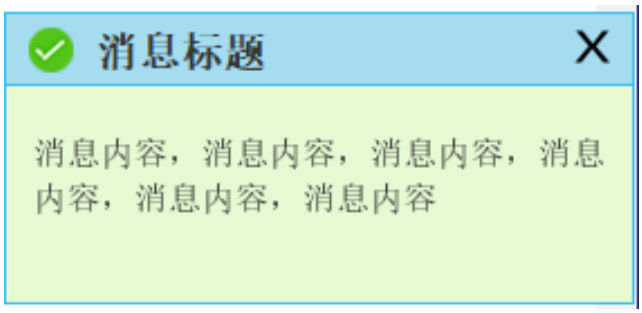


图 3-21: 成功消息页面效果图

展示错误提示消息：

```
1    SwingService.messageService.showMessage(change(resourceBundle.
   getString("message_title")), change(resourceBundle.getString("
   message-content")), MessageType.ERROR);
```

错误消息页面效果如图 3-22：

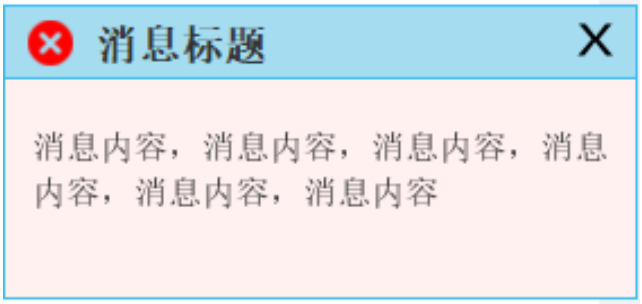


图 3-22: 错误消息页面效果图

展示警告提示消息：

```
1    SwingService.messageService.showMessage(change(resourceBundle.
   getString("message_title")), change(resourceBundle.getString("
   message-content")), MessageType.WARING);
```

警告消息页面效果如**图** 3-23：

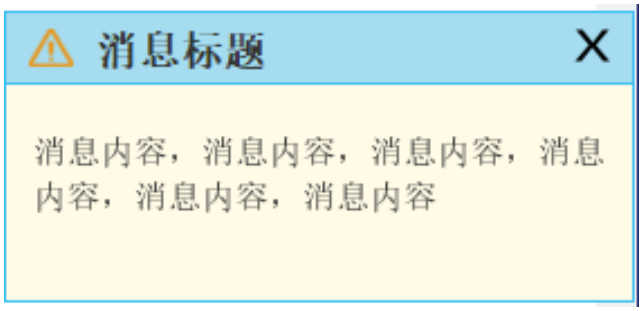


图 3-23: 警告消息页面效果图

具体使用可以参考 NavbarExtensionImpl.java 中的相关示例。

3.5.7 弹出框组件

为与 EliRobot 页面风格一致，ELITECO SDK API 提供了弹出框组件。开发人员可以通过 SwingService.dia logService 接口来创建弹出框进行使用。开发人员可以根据具体的业务流程来使用各个类型的弹出框，也可以自定义弹出框里面的内容和自定义按钮。弹出框遵循标准的 Swing 弹出框组件。

创建一个自定义弹出框面板内容和自定义按钮的弹出框：

```
1    Object[] options = new Object[] {"button1", "button2", "button3"};
2    JPanel panel1 = new JPanel(new BorderLayout());
3    panel1.setSize(new Dimension(500, 100));
4    JTextField textField = new JTextField("请输入姓名");
5    panel1.add(textField, BorderLayout.NORTH);
6    panel1.add(new JButton("Test"));
7    ("dialog_info_title")), DialogPaneModel.INFORMATION_MESSAGE,
    DialogPaneModel.YES_NO_OPTION, panel1, options, options[1]);
8    System.out.println(val);
```

页面展示效果如**图** 3-24：

图 3-24: 弹出框页面效果图

具体其他的弹出框使用可以参考 NavbarExtensionImpl.java 中的相关示例。

3.5.8 提示组件

为与 EliRobot 页面风格一致，ELITECO SDK API 提供了提示组件。开发人员可以通过 SwingService.tip Service 接口来展示提示消息。提供 4 种提示类型（普通提示、成功提示、错误提示、警告提示）的样式。可以根据组件在页面的位置将提示信息显示在组件的各个位置，分别为：上左、上中、上右、下左、下中、下右、左上、左中、左下、右上、右中、右下。

例如，创建一个输入框校验只能输入手机号，当输入错误的手机号提示手机号错误：

```
1    JLabel l1 = new JLabel(change(resourceBundle.getString("
     tip_tel_check")));
2    JTextField t1 = new JTextField(30);
3    String reg = "^[1](([3|5|8][\\d])|([4][4,5,6,7,8,9])|([6][2,5,6,7])
     |([7][^9])|([9][1,8,9]))
4    [\\d]{8}$";
5    t1.addMouseListener(new MouseAdapter() {
6        @Override
7        public void mouseClicked(MouseEvent e) {
8            SwingService.keyboardService.showLetterKeyboard(t1, null,
     false, new BaseKeyboardCa llback() {
9                @Override
10               public void onOk(Object o) {
11                   boolean flag = Pattern.compile(reg).matcher(o.
     toString()).matches();
12                   if (!flag) {
13                       SwingService.tipService.showTip(t1, change(
     resourceBundle.getString("ti p_tel_message")));
14                   }
15               }
16           });
17       }
18   });
```

页面展示效果如**图** 3-25：

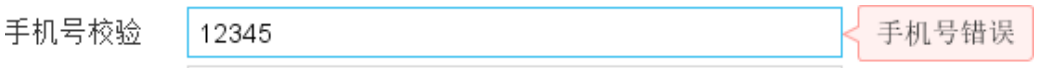


图 3-25: 弹提示信息页面效果图

其他具体使用可以参考 NavbarExtensionImpl.java 中的相关示例。

3.5.9 进度条组件

为与 EliRobot 页面风格一致，ELITECO SDK API 提供了进度条组件。开发人员可以通过 taskApiProvider.getComponentHubService 接口来展示进度条消息。提供 4 种进度条类型（未知进度条、已知进度条、未知进度取消条、已知进度取消条）的样式。

未知进度条：

```
1    Future<?> future = taskApiProvider.getComponentHubService().
     startProgress("未知进度条");
2    Runnable run = () -> {
3      WaitUtils.pause(5000, TimeUnit.MILLISECONDS);
4      taskApiProvider.getComponentHubService().stopProgress(future);
5    };
6    Thread thread = new Thread(run);
7    thread.start();
```

页面展示效果如**图** 3-26：

图 3-26: 未知进度效果图

已知进度条：

```
1    Future<?> future = taskApiProvider.getComponentHubService().
     startProgress("已知进度条", 100);
2    Runnable run = () -> {
3      for (int i = 0; i < 10; i++) {
4         WaitUtils.pause(500, TimeUnit.MILLISECONDS);
5         taskApiProvider.getComponentHubService().addProgress(10);
6      }
7      taskApiProvider.getComponentHubService().stopProgress(future);
8    };
9    Thread thread = new Thread(run);
10   thread.start();
```

页面展示效果如**图** 3-27：

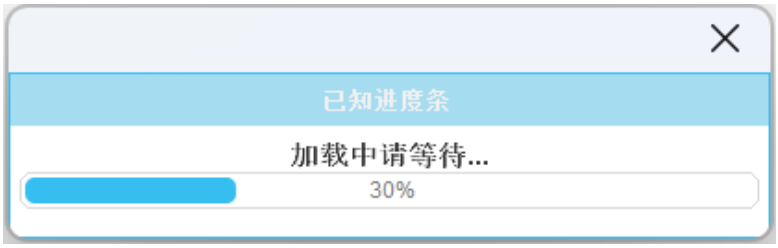


图 3-27: 已知进度效果图

未知进度取消条：

```
1     taskApiProvider.getComponentHubService().startProgress("未知进度取消
    条", action -> System.out.println("已取消未知进度条"));
2    });
```

页面展示效果如图 3-28：

图 3-28: 未知进度取消条效果图

已知进度取消条：

```
1    taskApiProvider.getComponentHubService().startProgress("已知进度取消条
    ", 100, action -> System.out.println("已取消已知进度条"));
2    Runnable run = () -> {
3      for (int i = 0; i < 10; i++) {
4        WaitUtils.pause(500, TimeUnit.MILLISECONDS);
5        taskApiProvider.getComponentHubService().addProgress(10);
6      }
7    });
```

页面展示效果如图 3-29：

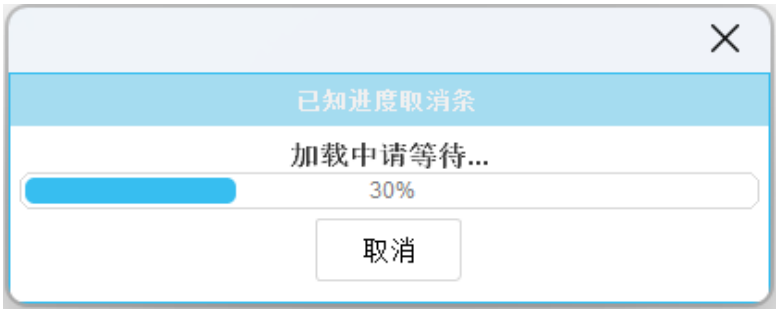


图 3-29: 已知进度取消条效果图

3.5.10 按钮样式

为与 EliRobot 页面风格一致，ELITECO SDK API 提供按钮组件。开发人员可以通过 SwingService.button UiService 接口来设置不同的按钮风格。提供的按钮风格：默认、主按钮、危险按钮、链接式按钮、前置图标按钮。创建一组默认风格按钮：

```
1     JLabel l1 = new JLabel(change(resourceBundle.getString("
    button_default")));
2     JButton b1 = new JButton("Default");
3     JButton b2 = new JButton("Default");
4     b2.setIcon(ImageHelper.loadImage("find.png"));
5     JButton b3 = new JButton();
6     b3.setIcon(ImageHelper.loadImage("find.png"));
7     JButton b4 = new JButton("Default");
8     b4.setIcon(ImageHelper.loadImage("find.png"));
9     b4.setEnabled(false);
```

页面展示效果如**图** 3-30：

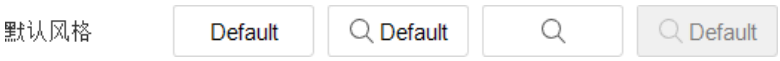


图 3-30: 普通按钮风格页面效果图

创建一组主风格按钮：

```
1     JLabel l2 = new JLabel(change(resourceBundle.getString("
    button_primary")));
2     JButton b5 = new JButton("Primary");
3     SwingService.buttonUiService.setPrimaryUi(b5);
4     JButton b6 = new JButton("Primary");
5     b6.setIcon(ImageHelper.loadImage("find_w.png"));
6     SwingService.buttonUiService.setPrimaryUi(b6);
```

```
    JButton b7 = new JButton();
    b7.setIcon(ImageHelper.loadImage("find_w.png"));
    SwingService.buttonUiService.setPrimaryUi(b7);
    JButton b8 = new JButton("Primary");
    b8.setIcon(ImageHelper.loadImage("find_w.png"));
    SwingService.buttonUiService.setPrimaryUi(b8);
    b8.setEnabled(false);
```

页面展示效果如**图** 3-31：

图 3-31: 主按钮风格页面效果图

创建一组危险风格按钮：

```
    JLabel l3 = new JLabel(change(resourceBundle.getString("
    button_danger")));
    JButton b9 = new JButton("Danger");
    SwingService.buttonUiService.setDangerUi(b9);
    JButton b10 = new JButton("Danger");
    b10.setIcon(ImageHelper.loadImage("find_w.png"));
    SwingService.buttonUiService.setDangerUi(b10);
    JButton b11 = new JButton();
    b11.setIcon(ImageHelper.loadImage("find_w.png"));
    SwingService.buttonUiService.setDangerUi(b11);
    JButton b12 = new JButton("Danger");
    b12.setIcon(ImageHelper.loadImage("find_w.png"));
    SwingService.buttonUiService.setDangerUi(b12);
    b12.setEnabled(false);
```

页面展示效果如**图** 3-32：

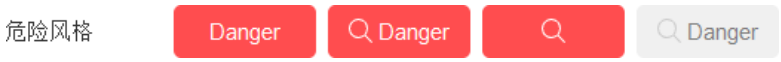


图 3-32: 危险按钮风格页面效果图

创建一个链接风格按钮：

```
    JLabel l4 = new JLabel(change(resourceBundle.getString("button_link
    ")));
    JButton b13 = new JButton("Link");
    SwingService.buttonUiService.setLinkUi(b13);
```

页面展示效果如**图** 3-33：

图 3-33: 链接按钮风格页面效果图

创建一组前置图标风格按钮：

```
1     JLabel l5 = new JLabel(change(resourceBundle.getString("button_pre")
      ));
2     JButton b14 = new JButton("Home");
3     SwingService.buttonUiService.setPreIconUi(b14, ImageHelper.loadImage
      ("home.png"));
4     JButton b15 = new JButton("HOme");
5     SwingService.buttonUiService.setPreIconUi(b15, ImageHelper.loadImage
      ("home.png"));
6     b15.setEnabled(false);
```

页面展示效果如**图** 3-34：

图 3-34: 前置图标按钮风格页面效果图

具体使用可以参考 NavbarExtensionImpl.java 中的相关示例。

3.5.11 字体样式

为与 EliRobot 机器人平台风格一致，ELITECO SDK API 机器人平台提供了不同的字体风格。开发人员可以通过 FontLibrary 类来设置不同的字体风格。可以根据不同需求选择不同级别的字体。

创建各级字体：

```
1     JLabel l1 = new JLabel("H1-一级标题");
2     l1.setFont(FontLibrary.H1_FONT);
3     JLabel l11 = new JLabel("H1-一级标题加粗");
4     l11.setFont(FontLibrary.H1_BOLD_FONT);
5
6     JLabel l2 = new JLabel("H2-二级标题");
7     l2.setFont(FontLibrary.H2_FONT);
```

```
    JLabel l21 = new JLabel("H2-二级标题加粗");
    l21.setFont(FontLibrary.H2_BOLD_FONT);

    JLabel l3 = new JLabel("H3-三级标题");
    l3.setFont(FontLibrary.H3_FONT);
    JLabel l31 = new JLabel("H3-三级标题加粗");
    l31.setFont(FontLibrary.H3_BOLD_FONT);

    JLabel l4 = new JLabel("H4-四级标题");
    l4.setFont(FontLibrary.H4_FONT);
    JLabel l41 = new JLabel("H4-四级标题加粗");
    l41.setFont(FontLibrary.H4_BOLD_FONT);

    JLabel l5 = new JLabel("H5-五级标题");
    l5.setFont(FontLibrary.H5_FONT);
    JLabel l51 = new JLabel("H5-五级标题加粗");
    l51.setFont(FontLibrary.H5_BOLD_FONT);

    JLabel l6 = new JLabel("默认-正文");
    l6.setFont(FontLibrary.NORMAL_FONT);
    JLabel l61 = new JLabel("默认-正文加粗");
    l61.setFont(FontLibrary.NORMAL_BOLD_FONT);

    JLabel l7 = new JLabel("小字体");
    l7.setFont(FontLibrary.SMALL_FONT);
    JLabel l71 = new JLabel("小字体加粗");
    l71.setFont(FontLibrary.SMALL_BOLD_FONT);
```

页面效果展示如**图** 3-35：

图 3-35: 字体风格页面效果图

具体使用可以参考 NavbarExtensionImpl.java 中的相关示例。

第 4 章 定制配置节点

ELITECO Plugin 支持定制配置节点: 定制配置节点及节点参数视图，并通过节点服务将其能够被使 EliRobot 能够正确加载该节点。通过定制配置节点，可以扩展 EliRobot 的功能，配置个性化的功能或设备参数。下面将通过一个简单的 demo-MyConfiguration 节点来说明配置节点的定制过程及相关特性的使用。

4.1 项目新建

项目新建过程、License、国际化及图标资源等相关配置详见第 3.1 节 ELITECO Plugin 项目新建，此处不再赘述。

配置节点定制项目实例-MyConfiguration 文件结构图如**图** 4-1所示：

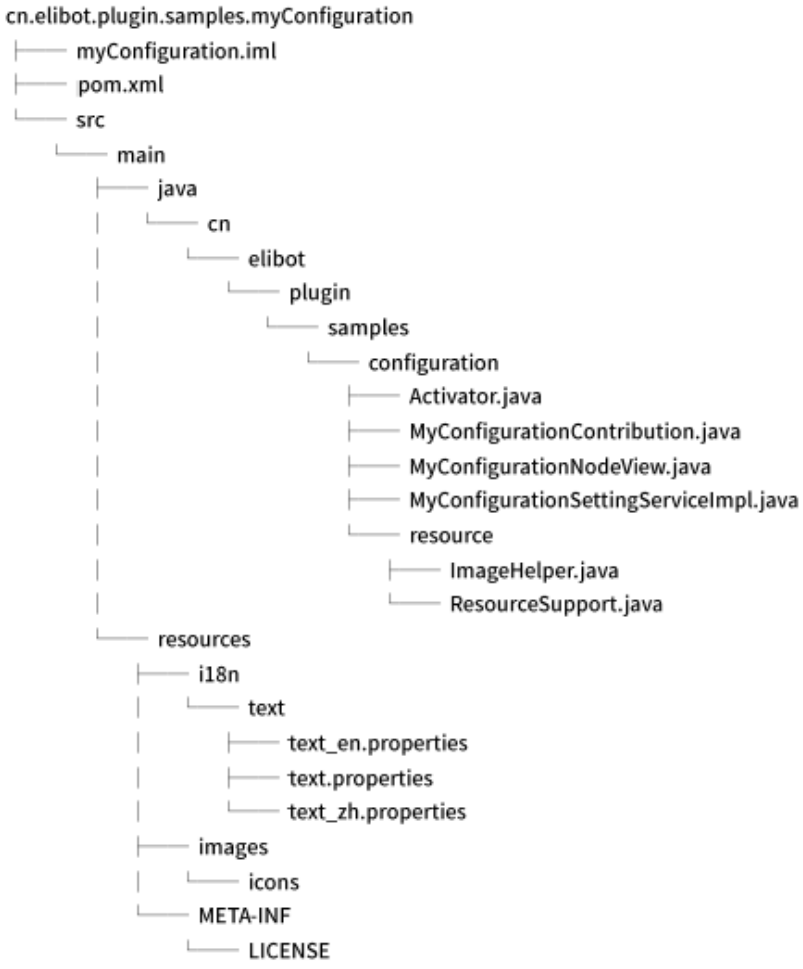


图 4-1: counter 项目文件结构

4.2 定制配置节点贡献

定制配置节点主要分三个步骤: 定制配置节点贡献、定制配置节点视图、定制配置节点服务。其中配置节点贡献主要定义功能实现；配置节点视图则是用于展示和交互参数的页面；而配置节点服务则定义了配置节点的公共属性、节点贡献及节点参数视图的创建等。

MyConfiguration 配置节点功能设计主要是用于展示配置节点定制流程，其内部功能主要是展示创建配置变量、设置变量名称、获取配置模块内部数据、数据存取等内部功能的使用，不实现具体的新功能。

下面章节将以 MyConfiguration 为例，详细说明配置节点定制流程及内部功能接口的使用。

定制配置节点贡献需要通过实现 ConfigurationNodeContribution 接口类。为便于描述，使用该接口的空实现类 MyConfigurationContribution.java 来描述定制过程，如下代码块4.1:

```
1   public class MyConfigurationContribution implements
    ConfigurationNodeContribution {
2       public MyConfigurationContribution(ConfigurationAPIProvider
    configurationApiProvider, MyConfigurationNodeView
    myConfigurationNodeView, DataModelWrapper dataModelWrapper) {
3       }
4
5       /**
6       * 配置节点视图 打开时激发(可用于视图打开时 功能或数据处理)
7       */
8       @Override
9       void onViewOpen(){
10          // to be implements
11      }
12
13      /**
14      * 配置节点视图 关闭时激发(可用于视图关闭时 功能或数据处理)
15      */
16      @Override
17      void onViewClose(){
18          // to be implements
19      }
20
21      /**
22      * (如果在任务中使用，且涉及生成脚本)生成脚本.
23      *
```

```
24          * scriptWriter 用于生成脚本的 脚本写入句柄
25          */
26          @Override
27          void generateScript(ScriptWriter scriptWriter){
28              // to be implements
29          }
30      }
```

代码块 4.1: 待实现的 MyConfigurationContribution.java

如上代码块4.1所示，一个 ConfigurationNodeContribution 接口的实现类，主要包括构造方法、配置节点视图打开事件方法、配置节点视图关闭事件方法及脚本生成方法。

MyConfigurationContribution 构造方法主要参数及描述如下**表** 4-1所示：

表 4-1. MyConfigurationContribution.java 构造方法参数

参数	描述
ConfigurationApiProvider configurationApiProvider	各种内部 api 的接口，可以使用配置模块独有的功能接口或获取的内部数据，如获取 I/O、获取变量、获取 TCP 及获取坐标系等 (具体参见相关该接口文档)
DataModelWrapper dataModelWrapper	节点数据持久化句柄，定制配置节点的数据可以通过该接口进行存取，需要注意的是通过该接口存储的数据会在保存/打开配置文件时序列化/加载 (具体参见 DataModelWrapper 接口文档)

onViewOpen/onViewClose 方法是配置节点视图事件方法，会在配置节点视图打开/关闭的时候激发，可以用于事件发生时数据处理，配置节点及配置节点视图同一类型都是单例的，并且一一对应。

generateScript 该方法用于保存/运行任务时生成写入必要数据到脚本文件前置数据位置，使得任务节点脚本中能够正常使用相关数据，而不会出现未定义问题，其参数及描述如下 **表** 5-4所示。

表 4-2. generateScript 方法参数

参数	描述
ScriptWriter scriptWriter	脚本生成句柄，提供写入脚本所需的各种 api

按照前边 MyConfiguration 配置节点功能设计所述，需要完成 MyConfigurationCon-

tribution.java 后的代码如以下代码块4.2所示：

```
 1    public class MyConfigurationContribution implements
     ConfigurationNodeContribution {
 2        private static final String CONFIG_VAR_VALUE_KEY = "
     config_ext_var";
 3        private static final String DEFAULT_VALUE = "Contribution
     extension script variable.";
 4        private final MyConfigurationNodeView view;
 5        private final ConfigurationAPIProvider configurationApiProvider;
 6
 7        private final DataModelWrapper model;
 8
 9        public MyConfigurationContribution(ConfigurationAPIProvider
     configurationApiProvider, MyConfigurationNodeView view,
     DataModelWrapper model) {
10            this.model = model;
11            this.view = view;
12            this.configurationApiProvider = configurationApiProvider;
13            model.registerConversionStrategy(
     TestPluginConversionStrategy.NODE_NAME, TestPluginConversionStrategy
     .class);
14        }
15
16        @Override
17        public void onViewOpen() {
18            System.out.println("enter MyConfigurationContribution");
19            view.setPopupText(getConfigVarValue());
20
21            System.out.println("Current all TCPs:");
22            configurationApiProvider.getConfigurationAPI().getTCPModel()
     .getTCPs().forEach(e -> {
23                System.out.println(e.getDisplayName());
24            });
25
26            System.out.println("Current all Frames:");
27            configurationApiProvider.getConfigurationAPI().getFrameModel
     ().getFrames().forEach(e -> {
28                System.out.println(e.getName());
29            });
30
31            configurationApiProvider.getConfigurationAPI().getIOModel().
     getIOs().forEach(io->{
32                if(io.getInterfaceType() == IO.InterfaceType.STANDARD &&
      io.getType() == IO.IOType.DIGITAL && !io.isInput()){
```

```
                    System.out.println("standard io " + io.getValueStr()
);
                }
            });

            System.out.println("test: "+this.getModel().getInteger("test
"));
            System.out.println("testDouble: "+this.getModel().getDouble
("testDouble"));
            System.out.println("test1: "+this.getModel().getPose("test1
"));
            System.out.println("test2: "+this.getModel().getBoolean("
test2"));
            System.out.println("test3: "+this.getModel().get("test3"));
        }

        @Override
        public void onViewClose() {
            System.out.println("exit MyConfigurationContribution");
            System.out.println("Current all TCPs:");
            configurationApiProvider.getConfigurationAPI().getTCPModel()
.getTCPs().forEach(e -> {
                System.out.println(e.getDisplayName());
            });
        }

        public DataModelWrapper getModel() {
            return this.model;
        }

        public boolean isDefined() {
            return !getConfigVarValue().isEmpty();
        }

        @Override
        public void generateScript(ScriptWriter writer) {
            if(!isDefined()){
                return;
            }
            writer.declareAndDefineGlobalVariable("config_ext_global_var
", "\"" + getConfigVarValue() + "\"");
        }

        public String getConfigVarValue() {
            String ret = model.getString(CONFIG_VAR_VALUE_KEY);
```

```
71              return ret == null ? DEFAULT_VALUE : ret;
72          }
73
74          public void setConfigVarValue(String value) {
75              if ("".equals(value)) {
76                  resetToDefaultValue();
77              } else {
78                  model.setString(CONFIG_VAR_VALUE_KEY, value);
79              }
80          }
81
82          private void resetToDefaultValue() {
83              view.setPopupText(DEFAULT_VALUE);
84              model.setString(CONFIG_VAR_VALUE_KEY, DEFAULT_VALUE);
85          }
86      }
```

代码块 4.2: 完全实现的 MyConfigurationContribution.java

由以上代码块4.2可见，MyConfiguration 作为一个展示配置节点定制流程的 demo，其实现功能较为简单，多是用来展示内部 api 的使用方法，如下所列：

参数视图打开及关闭事件方法：更新参数视图 UI 文本；以打印的方式展示 TCP、Frame、标准 I/O 及持久化数据获取方式；

- 脚本生成方法：在当前任务生成脚本时，MyConfiguration 将会直接在脚本中写入一个名称为”config_ext_global_var” 的全局变量的声明并初始化；
- 设置变量值方法：提供“setConfigVarValue”方法供参数视图设置更新”config_ext_global_var” 变量值。

4.3 定制配置节点参数视图

定制配置节点参数视图需要通过实现 SwingConfigurationNodeView 接口类。该定制类负责创建和管理参数视图的 UI 组件、构建参数视图、处理各种 UI 组件事件方法的实现等等(注意该定制类是参数视图的一个服务类，并不是参数视图本身)。为方便描述，使用该接口空的实现类 MyConfigurationNodeView.java 来描述定制过程，如以下代码块4.3所示:

```
1   public class MyConfigurationNodeView implements
    SwingConfigurationNodeView<MyConfigurationContribution> {
2
3       public MyConfigurationNodeView(Style style,
    ConfigurationViewAPIProvider configurationViewApiProvider) {
```

```
4         }
5
6         @Override
7         public void buildUI(JPanel jPanel, final
      MyConfigurationContribution configurationContribution) {
8
9         }
10    }
```

代码块 4.3: 待实现的 MyConfigurationNodeView.java

如以上代码块4.3所示，MyConfigurationNodeView.java，主要包括构造方法、buildUI这两个方法。

MyConfigurationNodeView 构造方法主要参数及描述如下**表** 5-4所示：

表 4-3. MyConfigurationNodeView.java 构造方法参数

参数	描述
Style style	UI 组件属性类，规定了节点参数视图的 UI 组件的属性数据
ConfigurationApiProvider configurationApiProvider	各种内部 api 的接口，可以使用配置模块独有的功能接口或获取的内部数据，如获取 I/O、获取变量、获取 TCP 及获取坐标系等 (具体参见相关该接口文档)

buildUI 方法则负责构建参数视图，有两个参数如下**表** 5-4所示：

表 4-4. buildUI 参数

参数	描述
JPanel jPanel	配置视图页面本身，承载 UI 组件
ConfigurationContribution configurationContribution	配置节点贡献，UI 组件可以从配置节点贡献中获取数据进行显示，也可以在 UI 组件的事件方法中通过配置节点贡献的接口进行数据更新。

按照前边 MyConfiguration 配置节点功能设计所述，需要完成 MyConfigurationNode-View.java 后的代码如以下代码块4.4所示：

```
 1    public class MyConfigurationNodeView implements
     SwingConfigurationNodeView<MyConfigurationContribution> {
 2        private final Style style;
 3        private final ConfigurationViewAPIProvider
     configurationViewApiProvider;
 4        private JTextField jTextField;
 5        private String properties = "i18n.text.text";
 6        private ResourceBundle resourceBundle;
 7        private Locale locale;
 8
 9        public MyConfigurationNodeView(Style style,
     ConfigurationViewAPIProvider configurationViewApiProvider) {
10            this.style = style;
11            this.configurationViewApiProvider =
     configurationViewApiProvider;
12            locale = ResourceSupport.getLocaleProvider().getLocale();
13            this.resourceBundle = ResourceBundle.getBundle(properties,
     locale != null ? locale: Locale.ENGLISH);
14        }
15
16        @Override
17        public void buildUI(JPanel jPanel, final
     MyConfigurationContribution configurationContribution) {
18            jPanel.setLayout(new BoxLayout(jPanel, BoxLayout.Y_AXIS));
19            jPanel.add(createInfo());
20            jPanel.add(createVerticalSpacing());
21            jPanel.add(createInput(configurationContribution));
22            jPanel.add(createVerticalSpacing());
23
24            JButton moveToTarget = new JButton(this.resourceBundle.
     getString("MoveToTarget"));
25            moveToTarget.setPreferredSize(new Dimension(160, 40));
26            moveToTarget.setMinimumSize(new Dimension(160, 40));
27            moveToTarget.setMaximumSize(new Dimension(160, 40));
28            moveToTarget.addActionListener(e -> {
29                double[] joints = new double[6];
30                joints[0] = 0;
31                joints[1] = -90;
32                joints[2] = 0;
33                joints[3] = -90;
34                joints[4] = 90;
35                joints[5] = 0;
36                JointPositions jointPositions = JointPositionsUtils.
     newJointPositions(joints, Angle.Unit.DEG);
37                configurationViewApiProvider.getConfigurationAPI().
```

```
    getRobotMovementService().requestUserToMoveRobot(jointPositions, new
     AutoMoveCallback() {
38                  @Override
39                  public void onComplete(AutoMoveCompleteEvent
    autoMoveCompleteEvent) {
40                      System.out.println("On continue");
41                      System.out.println("Is at target position: " +
    autoMoveCompleteEvent.isAtTargetPosition());
42                  }
43
44                  @Override
45                  public void onCancel(AutoMoveCancelEvent
    autoMoveCancelEvent) {
46                      System.out.println("On cancel");
47                      System.out.println("Is at target position: " +
    autoMoveCancelEvent.isAtTargetPosition());
48                  }
49              });
50          });
51          jPanel.add(moveToTarget);
52          jPanel.add(createVerticalSpacing());
53
54          JButton setRobotPosition = new JButton(this.resourceBundle.
    getString("SetRobotPosition"));
55          setRobotPosition.setPreferredSize(new Dimension(160, 40));
56          setRobotPosition.setMinimumSize(new Dimension(160, 40));
57          setRobotPosition.setMaximumSize(new Dimension(160, 40));
58          setRobotPosition.addActionListener(e ->
    configurationViewApiProvider.getConfigurationAPI().
    getRobotMovementService().requestUserToSetPosition(new
    RobotMovementService.ViewState(), new SetPositionCallback() {
59              @Override
60              public void onComplete(SetPositionCompleteEvent
    setPositionCompleteEvent) {
61                  System.out.println("Set robot position on complete
    .");
62              }
63          }));
64          jPanel.add(setRobotPosition);
65          jPanel.add(createVerticalSpacing());
66
67          JButton setDataModel = new JButton(this.resourceBundle.
    getString("SetData"));
68          setDataModel.setPreferredSize(new Dimension(160, 40));
69          setDataModel.setMinimumSize(new Dimension(160, 40));
```

```
70              setDataModel.setMaximumSize(new Dimension(160, 40));
71              setDataModel.addActionListener(e -> {
72                  configurationContribution.getModel().setInteger("test",
    123);
73                  configurationContribution.getModel().setDouble("
    testDouble", 123.21);
74                  configurationContribution.getModel().setPose("test1",
    PoseUtils.newPose(0.1, 1.1, 1, 1, 2.3, 5, Length.Unit.MM, Angle.Unit
    .RAD));
75                  configurationContribution.getModel().setBoolean("test2",
     true);
76                  configurationContribution.getModel().set("test3", new
    TestPluginConversionStrategy());
77              });
78              jPanel.add(setDataModel);
79          }
80
81          private Box createInfo() {
82              Box infoBox = Box.createVerticalBox();
83              infoBox.setAlignmentX(Component.LEFT_ALIGNMENT);
84              JTextPane pane = new JTextPane();
85              pane.setBorder(BorderFactory.createEmptyBorder());
86              SimpleAttributeSet attributeSet = new SimpleAttributeSet();
87              StyleConstants.setLineSpacing(attributeSet, 0.5f);
88              StyleConstants.setLeftIndent(attributeSet, 0f);
89              pane.setParagraphAttributes(attributeSet, false);
90              pane.setText(this.resourceBundle.getString("Describe"));
91              pane.setEditable(false);
92              pane.setMaximumSize(pane.getPreferredSize());
93              pane.setBackground(infoBox.getBackground());
94              infoBox.add(pane);
95              return infoBox;
96          }
97
98          private Box createInput(final MyConfigurationContribution
    myConfigurationContribution) {
99              Box inputBox = Box.createHorizontalBox();
100             inputBox.setAlignmentX(Component.LEFT_ALIGNMENT);
101
102             inputBox.add(new JLabel(this.resourceBundle.getString("
    ConfigurationVariable")));
103             inputBox.add(createHorizontalSpacing());
104
105             jTextField = new JTextField();
106             jTextField.setFocusable(false);
```

```
107             jTextField.setPreferredSize(style.getInputfieldSize());
108             jTextField.setMaximumSize(jTextField.getPreferredSize());
109             jTextField.addMouseListener(new MouseAdapter() {
110                 @Override
111                 public void mouseReleased(MouseEvent e) {
112                     SwingService.keyboardService.showLetterKeyboard(
    jTextField.getText(), "", new BaseKeyboardCallback() {
113                         @Override
114                         public void onOk(Object value) {
115                             jTextField.setText((String) value);
116                             myConfigurationContribution.
    setConfigVarValue((String) value);
117                         }
118                     });
119                 }
120             });
121             inputBox.add(jTextField);
122 
123             return inputBox;
124         }
125 
126         private Component createHorizontalSpacing() {
127             return Box.createRigidArea(new Dimension(style.
    getHorizontalSpacing(), 0));
128         }
129 
130         private Component createVerticalSpacing() {
131             return Box.createRigidArea(new Dimension(0, style.
    getVerticalSpacing()));
132         }
133 
134         public void setPopupText(String t) {
135             jTextField.setText(t);
136         }
137     }
```

代码块 4.4: 完全实现的 MyConfigurationNodeView.java

由以上代码块4.4可见，MyConfigurationNodeView.java 通过 buildUI 方法，在参数 jpanel 上布局若干 Swing UI 组件完成指定的功能或展示 api 的使用，主要如下：

- jTextFiel 配置变量赋值输入框。在代码块4.2中配置节点贡献中 generateScript 方法中在任务脚本中写入了一个名为”config_ext_global_var” 的全局变量，jTextField 输入的字符串被用作为该变量赋初值；
- moveToTarget 移动到目标位置按键。在其按键事件中通过 configurationViewApiProvider

提供的 api requestUserToMoveRobot(JointPositions, AutoMoveCallback) 弹出移动到目标位置页面，并移动到指定的 JointPositions 目标位置，并且在 AutoMoveCallback 中定制”onComplete”、”onCancel” 及”onError” 回调函数；

- setRobotPosition 设置机器人位置按键。在其按键事件中通过 configurationViewApiProvide 提供的 api requestUserToSetPosition(ViewState, SetPositionCallback) 弹出设置机器人位置页面，并使用指定的 ViewState 为设置机器人位置页面设置初始状态，并且在 SetPositionCallback 中定制”onComplete”、”onCancel” 回调函数；
- setDataModel 设置数据按键。在其按键事件中通过配置节点贡献 configurationContribution 中的数据模型 model 保存数据来演示配置节点贡献数据保存。

基于以上所述，配置节点贡献 configurationContribution 的参数视图如下**图** 4-2所示：

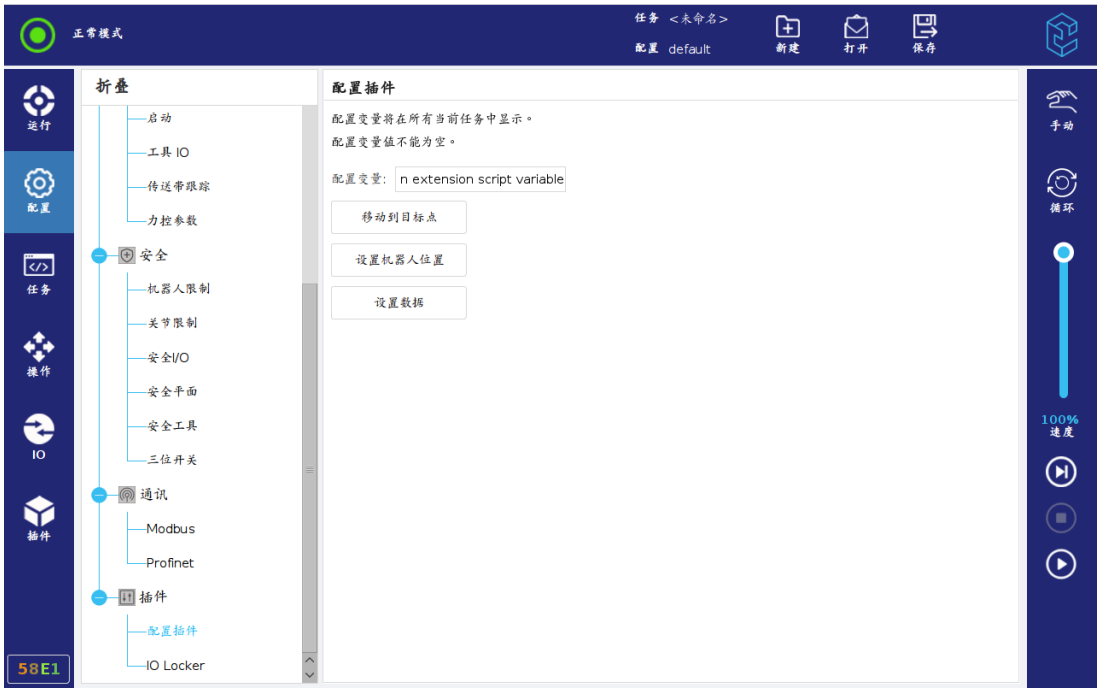


图 4-2: MyConfiguration 配置节点视图

4.4 定制配置节点服务

定制配置节点服务需要通过实现 SwingConfigurationNodeService 接口类。如前文所述，该定制类定义了配置节点贡献的公共属性、节点贡献及节点参数视图的创建等特征。为方便描述，使用该接口空的实现类 MyConfigurationSettingServiceImpl.java 来描述定制过程，如以下代码块4.5所示:

```
1   public class MyConfigurationSettingServiceImpl implements
    SwingConfigurationNodeService<MyConfigurationContribution,
    MyConfigurationNodeView> {
```

```
2
3         @Override
4         public void configureContribution(ContributionConfiguration
      contributionConfiguration) {
5         }
6
7         @Override
8         public String getTitle(Locale locale) {
9         }
10
11        @Override
12        public MyConfigurationNodeView createView(
      ConfigurationViewAPIProvider viewApiProvider) {
13        }
14
15        @Override
16        public MyConfigurationContribution createConfigurationNode(
      ConfigurationAPIProvider configurationApiProvider,
17                MyConfigurationNodeView myConfigurationContributionView,
18                DataModelWrapper context) {
19
20        }
21    }
```

代码块 4.5: 待实现的 MyConfigurationNodeView.java

由以上代码块4.5可见，SwingConfigurationNodeService 的实现类主要有 4 个方法：configureContribution、getTitle、createView 及 createConfigurationNode。

configureContribution 方法用于配置节点贡献，需注意的是，当前该方法没有实际作用，留作后用。

getTitle 为参数视图页面提供多语标题，有一个参数如下**表** 4-5所示：

表 4-5. getTitle 参数

参数	描述
Locale locale	指定的语言环境

createView 方法是用于创建参数视图的定制类，有一个参数如下**表** 4-6所示：

表 4-6. createView 参数

参数	描述
ConfigurationViewAPIProvider viewAPIProvider	前文有所提及，为参数视图定制类提供各种 api 接口

createConfigurationNode 方法是用于创建配置节点贡献类。有三个参数如下**表** 4-7所示：

表 4-7. createConfigurationNode 参数

参数	描述
ConfigurationAPIProvider configurationAPIProvider	前文有所提及，为配置节点贡献类提供各种 api 接口
MyConfigurationNodeView myConfigurationNodeView	为 4.3、定制配置节点参数视图中所述的参数视图定制类
DataModelWrapper context	则是为配置节点贡献类提供数据存储及持久化服务的数据模型

按照前边 MyConfiguration 配置节点功能设计所述需要完成 MyConfigurationSettingServiceImpl.java 后的的代码如以下代码块4.6所示：

```
1   public class MyConfigurationSettingServiceImpl implements
    SwingConfigurationNodeService<MyConfigurationContribution,
    MyConfigurationNodeView> {
2
3       @Override
4       public void configureContribution(ContributionConfiguration
    contributionConfiguration) {
5       }
6
7       @Override
8       public String getTitle(Locale locale) {
9           String title = "Configuration Plugin";
10          return ResourceSupport.getResourceBundle(locale).getString(
    title);
11      }
12
13      @Override
14      public MyConfigurationNodeView createView(
    ConfigurationViewAPIProvider viewApiProvider) {
```

```
15              return new MyConfigurationNodeView(new Style(),
    viewApiProvider);
16          }
17
18          @Override
19          public MyConfigurationContribution createConfigurationNode(
    ConfigurationAPIProvider configurationApiProvider,
20                  MyConfigurationNodeView myConfigurationContributionView,
21                  DataModelWrapper context) {
22              return new MyConfigurationContribution(
    configurationApiProvider, myConfigurationContributionView, context);
23          }
24      }
```

代码块 4.6: 完全实现的 MyConfigurationNodeView.java

基于前文所述，MyConfigurationSettingServiceImpl 的实现比较简单，不再过多描述。

最后，将定制完成的 MyConfigurationSettingServiceImpl 类，在 Activator 中注册，如以下代码块4.7中第 14 行。

```
1       public class Activator implements BundleActivator {
2           private ServiceReference<LocaleProvider>
    localeProviderServiceReference;
3           private ServiceRegistration<SwingConfigurationNodeService>
    registration;
4
5           @Override
6           public void start(BundleContext context) {
7               localeProviderServiceReference = context.getServiceReference
    (LocaleProvider.class);
8               if (localeProviderServiceReference != null) {
9                   LocaleProvider localeProvider = context.getService(
    localeProviderServiceReference);
10                  if (localeProvider != null) {
11                      ResourceSupport.setLocaleProvider(localeProvider);
12                  }
13              }
14              this.registration = context.registerService(
    SwingConfigurationNodeService.class, new
    MyConfigurationSettingServiceImpl(), null);
15          }
16
17          @Override
18          public void stop(BundleContext context) {
```

```
19              this.registration.unregister();
20          }
21      }
```

代码块 4.7: 完全实现的 MyConfigurationNodeView.java

完成注册后，即可按照第 3 章 ELITECO Plugin 项目前期中的构建部署流程，将其安装到 EliRobot 机器人平台下。

第 5 章 定制任务节点

ELITECO Plugin 支持开发者定制各种功能的任务节点，包括自定义全新的任务节点或封装若干内部节点形成功能模板等。这种特性使得开发者可以快速开发定制出符合个性化的任务节点，来满足多样化的需求。本章将主要描述任务节点定制流程。

同定制配置节点类似，定制任务节点主要分为三个步骤：定制节点贡献、定制节点参数视图、定制节点服务。在这个过程中通过使用 ELITECO SDK 中开放的相关接口，来实现任务节点的定制。下面将通过一个简单的 demo-计数器节点来说明任务节点的定制过程及相关特性的使用。

5.1 项目新建

项目新建过程、License、国际化及图标资源等相关配置详见第 3.1 节 ELITECO Plugin 项目新建部分，此处不再赘述。

本章将着重以计数器节点项目为例，说明任务节点的定制过程及相关特性的使用。任务节点定制项目实例-计数器节点文件结构图如下**图** 5-1所示：

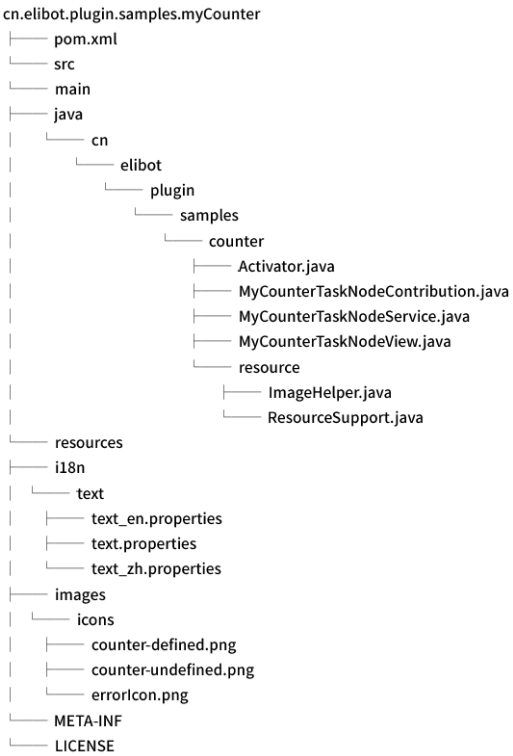


图 5-1: myCounter 项目文件结构

5.2 定制任务节点贡献

节点贡献是被任务节点包含，并规定任务节点个体特性及管理个体数据的” 贡献类”，通过实现 ELITECO SDK API 中 TaskNodeContribution 接口完成。

由 TaskNodeContribution 贡献的任务节点将在任务-指令栏-插件下展示。展示时，会根据插件的厂商信息（VendorName）和包标识（bundle symbolic）对节点进行归档与排序。同一厂商的节点将归为一组并按包名字母顺序排列；未提供有效厂商信息的节点将默认显示在最下方，按照包名排序。

计数器节点功能设计为：创建任务变量、并任选变量作为计算变量用来计算若干连续节点的运行次数；展示定制任务节点贡献文件类中创建及操作内部任务节点方法。

为便于描述，使用如以下代码块5.1该接口的空实现类 MyCounterTaskNodeContribution.java 来描述定制过程。

```
1   public class MyCounterTaskNodeContribution implements
    TaskNodeContribution {
2       public MyCounterTaskNodeContribution(TaskApiProvider
    taskApiProvider, TaskNodeDataModelWrapper taskNodeDataModelWrapper)
    {
3           // to be implements
4       }
5
6       /**
7        * 定制节点参数视图打开事件方法
8        */
9       @Override
10      public void onViewOpen() {
11          // to be implements
12      }
13
14      /**
15       * 定制节点参数视图关闭事件方法
16       */
17      @Override
18      public void onViewClose() {
19          // to be implements
20      }
21
22      /**
23       * 获取参数视图多语标题
24       */
```

```
25          @Override
26          public String getTitle() {
27              // to be implements
28          }
29
30          /**
31           * 获取贡献节点图标
32           */
33          @Override
34          public ImageIcon getIcon(boolean isUndefined) {
35              // to be implements
36          }
37
38          /**
39           * 获取定制节点在任务树上的显示文本
40           * 建议：建议使用参数给定的locale去获取对应语言的资源bundle，否则会出现树节点文本语言始终跟随系统而无法实现英文编程(即程序树及指令列表英文显示，不跟随系统)
41           */
42          @Override
43          public String getDisplayOnTree(Locale locale) {
44              // to be implements
45          }
46
47          /**
48           * 获取节点定义状态，节点数据定否定义完善(该方法与其子节点状态公共决定任务树对应节点状态，黄色则为未定义完善，反之则完善)
49           */
50          @Override
51          public boolean isDefined() {
52              // to be implements
53          }
54
55          /**
56           * 生成脚本
57           */
58          @Override
59          public void generateScript(ScriptWriter scriptWriter) {
60              // to be implements
61          }
62
63          /**
64           * 传递定制节点参数视图
65           */
66          @Override
67          public void setTaskNodeContributionViewProvider(
```

```
    TaskExtensionNodeViewProvider taskExtensionNodeViewProvider) {
68              // to be implements
69          }
70      }
```

代码块 5.1: 待实现的 MyCounterTaskNodeContribution.java

如以上代码块5.1所示，一个 TaskNodeContribution 接口的实现类，主要包括构造方法、任务节点参数页面打开/关闭事件方法、任务节点参数页面标题 getter 方法、任务节点图标 getter 方法、任务节点任务树上显示文本 getter 方法、任务节点定义状态 getter 方法、脚本生成方法及参数视图更新方法。

MyCounterTaskNodeContribution 构造方法主要参数及描述如下**表** 5-1所示：

表 5-1. MyCounterTaskNodeContribution.java 构造方法参数

参数	描述
TaskApiProvider taskApiProvider	各种内部 api 的接口，可以使用任务模块独有的功能接口或获取的内部数据，如创建变量、获取变量、获取 TCP 及获取坐标系等 (具体参见相关该接口文档)
TaskNodeDataModelWrapper TaskNodeDataModelWrapper	节点数据持久化句柄，定制任务节点的数据可以通过该接口进行存取，需要注意的是通过该接口存储的数据会在保存/打开任务时序列化/加载 (具体参见 TaskNodeDataModelWrapper 接口文档)

onViewOpen/onViewClose 方法是任务节点参数视图事件方法，会在任务节点参数视图打开/关闭的时候激发，可以用于事件发生时数据处理，需注意的是，不同于配置节点同一类型只有一个节点实例，并且与对应参数视图一一对应，任务节点通常可以在任务树上添加多个实例，因此同一类型的多个任务节点实例共用一个对应参数视图实例，因此建议在 onViewOpen 方法中至少有将当前任务节点实例更新到参数视图的逻辑。

getTitle 方法用于获取任务节点参数视图的标题，按照第 3.1 节 ELITECO Plugin 项目新建部分中的国际化文本示例代码返回当前默认 Locale 的标题文本。

getIcon 方法用于获取当前定制任务节点的节点图标，其参数及描述如下**表** 5-2所示。通常根据节点关键数据定义状态分为已定义和未定义状态的两种状态的图标。当任务树上有节点处于未定义状态，表示当前任务节点关键数据未定义完全，不允许运行，并使用不同于定义状态的图标来提示操作人员该节点需要完善数据设置。

为保持统一，建议图标样式与内部节点保持统一，图标遵循以下原则：

- 图标有效部分着色，无效部分不着色；

- 已定义状态图标着色部分建议使用 #37BEFF(R:55，G:190, B:255)；
- 未定义状态图标着色部分建议使用 #FFC800(R:255, G:200, B:0)。

表 5-2. getIcon 方法参数

参数	描述
boolean isUndefined	节点关键数据定义状态

getDisplayOnTree 方法用于获取定制任务节点在任务树上的显示文本，其参数及描述如下表 (**表** 5-5) 所示。需注意的是为支持国际化编程，该文本不能使用默认 Locale 的文本，而是需要给定参数 Locale 对应的文本。

表 5-3. getDisplayOnTree 方法参数

参数	描述
Locale locale	指定的 Locale

isDefined 该方法用于获取定制任务节点关键数据定义状态，当前任务树数据有改变时，都会重新绘任务树及节点，因此在任务树交互过程中，会被多次调用。

generateScript 该方法用于保存/运行任务时生成任务树对应的脚本过程中，写入定制任务节点相应的脚本行，其参数及描述如下**表** 5-4所示。

表 5-4. generateScript 方法参数

参数	描述
ScriptWriter scriptWriter	脚本生成句柄，提供写入脚本所需的各种 api

setTaskNodeContributionViewProvider 传递定制节点对应的参数视图句柄给定制节点贡献。建议接收参数视图实例作为定制节点贡献的属性，可用于在 openView() 方法中将当前节点贡献传递给参数视图用于更新数据，其参数及描述如下**表** 5-5所示。

表 5-5. getDisplayOnTree 方法参数

参数	描述
TaskExtensionNodeViewProvider provider	定制节点对应的参数视图实例 provider

基于以上 MyCounterTaskNodeContribution.java 主要方法的描述，按照计数器节点功能设计实现并完善 MyCounterTaskNodeContribution.java，其代码如下（代码块5.2）所示:

```
1   public class MyCounterTaskNodeContribution implements
    TaskNodeContribution, ContributionInsertedListener,
    ContributionRemovedListener, ContributionLoadCompletedListener {
2       private final ExpressionService expressionService;
3       private TaskApiProvider taskApiProvider;
4       private MyCounterTaskNodeView view;
5       private TaskNodeDataModelWrapper taskNodeDataModelWrapper;
6       private final VariableService variableService;
7       private static final String SELECTED_VAR = "selected_var";
8       private ResourceBundle resourceBundle;
9       private String properties = "i18n.text.text";
10
11      public MyCounterTaskNodeContribution(TaskApiProvider
    taskApiProvider, TaskNodeDataModelWrapper taskNodeDataModelWrapper)
    {
12          this.taskApiProvider = taskApiProvider;
13          Locale locale = ResourceSupport.getLocaleProvider().
    getLocale();
14          this.resourceBundle = ResourceBundle.getBundle(properties,
    locale != null ? locale: Locale.ENGLISH);
15          this.taskNodeDataModelWrapper = taskNodeDataModelWrapper;
16          this.variableService = this.taskApiProvider.
    getVariableService();
17          this.expressionService = this.taskApiProvider.
    getExpressionService();
18          TreeNode treeNode = this.taskApiProvider.getTaskModel().
    getContributionTreeNode(this);
19          this.taskApiProvider.getUndoRedoManager().recordChanges(() -
    > {
20              MyCounterTaskNodeContribution.this.addFolderNode(
    treeNode);
21          });
22      }
23
24      @Override
25      public String getDisplayOnTree(Locale locale) {
26          Variable variable = getSelectedVariable();
27          if (variable == null) {
28              return getSpecificSourceBundle(locale).getString("
    Counter") + " : " + getSpecificSourceBundle(locale).getString("
    NullVariable");
29          } else {
30              return getSpecificSourceBundle(locale).getString("
    Counter") + " : " + variable.getDisplayName();
31          }
```

```
32          }
33
34          @Override
35          public void onViewOpen() {
36              this.view.updateView(this);
37          }
38
39          @Override
40          public void onViewClose() {
41          }
42
43          @Override
44          public String getTitle() {
45              return this.resourceBundle.getString("Counter");
46          }
47
48          @Override
49          public ImageIcon getIcon(boolean isUndefined) {
50              if (!isUndefined) {
51                  return ImageHelper.loadImage("counter-defined.png");
52              } else {
53                  return ImageHelper.loadImage("counter-undefined.png");
54              }
55          }
56
57          @Override
58          public boolean isDefined() {
59              return getSelectedVariable() != null;
60          }
61
62          @Override
63          public void generateScript(ScriptWriter scriptWriter) {
64              Variable variable = getSelectedVariable();
65              if(variable != null) {
66                  String resolvedVariableName = scriptWriter.
    getResolvedVariableName(variable);
67                  scriptWriter.appendLine(resolvedVariableName + " = " +
    resolvedVariableName + " + 1");
68                  scriptWriter.writeChildren();
69              }
70          }
71
72          @Override
73          public void setTaskNodeContributionViewProvider(
    TaskExtensionNodeViewProvider taskExtensionNodeViewProvider) {
```

```
74              this.view = (CounterTaskNodeView)
    taskExtensionNodeViewProvider.get();
75          }
76
77          public Variable getSelectedVariable() {
78              return taskNodeDataModelWrapper.getVariable(SELECTED_VAR);
79          }
80
81          public TaskVariable createGlobalVariable(String variableName) {
82              TaskVariable variable = null;
83              try{
84                  variable = this.variableService.getVariableFactory().
    createTaskVariable(variableName, this.expressionService.
    createExpressionConstructor().appendTokenCell("0", "0").construct())
    ;
85              } catch (VariableException | IllegalExpressionException e) {
86                  SwingService.messageService.showMessage("Error", e.
    getMessage(), MessageType.ERROR);
87              }
88
89              return variable;
90          }
91
92          public void setVariable(final Variable variable) {
93              this.taskApiProvider.getUndoRedoManager().recordChanges(() -
    > taskNodeDataModelWrapper.setVariable(SELECTED_VAR, variable));
94              Frame frame = this.taskApiProvider.getFrameModel().
    getBaseFrame();
95              try {
96                  Expression expression = this.taskApiProvider.
    getExpressionService().createExpressionConstructor().appendFrame(
    frame).appendGreaterOrEqualSymbol().appendCharCell('1').construct();
97                  taskNodeDataModelWrapper.setExpression("exp", expression
    );
98              }catch (Exception e) {
99                  e.printStackTrace();
100             }
101         }
102
103         public void removeVariable() {
104             this.taskApiProvider.getUndoRedoManager().recordChanges(() -
    > taskNodeDataModelWrapper.remove(SELECTED_VAR));
105         }
106
107         public Collection<Variable> getGlobalVariables() {
```

```
108             return variableService.get(variable -> variable.getType().
    equals(Variable.Type.TASK) || variable.getType().equals(Variable.
    Type.CONFIGURATION));
109         }
110
111         @Override
112         public void onInserted(ContributionInsertedContext
    insertedContext) {
113             System.out.println("Counter is inserted");
114         }
115
116         @Override
117         public void loadComplete(ContributionLoadCompleteContext
    completeContext) {
118             System.out.println("Counter has been loaded in task tree
    model");
119         }
120
121         @Override
122         public void onRemoved(ContributionRemovedContext
    contributionRemovedContext) {
123             System.out.println("Counter is removed");
124         }
125
126         private ResourceBundle getSpecificSourceBundle(Locale locale) {
127             return ResourceBundle.getBundle("i18n.text.text", locale !=
    null ? locale: Locale.ENGLISH);
128         }
129
130
131         protected void addFolderNode(TreeNode treeNode) {
132             TreeNode currentTreeNode = treeNode;
133             TaskNodeFactory factory = this.taskApiProvider.getTaskModel
    ().getTaskNodeFactory();
134             FolderNode folderNode = factory.createFolderNode();
135             folderNode.setDisplay("Hello world!");
136             try {
137                 currentTreeNode.addChild(folderNode);
138                 List<TreeNode> children = currentTreeNode.getChildren();
139                 for (TreeNode treeNode1 : children) {
140                     if (treeNode1.getTaskNode().equals(folderNode)) {
141                         addCommentNode(treeNode1);
142                         addHaltNode(treeNode1);
143                         addPopupNode(treeNode1);
144                         addCommentNode(treeNode1);
```

```
                    }
                }
            } catch (TreeStructureException treeStructureException) {
                treeStructureException.printStackTrace();
            }
        }

        private void addPopupNode(TreeNode treeNode) {
            TreeNode currentTreeNode = treeNode;
            TaskNodeFactory factory = this.taskApiProvider.getTaskModel
    ().getTaskNodeFactory();
            PopupNode popupNode = factory.createPopupNode();
            popupNode.setMessage("Hello world!");
            popupNode.setMessageDialogType(MessageType.INFO);
            try {
                currentTreeNode.addChild(popupNode);
            } catch (TreeStructureException treeStructureException) {
                treeStructureException.printStackTrace();
            }
        }

        private void addHaltNode(TreeNode treeNode) {
            TreeNode currentTreeNode = treeNode;
            TaskNodeFactory factory = this.taskApiProvider.getTaskModel
    ().getTaskNodeFactory();
            HaltNode haltNode = factory.createHaltNode();
            try {
                currentTreeNode.addChild(haltNode);
            } catch (TreeStructureException treeStructureException) {
                treeStructureException.printStackTrace();
            }
        }

        private void addCommentNode(TreeNode treeNode) {
            TreeNode currentTreeNode = treeNode;
            TaskNodeFactory factory = this.taskApiProvider.getTaskModel
    ().getTaskNodeFactory();
            CommentNode commentNode = factory.createCommentNode();
            commentNode.setCommentString("Hello world!");
            try {
                currentTreeNode.addChild(commentNode);
            } catch (TreeStructureException treeStructureException) {
                treeStructureException.printStackTrace();
            }
        }
```

```
187     }
```

代码块 5.2: 完全实现的 MyCounterTaskNodeContribution.java

由以上代码块5.2可见，除代码块5.1中已有的接口外，MyCounterTaskNodeContribution.java 中新加了一部分接口。主要如下：

- onInserted/onRemoved/loadComplete: 这三个方法是 ELITECO SDK API 中的用于任务模块的事件方法，分别表示定制任务节点插入到任务树上/定制任务节点从任务树上移除/新任务文件已加载完成，分别来源于 MyCounterTaskNodeContribution 类实现 ContributionInsertedListener / ContributionRemovedListener / ContributionLoadCompletedListener 接口，通过监听这些事件，可以在对应的事件方法中作相关数据处理；
- addFolderNode/addPopupNode/addHaltNode/addCommentNod: 这些方法主要被 MyCounterTaskNodeContribution 类的构造方法调用，用于展示如何初始化定制任务节点子节点序列；
- getSpecificSourceBundle: 用于获取给定 Locale 的资源 Bundle，用于 getDisplayOnTree 方法中获取给定 Locale 的定制节点在任务树上的显示文本。

如上所示，一个计数器节点贡献类已经定制完毕，此处有所简略，主要为说明 TaskNodeContribution 接口的作用以及实现，后面将有专门的章节说明任务节点定制过程中使用 ELITECO Plugin 提供的其他的接口来定制功能更加复杂的节点。

5.3 定制任务节点参数视图

定制任务节点参数视图需要通过实现 SwingTaskNodeView 接口类。该定制类负责创建和管理参数视图的 UI 组件、构建参数视图、处理各种 UI 组件事件方法的实现等等 (注意该定制类是参数视图的一个服务类，并不是参数视图本身)。为方便描述，使用该接口空的实现类 MyCounterTaskNodeView.java 来描述定制过程，如下代码块5.3所示:

```
1   public class MyCounterTaskNodeView implements SwingTaskNodeView<
    CounterTaskNodeContribution> {
2   public MyCounterTaskNodeView(TaskNodeViewApiProvider
    taskNodeViewApiProvider) {
3   // to be implements
4   }
5
6   @Override
7   public void buildUI(JPanel jPanel, MyCounterTaskNodeContribution
    taskNodeContribution) {
```

```
8       // to be implements
9       }
10      }
```

代码块 5.3: 待实现的 MyCounterTaskNodeView.java

如上代码块5.3所示，MyCounterTaskNodeView.java，主要包括构造方法、buildUI 这两个方法。MyCounterTaskNodeView 构造方法主要参数及描述如下**表** 5-6所示：

表 5-6. MyCounterTaskNodeView.java 构造方法参数

参数	描述
TaskNodeViewApiProvider taskNodeViewApiProvider	各种内部 api 的接口，可以使用任务模块独有的功能接口、获取的内部数据以及，如创建变量、获取变量、获取 TCP 及获取坐标系等 (具体参见相关该接口文档)

buildUI 方法则负责构建参数视图，有两个参数如下**表** 6-2所示：

表 5-7. buildUI 参数

参数	描述
JPanel jPanel	用于承载 UI 组件，是参数视图的主体部分，定制任务节点的参数交互主要在该页面上完成。参数视图标题将由 EliRobot 机器人平台任务模块自动设置，标题内容由定制的配置节点服务中 getTitle(Locale) 方法中定制，参见 getTitle 方法描述
MyCounterTaskNodeContribution taskNodeContribution	任务节点贡献，UI 组件可以从任务节点贡献中获取数据进行显示，也可以在 UI 组件的事件方法中通过任务节点贡献的接口进行数据更新。

按照前边所述 (计数器节点功能设计) 需要完成 MyCounterTaskNodeView.java 后的代码如代码块5.4所示：

```
1   public class MyCounterTaskNodeView implements SwingTaskNodeView<
    CounterTaskNodeContribution> {
2       private JTextField textNewVariable;
3       private final JButton btnNewVariable = new JButton();
```

```
4         private final JComboBox<Object> cmbVariables = new JComboBox<>()
      ;
5         private CounterTaskNodeContribution contribution;
6         private final JLabel errorLabel = new JLabel();
7         private final ImageIcon errorIcon;
8         private final TaskNodeViewApiProvider taskNodeViewApiProvider;
9         private final TaskApiProvider taskApiProvider;
10        private String properties = "i18n.text.text";
11        private ResourceBundle resourceBundle;
12        private Locale locale;
13
14        public MyCounterTaskNodeView(TaskNodeViewApiProvider
      taskNodeViewApiProvider) {
15            this.taskNodeViewApiProvider = taskNodeViewApiProvider;
16            this.taskApiProvider = this.taskNodeViewApiProvider.
      getTaskApiProvider();
17            this.errorIcon = getErrorIcon();
18        }
19
20        private ImageIcon getErrorIcon() {
21            return ImageHelper.loadImage("errorIcon.png");
22        }
23
24        @Override
25        public void buildUI(JPanel jPanel, MyCounterTaskNodeContribution
       taskNodeContribution) {
26            locale = ResourceSupport.getLocaleProvider().getLocale();
27            this.resourceBundle = ResourceBundle.getBundle(properties,
      locale != null ? locale: Locale.ENGLISH);
28
29            this.contribution = taskNodeContribution;
30            jPanel.setLayout(new BoxLayout(jPanel, BoxLayout.Y_AXIS));
31
32            jPanel.add(createInfo(resourceBundle.getString("
      ContributionDescribe")));
33            jPanel.add(createVerticalSpacing());
34            jPanel.add(createComboBox(taskNodeContribution));
35            jPanel.add(createInfo(resourceBundle.getString("
      CreateNewButtonDescribe")));
36            jPanel.add(createButtonBox(taskNodeContribution));
37            jPanel.add(Box. createRigidArea(new Dimension(0, 500));
38        }
39
40        private Box createInfo(String info) {
41            Box infoBox = Box.createHorizontalBox();
```

```
42              infoBox.setAlignmentX(Component.LEFT_ALIGNMENT);
43              JLabel label = new JLabel();
44              label.setText(info);
45              label.setSize(label.getPreferredSize());
46              infoBox.add(label);
47              return infoBox;
48          }
49
50          private Component createVerticalSpacing() {
51              return Box.createRigidArea(new Dimension(0, 10));
52          }
53
54          private Component createHorizontalSpacing() {
55              return Box.createRigidArea(new Dimension(10, 0));
56          }
57
58          private Box createComboBox(CounterTaskNodeContribution
    taskNodeContribution) {
59              Box inputBox = Box.createHorizontalBox();
60              inputBox.setAlignmentX(Component.LEFT_ALIGNMENT);
61
62              cmbVariables.setFocusable(false);
63              cmbVariables.setPreferredSize(new Dimension(250, 30));
64              cmbVariables.addItemListener(e -> {
65                  if(e.getStateChange() == ItemEvent.SELECTED) {
66                      if(e.getItem() instanceof Variable) {
67                          Variable variable = (Variable) e.getItem();
68                          if(!variable.equals(CounterTaskNodeView.this.
    contribution.getSelectedVariable())) {
69                              MyCounterTaskNodeView.this.contribution.
    setVariable((Variable) e.getItem());
70                          }
71                      }else {
72                          MyCounterTaskNodeView.this.contribution.
    removeVariable();
73                      }
74                  }
75              });
76
77              JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
78              panel.add(cmbVariables, BorderLayout.CENTER);
79
80              inputBox.add(panel);
81              return inputBox;
82          }
```

```
83
84          private Box createButtonBox(MyCounterTaskNodeContribution
    taskNodeContribution) {
85              Box horizontalBox = Box.createHorizontalBox();
86              horizontalBox.setAlignmentX(Component.LEFT_ALIGNMENT);
87
88              textNewVariable = new JTextField();
89              textNewVariable.setFocusable(false);
90              textNewVariable.setPreferredSize(new Dimension(200,30));
91              textNewVariable.setMaximumSize(textNewVariable.
    getPreferredSize());
92              textNewVariable.addMouseListener(new MouseAdapter() {
93                  @Override
94                  public void mouseReleased(MouseEvent e) {
95                      SwingService.keyboardService.showLetterKeyboard(
    textNewVariable, "", false, new BaseKeyboardCallback() {
96                          @Override
97                          public void onOk(Object value) {
98                              MyCounterTaskNodeView.this.setNewVariable((
    String) value);
99                          }
100                     });
101                 }
102             });
103
104             horizontalBox.add(textNewVariable);
105             horizontalBox.add(createHorizontalSpacing());
106
107             btnNewVariable.setPreferredSize(new Dimension(120,30));
108             btnNewVariable.setText(resourceBundle.getString("CreateNew")
    );
109             btnNewVariable.addActionListener(e -> {
110                 TaskVariable variable = MyCounterTaskNodeView.this.
    contribution.createGlobalVariable(textNewVariable.getText());
111                 if (variable != null) {
112                     MyCounterTaskNodeView.this.contribution.setVariable(
    variable);
113                     updateView(MyCounterTaskNodeView.this.contribution);
114                 }
115             });
116
117             horizontalBox.add(btnNewVariable);
118             return horizontalBox;
119         }
120
```

```
121         private Box createErrorLabel() {
122             Box infoBox = Box.createHorizontalBox();
123             infoBox.setAlignmentX(Component.LEFT_ALIGNMENT);
124             errorLabel.setVisible(false);
125             infoBox.add(errorLabel);
126             return infoBox;
127         }
128 
129         public void updateView(MyCounterTaskNodeContribution
    taskNodeContribution) {
130             this.contribution = taskNodeContribution;
131             update(taskNodeContribution);
132         }
133 
134         public void setNewVariable(String name) {
135             textNewVariable.setText(name);
136         }
137 
138         private void clearInputVariableName() {
139             textNewVariable.setText("");
140         }
141 
142         private void updateComboBox(MyCounterTaskNodeContribution
    contribution) {
143             List<Object> items = new ArrayList<>();
144             items.addAll(contribution.getGlobalVariables());
145             Collections.sort(items, (o1, o2) -> {
146                 if (o1.toString().toLowerCase().compareTo(o2.toString().
    toLowerCase()) == 0) {
147                     return o1.toString().compareTo(o2.toString());
148                 } else {
149                     return o1.toString().toLowerCase().compareTo(o2.
    toString().toLowerCase());
150                 }
151             });
152 
153             items.add(0, resourceBundle.getString("SelectVariable"));
154 
155             cmbVariables.setModel(new DefaultComboBoxModel<>(items.
    toArray()));
156 
157             Variable selectedVar = contribution.getSelectedVariable();
158             if (selectedVar != null && !cmbVariables.getSelectedItem().
    equals(selectedVar)) {
159                 cmbVariables.setSelectedItem(selectedVar);
```

```
160             }
161         }
162
163         public void update(MyCounterTaskNodeContribution contribution) {
164             clearInputVariableName();
165             updateComboBox(contribution);
166         }
167     }
```

代码块 5.4: 完全实现的 MyCounterTaskNodeView.java

由上代码块5.4可见，MyCounterTaskNodeView.java 通过 buildUI 方法，在参数 jpanel 上布局若干 Swing UI 组件完成指定的功能或展示 api 的使用，主要如下：

- cmbVariables 变量下拉框，用于展示当前任务树上的任务变量及配置模块中的配置变量，并用作选择计数器变量，选定计数器变量后，在任务运行过程中每次执行该定制节点都会对选定的计数器变量进行 +1 操作；
- textNewVariable 变量名称输入框。如需要重新创建任务变量作为计数器变量，则可输入合法名称，点击后方的“新建”按键，即可创建任务变量用作计数器变量；
- btnNewVariable 新建变量按键。

基于以上所述，计数器节点参数视图如下**图** 5-2所示:

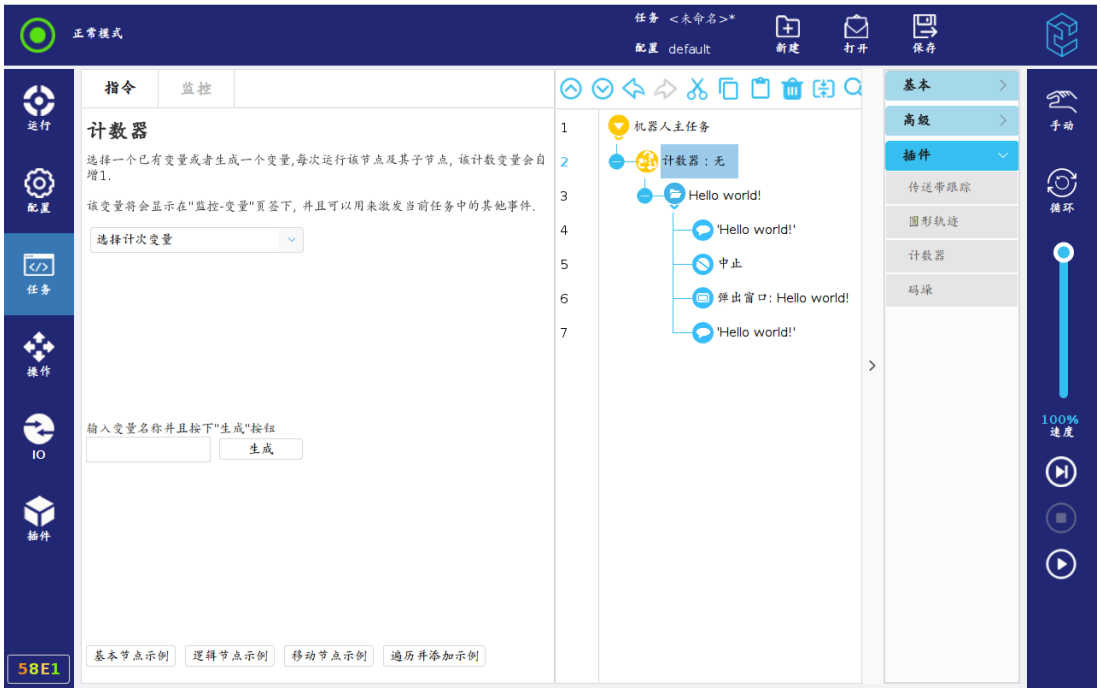


图 5-2: 计数器节点参数视图

由**图** 5-2可见，计数器节点参数视图下方有若干个示例节点创建按键，用于展示并测试内部节点创建及插入任务树上方法，代码较长，与本节主要讲述计数器节点参数视图构建关

联度较低，因此代码块5.4有所缩略，后续章节会详细讲解任务节点定制过程中，如何通过ELITECO SDK API 接口创建及操作内部任务节点。

5.4 定制任务节点服务

节点贡献和参数视图共同完成了任务节点特性定义、数据管理和参数页面的创建及管理，而任务节点服务则负责完成任务节点的公共特性定义和系统特性，如定制任务节点在任务树上的公共交互特性、节点贡献生成、参数视图生成以及 ID 等。

定制任务节点服务需要通过实现 SwingTaskNodeService<N extends TaskNodeContribution, extends SwingTaskNodeView<N» 接口类，为方便描述，使用该接口空的实现类 MyCounterTaskNodeService.java 来描述定制过程，如代码块5.5所示：

```
1   public class MyCounterTaskNodeService implements
    SwingTaskNodeService<MyCounterTaskNodeContribution,
    MyCounterTaskNodeView> {
2       @Override
3       public String getId() {
4           // to be implements
5       }
6
7       @Override
8       public String getTypeName(Locale locale) {
9           // to be implements
10      }
11
12      @Override
13      public void configureContribution(TaskNodeFeatures
    taskNodeFeatures) {
14          // to be implements
15      }
16
17      @Override
18      public MyCounterTaskNodeView createView(TaskNodeViewApiProvider
    taskNodeViewApiProvider) {
19          // to be implements
20      }
21
22      @Override
23      public MyCounterTaskNodeContribution createNode(TaskApiProvider
    taskApiProvider, TaskNodeDataModelWrapper taskNodeDataModelWrapper,
    boolean isCloningOrLoading) {
```

```
24              // to be implements
25          }
26      }
```

代码块 5.5: 待实现的 MyCounterTaskNodeService.java

如上代码块5.5所示，MyCounterTaskNodeView.java，主要包括 5 个方法：getId、getTypeName、configureContribution、createView 和 createNode。

getId 方法用于获取定制节点的专属 id，用作 EliRobot 机器人平台任务页签下指令栏中的定制任务节点按顺序显示。

getTypeName 方法用于获取定制节点的国际化类型名称，用作 EliRobot 机器人平台中该定制节点操作过程中上下文中标识该节点，有一个参数如下**表** 5-8所示：

表 5-8. getTypeName 参数

参数	描述
Locale locale	当前默认 Locale

configureContribution 方法用于配置定制任务节点的任务树上特征属性，有一个参数如下 **表** 5-9所示：

表 5-9. configureContribution 参数

参数	描述
TaskNodeFeatures configuration	提供了若干设置定制任务节点的任务树上特征属性的接口，具体参见 TaskNodeFeatures 接口类

createView 创建参数视图的定制类实例，需注意的是该方法仅在首个对应定制任务节点被创建时创建，且同一类型的定制任务节点共用一个参数视图的定制类实例。TaskNodeViewApiProvider 参数前文有所提及，此处不再赘述。

createNode 创建任务节点贡献类实例，两个参数在前文有所提及，此处不再赘述。

按照前边所述 (计数器节点功能设计) 需要完成 MyCounterTaskNodeService.java 后的代码如下代码块5.6所示:

```
1   public class MyCounterTaskNodeService implements
    SwingTaskNodeService<MyCounterTaskNodeContribution,
    MyCounterTaskNodeView> {
2       @Override
```

```
3         public String getId() {
4             return "Counter";
5         }
6
7         @Override
8         public String getTypeName(Locale locale) {
9             ResourceBundle resourceBundle = ResourceBundle.getBundle("
    i18n.text.text", locale != null ? locale: Locale.ENGLISH);
10            return resourceBundle.getString("Counter");
11        }
12
13        @Override
14        public void configureContribution(TaskNodeFeatures
    taskNodeFeatures) {
15            taskNodeFeatures.setChildrenAllowed(true);
16            taskNodeFeatures.setDeprecated(false);
17            taskNodeFeatures.setUserInsertable(true);
18            taskNodeFeatures.setPlaceHolderRequired(false);
19        }
20
21        @Override
22        public MyCounterTaskNodeView createView(TaskNodeViewApiProvider
    taskNodeViewApiProvider) {
23            return new MyCounterTaskNodeView(taskNodeViewApiProvider);
24        }
25
26        @Override
27        public MyCounterTaskNodeContribution createNode(TaskApiProvider
    taskApiProvider, TaskNodeDataModelWrapper taskNodeDataModelWrapper,
    boolean isCloningOrLoading) {
28            return new MyCounterTaskNodeContribution(taskApiProvider,
    taskNodeDataModelWrapper);
29        }
30    }
```

代码块 5.6: 完全实现的 MyCounterTaskNodeService.java

基于前文所述，MyCounterTaskNodeService.java 的实现比较简单，不再过多描述。

最后，将定制完成的 MyCounterTaskNodeService 类，在 Activator 中注册，如下代码块7.7代码中第 14 行。

```
1     public class Activator implements BundleActivator {
2     private ServiceReference<LocaleProvider>
    localeProviderServiceReference;
```

```
3
4     @Override
5     public void start(BundleContext bundleContext) throws Exception {
6     localeProviderServiceReference = bundleContext.getServiceReference(
      LocaleProvider.class);
7     if (localeProviderServiceReference != null) {
8     LocaleProvider localeProvider = bundleContext.getService(
      localeProviderServiceReference);
9     if (localeProvider != null) {
10    ResourceSupport.setLocaleProvider(localeProvider);
11    }
12    }
13    System.out.println("cn.elibot.plugin.samples.counter.Activator says
      Hello World!");
14    bundleContext.registerService(SwingTaskNodeService.class, new
      MyCounterTaskNodeService(), null);
15    }
16
17    @Override
18    public void stop(BundleContext bundleContext) throws Exception {
19    System.out.println("cn.elibot.plugin.example.counter.Activator says
      Goodbye World!");
20    }
21    }
```

代码块 5.7: Activator 中注册 MyCounterTaskNodeService 类

完成注册后，即可按照第 3 章见上方中的构建部署流程，将其安装到 EliRobot 机器人平台下。

5.5 任务模块其他功能特性

在本章第 5.2 节 定制任务节点贡献-第 5.4 节 定制任务节点服务 过程中，涉及到多种服务特性，如内部任务节点创建、内部任务节点插入删除、任务树及任务树上节点访问、操作撤消重做、任务节点事件、数据持久化、脚本生成以及配置模块数据访问等，这些服务特性对于定制高级任务节点必不可少。接下来的将对仅限于任务模块涉及的服务特性进行详细描述，其他的特性将在第 8 章 其他功能特性中进行描述。

5.5.1 任务节点创建

EliRobot 中内置了较多的具有基础功能的节点如移动、路点、线程、子任务、计时器等等，因此在节点贡献中创建这些内部节点并进行一定的组合就可以自定义更加复杂能够完成更加具体的应用场景工作任务的节点。

本小节通过 EliRobot 机器人平台开放的接口创建选择两个具有代表性的节点:Move 节点和计数器定制任务节点，来讲解任务节点创建流程，由于每个节点的开放的接口都较多，因此这里只着重讲解节点创建流程，至于其他节点的创建流程都比较类似，示例代码具体可以参考附录 1 内部任务节点创建及配置示例。

1. Move 节点

Move 有三种运动类型 MoveJ(Move Joint 关节运动)、MoveL(Move Linear) 直线运动、MoveP(Move Process 过程运动)，三者之间具体应用场景这里不做赘述，EliRobot 机器人平台说明书中会有相关描述，这里只涉及与内部节点创建相关的交互特性。

Move 节点规定了其后代节点中的运动类节点的一些公共参数，任务脚本运行时 Move 节点本身并不是实际的运行指令，因此通常 Move 节点的后代节点中都需要有路点、方向等这类实际的运行指令，而 Move 作为一个运动类型及公共参数的管理节点存在。对于 MoveJ、MoveL 和 MoveP 这三种类型的 Move 节点对后代的实际运行节点的要求有些许差异。因此在介绍 Move 节点创建前有必要将其描述清楚。

MoveJ 是关节运动，因此其后代节点需要有一个或多个路点 WaypointNode 去执行实际运动动作，这些路点的运动参数可在通过继承的方式继承 MoveJ 的运动参数，也可以进行自定义，但需注意的是路点继承运动参数只会继承其祖先节点中与其距离最近的祖辈 Move 节点的参数，因此如下代码块5.8所示创建了包含一个路点的 MoveJ 节点:

```
1    public MoveNode createMoveNode() {
2            // 从TaskApiProvider中获取任务节点工厂类，并创建一个包含了一个路点的Move节点
3            TaskNodeFactory factory = this.taskApiProvider.getTaskModel().getTaskNodeFactory();
4            MoveNode moveWaypointNodeTemplate = factory.createMoveWaypointNodeTemplate();
5
6            // 获取计量类数据工厂，类似的速度、加速度、长度等机器人领域内有单位的计量类数据都可以通过该工厂类进行创建
7            ValueFactory valueFactory = this.taskApiProvider.getValueFactory();
8
```

```
9           // 创建出的move节点参数数据都是默认的，如需要设置参数，则需要如本行代码所示，通过Move节点获取其(参数)配置构建工厂类创建符合需要的配置构造器，如此处创建了一个MoveJ类型的配置构造器
10          MoveJointConfigBuilder moveWaypointTempConfigBuilder = moveWaypointNodeTemplate.getConfigBuilderFactory().createMoveJConfigBuilder();
11
12          // 配置构造器允许开发者通过开放的接口进行参数设置，如本行所示设置关节运动速度为 80 deg/S，同时该接口会有数据合法性验证，如输入不在合法范围内，那么实际设置的数据是最接近目标值的合法数据
13          moveWaypointTempConfigBuilder.setJointSpeed(valueFactory.createAngularSpeed(80.0D, AngularSpeed.Unit.DEG_S));
14          moveWaypointTempConfigBuilder.setJointAcceleration(valueFactory.createAngularAcceleration(1200.0D, AngularAcceleration.Unit.DEG_S2));
15          moveWaypointTempConfigBuilder.setTCPSelection(moveWaypointNodeTemplate.getTCPSelectionFactory().createIgnoreActiveTCPSelection());
16
17          // 设置参数的配置构造器，通过build方法即可构建出一个对应类型的参数配置数据，直接设置给相应节点，用于设置数据，如本行所示，构建出了一个MoveJ类型的参数配置数据，作为参数用于move节点设置数据
18          moveWaypointNodeTemplate.setConfig(moveWaypointTempConfigBuilder.build());
19          return moveWaypointNodeTemplate;
20      }
```

代码块 5.8: 包含一个路点的 MoveJ 节点创建

如上所示创建了一个包含一个路点的 MoveJ 节点，并演示了其参数设置。当然用户也可以通过创建 Move 节点和 Waypoint 节点按照规则进行组合，涉及到节点插入删除将在第 5.5.2 小节 节点插入删除中进行描述。

MoveL 是直线运动，其后代节点执行实际运动动作的节点可以路点 WaypointNode 或方向节点 DirectionNode，创建包含路点的 MoveL 类似于代码块5.8中所示的流程，不同在于创建的 MoveL 类型的配置构造器，同时设置的参数是直线运动相关的参数。此处不再单独介绍，重点介绍方向节点作为实际运动节点的情况，如下代码块5.9所示为创建包含一个方向节点的 MoveL 节点的流程：

```
1   public MoveNode createMoveNode() {
2           // 同Code:4-4-1，获取计量类数据工厂和节点工厂
3           ValueFactory valueFactory = this.taskApiProvider.getValueFactory();
```

```
4           TaskNodeFactory factory = this.taskApiProvider.getTaskModel
    ().getTaskNodeFactory();
5
6           // 创建包含方向节点的Move节点模板
7           MoveNode moveDirectionUntilNodeTemplate = factory.
    createMoveDirectionUntilNodeTemplate();
8           // 创建MoveL(参数)配置构造器
9           MoveLinerConfigBuilder moveLConfigBuilder =
    moveDirectionUntilNodeTemplate.getConfigBuilderFactory().
    createMoveLConfigBuilder();
10          // 设置MoveL参数
11          moveLConfigBuilder.setToolSpeed(valueFactory.createSpeed
    (80.0D, Speed.Unit.MM_S));
12          moveLConfigBuilder.setToolAcceleration(valueFactory.
    createAcceleration(15000, Acceleration.Unit.MM_S2));
13          moveLConfigBuilder.setTCPSelection(
    moveDirectionUntilNodeTemplate.getTCPSelectionFactory().
    createIgnoreActiveTCPSelection());
14
15          // 构造MoveL参数配置并作为参数用来更新MoveL节点配置
16          moveDirectionUntilNodeTemplate.setConfig(moveLConfigBuilder.
    build());
17          return moveDirectionUntilNodeTemplate;
18      }
```

代码块 5.9: 包含一个方向节点的 MoveL 节点创建

如上所示其创建过程类似于代码块5.8中的移动路点模板的创建过程。

MoveP 是过程运动，其特征是运动过程中末端的工具速度均匀，以保证在某些工艺场景下工艺加工效果，MoveP 后代节点中执行实际运动动作的节点可以路点 WaypointNode、方向节点 DirectionNode 或包含两个路点的圆弧运动节点，创建包含路点 WaypointNode 或方向节点 DirectionNode 的方式类似代码块5.8与代码块5.9中的流程，如下代码块5.10所示包含圆弧运动节点作为子节点的 MoveP 节点的创建流程:

```
1       public MoveNode createMoveNode() {
2           // 同Code:4-4-1，获取计量类数据工厂和节点工厂
3           ValueFactory valueFactory = CounterTaskNodeView.this.
    taskApiProvider.getValueFactory();
4           TaskNodeFactory factory = CounterTaskNodeView.this.
    taskApiProvider.getTaskModel().getTaskNodeFactory();
5
6           // 创建包含圆弧运动节点的Move节点模板
7           MoveNode moveArcMotionTemplate = factory.
    createMoveArcMotionTemplate();
```

```
8           MoveProcessConfigBuilder moveArcMotionTempConfigBuilder =
    moveArcMotionTemplate.getConfigBuilderFactory().
    createMovePConfigBuilder();
9           // 设置MoveP参数
10          moveArcMotionTempConfigBuilder.setToolSpeed(valueFactory.
    createSpeed(80.0D, Speed.Unit.MM_S));
11          moveArcMotionTempConfigBuilder.setToolAcceleration(
    valueFactory.createAcceleration(15000, Acceleration.Unit.MM_S2));
12          moveArcMotionTempConfigBuilder.setTCPSelection(
    moveArcMotionTemplate.getTCPSelectionFactory().
    createIgnoreActiveTCPSelection());
13          moveArcMotionTempConfigBuilder.setTransitionRadius(
    valueFactory.createLength(50.0D, Length.Unit.MM));
14          // 构造MoveP参数配置并作为参数用来更新MoveP节点配置
15          moveArcMotionTemplate.setConfig(
    moveArcMotionTempConfigBuilder.build());
16          return moveArcMotionTemplate;
17      }
```

代码块 5.10: 包含圆弧运动节点子节点的 MoveP 节点创建

2. 计时器节点

定制任务节点贡献类或参数视图类中，也可以调用 ELITECO Plugin 提供的接口创建该项目下其他定制任务节点，并操作其插入到当前定制任务节点实例的子节点序列或任务树上其他位置。如下代码块5.11所示为在同一项目下其他定制任务节点创建计时器节点：

```
1       TaskExtensionNode createCounterNode() {
2           TaskNodeFactory factory = this.taskApiProvider.getTaskModel
    ().getTaskNodeFactory();
3           TaskExtensionNode counterTaskNode = factory.
    createExtensionNode(CounterTaskNodeService.class);
4           return counterTaskNode;
5       }
```

代码块 5.11: 其他定制任务节点创建计时器节点

5.5.2 节点插入删除

ELITECO SDK API 提供了允许用户进行节点插入删除的接口，可以方便地定制功能复杂的模板任务节点。

从前几章介绍可以知道节点贡献只是规定了该节点的业务逻辑，其在任务树上的交互特性相关接口并没在节点贡献中定义，而是定义在 TreeNode 类接口中，通过给定的方法，可以获取与节点贡献实例一一对应 TreeNode 示例，需注意的是 TreeNode 实例与节点贡献一一对应，可以将 TreeNode 类理解为任务树上实际节点的一个临时代理，通过 TreeNode 实例可以进行任务树节点子节点的添加、插入、删除等操作，由于是临时代理，因此每次通过给定的方法获取到的与指定节点贡献实例对应的 TreeNode 实例都是重新创建的，因此即使是同一个节点贡献两次获得的 TreeNode 实例也不是相同的一个，因此 TreeNode 实例不能拿来判等。

如下代码块5.12所示为通过指定的节点贡献实例获取对应的 TreeNode 实例方法：

```
1    TreeNode treeNode = this.taskApiProvider.getTaskModel().
   getContributionTreeNode(this);
```

代码块 5.12: 通过定制任务节点贡献实例获取对应的 TreeNode 实例

从上面可以看出，通过节点贡献或节点参数视图中的 TaskApiProvider 实例可以获取 TaskModel 实例，通过 TaskModel 提供的接口来获取节点贡献对应的 TreeNode 实例。TaskModel 类一共有两个接口，除了此处获取节点贡献对应实例 getContributionTreeNode 这一接口外，其另外一个接口在第 5.5.1 小节 任务节点创建 中有所涉及，在第 5.5.1 小节 任务节点创建中通过 TaskModel 获取节点工厂类。

下面代码块5.13对 TreeNode 类中提供的接口进行介绍：

```
1    public interface TreeNode {
2        /**
3         * 添加 newChild 到当前TreeNode 代理任务节点子节点序列的末尾. 若
   newChild为空，当前节点为空或newChild已经在任务树上则抛出异常
4         * <br><br>
5         *
6         * 参数 newChild  为要添加到当前TreeNode 代理任务节点下的子节点实例
7         * 返回 newChild 对应TreeNode 实例
8         */
9        TreeNode addChild(TaskNode newChild) throws
   TreeStructureException;
10
11       /**
12        * 添加 newChild 到 当前TreeNode 代理任务节点的子节点 postSibling的后
   一位置. 若newChild为空、postSibling为空、postSibling不在任务树上或newChild
   已经在任务树上则抛出异常
13        * <br><br>
14        *
```

```
15          * 参数postSibling当前TreeNode 代理任务节点的一个子节点
16          * 参数 newChild 要添加到 postSibling前的新节点
17          * 返回 newChild 对应TreeNode 实例
18          *
19          */
20         TreeNode insertChildBefore(TreeNode postSibling, TaskNode
    newChild) throws TreeStructureException;
21
22         /**
23          * 添加 newChild 到 当前TreeNode 代理的任务节点子节点 previousSibling
    的前一位置. 若newChild为空、previousSibling为空、previousSibling不在任务树
    上或newChild已经在任务树上则抛出异常
24          * <br><br>
25          *
26          * 参数 previousSibling当前TreeNode 实例代理任务节点的一个子节点
27          * 参数 newChild 要添加到 previousSibling 后的新节点
28          * 返回 newChild 代理TreeNode 实例
29          */
30         TreeNode insertChildAfter(TreeNode previousSibling, TaskNode
    newChild) throws TreeStructureException;
31
32         /**
33          * 移除当前TreeNode 实例代理任务节点的子节点childNode. 若childNode为空
    或childNode不在任务树上则抛出异常
34          * <br><br>
35          *
36          * 参数 childNode 删除目标节点，当前TreeNode 实例代理任务节点的子节点
37          * 返回 删除结果
38          */
39         boolean removeChild(TreeNode childNode) throws
    TreeStructureException;
40
41         /**
42          * 移除当前TreeNode 实例代理任务节点的所有子节点，若删除失败则抛出异常。
43          * <br><br>
44          *
45          * 返回 删除结果
46          */
47         boolean removeAllChildren() throws TreeStructureException;
48
49         /**
50          * 获取TreeNode代理任务节点的子节点TreeNode实例。
51          * <br><br>
52          *
53          * 返回 子节点TreeNode实例list
```

```
54          */
55         List<TreeNode>  getChildren();
56 
57         /**
58          * 获取当前TreeNode代理的任务节点的父节点代理TreeNode。若当前TreeNode代理的任务节点不在任务树上，则抛出异常
59          * <br><br>
60          *
61          * 返回 父节点代理TreeNode
62          */
63         TreeNode getParent() throws TreeStructureException;
64 
65         /**
66          * 获取当前TreeNode代理的任务节点
67          * <br><br>
68          *
69          * 返回 当前TreeNode代理的任务节点
70          */
71         TaskNode getTaskNode();
72 
73         /**
74          * 按深度和广度遍历当前TreeNode代理的任务节点的后代节点，深度优先
75          * <br><br>
76          *
77          * 参数 nodeVisitor 节点遍历器实例
78          */
79         void traverse(TaskNodeVisitor nodeVisitor);
80 
81         /**
82          * 设置当前TreeNode代理的任务节点的子节点锁定状态(若锁定，不能进行插入、删除及移动等操作)
83          * <br><br>
84          *
85          * 参数 lock 锁定状态
86          * 返回 当前TreeNode
87          */
88         TreeNode setChildSequenceLocked(boolean lock);
89 
90         /**
91          * 选中当前TreeNode代理的任务节点
92          */
93         void select();
94     }
```

代码块 5.13: TreeNode 接口类

从上可以看出，任务树节点代理 TreeNode 类主要有添加/删除子节点、获取父节点、获取子节点列表、遍历子节点、锁定/解锁子节点序列这几个接口，不言而喻获取父 /子节点和添加删除子节点是最常用的方法，也比较简单，后面会有相关实例介绍其用法。锁定/解锁子节点序列虽然不常使用，但是顾名思义设置后可以使子节点被删除/添加/移动等行为被禁止/允许，使用也比较简单，因此这里不再赘述。接下来以一段示例代码代码块5.14示范如何使用 TreeNode 中的接口进行节点操作。

```
1    void test() {
2         // 获取当前 节点贡献 的节点代理 TreeNode实例
3         // 获取节点工厂类
4         TreeNode treeNode = this.taskApiProvider.getTaskModel().
     getContributionTreeNode(this);
5         TaskNodeFactory factory = this.taskApiProvider.getTaskModel().
     getTaskNodeFactory();
6
7         // 创建 名称为 “Hello world!”的 FolderNode
8         // 创建 内容为 “after folderNode!”的 CommentNode
9         // 创建 内容为 “before folderNode!”的 CommentNode
10        FolderNode folderNode = factory.createFolderNode();
11        folderNode.setDisplay("Hello world!");
12        CommentNode postCommentNode = factory.createCommentNode();
13        postCommentNode.setCommentString("after folderNode!");
14        CommentNode prevCommentNode = factory.createCommentNode();
15        prevCommentNode.setCommentString("before folderNode!");
16
17        try {
18             // 调用 addChild    将 folderNode   插入到 当前定制任务节点下
19             // 调用 insertChildAfter 将 postCommentNode  插入到
     folderTreeNode 后
20            // 调用 insertChildBefore 将 prevCommentNode 插入到
     folderTreeNode 前
21            TreeNode folderTreeNode = treeNode.addChild(folderNode);
22            treeNode.insertChildAfter(folderTreeNode, postCommentNode);
23            treeNode.insertChildBefore(folderTreeNode, prevCommentNode)
     ;
24
25            // 通过 getParent   获取 folderTreeNode 父节点的 TreeNode代理
26            // 通过 getChildren 获取 folderTreeNode 父节点的 子节点列表
27            // 通过 removeChild 删除 folderTreeNode 父节点的 第3个子节点
28            TreeNode parentTreeNode = folderTreeNode.getParent();
29            TreeNode removeTarget = parentTreeNode.getChildren().get(2)
     ;
30            treeNode.removeChild(removeTarget);
```

```
31
32              // 通过 getTaskNode 获取 folderTreeNode 的 TaskNode 代理
33              // 通过 select              选中 folderTreeNode
34              // 通过 setChildSequenceLocked 锁定 folderTreeNode
35              // 通过 folderTreeNode 的 TaskNode 代理接口设置名称为"has been
    locked"
36              TreeNode firstChildTreeNode = parentTreeNode.getChildren().
    get(0);
37              TaskNode firstChild = firstChildTreeNode.getTaskNode();
38              if (firstChild instanceof FolderNode){
39                 firstChildTreeNode.select();
40                 firstChildTreeNode.setChildSequenceLocked(true);
41                 ((FolderNode) firstChild).setDisplay("has been locked");
42              }
43
44              treeNode.removeAllChildren();
45          }catch (TreeStructureException treeStructureException) {
46              treeStructureException.printStackTrace();
47          }
48      }
```

代码块 5.14: TreeNode 类接口使用示例

代码块5.14中对 TreeNode 接口类中除 traverse 外的所有接口使用方法都有所涉及，其他的内部节点均可按照代码块5.14中方法进行节点操作，但需注意的是，因为定制节点的数据操作在定制节点的贡献类中定义，因此定制节点的 treeNode 实例通过 getTaskNode 方法获得的定制节点对应的 TaskNode 无法进行定制节点的数据访问和设置。

5.5.3 任务树及节点访问

第 5.5.2 小节 节点插入删除一节中对 TreeNode 接口类中除 traverse 外的所有节点操作方法都有涉及，那么本节就重点描述下 traverse 方法遍历节点、任务树访问及内部节点业务接口的使用方法。

1. traverse 方法遍历节点

开发者可以通过任务节点 TreeNode 代理中的 traverse 接口进行任务节点后代节点的遍历及查找。traverse 接口支持两种方式进行访问：按节点类型遍历和按深度广度遍历访问，其参数 TaskNodeVisitor 用于定制访问处理逻辑，该接口类主要接口如下代码块5.15所示：

```
public abstract class TaskNodeVisitor {
    private boolean terminated = false;

    public TaskNodeVisitor() {
    }

// 终止遍历
    public void terminate() {
        this.terminated = true;
    }

// 获取当前遍历状态
    public boolean isTerminated() {
        return terminated;
    }

// 按深度 序号遍历，currentLocation为当前遍历节点
    public void visit(int index, int depth, TreeNode currentLocation
);

// 按AssignmentNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
    public void visit(AssignmentNode assignmentNode, int index, int
depth);

// 按ArcMotionNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
    public void visit(ArcMotionNode arcMotionNode, int index, int
depth);

// 按CommentNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
    public void visit(CommentNode commentNode, int index, int depth)
;

// 按DirectionNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
    public void visit(DirectionNode directionNode, int index, int
depth);

// 按 ElseIfNode 类型进行遍历，depth和index分别为当前遍历节点的深度和序号
    public void visit(ElseIfNode elseIfNode, int index, int depth);

// 按 ElseNode 类型进行遍历，depth和index分别为当前遍历节点的深度和序号
    public void visit(ElseNode elseNode, int index, int depth);

// 按 FolderNode 类型进行遍历，depth和index分别为当前遍历节点的深度和序号
    public void visit(FolderNode folderNode, int index, int depth);

```

```
41      // 按 HaltNode 类型进行遍历，depth和index分别为当前遍历节点的深度和序号
42          public void visit(HaltNode haltNode, int index, int depth);
43
44      // 按 IfNode 类型进行遍历，depth和index分别为当前遍历节点的深度和序号
45          public void visit(IfNode ifNode, int index, int depth);
46
47      // 按 LoopNode 类型进行遍历，depth和index分别为当前遍历节点的深度和序号
48          public void visit(LoopNode loopNode, int index, int depth);
49
50      // 按MoveNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
51          public void visit(MoveNode moveNode, int index, int depth);
52
53      // 按PopupNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
54          public void visit(PopupNode popupNode, int index, int depth);
55
56      // 按SetNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
57          public void visit(SetNode setNode, int index, int depth);
58
59      // 按UntilNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
60          public void visit(UntilNode untilNode, int index, int depth);
61
62      // 按WaitNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
63          public void visit(WaitNode waitNode, int index, int depth);
64
65      // 按WaypointNode类型进行遍历，depth和index分别为当前遍历节点的深度和序号
66          public void visit(WaypointNode waypointNode, int index, int
    depth);
67
68        // 按 TaskExtensionNode 类型进行遍历，depth和index分别为当前遍历节点的深度
    和序号
69          public void visit(TaskExtensionNode taskExtensionNode, int index
    , int depth);
70      }
```

代码块 5.15: TaskNodeVisitor 类接口

代码块5.15中主要有三类接口:

- 遍历状态设置及获取接口:terminate() 中止遍历及 isTerminated() 遍历是否中止访问接口；
- 按深度广度访问接口:visit(int, int, TreeNode)；
- 按节点类型访问接口：剩余其他。

在实际应用中通常通过重写 TaskNodeVisitor 的其中一个访问接口按照相应模式进行访问。建议在深度广度模式下最后一个目标节点相关处理完毕后，调用 terminate() 中止遍历。

下面以一段示例代码代码块5.16示范如何使用 TreeNode 中的接口进行节点操作。

```
 void test() {
      TreeNode treeNode = this.taskApiProvider.getTaskModel().
getContributionTreeNode(this);
      TaskNodeFactory factory = this.taskApiProvider.getTaskModel().
getTaskNodeFactory();
      FolderNode folderNode = factory.createFolderNode();
      folderNode.setDisplay("Hello world!");
      CommentNode postCommentNode = factory.createCommentNode();
      postCommentNode.setCommentString("after folderNode!");
      CommentNode prevCommentNode = factory.createCommentNode();
      prevCommentNode.setCommentString("before folderNode!");
      try {

          TreeNode folderTreeNode = treeNode.addChild(folderNode);
          treeNode.insertChildAfter(folderTreeNode, postCommentNode);
          treeNode.insertChildBefore(folderTreeNode, prevCommentNode)
;

          // 按节点类型遍历访问后代节点中的 CommentNode,并设置内容为 "
traversed"
          treeNode.traverse(new TaskNodeVisitor() {
              @Override
              public void visit(CommentNode commentNode, int index,
int depth) {
                  if (commentNode != null){
                      commentNode.setCommentString("traversed");
                  }
              }
          });

          // 按深度广度遍历访问后代节点中的 FolderNode,并设置名称为为 "
traversed", 然后调用terminate中止遍历
          treeNode.traverse(new TaskNodeVisitor() {
              @Override
              public void visit(int index, int depth, TreeNode
currentLocation) {
                  if (depth == 1 && index == 1){
                      if (currentLocation.getTaskNode() instanceof
FolderNode){
                          ((FolderNode) currentLocation.getTaskNode()
).setDisplay("traversed");
                      }
```

```
                        terminate();
                    }
                }
            });
        }catch (TreeStructureException treeStructureException) {
            treeStructureException.printStackTrace();
        }
    }
```

代码块 5.16: traverse 接口遍历访问后代节点

2. 任务树访问

前文对于 TaskNodeModel 类有所涉及但不系统，这里系统描述该类的作用及使用方法。

TaskNodeModel 类主要用于获取任务节点贡献的 TreeNode 代理及获取任务节点工厂，其内部接口如下代码块5.17所示：

```
public interface TaskNodeModel {
    /**
     * 获取节点工厂TaskNodeFactory实例-可用于创建内部节点及定制任务节点
     */
    TaskNodeFactory getTaskNodeFactory();

    /**
     * 获取任务节点贡献的TreeNode代理
     */
    TreeNode getContributionTreeNode(TaskNodeContribution
contribution);
}
```

代码块 5.17: TaskNodeModel 接口

TreeNode 接口类前文有详细描述，此处不再赘述。

TaskNodeFactory 则是任务节点工厂类，可以用于创建内部任务节点及定制任务节点，其内部接口如下代码块5.18所示：

```
public interface TaskNodeFactory {
    /**
     * 创建 taskExtensionNodeServiceClass 对应的定制节点
     */
```

```
5         TaskExtensionNode createExtensionNode(Class
      taskExtensionNodeServiceClass);
6
7         /**
8          * 创建 ArcMotionNode 节点
9          */
10        ArcMotionNode createArcMotionNode();
11
12        /**
13         * 创建 ArcMotionNode 节点模板
14         */
15        ArcMotionNode createArcMotionNodeTemplate();
16
17        /**
18         * 创建 AssignmentNode
19         */
20        AssignmentNode createAssignmentNode();
21
22        ...
23
24    }
```

代码块 5.18: TaskNodeFactory 接口

基于以上描述，开发者可以通过定制节点贡献类的 taskApiProvider 及节点参数视图定制类中的 taskNodeViewApiProvider 接口获取 TaskNodeModel 实例进行节点创建或者节点 TreeNode 代理获取。如下代码块5.19所示分别为在定制节点贡献类/节点参数视图定制类获取 TaskNodeModel 实例方法：

```
1     // 定制节点贡献类获取TaskNodeModel实例
2     TaskNodeModel taskNodeModel = this.taskApiProvider.getTaskModel();
3
4     ...
5
6     // 节点参数视图定制类获取TaskNodeModel实例
7     this.taskApiProvider = this.taskNodeViewApiProvider.
      getTaskApiProvider();
8     TaskNodeModel taskNodeModel = this.taskApiProvider.getTaskModel();
```

代码块 5.19: TaskNodeModel 实例获取方法

3. 内部节点业务接口

任务节点定制过程中，必不可少的会需要调用内部任务节点接口，或是业务接口或是交互接口，其中这里的交互接口主要指前文代码块5.13 TreeNode 接口类中的接口，主要用于节点插入/删除/获取父子节点/选中/锁定等交互操作，前文有详细描述，此处不再赘述。

业务接口则主要提供节点具体业务接口及数据访问更改等接口，因此其实一定程度上定制任务节点贡献类也可以理解为定制节点的业务接口类。定制任务节点访问同一项目内的其他定制任务节点业务接口可通过访问其贡献类实例的实现。

而访问内部任务节点的业务接口，则需要获取其业务接口代理类 TaskNode，如下代码块5.20所示为内部节点 PopupNode 的业务代理类 PopupNode:

```
1   public interface PopupNode extends TaskNode {
2       /**
3        * 获取弹出框消息内容
4        */
5       String getMessage();
6
7       /**
8        * 设置弹出框消息内容
9        */
10      PopupNode setMessage(String message);
11
12      /**
13       * 显示弹出框消息类型
14       */
15      MessageType getMessageDialogType();
16
17      /**
18       * 设置弹出框消息类型
19       */
20      PopupNode setMessageDialogType(MessageType messageType);
21  }
```

代码块 5.20: PopupNode 业务接口

其他内部节点的业务接口详情参考 ELITECO SDK API 接口文档，而内部节点的业务接口代理类的获取方式前文有说涉及，即通过内部节点的 TreeNode 代理实例获取，可参考代码块5.14中代码。

5.5.4 撤销重做

ELITECO Plugin 支持 GUI 操作人员对节点插入/删除及节点数据操作的撤销重做，可撤销重做已更改的节点关键数据和任务树结构，并进行同步刷新。因此，请务必通过撤销重做功能来记录那些会影响相应节点显示内容的关键数据更改及任务树结构操作。ELITECO Plugin 支持以下几种数据操作的记录，具体包括以下几种：

- 节点贡献数据 model 中设置数据；
- 节点贡献数据 model 中删除数据；
- 节点贡献中操作任务树上内部节点；
- 任务树上添加节点；
- 任务树上删除节点。

下面代码块5.21以计数器节点扩展中变量下拉框选中变量后设置变量过程为例介绍撤销重做服务:

```
1     public void setVariable(final Variable variable) {
2         this.taskApiProvider.getUndoRedoManager().recordChanges(() ->
      taskNodeDataModelWrapper.setVariable("selected_var", variable)
3         );
4     }
```

代码块 5.21: TreeNode 类接口使用示例

如上所示，可以看出使用撤销重做服务记录操作比较简单，ELITECO Plugin 支持的 5 种操作都可以使用撤销重做服务进行记录，上面代码中记录了节点贡献数据 model 中设置变量的操作，该方法正确执行完毕后，即可通过 EliRobot 机器人平台任务页面任务树编辑工具中的撤销重做按钮进行数据的重做和回退。

如上所述，首先务必保证撤销重做服务只用来记录 GUI 操作人员的操作引起的数据改变或者任务树结构改变。因为撤销重做服务是用来记录 GUI 操作人员的操作，在非直接由 GUI 操作人员操作引起的数据处理或节点处理中不建议使用撤销重做记录，而且诸如节点贡献构造方法中、某些事件方法中以及其他不被推荐的位置使用是不安全的，会抛出异常或引起嵌套刷新问题。

同时需要注意的是，节点贡献类设置/删除数据时需要保证对应的定制任务节点已经被添加到任务树上，否则不会被记录。

另外，对于影响对应节点显示内容的关键数据及任务树结构的更改操作，请务必使用撤销重做功能进行记录，否则任务树或参数页面不会刷新。

5.5.5 任务树 UI 句柄、节点策略及事件

ELITECO Plugin 提拱了用于开发者监听任务树上与当前节点贡献相关事件的三个监听器类: ContributionInsertedListener、ContributionRemovedListener、ContributionLoad-Completed Listener。通过这三个监听器开发者可以根据需要定制个性化的事件处理逻辑，如下代码块5.22所示展示了这三个监听器的使用方法:

```
1   public class CounterTaskNodeContribution implements
    TaskNodeContribution, ContributionInsertedListener,
    ContributionRemovedListener, ContributionLoadCompletedListener {
2           ......
3           ......
4       ......
5
6       @Override
7       public void onInserted(ContributionInsertedContext
    insertedContext) {
8         // ContributionInsertedListener  对应的定制节点被插入到树上时事件激发
9           System.out.println("Counter is inserted");
10      }
11
12      @Override
13      public void loadComplete(ContributionLoadCompleteContext
    completeContext) {
14        // ContributionInsertedListener 新加载的任务文件有节点的定制节点贡献
    类实现了该接口，则会被触发
15          System.out.println("Counter has been loaded in task tree
    model");
16      }
17
18      @Override
19      public void onRemoved(ContributionRemovedContext
    contributionRemovedContext) {
20        // ContributionLoadCompletedListener 对应的定制节点从树上被移除时事
    件激发
21          System.out.println("Counter is removed");
22      }
23  }
```

代码块 5.22: TreeNode 类接口使用示例

如代码块5.22所示，上面所述三个事件接口类使用比较简单。

ELITECO Plugin 还提拱了用于控制节点交互策略的节点插入/删除策略接口，只需将定

制节点贡献类实现 ContributionInsertionRule/ContributionRemovalRule 接口即可。如改造定制任务节点贡献，使其实现上述两接口类中的接口，并规定插入/删除策略如下：计数器节点初始有一个 FolderNode 子节点；只能被作为第一个子节点插入到 FolderNode 下；作为 FolderNode 第一个子节点时不能被删除；其第一个子节点 FolderNode 也不能被删除。这里规定的策略仅用作示例，考虑其实际应用中的合理性，则其实现代码如下 (代码块5.23) 所示:

```
1   public class MyCounterTaskNodeContribution implements
    TaskNodeContribution, ContributionInsertedListener,
    ContributionRemovedListener, ContributionLoadCompletedListener {
2       ....
3       // 只允许被插入到FolderNode 下并作为第一个子节点
4       @Override
5       public boolean canInsertToTargetParent(TaskNode parent, int
    index) {
6           return parent instanceof FolderNode && index == 0;
7       }
8
9       // 只允许插入一个子节点
10      @Override
11      public boolean canInsertAChild(TaskNode child, int index) {
12          TreeNode treeNode = this.taskApiProvider.getTaskModel().
    getContributionTreeNode(this);
13          return treeNode.getChildren().size() < 2;
14      }
15
16      // 只允许在非 FolderNode 第一个子节点时从任务树上删除
17      @Override
18      public boolean canRemove(TaskNode parent, int index) {
19          return !(parent instanceof FolderNode && index == 0);
20      }
21
22      // 目标节点是 FolderNode 且是第一个子节点时，不允许从任务树上删除
23      @Override
24      public boolean canRemoveAChild(TaskNode child, int index) {
25          return !(child instanceof FolderNode && index == 0);
26      }
27  }
```

代码块 5.23: MyCounterTaskNodeContribution 规定节点插入/删除策略

ELITECO Plugin 还提拱了用于任务树及节点参数视图的 UI 句柄类-TaskUIHandler，用于在需要的时候主动构建任务树、刷新任务树文本、刷新节点参数视图，如下代码块 5.24所示为:

```
1   public interface TaskUIHandler {
2       /**
3        * 更新任务树节点显示
4        */
5       void updateTaskTreeDisplay();
6
7       /**
8        * 重构任务树
9        */
10  void updateTaskTreeStructure();
11
12      /**
13       * 更新节点参数视图
14       */
15      void updateParamsView();
16  }
```

代码块 5.24: TaskUIHandler 接口

TaskUIHandler 句柄类在节点定制过程中，可以通过如下 (代码块5.25) 代码中的方式获取并使用:

```
1   // 定制节点贡献类获取TaskNodeModel实例
2   TaskUIHandler taskUIHandler= this.taskApiProvider.getTaskUIHandler()
    ;
3
4   ...
5
6   // 节点参数视图定制类获取TaskNodeModel实例
7   this.taskApiProvider = this.taskNodeViewApiProvider.
    getTaskApiProvider();
8   TaskUIHandler taskUIHandler = this.taskApiProvider. getTaskUIHandler
    ();
```

代码块 5.25: TaskUIHandler 实例获取方法

TaskUIHandler 接口类中的 3 个接口使用比较简单，不再过多描述。该接口类中的接口仅限于十分必要的时候调用，因为 ELITECO Plugin 中的绝大多数节点、数据操作涉及的刷新任务树文本、重构任务树、刷新参数视图都会在合适的时机自动刷新，无需开发者在代码层面主动调用。而且这 3 个接口设计不宜用在多线程中以及可能引起循环调用的场合，因此有必要进行说明。

updateTaskTreeDisplay 通常不推荐使用该接口主动刷新任务树显示，因为大多数需要刷新的场合都会自动刷新，仅在特殊情况下才需要主动调用刷新。同时不建议在以下场合使用：

- 在多线程中较短时间内多次调用；
- 循环调用；
- 在记录的可撤销重做的操作内调用；
- 删除、添加、移动、压缩及解压缩节点和其他的会引起任务树结构改变的操作中使用；

以上 4 种情况或是会因为调用过快或循环调用引起任务树闪烁或系统卡顿，或是在系统会自动处理刷新的情况重复刷新。

updateTaskTreeStructure 用于重构任务树，相较 updateTaskTreeDisplay 仅刷新任务树显示而言，该接口显得更加重型，因此在 updateTaskTreeDisplay 不建议使用的几种场景也不被建议使用。仅限于特殊情况需要使用，且 updateTaskTreeDisplay 无法满足需求时使用。

updateParamsView 用于更新节点参数视图，若当前显示参数页面与选中任务节点不匹配，该方法会加载显示任务节点对应的参数视图，否则只会刷新任务视图。同理该接口也是非必要场景下限制使用的，而且除在 updateTaskTreeDisplay 不建议使用的几种场景也不被建议使用，该接口明确禁止在定制节点贡献类的 onViewOpen() 方法中使用，因为该方法会调用 onViewOpen() 接口，这会因循环调用造成页面闪烁及卡顿。

5.5.6 变量表达式

在任务模块中，有较多场景下，需要通过表达式来为变量赋值，因此 ELITECO Plugin 也支持表达式创建。一个合法的表达式通常是由一个或多个元素构成，这些元素通常可以是：字符串，字符，I/O，变量，坐标系，位姿，路点，内置方法，数组以及一个或多个以逻辑运算符结合或嵌套的结合体等。如“I/O”、“变量 and I/O”、“(变量 or I/O) xor 路点”以及“路点?=数组”等 (为便于理解表达式的元素构成，实际表达式通常是更具体更复杂的内容)。

因此 ELITECO Plugin 中提供了 ExpressionService 类去创建表达式，在节点贡献和节点参数视图中分别可以通过 TaskApiProvider 及 TaskNodeViewApiProvider 获取表达式服务 ExpressionService 实例。如下代码块5.26所示，在节点贡献构造方法中获取表达式服务:

```
1    public MyCounterTaskNodeContribution(TaskApiProvider taskApiProvider
     , TaskNodeDataModelWrapper taskNodeDataModelWrapper) {
2        this.taskApiProvider = taskApiProvider;
3        this.expressionService = this.taskApiProvider.
     getExpressionService();
4    }
```

代码块 5.26: ExpressionService 实例获取方法

通过 ExpressionService 类可以创建一个表达式构造器 (ExpressionConstructor)，而通过表达式构造器中的众多接口可以允许开发人员无需 gui 辅助创建出复杂的表达式用于为写

入脚本时变量赋值或作为判断条件，如下代码块5.27所示 ExpressionConstructor 中接口 (有缩略)：

```
public interface ExpressionConstructor {
    /**
     * 添加一个字符串元素到表达式中
     */
    ExpressionConstructor appendStringCell(String str);

    /**
     * 添加一个字符元素到表达式中
     */
    ExpressionConstructor appendCharCell(char c);

    /**
     * 添加一个表征元素到表达式中,其中参数expCell该元素的显示内容,
scriptCode则是被写入到脚本中用于运行的真实内容,
     * 可用于将某些不利于表达式理解或不便于显示的内容用表征内容展示，而运行时有
效内容是scriptCode
     */
    ExpressionConstructor appendTokenCell(String expCell, String
scriptCode);

    /**
     * 添加一个变量元素到表达式中
     */
    ExpressionConstructor appendVariableCell(Variable variable);

    /**
     * 添加一个坐标系元素到表达式中
     */
    ExpressionConstructor appendFrame(Frame frame);

    ...

    /**
     * 添加一个boolean值True用于判断，通常可用于 元素 ?= True 之类的场景
     */
    ExpressionConstructor appendTrueSymbol();

    /**
     * 添加一个boolean值False用于判断，通常可用于 元素 ?= False 之类的场景
     */
    ExpressionConstructor appendFalseSymbol();
```

```
39
40          /**
41           * 添加一个逻辑与 "and" 元素到表达式中
42           */
43          ExpressionConstructor appendAndSymbol();
44
45          ...
46
47          /**
48           * 添加一个左半括号 "(" 元素到表达式中
49           */
50          ExpressionConstructor appendLeftBracketSymbol();
51
52          /**
53           * 添加一个右半括号 ")" 元素到表达式中
54           */
55          ExpressionConstructor appendRightBracketSymbol();
56
57          ...
58
59          /**
60           * 构造表达式
61           */
62          Expression construct() throws IllegalExpressionException;
63      }
```

代码块 5.27: ExpressionConstructor 中的接口

表达式构造器将开放了众多构造表达式常用的接口，开发者可以较为便捷的通过该接口创建满足自身需求的表达式，但是这种方式需要开发者直接在代码中将表达式创建代码确定，因此通常是固定的表达式，无法满足多种多样的表达式编辑需求，同时也无法支持 gui 用户的个性化编辑，因此通常适用于变量的初始化，也就是在创建变量的同时给予变量一个写入脚本时用于初始化变量的表达式，如下代码块5.28所示计数器节点贡献中创建变量的方法中在创建任务变量的同时给予其初始化表达式 (创建变量将会在 第 8.1 节 变量中详细描述):

```
1   public TaskVariable createGlobalVariable(String variableName) {
2       TaskVariable variable = null;
3       // 创建了一个含有一个token元素的表达式
4       Expression initExp = this.expressionService.
    createExpressionConstructor().appendTokenCell("0", "0").construct();
5       try{
6        // 创建变量同时传递初始化表达式用于运行脚本时进行变量初始化
7        variable = this.variableService.getVariableFactory().
    createTaskVariable(variableName, initExp);
```

```
        } catch (VariableException | IllegalExpressionException e) {
         e.printStackTrace();
        }

        return variable;
    }
```

代码块 5.28: ExpressionConstructor 创建具有初始表达式值的变量

当然，鉴于上述代码式创建表达式的弊端，ELITECO Plugin 也提供了 gui 交互式的表达式创建方法：允许开发者在创建一个表达式输入框和表达式键盘，并定制点击事件用于弹出表达式键盘及处理键盘提交事件处理生成的表达式。

如下代码块5.29所示为一段弹出节点贡献中创建表达式输入框、表达式键盘并处理相关事件用于生成个性化表达式的代码：

```
    private void createExpressionTextField() {
     // 创建表达式输入框，
     ExpressionTextField expressionField = this.SwingService.
    expressionJTextFieldService.createExpressionTextField();
     // 添加表达式输入框点击事件处理-弹出表达式键盘，并在表达式键盘提交时给表达式输
    入框设置表达式
     textField.addMouseListener(new MouseAdapter() {
         @Override
            public void mousePressed(MouseEvent e)  {
             SwingService.keyboardService.showExpressionKeyboard(
    textField, new BaseKeyboardCallback() {
                    @Override
                    public void onOk(Object value) {
                        if (value instanceof Expression) {
                         textField.setExpression((Expression) value);
                        }
                    }
                });
            }
        });

        return expressionField;
    }
```

代码块 5.29: gui 创建表达式

如上代码块5.29创建了一个表达式输入框，并为表达式输入框添加了点击事件-弹出表达式键盘，并在表达式键盘提交时处理表达式，并返回创建的表达式，可以将以上返回的表达

式输入框构造节点参数页面时候添加到页面上，用于用户通过表达式输入框查看并编辑表达式。

第 6 章 定制导航栏贡献

ELITECO Plugin 支持定制导航栏贡献：通过定制导航栏贡献及导航栏服务使得导航栏贡献能被正确加载为 EliRobot 主页” 插件” 选项卡管理的定制导航栏贡献之一。通过导航栏贡献，可以扩展 EliRobot 的功能，配置个性化的功能或设备参数。下面将通过一个简单的 demo-myNavbar 按钮来说明导航栏贡献的定制过程及相关特性的使用。

6.1 项目新建

项目新建过程、License、国际化及图标资源等相关配置详见 第 3.1 节 ELITECO Plugin 项目新建，此处不再赘述。

导航栏贡献定制项目实例-myNavbar 文件结构图如下**图** 6-1所示：

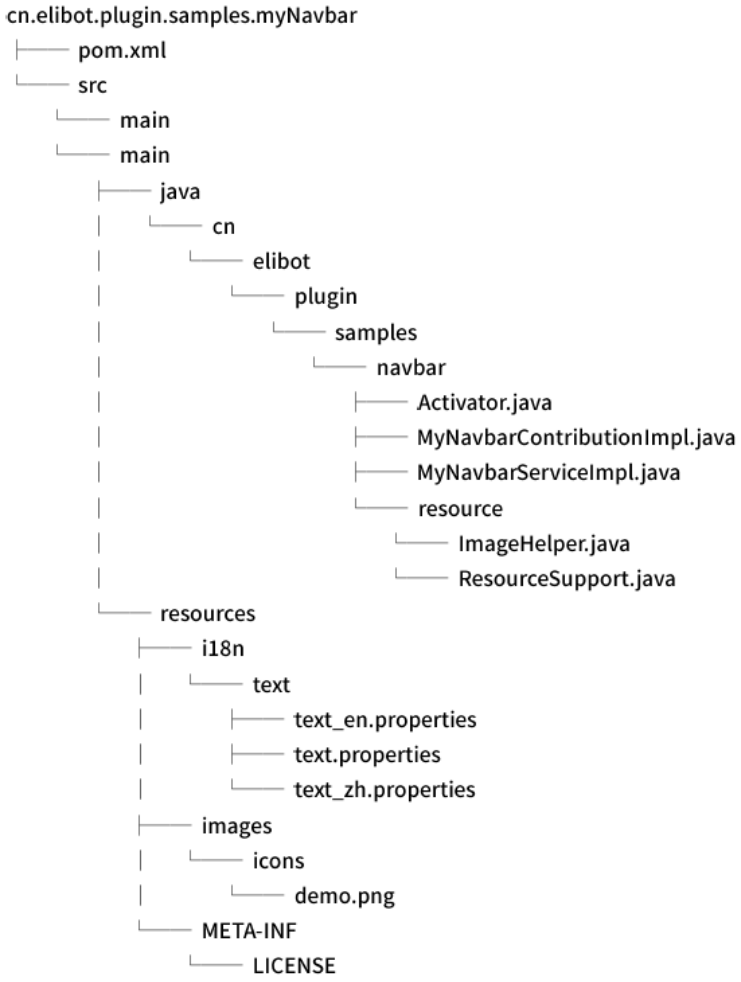


图 6-1: myNavbar 项目文件结构

6.2 定制工具栏服务

定制导航栏贡献主要分两个步骤: 定制导航栏服务类、定制导航栏贡献类。其中导航栏贡献类主要定义功能及选项卡视图；导航栏服务类则定义了导航栏贡献的公共属性、导航栏贡献类实例创建等。

myNavbar 导航栏贡献功能设计为展示导航栏贡献定制流程。因此，其贡献页面比较简单，仅有一个文本标签。

下面章节将以 myNavbar 为例，详细说明导航栏贡献定制流程及内部功能接口的使用。

定制导航栏贡献服务需要通过实现 NavbarService 接口类。如前所述，该定制类定义了导航栏贡献的公共属性、导航栏贡献类创建等。为方便描述，使用该接口空的实现类 MyNavbarServiceImpl.java 来描述定制过程，如下代码块6.1所示:

```
1   public class MyNavbarServiceImpl implements NavbarService {
2    /**
3        * 配置 工具栏 上下文
4        */
5       @Override
6       public void configure(NavbarContext navbarContext) {
7        // to be implements
8       }
9
10   /**
11      * 创建 工具栏贡献
12      */
13      @Override
14      public NavbarContribution createContribution() {
15       // to be implements
16      }
17  }
```

代码块 6.1: 待实现的 MyNavbarServiceImpl.java

由上代码块6.1可见，NavbarService 的实现类主要有以下 configure 和 createContribution 方法。

configure 方法主要用于通过参数 NavbarContext 配置导航栏贡献的上下文，其参数及描述如下**表** 6-1所示：

表 6-1. configure 方法参数

参数	描述
NavbarContext navbarContext	系统内部 api 的接口、数据存储接口、导航栏贡献激活条目文本图标名称等配置接口

createContribution 方法用于创建导航栏贡献实例，如有需要可将 configure 中的参数 NavbarContext 保存为 MyNavbarServiceImpl 类的属性，以便提供给导航栏贡献，为其提供接口服务。

按照前边 myNavbar 所述，需要完成 MyNavbarServiceImpl.java 后的代码如代码块6.2所示：

```
1   public class MyNavbarServiceImpl implements NavbarService {
2       @Override
3       public void configure(NavbarContext navbarContext) {
4           context.setNavbarIcon(ImageHelper.loadImage("demo.png"));
5           Context.setNavbarName(ResourceSupport.
    getDefaultResourceBundle().getString("hello_world"));
6       }
7
8       @Override
9       public NavbarContribution createContribution() {
10          return new MyNavbarServiceImpl(this.navbarContext);
11      }
12  }
```

代码块 6.2: 完全实现的 MyNavbarServiceImpl.java

MyNavbarServiceImpl 的实现比较简单，不再过多描述。

6.3 定制导航栏贡献

定制导航栏贡献需要通过实现 NavbarContribution 接口类。为便于描述，使用该接口的空实现类 MyNavbarServiceImpl.java 来描述定制过程，如下代码块6.3所示:

```
1   public class MyNavbarServiceImpl implements NavbarContribution {
2       MyNavbarServiceImpl(NavbarContext navbarContext) {
3       }
```

```
4
5         @Override
6         public void buildUI(JPanel panel) {
7         }
8
9         @Override
10        public void openView() {
11        }
12
13        @Override
14        public void closeView() {
15        }
16    }
```

代码块 6.3: 待实现的 MyNavbarServiceImpl.java

如上代码块6.3所示，一个 MyNavbarServiceImpl 接口的实现类，主要包括构造方法、导航栏贡献视图构建方法 buildUI、导航栏贡献视图打开事件方法 openView 及关闭事件方法 closeView。

MyNavbarServiceImpl 构造方法主要参数及描述同**表** 6-1所示。

buildUI 主要用于构建导航栏贡献视图，其参数及描述如下**表** 6-2所示：

表 6-2. buildUI 方法参数

参数	描述
JPanel panel	主页选项卡视图页面本身，承载 UI 组件

至于导航栏贡献视图打开事件方法 openView 及关闭事件方法 closeView，主要用做导航栏贡献视图打开/关闭时的事件监听，比较简单，不再过多描述。激活条目属性

按照前边 myNavbar 所述，需要完成 MyNavbarServiceImpl.java 后的代码如代码块6.4所示：

```
1     public class MyNavbarServiceImpl implements NavbarContribution {
2         private final ResourceBundle defaultLocaleBundle;
3         MyNavbarContributionImpl() {
4             DefaultLocaleBundle = ResourceSupport.
    getDefaultResourceBundle()
5         }
6
7         @Override
```

```
8         public void buildUI(JPanel panel) {
9             panel.setLayout(new BorderLayout(5, 5));
10
11            JPanel mainPanel = SwingService.seniorPaneService.
    createSeniorPane(this.defaultLocaleBundle.getString("hello_world"));
12            mainPanel.setLayout(new BorderLayout());
13            panel.add(mainPanel, BorderLayout.CENTER);
14
15            JLabel elitePluginLabel = new JLabel(this.
    defaultLocaleBundle.getString("hello_elite_plugin"));
16             elitePluginLabel.setHorizontalAlignment(SwingConstants.
    CENTER);
17             elitePluginLabel.setPreferredSize(new Dimension(280, 36));
18             elitePluginLabel.setMaximumSize(new Dimension(280, 36));
19             elitePluginLabel.setMinimumSize(new Dimension(280, 36));
20             mainPanel.add(elitePluginLabel, BorderLayout.CENTER);
21        }
22
23        @Override
24        public void openView() {
25        System.out.printIn("Hello World Navigator Contribution View
    Opened.");
26        }
27
28        @Override
29        public void closeView() {
30        System.out.printIn("Hello World Navigator Contribution View
    Opened.");
31        }
32    }
```

代码块 6.4: 完全实现的 NavBarContributionImpl.java

代码块6.4中所示，myNavbar 导航栏贡献视图中仅有一个文本标签：

最后，将定制完成的 MyNavbarServiceImpl 类，在 Activator 中注册，如下代码块7.7代码中第 14 行。

```
1     public class Activator implements BundleActivator {
2         private ServiceReference<LocaleProvider>
    localeProviderServiceReference;
3
4       @Override
5        public void start(BundleContext bundleContext) throws Exception{
```

```
6           localeProviderServiceReference = bundleContext.
    getServiceReference (LocaleProvider.class);
7           if (localeProviderServiceReference != null) {
8           LocaleProvider localeProvider = bundleContext.getService(
    localeProviderServiceReference);
9           if (localeProvider != null) {
10              ResourceSupport.setLocaleProvider(localeProvider);
11           }
12          }
13          System.out.printIn("cn.elibot.plugin.samples.myNavbar.Activator
    says Hello World!");
14          BundleContext.registerService(NavbarService.class, new
    MyNavbarServiceImpl(), null);
15          }
16
17          @Override
18          public void stop(BundleContext bundlecontext) throws Exception {
19              System.out.printIn("cn.elibot.plugin.samples.myNavbar.
    Activator says Goodbye World!");
20          }
21      }
```

代码块 6.5: Activator 中注册 MyNavbarServiceImpl 类

完成注册后，即可按照第 3 章 ELITECO Plugin 项目前期中的构建部署流程，将其安装到 EliRobot 机器人平台下。

myNavbar 导航栏贡献激活条目及视图如**图** 6-2及**图** 6-3所示。

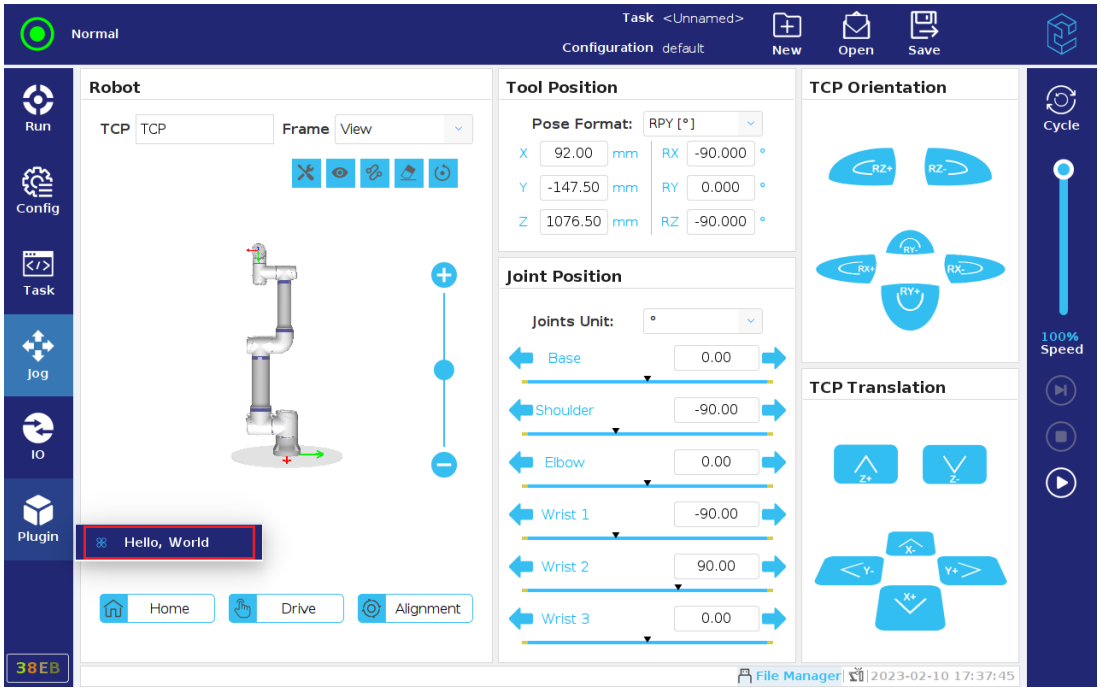


图 6-2: myNavbar 导航栏贡献激活条目

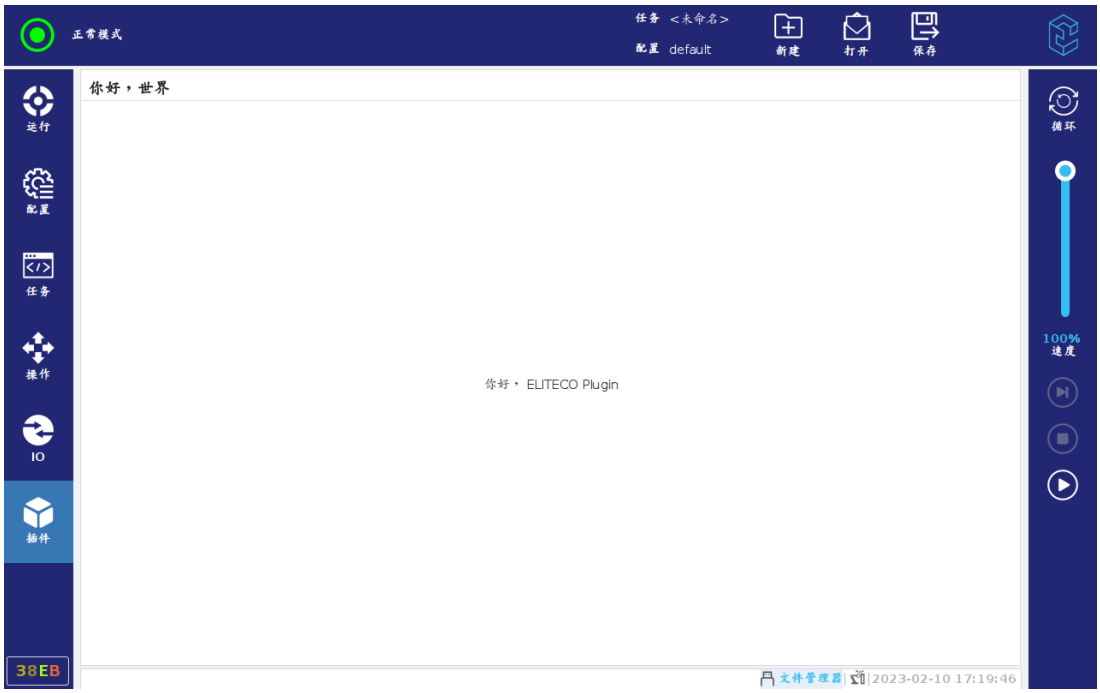


图 6-3: myNavbar 导航栏贡献视图

第 7 章 定制 Daemon 程序

ELITECO Plugin 支持开发者在 Linux 平台将其他语言的程序交由 EliRobot 机器人平台作为 Daemon 程序启动并长时间后台运行，该程序可以是任何可执行的脚本或二进制文件。Daemon 程序无需与标准输入输出设备交互，对系统或者某个用户程序提供后台监听、日志记录或者通信等服务，随系统关闭而终止，依赖事件触发或者周期性的执行某个任务。如 Linux 平台的打印、系统日志、服务器守护等服务都是由 Daemon 程序提供。

下面将通过一个简单的 demo-myDaemon 项目来说明 Daemon 程序的定制过程及相关特性的使用。

7.1 项目新建

项目新建过程、License、国际化及图标资源等相关配置详见 第 3 章 ELITECO Plugin 项目前期，此处不再赘述。

定制 Daemon 程序的定制项目实例-myDaemon 文件结构图如下**图** 7-1所示：

图 7-1: myDaemon 项目文件结构

7.2 定制 Daemon 程序

定制 Daemon 程序通过实现 DaemonService 接口类实现。DaemonService 实现类需指定 Daemon 程序的加载路径以及可执行文件。

为了开发者能够更好的理解，myDaemon 项目功能设计为展示 Daemon 程序定制流程，其内部功能主要是启动一个 python 脚本作为 Daemon 程序，该脚本创建了一个 web 服务器和 rpc 服务器分别提供 web 服务和 rpc 服务；同时定制一个导航栏贡献，导航栏贡献视图中的输入框会在输入时间后数据并通过 rpc 客户端向 rpc 服务器调用 setMessage 方法，方法参数为输入框输入数据；rpc 服务器响应 rpc 客户端的请求，执行 setMessage 方法，将数据设置到一个线程共享数据 message 中，消息内容可通过 web 服务器获取。

下面章节将以 myDaemon 为例，详细说明 Daemon 程序定制流程及内部功能接口的使用。为方便描述，使用该接口空的实现类 MyDaemonServiceImpl.java 来描述定制过程，如下代码块7.1所示:

```
1   public class MyDaemonServiceImpl implements DaemonService {
2       /**
3        * 指定 Daemon程序贡献的加载路径
4        */
5       @Override
6       public void init(DaemonContribution contribution) {
7           // to be implements
8       }
9
10      /**
11       * 指定 Daemon程序可执行程序
12       */
13      @Override
14      public URL getExecutable() {
15          // to be implements
16      }
17  }
```

代码块 7.1: 待实现的 MyDaemonServiceImpl.java

由上代码块7.1可见，DaemonService 实现类主要有以下 init 和 getExecutable 方法。

init 方法主要用于为参数 DaemonContribution 指定 Daemon 程序加载路径，其参数及描述如下**表** 7-1所示：

表 7-1. init 方法参数

参数	描述
DaemonContribution contribution	Daemon 程序贡献类，负责加载、启动、停止 Daemon 程序及查询其状态等操作

getExecutable 方法主要用于指定 Daemon 程序的可执行文件路径。

按照前边 MyDaemon 项目功能设计所述，需要完成 MyDaemonServiceImpl.java 后的代码如代码块7.2所示：

```
1   public class MyDaemonServiceImpl implements DaemonService {
2       private DaemonContribution daemonContribution;
3
4       @Override
5       public void init(DaemonContribution contribution) {
6           this.daemonContribution = contribution;
7           try {
8               daemonContribution.installResource(new URL("file:daemon
    /"));
9           } catch (MalformedURLException e) {
10          }
11      }
12
13      @Override
14      public URL getExecutable() {
15          try {
16              return new URL("file:daemon/server.py");
17          } catch (MalformedURLException e) {
18              return null;
19          }
20      }
21
22      public DaemonContribution getDaemon() {
23          return daemonContribution;
24      }
25  }
```

代码块 7.2: 完全实现的 MyDaemonServiceImpl.java

如上 (代码块7.2) 所示，完全实现的 MyDaemonServiceImpl 添加了 getDaemon 方法用于项目内其他文件获取 Daemon 程序贡献类访问其内部数据。MyDaemonServiceImpl 的 init 和 getExecutable 方法实现比较简单，不再过多描述。

/etc/service 目录包含指向当前运行的插件的 daemon 程序可执行文件的链接。如果 daemon 程序可执行文件存在链接，但实际上没有运行，则错误状态为在查询守护进程的状态时返回。指向 daemon 可执行文件的链接遵循生命周期将在移除插件时移除。

但是，如果需要，可以通过在中调用 start() 来实现自动启动。在 daemon 进程安装了其资源之后立即使用 init(DaemonContribution) 方法。

7.3 定制导航栏贡献

第 6 章定制导航栏贡献中已经详细描述了导航栏贡献定制方法，此处不再做过多赘述。按照前边 myDaemon 所述，实现用于 rpc 客户端数据输入及 Daemon 程序启动/停止交互的导航栏贡献。

myDaemon 导航栏贡献服务类代码如下代码块7.3所示：

```
1   public class MyDaemonNavbarServiceImpl implements NavbarService {
2       private MyDaemonServiceImpl daemonService;
3
4       public MyDaemonNavbarServiceImpl(MyDaemonServiceImpl
    daemonService) {
5           this.daemonService = daemonService;
6
7       }
8
9       @Override
10      public void configure(NavbarContext navbarContext) {
11          navbarContext.setNavbarIcon(ImageHelper.loadImage("
    small_plugin.png"));
12          navbarContext.setNavBarName(ResourceSupport.
    getDefaultResourceBundle().getString ("plugin_title"));
13      }
14
15      @Override
16      public NavbarContribution createContribution() {
17          return new MyDaemonViewImpl(this.daemonService);
18      }
19  }
```

代码块 7.3: MyDaemonNavBarServiceImpl 文件内容

myDaemon 导航栏贡献视图代码如下代码块7.4所示：

```
public class MyDaemonViewImpl implements NavbarContribution {
    private MyDaemonService daemonService;
    private XmlRpcMyDaemonFacade xmlRpcMyDaemonFacade;
    private JButton startButton;
    private JButton stopButton;
    private JTextField textField;

    public MyDaemonViewImpl(MyDaemonServiceImpl daemonService) {
        this.daemonService = daemonService;
        this.xmlRpcMyDaemonFacade = new XmlRpcMyDaemonFacade
("127.0.0.1", 4444);
    }

    @Override
    public void buildUI(JPanel panel) {
        panel.setBackground(Color.WHITE);
        panel.setLayout(new GridBagLayout());

        JLabel label = new JLabel(ResourceSupport.
getDefaultResourceBundle().getString("daemon view"));
        textField = new JTextField("Hello, world");
        textField.setPreferredSize(new Dimension(200, 32));
        startButton = new JButton(ResourceSupport.
getDefaultResourceBundle().getString("start_daemon"));
        stopButton = new JButton(ResourceSupport.
getDefaultResourceBundle().getString("stop_daemon"));

        textField.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                SwingService.keyboardService.showLetterKeyboard(
textField, null, false, new BaseKeyboardCallback() {
                    @Override
                    public void onOk(Object o) {
                        try {
                            xmlRpcMyDaemonFacade.setMessage(
textField.getText());
                        } catch (XmlRpcException ex) {
                            ex.printStackTrace();
                        }
                    }
                });
            }
        });

```

```
40              startButton.addActionListener(e-> startDaemon());
41
42              startButton.addActionListener(e-> stopDaemon());
43
44              addComponent(panel, label, 0, 0, 2, 1, 0.0D, 0.0D, 0, 0, 0,
    0, 1, 10);
45              addComponent(panel, textField, 0, 1, 2, 1, 0.0D, 0.0D, 20,
    0, 0, 0, 1, 10);
46              addComponent(panel, startButton, 0, 2, 1, 1, 0.0D, 0.0D, 20,
     0, 0, 0, 3, 17);
47              addComponent(panel, stopButton, 1, 2, 1, 1, 0.0D, 0.0D, 20,
    0, 0, 0, 3, 13);
48          }
49
50          public static void addComponent(JPanel panel, Component c, int x
    , int y, int width, int height, double wx, double wy, int top, int
    left, int bottom, int right, int fill, int anchor) {
51              GridBagConstraints gridBagConstraints = new
    GridBagConstraints();
52              gridBagConstraints.gridx = x;
53              gridBagConstraints.gridy = y;
54              gridBagConstraints.gridwidth = width;
55              gridBagConstraints.gridheight = height;
56              gridBagConstraints.weightx = wx;
57              gridBagConstraints.weighty = wy;
58              gridBagConstraints.anchor = anchor;
59              gridBagConstraints.fill = fill;
60              gridBagConstraints.insets = new Insets(top, left, bottom,
    right);
61              panel.add(c, gridBagConstraints);
62          }
63
64          public void updateStatus() {
65              if (this.daemonService.getDaemon().getState() ==
    DaemonContribution.State.RUNNING) {
66                  startButton.setEnabled(false);
67                  stopButton.setEnabled(true);
68                  textField.setEnabled(true);
69              } else {
70                  startButton.setEnabled(true);
71                  stopButton.setEnabled(false);
72                  textField.setEnabled(false);
73              }
74          }
75
```

```
76          public void startDaemon() {
77              System.out.println("my daemon start!");
78              daemonService.getDaemon().start();
79              updateStatus();
80          }
81
82          public void stopDaemon() {
83              System.out.println("my daemon stop!");
84              daemonService.getDaemon().stop();
85              updateStatus();
86          }
87
88          @Override
89          public void openView() {
90              updateStatus();
91              startDaemon();
92          }
93
94          @Override
95          public void closeView() {
96              stopDaemon();
97          }
98      }
```

代码块 7.4: MyDaemonViewImpl 文件内容

由上代码块7.3及代码块7.4可知，myDaemon 导航栏贡献视图中输入框输入完成事件中通过 XmlRpc 客户端向 xmlRpc 服务器请求调用 setMessage 方法，启动/停止按钮用于启动/停止 Daemon 程序。

同时进入/关闭 MyDaemon 导航栏贡献视图会主动启动/停止 Daemon 程序。

7.4 XmlRpc 客户端

由前文可知，myDaemon 需要通过 XmlRpc 客户端向 XmlRpc 服务器请求调用方法，因此实现 XmlRpc 配置类如下代码块7.5所示：

```
1       public class XmlRpcMyDaemonFacade {
2           private final XmlRpcClient client;
3
4           public XmlRpcMyDaemonFacade(String host, int port) {
5               XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl()
    ;
```

```
            config.setEnabledForExtensions(true);
            try {
            config.setServerURL(new URL("http://" + host + ":" + port +
    "/RPC2"));
        } catch (MalformedURLException e) {
          e.printStackTrace();
        }
        //1s
        config.setConnectionTimeout(1000);
        client = new XmlRpcClient();
        client.setConfig(config);
      }

      public String setMessage(String mes) throws XmlRpcException {
        ArrayList<String> args = new ArrayList<String>();
        args.add(mes);
        Object result = client.execute("set_message", args);
        return processString(result);
      }

      private String processString(Object response) {
        if (response instanceof String) {
          return (String) response;
        } else {
          return "";
        }
      }
    }
```

代码块 7.5: XmlRpcMyDaemonFacade 文件内容

7.5 Daemon 程序

按照 myDaemon 项目功能设计所述，Daemon 程序 server.py 内容如下代码块7.6所示:

```
#!/usr/bin/env python3

import time
import sys
import _thread
import re

```

```
    from xmlrpc.server import SimpleXMLRPCServer
    from http.server import BaseHTTPRequestHandler, HTTPServer

    message ='Hello, world'

    def set_message(mes):
      global message
      message = mes
      print ("message from client: " + mes)
      return message

    class RequestHandler(BaseHTTPRequestHandler):
      '''处理请求并返回页面'''
      global message

      # 页面模板
      Page = '''\
        <html>
        <body>
        <p>{message} </p>
        </body>
        </html>
      '''

      # 处理一个GET请求
      def do_GET(self):
        print ("do_GET")
        print ("do_GET: " + message)
        mes = re.sub(r'\{message\}', self.Page, message)
        self.send_response(200)
        self.send_header("Content-Type", "text/html")
        self.send_header("Content-Length", str(len(mes)))
        self.end_headers()
        self.wfile.write(mes.encode('utf-8'))

    def rpc_server(threadName, delay):
      sys.stdout.write("MyDaemon daemon started")
      sys.stderr.write("MyDaemon daemon started")
      server = SimpleXMLRPCServer(("127.0.0.1", 4444))
      server.register_function(set_message, "set_message")
      server.serve_forever()

    def http_server(threadName, delay):
      serverAddress = ('127.0.0.1', 5555)
      server = HTTPServer(serverAddress, RequestHandler)
```

```
53      server.serve_forever()
54
55
56    if __name__ == '__main__':
57      # 创建两个线程
58      try:
59        _thread.start_new_thread( http_server, ("Thread-1", 1, ) )
60        _thread.start_new_thread( rpc_server, ("Thread-2", 2, ) )
61      except:
62        print ("Error: 无法启动线程")
63      while 1:
64        pass
```

代码块 7.6: server.py 文件内容

server.py 提供 2 个服务：

- XMLRPCServer: 提供本地 RPC 服务，端口为 4444。
  主要方法 set_message(mes)，RPC 客户端可以连接该服务通过调用该方法设置一条消息。
- HTTPServer: 提供本地 web 服务，端口为 5555。
  实现了一个 get 请求，用来返回 RPC 服务中设置的消息内容。

最后，将定制完成的 MyDaemonNavbarServiceImpl 类及 MyDaemonServiceImpl 类在 Activator 中注册，如下代码块**??**代码中第 6-7 行所示。

```
1    public class Activator implements BundleActivator {
2        private ServiceRegistration<DaemonService> registration;
3        private ServiceReference<LocaleProvider>
     localeProviderServiceReference;
4
5        @Override
6        public void start(BundleContext bundleContext) throws Exception
     {
7            localeProviderServiceReference = bundleContext.
     getServiceReference(LocaleProvider.class);
8            if (localeProviderServiceReference != null) {
9                LocaleProvider localeProvider = bundleContext.getService
     (localeProviderServiceReference);
10               if (localeProvider != null) {
11                   ResourceSupport.setLocaleProvider(localeProvider);
12               }
13           }
14           System.out.println("cn.elibot.plugin.example.myDaemon.impl.
     Activator says Hello World!");
```

```
15              MyDaemonServiceImpl daemonService = new MyDaemonServiceImpl
    ();
16              registration = bundleContext.registerService(DaemonService.
    class, daemonService, null);
17              bundleContext.registerService(NavbarService.class, new
    MyDaemonNavbarServiceImpl(daemonService), null);
18          }
19
20          @Override
21          public void stop(BundleContext bundleContext) throws Exception {
22              System.out.println("cn.elibot.plugin.example.myDaemon.impl.
    Activator says Goodbye World!");
23              this.registration.unregister();
24          }
25      }
```

代码块 7.7: Activator 文件内容

完成注册后，即可按照第 3 章 ELITECO Plugin 项目前期中的构建部署流程，将其安装到 EliRobot 机器人平台下。

myDaemon 导航栏贡献视图如**图** 7-2所示。

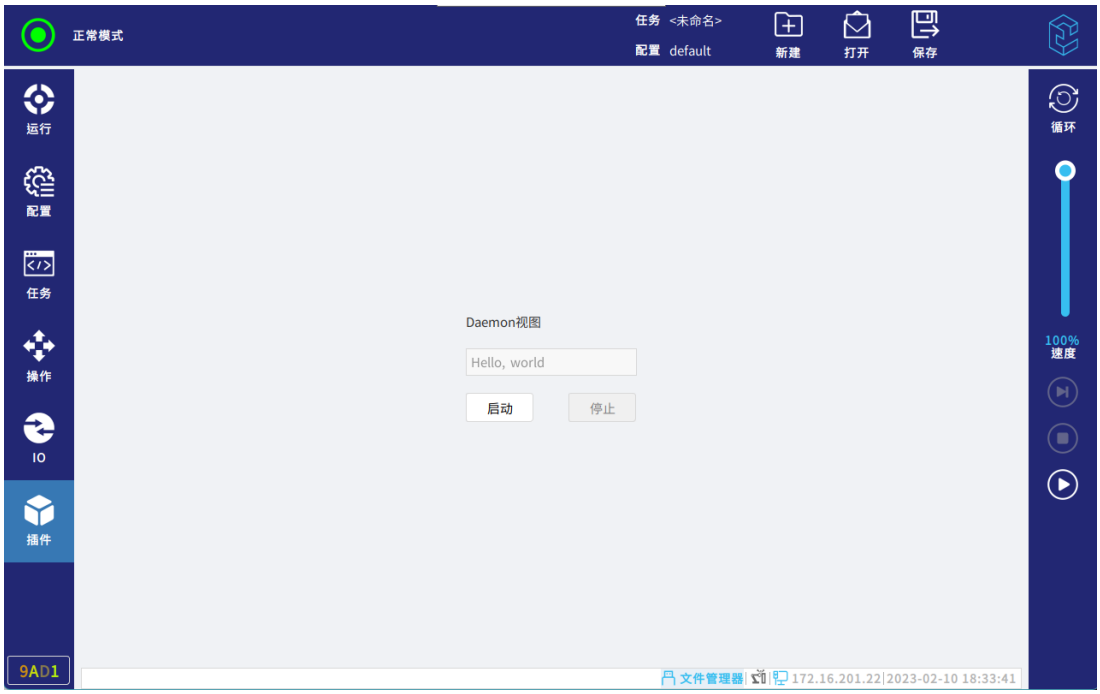


图 7-2: myDaemon 导航栏贡献视图

在文本输入框中输入消息内容。例如如下**图** 7-3所示输入“Hello, test message!”。

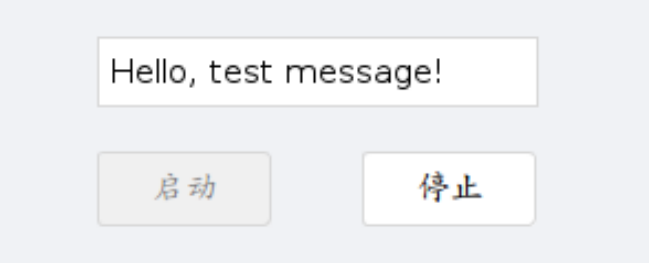


图 7-3: MyDaemon 导航栏贡献视图

此时打开浏览器输入“http://localhost:5555/”或“http://127.0.0.1:5555/”，web 服务器响应 get 请求，输出消息，内容即如**图** 7-4所示的：

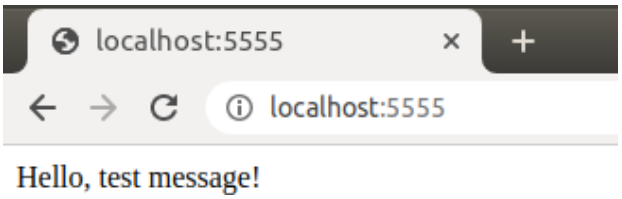


图 7-4: web 服务器响应 get 请求输出消息

显示内容与输入内容一致表示成功。如果浏览器已打开该页面后重新输入了消息内容，则需要按“F5”刷新一下浏览器。

第 8 章 其他功能特性

8.1 变量

EliRobot 机器人平台中变量主要有三类：配置变量、坐标系变量及任务变量三种类型变量，除会被写入到脚本、脚本中完成赋值及需要保证名称唯一等共同特征外，这三种变量还有各自不同的特征：配置变量由配置模块中定义并管理，生命周期跟随该当前配置文件；坐标系变量由配置模块中定义并管理，生命周期跟随该当前配置文件，值仅支持位姿数据；任务变量由当前任务定义并管理，生命周期跟随当前任务文件，使用范围仅限当前任务。

开发者可以在 ELITECO Plugin 中进行对任务变量和配置变量的增删查操作及对坐标系变量的查询操作：通过 VariableService 接口类 (如代码块8.1所示) 获取所需变量实现查询操作，继而通过变量实例的公共基接口 Variable(如代码块8.2所示) 获取变量类型及名称等属性；变量的创建和删除则需通过 VariableService 中的接口获取 VariableFactory 接口类实现 (如代码块8.3所示)。

```
1   public interface VariableService {
2       /**
3        * 获取变量工厂接口类
4        */
5       VariableFactory getVariableFactory();
6
7       /**
8        * 通过过滤器获取变量集合
9        */
10      Collection<Variable> get(Filter<Variable> filter);
11
12      /**
13       * 获取所有配置变量
14       */
15      List<ConfigurationVariable> getConfigurationVariables();
16
17      /**
18       * 获取所有坐标系变量
19       */
20      List<FrameVariable> getFrameVariables();
21
22      /**
23       * 获取指定名称“frame”对应的坐标系变量
```

```
         */
        FrameVariable getFrameVariable(Frame frame);
    }
```

代码块 8.1: VariableService 接口类

```
    public interface Variable {
        /**
         * 获取变量类型
         */
        Type getType();

        /**
         * 获取变量显示名称
         */
        String getDisplayName();

        /**
         * 获取变量用于写入脚本的名称(防止写入脚本乱码，可能会被与显示名称不同)
         */
        String getScriptName();

        enum Type {

            /**
             * 任务变量
             */
            TASK,
            /**
             * 配置变量
             */
            CONFIGURATION,
            /**
             * 坐标系变量
             */
            FRAME;

            Type() {
            }
        }
    }
```

代码块 8.2: Variable 接口

```
1   public interface VariableFactory {
2       /**
3        * 以baseName为基础申请一个唯一名称
4        */
5       String makeAUniqueName(String baseName) throws
    PluginEntityNameException;
6
7       /**
8        * 创建一个名为name的任务变量
9        */
10      TaskVariable createTaskVariable(String name) throws
    VariableException;
11
12      /**
13       * 创建一个名为name的任务变量，并使用expression作为其初始值(在生成脚本时
    生效)
14       */
15      TaskVariable createTaskVariable(String name, Expression
    expression) throws VariableException;
16
17      /**
18       * 生成配置变量
19       */
20      ConfigurationVariable createConfigurationVariable(String name,
    String value) throws VariableException;
21
22      /**
23       * 删除配置变量
24       */
25      boolean removeConfigurationVariable(ConfigurationVariable
    variable);
26  }
```

代码块 8.3: VariableFactory 接口类

具体每种变量的操作及使用示例将会在接下来的章节中具体描述。

8.1.1 配置变量

如前文所述，配置变量是由配置模块创建并管理的变量，在当前配置文件生效期间，加载的不同任务都可以访问到当前配置模块中的配置变量，其生命周期跟随当前配置文件，因此一定程度上可以将其视作为全局变量。

在 EliRobot 机器人平台“配置-变量”页签下可以看到当前配置文件管理的配置变量，也可以通过该页面的 UI 组件进行配置变量的增删改管理。

由于配置变量一定程度上可视作全局变量，因此在 ELITECO Plugin 多种模块的功能扩展中都可以配置变量进行增删查操作。下面就 ELITECO Plugin 中配置变量的使用及不同模块功能扩展过程中配置变量相关接口的使用进行详细描述。

1. 配置变量查询

参见前文代码块8.1可知，配置变量的查询可以通过 VariableService 接口类中的 get 和 getConfigurationVariables 两个接口实现，示例代码如代码块8.4所示：

```
1    public Collection<Variable> getConfigurationVariablesByType() {
2        return this.configurationApiProvider.getVariableService().get(
     cell -> cell.getType().equals(Variable.Type.CONFIGURATION));
3    }
4
5    public List<ConfigurationVariable> getConfigurationVariables() {
6        return this.configurationApiProvider.getVariableService().
     getConfigurationVariables();
7    }
```

代码块 8.4: 配置变量查询

通过代码块8.4中所示方法可以获取当前 EliRobot 机器人平台中的全部配置变量，实际使用场景中可以对所获的配置变量集合进一步过滤获取所需的变量，并进一步获取配置变量个体数据。

2. 配置变量查询

如前文所述，配置变量增删需要通过代码块8.3中的 VariableFactory 接口类中的 createConfigurationVariable 和 removeConfigurationVariable 两个接口实现，如代码块8.5所示：

```
1    public void configurationVariableDemo() {
2        String name = "config_var";
3        String initialValue = "demo configuration variable";
4        ConfigurationVariable configurationVariable =
     createConfigurationVariable(name, initialValue);
5        if (configurationVariable != null) {
```

```
6             System.out.println("configuration variable create success.")
    ;
7             boolean removeResult = removeConfigurationVariable(
    configurationVariable);
8             if (removeResult) {
9                 System.out.println("configuration variable remove
    success.");
10            }else{
11                System.out.println("configuration variable remove failed
    .");
12            }
13        }else{
14            System.out.println("configuration variable create failed.");
15        }
16    }
17
18    public ConfigurationVariable createConfigurationVariable(String
    baseName, String initialValue) {
19        ConfigurationVariable configurationVariable = null;
20        try {
21            this.configurationApiProvider.getVariableService().
    getVariableFactory().createConfigurationVariable(baseName,
    initialValue);
22        } catch (VariableException e) {
23            SwingService.messageService.showMessage("Error", e.
    getMessage(), MessageType.ERROR);
24        }
25
26        return configurationVariable;
27    }
28
29    public boolean removeConfigurationVariable(ConfigurationVariable
    toBeRemoved) {
30        return this.configurationApiProvider.getVariableService().
    getVariableFactory().removeConfigurationVariable(toBeRemoved);
31    }
```

代码块 8.5: 配置变量增删

代码块8.5中创建了一个名为 “config_var”，初始化值为 “demo configuration variable” 的配置变量，创建成功/失败都会打印结果消息，若创建成功则会删除该变量并打印结果消息。需注意的是，由于变量需要保持命名唯一且符合 “大小写字母数字下划线组合长度不超过 14 个字符且必须大小写字母开头” 要求，因此会出现因名称不合法会导致创建失败抛出 VariableException 异常，因此需要注意捕获异常。

另外由代码块8.5可知配置变量在创建时就需要进行赋值，因此通过 VariableService 接口访问的配置变量都有 getValue 接口 (如下代码块8.6所示) 可用于获取其当前值。

```
1    public interface ConfigurationVariable extends PersistedVariable{
2        /**
3         * 返回变量的值
4         *
5         * @return 如果引用的配置变量不错在，则返回null
6         */
7        String getValue();
8    }
```

代码块 8.6: ConfigurationVariable 接口类

配置变量一经创建，除主动删除及放弃保存切换配置文件外，该配置变量会一直在该配置中存续，因此建议在 ELITECO Plugin 定制开发中注意配置变量的声明周期，使用完毕的配置变量要及时删除。

配置变量的增删查接口在定制配置节点、任务节点及导航栏贡献的相关类中都可以使用，因此这里简要说明一下几种贡献定制过程中使用配置变量的增删查接口的方法。

配置节点定制过程中主要通过配置节点贡献中的 ConfigurationAPIProvider 实例获取 VariableService 接口类实例，因此建议参考代码块4.6第 22 行将 ConfigurationAPIProvider 实例作为配置节点贡献类构造方法的参数传递到配置节点贡献类中；在配置节点参数视图中则需先通过 ConfigurationViewAPIProvider 实例获取 ConfigurationAPIProvider 实例，继而获取 VariableService 接口类实例，因此建议参考代码块4.6第 15 行将 ConfigurationViewAPIProvider 或 ConfigurationAPIProvider 实例作为配置节点参数视图类构造方法的参数传递到配置节点贡献类中。

任务节点定制过程中则类似的分别通过任务节点贡献中的 TaskApiProvider 实例及任务节点参数视图中 TaskNodeViewApiProvider 实例直接或间接的获取 VariableService 接口类实例，因此建议参考代码块5.6第 28 及 23 行将 TaskApiProvider 实例及 TaskNodeViewApiProvider 实例分别作为任务节点贡献类及任务节点参数视图类构造方法的参数。

在导航栏贡献定制过程中则有所不同，需要通过 NavbarContext 实例获取 NavbarApiProvider 实例继而获取 VariableService 接口类实例实现对配置变量的增删查，因此建议参考代码块6.2第 14 及 19 行保存 configure 方法中的 NavbarContext 实例并将其作为导航栏贡献类构造方法的参数传递到导航栏贡献类中。

表 8-1. VariableService 接口类实例获取方式

定制类型	VariableService 接口类获取
配置节点	配置节点贡献中 ConfigurationAPIProvider 接口类直接获取或配置点参数视图中 ConfigurationViewAPIProvider 接口类间接获取
任务节点	任务节点贡献中 TaskApiProvider 接口类直接获取或任务节点参数视图中 TaskNodeViewApiProvider 接口类间接获取
导航栏贡献	导航栏服务中 NavbarContext 接口类直接获取 (建议作为构造方法参数传递到导航栏贡献中)

8.1.2 任务变量

如本章节开始所述，任务变量具有任务中定义并管理、使用范围仅限当前任务、名称唯一及运行任务时会被写入脚本用做脚本变量特征，这些特征决定了任务变量的其作用、使用场景和使用方法。下面通过示例来讲述任务变量的使用方法和使用场景。

1. 任务变量查询

参见前文代码块8.1可知，配置变量的查询可以通过 VariableService 接口类中的 get 接口实现，示例代码如代码块8.7所示：

```
1    public Collection<Variable> getTaskVariables() {
2      return this.taskApiProvider.getVariableService().get(variable ->
     variable.getType().equals(Variable.Type.TASK);
3    }
```

代码块 8.7: 任务变量查询

通过代码块8.7中所示方法可以获取当前任务中注册的全部任务变量，实际使用场景中可以对所获的配置变量集合进一步过滤获取所需的变量，并进一步获取配置变量个体数据。

2. 任务变量创建

如前文所述，变量创建需要通过代码块8.3中的 VariableFactory 接口类中的 createTaskVariable 接口实现，示例代码如代码块8.8所示：

```
1    public TaskVariable createGlobalVariable(String baseName) {
2        TaskVariable variable = null;
3        try{
4            String uniqueName = this.variableService.getVariableFactory
     ().makeAUniqueName(baseName);
5            variable = this.variableService.getVariableFactory().
     createTaskVariable(uniqueName, this.expressionService.
     createExpressionConstructor().appendTokenCell("0", "0").construct())
     ;
6        } catch (PluginEntityNameException | VariableException |
     IllegalExpressionException e) {
7            SwingService.messageService.showMessage("Error", e.
     getMessage(), MessageType.ERROR);
8        }
9
10       return variable;
11   }
```

代码块 8.8: 任务变量创建

任务变量需要保持命名唯一且符合“大小写字母数字下划线组合长度不超过 14 个字符且必须大小写字母开头”要求，因此会出现因名称不合法会导致创建失败抛出 VariableException 异常，因此需要注意捕获异常。

为保证任务变量创建成功，代码块8.8中通过 VariableFactory 接口类实例的 makeAUniqueName 接口在系统中以 baseName 为基础申请了一个唯一的合法名称，但需注意 baseName 为 null 或者空字符串时，会抛出 PluginEntityNameException 异常，因此需要注意捕获异常。

代码块8.3中可知，有两个 createTaskVariable 重载方法，二者区别在于是否传递变量初始化表达式参数，表达式创建相关参见第 5.5.6 小节 变量表达式。任务变量的初始化表达式由任务变量管理无需关注持久化问题，同时在生成脚本时变量初始化区被写入变量赋初值语句，但是任务变量本身并不会保存任务变量的实时值，因此任务变量没有类似配置变量的 getValue 接口可以获取实时值。任务变量的实时值在任务运行时可通过“任务-监控-变量”页面查看，也仅在任务运行时任务变量的实时值才有意义。

任务变量一经创建成功，则同时将其进行名称注册，则保证系统中不会有其他的命名实体再申请到该名称，出现名称冲突。但需注意，由于任务模块特殊性，会动态更新注册状态，对于定制任务节点中的任务变量规定刷新时当前任务节点在任务树上且经其节点贡献节点数据持久化句柄存储的任务变量会保持其注册状态，否则删除该注册信息。因此建议创建的任务变量通过**表** 5-1中的 TaskNodeDataModelWrapper 节点数据持久化句柄实例存储变量，以防止其他地方创建重名命名实体。

当任务变量不再使用时，将其通过 TaskNodeDataModelWrapper 实例从数据中删除即可，下一次更新任务变量注册状态时会将其注册信息删除，不需要专有的接口去删除。

8.1.3 坐标系变量

同配置变量类似，坐标系变量是由配置模块创建并管理的一种变量，在当前配置文件生效期间，加载的不同任务都可以访问到当前配置模块中的坐标系变量，其生命周期跟随当前配置文件，因此一定程度上可以将其视作为全局变量。

与配置变量不同的是,坐标系变量的值是一个坐标系 Frame 对象,并且由于坐标系 Frame 相对于普通数据略复杂，因此 ELITECO plugin 中仅支持查询操作，不支持坐标系变量创建和删除。下面就 ELITECO Plugin 中坐标系变量的使用进行详细描述。

参见前文代码块8.1可知，坐标系变量的查询可以通过 VariableService 接口类中的多个接口实现，示例代码如代码块8.9所示：

```
1     public Collection<Variable> getFrameVariablesByType() {
2         return this.configurationApiProvider.getVariableService().get(
    cell -> cell.getType().equals(Variable.Type.FRAME));
3     }
4
5     public List<FrameVariable> getFrameVariables() {
6         return this.configurationApiProvider.getVariableService().
    getFrameVariables();
7     }
8
9     public FrameVariable getFrameVariable(Frame frame) {
10        return this.configurationApiProvider.getVariableService().
    getFrameVariable(frame);
11    }
```

代码块 8.9: 坐标系变量查询

通过代码块8.9中可知，开发者可通过 VariablesService 接口类中的接口获取 EliRobot 机器人平台中的全部坐标系变量，同时也可以获取给定坐标系对象对应的坐标系变量。

坐标系变量查询接口比较简单，且在 ELITECO Plugin 中不支持主动创建和删除，因此这里不再做过多赘述。至于坐标系 Frame 类将在后续的章节进行描述。

8.2 计量类数据

EliRobot 机器人平台中有较多的如长度、速度、加速度、位姿及转矩等等机器人相关的计量类数据，在使用过程中会涉及到较多涉及单位转换，因此为便于这类计量类数据转换单位，ELITECO Plugin 中开放了较多的计量类数据类型及 ValueFactory，用于开发者处理相关类型的数据。

ValueFactory 接口类中部分接口如下代码块8.10所示，ValueFactory 接口类全部接口参见接口文档或 ELITECO SDK API 中 ValueFactory 接口类源码。

```
1   public interface ValueFactory {
2       /**
3        * 创建Length对象。
4        */
5       Length createLength(double value, Length.Unit unit);
6
7       /**
8        * 创建Position对象。
9        */
10      Position createPosition(double x, double y, double z, Length.
    Unit unit);
11
12      /**
13       * 创建Rotation对象。
14       */
15      Rotation createRotation(double rx, double ry, double rz, Angle.
    Unit unit);
16
17      ...
18
19      /**
20       * 创建Pose对象。
21       */
22      Pose createPose(double x, double y, double z, double rx, double
    ry, double rz, Length.Unit lengthUnit, Angle.Unit angleUnit);
23  }
```

代码块 8.10: ValueFactory 接口类

ValueFactory 接口类覆盖了几乎全部机器人相关的数据类型的创建接口，如下代码块8.11所示为 ValueFactory 接口类中 Length 及 Pose 的创建及单位转换示例，其他数据类型类似，不再单独示例。

```
1    public void valueFactoryDemo() {
2        ValueFactory valueFactory = this.taskApiProvider.getValueFactory
     ();
3
4        Length length = valueFactory.createLength(101.1, Length.Unit.MM)
     ;
5        double length_M = length.getAs(Length.Unit.M);
6
7        Pose pose = valueFactory.createPose(92, -140.48, 492.22, 3.13,
     0, -1.571, Length.Unit.MM, Angle.Unit.RAD);
8        double rz_deg = pose.getRotation().getRz(Angle.Unit.DEG);
9    }
```

代码块 8.11: Length 及 Pose 使用示例

代码块8.11可知，计量类数据类型使用比较简单，不再过多赘述。需注意的是在 ValueFactory 接口类实例的获取方式同第 8.1 节 变量中 VariableService 接口类获取方式一样，可参考**表** 8-1或其上方文字描述。

8.3 I/O

EliRobot 机器人平台中多种类型的 I/O，ELITECO Plugin 中支持对这些 I/O 的查询及设置，以便于开发者在 ELITECO Plugin 中能够根据需要实时设置/获取 I/O 相关信息。

ELITECO Plugin 中支持的 I/O 主要分类数字 I/O(DigitalI/O)、模拟 I/O(AnalogI/O)、整型寄存器 (IntegerRegister)、布尔寄存器 (BooleanRegister) 及浮点型寄存器 (FloatRegister)，几种 I/O 类型有公共基接口类 I/O，可以用于获取 I/O 的一部分诸如名称、类型及字符串值等基础数据，I/O 接口类内容如代码块8.12所示。

```
1    public interface IO {
2    /**
3    获取I/O名称
4    */
5    String getName();
6
7    /**
8    获取I/O默认名称
9    */
10   String getDefaultName();
11
12   /**
```

```
    获取I/O数据类型
    */
    IOType getType();

    /**
    获取I/O类型
    *
    @return interface type
    */
    InterfaceType getInterfaceType();

    /**
    获取当前I/O是否是输入I/O
    */
    boolean isInput();

    /**
    获取I/O数据的字符串形式
    */
    String getValueStr();

    /**
    获取I/O是否可解析
    */
    boolean isResolvable();

    enum InterfaceType {
    /**
    标准I/O，工具I/O，可配置I/O，MODBUS I/O，通用寄存器
    */
    STANDARD, TOOL, CONFIGURABLE, MODBUS, GENERAL_PURPOSE
    }

    enum IOType {
    /**
    数字I/O，模拟I/O，整型寄存器，布尔寄存器，浮点型寄存器
    */
    DIGITAL, ANALOG, INTEGER, BOOLEAN, FLOAT
    }
    }
```

代码块 8.12: I/O 接口类

如代码块8.12所示，I/O 根据用途又分为标准 I/O、工具 I/O、可配置 I/O、Modbus I/O 及通用寄存器。这些 I/O 分类中数字 I/O、模拟 I/O 及 Modbus I/O 又有单独的接口支持进

一步查询/设置其数据，如代码块8.13-代码块8.15所示。

```
public interface DigitalIO extends IO {
    /**
     * 设置当前数字I/O值
     */
    boolean setValue(boolean value);

    /**
     * 获取当前数字I/O值
     */
    boolean getValue();
}
```

代码块 8.13: Digital I/O

```
public interface AnalogIO extends IO {
    /**
     * 获取数值范围下限
     */
    double getMinRangeValue();

    /**
     * 获取数值范围上限
     */
    double getMaxRangeValue();

    /**
     * 获取当前模拟I/O是否是表征电流
     */
    boolean isCurrent();

    /**
     * 获取当前模拟I/O是否是表征电压
     */
    boolean isVoltage();

    /**
     * 设置当前I/O值
     */
    boolean setValue(double value);

    /**
```

```
28          * 获取当前I/O值
29          */
30         double getValue();
31     }
```

代码块 8.14: Analog I/O

```
1      public interface ModbusIO extends IO {
2          /**
3           * 获取Modbus ip地址
4           */
5          String getIpAddress();
6
7          /**
8           * 获取Modbus 信号地址
9           */
10         String getSignalAddress();
11
12         /**
13          * 设置信号值
14          */
15         boolean setValue(int value);
16
17         /**
18          * 获取信号值
19          */
20         int getValue();
21     }
```

代码块 8.15: Modbus I/O

由代码块8.13-代码块8.15可知，由于 I/O 类型本身特性，Digital I/O、Analog I/O 及 Modbus I/O 都支持查询/设置值，但三者值类型不同，同时 Analog I/O 又允许查询当前 I/O 值的电流和电压属性及值的范围，而 Modbus I/O 则允许查询 ip 地址即信号地址。

在 ELITECO Plugin 中通过 I/OModel 接口类可以方便以多种方式获取当前 EliRobot 机器人平台中的各种 I/O，I/OModel 接口类如代码块8.16所示。

```
1      public interface IOModel {
2          /**
3           * 获取全部I/O
4           */
5          Collection<IO> getIOs();
```

```
6
7           /**
8            * 获取指定类型全部I/O
9            */
10          <T extends IO> Collection<T> getIOs(Class<T> clazz);
11
12          /**
13           * 使用过滤器获取指定I/O集合
14           */
15          Collection<IO> getIOs(Filter<IO> filter);
16      }
```

代码块 8.16: I/OModel

由代码块 8-16 可以看到，I/OModel 允许直接获取全部 I/O、通过反射获取指定类型 I/O 及通过过滤器获取指定 I/O，这三个接口都比较简单，不在做过多描述。不过，为方便开发者快速创建用于过滤指定类型 I/O 的 I/O 过滤器，ELITECO SDK API 中提供了 I/OFilterFactory 接口类用于开发者快速通过其中的静态方法创建指定类型的 I/O 过滤器。I/OFilterFactory 接口类如代码块8.17所示 (有缩略，详见接口文档或 ELITECO SDK API 接口源码)。

```
1   public class IOFilterFactory {
2       /**
3        * 创建数字I/O过滤器
4        */
5       public static Filter<IO> createDigitalFilter() {
6           return element -> element.getType() == IOType.DIGITAL;
7       }
8
9       /**
10       * 创建整型寄存器I/O过滤器
11       */
12      public static Filter<IO> createIntegerFilter() {
13          return element -> element.getType() == IOType.INTEGER;
14      }
15
16      /**
17       * 创建模拟I/O过滤器
18       */
19      public static Filter<IO> createAnalogFilter() {
20          return element -> element.getType() == IOType.ANALOG;
21      }
22
23      /**
24       * 创建工具I/O过滤器
```

```
25          */
26         public static Filter<IO> createToolFilter() {
27             return element -> element.getInterfaceType() ==
    InterfaceType.TOOL;
28         }
29
30         ...
31     }
```

代码块 8.17: IOFilterFactory

定制配置节点、任务节点及导航栏贡献的相关类中都可以 I/OModel 获取 I/O 进行相关的逻辑处理，I/OModel 的获取方式类似于变量服务接口 VariableService，可参见**表** 8-1中 VariableService 的获取方式。

如下代码块8.18所示为 I/O 相关接口简单使用示例，示例中主要涉及了 I/O 的几种获取方式及基本接口的使用，开发者在 ELITECO Plugin 中进行 I/O 相关逻辑功能开发过程中可以参考。

```
1     public Collection<IO> getAllIOsDemo() {
2         IOModel ioModel = this.taskNodeViewApiProvider.
    getTaskApiProvider().getIOModel();
3         return ioModel.getIOs();
4     }
5
6     public Collection<AnalogIO> getAnalogIOsDemo() {
7         IOModel ioModel = this.taskNodeViewApiProvider.
    getTaskApiProvider().getIOModel();
8         return ioModel.getIOs(AnalogIO.class);
9     }
10
11    public SetNode createAnalogOutputVoltageSetNodeDemo(Voltage voltage)
     {
12        if (voltage != null) {
13            TaskNodeFactory factory = this.taskApiProvider.getTaskModel
    ().getTaskNodeFactory();
14            SetNode analogOutVSet = factory.createSetNode();
15            SetNodeConfigFactory setNodeConfigFactory = analogOutVSet.
    getConfigFactory();
16
17            List<IO> listAO = new ArrayList<>(this.taskApiProvider.
    getIOModel().getIOs(IOFilterFactory.createAnalogOutputFilter()));
18            IO out_1 = listAO.get(0);
19            AnalogIO analogIO_1 = (AnalogIO) out_1;
```

```
20              if (analogIO_1.isVoltage()) {
21                  analogOutVSet.setConfig(setNodeConfigFactory.
    createAnalogOutputVoltageConfig(analogIO_1, voltage));
22                  return analogOutVSet;
23              }
24          }
25
26          return null;
27      }
```

代码块 8.18: I/O 相关接口简单使用示例

以上仅对 ELITECO Plugin 中 I/O 获取方式、主要接口及相关接口使用方法进行了描述及示例，未对 I/O 功能本身、其使用场景及方法进行描述，如有该类需要，可参考 EliRobot 使用文档。

8.3.1 工具 I/O 锁

前文描述了 ELITECO Plugin 中的 I/O 根据用途又分为标准 I/O、工具 I/O 及可配置 I/O 等，其中工具 I/O 用来配置与工具通信，具体配置及使用可参考 EliRobot 使用文档。

为防止当前工具 I/O 配置不被其他操作人员无意间篡改，保证当前工具通信安全，ELITECO Plugin 中允许开发者通过实现 ToolIoLockerInterface 接口创建工具 I/O 锁注册到 EliRobot 中。注册到 EliRobot 中的 I/O 锁会在“配置-工具 I/O”页面“I/O 接口控制”栏工具 I/O 锁列表中展示出来。ToolIoLockerInterface 接口类如代码块8.19所示。

```
1       public interface ToolIoLockerInterface {
2           /**
3            * 当前工具I/O锁被激活事件
4            */
5           void onLockGranted(ToolIoInterfaceLockerEvent
    toolIoInterfaceLockerEvent);
6
7           /**
8            * 当前工具I/O锁被取消事件
9            */
10          void onLockToBeRevoked(ToolIoInterfaceLockerEvent
    toolIoInterfaceLockerEvent);
11      }
```

代码块 8.19: ToolIoLockerInterface

由代码块8.19可以看到ToolIoLockerInterface接口类有两个接口,分别是onLockGranted工具I/O锁被激活事件接口及onLockToBeRevoked工具I/O锁被取消事件。当工具I/O锁被激活时，当前工具I/O是不允许通过GUI显示设置，唯有切换到默认的“用户”模式工具I/O数据才可以被设置，以防止工具I/O数据被篡改，保证当前工具通信安全。

通过工具I/O锁事件接口参数ToolIoInterfaceLockerEvent接口类的getInterfaceLockerResource接口可以得到用于设置工具I/O配置的ToolIoInterfaceLocker接口类如代码块8.20所示。

```
public interface ToolIoInterfaceLocker {
    /**
     * Whether to be controlled.
     *
     * @return if ture Have control else false NO Control
     */
    boolean hasLocked();

    /**
     * 设置工具输出电压
     */
    void setToolVoltage(ToolVoltage toolVoltage);

    /**
     * 设置工具通用配置
     */
    void setToolCommConfig(ToolCommConfig toolCommConfig);

    /**
     * 设置工具模拟I/O工作模式(ToolAnalogIoWorkMode )
     */
    void setToolAnalogIoWorkMode(ToolAnalogIoWorkMode
toolAnalogInputWorkMode);

    /**
     * 设置工具数字I/O工作模式(ToolDigitalIoWorkMode )
     */
    void setToolDigitalIoWorkMode(ToolDigitalIoWorkMode
toolDigitalIoWorkMode);

    /**
     * 设置工具模拟输入域
     */
    void setToolAnalogInputDomain(AnalogDomain analogDomain);

```

```
34          /**
35           * 设置工具模拟输出域
36           */
37          void setToolAnalogOutputDomain(AnalogDomain analogDomain);
38
39          /**
40           * 设置工具数字I/O配置
41           */
42          void setToolDigitalIoConfig(int index, ToolDigitalIo
    toolDigitalIo);
43      }
```

代码块 8.20: ToolIoInterfaceLocker

通过 ToolIoInterfaceLocker 中的接口可以对当前工具 I/O 进行配置，其具体含义此处不做过多描述，详见 EliRobot 使用文档。

下面一个简单的工具 I/O 锁示实现例如代码块8.21所示。

```
1   public class ToolIoLocker implements ToolIoLockerInterface {
2       private static final ToolDigitalIo[] REQUIRED_TOOL_DIGITAL_IO =
    new ToolDigitalIo[4];
3       private static final ToolDigitalIo[] SHUTDOWN_TOOL_DIGITAL_IO =
    new ToolDigitalIo[4];
4       @Override
5       public void onLockGranted(ToolIoInterfaceLockerEvent
    toolIoInterfaceLockerEvent) {
6           ToolIoInterfaceLocker controllableInstance =
    toolIoInterfaceLockerEvent.getInterfaceLockerResource();
7           controllableInstance.setToolVoltage(ToolVoltage.
    TOOL_VOLTAGE_12V);
8           ToolCommConfig toolCommConfig = new ToolCommConfig(true,
    BaudRate.BAUD_2400, Parity.EVEN, StopBits.TWO, Boolean.TRUE);
9           controllableInstance.setToolCommConfig(toolCommConfig);
10          controllableInstance.setToolAnalogInputDomain(AnalogDomain.
    VOLTAGE);
11          controllableInstance.setToolAnalogOutputDomain(AnalogDomain.
    VOLTAGE);
12          controllableInstance.setToolAnalogIoWorkMode(
    ToolAnalogIoWorkMode.USART_MODE);
13          controllableInstance.setToolDigitalIoWorkMode(
    ToolDigitalIoWorkMode.DUAL_PIN_MODE_1);
14          for (int i = 0; i < REQUIRED_TOOL_DIGITAL_IO.length; i++) {
15              REQUIRED_TOOL_DIGITAL_IO[i] = new ToolDigitalIo();
```

```
16                 controllableInstance.setToolDigitalIoConfig(i,
    REQUIRED_TOOL_DIGITAL_IO[i]);
17             }
18         }
19
20         @Override
21         public void onLockToBeRevoked(ToolIoInterfaceLockerEvent
    toolIoInterfaceLockerEvent) {
22             ToolIoInterfaceLocker controllableInstance =
    toolIoInterfaceLockerEvent.getInterfaceLockerResource();
23             controllableInstance.hasLocked();
24             controllableInstance.setToolVoltage(ToolVoltage.
    TOOL_VOLTAGE_0V);
25             ToolCommConfig toolCommConfig = new ToolCommConfig(false,
    BaudRate.BAUD_115200, Parity.NONE, StopBits.ONE, Boolean.FALSE);
26             controllableInstance.setToolCommConfig(toolCommConfig);
27             controllableInstance.setToolAnalogInputDomain(AnalogDomain.
    CURRENT);
28             controllableInstance.setToolAnalogOutputDomain(AnalogDomain.
    CURRENT);
29             controllableInstance.setToolAnalogIoWorkMode(
    ToolAnalogIoWorkMode.ANALOG_IN_AND_OUT);
30             controllableInstance.setToolDigitalIoWorkMode(
    ToolDigitalIoWorkMode.SINGLE_PIN_MODE);
31             for (int i = 0; i < SHUTDOWN_TOOL_DIGITAL_IO.length; i++) {
32                 SHUTDOWN_TOOL_DIGITAL_IO[i] = new ToolDigitalIo();
33                 controllableInstance.setToolDigitalIoConfig(i,
    SHUTDOWN_TOOL_DIGITAL_IO[i]);
34             }
35         }
36     }
```

代码块 8.21: ToolIoLockerInterface 示例

代码块8.21中在工具锁 I/O 激活及取消事件中都对工具 I/O 数据进行了设置，以保证工具使用过程中、工具 I/O 锁之间切换及切换回“用户”模式过程中工具设备及通信安全。需要注意的是工具 I/O 锁 ToolIoeLockerInterface 实现类需要注册到 EliRobot 系统中才能生效，其注册方式需要类似8-1获取 VariableService 方式获取 ToolIoLockerModel 接口类实例来注册 ToolIoeLockerInterface 实现如代码块8.22所示。

```
1     private void registerToolIOLocker() {
2       toolIOLocker = new ToolIoLocker();
3       ToolIoLockerModel toolIoLockerModel = this.
    configurationApiProvider.getToolIoLockerModel();
```

```
4       toolIoLockerModel.requestLocker(toolIOLocker);
5   }
```

代码块 8.22: 注册 ToolIoeLockerInterface 实现

8.4 工具

ELITECO SDK API 允许开发者在 ELITECO Plugin 中对 TCP 进行诸如查询、创建、设置及删除等操作，以便于在能够在 ELITECO Plugin 中完成涉及 TCP 的复杂逻辑。EliRobot 中 TCP 表征工具中心点相对于机器人法兰盘的位姿，因此主要包括表征该偏移的位姿及唯一名称，其接口类如代码块8.23所示。

```
1   public interface TCP {
2       /**
3        * 获取tcp相对于机器人法兰盘的偏移位姿
4        */
5       Pose getOffset();
6
7       /**
8        * 获取tcp唯一名称
9        */
10      String getName();
11
12      /**
13       * 获取tcp显示名称
14       */
15      String getDisplayName();
16
17      /**
18       * 判断tcp是否可以解析
19       */
20      boolean isResolvable();
21  }
```

代码块 8.23: TCP

类似8-1获取 VariableService 方式可以获取 TCPModel 实例，继而获取 EliRobot 中的全部 TCP，TCPModel 接口如代码块8.24所示。

```
1    public interface TCPModel {
2        /**
3         * 获取全部TCP集合
4         */
5        Collection<TCP> getTCPs();
6    }
```

代码块 8.24: TCPModel

其他的对 TCP 诸如创建、更新及删除等操作有赖于 TCPContributionModel 实例，TCP-ContributionModel 接口如代码块8.25所示。

```
1    public interface TCPContributionModel {
2        /**
3         * 创建TCP
4         */
5        TCP addTCP(String tcpName, Pose tcp);
6
7        /**
8         * 获取指定名称TCP
9         */
10       TCP getTCP(String tcpName);
11
12       /**
13        * 更新指定名称TCP位姿
14        */
15       void updateTCP(String tcpName, Pose tcp);
16
17       /**
18        * 删除TCP
19        */
20       void removeTCP(String tcpName);
21   }
```

代码块 8.25: TCPContributionModel

如代码块8.25所示，TCPContributionModel 中接口比较简单，不做过多描述。但需注意的是，TCPContributionModel 实例的获取方式 TCPModel 略有不同，其在不同模块获取方式如**表** 8-2所示。

表 8-2. TCPContributionModel 实例获取方式

定制类型	TCPContributionModel 接口类获取
配置节点	配置节点贡献中 ConfigurationAPIProvider 接口类直接获取或配置点参数视图中 ConfigurationViewAPIProvider 接口类间接获取
任务节点	任务节点贡献中通过 TaskApiProvider 实例获取 ConfigurationAPIProvider 实例间接获取或任务节点参数视图中 TaskNodeViewApiProvider 实例获取 ConfigurationAPIProvider 实例间接获取
导航栏贡献	导航栏服务中 NavbarContext 实例获取 ConfigurationAPIProvider 实例间接获取 (建议作为构造方法参数传递到导航栏贡献中)

如代码块8.26所示为本节所述 TCP 相关接口的使用示例，主要为演示相关接口用法，没有实际功能意义。

```
1    private void tcpDemo() {
2        TCPModel tcpModel = this.taskApiProvider.getTCPModel();
3        ValueFactory valueFactory = this.taskApiProvider.getValueFactory
     ();
4        ConfigurationAPIProvider configurationAPIProvider = this.
     taskApiProvider.getConfigurationApi();
5        TCPContributionModel tcpContributionModel =
     configurationAPIProvider.getTCPContributionModel();
6
7        String tcpName = "demoTCP";
8        Pose tcpPose = valueFactory.createPose(0, 0, 100, 0, 0, 0,
     Length.Unit.MM, Angle.Unit.RAD);
9        TCP createdTCP = tcpContributionModel.addTCP(tcpName, tcpPose);
10       TCP acquiredTCP = tcpContributionModel.getTCP(tcpName);
11       Collection<TCP> tcps = tcpModel.getTCPs();
12       if (createdTCP != null && tcps.contains(createdTCP) &&
     createdTCP == acquiredTCP) {
13           System.out.println("demoTCP created success");
14       }else{
15           System.out.println("demoTCP created failed");
16       }
17   }
```

代码块 8.26: TCP 相关接口使用示例

8.5 坐标系

ELITECO SDK API 允许开发者在 ELITECO Plugin 中获取 EliRobot 中的坐标系，以便于在能够在 ELITECO Plugin 中完成涉及坐标系的复杂逻辑。EliRobot 中坐标系主要包括坐标系的位姿及唯一名称等数据，其接口类 Frame 如代码块8.27所示。

```
public interface Frame {
    /**
     * 获取坐标系唯一名称
     */
    String getName();

    /**
     * 获取坐标系显示名称
     */
    String getDisplayName();

    /**
     * 获取坐标系写入脚本的名称
     */
    String getScriptName();

    /**
     * 判断坐标系是否已经定义
     */
    boolean isDefined();

    /**
     * 获取坐标系姿态
     */
    Pose getPose();

    /**
     * 设置坐标系姿态，仅变量坐标系支持此操作。
     */
    void setPose(@Nonnull Pose pose);
}
```

代码块 8.27: Frame

类似8-1获取 VariableService 方式可以获取 FrameModel 实例，继而获取 EliRobot 中的坐标系，FrameModel 接口如代码块8.28所示。

```
1   public interface FrameModel {
2       /**
3        * 获取全部坐标系
4        */
5       List<Frame> getFrames();
6
7       /**
8        * 获取全部自定义坐标系(除机器人基坐标系及工具坐标系外)
9        */
10      List<Frame> getCustomFrames();
11
12      /**
13       * 获取机器人基坐标系
14       */
15      RobotBaseFrame getBaseFrame();
16
17      /**
18       * 获取工具坐标系
19       */
20      ToolFrame getToolFrame();
21
22      /**
23       * 获取机器人安装角度
24       */
25      Vector3d getGravity();
26  }
```

代码块 8.28: FrameModel

如代码块8.28所示，通过 FrameModel 实例可以获取 EliRobot 中机器人基坐标系、工具坐标系、自定义坐标系及全部坐标系等坐标系数据。坐标系相关接口比较简单，此处不再给出使用示例。

8.6 数据持久化

前文第 4 章 定制配置节点及第 5 章 定制任务节点章节中都有涉及到数据持久化接口来存储配置节点及任务节点贡献中的数据，但不够系统。这一节，就系统性的描述 ELITECO Plugin 中的数据持久化。

ELITECO Plugin 实现数据持久化有赖于 DataModelWrapper 实例，DataModelWrapper 接口类中提供了大量基础数据类型、机器人相关计量类型及其他诸如变量等高级数据类

型的存储、查询、删除及其他接口，其接口如代码块8.29所示 (有缩略，仅列举有代表性的，详见 ELITECO SDK API 或接口文档)。

```
    public interface DataModelWrapper {
        /**
         * 获取key为property布尔型数据
         */
        Boolean getBoolean(String property);

        /**
         * 设置key为property布尔型数据
         */
        boolean setBoolean(String property, Boolean value);

        /**
         * 获取key为property字符串数据
         */
        String getString(String property);

        /**
         * 设置key为property字符串数据
         */
        boolean setString(String property, String value);

        /**
         * 获取key为property变量
         */
        Variable getVariable(String property);

        /**
         * 设置key为property变量
         */
        boolean setVariable(String property, Variable variable);

        /**
         * 获取key为property角度数据
         */
        Angle getAngle(String property);

        /**
         * 设置key为property角度数据
         */
        boolean setAngle(String property, Angle angle);

```

```
42          /**
43           * 获取key为property表达式
44           */
45          Expression getExpression(String property);
46
47          /**
48           * 设置key为property表达式
49           */
50          boolean setExpression(String property, Expression expression);
51
52          /**
53           * 获取key为property数据
54           */
55          boolean set(String property, Object value);
56
57          /**
58           * 设置key为property数据(对象类型，需自行转换)
59           */
60          Object get(String property);
61
62          /**
63           * 获取key为property数据，若不存在则返回给定默认值
64           */
65          Object get(String property,Object defaultValue);
66
67          /**
68           * 删除key为property数据
69           */
70          boolean remove(String property);
71
72          /**
73           * 获取所有数据的key集合
74           */
75          Set<String> getDataKeySet();
76
77          /**
78           * 注册自定义数据转换策略
79           */
80          default void registerConversionStrategy(String id, Class<?
    extends PluginConversionStrategy> clazz) {
81          }
82
83        ...
84      }
```

代码块 8.29: DataModelWrapper

其中配置节点贡献及任务节点贡献中用于持久化的 ConfigurationDataModelWrapper 接口及 TaskNodeDataModelWrapper 接口都是 DataModelWrapper 的派生接口。其中数据获取、设置及删除接口的使用比较简单且代码块4.2及代码块 5.2都有所涉及，此处不再示例使用方法。

另外代码块8.29中有 registerConversionStrategy 接口，主要用于 DataModelWrapper 已有数据类型接口无法满足开发者实际需要需要自定义数据类型时，通过实现 PluginConversionStrategy 接口类定义该数据类型转换策略以保证该数据能够实现持久化。但需注意的是，目前 TaskNodeDataModelWrapper 接口不支持该特性，也即任务节点定制不支持注册自定义数据类型转换策略。PluginConversionStrategy 接口类如代码块8.30所示。

```
1  public interface PluginConversionStrategy<T> extends
   ConversionStrategy<T> {
2      /**
3       * 获取数据节点名称
4       */
5      String getNodeName();
6  }
```

代码块 8.30: PluginConversionStrategy

PluginConversionStrategy 接口是 ConversionStrategy 接口的派生接口，自定义数据类型转换策略要实现的主体接口都在 ConversionStrategy 接口中。ConversionStrategy 接口如代码块8.31所示。

```
1  public interface ConversionStrategy<T> {
2      /**
3        * 获取转换策略支持数据类型
4        */
5      Class<? extends T> getSupportedType();
6
7      /**
8        * 当前PersistReader读取到的数据否可以被当前转换策略解析
9        */
10     boolean canUnmarshalFrom(PersistReader reader);
11
12     /**
13       * PersistWriter实例写入数据
14       */
```

```
15          void marshal(T source, PersistWriter writer);
16
17          /**
18            * PersistWriter实例写入数据 重载方法
19            */
20          default void marshal(T source, PersistWriter writer, String key)
        {
21              marshal(source, writer);
22          }
23
24          /**
25            * 解析PersistReader读取到的数据为数据实例
26            */
27          T unmarshal(PersistReader reader);
28
29          /**
30            * 获取 数据类 别名
31            */
32          Map<String, Class<? extends T>> getClassAliases(boolean aliases)
       ;
33
34          /**
35            * 获取 数据类 别名
36            */
37          Map<String, Class<? extends T>> getTypeAliases(boolean aliases);
38      }
```

代码块 8.31: ConversionStrategy

下面就通过自定义数据类型 CustomDataDemo 及其转换策略 CustomDataDemoConversion Strategy 为例讲解自定义数据转换策略功能特性。自定义数据类型 CustomDataDemo 如代码块8.32所示。

```
1       public class CustomDataDemo{
2           private int intData = 0;
3           private String stringData = "";
4           private Angle angleData;
5
6           public CustomDataDemo(int intData, String stringData, Angle
       angleData) {
7               this.intData = intData;
8               this.stringData = stringData;
9               this.angleData = angleData;
10          }
```

```
11
12          public int getIntData() {
13              return intData;
14          }
15
16          public void setIntData(int intData) {
17              this.intData = intData;
18          }
19
20          public String getStringData() {
21              return stringData;
22          }
23
24          public void setStringData(String stringData) {
25              this.stringData = stringData;
26          }
27
28          public Angle getAngleData() {
29              return angleData;
30          }
31
32          public void setAngleData(Angle angleData) {
33              this.angleData = angleData;
34          }
35      }
```

代码块 8.32: CustomDataDemo

下面实现 CustomDataDemo 数据类型的转换策略 CustomDataDemoConversionStrategy，分别如代码块8.33和代码块8.34所示。

```
1       public class CustomDataDemoConversionStrategy implements
        PluginConversionStrategy<CustomDataDemo> {
2           public static final String NODE_NAME = "CustomDataDemo";
3
4           @Override
5           public Class<? extends CustomDataDemo> getSupportedType() {
6               return CustomDataDemo.class;
7           }
8
9           @Override
10          public boolean canUnmarshalFrom(PersistReader reader) {
11              return NODE_NAME.equals(reader.getNodeName());
12          }
```

```
13
14          @Override
15          public void marshal(CustomDataDemo source, PersistWriter writer)
     {
16          }
17
18          @Override
19          public void marshal(CustomDataDemo source, PersistWriter writer,
     String key) {
20              writer.startNode(NODE_NAME);
21              writer.addAttribute("key", key);
22              writer.addAttribute("intData", source.intData);
23              writer.addAttribute("stringData", source.stringData);
24              writer.endNode();
25          }
26
27          @Override
28          public CustomDataDemo unmarshal(PersistReader reader) {
29              String key = reader.getStringAttribute("key");
30              int intData = reader.getIntegerAttribute("intData", 0);
31              String stringData = reader.getStringAttribute("stringData");
32              CustomDataDemo customDataDemo = new CustomDataDemo(intData,
    stringData);
33              return customDataDemo;
34          }
35
36          @Override
37          public Map<String, Class<? extends CustomDataDemo>>
    getClassAliases(boolean aliases) {
38              return Collections.singletonMap(NODE_NAME, CustomDataDemo.
    class);
39          }
40
41          @Override
42          public Map<String, Class<? extends CustomDataDemo>>
    getTypeAliases(boolean aliases) {
43              return Collections.singletonMap(NODE_NAME, CustomDataDemo.
    class);
44          }
45
46          @Override
47          public String getNodeName() {
48              return NODE_NAME;
49          }
50      }
```

代码块 8.33: CustomDataDemo

```
1   public class CustomDataDemoConversionStrategy implements
    PluginConversionStrategy<CustomDataDemo> {
2       public static final String NODE_NAME = "CustomDataDemo";
3
4       @Override
5       public Class<? extends CustomDataDemo> getSupportedType() {
6           return CustomDataDemo.class;
7       }
8
9       @Override
10      public boolean canUnmarshalFrom(PersistReader reader) {
11          return NODE_NAME.equals(reader.getNodeName());
12      }
13
14      @Override
15      public void marshal(CustomDataDemo source, PersistWriter writer)
     {
16      }
17
18      @Override
19      public void marshal(CustomDataDemo source, PersistWriter writer,
     String key) {
20          writer.startNode(NODE_NAME);
21          writer.addAttribute("key", key);
22          writer.addAttribute("intData", source.intData);
23          writer.addAttribute("stringData", source.stringData);
24          writer.endNode();
25      }
26
27      @Override
28      public CustomDataDemo unmarshal(PersistReader reader) {
29          int intData = reader.getIntegerAttribute("intData", 0);
30          String stringData = reader.getStringAttribute("stringData");
31          CustomDataDemo customDataDemo = new CustomDataDemo(intData,
    stringData);
32          return customDataDemo;
33      }
34
35      @Override
36      public Map<String, Class<? extends CustomDataDemo>>
    getClassAliases(boolean aliases) {
```

```
37              return Collections.singletonMap(NODE_NAME, CustomDataDemo.
    class);
38          }
39
40          @Override
41          public Map<String, Class<? extends CustomDataDemo>>
    getTypeAliases(boolean aliases) {
42              return Collections.singletonMap(NODE_NAME, CustomDataDemo.
    class);
43          }
44
45          @Override
46          public String getNodeName() {
47              return NODE_NAME;
48          }
49      }
```

代码块 8.34: CustomDataDemoConversionStrategy

如代码块8.34所示，三参数的的 marshal 方法实现数据序列化，将数据对应的 key 及数据本身通过 PersistWriter 接口序列化，而 unmarshal 则通过 PersistReader 接口读取序列化数据解析出数据实例。

开发者需注意两点：1. 两参数的 marshal 方法目前不会被调用，空内容即可；2.marshal 方法中务必如代码块8.34中 21 行所示将数据实例的 key 以数据节点属性序列化，并且属性的 name 必须是“key”，否则将无法被正确解析。

8.7 脚本生成

EliRobot 中运行任务的实质其实是运行 python 脚本，任务树所有节点及配置模块数据都会写入脚本中，因此若定制配置节点贡献及定制任务节点贡献都有 generateScript 接口，结合 EliServer 脚本文档给出的接口及节点数据生成节点脚本驱动机器人完成节点预期的逻辑功能。

ScriptWriter 实例是 generateScript 接口唯一的参数，含有用于写 python 脚本的接口方法，通过该接口实例结合节点数据可以较方便且简单的写入符合预期逻辑需求的脚本。ScriptWriter 接口类如代码块8.35所示 (有缩略，详见 ELITECO SDK API 或接口文档)。

```
1   public interface ScriptWriter {
2       /**
3        * 将输入的包含不合法字符的字符串转换为合法的脚本字符串
```

```
         */
        String toScriptString(String string);


        /**
         * 写入当前节点子节点的脚本代码
         */
        void writeChildren();

        /**
         * 写入一行文本
         */
        void appendLine(String singleLineText);

        /**
         * 填充原始脚本
         */
        void appendRaw(String rawScript);

        /**
         * 使用方法名name写入定义函数 脚本行
         */
        void defineFunction(String name);

        /**
         * 使用方法名name及方法参数args写入定义函数 脚本行
         */
        void defineFunction(String name, String... args);

        /**
         * 写入return脚本行
         */
        void returnMethod();

        ...
    }
```

代码块 8.35: ScriptWriter

下面以代码块5.2中 62 行的方法为例 generateScript 描述 ScriptWriter 的使用，该方法内容如代码块8.36所示。

```
    ...

```

```
3     public void generateScript(ScriptWriter scriptWriter) {
4         Variable variable = getSelectedVariable();
5         if(variable != null) {
6             String resolvedVariableName = scriptWriter.
    getResolvedVariableName(variable);
7             scriptWriter.appendLine(resolvedVariableName + " = " +
    resolvedVariableName + " + 1");
8             scriptWriter.writeChildren();
9         }
10    }
11
12    ...
```

代码块 8.36: generateScript 示例

如代码块8.36所示，该节点变量不为空时，获取变量的可解析名称，填充变量自增脚本行，完成该节点本身的脚本写入，最后再通过 writeChildren 写入后代节点脚本。

代码块8.36有两个要点需要注意：1. 要写入脚本中的作为脚本中变量、方法及其他名称时为防止字符串本身包含不合法的字符，建议变量和其他字符串分别通过 getResolvedVariableName 和 toScriptString 接口转换为合法名称，字符面常量不需要；2. 定制任务节点生成脚本时若有子节点，需要在合适位置调用 writeChildren 写入子节点脚本，调用后 ScriptWriter 会自动遍历调用所有子节点的 generateScript 方法。

代码块8.36仅涉及到了 ScriptWriter 中较少的接口，没有穷尽，更多的接口使用方法还需要开发者参考 ELITECO SDK API 接口文档在实际中探索。

8.8 系统

除本章前几小节介绍的功能特性，ELITECO SDK API 中还提供了一些功能特性：机器人运动服务、机器人仿真服务、Rpc 服务、命名服务、机器人状态以、系统设置服务及软件版本。这些功能特性接口使用比较简单，因此同意在本节进行描述。

8.8.1 机器人运动服务

机器人运动服务主要提供了运行机器人相关接口，支持开发者手动运行机器人及运行机器人到给定位姿或关节位置等操作,机器人运动服务 RobotMovementService 如代码块8.37所示。

```
public interface RobotMovementService {
    /**
     * 获取机器人当前TCP位姿
     */
    Pose getCurrentTCPPose();

    /**
     * 获取机器人当前关节位置
     */
    JointPositions getCurrentJointPositions();

    /**
     * 机器人关节运动到给定关节位置
     */
    void requestUserToMoveRobot(JointPositions jointPositions,
AutoMoveCallback autoMoveCallback);

    /**
     * 机器人线性运动到给定位姿
     */
    void requestUserToMoveRobot(Pose pose, AutoMoveCallback
autoMoveCallback);

    /**
     * 进入示教页面手动运动机器人
     */
    void requestUserToSetPosition(ViewState viewState,
SetPositionCallback setPositionCallback);

     /**
     * 机器人示教页面状态
     */
    class ViewState {
        private boolean showAcceptButton = true;
        private boolean showCancelButton = true;
        private boolean showPreviewRobot = false;
        private boolean isSafeHomeButtonEnable = true;
        private boolean isZeroPositionButtonEnable = true;
        private JointPositions targetJointPose;

        public ViewState() {
            double[] joints = new double[6];
            joints[0] = 0.0D;
            joints[1] = -1.5707963267948966D;
            joints[2] = 0.0D;
```

```
43                joints[3] = -1.5707963267948966D;
44                joints[4] = 0.0D;
45                joints[5] = 0.0D;
46                this.targetJointPose = JointPositionsUtils.
    newJointPositions(joints, Angle.Unit.RAD);
47            }
48
49            public ViewState(boolean showAcceptButton, boolean
    showCancelButton,
50                             boolean isSafeHomeButtonEnable, boolean
    isZeroPositionButtonEnable,
51                             boolean showPreviewRobot, JointPositions
    targetJointPose) {
52                this.showAcceptButton = showAcceptButton;
53                this.showCancelButton = showCancelButton;
54                this.isSafeHomeButtonEnable = isSafeHomeButtonEnable;
55                this.isZeroPositionButtonEnable =
    isZeroPositionButtonEnable;
56                this.showPreviewRobot = showPreviewRobot;
57                this.targetJointPose = JointPositionsUtils.
    newJointPositions(targetJointPose);
58            }
59
60            public boolean isShowAcceptButton() {
61                return showAcceptButton;
62            }
63
64            public void setShowAcceptButton(boolean showAcceptButton) {
65                this.showAcceptButton = showAcceptButton;
66            }
67
68            public boolean isShowCancelButton() {
69                return showCancelButton;
70            }
71
72            public void setShowCancelButton(boolean showCancelButton) {
73                this.showCancelButton = showCancelButton;
74            }
75
76            public boolean isShowPreviewRobot() {
77                return showPreviewRobot;
78            }
79
80            public void setShowPreviewRobot(boolean showPreviewRobot) {
81                this.showPreviewRobot = showPreviewRobot;
```

```
82              }
83
84              public boolean isSafeHomeButtonEnable() {
85                  return isSafeHomeButtonEnable;
86              }
87
88              public void setSafeHomeButtonEnable(boolean
    safeHomeButtonEnable) {
89                  isSafeHomeButtonEnable = safeHomeButtonEnable;
90              }
91
92              public boolean isZeroPositionButtonEnable() {
93                  return isZeroPositionButtonEnable;
94              }
95
96              public void setZeroPositionButtonEnable(boolean
    zeroPositionButtonEnable) {
97                  isZeroPositionButtonEnable = zeroPositionButtonEnable;
98              }
99
100             public JointPositions getTargetJointPose() {
101                 return targetJointPose;
102             }
103
104             public void setTargetJointPose(JointPositions
    targetJointPose) {
105                 this.targetJointPose = JointPositionsUtils.
    newJointPositions(targetJointPose);
106             }
107         }
108     }
```

代码块 8.37: RobotMovementService 接口

如代码块8.37所示，RobotMovementService 中关键接口主要有涉及机器人移动的接口主要有三个：requestUserToMoveRobot 关节移动到给定关节位置、requestUserToMoveRobot 线性运动到给定位姿及 requestUserToSetPosition 打开示教页面手动示教。下面以代码块 8.38为例讲解 RobotMovementService 接口类中的接口使用方法。

```
1     private void moveRobotToPoseDemo(){
2         ValueFactory valueFactory = this.taskNodeViewApiProvider.
    getTaskApiProvider().getValueFactory();
3         Pose pose = valueFactory.createPose(92, -140.48, 492.22, 3.13,
    0, -1.571, Length.Unit.MM, Angle.Unit.RAD);
```

```
4         this.taskNodeViewApiProvider.getTaskApiProvider().
    getRobotMovementService().requestUserToMoveRobot(pose, new
    AutoMoveCallback() {
5             @Override
6             public void onComplete(AutoMoveCompleteEvent
    autoMoveCompleteEvent) {
7                 System.out.println("On continue");
8                 System.out.println("Is at target position: " +
    autoMoveCompleteEvent.isAtTargetPosition());
9             }
10
11            @Override
12            public void onCancel(AutoMoveCancelEvent autoMoveCancelEvent
    ) {
13                System.out.println("On cancel");
14                System.out.println("Is at target position: " +
    autoMoveCancelEvent.isAtTargetPosition());
15            }
16        });
17    }
18
19    private void moveRobotToPositionDemo(){
20        ValueFactory valueFactory = this.taskNodeViewApiProvider.
    getTaskApiProvider().getValueFactory();
21        JointPositions JointPositions = valueFactory.
    createJointPositions(0, -90, 0, -90, 90, 0, Angle.Unit.DEG);
22        this.taskNodeViewApiProvider.getTaskApiProvider().
    getRobotMovementService().requestUserToMoveRobot(JointPositions, new
     AutoMoveCallback() {
23            @Override
24            public void onComplete(AutoMoveCompleteEvent
    autoMoveCompleteEvent) {
25                System.out.println("On continue");
26                System.out.println("Is at target position: " +
    autoMoveCompleteEvent.isAtTargetPosition());
27            }
28
29            @Override
30            public void onCancel(AutoMoveCancelEvent autoMoveCancelEvent
    ) {
31                System.out.println("On cancel");
32                System.out.println("Is at target position: " +
    autoMoveCancelEvent.isAtTargetPosition());
33            }
34        });
```

```
35      }
36
37      private void setPositionDemo() {
38          RobotMovementService robotMovementService = this.
        taskNodeViewApiProvider.getTaskApiProvider().getRobotMovementService
        ();
39          this.taskNodeViewApiProvider.getTaskApiProvider().
        getRobotMovementService().requestUserToSetPosition(new
        RobotMovementService.ViewState(), new SetPositionCallback() {
40              @Override
41              public void onComplete(SetPositionCompleteEvent
        setPositionCompleteEvent) {
42                  taskApiProvider.getUndoRedoManager().recordChanges(() ->
         {
43                      contribution.setPositions(robotMovementService.
        getCurrentTCPPose(), robotMovementService.getCurrentJointPositions()
        );
44                  });
45              }
46          });
47      }
```

代码块 8.38: RobotMovementService 接口

代码块8.38很好得演示了 RobotMovementService 中的接口的用法，不再过多赘述。

8.8.2 机器人仿真服务

ELITECO SDK API 支持开发者使用机器人仿真服务 SimulationService 接口创建机器人仿真视图，在视图中布局机器人仿真视图，可以使开发过程中涉及机器人的位姿或关节位置数据更加直观。SimulationService 接口类如代码块8.39所示。

```
1   public interface SimulationService {
2       /**
3        * 创建机器人仿真视图
4        */
5       SimulationCanvas createSimulationCanvas();
6   }
```

代码块 8.39: SimulationService 接口

SimulationService 接口类有唯一接口 createSimulationCanvas 用于创建 SimulationCanvas 实例。通过 SimulationCanvas 实例可以获取机器人仿真可视化组件并对进行相关设定。SimulationCanvas 接口类如代码块8.40所示。

```
public interface SimulationCanvas {
    /**
     * 更新仿真机器人当前关节位置
     */
    void setRobotJoints(JointPositions joints);

    /**
     * 更新仿真机器人目标关节位置
     */
    void setTargetRobotJoints(JointPositions joints);

    /**
     * 获取机器人显示状态
     */
    boolean isRobotVisible();

    /**
     * 设置机器人显示状态
     */
    void setRobotVisible(boolean robotVisible);

    /**
     * 获取机器人目标关节位置显示状态
     */
    boolean isTargetRobotVisible();

    /**
     * 设置机器人目标关节位置显示状态
     */
    void setTargetRobotVisible(boolean targetRobotVisible);

    /**
     * 获取工具显示状态
     */
    boolean isToolVisible();

    /**
     * 设置工具显示状态
     */
    void setToolVisible(boolean toolVisible);
```

```
41
42          /**
43           * 获取工具坐标轴显示状态
44           */
45          boolean isToolAxisVisible();
46
47          /**
48           * 设置工具坐标轴显示状态
49           */
50          void setToolAxisVisible(boolean toolAxisVisible);
51
52          /**
53           * 获取机器人实时轨迹显示状态
54           */
55          boolean isRealtimePathVisible();
56
57          /**
58           * 设置机器人实时轨迹显示状态
59           */
60          void setRealtimePathVisible(boolean showRealtimePath);
61
62          /**
63           * 清除机器人实时轨迹
64           */
65          void clearRealtimePath();
66
67          /**
68           * 添加显示节点
69           */
70          void addCanvasNode(Node node);
71
72          /**
73           * 设置缩放控制显示状态(默认显示)
74           */
75          void setShowZoomControl(boolean showZoomControl);
76
77          /**
78           * 获取机器人仿真显示组件
79           */
80          Component getComponent();
81      }
```

代码块 8.40: SimulationCanvas 接口

需注意的是 getCompponent 接口获取的 Component 实例才是机器人仿真可视化组件，

其他接口则是设置或者查询其参数设定接口，下面以代码块8.41为例描述 SimulationCanvas 使用方法。

```
1     private void createsimulationCanvasDemo{
2         this.simulationCanvas = this.taskApiProvider.
      getSimulationService().createSimulationCanvas();
3         simulationCanvas.setRealtimePathVisible(false);
4         simulationCanvas.setShowZoomControl(false);
5         JPanel robotPanel = new JPanel();
6         robotPanel.setLayout(new BorderLayout());
7         Component component = simulationCanvas.getComponent();
8         component.setEnabled(false);
9         robotPanel.add(component,BorderLayout.CENTER);
10    }
11
12    public void updateSimulationCanvasDemo() {
13        simulationCanvas.setRobotJoints(this.contribution.
      getJointPositions());
14    }
```

代码块 8.41: SimulationCanvas 示例

代码块8.41中 createsimulationCanvasDemo 方法中创建了 SimulationCanvas 实例，并设置不显示机器人实时位姿及缩放控制，其他设置按默认设定，并将 SimulationCanvas 实例的机器人仿真显示可视化组件布局在一个页面上。同时在另外一个方法 updateSimulationCanvasDemo 中更新了机器人关节位置。

SimulationCanvas 其他的接口多是设置/获取参数类接口，比较简单，代码块8.41中未涉及。

8.8.3 Rpc 服务

ELITECO Plugin 中支持开发者特定场景下给 EliServer 发送文本脚本完成特定的功能，实现该功能需要通过 RpcService 服务类接口，RpcService 如代码块8.42所示。

```
1     public interface RpcService {
2         /**
3          * 运行机器人脚本
4          */
5         boolean runScript(String script);
6
```

```
7         /**
8          * 运行匿名脚本
9          */
10        boolean runIncognitoScript(Consumer<StringBuilder> scriptBody);
11
12        /**
13         * 运行Sec脚本
14         */
15        boolean runSecScript(String methodName, Consumer<StringBuilder>
    scriptBody);
16    }
```

代码块 8.42: RpcService

通过 RpcService 发送的脚本，通过时由脚本函数的方式构成，EliServer 脚本函数包含：函数体，函数定义及函数结束标志 end，如代码块8.43所示。

```
1     def $methodName:
2       ...
3     end
```

代码块 8.43: RpcService 支持脚本格式

代码块8.42中 runScript 参数 script 包含脚本函数全部部分，而 runIncognitoScript 则是发送匿名脚本，也即只需要发送函数体即可，两个方法所发送的脚本都与任务运行相冲突，也即运行任务、runScript 及 runIncognitoScript 三者任一发送的脚本正在运行中，其他的任一发送的脚本会先把前者脚本停止，然后运行当前脚本内容。

另外一个接口 runSecScript 接口则有所不同，其参数是方法名及函数体，不需要开发者封装好的脚本函数代码；同时 runSecScript 发送的脚本和运行任务 runScript 及 runIncognitoScript 三者运行的脚本不冲突，可以并行运行，但两次调用 runSecScript 发送的脚本会相互冲突，停止前者，运行后者。

更多关于 Rpc 脚本的内容可以参考 EliRobot 使用说明或 EliServer 接口文档。

RpcService 接口实例的获取方式同8-1中 VariableService 获取方式相同，此处不再赘述。

8.8.4 命名服务

为方便开发者在写入脚本时方便将一些可能会用作 python 脚本变量的字符串判断是否唯一或转换为合法名称，ELITECO SDK API 中提供该服务的接口类是 NamingService，NamingService 如代码块8.44所示。

```
1   public interface NamingService {
2       /**
3        * 判断字符串作为脚本变量是否被占用
4        */
5       boolean isNameUsed(String name);
6
7       /**
8        * 基于name由命名服务创建一个合法变量名
9        */
10      String makeUniqueName(String name);
11  }
```

代码块 8.44: NamingService

NamingService 接口实例的获取方式同8-1中 VariableService 获取方式相同，此处不再赘述。

NamingService 接口实例中接口都比较简单，此处不再示例。

8.8.5 机器人状态

ELITECO SDK API 支持开发者获取机器人状态，ELITECO SDK API 中提供该服务的接口类是 RobotStateApiProvider 接口实例，RobotStateApiProvider 如代码块8.45所示。

```
1   public interface RobotStateApiProvider extends ApplicationAPI {
2       /**
3        * 获取机器人模式
4        */
5       RobotMode getRobotMode();
6
7       /**
8        * 获取机器人安全模式
9        */
10      SafetyMode getSafetyMode();
11  }
```

代码块 8.45: NamingService

RobotStateApiProvider 接口实例的获取方式同8-1中 VariableService 获取方式相同，此处不再赘述。

RobotStateApiProvider 中接口都比较简单，返回值也都是表示模式的枚举类，此处不再示例使用方法。

8.8.6 系统设置服务

ELITECO SDK API 支持开发者获取 EliRobot 中系统设置，提供该服务的接口类是 SystemSettings 接口实例，SystemSettings 如代码块 8-46 所示。

```
public interface SystemSettings {
    /**
     * 返回当前Localization 实例
     */
    Localization getLocalization();

    /**
     * 返回当前任务文件路径(包含脚本及配置文件)
     */
    File getProgramPath();

    /**
     * 获取Home路径
     */
    File getHomePath();

    /**
     * 获取配置路径
     */
    File getConfigPath();

    /**
     * 获取数据文件路径
     */
    File getDataPath();

    /**
     * 获取日志文件路径
     */
    File getLogPath();

    /**
     * 获取USB设备根路径
     */
    File getUSBRootPath();

    /**
     * 获取当前端口映射模式
     * @return 0:未使用,2: RS485
```

```
40          */
41          int getPortMode();
42
43          /**
44           * 设置端口映射模式
45           * @param mode 0: 不使用,2: RS485
46           */
47           void setPortMode(int mode);
48
49         /**
50           * 获取FB1网络信息
51           * @return NetworkInf
52           */
53           NetworkInfo getNetWorkFB1();
54
55           /**
56           * 获取FB2网络信息
57           * @return NetworkInf
58           */
59           NetworkInfo getNetWorkFB2();
60
61           /**
62           * 获取FB2路由模式是否启用
63           * @return enable
64           */
65           boolean getFB2RouteEnable();
66
67           /**
68            * 启用、禁用FB2网络路由模式
69            * @param enable
70            */
71           void setFB2RouteEnable(boolean enable);
72
73           /**
74           * 网络信息
75           */
76           public interface NetworkInfo {
77              .....
78    }
```

代码块 8.46: SystemSettings

SystemSettings 接口实例的获取方式同8-1中 VariableService 获取方式相同，此处不再赘述。

SystemSettings 中接口多是获取本地化实例及其他系统文件路径，都比较简单，此处不再示例使用方法。

8.8.7 系统 IP 服务

因 EliRobot 与 EliServer 可以在真机、虚拟机开发平台等多个平台上运行，致使两者之间建立通讯的 IP 地址并不总是回环地址（127.0.0.1）。为了保证通讯无误，ELITECO SDK API 中提供了 SystemIpService 接口类，以实现 EliRobot 与 EliServer 之间的无障碍通信，如代码块8.47所示。

```
1   public interface SystemIpService {
2       /**
3        * 获取提供机器人数据和支持脚本运行的控制器-EliServer
4        */
5       String getEliServerIP();
6
7       /**
8        * 获取HMI-EliRobot的ip地址
9        */
10      String getEliRobotIP();
11  }
```

代码块 8.47: SystemIpService

8.8.8 任务状态服务

ELITECO SDK API 为开发者提供了一系列接口，用于获取、控制和监听任务的当前运行状态，如代码块 8-48 所示。

```
1   public interface TaskStateApiProvider extends ApplicationAPI {
2       /**
3        * 判断任务是否处于运行状态
4        */
5       boolean isTaskRunning();
6
7       /**
8        * 判断任务是否处于暂停状态
9        */
10      boolean isTaskPaused();
```

```
11
12          /**
13           * 判断任务是否处于停止状态
14           */
15          boolean isTaskStopped();
16
17          /**
18           * 用参数“taskState”的值来改变任务运行状态
19           */
20          void taskControl(TaskState taskState);
21
22          /**
23           * 添加任务状态监听器来监听任务状态改变事件
24           */
25          void addTaskStateListener(TaskStateListener taskStateListener);
26
27          /**
28           * 移除指定任务状态监听器，不再监听任务状态改变事件
29           */
30          void removeTaskStateListener(TaskStateListener taskStateListener
    );
31      }
```

代码块 8.48: 任务状态服务

8.8.9 软件版本

ELITECO SDK API 支持开发者获取 EliRobot 中软件版本，提供该服务的接口类是 SoftwareVersion 接口实例，SoftwareVersion 如代码块8.49所示。

```
1       public interface SoftwareVersion {
2           /**
3            * 获取Major版本
4            */
5           int getMajorVersion();
6
7           /**
8            * 获取Minor版本
9            */
10          int getMinorVersion();
11
12          /**
```

```
13          * 获取Bugfix版本
14          */
15         int getBugfixVersion();
16
17         /**
18          * 获取构建版本
19          */
20         int getBuildNumber();
21     }
```

代码块 8.49: SoftwareVersion

SoftwareVersion 接口实例的获取方式同8-1中 VariableService 获取方式相同，此处不再赘述。

SoftwareVersion 中接口主要是获取 EliRobot 版本号，都比较简单，此处不再示例使用方法。

第 9 章 附录

9.1 内部任务节点创建及配置示例

不同内部任务节点的创建及配置略有不同，与其本身特性有关，此处只对节点创建及配置进行代码示例，节点本身特性详见 EliRobot 使用说明。

9.1.1 Move 节点创建

- MoveJ 节点创建及初始化

```
MoveNode moveNode = factory.createMoveNode();

// 创建默认参数的 关节运动模式参数构造器(MoveJointConfigBuilder)
MoveJointConfigBuilder moveJointConfigBuilder = moveNode.getConfigBuilderFactory().createMoveJConfigBuilder();
// 初始化 描述(describe)、关节速度(jointSpeed)、关节加速度(jointAcceleration)、TCP等参数
AngularSpeed jointSpeed = valueFactory.createAngularSpeed(80.0D, AngularSpeed.Unit.DEG_S);
AngularAcceleration angularAcceleration = valueFactory.createAngularAcceleration(1200.0D, AngularAcceleration.Unit.DEG_S2);
TCPSelection tcpSelection = moveJ.getTCPSelectionFactory().createIgnoreActiveTCPSelection();
moveJointConfigBuilder.setDescribe("Demo");
moveJointConfigBuilder.setJointSpeed(jointSpeed);
moveJointConfigBuilder.setJointAcceleration(angularAcceleration);
moveJointConfigBuilder.setTCPSelection();

// 使用 关节运动模式参数构造器(MoveJointConfigBuilder) 初始化moveNode节点
moveJ.setConfig(moveJointConfigBuilder.build());
```

- MoveL 节点创建及初始化

```
MoveNode moveNode = factory.createMoveNode();

// 创建默认参数的 直线运动模式参数构造器(MoveJointConfigBuilder)
MoveLinerConfigBuilder moveLConfigBuilder = moveNode.getConfigBuilderFactory().createMoveLConfigBuilder();

// 初始化 工具速度(toolSpeed)、工具加速度(toolAcceleration)、TCP等参数
Speed speed = valueFactory.createSpeed(80.0D, Speed.Unit.MM_S);
Acceleration acceleration = valueFactory.createAcceleration(15000, Acceleration.Unit.MM_S2);
TCPSelection tcpSelection = moveNode.getTCPSelectionFactory().createIgnoreActiveTCPSelection();
moveLConfigBuilder.setToolSpeed(speed);
moveLConfigBuilder.setToolAcceleration(angularAcceleration);
moveLConfigBuilder.setTCPSelection(tcpSelection);

// 使用 直线运动模式参数构造器(MoveLinerConfigBuilder) 初始化moveNode节点
moveNode.setConfig(moveLConfigBuilder.build());
```

- MoveP 节点创建及初始化

```
MoveNode moveNode = factory.createMoveNode();

// 创建默认参数的 过程运动模式参数构造器(MoveProcessConfigBuilder)
MoveProcessConfigBuilder movePConfigBuilder = moveNode.getConfigBuilderFactory().createMovePConfigBuilder();
// 初始化 关节速度(jointSpeed)、关节加速度(jointAcceleration)、TCP、转接半径(transitionRadius)等参数
Speed speed = valueFactory.createSpeed(80.0D, Speed.Unit.MM_S);
Acceleration acceleration = valueFactory.createAcceleration(15000, Acceleration.Unit.MM_S2);
TCPSelection tcpSelection = moveNode.getTCPSelectionFactory().createIgnoreActiveTCPSelection();
Length length = valueFactory.createLength(50.0D, Length.Unit.MM);
movePConfigBuilder.setToolSpeed(speed);
movePConfigBuilder.setToolAcceleration(angularAcceleration);
movePConfigBuilder.setTCPSelection(tcpSelection);
movePConfigBuilder.setTransitionRadius(length);

```

```
15      // 使用 关节运动模式参数构造器(MoveJointConfigBuilder) 初始化moveJ节
        点
16      moveNode.setConfig(movePConfigBuilder.build());
```

以上仅展示了 Move 节点的创建及初始化，但任务树上 Move 节点需要带有 WaypointNode 节点或者 DirectionNode(仅 MoveL 或 MoveP) 作为子节点, 因此以上实例中还需要为 Move 节点添加 WaypointNode 或者 DirectionNode 节点作为子节点才能插入到任务树上。

当然也可以使用使用节点工厂 (TaskNodeFactory) 中的模板方法，创建诸如 Move-Waypoint (包含 WaypointNode 子节点的 Move 节点)、Move-Direction(包含 DirectionNode 子节点的 Move 节点) 及 Move-ArcMotion(包含 ArcMotion 子节点的 Move 节点) 等模板。

- Move-Waypoint 节点模板创建及初始化

```
1       // 创建 Move-Waypoint 模板
2       MoveNode moveWaypointNodeTemplate = factory.
        createMoveWaypointNodeTemplate();
3
4       // 创建默认参数的 关节运动模式参数构造器(MoveJointConfigBuilder)
5       MoveJointConfigBuilder moveWaypointTempConfigBuilder =
        moveWaypointNodeTemplate.getConfigBuilderFactory().
        createMoveJConfigBuilder();
6       AngularSpeed jointSpeed = valueFactory.createAngularSpeed(80.0D
        , AngularSpeed.Unit.DEG_S);
7       AngularAcceleration jointAcceleration = valueFactory.
        createAngularAcceleration(1200.0D, AngularAcceleration.Unit.
        DEG_S2);
8       moveWaypointNodeTemplate.getTCPSelectionFactory().
        createIgnoreActiveTCPSelection();
9       moveWaypointTempConfigBuilder.setJointSpeed();
10      moveWaypointTempConfigBuilder.setJointAcceleration();
11      moveWaypointTempConfigBuilder.setTCPSelection();
12
13      moveWaypointNodeTemplate.setConfig(
        moveWaypointTempConfigBuilder.build());
```

- Move-Direction 节点模板创建及初始化

```
1       // 创建 Move-Direction模板
2       MoveNode moveDirectionUntilNodeTemplate = factory.
        createMoveDirectionUntilNodeTemplate();
3
```

```
4     // 创建默认参数的 线性运动模式参数构造器(MoveLinerConfigBuilder)
5     MoveLinerConfigBuilder moveLConfigBuilder =
      moveDirectionUntilNodeTemplate.getConfigBuilderFactory().
      createMoveLConfigBuilder();
6     moveLConfigBuilder.setToolSpeed(valueFactory.createSpeed(80.0D,
       Speed.Unit.MM_S));
7     moveLConfigBuilder.setToolAcceleration(valueFactory.
      createAcceleration(15000, Acceleration.Unit.MM_S2));
8     moveLConfigBuilder.setTCPSelection(
      moveDirectionUntilNodeTemplate.getTCPSelectionFactory().
      createIgnoreActiveTCPSelection());
9
10    moveDirectionUntilNodeTemplate.setConfig(moveLConfigBuilder.
      build());
```

- Move-ArcMotion 节点模板创建及初始化

```
1     MoveNode moveArcMotionTemplate = factory.
      createMoveArcMotionTemplate();
2     MoveProcessConfigBuilder moveArcMotionTempConfigBuilder =
      moveArcMotionTemplate.getConfigBuilderFactory().
      createMovePConfigBuilder();
3     moveArcMotionTempConfigBuilder.setToolSpeed(valueFactory.
      createSpeed(80.0D, Speed.Unit.MM_S));
4     moveArcMotionTempConfigBuilder.setToolAcceleration(valueFactory
      .createAcceleration(15000, Acceleration.Unit.MM_S2));
5     moveArcMotionTempConfigBuilder.setTCPSelection(
      moveArcMotionTemplate.getTCPSelectionFactory().
      createIgnoreActiveTCPSelection());
6     moveArcMotionTempConfigBuilder.setTransitionRadius(valueFactory
      .createLength(50.0D, Length.Unit.MM));
7     moveArcMotionTemplate.setConfig(moveArcMotionTempConfigBuilder.
      build());
```

当然也可以自行进行WaypointNode及DirectionNode的创建及初始化，并插入到Move节点下，但需注意按照EliRobot中Move、Waypoint及Direction节点的特性进行组织，否则组织出的Move节点不合法，EliRobot将出现无法预料的问题。

- Waypoint 节点创建及初始化

```
1   //创建固定位置路点
2   WaypointNode waypointNode = factory.createWaypointNode();
3   WaypointNodeConfigFactory waypointNodeConfigFactory =
    waypointNode.getWaypointNodeConfigFactory();
4   Length transitionRadius = valueFactory.createLength(-60.0D,
    Length.Unit.MM);
5   TransitionSelection transitionSelection =
    waypointNodeConfigFactory.createWaypointTransitionSelection(
    transitionRadius);
6   FixedPositionConfig fixedPositionConfig =
    waypointNodeConfigFactory.createFixedPositionConfig(
    transitionSelection, waypointNodeConfigFactory.
    createInheritedMotion());
7   waypointNode.setConfig(fixedPositionConfig);
8
9   // 创建可变位置路点
10  WaypointNode varWaypointNode = factory.createWaypointNode();
11  TransitionSelection varTransitionSelection =
    waypointNodeConfigFactory.createInheritedTransitionSelection();
12  MotionSelection varMotionSelection = waypointNodeConfigFactory.
    createTimeMotion(valueFactory.createTime(-3.0D, Time.Unit.S));
13  VariablePositionConfig varPositionConfig =
    waypointNodeConfigFactory.createVariablePositionConfig(
    varMotionSelection, varTransitionSelection);
14  varWaypointNode.setConfig(varPositionConfig);
```

- Wait 节点创建及初始化

```
1   // 创建 不等待 类型WaitNode
2   WaitNode noWaitNode = factory.createWaitNode();
3   WaitNodeConfig config = noWaitNode.getWaitNodeConfigFactory().
    createNoWaitConfig();
4   noWaitNode.setConfig(config);
5
6   // 创建 等待时间为1s的 WaitNode
7   WaitNode timeWaitNode = factory.createWaitNode();
8   Time oneSecondWait = CounterTaskNodeView.this.taskApiProvider.
    getValueFactory().createTime(1, Time.Unit.S);
9   TimeConfig timeConfig = timeWaitNode.getWaitNodeConfigFactory()
    .createTimeWaitNodeConfig(oneSecondWait);
10  timeWaitNode.setConfig(timeConfig);
11
12  // 创建 等待数字输入digital_in[0] ?= True的WaitNode
```

```
13      ExpressionService expressionService = taskApiProvider.
        getExpressionService();
14      List<IO> listIO = new ArrayList<>(taskApiProvider.getIOModel().
        getIOs(IOFilterFactory.createDigitalInputFilter()));
15      IO io = listIO.get(0);
16      try{
17          Expression expression = expressionService.
        createExpressionConstructor().appendIOCell(io).appendTokenCell
        ("?=", "==").appendTokenCell("True", "True").construct();
18          ExpressionInputConfig expressionInputConfig = timeWaitNode.
        getWaitNodeConfigFactory().createExpressionConfig(expression);
19          WaitNode expressionInputWaitNode = factory.createWaitNode()
        ;
20          expressionInputWaitNode.setConfig(expressionInputConfig);
21      } catch (Exception exception) {
22          exception.printStackTrace();
23      }
```

- Set 节点创建及初始化

```
1       // 创建 无动作 类型SetNode
2       SetNode noActionSet = factory.createSetNode();
3       SetNodeConfigFactory setNodeConfigFactory = noActionSet.
        getConfigFactory();
4       noActionSet.setConfig(setNodeConfigFactory.createNoActionConfig
        ());
5
6       // 创建 数字输出 SetNode
7       SetNode digitalOutSet = factory.createSetNode();
8       List<IO> listDO = new ArrayList<>(taskApiProvider.getIOModel().
        getIOs(IOFilterFactory.createDigitalOutputFilter()));
9       IO out0 = listDO.get(0);
10      digitalOutSet.setConfig(setNodeConfigFactory.
        createDigitalOutputConfig(out0, DigitalPinValueType.HIGH));
11
12      // 创建 模拟输出 SetNode
13      SetNode analogOutVSet = factory.createSetNode();
14      List<IO> listAO = new ArrayList<>(taskApiProvider.getIOModel().
        getIOs(IOFilterFactory.createAnalogOutputFilter()));
15      IO out_1 = listAO.get(0);
16      AnalogIO analogIO_1 = (AnalogIO) out_1;
17      if (analogIO_1.isVoltage()) {
```

```
18          analogOutVSet.setConfig(setNodeConfigFactory.
        createAnalogOutputVoltageConfig(analogIO_1, valueFactory.
        createVoltage(8.0D, Voltage.Unit.V)));
19      }
20
21      // 创建 模拟输出 SetNode
22      SetNode analogOutCSet = factory.createSetNode();
23      IO out_2 = listAO.get(1);
24      AnalogIO analogIO_2 = (AnalogIO) out_2;
25      if (analogIO_2.isCurrent()) {
26          analogOutCSet.setConfig(setNodeConfigFactory.
        createAnalogOutputCurrentConfig(analogIO_2, valueFactory.
        createCurrent(8.0D, Current.Unit.MA)));
27      }
28
29      // 创建 表达式 SetNode
30      SetNode outSet = factory.createSetNode();
31      List<IO> listMB = new ArrayList<>(taskApiProvider.getIOModel().
        getIOs(IOFilterFactory.createDigitalOutputFilter()));
32      IO mbIO = listMB.get(3);
33      ExpressionService expressionService = CounterTaskNodeView.this.
        taskApiProvider.getExpressionService();
34      try {
35          outSet.setConfig(setNodeConfigFactory.
        createExpressionOutputConfig(mbIO, expressionService.
        createExpressionConstructor().appendIOCell(mbIO).
        appendTokenCell(" ?= ", " == ").appendTokenCell("True", "true")
        .construct()));
36      } catch (IllegalExpressionException e) {
37          e.printStackTrace();
38      }
39
40      // 创建 单脉冲 SetNode
41      SetNode digitalSet = factory.createSetNode();
42      digitalSet.setConfig(setNodeConfigFactory.
        createSinglePulseDigitalOutputConfig((DigitalIO) listDO.get(1),
         valueFactory.createTime(2.0D, Time.Unit.S)));
```

- Popup 节点创建及初始化

```
1       PopupNode popupNode = factory.createPopupNode();
2       popupNode.setMessage("Hello world!");
3       popupNode.setMessageDialogType(MessageType.INFO);
```

- Halt 节点创建

```
HaltNode haltNode = factory.createHaltNode();
```

- Comment 节点创建及初始化

```
CommentNode commentNode = factory.createCommentNode();
commentNode.setCommentString("Hello world!");
```

- Folder 节点创建及初始化

```
FolderNode folderNode = factory.createFolderNode();
folderNode.setDisplay("Hello world!");
```

- Loop 节点创建及初始化

```
// 创建 始终循环 Loop节点
LoopNode alwaysLoopNode = factory.createLoopNode();
LoopNodeConfigFactory loopNodeConfigFactory = alwaysLoopNode.getConfigFactory();
alwaysLoopNode.setConfig(loopNodeConfigFactory.createLoopNodeAlwaysLoopConfig());

// 创建 指定循环次数 Loop节点
LoopNode nTimesLoopNode = factory.createLoopNode();
nTimesLoopNode.setConfig(loopNodeConfigFactory.createLoopNodeCounterConfig(15));

// 创建 表达式循环条件 Loop节点
LoopNode expressionLoopNode = factory.createLoopNode();
ExpressionService expressionService = CounterTaskNodeView.this.taskApiProvider.getExpressionService();
List<IO> listIO = new ArrayList<>(taskApiProvider.getIOModel().getIOs(IOFilterFactory.createAnalogInputFilter()));
IO io = listIO.get(0);
Expression expression;
try{
```

```
17          expression = expressionService.createExpressionConstructor
    ().appendIOCell(io).appendTokenCell(" ?= ", "==").
    appendTokenCell("10", "10").construct();
18          expressionLoopNode.setConfig(loopNodeConfigFactory.
    createExpressionConfig(expression, true));
19      }catch (Exception e) {
20          e.printStackTrace();
21      }
```

- Assignment 节点创建及初始化

```
1       AssignmentNode assignmentNode = factory.createAssignmentNode();
2       AssignmentNodeConfigFactory assignmentNodeConfigFactory =
    assignmentNode.getConfigFactory();
3       ExpressionService expressionService = CounterTaskNodeView.this.
    taskApiProvider.getExpressionService();
4       List<IO> listIO = new ArrayList<>(taskApiProvider.getIOModel().
    getIOs(IOFilterFactory.createAnalogInputFilter()));
5       IO io = listIO.get(0);
6       Expression expression;
7       try{
8           expression = expressionService.createExpressionConstructor
    ().appendIOCell(io).appendTokenCell(" ?= ", "==").
    appendTokenCell("10", "10").construct();
9           TaskVariable variable = CounterTaskNodeView.this.
    contribution.createGlobalVariable("Variable_77");
10          assignmentNode.setConfig(assignmentNodeConfigFactory.
    createExpressionConfig(variable, expression));
11      }catch (Exception e) {
12          e.printStackTrace();
13      }
```

- If 节点创建及初始化

```
1       IfNode ifNode = factory.createIfNode();
2
3       ExpressionService expressionService = CounterTaskNodeView.this.
    taskApiProvider.getExpressionService();
4       List<IO> listIO = new ArrayList<>(CounterTaskNodeView.this.
    taskApiProvider.getIOModel().getIOs(IOFilterFactory.
    createAnalogInputFilter()));
```

```
    IO io = listIO.get(0);
    Expression expression;
    try{
        expression = expressionService.createExpressionConstructor
    ().appendIOCell(io).appendTokenCell(" ?= ", "==").
    appendTokenCell("10", "10").construct();
        ifNode.setExpression(expression);
    }catch (Exception e) {
        e.printStackTrace();
    }

    ElseIfNode elseIfNode = factory.createElseIfNode();
    Expression expressionElseIf;
    try{
        expressionElseIf = expressionService.
    createExpressionConstructor().appendIOCell(io).appendTokenCell
    (" ?= ", "==").appendTokenCell("15", "15").construct();
        elseIfNode.setExpression(expressionElseIf);
    }catch (Exception e) {
        e.printStackTrace();
    }
```

ELITE ROBOTS
艾利特机器人

明天比今天更简单一点

联系我们

商务合作：market@elibot.cn
技术咨询：technical@elibot.cn

- **苏州公司（生产基地）**
  苏州市工业园区长阳街 259 号中新钟园工业坊 4 栋
  +86-400-189-9358
- **北京公司**
  北京市经济技术开发区荣华南路 10 号院 5 号楼 611 室
- **上海公司（研创中心）**
  上海市浦东新区张江人工智能岛川和路 55 弄 20 号楼 3 层
- **深圳公司**
  深圳市宝安区航空路泰华梧桐岛科技创新园 1A 栋 202 室
- **美国公司**
  10521 Research Dr., Ste. 104, 37932, Knoxville, TN (USA)
- **德国公司**
  Münchener Str. 53, 85290, Geisenfeld, Bavaria (Germany)
- **日本公司**
  TOSHIN Hirokoji Honmachi Bldg., 1F, 2-4-3 Sakae, Naka-ku, 460-0008, Nagoya (Japan)
- **墨西哥公司**
  Calzada del pedregal 523, fraccionamiento el pedregal

关注公众号了解更多