CodeArts IDE

用户指南

文档版本 01

发布日期 2026-03-04

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

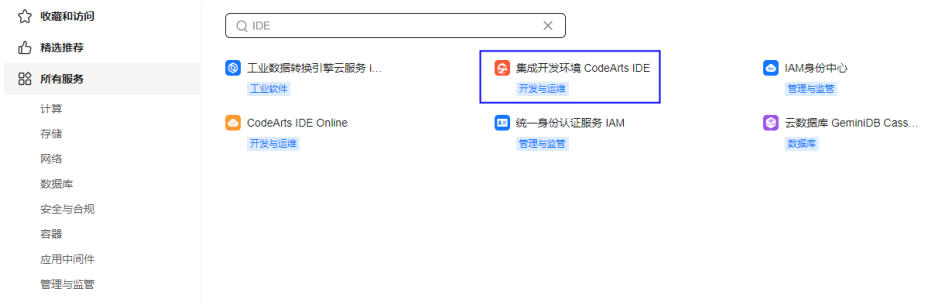
目 录

1 下载并安装 CodeArts IDE 客户端

步骤1 使用华为账号或IAM账号登录管理控制台。

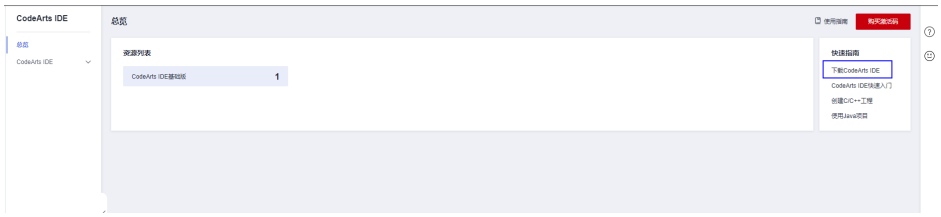
步骤2 在云服务搜索框中搜索“IDE”，单击“集成开发环境 CodeArts IDE”，进入CodeArts IDE管理控制台。

图 1-1 集成开发环境 CodeArts IDE



步骤3 单击右侧“快速指南”模块下的“下载CodeArts IDE”进入客户端下载页面。如下图所示：

图 1-2 下载 CodeArts IDE



步骤4 根据需要下载不同的客户端，目前支持的客户端如下：

- CodeArts IDE for C/C++，如何使用C/C++客户端，请参考**使用CodeArts IDE for C/C++**。
- CodeArts IDE for Remoteshell，如何使用Remoteshell客户端，请参考**使用CodeArts IDE for RemoteShell**。
- CodeArts IDE for Java，如何使用Java客户端，请参考**使用CodeArts IDE for Java**。
- CodeArts IDE for Python，如何使用Python客户端，请参考**使用CodeArts IDE for Python**。

- CodeArts IDE for Cangjie，如何使用Cangjie客户端，请参考**Cangjie产品文档**。

----结束

2 配置 CodeArts IDE 用户权限

2.1 创建用户并授权使用 CodeArts IDE

概述

如果用户需要对所拥有的CodeArts IDE服务进行精细的权限管理，可以使用**统一身份认证服务**（Identity and Access Management，简称IAM），通过IAM，可以：

- 根据企业的业务组织，在华为云账号中，给不同职能部门的员工创建IAM用户，让员工拥有唯一安全凭证使用CodeArts IDE资源。
- 根据企业用户的职能不同，设置不同的访问权限，实现用户之间的权限隔离。
- 将CodeArts IDE资源委托给更专业、高效的其他华为云账号或者云服务，这些账号或者云服务可以根据权限进行代运维。

说明

如果华为云账号可以满足用户对权限管理的需求，则不需要创建独立的IAM用户，可以跳过本章节，不影响使用CodeArts IDE服务的其它功能。

本章节介绍对用户授权的方法，操作流程如**示例流程**所示。

前提条件

已了解用户组可以添加的CodeArts IDE系统策略，并结合实际需求进行选择。

CodeArts IDE支持的系统权限，请参见：**权限说明**。若您需要对除CodeArts IDE之外的其它服务授权，IAM支持服务的所有策略请参见**系统权限**。

示例流程

图 2-1 用户授权 CodeArts IDE 权限操作流程

1. **创建用户组并授权**

   在IAM控制台创建用户组，并授予CodeArts IDE只读权限“CodeArts IDE ReadOnly”。
2. **创建用户并加入用户组**

   在IAM控制台创建用户，并将其加入步骤**1**中创建的用户组。
3. **用户登录**并验证权限

   使用新创建的用户登录控制台，切换至授权区域，通过以下两种方式验证权限是否生效：
   - 在“服务列表”中选择CodeArts IDE，进入CodeArts IDE主界面，单击“购买激活码”，如果无法购买（假设当前权限仅包含CodeArts IDE ReadOnly），表示“CodeArts IDE ReadOnly”已生效。
   - 在“服务列表”中选择除CodeArts IDE外（假设当前权限仅包含CodeArts IDE ReadOnly）的任意服务，若提示权限不足，表示“CodeArts IDE ReadOnly”已生效。

2.2 CodeArts IDE 自定义策略

如果系统预置的权限策略，不满足用户授权需求，CodeArts IDE支持自定义权限策略。自定义权限策略具体创建步骤请参见**创建自定义策略**。

本章为您介绍CodeArts IDE常用的自定义权限策略代码样例。

自定义策略样例

- 授权用户购买、绑定、和查看激活码权限。

```
{
    "Version": "1.1",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "CodeArtsIDE:license:purchase",
        "CodeArtsIDE:license:bind",
        "CodeArtsIDE:license:list"
      ]
    }
  ]
}
```

- 授权用户使用CodeArts IDE所有权限。

```
{
  "Version": "1.1",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "CodeArtsIDE:*:*"
      ]
    }
  ]
}
```

- 禁止用户购买CodeArts IDE激活码。

  用户被授予的策略中，一个授权项的作用如果同时存在Allow和Deny，则遵循Deny优先原则。因此禁止策略需要同时配合其他策略使用，否则没有实际作用。

  例如：如果授予用户CodeArts IDE FullAccess的系统策略，但不希望用户拥有CodeArts IDE FullAccess中定义的购买CodeArts IDE激活码权限，可以创建一条禁止购买CodeArts IDE激活码的自定义策略，同时将CodeArts IDE FullAccess和禁止策略授予用户，根据Deny优先原则，则用户可以对CodeArts IDE执行除了购买CodeArts IDE激活码外的所有操作。禁止策略示例如下：

```
{
  "Version": "1.1",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "CodeArtsIDE:license:purchase"
      ]
    }
  ]
}
```

3 登录 CodeArts IDE 客户端

3.1 使用 IAM 用户登录

前提条件

若没有IAM账号，请参考文档**创建IAM用户**注册或创建。创建IAM用户时，请确认“编程访问”已勾选，开启编程访问权限。

说明

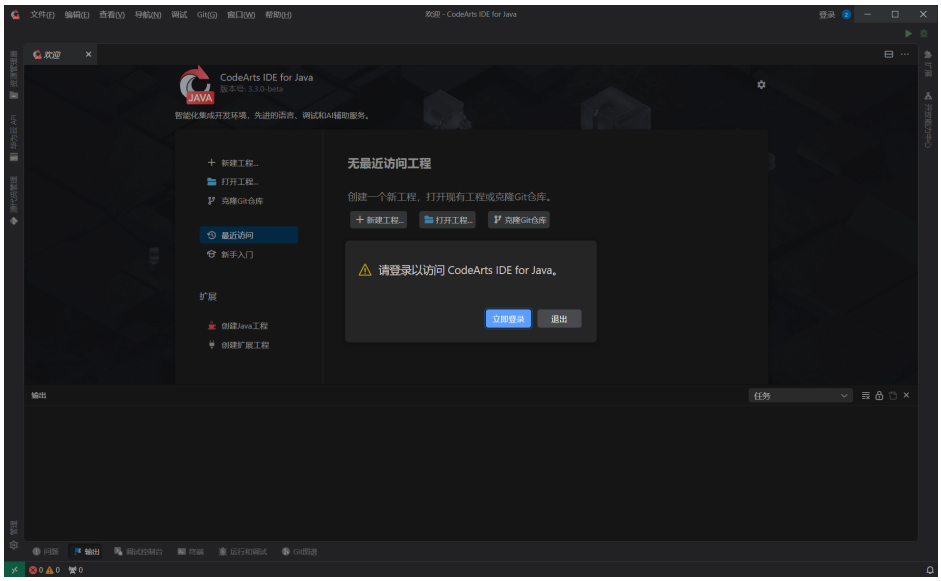
若未开启“编程访问”权限，登录时会出现提示：**此用户仅支持控制台访问，不支持编程访问。请联系管理员添加编程访问的权限。**

说明该IAM账号缺少编程访问权限，可参考**IAM账号如何开启编程访问权限**打开编程访问权限。

操作步骤

步骤1 以CodeArts IDE for Java为例，单击“立即登录”，选择“IAM用户登录”。

图 3-1 立即登录



步骤2 在“IAM用户登录”页面，输入租户名/原华为云账号、IAM用户名/邮件地址和密码。如下图所示：

图 3-2 IAM 用户登录

步骤3 单击“登录”，完成登录。

步骤4 登录成功后即可使用CodeArts IDE客户端。

----结束

3.2 使用华为账号登录

单击IAM用户登录弹窗底部的其他登录方式“华为账号”，即可切换华为账号登录。

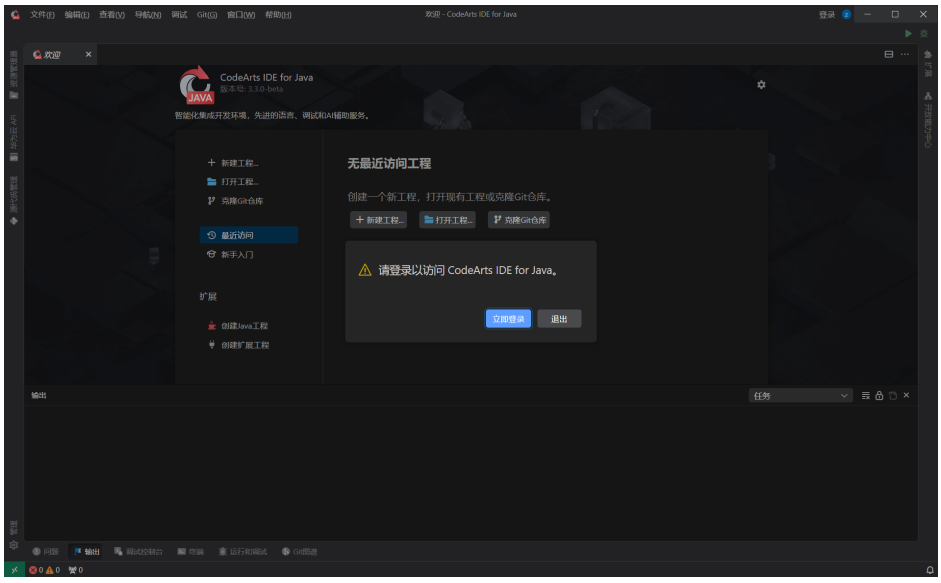
约束与限制

华为账号登录有效期仅为24小时，推荐使用IAM用户登录。

操作步骤

步骤1 以CodeArts IDE for Java为例，单击“立即登录”，选择“华为账号登录”。

图 3-3 立即登录



步骤2 在“华为账号登录”页面，输入手机号/邮件地址/账号名/原华为云账号、和密码。如下图所示：

图 3-4 华为账号登录

步骤3 单击“登录”，登录到CodeArts IDE客户端。

说明

如果用户账号开启了双重验证，单击登录后，将进一步验证用户身份，可以选择以下任一方式验证。

- 发送验证码至已登录账号设备。
- 使用手机号码获取验证码。
- 使用邮箱获取验证码。

获取并输入验证码之后，单击确认，登录到CodeArts IDE客户端。

----结束

4 购买并激活 License

申请激活码

步骤1 进入到**激活码购买页面**后，单击右上角“购买激活码”。

步骤2 根据需要选择是否自动续费和购买激活码的数量，单次购买激活码数量最多为10个。

步骤3 勾选“我已经阅读并同意《CodeArts IDE服务使用说明》”。

步骤4 确认是否绑定当前用户（默认勾选）。勾选绑定当前用户可以帮您省略购买后手动绑定激活码的过程。如下图所示：

图 4-1 激活码购买页面

----结束

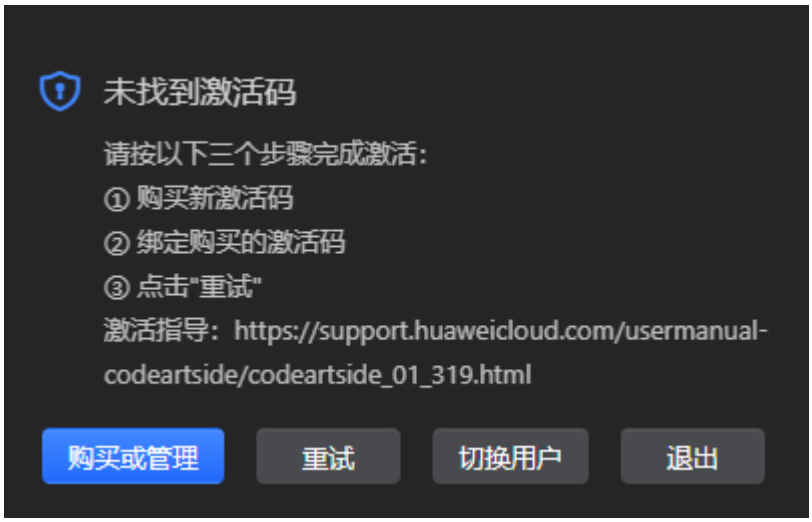
绑定和激活

步骤1 如果在**申请激活码**时勾选了绑定当前用户，可以省略该步骤。如果未勾选，则需要在购买后返回CodeArts IDE控制台。在申请的激活码右侧选择“绑定”，单击后将会帮绑定到当前账号。如下图所示：

图 4-2 绑定激活码

步骤2 完成绑定后，登录到CodeArts IDE客户端。如果出现激活失败，在弹出的提示窗口单击“重试”按钮，或者单击右上角的“立即激活”按钮（在试用期内），完成激活。

图 4-3 单击重试按钮



----结束

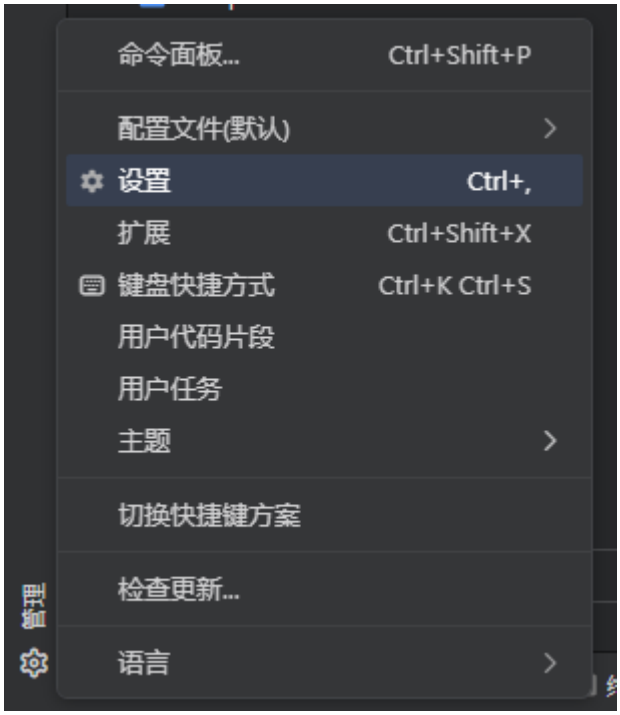
5 配置 CodeArts IDE 开发环境

CodeArts IDE提供了“**设置**”编辑器，用户可以通过执行以下任何操作打开该编辑器。

- 按“**Ctrl+,**”打开“**设置**”编辑器。
- 在左侧导航栏中选择“管理 > 设置”。
- 在“命令”面板（“**Ctrl+Shift+P**”或“**双击Ctrl**”）中，搜索并运行**首选项: 打开设置 (ui)**命令。

在“设置”编辑器中，使用搜索字段来搜索所需的设置。这将显示搜索条件匹配的设置，并过滤掉不匹配的设置。如下图所示：

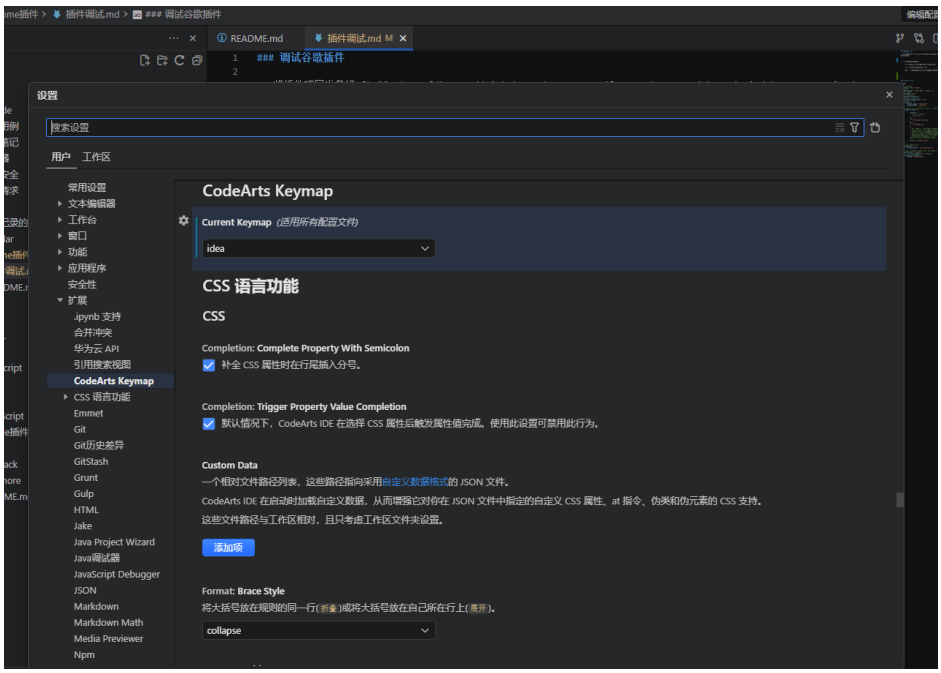
图 5-1 管理菜单



更改设置

CodeArts IDE会自动动态应用对设置的更改。修改后的设置用一条类似于编辑器中修改后的行的蓝线表示。如下图所示：

图 5-2 更改设置



重置设置

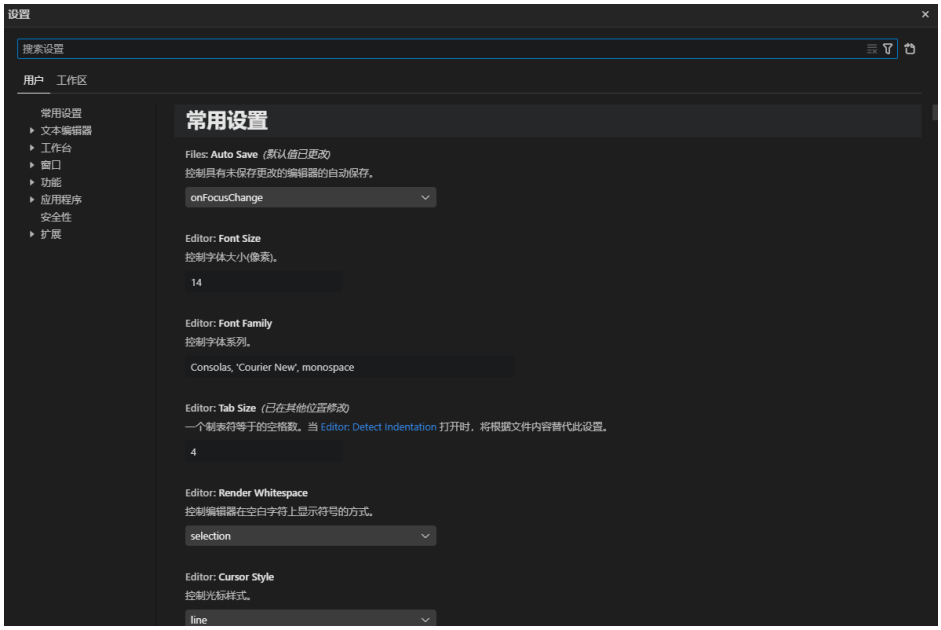
选择一个设置，然后单击“**更多操作**”（  ）按钮，或按“Shift+F9”打开上下文菜单，允许用户将设置重置为其默认值、复制设置ID、将设置复制为JSON文本和将设置应用于所有配置文件。如下图所示：

图 5-3 重置设项

设置组

设置以组表示，以便用户可以轻松导航。顶部的“**常用设置**”组列出了常用的自定义设置项。CodeArts IDE扩展还可以添加自定义设置，这些设置收集在“**扩展**”部分下。如下图所示：

图 5-4 设置组



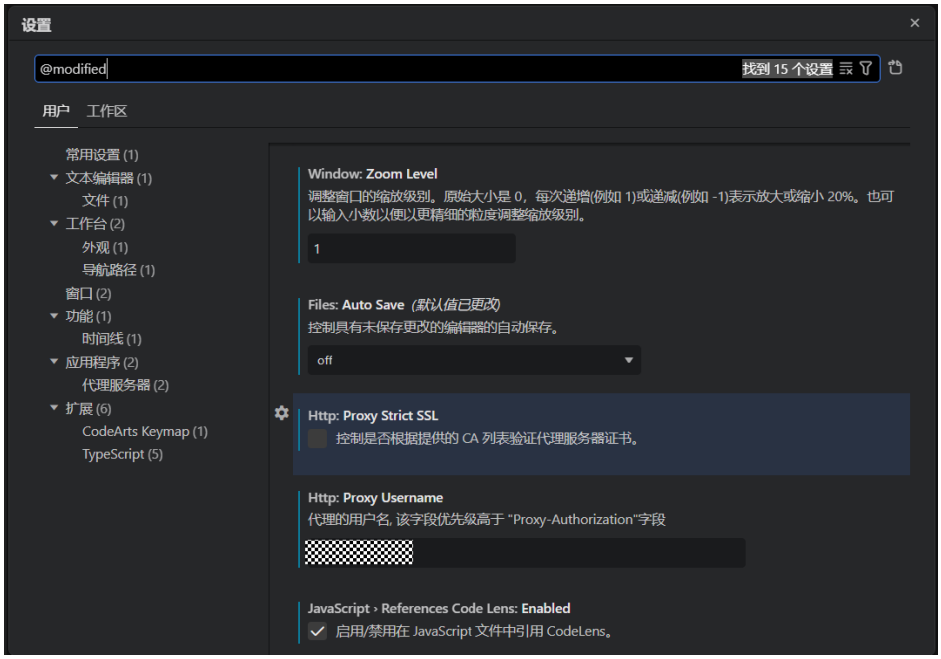
设置筛选器

“设置”编辑器的搜索栏提供了几个筛选器，使用户更容易管理设置。使用搜索栏中的**“筛选器”**按钮（ ）轻松添加筛选器。如下图所示：

图 5-5 筛选器

要检查用户配置的设置，请使用**@modified**的筛选器。如果设置的值与默认值不同，或者其值在相应的设置JSON文件中显式设置，则会显示在此筛选器下。如下图所示：

图 5-6 已修改筛选器



还有几个其他方便的过滤器可以帮助搜索设置：

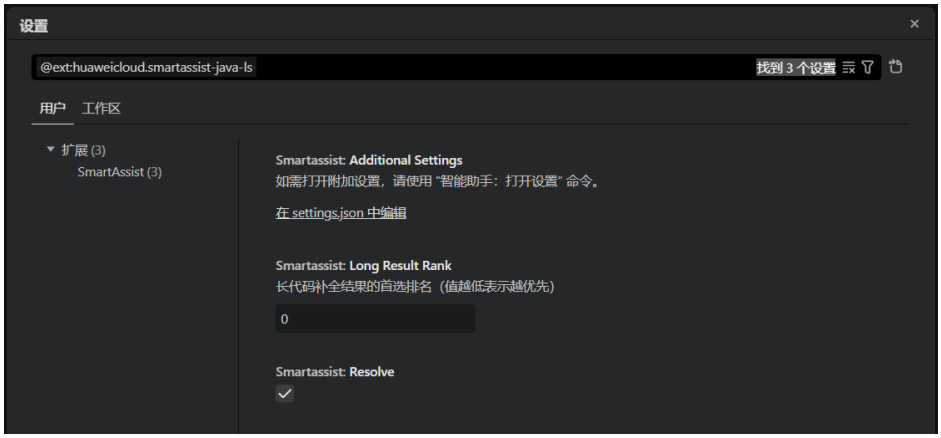
- **@ext**：特定于扩展的设置。提供扩展ID，例如，**@ext:markdown-language-features**。
- **@feature**：特定于功能子组的设置。例如，**@feature:explorer**显示资源管理器的设置。
- **@id**：根据设置ID查找设置。例如，**@id:workbench.activityBar.visible**。
- **@lang**：根据语言ID应用语言筛选器。例如，**@lang:typescript**。
- **@tag**：特定于CodeArts IDE子系统的设置。

搜索栏会记住用户的设置搜索查询，并支持撤销/重做（“Ctrl+Z”/“Ctrl+Shift+Z”/“Ctrl+Y”）。用户可以使用搜索栏右侧的**“清除设置搜索输入”**按钮（ ）快速清除搜索项或筛选器。

扩展设置

安装的CodeArts IDE扩展也可以贡献自己的设置，用户可以在设置编辑器的**“扩展”**部分查看这些设置。如下图所示：

图 5-7 扩展设置



说明

要打开SmartAssist for Java设置，请在“设置”中搜索**@ext:huaweicloud.smartassist-java-ls**。

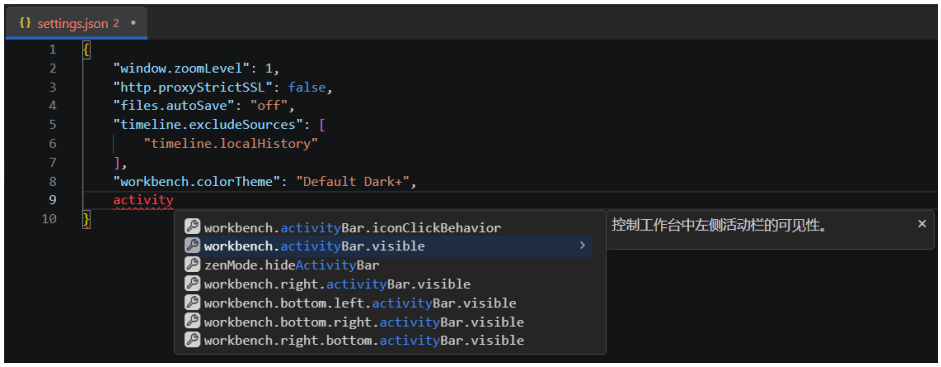
编辑 settings.json 文件

设置编辑器是允许用户查看和修改存储在**settings.json**文件中的设置值的UI。用户可以通过快捷键**“Ctrl+Shift+P”**打开“命令”面板，使用**“首选项: 打开用户设置(JSON)”**命令在编辑器中打开此文件，直接查看和编辑此文件。通过指定设置ID和值，设置将写入JSON。如下图所示：

图 5-8 设置文件

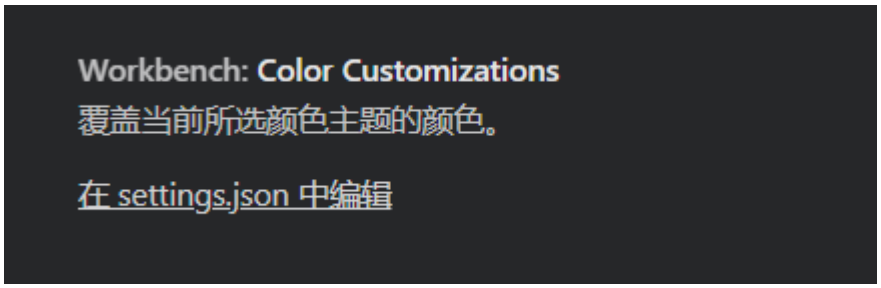
settings.json文件提供设置及其值的代码完成和描述悬停。由于设置名称或JSON格式不正确而导致的错误也会突出显示。如下图所示：

图 5-9 设置文件格式问题



某些设置，例如“**Workbench: Color Customizations**”只能在**settings.json**中编辑。如下图所示：

图 5-10 设置文件



如果用户喜欢始终直接使用**settings.json**，用户可以将“**workbench.settings.editor**”设置为“**json**”，以便将“管理 > 设置”命令和“**Ctrl+,**”键绑定，始终打开**settings.json**文件，而不是设置编辑器。

设置文件位置

不同的子产品，用户设置文件的路径不同。

- CodeArts IDE for C/C++： %APPDATA%\codearts-cpp\User\settings.json
- CodeArts IDE for RemoteShell：%APPDATA%\codearts-remoteshell\User\settings.json
- CodeArts IDE for Java：%APPDATA%\codearts-java\User\settings.json
- CodeArts IDE for Python： %APPDATA%\codearts-python\User\settings.json
- CodeArts IDE for Cangjie： %APPDATA%\codearts-cangjie\User\settings.json

重置所有设置

虽然用户可以通过设置编辑器中的“**重置此设置**”命令单独重置设置，但用户也可以通过打开**settings.json**并删除大括号**{}**之间的条目来重置所有更改的设置。

说明

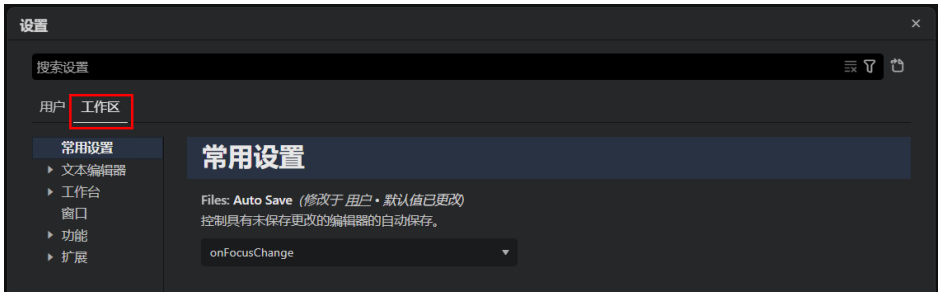
当用户通过清除**settings.json**重置设置时，无法恢复其以前的值。

设置工作区

工作区设置与项目一起存储，因此可以在项目的开发人员之间共享。工作区设置覆盖用户设置。

用户可以在设置编辑器的**“工作区”**选项卡上编辑工作区设置。要快速打开此选项卡，请使用“命令面板”（“Ctrl+Shift+P”/“Ctrl+Ctrl”）中的**首选项: 打开工作区设置**。如下图所示：

图 5-11 设置工作区



并非所有用户设置都可用作工作区设置，例如，与更新和安全性相关的应用程序范围的设置不能被工作区设置覆盖。

6 配置 CodeArts IDE 快捷键

6.1 快捷键功能简介

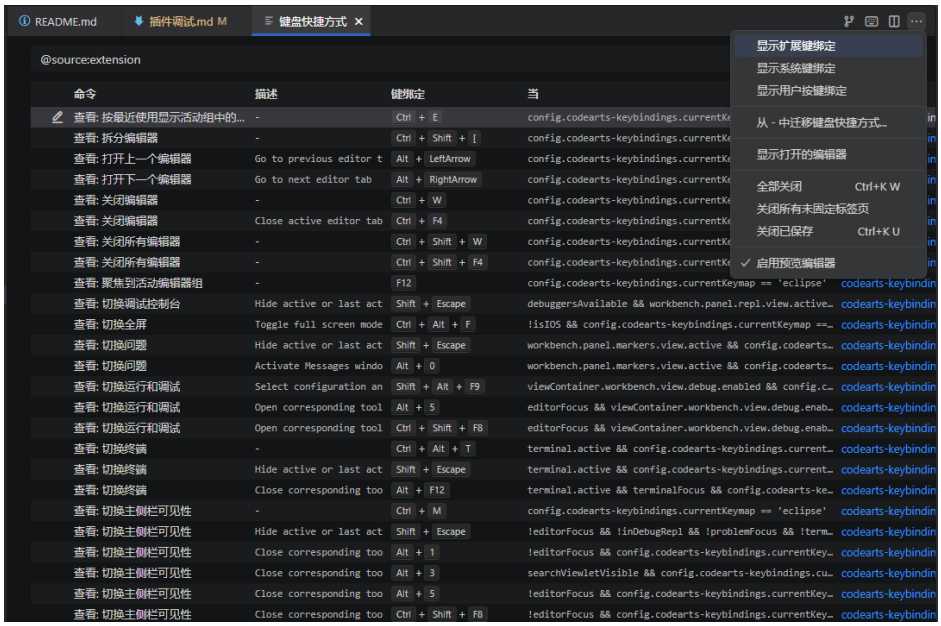
CodeArts IDE中提供了丰富的快捷键功能，可以根据需求查看或修改快捷键方案。

如需查看当前CodeArts IDE的快捷键方案，单击CodeArts IDE左下角的**管理**菜单。并选择**键盘快捷键方式**选项，或者通过快捷键：“Ctrl+K Ctrl+S”直接唤起**键盘快捷方式**编辑器。如下图所示：

图 6-1 键盘快捷方式

打开**键盘快捷键方式**编辑器后，您可以在**更多操作**菜单选择**显示默认按键绑定**，这将在**键盘快捷键**编辑器中应用“@source:extension”过滤器。如下图所示：

图 6-2 键盘快捷键编辑器



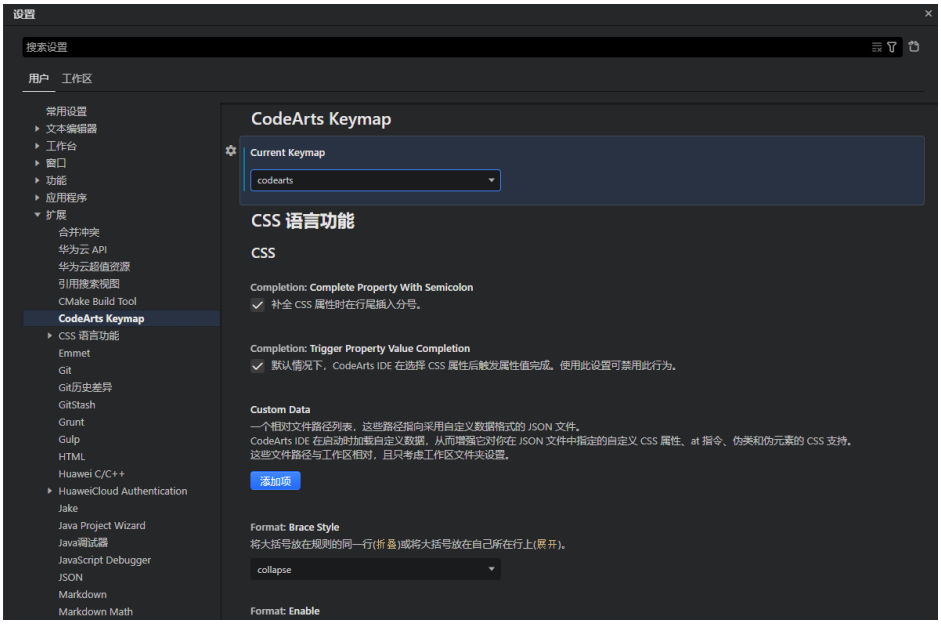
可以通过使用**首选项：打开默认键盘快捷键(JSON)**命令，将默认键盘快捷方式视为JSON文件进行查看。

CodeArts IDE默认提供了两个预定义的键位映射：本机CodeArts IDE和IDEA，后者使用IntelliJ IDEA中对应的映射覆盖了一些最常用的快捷方式。如果您习惯在IntelliJ IDEA中工作，您可以选择IDEA键位映射，从而继续使用熟悉的快捷方式。

要在键位映射之间切换，请执行以下任一操作：

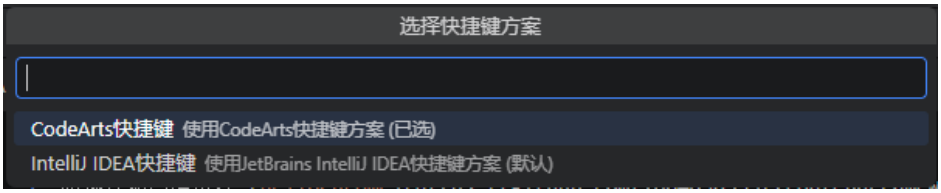
- 在**设置**编辑器（“Ctrl+,”）中，转到**扩展**>**CodeArts Keymap**。然后从**Current Keymap**列表中选择所需的键位映射。如下图所示：

图 6-3 扩展 Codearts Keymap 配置项



- 单击“管理 > 切换快捷键方案”菜单，弹出弹窗选择快捷键方案。如下图所示：

图 6-4 选择快捷方案弹窗

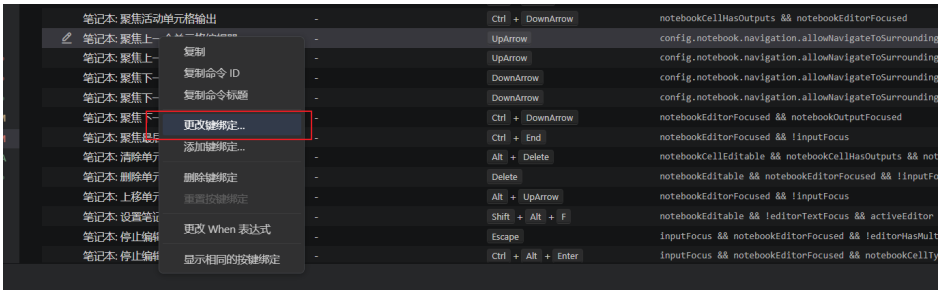


6.2 配置快捷键

您可以在**键盘快捷方式**编辑器中通过两种方式修改默认的快捷键。

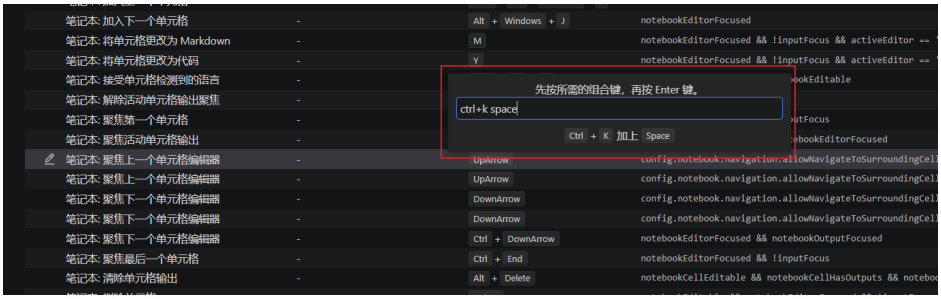
- 右键单击您想要修改的快捷键，选择**更改键绑定菜单**。如下图所示：

图 6-5 更改键绑定菜单



- 双击该快捷键方案所在的行，在唤起的弹窗中输入想要绑定的新快捷键。如下图所示：

图 6-6 更改键绑定弹窗



查看和重置快捷键

查看CodeArts IDE中已修改的键盘快捷键，单击**“更多操作”**按钮（ ）并在展示的菜单中选择**“显示用户按键绑定”**。这将在“键盘快捷方式”编辑器中应用“@source:user”过滤器（**Source**为“user”）。如下图所示：

图 6-7 显示用户按键绑定

要重置快捷键绑定，执行以下操作之一：

- 重置单个按键绑定，右键单击列表中相应的条目，选择“**重置按键绑定**”。如下图所示：

  图 6-8 重置按键绑定

- 要一次性重置所有按键绑定，请找到用户的**keybindings.json**文件并清空其内容。默认情况下，该文件位于“%USERPROFILE%\AppData\Roaming\codearts-java\User\keybindings.json”。可以单击**打开键盘快捷方式(JSON)**按钮打开**keybindings.json**文件，如下图所示：

图 6-9 打开键盘快捷方式(JSON)

6.3 文件菜单快捷键

在CodeArts IDE编辑器界面，在**文件(F)**菜单下，可以使用如下快捷键。如下图所示：

图 6-10 文件菜单快捷键

表 6-1 文件菜单快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
新建文件	Ctrl+N	Alt+Insert	workbench.action.files.newUntitledFile
新建工程	Alt+P	Alt+P	workbench.action.openProjectWizard
打开文件	Ctrl+O	Ctrl+O	workbench.action.files.openFile
打开工程	Ctrl+K Ctrl+O	Ctrl+K O	workbench.action.files.showOpenedFileInNewWindow
关闭编辑器	Ctrl+W Ctrl+F4	Ctrl+F4	workbench.action.closeActiveEditor
关闭工程	Ctrl+K F	Ctrl+K F	workbench.action.closeFolder
保存	Ctrl+S	Ctrl+S	workbench.action.files.save
另存为	Ctrl+Shift+S	Ctrl+Shift+S	workbench.action.files.saveAs
全部保存	Ctrl+K S	Ctrl+K S	saveAll
设置	Ctrl+,	Ctrl+,	workbench.action.openSettings
打开文件	Ctrl+O	Ctrl+O	workbench.action.files.openFile
关闭组	Ctrl+K W	Ctrl+K W	workbench.action.closeEditorsInGroup
关闭所有	Ctrl+K Ctrl+W	Ctrl+K Ctrl+W	workbench.action.closeAllEditors
重新打开关闭的编辑器	Ctrl+Shift+T	Ctrl+Shift+T	workbench.action.reopenClosedEditor
保持编辑器打开	Ctrl+K Enter	Ctrl+K Enter	workbench.action.keepEditor
在Windows中显示活动文件	Ctrl+K R	Ctrl+K R	workbench.action.files.revealActiveFileInWindows
在新窗口中显示打开的文件	Ctrl+K O	Ctrl+K O	workbench.action.files.showOpenedFileInNewWindow
比较打开的文件	--	Ctrl+D	workbench.files.action.compareFileWith

6.4 编辑菜单快捷键

在CodeArts IDE编辑器界面，在**编辑(E)**菜单下，可以使用如下快捷键。如下图所示：

图 6-11 编辑菜单快捷键

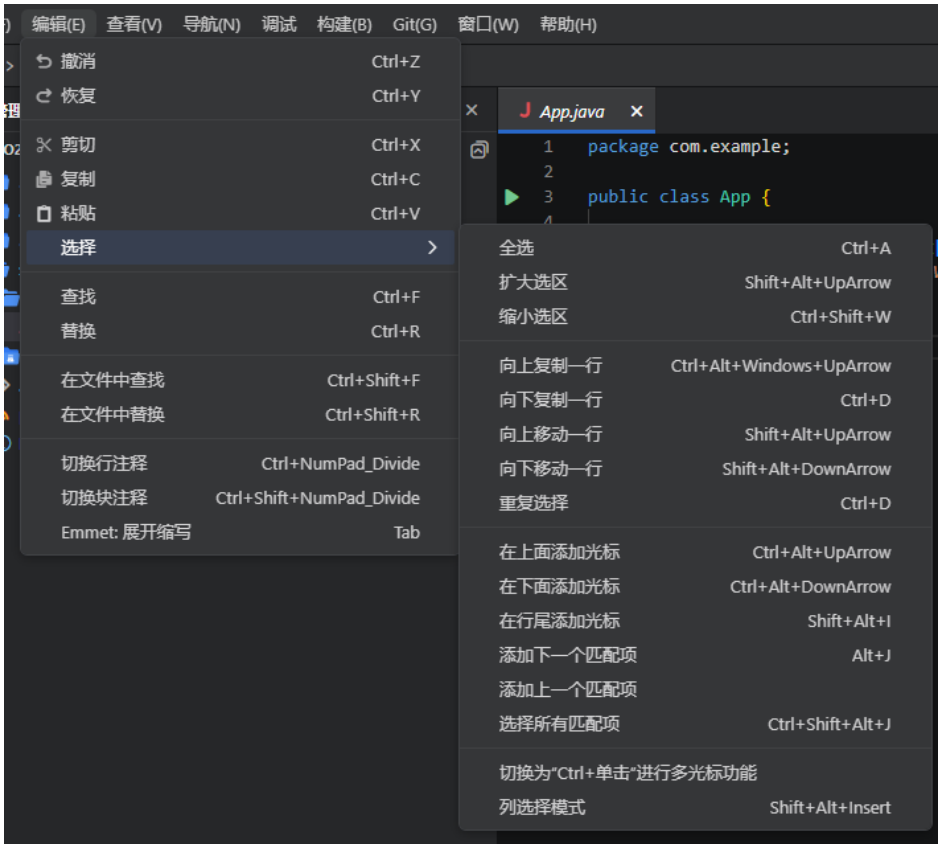


表 6-2 编辑菜单快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
撤销	Ctrl+Z	Ctrl+Z	undo
恢复	Ctrl+Shift+Z Ctrl+Y	Ctrl+Shift+Z Ctrl+Y	redo
剪切	Shift+Delete Ctrl+X	Shift+Delete Ctrl+X	editor.action.clipboardCutAction
复制	Ctrl+C Ctrl+Insert	Ctrl+C Ctrl+Insert	editor.action.clipboardCopyAction

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
粘贴	Ctrl+V Shift+Insert	Ctrl+V Shift+Insert	editor.action.clipboardPasteAction
全选	Ctrl+A	Ctrl+A	list.selectAll
列选择模式	--	Shift+Alt+Insert	editor.action.toggleColumnSelection
在文件中查找	Ctrl+Shift+F	Ctrl+Shift+F	omnisearch.open.file
在文件中替换	Ctrl+Shift+R	Ctrl+Shift+R	rerunSearchEditorSearch

代码编辑快捷键

表 6-3 编码快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
触发代码完成	Ctrl+I Ctrl+Space Ctrl+Shift+Space	Ctrl+Shift+Space	editor.action.triggerSuggest
触发参数提示	Ctrl+Shift+Space	Ctrl+P	editor.action.triggerParameterHints
格式化文档	Shift+Alt+F	Ctrl+Alt+L	editor.action.formatDocument
格式选择	Ctrl+K Ctrl+F	Ctrl+Alt+L	editor.action.formatSelection
快速信息	Ctrl+K Ctrl+I	Ctrl+K Ctrl+I	editor.action.showHover
向侧面开放定义	Ctrl+K F12 Ctrl+K Ctrl+F12	Ctrl+K F12 Ctrl+K Ctrl+F12	editor.action.revealDefinitionAside
快速解决	Ctrl+.	Alt+Enter	editor.action.quickFix
扩大选择	Shift+Alt+Right	Shift+Alt+Right	editor.action.smartSelect.expand
缩小选择	Shift+Alt+Left	Ctrl+Shift+W	editor.action.smartSelect.shrink

代码搜索快捷键

表 6-4 搜索类快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
在文件中替换	Ctrl+Shift+H	Ctrl+Shift+R	omnisearch.open.file.replace
切换匹配大小写	Alt+C	Alt+C	toggleSearchCaseSensitive
切换全字匹配	Alt+W	Alt+W	toggleSearchWholeWord
Toggle使用正则表达式	Alt+R	Alt+R	toggleSearchRegex
切换搜索详细信息	Ctrl+Shift+J	Ctrl+Shift+J	workbench.action.search.toggleQueryDetails
聚焦下一个搜索结果	F4	F4	search.action.focusNextSearchResult
聚焦上一个搜索结果	Shift+F4	Shift+F4	search.action.focusPreviousSearchResult
显示下一个搜索词	Alt+Down Down	Alt+Down Down	history.showNext
显示上一个搜索词	Alt+Up Up	Alt+Up Up	history.showPrevious
在编辑器中打开结果	Alt+Enter	Alt+Enter	search.action.openInEditor
焦点搜索编辑器输入	Escape	Escape	search.action.focusQueryEditorWidget
删除文件结果	Ctrl+Shift+Backspace	Ctrl+Shift+Backspace	search.searchEditor.action.deleteFileResults

代码导航快捷键

表 6-5 导航相关快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
显示所有符号	Ctrl+T	Ctrl+T	workbench.action.showAllSymbols
前往线路	Ctrl+G	Ctrl+G	workbench.action.gotoLine

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
转到文件，快速打开	Ctrl+E Ctrl+P	Ctrl+Shift+N	workbench.action.quickOpen
转到符号	Ctrl+Shift+O	Ctrl+Shift+O	workbench.action.gotoSymbol
转到定义	F12	F12	editor.action.revealDefinition
前往声明	Ctrl+F12 F12	Ctrl+F12 F12	editor.action.goToDeclaration
前往实现	Ctrl+F12	Ctrl+Alt+B	editor.action.goToImplementation
转到类型	Shift+Alt+T Ctrl+Shift+O Ctrl+T	Ctrl+N Ctrl+Shift+Alt+N	workbench.action.smartSearchTypes
查找用法	Shift+Alt+F12	Alt+F7	references-view.findReferences
显示问题	Ctrl+Shift+M	Shift+Escape Alt+0	workbench.actions.view.problems
转到下一个错误或警告	Alt+F8	F2	editor.action.marker.next
转到上一个错误或警告	Shift+Alt+F8	Shift+F2	editor.action.marker.prev
显示所有命令	Ctrl+Shift+P Ctrl Ctrl	Ctrl+Shift+P Ctrl Ctrl	omnisearch.open.command
回退	Alt+Left	Ctrl+Alt+Left	workbench.action.navigateBack
前进	Alt+Right	Ctrl+Alt+Right	workbench.action.navigateForward
打开Settings	Ctrl+,	Ctrl+,	workbench.action.openSettings
打开Keyboard Shortcuts	Ctrl+K Ctrl+S	Ctrl+K Ctrl+S	workbench.action.openGlobalKeybindings
选择Color Theme	Ctrl+K Ctrl+T	Ctrl+`	workbench.action.selectTheme

代码重构快捷键

表 6-6 重构快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
查看可用的重构	Ctrl+Shift+R	Ctrl+Shift+Alt+T	editor.action.refactor
复制类	Alt+F6	F5	refactor.copy.class
安全删除	Alt+Delete	Alt+Delete	refactor.safe.delete
重命名符号	F2	Shift+F6	editor.action.rename
移动	--	F6	refactor.move
移动类	F6	F6	refactor.move.classes
引入变量	Ctrl+Alt+V	Ctrl+Alt+V	refactor.extract.variable
提取方法	Ctrl+Shift+Alt+M	Ctrl+Shift+Alt+M	refactor.extract.method
介绍领域	Ctrl+Shift+Alt+F	Ctrl+Shift+Alt+F	refactor.extract.field
引入常数	Ctrl+Alt+C	Ctrl+Alt+C	refactor.extract.constant
介绍参数	Ctrl+Shift+Alt+P	Ctrl+Shift+Alt+P	refactor.introduce.parameter
内联变量	Ctrl+Alt+N	Ctrl+Alt+N	refactor.inline.variable
内联参数	--	Ctrl+Shift+Alt+P	refactor.inline.parameter
内联方法	Ctrl+Shift+Alt+L	Ctrl+Shift+Alt+L	refactor.inline.method
更改签名	Ctrl+F6	Ctrl+F6	refactor.change.signature

6.5 查看菜单快捷键

在CodeArts IDE编辑器界面，在**查看(V)**菜单下，可以使用如下快捷键。如下图所示：

图 6-12 查看菜单快捷键

表 6-7 查看菜单快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
工程	Ctrl+Shift+E	Alt+1	workbench.view.explorer

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
源代码管理	Ctrl+Shift+G	Alt+9	workbench.view.scm
扩展	Ctrl+Shift+X	Ctrl+Shift+X	workbench.view.extensions
问题	Ctrl+Shift+M	Shift+Escape Alt+0	workbench.actions.view.problems
输出	Ctrl+Shift+U	Ctrl+Shift+U	workbench.action.output.toggleOutput
调试控制台	Ctrl+Shift+Y	Shift+Escape	workbench.debug.action.toggleRepl
运行	Ctrl+Shift+D	Shift+Alt+F9 Ctrl+Shift+F8 Alt+5	workbench.view.debug
终端	Ctrl+`	Shift+Escape Alt+F12	workbench.action.terminal.toggleTerminal
自动换行	Alt+Z	Alt+Z	editor.action.toggleWordWrap
全屏	F11	Ctrl+Alt+F	workbench.action.toggleFullScreen
禅模式	Ctrl+K Z	Ctrl+K Z	workbench.action.toggleZenMode
显示右侧栏	Ctrl+Alt+B	Ctrl+Alt+B	workbench.action.toggleAuxiliaryBar
显示左侧栏	Ctrl+B	Shift+Escape Alt+1 Alt+3 Ctrl+Shift+F8 Alt+9	workbench.action.toggleSidebarVisibility
放大	Ctrl+Numpad+ Ctrl+Shift+= Ctrl+=	Ctrl+Numpad+ Ctrl+Shift+= Ctrl+=	workbench.action.zoomIn
缩小	Ctrl+Numpad- Ctrl+Shift+- Ctrl+-	Ctrl+Numpad- Ctrl+Shift+- Ctrl+-	workbench.action.zoomOut

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
重置缩放	Ctrl +Numpad0	Ctrl +Numpad0	workbench.action.zoomReset

6.6 导航菜单快捷键

在CodeArts IDE编辑器界面，在**导航(N)**菜单下，可以使用如下快捷键。如下图所示：

图 6-13 导航菜单快捷键

表 6-8 导航菜单快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
显示所有符号	Ctrl+T	Ctrl+T	workbench.action.showAllSymbols
前往线路	Ctrl+G	Ctrl+G	workbench.action.gotoLine
转到文件，快速打开	Ctrl+E Ctrl+P	Ctrl+Shift+N	workbench.action.quickOpen
转到符号	Ctrl+Shift+O	Ctrl+Shift+O	workbench.action.gotoSymbol
转到定义	F12 Ctrl+F12	F12 Ctrl+F12	editor.action.revealDefinition
前往声明	--	Alt+F11 F4 Ctrl+Enter Ctrl+B	editor.action.goToDeclaration
转到类型	Shift+Alt+T Ctrl+Shift+O Ctrl+T	Ctrl+N Ctrl+Shift+Alt+N	workbench.action.smartSearchTypes
查找用法	Shift+Alt+F12	Alt+F7 Ctrl+Shift+G	references-view.findReferences
显示问题	Ctrl+Shift+M	Shift+Escape Alt+0	workbench.actions.view.problems
转到下一个错误或警告	Alt+F8	Ctrl+` F2	editor.action.marker.next
转到上一个错误或警告	Shift+Alt+F8	Shift+F2	editor.action.marker.prev
显示所有命令	Ctrl+Shift+P Ctrl Ctrl	Ctrl+Shift+P Ctrl Ctrl	omnisearch.open.command
回退	Alt+Left	Ctrl+Alt+Left	workbench.action.navigateBack
前进	Alt+Right	Ctrl+Alt+Right	workbench.action.navigateForward
打开Settings	Ctrl+,	Ctrl+,	workbench.action.openSettings

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
打开Keyboard Shortcuts	Ctrl+K Ctrl+S	Ctrl+K Ctrl+S	workbench.action.openGlobalKeybindings
选择Color Theme	Ctrl+K Ctrl+T	Ctrl+`	workbench.action.selectTheme

6.7 调试菜单快捷键

在CodeArts IDE编辑器界面，在**调试(D)**菜单下，可以使用如下快捷键。如下图所示：

图 6-14 调试菜单快捷键

表 6-9 调试快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
切换断点	F9	Ctrl+F8	editor.debug.action.toggleBreakpoint
开始	F5	Shift+F9	workbench.action.debug.start
继续	F5	F9	workbench.action.debug.continue
启动（不调试）	Ctrl+F5	Shift+F10	workbench.action.debug.run
暂停	F6	F6	workbench.action.debug.pause
步入	F11	F7	workbench.action.debug.stepInto

6.8 Git 版本控制快捷键

表 6-10 版本控制快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
拉取	--	Ctrl+T	git.pull
全部提交	--	Ctrl+Alt+K	git.commitAll
暂存选定的范围	Ctrl+K Ctrl+Alt+S	Ctrl+K Ctrl+Alt+S	git.stageSelectedRanges
取消暂存选定范围	Ctrl+K Ctrl+N	Ctrl+K Ctrl+N	git.unstageSelectedRanges
还原所选更改	Ctrl+K Ctrl+R	Ctrl+Alt+Z	git.revertSelectedRanges

6.9 管理菜单快捷键

在CodeArts IDE编辑器界面，在**管理**菜单下，可以使用如下快捷键。如下图所示：

图 6-15 管理菜单快捷键

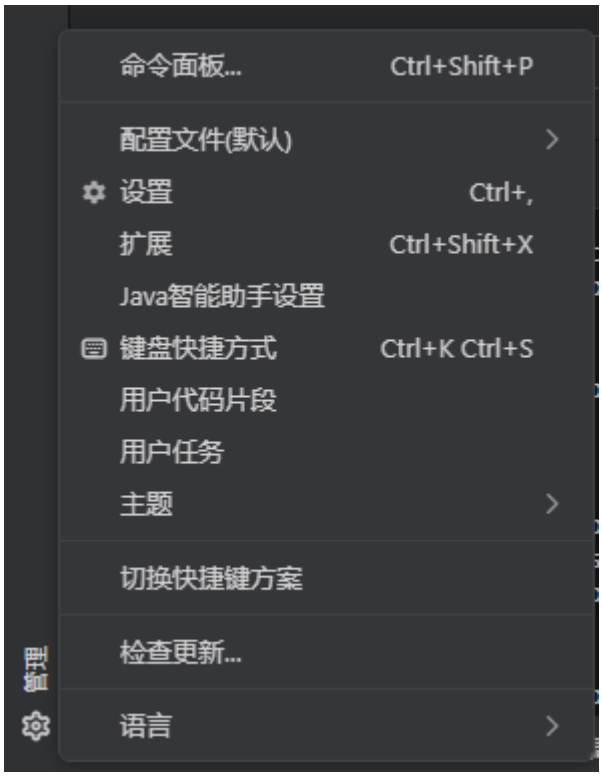


表 6-11 管理菜单快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
设置	Ctrl+,	Ctrl+,	workbench.action.openSettings
扩展	Ctrl+Shift+X	Ctrl+Shift+X	workbench.view.extensions
键盘快捷方式	Ctrl+K Ctrl+S	Ctrl+K Ctrl+S	workbench.action.openGlobalKeybindings
颜色主题	Ctrl+K Ctrl+T	Ctrl+`	workbench.action.selectTheme

6.10 编辑器快捷键

表 6-12 编辑器管理快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
关闭窗口	Ctrl+Shift+W Alt+F4	Ctrl+Shift+W Alt+F4	workbench.action.closeWindow

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
关闭编辑器	Ctrl+W Ctrl+F4	Ctrl+F4	workbench.action.closeActiveEditor
关闭文件夹	Ctrl+K F	Ctrl+K F	workbench.action.closeFolder
分割编辑器	Ctrl+\	Ctrl+\	workbench.action.splitEditor
聚焦第一编辑组	Ctrl+1	Ctrl+1	workbench.action.focusFirstEditorGroup
聚焦第二编辑组	Ctrl+2	Ctrl+2	workbench.action.focusSecondEditorGroup
聚焦第三编辑组	Ctrl+3	Ctrl+3	workbench.action.focusThirdEditorGroup
向左移动编辑器	Ctrl+Shift+Pageup	Ctrl+Shift+Pageup	workbench.action.moveEditorLeftInGroup
向右移动编辑器	Ctrl+Shift+Pagedown	Ctrl+Shift+Pagedown	workbench.action.moveEditorRightInGroup
向左移动活动编辑器组	Ctrl+K Left	Ctrl+K Left	workbench.action.moveActiveEditorGroupLeft
向右移动活动编辑器组	Ctrl+K Right	Ctrl+K Right	workbench.action.moveActiveEditorGroupRight
将编辑器移至下一组	Ctrl+Alt+Right	Ctrl+Alt+Right	workbench.action.moveEditorToNextGroup
将编辑器移动到上一个组	Ctrl+Alt+Left	Ctrl+Alt+Left	workbench.action.moveEditorToPreviousGroup

屏幕显示快捷键

表 6-13 屏幕显示相关快捷键

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
切换全屏	F11	Ctrl+Alt+F	workbench.action.toggleFullScreen
切换禅宗模式	Ctrl+K Z	Ctrl+K Z	workbench.action.toggleZenMode
离开禅宗模式	Escape Escape	Escape Escape	workbench.action.exitZenMode
放大	Ctrl+Numpad+ Ctrl+Shift+= Ctrl+=	Ctrl+Numpad+ Ctrl+Shift+= Ctrl+=	workbench.action.zoomIn
缩小	Ctrl+Numpad- Ctrl+Shift+- Ctrl+-	Ctrl+Numpad- Ctrl+Shift+- Ctrl+-	workbench.action.zoomOut
显示资源管理器	Ctrl+Shift+E	Alt+1	workbench.view.explorer
显示搜索	Ctrl+Shift+F	Ctrl+Shift+F	omnisearch.open.file
显示源代码控制	Ctrl+Shift+G	Alt+9	workbench.view.scm
显示运行	Ctrl+Shift+D	Shift+Alt+F9 Alt+5 Ctrl+Shift+F8	workbench.view.debug
显示扩展	Ctrl+Shift+X	Ctrl+Shift+X	workbench.view.extensions
显示输出	Ctrl+Shift+U	Ctrl+Shift+U	workbench.action.output.toggleOutput
切换集成终端	Ctrl+`	Shift+Escape Alt+F12	workbench.action.terminal.toggleTerminal

7 配置 Git 版本管理

7.1 源代码控制界面介绍

本章节主要介绍的是Git，但大多数源代码控制界面和工作流程在其他源代码管理系统中也是通用的。

如果您对Git还不熟悉，可以从**git-scm**网站开始，那里有一本流行的**在线书籍**和**入门视频**。

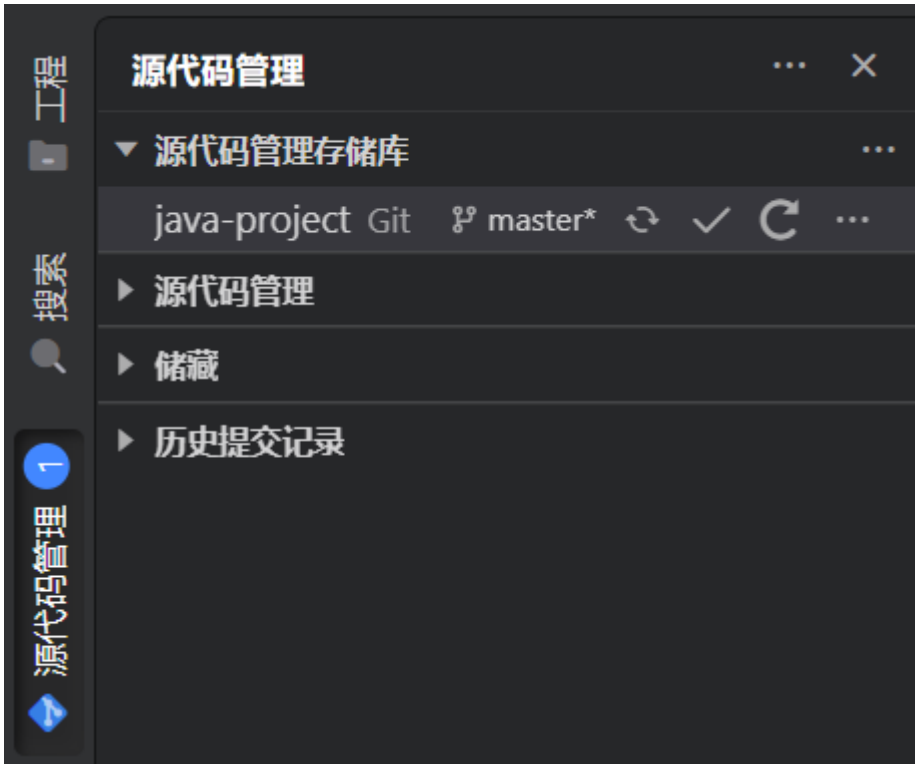
在使用CodeArts IDE的源代码管理（SCM）功能前，请确保您的机器已安装Git 2.0.0或更高版本。

大多数与源代码管理相关的操作可以在“源代码管理”视图中执行。要打开它，请执行以下任一操作：

- 单击左侧导航栏中的**“源代码管理”**按钮（ ）。
- 在主菜单中，选择“查看 > 源代码管理”。
- 按**“Ctrl+Shift+G”**或**“Alt+9”**(IDEA快捷键方案)。

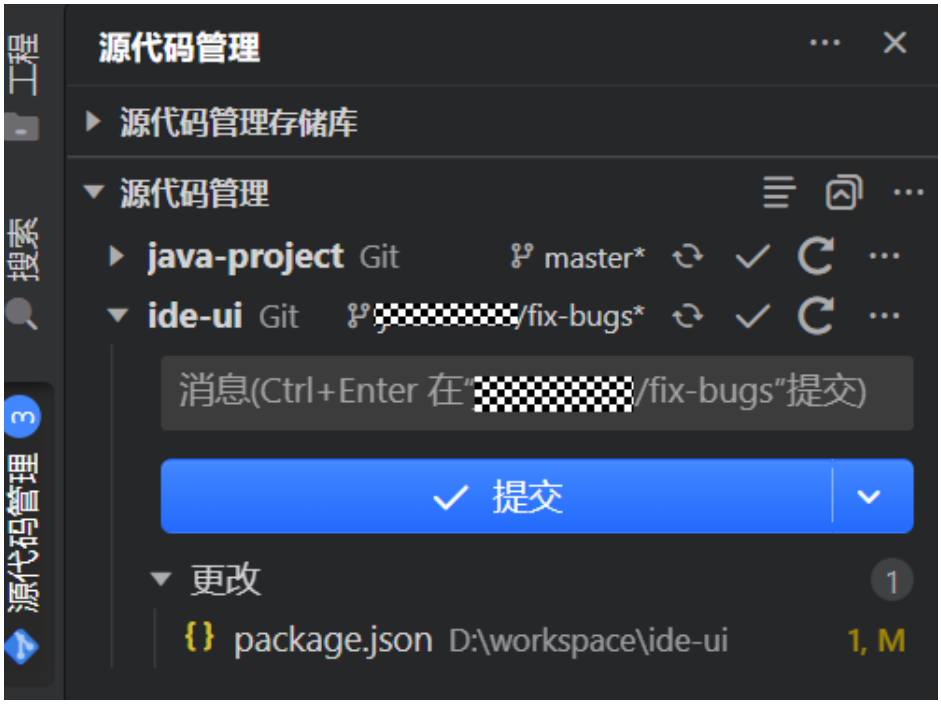
“源代码管理存储库”部分列出了当前工作区中检测到的存储库。如下图所示：

图 7-1 源代码管理存储库



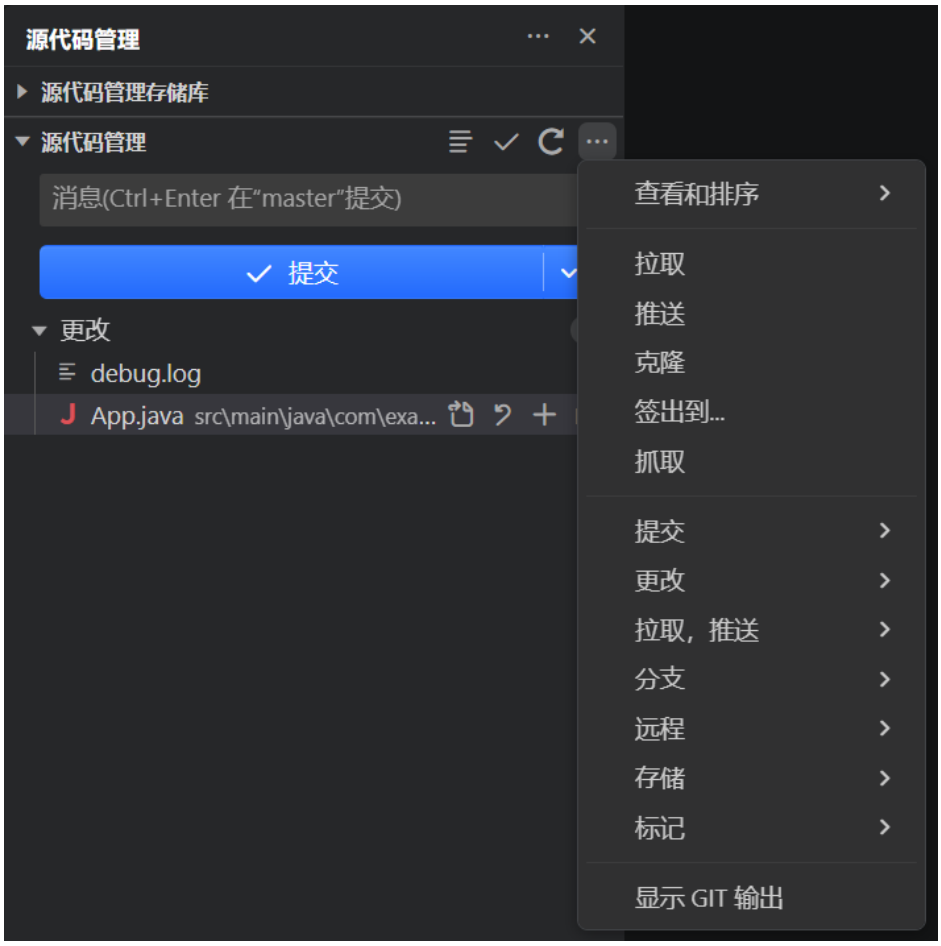
“源代码管理”部分显示了打开的存储库中的更改。可以通过选择需要操作的存储库来限定更改的显示。如下图所示：

图 7-2 源代码管理仓库更改



在源代码管理部分的“**更多操作**”（ ）菜单中，用户可以快速访问大多数与“**源代码管理**”相关的操作。如下图所示：

图 7-3 源代码管理更多操作



装订线指示器

当用户对一个被纳入源代码控制的文件进行更改时，CodeArts IDE会在装订线和概览标尺上添加注释。

- 红色三角形( )表示已删除的行。
- 绿色条( )表示新增的行。
- 蓝色条( )表示修改的行。

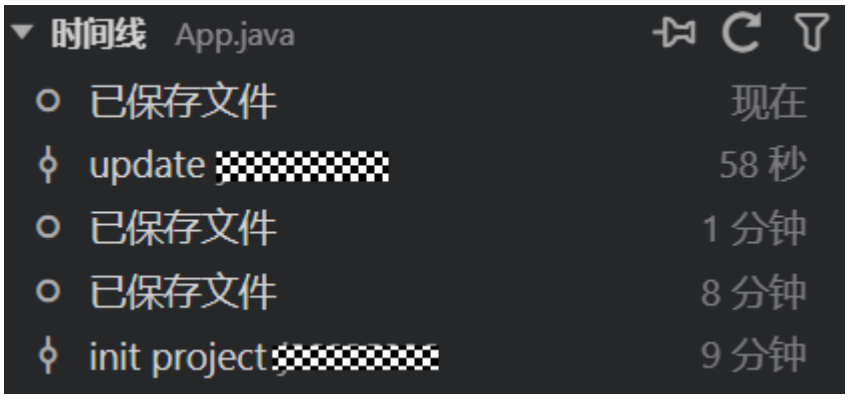
如下图所示：

图 7-4 装订线指示器

时间线视图

“时间线”视图可以在默认情况下通过左侧导航中的资源管理器中（ ）访问，它是用于可视化当前在代码编辑器中打开的文件的时间序列事件的统一视图。如下图所示：

图 7-5 时间线视图



以下是对“Git历史记录”的操作：

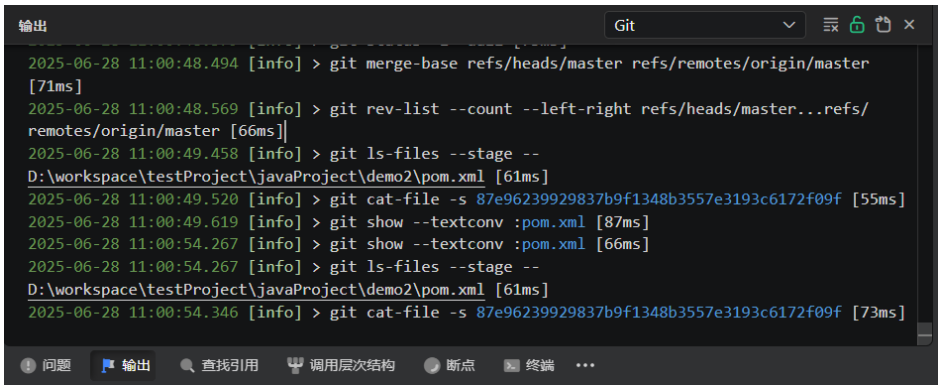
- 单击提交记录以展示当前文件在本次提交的更改。
- 右键单击提交记录展示复制提交ID和复制提交消息的菜单。

查看 Git 输出

用户始终可以查看当前执行的Git命令的详细信息。

通过单击底部任务栏中“输出视图 ”或顶部菜单“查看>输出”打开输出视图，再选择Git面板，查看日志信息。如下图所示：

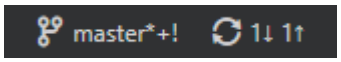
图 7-6 查看 Git 输出



状态栏显示

CodeArts IDE状态栏可以查看当前分支名，当前分支中传入和传出的提交数量等信息，并且允许用户运行“**同步更改**”命令，将远程更改拉取到本地仓库，后将本地更改的代码提交推送到远程分支。

图 7-7 状态栏分支名



用户还可以通过状态栏创建新分支并在分支之间切换。单击分支名，弹出弹窗展示当前项目的分支信息。如下图所示：

图 7-8 选择要签出的分支

7.2 配置 Git 编辑器

当您使用命令行操作CodeArts IDE，参见使用命令行运行文件，可以通过传递--wait参数使启动命令等待，直到您关闭新的CodeArts实例。这在将CodeArts IDE配置为Git的外部编辑器时非常有用，这样Git会等待您关闭启动的CodeArts实例。

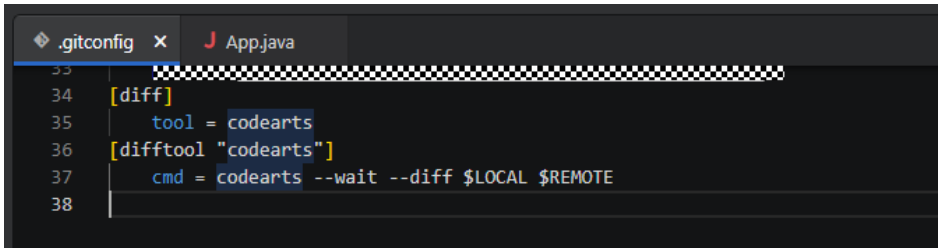
1. 确保您可以从命令行运行codearts --help命令，并且能够看到帮助信息。如果您没有看到帮助信息，请确保在安装过程中选择了**添加到PATH（重启后生效）**。
2. 从命令行运行git config --global core.editor "codearts --wait"命令。

现在您可以运行git config --global -e命令，并将CodeArts作为编辑器来配置Git。

CodeArts 作为 Git 差异工具

CodeArts作为Git的差异工具将以下内容添加到您的Git配置中，以将CodeArts作为差异工具使用：如下图所示：

图 7-9 .gitconfig 文件配置



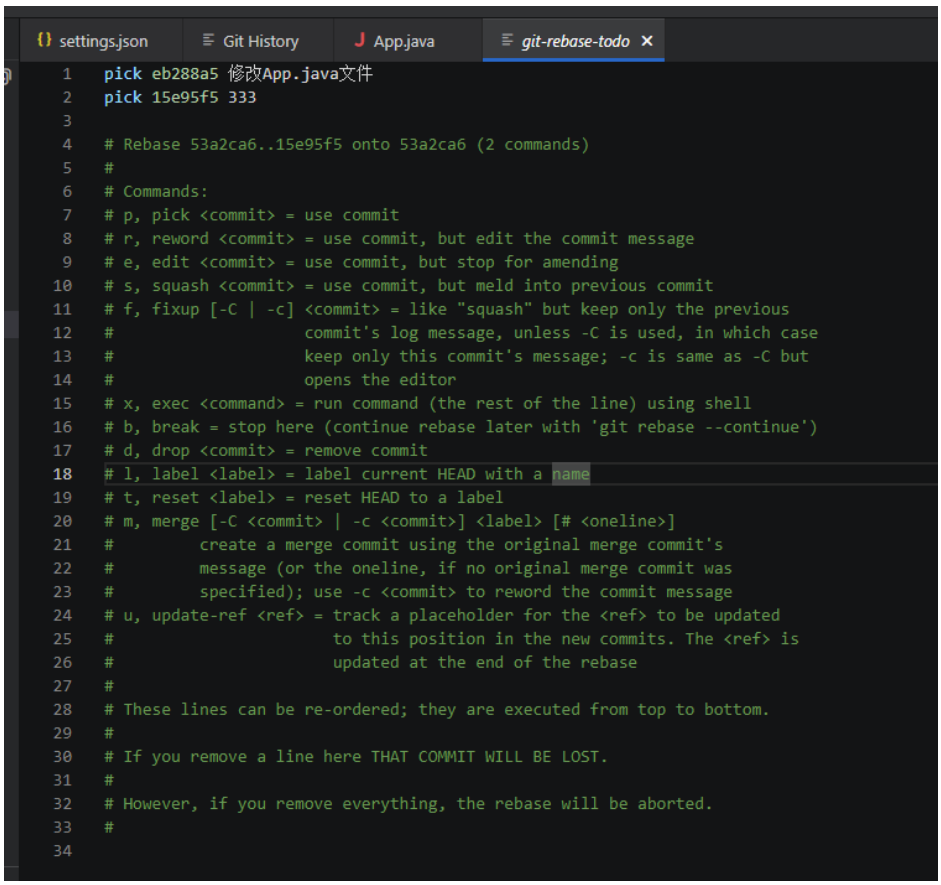

```
[diff]
   tool = default-difftool
[difftool "default-difftool"]
   cmd = codearts --wait --diff $LOCAL $REMOTE
```

这利用了您可以传递给CodeArts的--diff选项，以便比较两个文件的差异。

以下是一些可以使用CodeArts作为编辑器的示例：

git rebase HEAD~2 -i：使用CodeArts进行交互式变基。如下图所示：

图 7-10 git-rebase-todo 编辑器

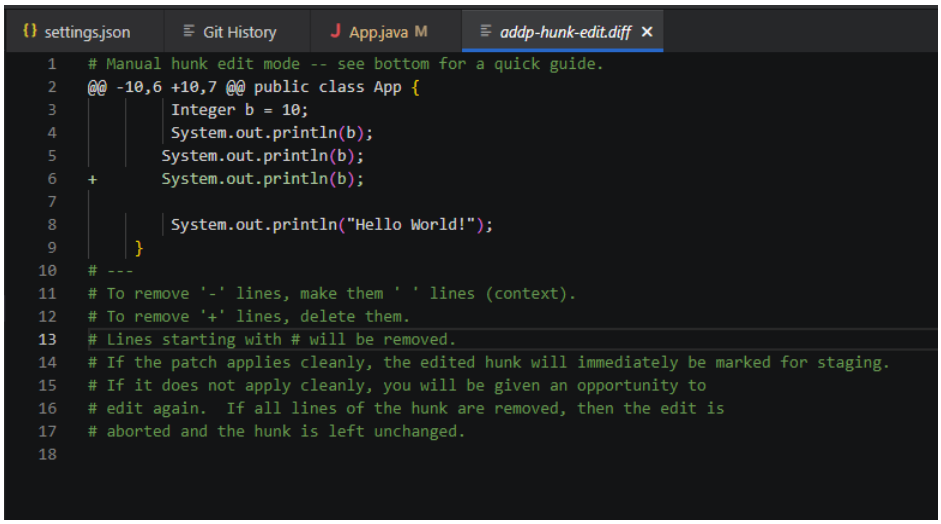


git commit：将CodeArts用作提交消息的编辑器。如下图所示：

图 7-11 commit_editMsg 编辑器

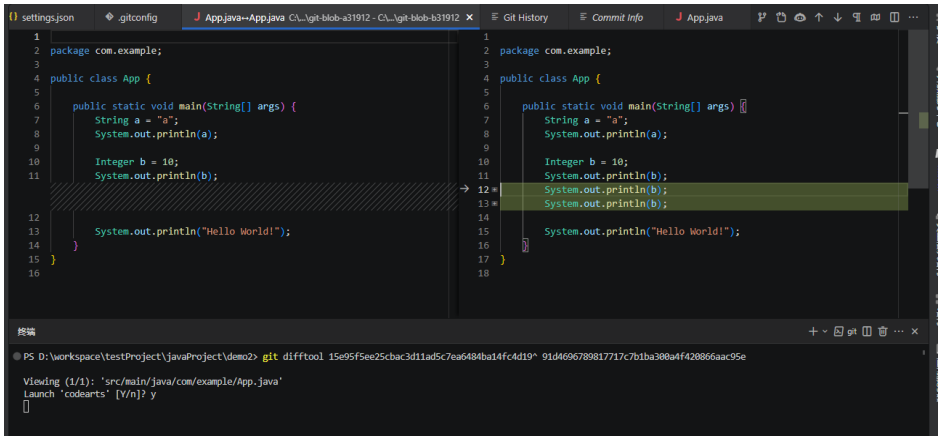
git add -p：接着输入e进行交互式添加。如下图所示：

图 7-12 addp-hunk-edit.diff 编辑器



git difftool <commit>^ <commit>：将CodeArts作为差异编辑器来查看更改。如下图所示：

图 7-13 diff 编辑器



7.3 管理仓库

初始化仓库

当用户打开一个本地文件夹时，可以通过在其中初始化一个Git仓库来启用Git源代码控制。

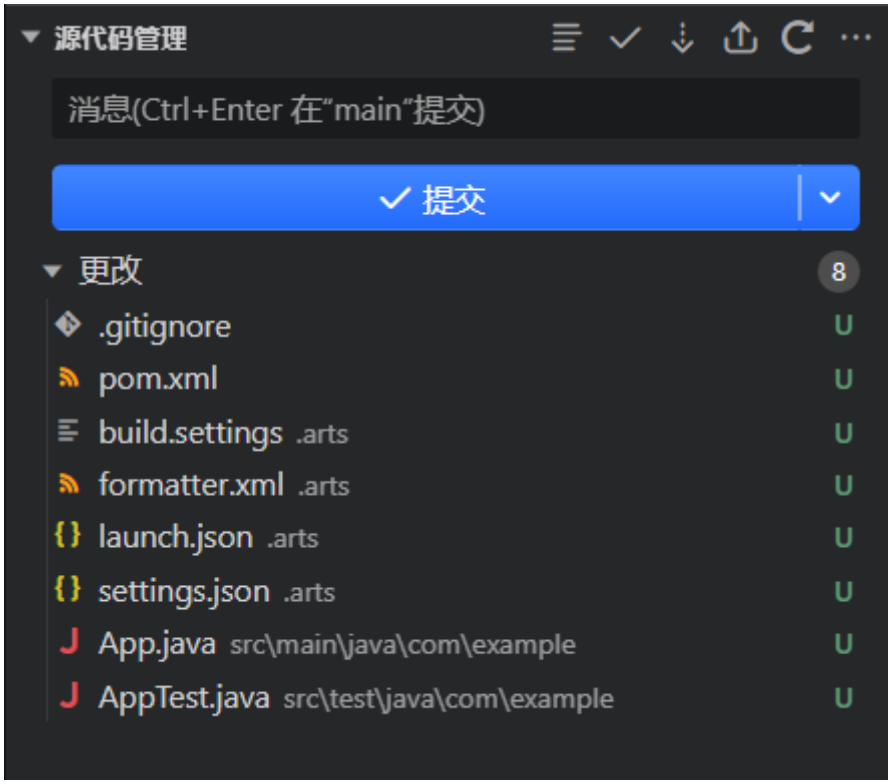
1. 打开“源代码管理”视图（按**“Ctrl+Shift+G”**或**“Alt+9”**(IDEA快捷键方案)），单击**“初始化仓库”**。如下图所示：

   图 7-14 源代码管理初始化仓库

2. CodeArts IDE将创建必要的Git仓库元数据文件，并将工作区文件显示为未跟踪的更改，准备进行暂存。如下图所示：

图 7-15 源代码管理文件变更



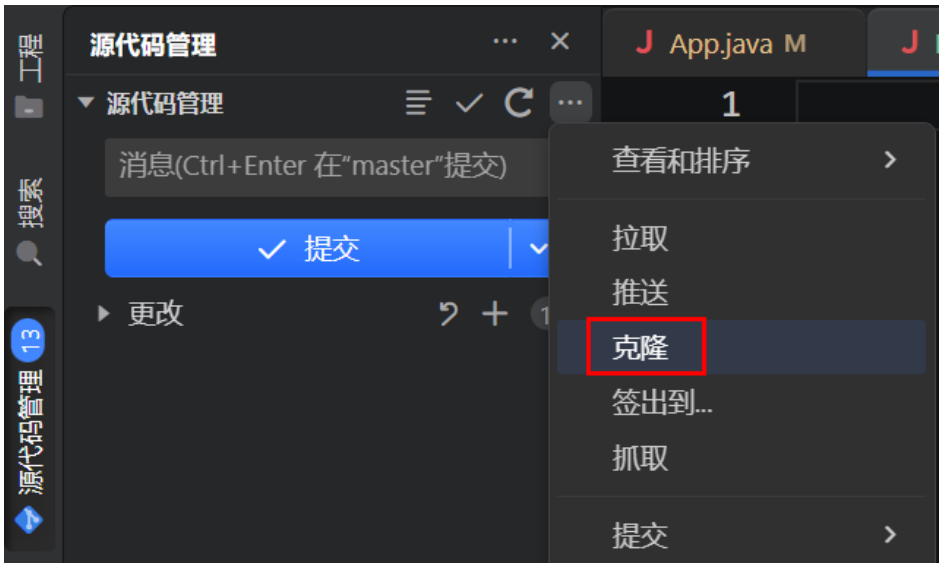
克隆现有仓库

- 如果尚未打开任何文件夹，则“源代码管理”视图可让用户打开本地文件夹或克隆仓库。如下图所示：

图 7-16 源代码管理克隆仓库

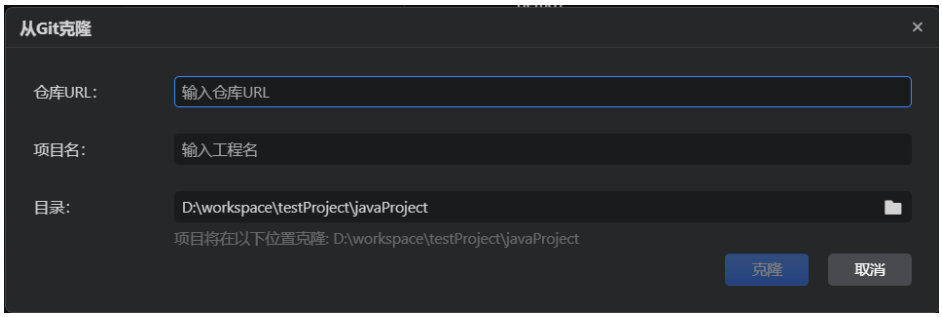
- 如果已经打开某个文件夹，请按以下步骤克隆仓库。
  a. 在“源代码管理”视图中，展开**“源代码管理”**部分。
  b. 单击**“更多操作”**按钮（ ）并选择**“克隆”**。如下图所示：

图 7-17 源代码管理克隆菜单



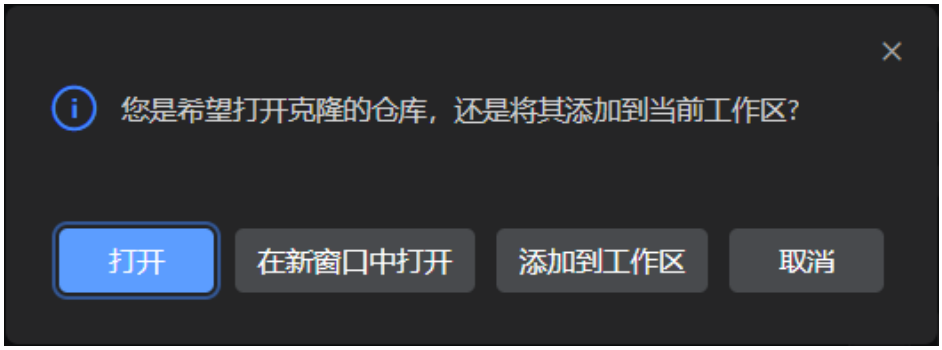
c. 在打开的弹出窗口中，提供远程仓库的URL，项目名和存储目录，单击**“克隆”**。如下图所示：

图 7-18 克隆弹窗



d. 仓库克隆完成后，CodeArts IDE会提示用户打开它。如下图所示：

图 7-19 项目打开方式



管理远程仓库

在创建本地Git仓库之后，可以在“源代码管理”视图添加多个远程仓库，以便能够在Git项目上进行协作。

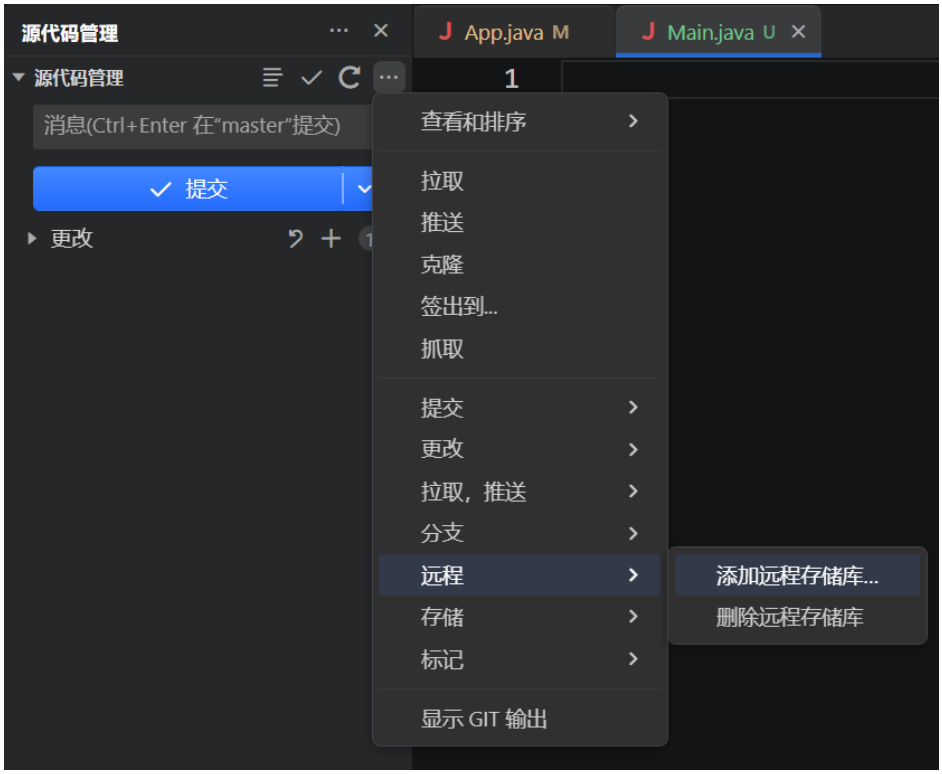
说明

提示：您应该设置一个**凭据助手**，以避免每次CodeArts IDE与Git远程仓库通信时都被要求输入凭据。否则，考虑通过git.autofetch设置禁用自动获取以减少身份验证提示的次数。

添加远程存储库

1. 添加远程存储库在用户选择的SCM托管平台上创建一个空仓库。
2. 在“源代码管理”视图中，展开**“源代码管理”**部分。
3. 单击要添加新的远程仓库的仓库旁边的**“更多操作”**按钮（···），然后选择“远程 > 添加远程存储库...”。如下图所示：

图 7-20 远程存储库菜单



4. 单击添加远程存储库菜单，弹出弹窗，输入远程仓库地址和远程存储库名称。如下图所示：

图 7-21 仓库 URL

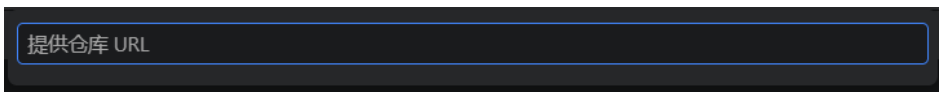
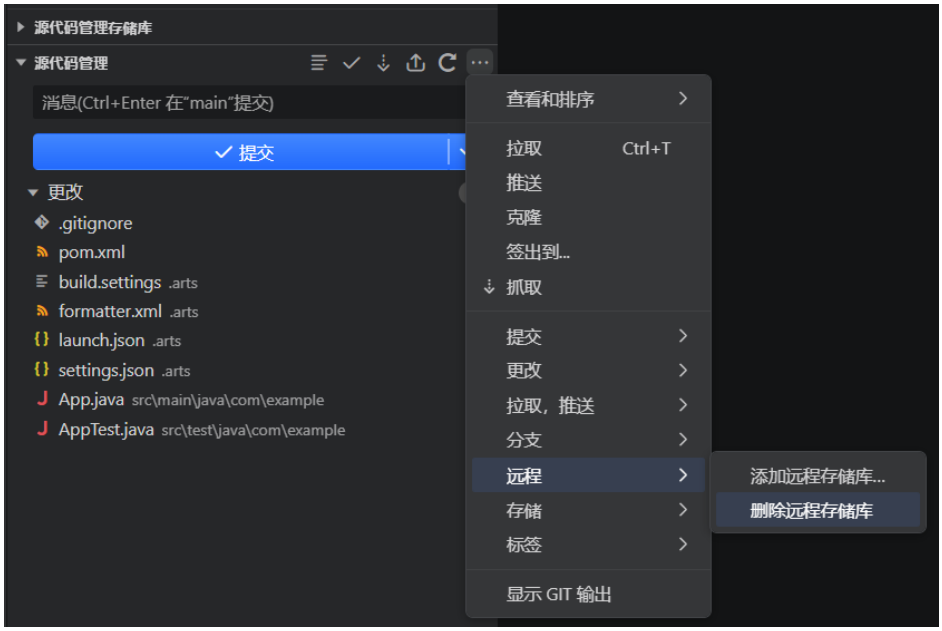


图 7-22 仓库名称

删除远程存储库

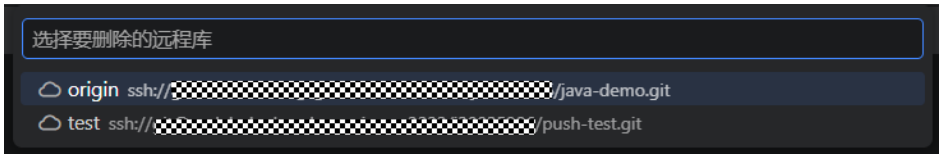
1. 在“源代码管理”视图中，展开**“源代码管理”**部分。
2. 单击要移除远程仓库的仓库旁边的**“更多操作”**按钮（···），然后选择“远程 > 删除远程存储库”。如下图所示：

图 7-23 删除远程存储库菜单



3. 在打开的弹出窗口中，选择要移除的远程仓库并按Enter键。如下图所示：

图 7-24 选择要删除的远程仓库



7.4 管理 Git 分支

CodeArts IDE可以方便地处理Git分支，让您创建和切换分支，并将一个分支的更改合并到另一个分支中。

也可以通过使用合并（Merge）和变基（Rebase）命令在Git分支之间应用代码更改。

创建分支

1. 在“源代码管理”视图中，展开**“源代码管理”**部分。
2. 单击要在其中创建新分支的存储库旁边的**“更多操作”**按钮（···），指向**“分支”**，然后通过以下任意一种方式创建分支。
   - 要从当前正在工作的分支创建新分支，请选择**“创建分支”**，并在打开的弹出窗口中提供新分支的名称，然后按“Enter”键。如下图所示：

图 7-25 分支名称

- 要从存储库中的其他分支创建新分支，请选择**“从现有来源创建新的分支”**，并在打开的弹出窗口中选择源分支。如下图所示：

图 7-26 分支列表

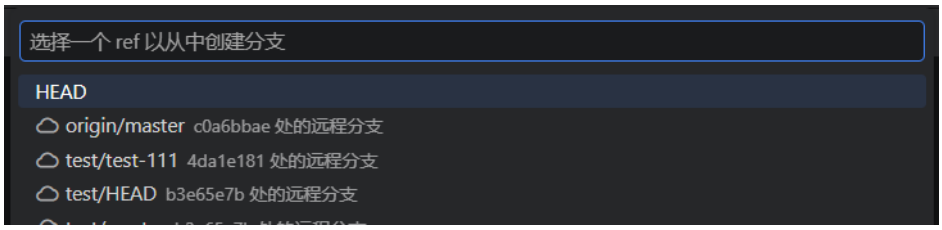


图 7-27 分支名称

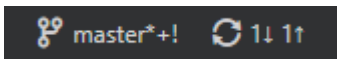
然后在打开的弹出窗口中提供新分支的名称，然后按“Enter”键。

CodeArts IDE会自动创建一个新分支并切换到该分支。如果Git存储库已设置远程，可以在**“源代码管理”**部分或CodeArts IDE状态栏中单击**“发布”**按钮（ ）将当前分支发布到远程。

切换分支

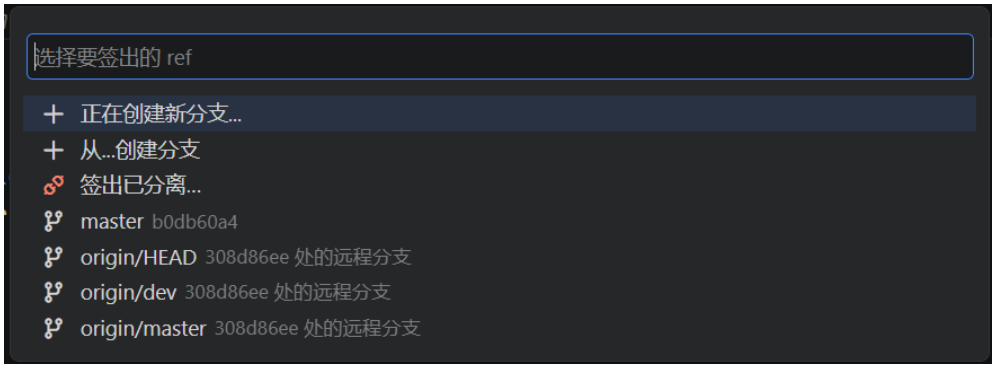
1. 执行以下操作之一：
   - 在“源代码管理”视图中，展开**“源代码管理”**部分，单击要切换到另一个分支的存储库旁边的**“更多操作”**按钮（ ），然后选择**“签出到”**。
   - 在CodeArts IDE状态栏中单击分支名称。如下图所示：

图 7-28 状态栏分支名



2. 在打开的弹出窗口中，选择要切换到的分支，然后按Enter键。如果选择了一个尚不存在本地分支的远程分支，则CodeArts IDE将自动创建本地分支。如下图所示：

图 7-29 分支列表



说明

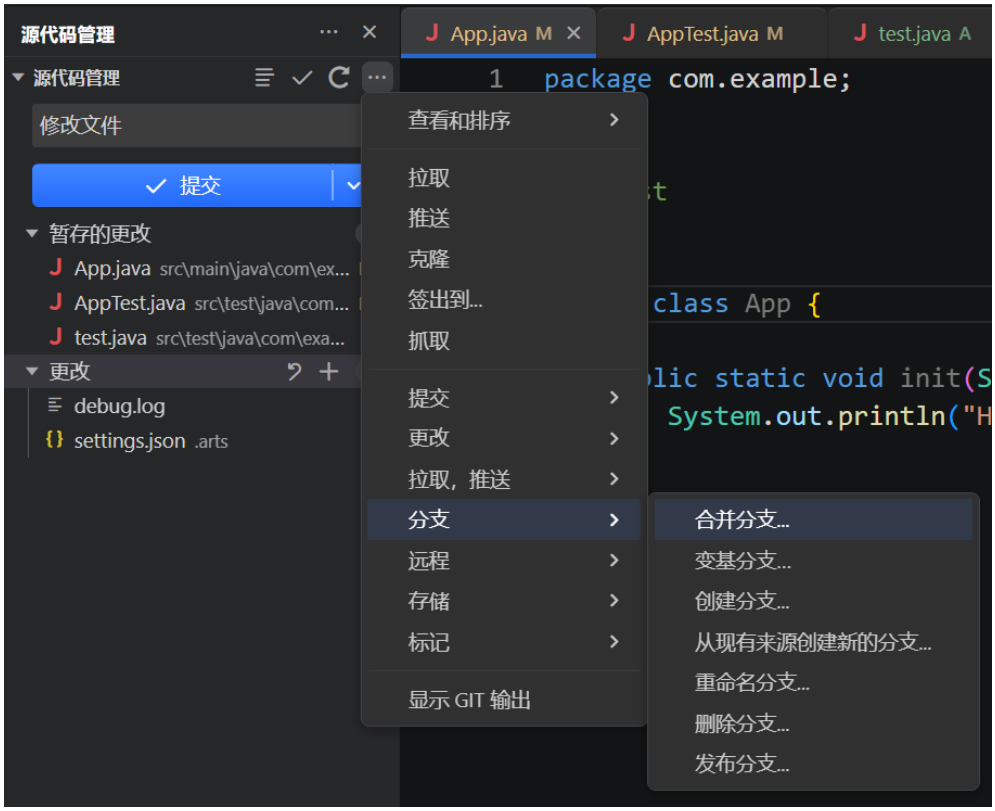
用户还可以通过**“签出到”**弹出窗口创建新的本地分支。

合并分支

Merge命令允许用户将源分支的更改集成到目标分支的HEAD中。Git会创建一个新的提交（称为“合并提交”），将源分支和目标分支从两个分支分叉点开始的更改合并在一起。

1. 切换到目标分支，即用户想要将更改合并到的分支。有关详细信息，请参阅**切换分支**。
2. 在“源代码管理”视图中，展开**“源代码管理”**部分。
3. 单击要将一个分支的更改合并到另一个分支的存储库旁边的**“更多操作”**按钮（···），指向**“分支”**，然后选择**“合并分支”**。如下图所示：

图 7-30 合并分支菜单



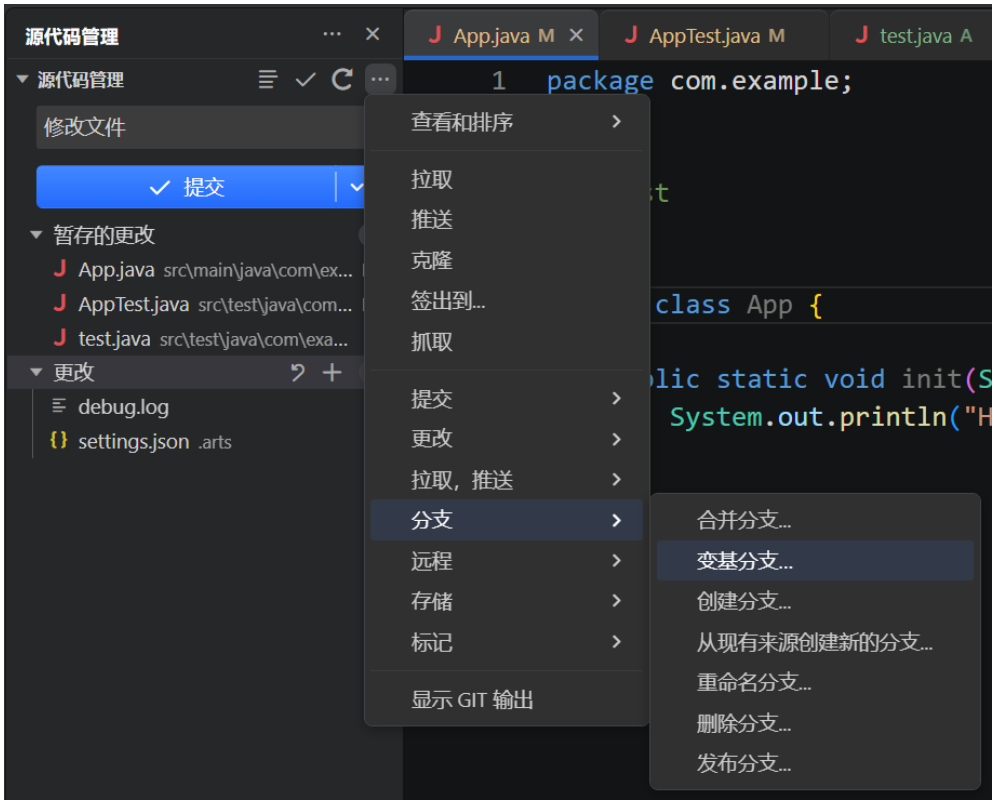
4. 在打开的弹出窗口中，选择要从中合并更改的分支。如果发生合并冲突，请按照**解决合并冲突**中描述的方法解决它。

变基分支

Rebase命令允许用户将源分支的提交应用到目标分支的HEAD提交之上。

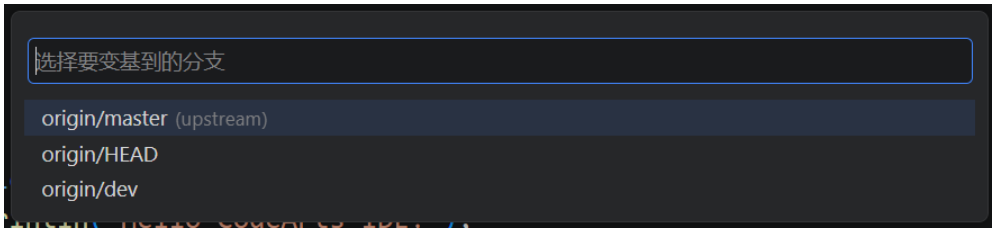
1. 切换到源分支，即用户想要将其提交应用到另一个分支上的分支。有关详细信息，请参阅**切换分支**。
2. 在“源代码管理”视图中，展开**“源代码管理”**部分。
3. 单击要将一个分支的更改合并到另一个分支中的存储库旁边的**“更多操作”**按钮（ ），指向**“分支”**，然后选择**“变基分支”**。如下图所示：

图 7-31 变基分支



4. 在打开的弹出窗口中，选择用户要将更改应用到的目标分支。如下图所示：

图 7-32 分支列表



处理分支冲突

在某些情况下，用户在本地对文件所做的更改可能与其他人对同一文件所做的更改冲突。另一个常见的原因是将一个分支**合并到另一个分支**。CodeArts IDE会识别这种合并冲突并显示相应的通知。如下图所示：

图 7-33 合并冲突提示

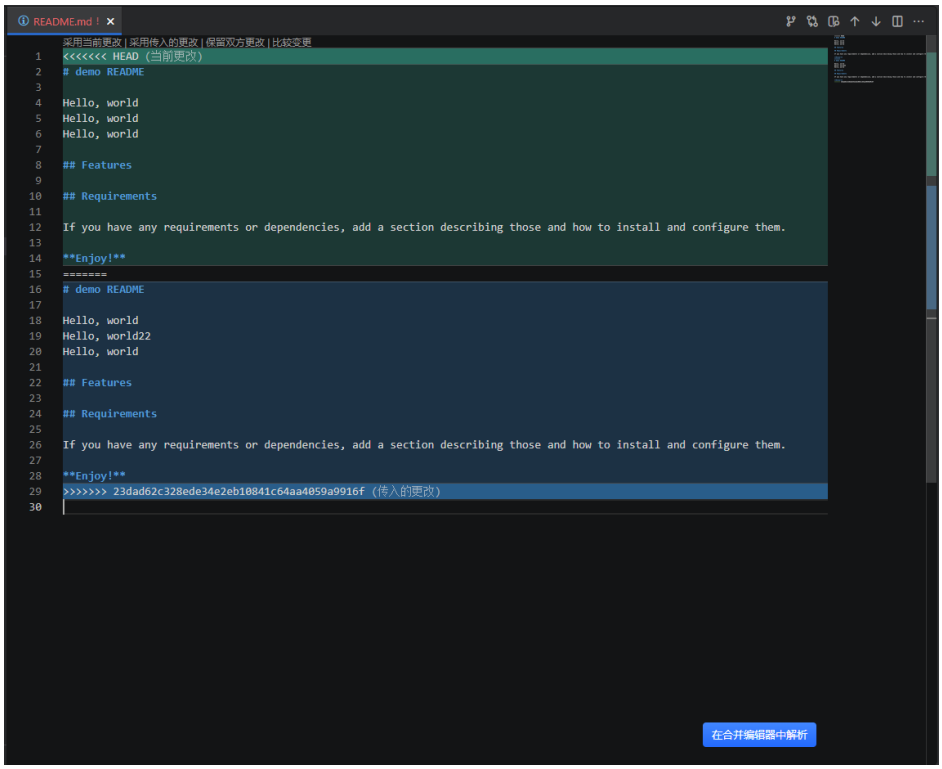
处理合并冲突的步骤如下：

1. 在源代码控制视图的合并更改部分，找到包含冲突更改的文件。如下图所示：

图 7-34 冲突文件

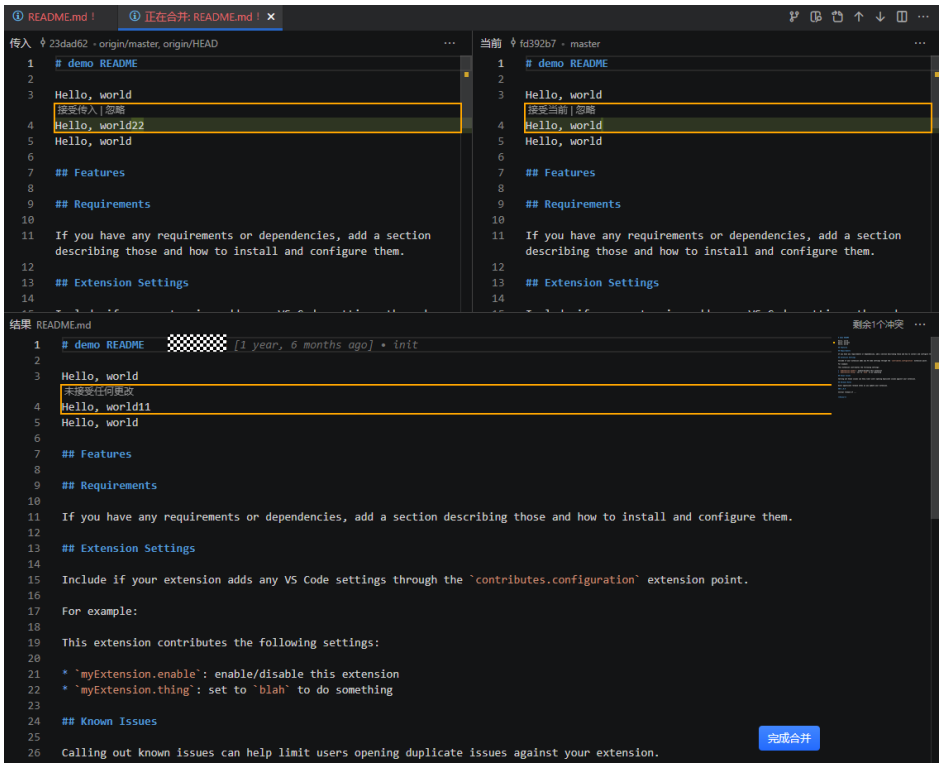
2. 单击文件，会打开代码编辑器，进入专门的冲突视图，通过**差异查看器**详细查看更改。如下图所示：

图 7-35 代码编辑器



可以直接在当前编辑器中解决冲突，或者单击“合并编辑器中解析按钮”，会打开“合并编辑器”，将会在合并编辑器中解决代码冲突。如下图所示：

图 7-36 合并编辑器

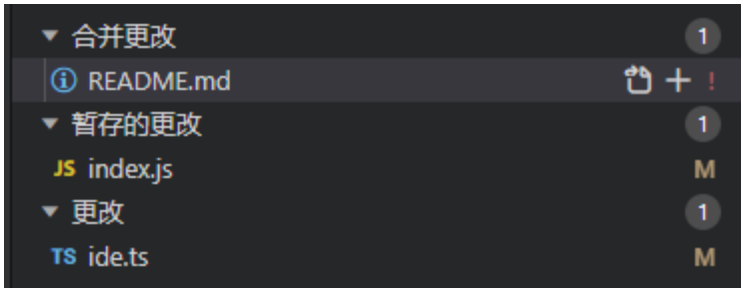


一旦冲突解决完毕，用户可以将冲突的文件暂存并**提交更改**。

7.5 提交更改

当用户的项目与源代码管理（SCM）系统关联时，CodeArts IDE会跟踪项目文件的所有更改。左侧导航栏中的**“源代码管理”**按钮显示当前存储库中的文件更改数量。**“源代码管理”**视图详细列出了当前存储库的更改，分为**“合并更改”**、**“暂存的更改”**和**“更改”**三个部分。如下图所示：

图 7-37 更改分组



- 合并更改：显示在合并分支过程中产生的修改，可能包含解决冲突后的文件。
- 暂存的更改：显示已经用git add命令添加到暂存区的文件。
- 更改：显示工作目录中所有未暂存的修改，包括新增、修改或删除的文件。

说明

对于未暂存的更改，右侧的编辑器仍允许用户编辑文件。

获取更改

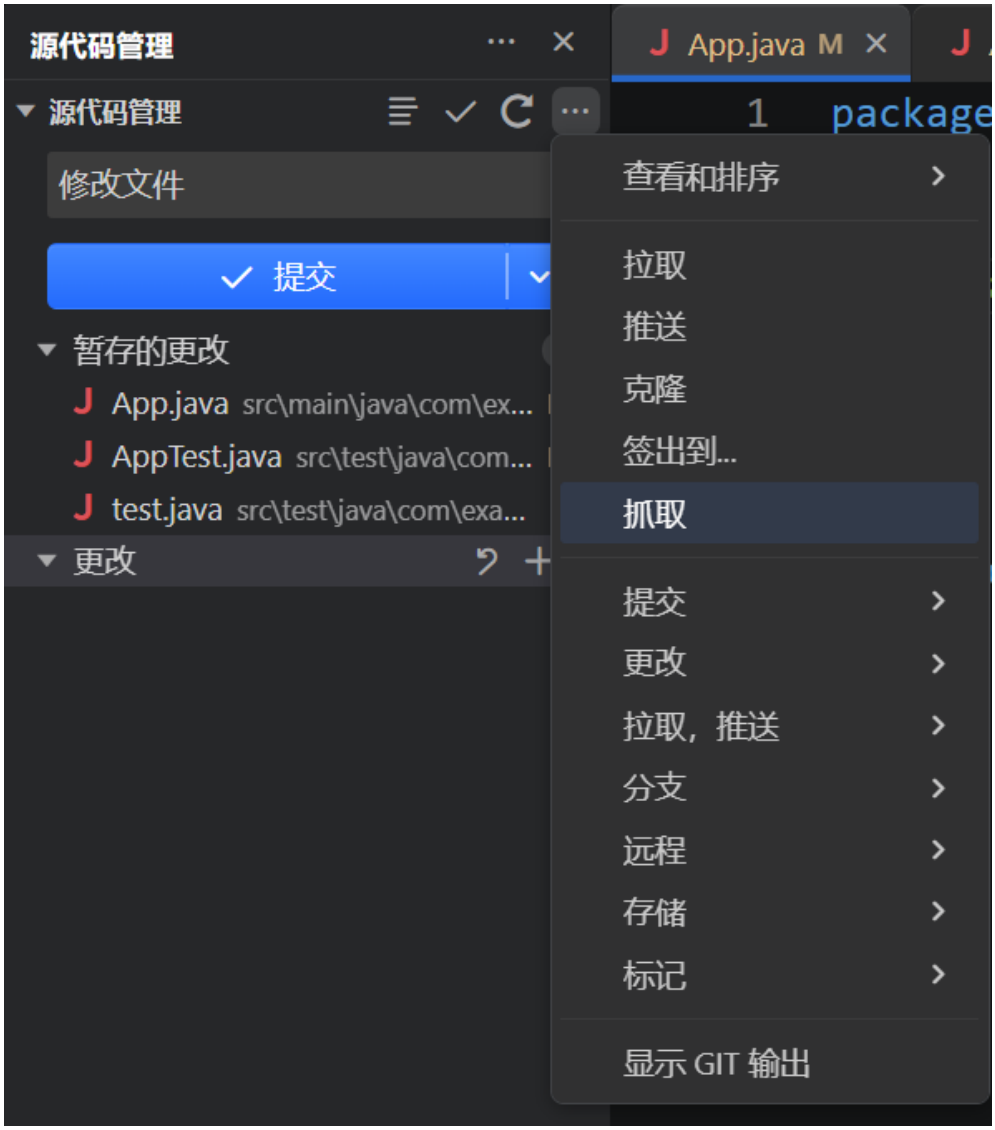
当存储库连接到远程，并且检出分支与远程的分支有上游链接时，CodeArts IDE允许获取更改。

从远程获取更改可以显示本地存储库相对于远程存储库的超前或落后情况。这些更改不会自动合并到本地工作树中。CodeArts IDE支持自动定期获取，此功能默认禁用，但用户可以通过git.autofetch设置启用它。手动获取远程更改的方法如下：

步骤1 在“源代码管理”视图中，展开“源代码管理存储库”部分。

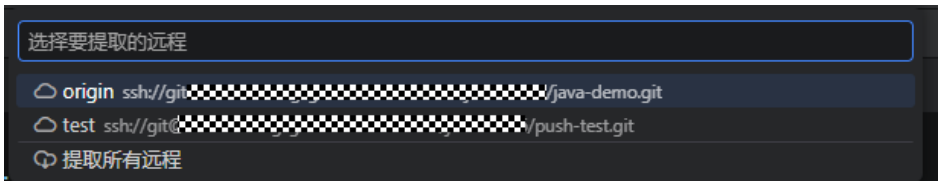
步骤2 单击要获取更改的存储库旁边的**“更多操作”**按钮（···），然后选择**“抓取”**。如下图所示：

图 7-38 抓取菜单



步骤3 如果配置了多个远程，可以通过选择“拉取，推送 > 从所有远程存储库中拉取”来从所有远程获取更改。如下图所示：

图 7-39 远程仓库列表



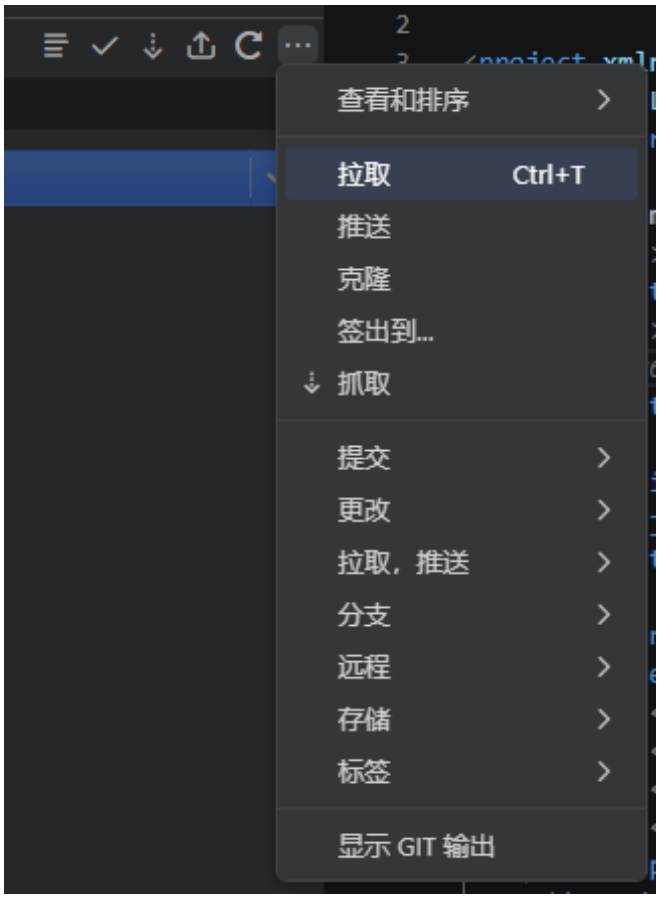
----结束

拉取更改

运行拉取命令时，CodeArts IDE会从远程存储库获取更改并将其集成到本地工作树中。

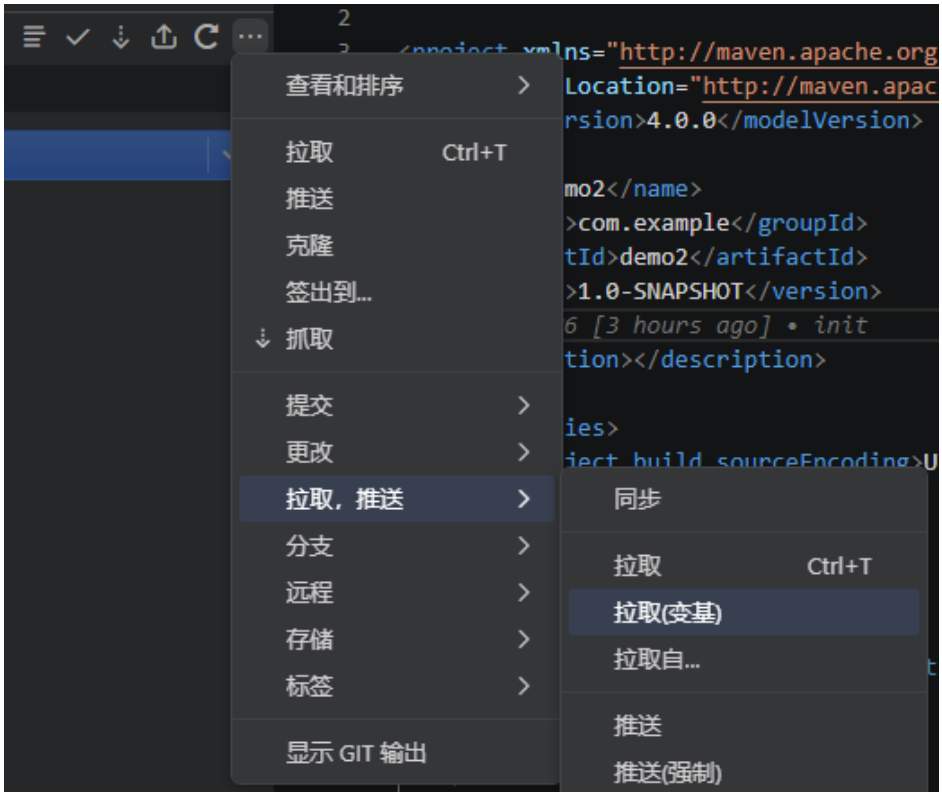
1. 在“源代码管理”视图中，展开**“源代码管理”**部分。
2. 单击要将更改拉取到的存储库旁边的**“更多操作”**按钮（ ），然后执行以下操作之一：
   - 要将更改从远程跟踪分支拉取到当前本地分支，请选择**“拉取”**，或按**“Ctrl+T”**（IDEA快捷键方案）。如下图所示：

     图 7-40 拉取菜单

   - 要拉取更改并同时将本地未推送的更改rebase到已拉取的更改上，请选择“拉取，推送 > 拉取（变基）”。如下图所示：

图 7-41 拉取变基菜单



- 要从不同配置的远程存储库拉取更改，请选择“拉取，推送 > 拉取自”。然后在打开的弹出窗口中选择所需的远程存储库。如下图所示：

图 7-42 拉取自菜单

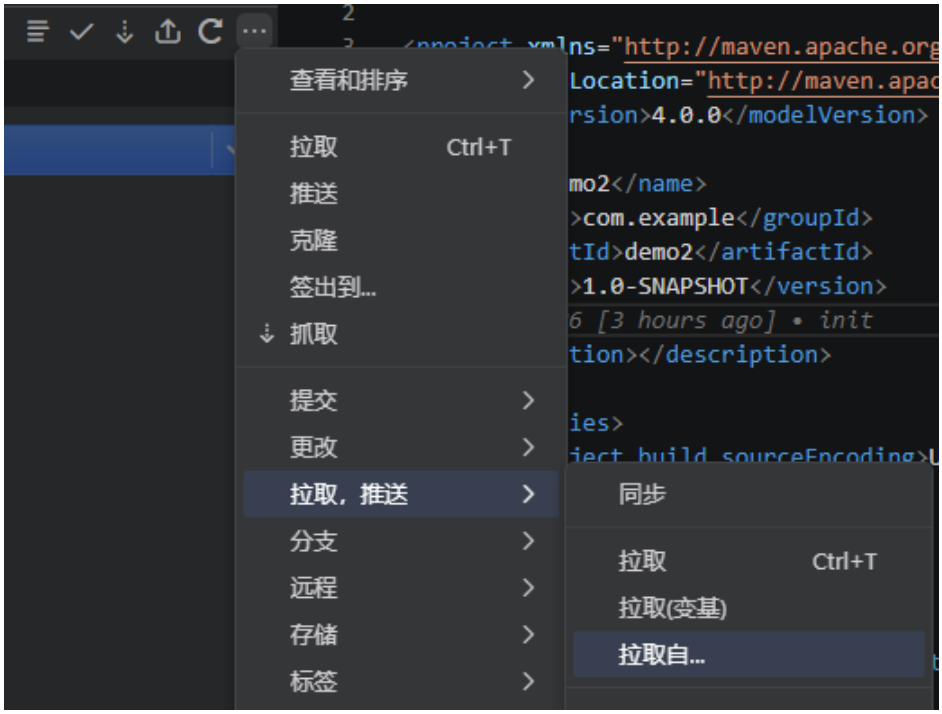
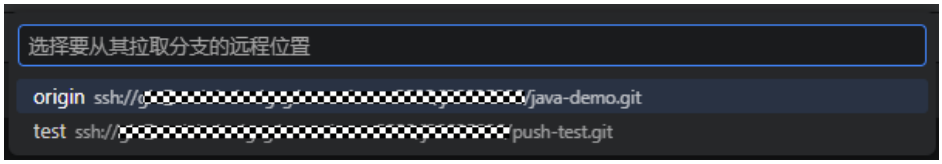


图 7-43 远程存储库列表



提交代码

通过提交代码操作将本地修改的代码提交到远程仓库，以便团队成员使用。

说明

配置Git的用户名和电子邮件，可以通过以下任一操作：

- 全局配置
  git config --global user.name "用户名"
  git config --global user.email "电子邮件"
- 项目配置
  git config user.name "用户名"
  git config user.email "电子邮件"
- 不配置
  每次在拉取或提交代码，都会让用户输入用户名、邮箱。

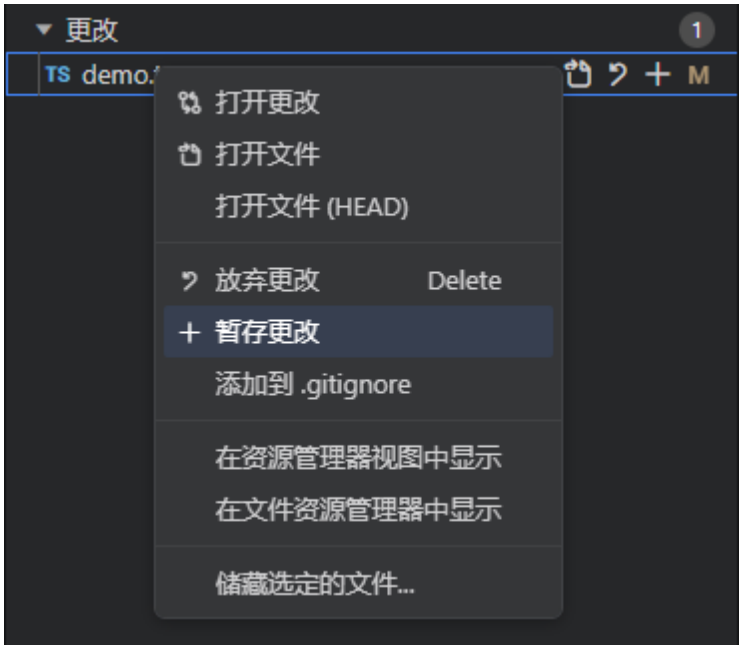
1. 通过将更改添加到暂存区来提交代码要执行如下操作，请在源代码控制视图的更改部分中执行以下操作之一。
   - 要暂存整个文件，执行以下任一操作。
     单击**“暂存更改”**按钮（ ）。如下图所示：

     图 7-44 暂存更改操作

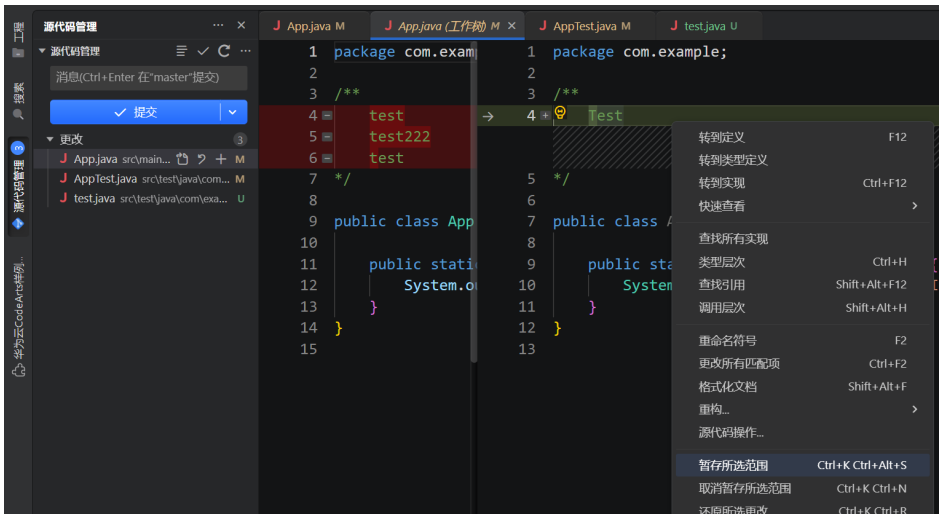
     右键单击文件并选择**“暂存更改”**。如下图所示：

图 7-45 暂存更改菜单



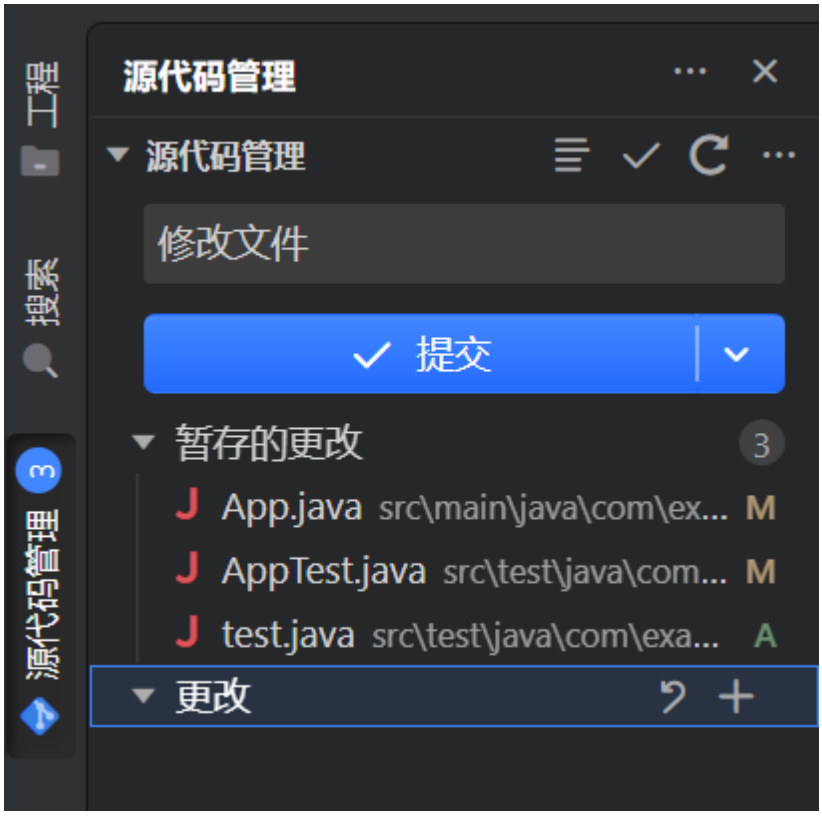
- 要暂存文件的一部分，请单击文件以打开差异视图，该视图提供更改的概述。选择要暂存的更改，右键单击并选择**“暂存所选范围”**，或先按下“Ctrl+K”再按下“Ctrl+Alt+S”快捷键触发。如下图所示：

图 7-46 暂存更改范围菜单



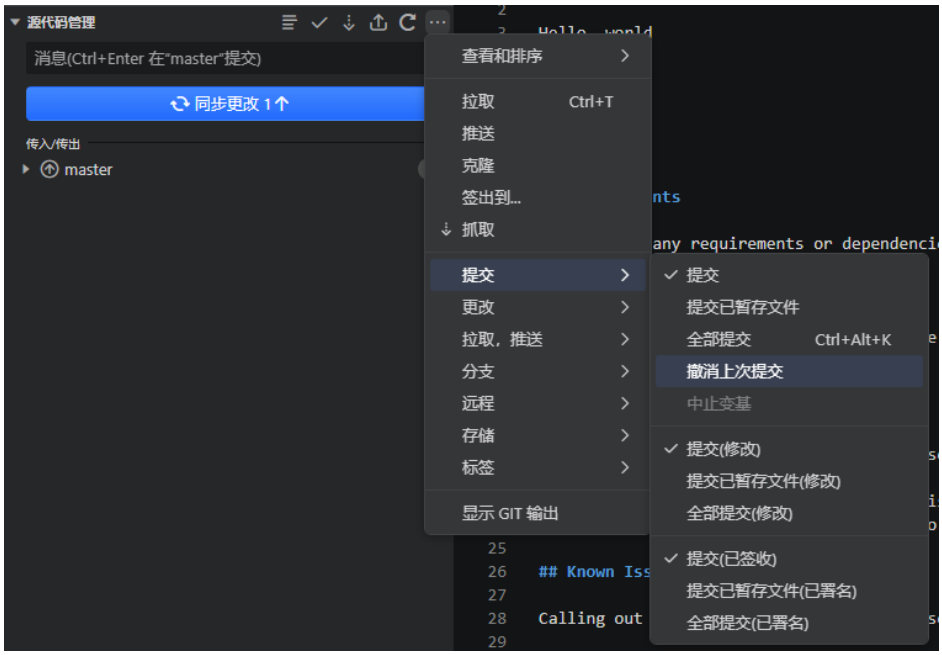
2. 在“源代码管理”视图中，在字段中输入提交消息，然后单击**“提交”**按钮或按**“Ctrl+Enter”**。如下图所示：

图 7-47 输入提交信息“修改文件”



3. 如果要撤销提交，请单击**“更多操作”**按钮（···）并选择**“撤销上次提交”**。更改将重新添加到**“暂存的更改”**部分。如下图所示：

图 7-48 撤销上次提交菜单

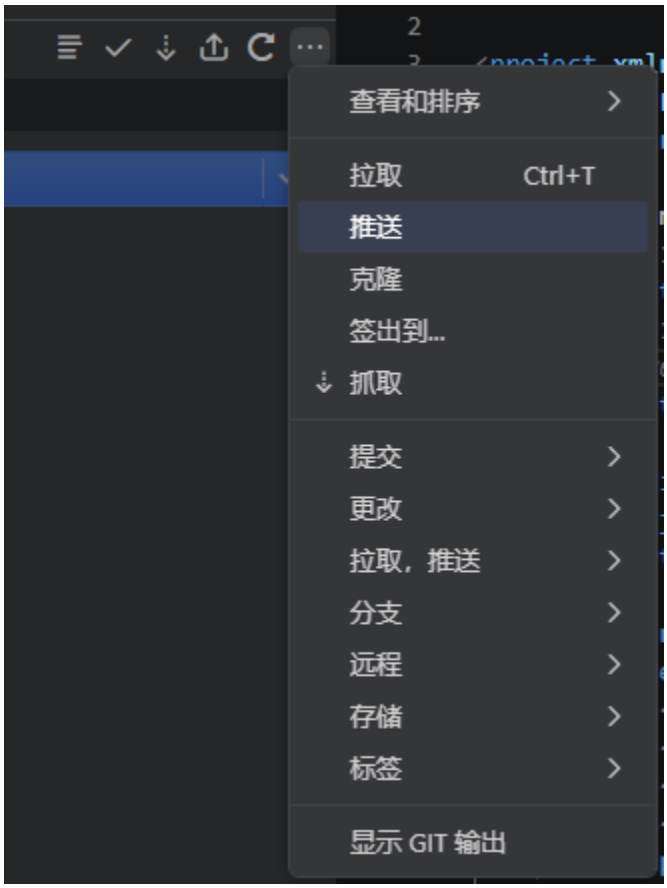


推送更改

在本地提交更改后，用户需要运行推送命令将其上传到远程存储库。

1. 在“**源代码管理**”视图中，展开“**源代码管理**”部分。
2. 单击要推送更改的存储库旁边的“**更多操作**”按钮（···），然后执行以下操作之一：
   - 要将更改从当前本地分支推送到远程跟踪分支，请选择“**推送**”。如下图所示：

     图 7-49 推送菜单

   - 要将更改推送到不同配置的远程存储库，请选择“拉取，推送 > 推送到”。然后在打开的弹出窗口中选择所需的远程存储库。如下图所示：

图 7-50 推送到菜单

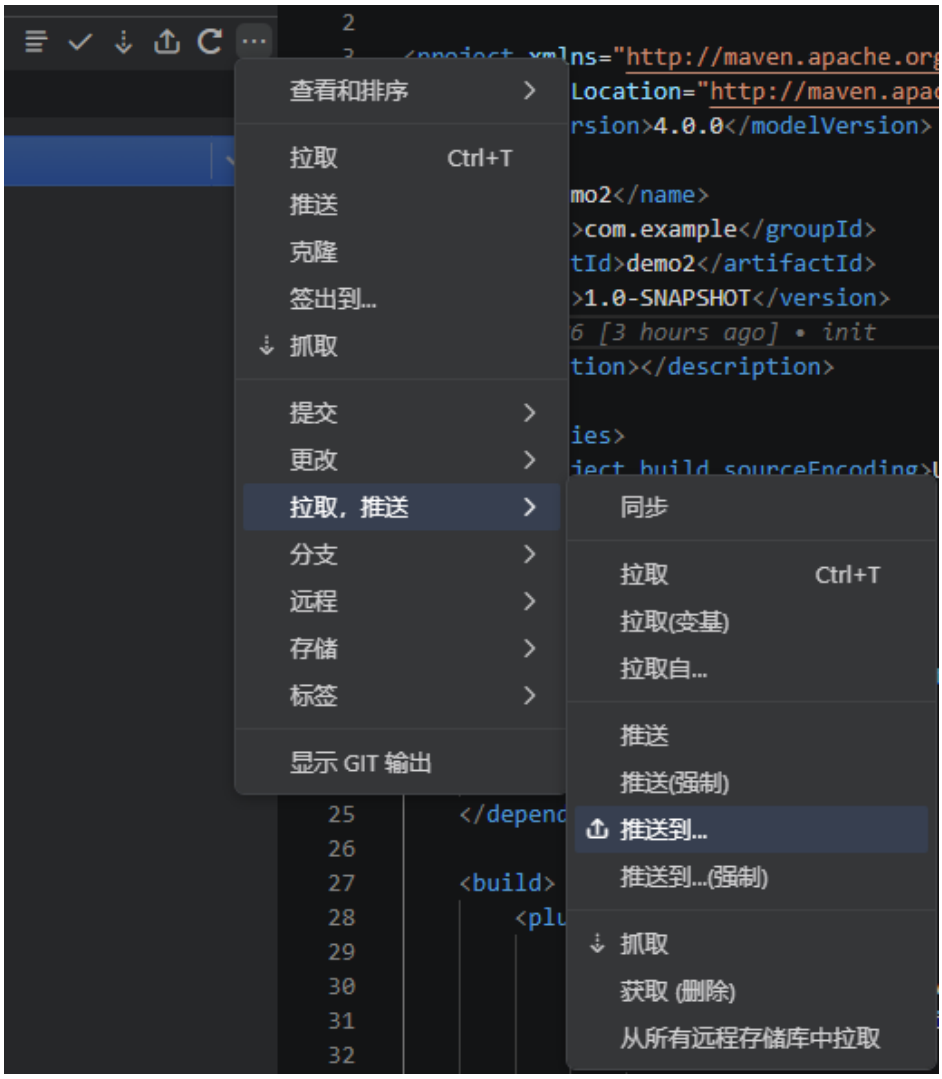
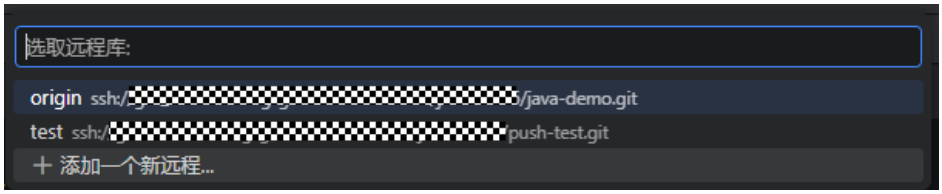


图 7-51 远程存储库列表



7.6 储藏更改

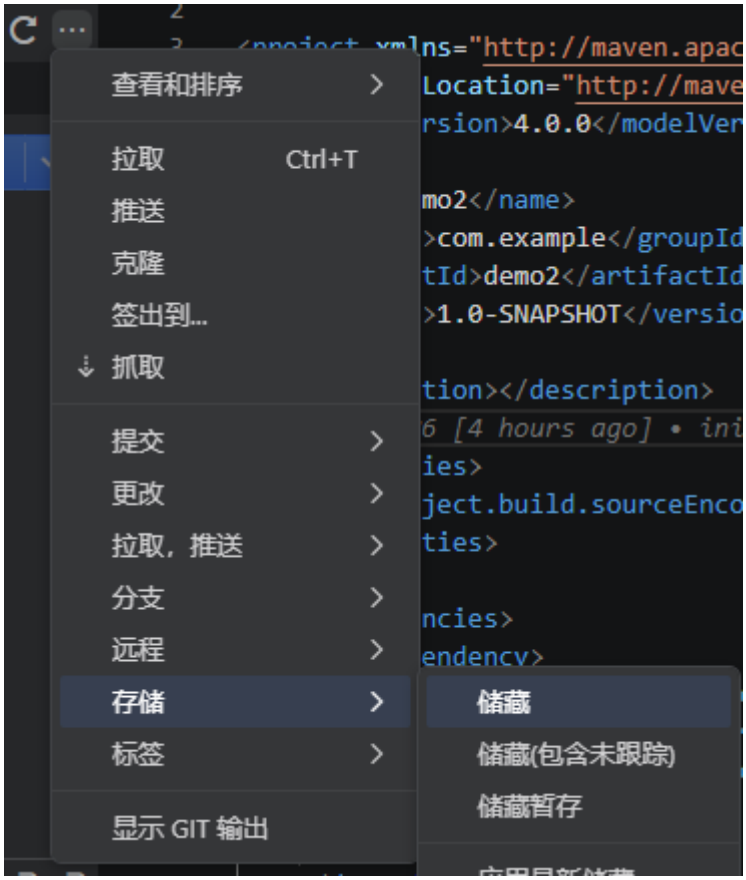
添加存储

使用**stash**存储，可以将当前更改移动到临时位置，而无需将其提交，从而将工作副本恢复到“干净”（即HEAD提交）状态。

要存储更改，请执行以下任一操作：

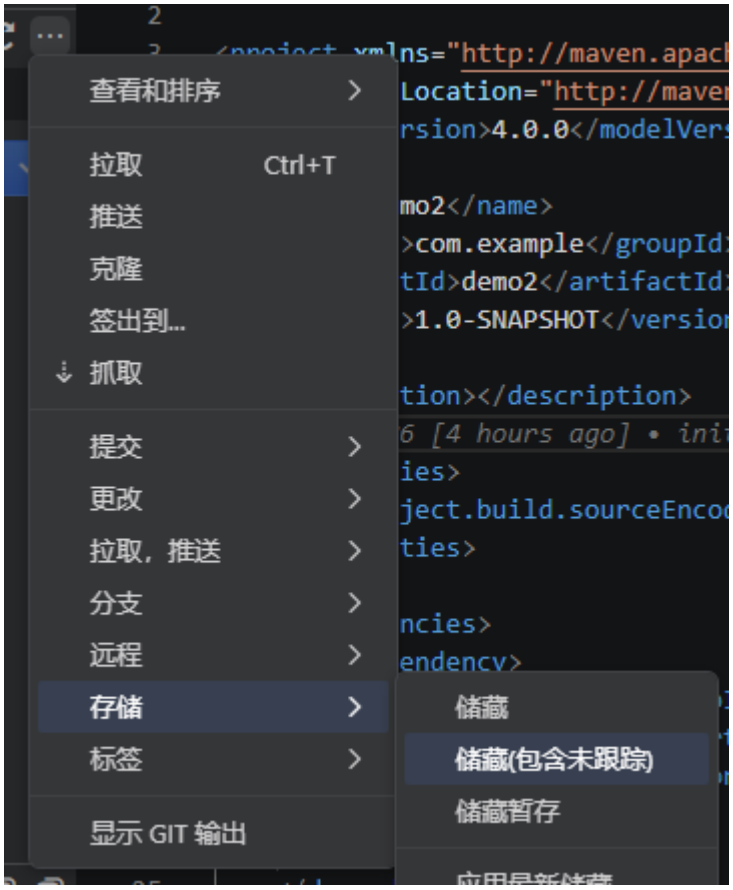
- 在“**源代码管理**”部分操作
  a. 在“源代码管理”视图中，展开“**源代码管理**”部分。
  b. 单击要存储更改的存储库旁边的“**更多操作**”按钮（···），指向“**存储**”，然后执行以下操作之一：
     - 要存储已暂存的更改，请选择“**储藏**”。如下图所示：

       图 7-52 储藏菜单

     - 要存储所有更改，包括未暂存和未版本化的文件，请选择“**储藏（包含未跟踪）**”。如下图所示：

图 7-53 储藏包含未跟踪菜单



- 在“储藏”部分操作
  a. 在“源代码管理”视图中，展开“储藏”部分。在“储藏”视图中，展开存储列表
  b. 单击“储藏...”。如下图所示：

图 7-54 储藏菜单

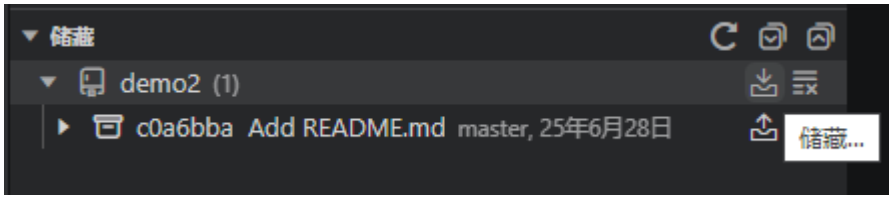
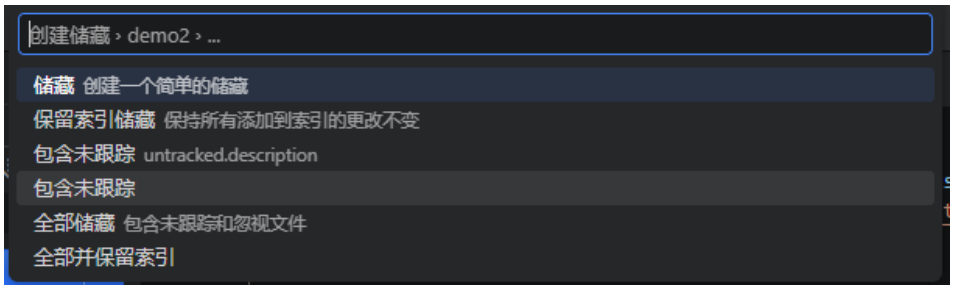


图 7-55 创建储藏



应用存储

移动更改后，用户可以随时重新应用它们到用户的工作副本中。要应用存储，请执行以下任一操作：

- 在“**源代码管理**”部分操作
  a. 在“源代码管理”视图中，展开“**源代码管理**”部分。
  b. 单击要重新应用更改的存储库旁边的“**更多操作**”按钮（···），指向“**存储**”，然后执行以下操作之一：
     - 要应用最近的存储，并且当前存储不从存储堆栈中删除，请选择“应用最新储藏”。如下图所示：

图 7-56 应用储藏

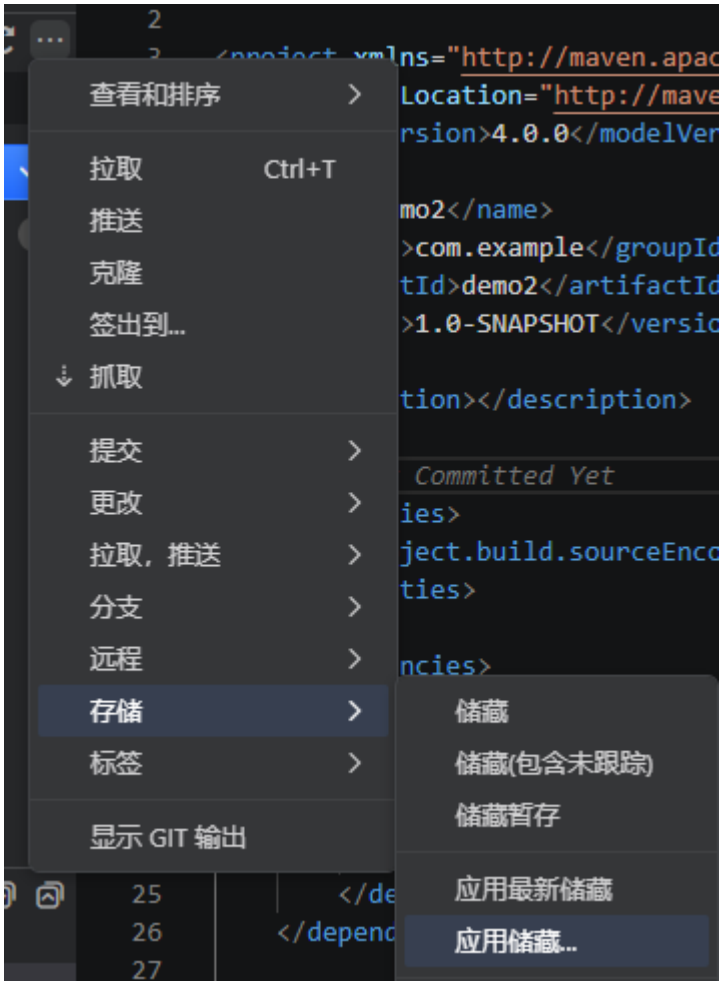
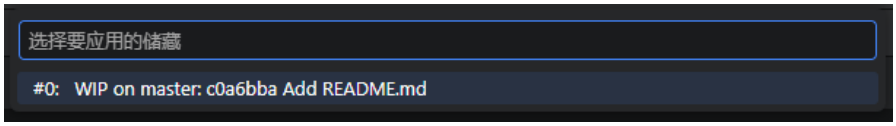


图 7-57 储藏列表



     - 要应用任意存储，并且删除存储堆栈中的记录，请选择“**弹出储藏**”。如下图所示：

图 7-58 弹出储藏

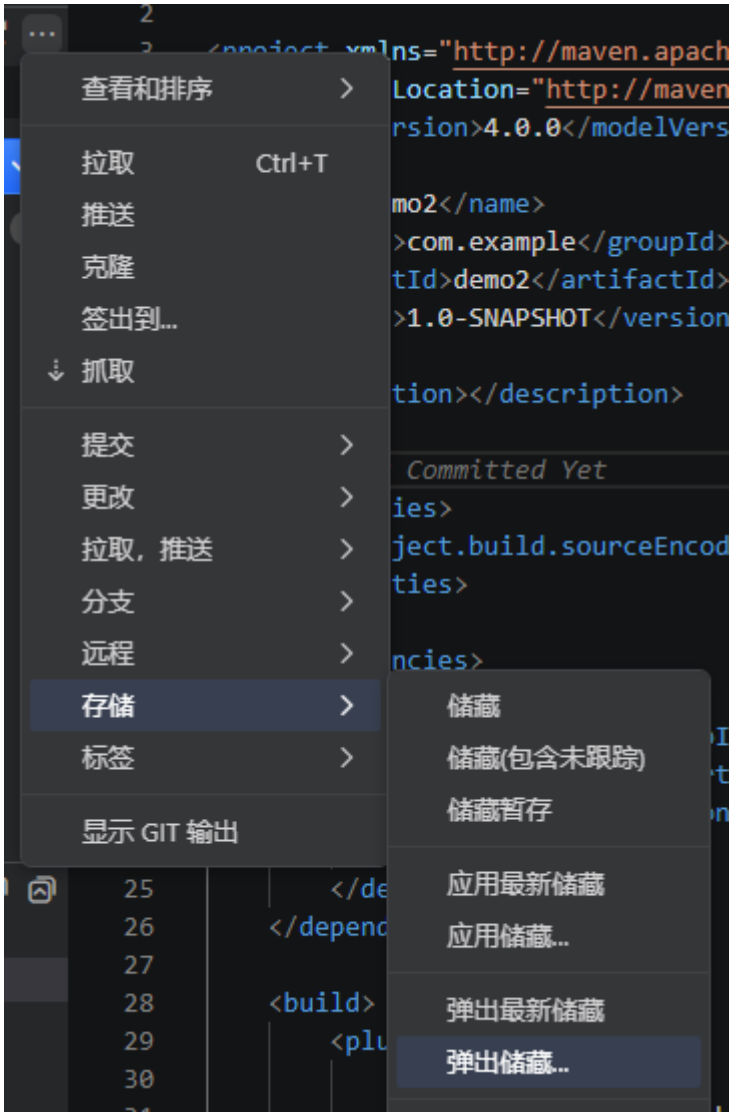
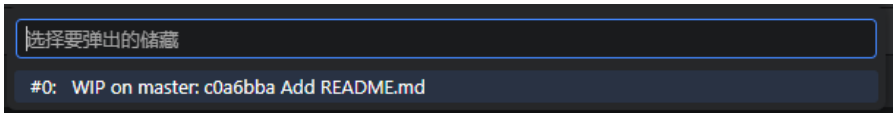
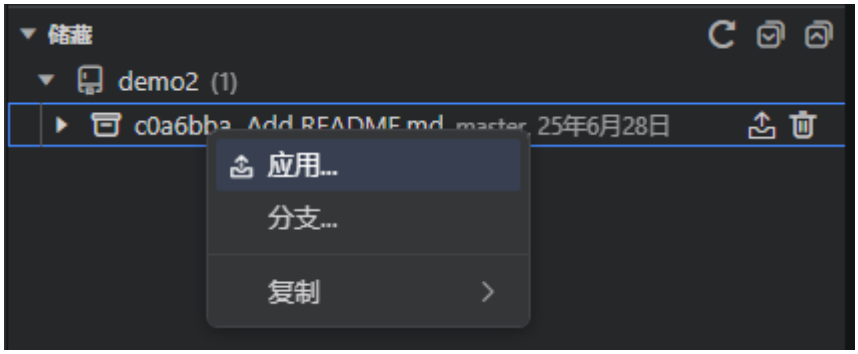


图 7-59 储藏列表



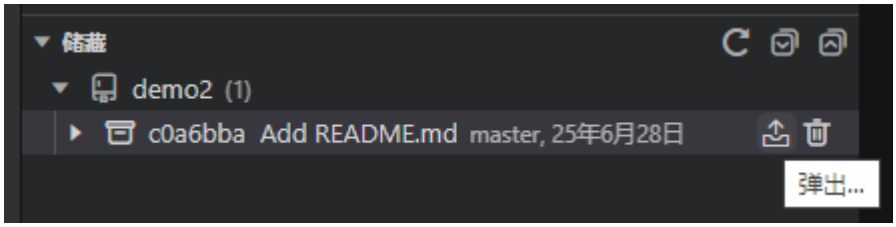
- 在“储藏”部分操作
  a. 在“源代码管理”视图中，展开“储藏”部分。在“储藏”视图中，展开存储列表
  b. 选择需要应用的储藏，然后执行以下操作之一：
     - 要应用最近的存储，并且当前存储不从存储堆栈中删除，右键展示菜单，选择“应用”。如下图所示：

图 7-60 应用储藏



- 要应用任意存储，并且删除存储堆栈中的记录，请选择“弹出...”。如下图所示：

图 7-61 弹出储藏



删除存储

用户可以清理存储堆栈以删除不再需要的存储。要删除存储，请执行以下任一操作：

- 在**“源代码管理”**部分操作
  a. 在“源代码管理”视图中，展开**“源代码管理”**部分。
  b. 单击要重新应用更改的存储库旁边的**“更多操作”**按钮（···），指向**“存储”**，然后执行以下操作之一：
     - 要删除任意存储，请选择“删除储藏”，并在打开的弹出窗口中选择所需的存储。如下图所示：

图 7-62 删除储藏

- 要删除所有存储，请选择“删除所有储藏”。如下图所示：

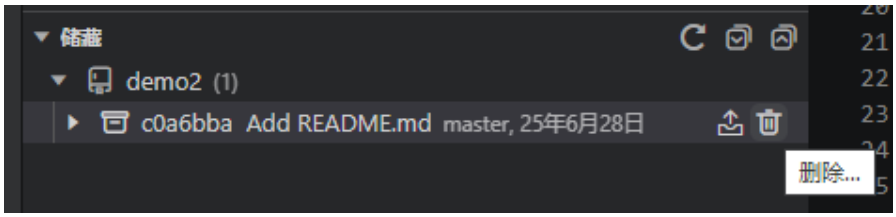
图 7-63 删除所有储藏

- 在“储藏”部分操作

1. 在“源代码管理”视图中，展开“储藏”部分。在“存储”视图中，展开存储列表。
2. 选择需要应用的储藏，然后执行以下操作之一：

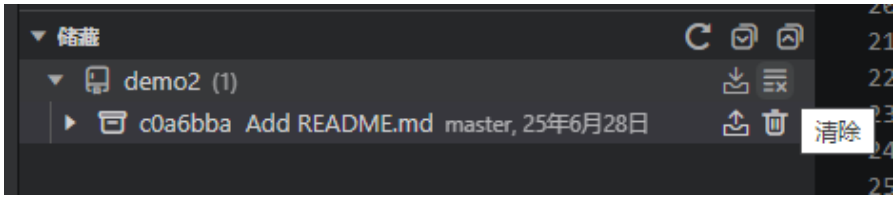
- 要删除任意存储，请选择“删除储藏”，并在打开的弹出窗口中选择所需的存储。如下图所示：

图 7-64 删除储藏



- 要删除所有存储，请选择“删除所有储藏”。如下图所示：

图 7-65 清除储藏



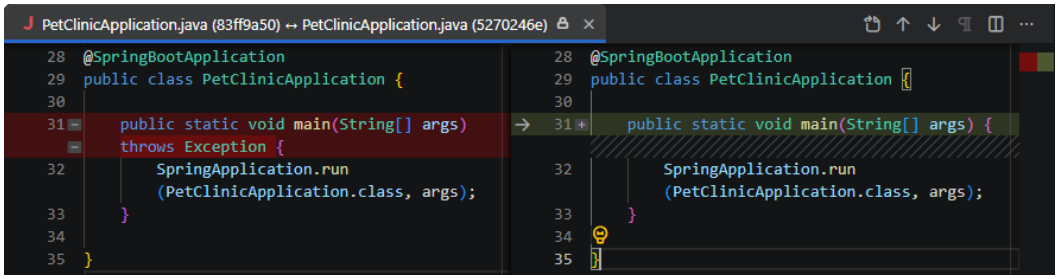
7.7 查看版本记录

CodeArts IDE支持同时处理多个源代码管理（SCM）提供商，其中Git支持已经内置。CodeArts IDE还维护了本地历史版本记录，即使您的项目没有关联的SCM提供商，也可以跟踪文件的更改。

差异查看器

CodeArts IDE提供了一个内置的“差异查看器”。如下图所示：

图 7-66 差异编辑器



说明

您可以通过首先在资源管理器或打开的编辑器列表中右键单击文件，然后单击**选择以进行比较**，再右键单击要与之比较的第二个文件，并选择**与已选项目进行比较**来比较任意两个文件。或者，按下“Ctrl+Shift+P”/“Ctrl Ctrl”，然后选择**文件：比较活动文件与...**，您将看到最近文件的列表。

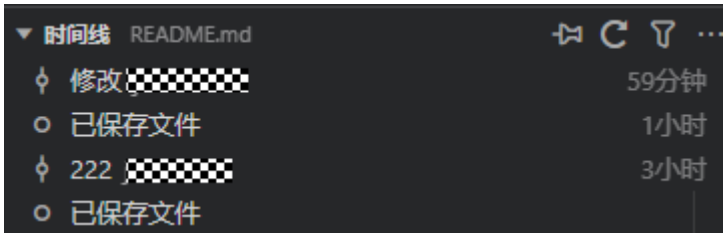
差异的默认视图是并排视图。通过单击右上角的更多操作（···）按钮，然后选择**内联视图**来切换到内联视图。如果您喜欢内联视图，可以将“diffEditor.renderSideBySide”设置为false。

本地历史记录

使用本地历史记录，CodeArts IDE可以记录文件上的修改历史。您可以查看和恢复文件在任何事件发生时的内容。

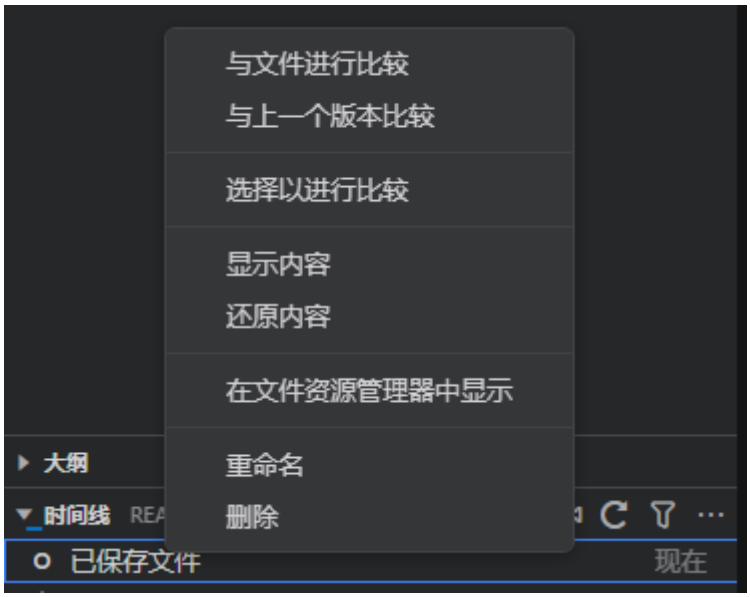
单击左侧导航栏中的**资源管理器**（ ）并展开**时间线**视图。每次保存文件时，都会向**时间线**事件列表中添加一个新记录。如下图所示：

图 7-67 时间线视图



在“时间线”视图中，右键单击一个“本地历史记录”，然后从上下文菜单中选择所需的操作。如下图所示：

图 7-68 本地历史记录



- **与文件进行比较 (Compare with File)**：将文件内容与文件的当前状态进行比较。
- **与上一个版本进行比较 (Compare with Previous)**：比较此次修改和上一次修改之间的文件内容变化。
- **选择以进行比较 (Select for Compare)**：选择要进行比较的修改。选择了一个修改后，右键单击要将文件内容与之比较的另一个修改，然后从上下文菜单中选择**“选择以进行比较”**。
- **显示内容（Show contents）**：在单独的只读编辑器选项卡中查看修改的文件内容。
- **还原内容（Restore contents）**：重新应用文件内容，从而覆盖所有后续更改。请注意，这将丢弃此文件的任何未保存更改。

提交历史记录

CodeArts IDE可以查看历史提交记录。以下操作都可以查看提交历史：

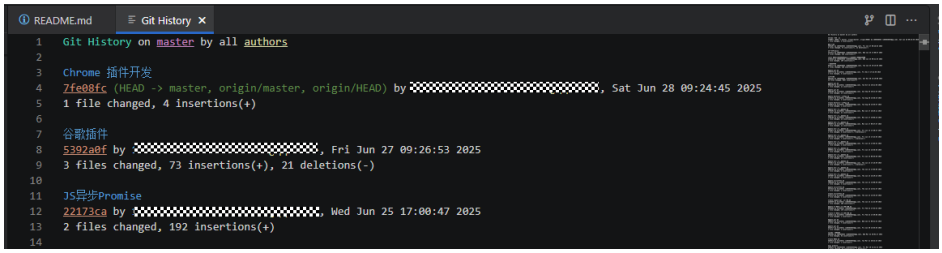
可以从项目，文件夹，文件和行几个维度来查看历史记录：

1. 查看项目历史记录
   a. 在编辑器工具栏单击“查看项目历史记录”按钮，打开“Git History编辑器”，查看项目的提交记录。如下图所示：

图 7-69 查看项目历史记录

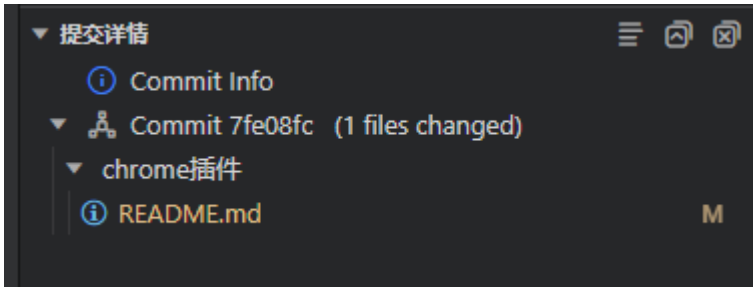
可以在“Git History编辑器”中修改分支和作者来查看当前项目不同分支或不同作者的提交记录。如下图所示：

图 7-70 Git History 编辑器



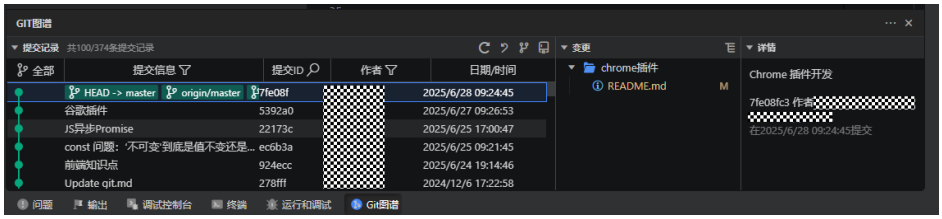
单击提交记录列表上的哈希值，可以在“源代码管理”中打开提交详情视图，查看当前提交的修改详情。如下图所示：

图 7-71 提交详情



   b. 单击底部“Git图谱”视图，展示提交记录视图、变更视图、详情视图，查看项目的提交记录。如下图所示：

图 7-72 Git 图谱



2. 查看文件夹历史记录

   打开资源管理器视图，选中目录，右键单击“查看文件夹历史记录”，打开“Git History编辑器”，查看文件夹历史记录。如下图所示：

图 7-73 查看文件夹历史记录

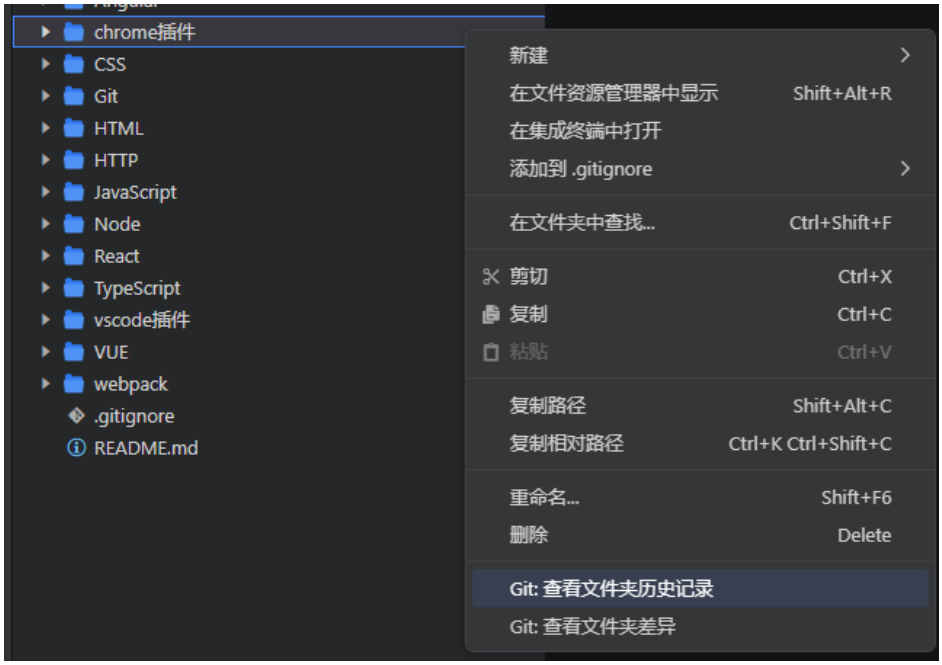
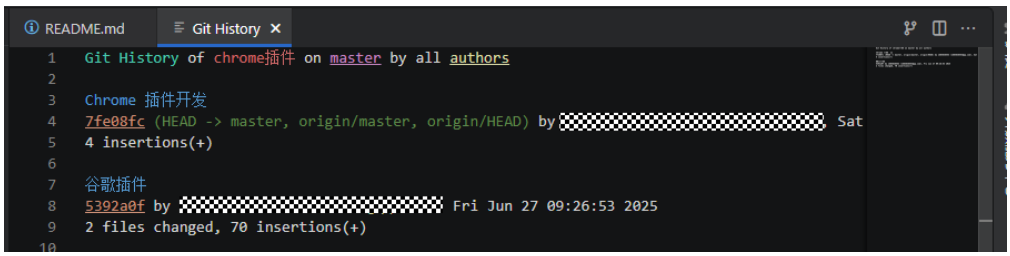


图 7-74 Git History



3. 查看文件历史记录

   打开资源管理器视图，选中文件，右键单击“查看文件历史记录”，打开“Git History编辑器”，查看文件历史记录。如下图所示：

图 7-75 查看文件历史记录

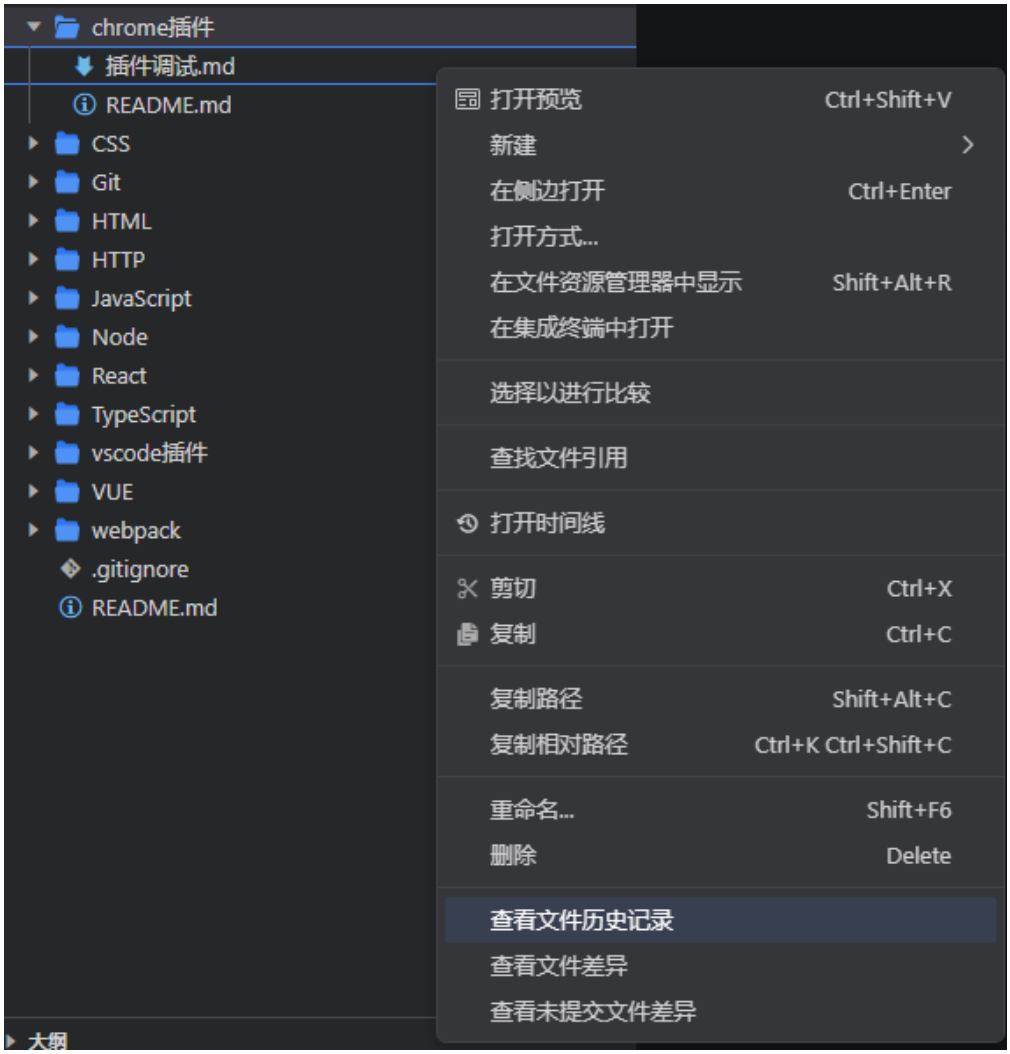
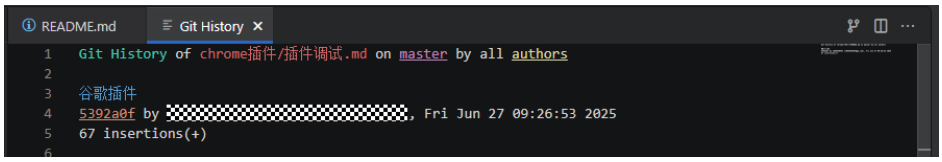
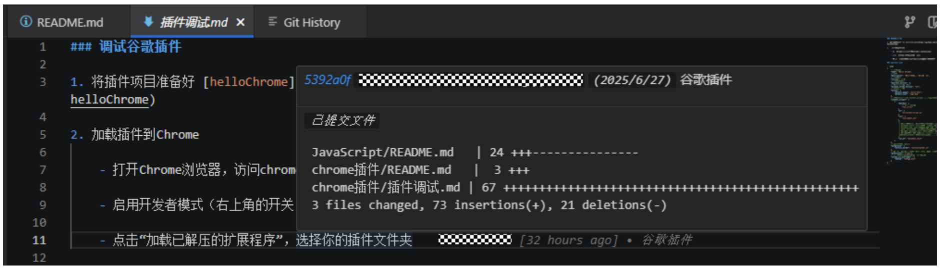


图 7-76 Git History



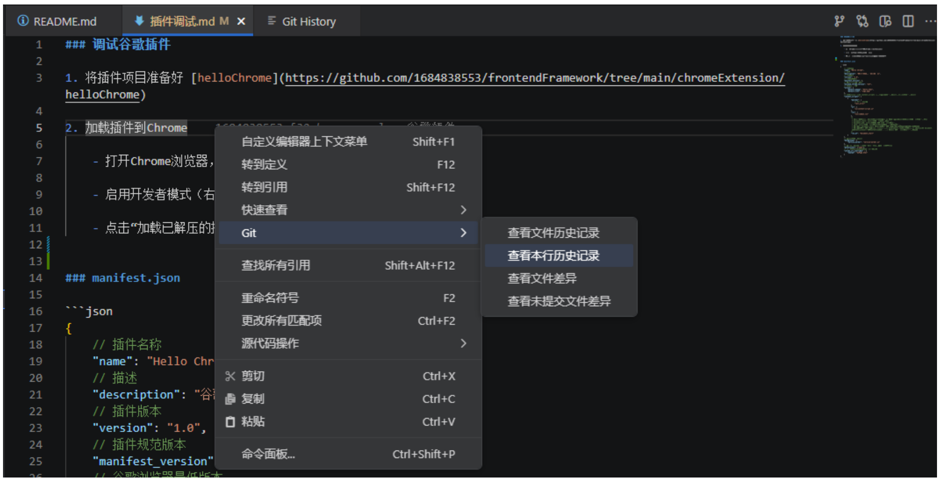
4. 查看本行历史记录
   a. 打开文件，选中某一行，在本行后面展示本行最新的提交信息。如下图所示：

图 7-77 追溯信息



b. 打开文件，右键单击“Git > 查看本行历史记录”，打开“Git History编辑器”，查看本行历史记录。如下图所示：

图 7-78 查看本行历史记录



8 使用 CodeArts IDE for C/C++

8.1 在 CodeArts IDE for C/C++创建 C/C++工程

CodeArts IDE for C/C++ 提供了创建C或C++工程的能力，参考以下步骤进行创建。

步骤1 单击顶部菜单“**文件 > 新建 >工程**”。

步骤2 在“新建工程”界面，参照表8-1完成参数填写。

图 8-1 新建工程

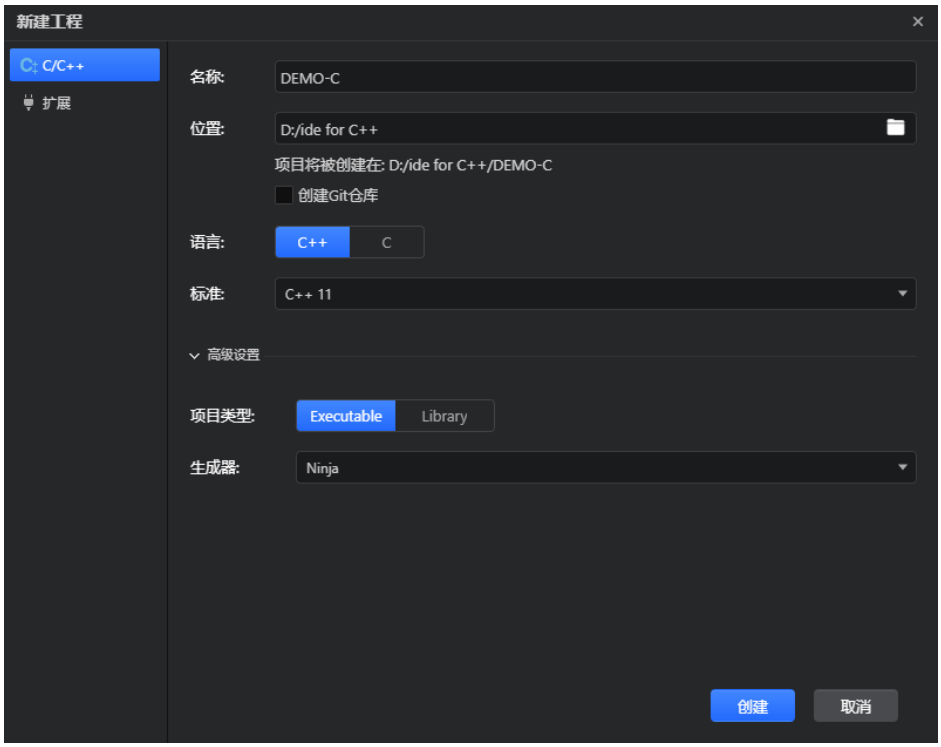
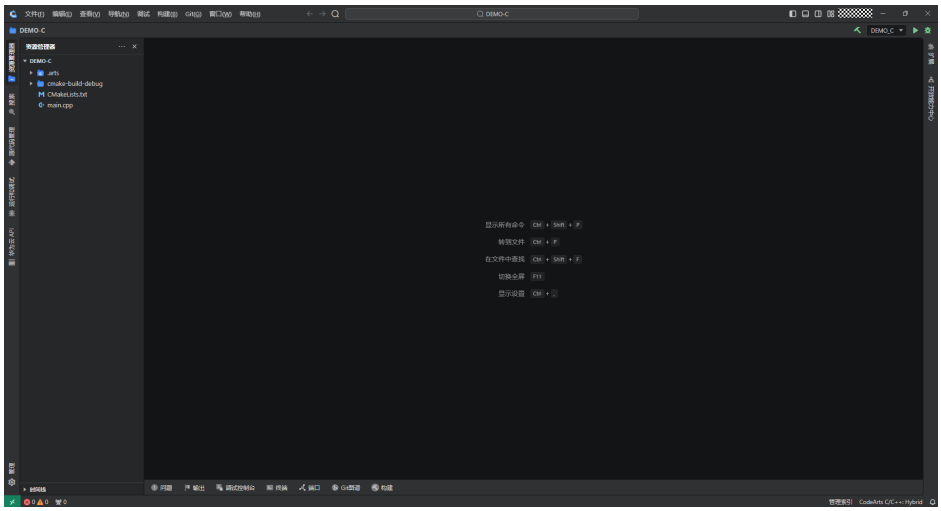


表 8-1 新建工程参数

参数	说明	示例
名称	输入工程的名称，以字母开头，支持字母、数字、中划线、下划线，长度不超过64。	DEMO-C
位置	用户自定义的工程存放的位置，支持中文路径。	D:/ide for C++
创建Git仓库	创建工程时是否需要创建Git仓库。 勾选该参数表示自从创建Git仓库，可以参考在创建的项目内自动**初始化Git存储库**。	勾选“创建Git仓库”
语言	选择C或C++任意一种语言。	C++
标准	在下拉框选择语言的标准版本。 • C++支持的版本：C++ 11、C++14、C++17、C++20、C++23、C++ 98 • C支持的版本：C 11、C 17、C 20、C 23、C 90、C 99	C++ 20
项目类型	• **Executable**：生成可直接运行的可执行文件，包含main函数，独立运行。 • **Library**：生成库文件（静态库或动态库），包含可被其他项目使用的函数和类，提高代码的复用性和模块化。	Executable
生成器类型	构建项目时，默认使用Ninja生成器。 • **MinGW Makefiles**：Windows系统上适用于对性能要求不高的场景。 • **Unix Makefiles**：Linux系统上适用于对性能要求不高的场景。 • **Ninja**：适合高性能构建需求。	Ninja

步骤3 单击“创建”按钮，等待工程创建完成并打开项目，如**图8-2**所示。

图 8-2 打开项目

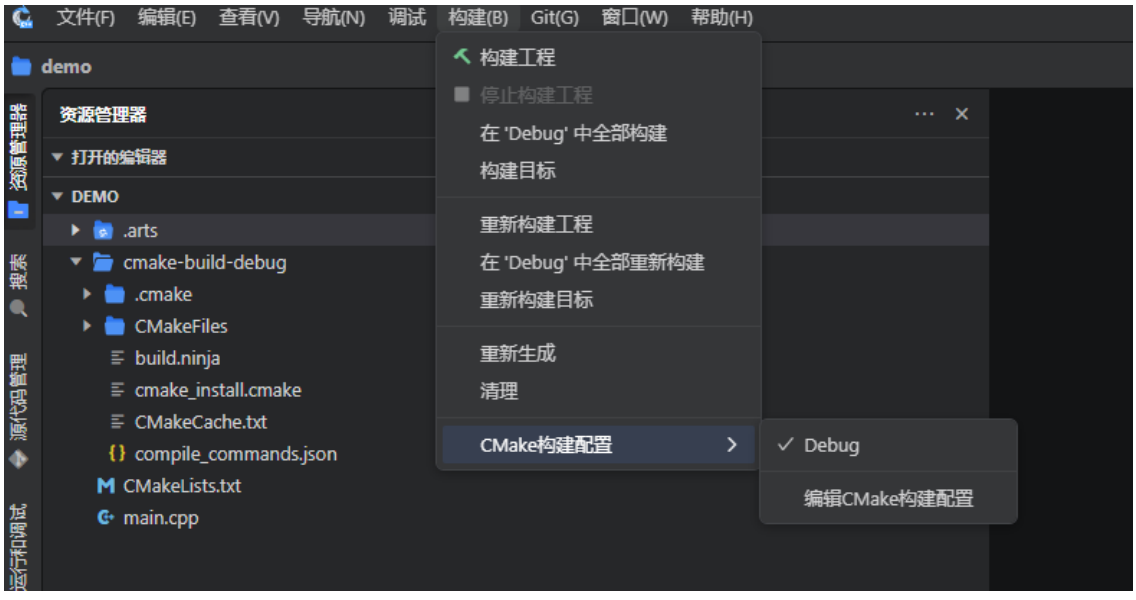


----结束

8.2 在 CodeArts IDE for C/C++构建 CMake 工程

CodeArts IDE for C/C++提供了构建工程、构建目标、重新构建工程、清理等不同的构建能力，如图8-3所示：

图 8-3 构建菜单



在“构建”菜单下，可以完成如下构建：

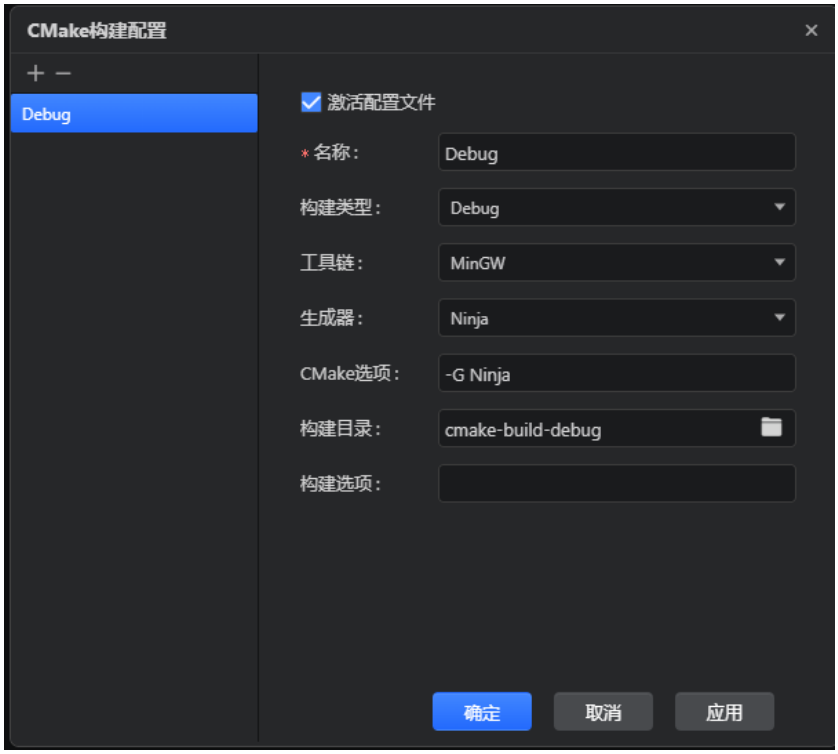
- 构建工程：构建整个工程的所有构建类型，并将源代码文件编译成可执行文件或库。
- 停止构建工程：停止正在进行的工程构建。
- 在'Debug'中全部构建：根据Debug类型构建整个工程。

- 构建目标：构建指定目标，是CMake构建系统生成的最终输出，这些输出可以是可执行文件、静态库、动态库或其他类型的文件。
- 重新构建工程：根据构建配置重新构建出整个工程的所有构建类型。
- 在'Debug'中全部重新构建：根据Debug类型重新构建整个工程。
- 重新构建目标：重新构建指定目标。
- 重新生成：重新生成CMake构建配置文件。
- 清理：清理构建产物。
- CMake构建配置：可切换当前的配置类型以及编辑构建配置。

构建 CMake 工程

步骤1 通过顶部菜单栏选择“构建 > CMake构建配置 > 编辑CMake构建配置”打开CMake配置构建面板来新增或修改配置，如图8-4所示，参数会自动填充，无需手动填写。

图 8-4 CMake 构建配置面板

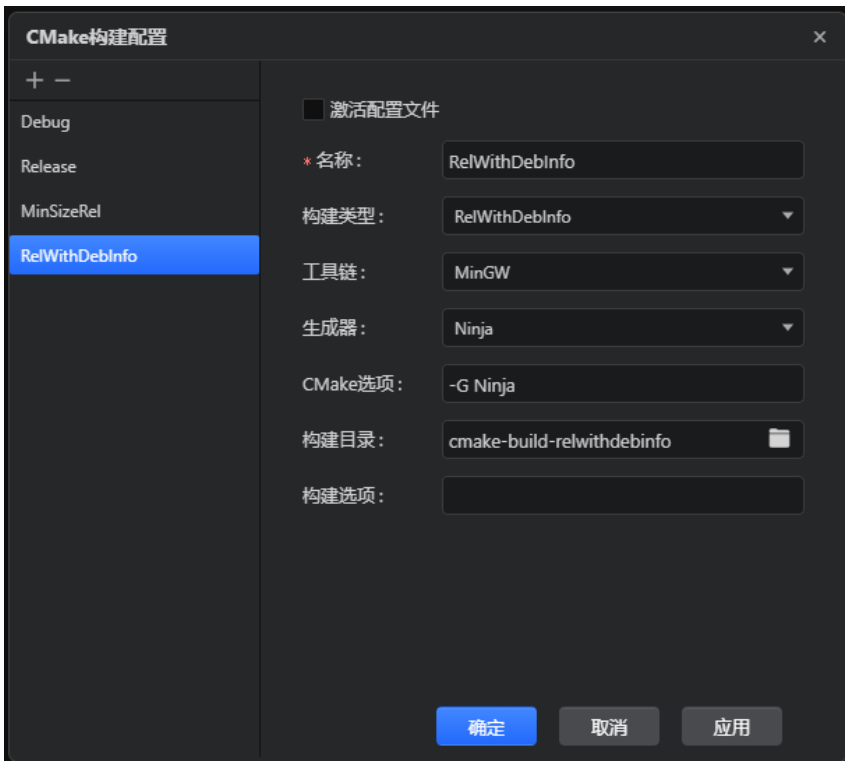


- CMake工程一共有四种**构建类型**，分别是Debug、MimSizeRel、Release和RelWithDebInfo。默认构建Debug类型，构建文件生成在cmake-build-debug文件夹下。
  - Debug：禁用优化并包含调试信息。
  - Release：包括优化且没有调试信息。
  - MinRelSize：优化大小，没有调试信息。
  - RelWithDebInfo：优化速度，包含调试信息。
- **生成器**：一共有三种类型，可以从下拉框中选择，默认为Ninja生成器。
  - **MinGW Makefiles**：Windows系统上适用于对性能要求不高的场景。

  - **Unix Makefiles**：Linux系统上适用于对性能要求不高的场景。
  - **Ninja**：适合高性能构建需求。
- **CMake选项**：与生成器相关联，选择不同的生成器显示不同。
  - 当生成器选择**Ninja**时，显示为**-G Ninja**。
  - 当生成器选择**MinGW Makefiles**时，显示为**-G "MinGW Makefiles"**。
  - 当生成器选择**Unix Makefiles**时，显示为**-G "Unix Makefiles"**。
- 构建目录：根据构建名称自动生成，格式为cmake-build-*名称*。

步骤2 CodeArts IDE提供了默认的构建选项，通过面板左侧菜单栏的新增➕按钮，可快速创建出Release、MinRelSize、RelWithDebInfo类型，如图8-5所示：

图 8-5 新增构建类型



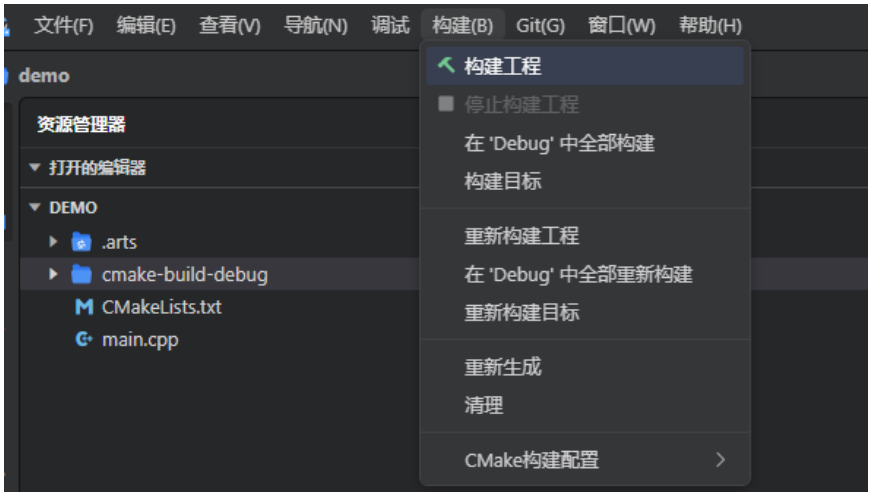
----结束

构建工程

步骤1 构建工程提供了以下三种方式，选择任意一种即可完成构建。

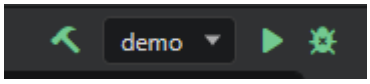
- 通过顶部菜单栏“构建 >构建工程”

图 8-6 顶部菜单栏构建工程



- 通过CodeArts IDE界面右上侧构建按钮，单击该按钮完成构建。

图 8-7 右上角构建工程



- 通过构建面板的左下角构建按钮，单击构建。

图 8-8 左下角构建工程

步骤2 构建完成后，日志将展示在构建面板右侧。构建日志记录了构建过程中发生的各种事件，包括编译器命令、警告、错误等。这些日志对于调试构建问题和理解构建过程很有帮助。

图 8-9 构建日志

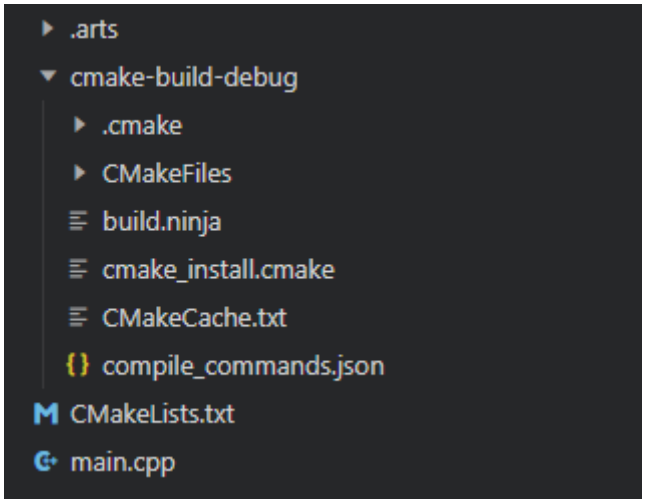
----结束

8.3 在 CodeArts IDE for C/C++加载 CMake 工程

CodeArts IDE for C/C++识别到打开的工程为CMake工程时，将会加载整个工程并自动完成以下操作。

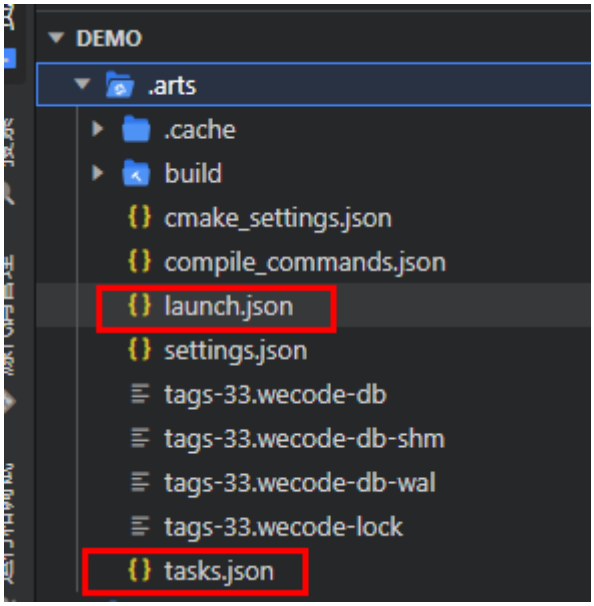
- 按照默认的Debug类型自行完成CMake的配置，构建文件生成在项目根目录下的cmake-build-debug目录，如下图所示：

  图 8-10 构建目录

- 构建目录中会生成compile_commands.json文件并自动导入，C/C++工程将会融合Huawei C/C++插件提供的精准跳转、代码重构，编译错误实时检查等能力。
- 在.arts目录下生成tasks.json和launch.json文件，如下图所示。可直接对C/C++工程进行调试和运行。
  - tasks.json是配置构建任务的文件。
  - launch.json是启动程序的配置文件。

  图 8-11 配置文件

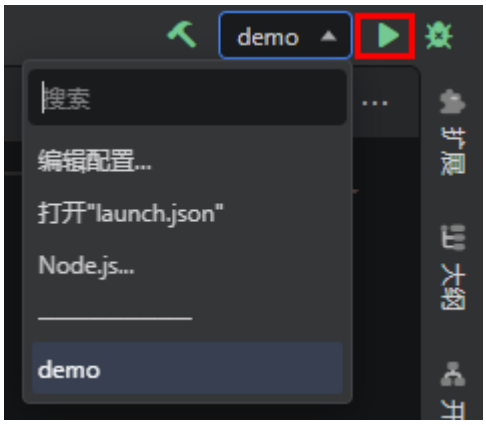
  

8.4 在 CodeArts IDE for C/C++运行 CMake 工程

可通过以下几种方式运行：

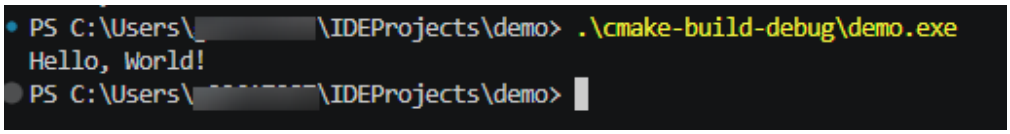
- 在运行和调试下拉框中选择需要运行的配置，单击运行按钮。

  图 8-12 通过工程界面运行

- 通过终端命令行运行可执行文件。

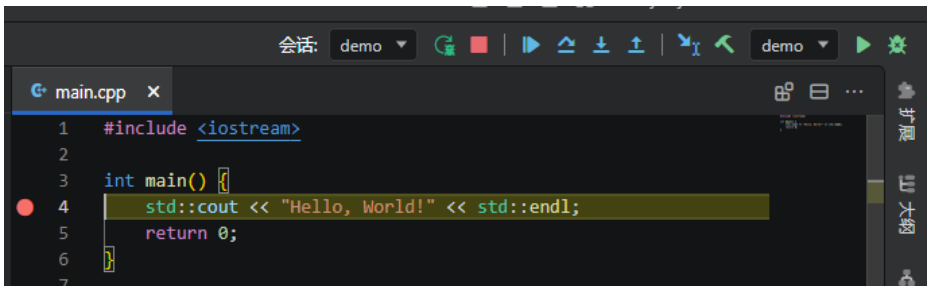
  图 8-13 通过终端命令行运行

调试 CMake 工程

打开cpp文件，设置程序断点，然后单击调试图标，如图8-14所示：

图 8-14 调试程序



8.5 使用 C/C++编辑代码

CodeArts IDE for C/C++ 包含了内置的语法着色、定义预览、跳转定义、类继承关系图和调用关系图等一些编码基础功能。

语法着色

该功能对函数、类型、局部变量、全部变量、宏、枚举和成员变量等上色。不同的颜色主题将展示出不同的颜色。

单击“管理 > 主题 > 颜色主题”进行切换。如下图所示：

图 8-15 语法着色

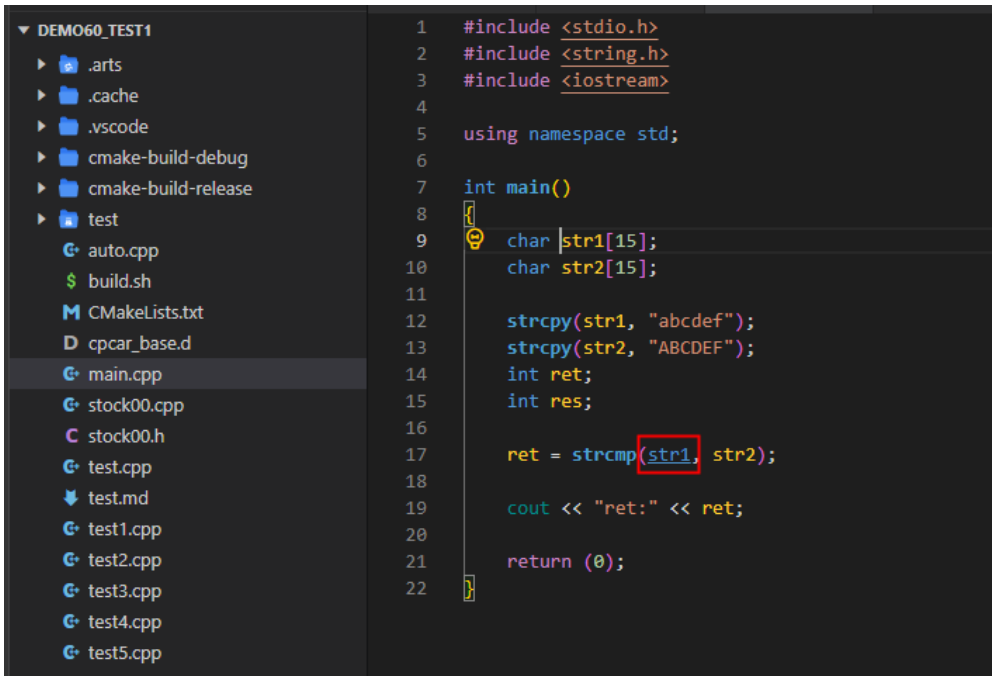
```
    exit(1);
  }
}else{
  fp_r = stdin;
}
mode[0]='w';
mode[1] = '0' + level;
mode[2] = '\0';

if((fn_w == NULL && (BZ2fp_w = BZ2_bzdopen(fileno(stdout),mode))==NULL)
  || (fn_w !=NULL && (BZ2fp_w = BZ2_bzopen(fn_w,mode))==NULL)){
  printf("can't bz2openstream\n");
  exit(1);
}
while((len=fread(buff,1,0x1000,fp_r))>0){
  BZ2_bzwrite(BZ2fp_w,buff,len);
}
BZ2_bzclose(BZ2fp_w);
if(fp_r!=stdin)fclose(fp_r);
```

跳转定义

当光标放在已定义的变量处，**“Ctrl+单击”**、**“F12”**或者使用**“Ctrl+Alt+单击”**光标会跳转到变量的定义处。

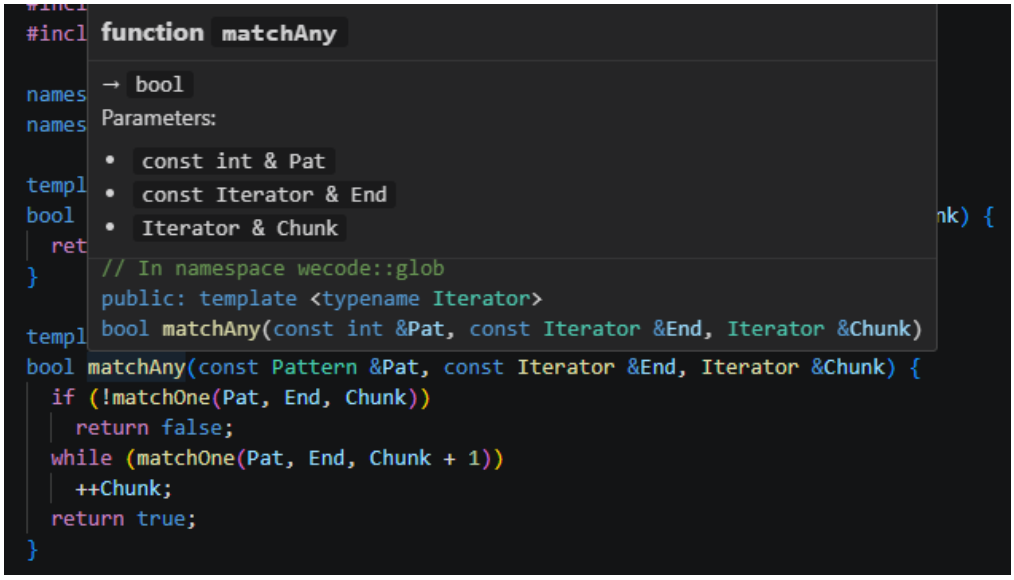
图 8-16 跳转定义



定义预览

当光标移至符号处，则会有符号定义的悬停预览，也可以用“alt+F12”的快捷键进行文件内的符号预览。

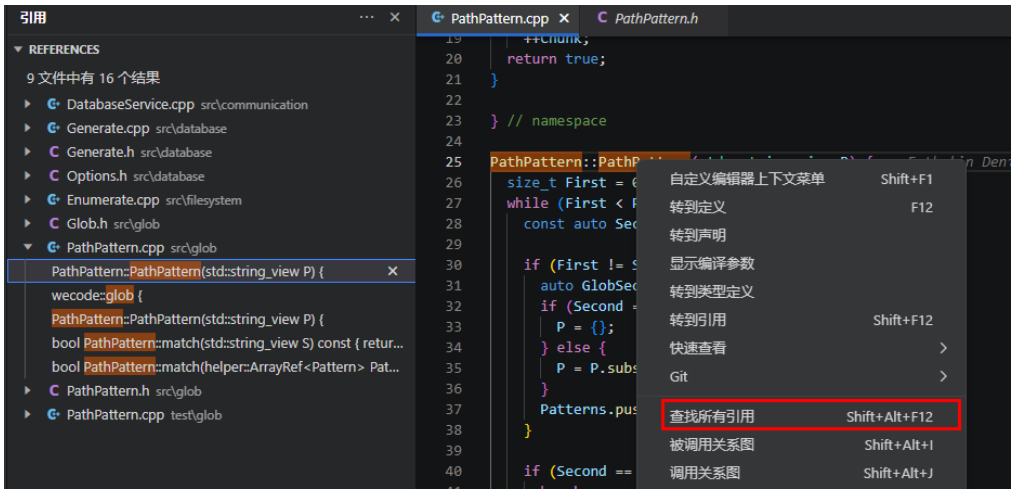
图 8-17 符号预览



查找所有引用

当光标单击或者选择到需要查找的符号，右键“查找所有引用”或者使用快捷键“Shift+Alt+F12”会打开定义在页面左侧，如下图所示：

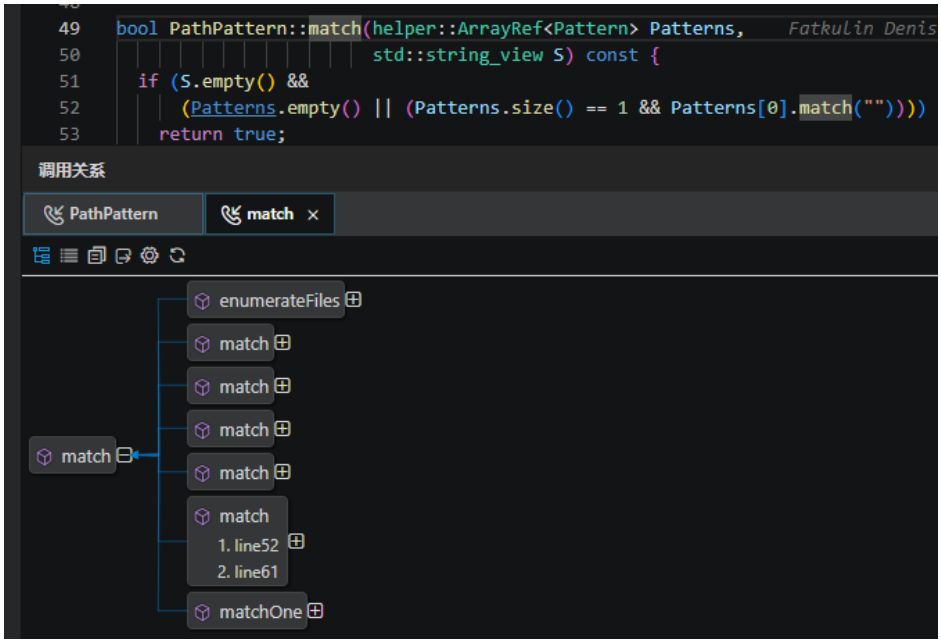
图 8-18 查找引用



调用关系图

当光标选中需要调用关系图的函数时，右键“调用关系图”，或可以使用快捷键“Shift+Alt+J”调出。在关系图中，可以双击需要查看的函数并导航到该函数。

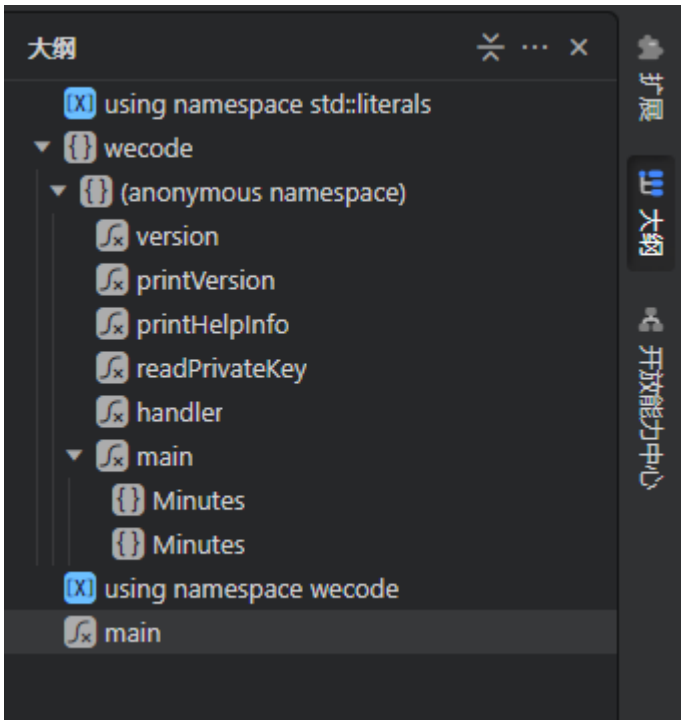
图 8-19 调用关系图



符号大纲

右侧导航栏，单击“大纲”即可打开符号大纲，在大纲视图双击符号可跳转到符号定义的位置，并且当前符号大纲可跟随光标移动（此功能需要在大纲菜单栏中打开**跟随光标**选项）。

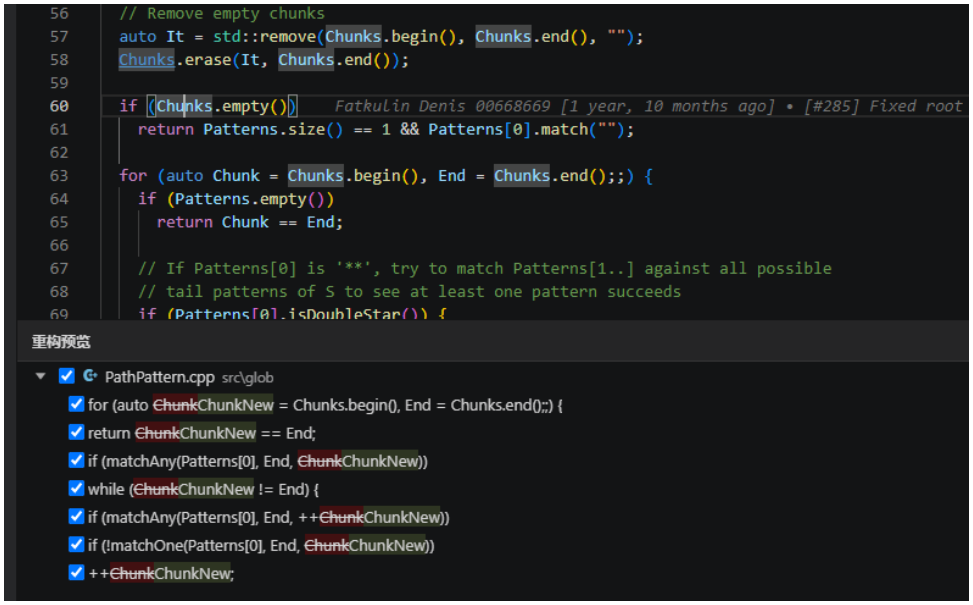
图 8-20 打开符号大纲



符号重命名（Rename Symbol）

光标选中某个符号，右键菜单选择重命名符号或直接按“**F2**”，来重命名工程中所有用到该符号的地方。

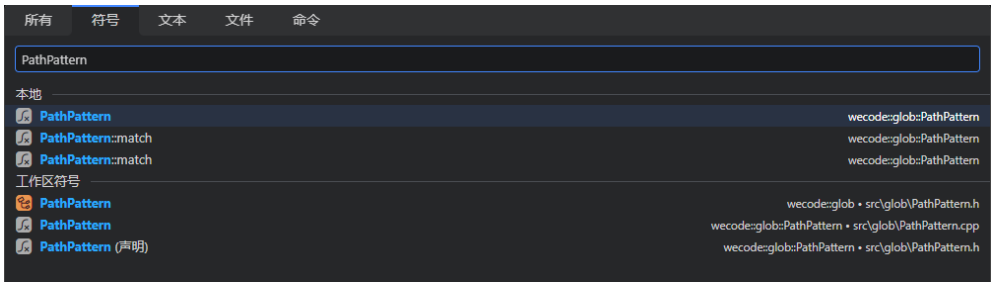
图 8-21 符号重命名



全局符号搜索（Global Symbol Search）

Ctrl+T打开搜索框，输入需要查找的符号，页面会显示出当前工程所有包含此符号的文件，单击即可跳转到对应的文件内容。或者按**向上**或**向下**选择并按**Enter**导航到想要的位置。

图 8-22 全局符号搜索

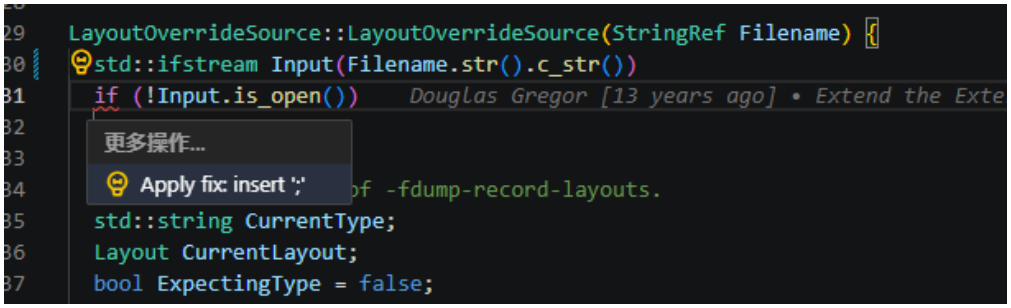


编译错误实时检查

编译错误实时检查是在代码编写过程中实时检查代码语法错误，该功能依赖compile_commands.json文件。

当出现语法错误时，会在错误处出现波浪线。将光标移动到错误代码位置，会显示黄色“灯泡”图标，表示可以使用快速修复。单击图标或按**Ctrl+.**会显示可用的快速修复。

图 8-23 编译错误实时检查



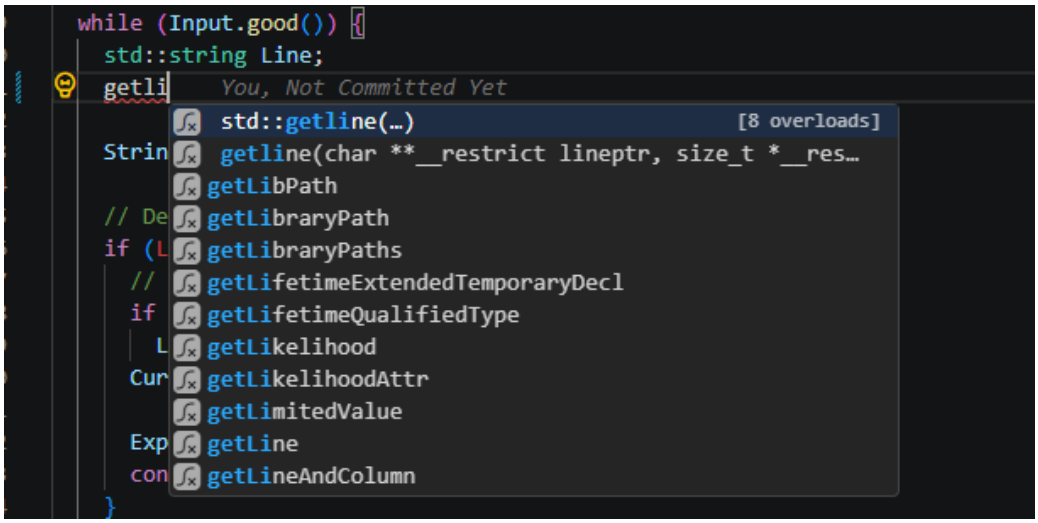
8.6 使用 C/C++补全代码

CodeArts IDE for C/C++包含了内置代码补全/提示（Code Completion/Hinting）的功能。

CodeArts IDE for C/C++提供了对函数、变量、类成员、结构体、枚举和命名空间等符号的补全能力。

当编写代码时，根据已输入字符来匹配工程中可以补全的符号，弹出补全列表。可通过光标单击或者按“**Enter**”或“**Tab**”键插入选定的补全符号，帮助更加方便快速地编辑代码。

图 8-24 代码补全



8.7 使用 C/C++重构代码

重构是通过改变现有程序结构而不改变其功能和用途来提高代码的可重用性和可维护性。

CodeArts IDE提供了多种重构方法，在使用重构之前，请确保compile_commands.json文件已自动导入。构建文件会生成在项目根目录下，如cmake-build-debug目录。具体请参见**在CodeArts IDE for C/C++加载CMake工程**。

选择需重构的代码块，如有可用的重构操作，代码处会出现黄色小灯泡提示，或右键菜单选择“重构”查看可用的重构操作。

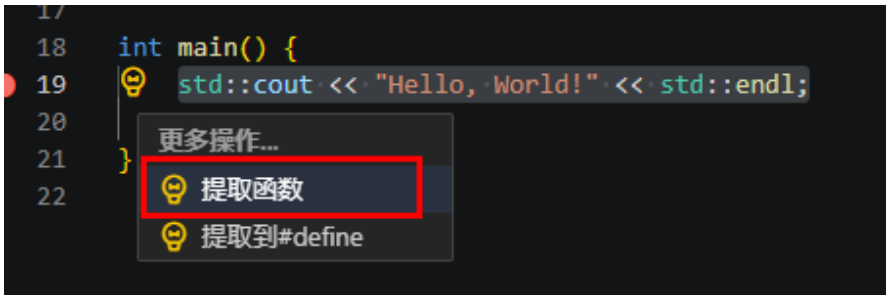
提取重构（Extraction refactoring）

CodeArts IDE for C/C++ 支持将字段、方法和参数提取到新类中，根据提取的内容会提供不同的重构类型。

- **提取函数/方法（Extract method）**

  将选定的语句或表达式提取到文件中的新方法或新函数。在选择**提取函数/方法**重构后，输入提取的方法/函数的名称。

  图 8-25 提取函数

  执行重构示例

  重构前：

  ```
  int main() {
      std::cout << "Hello, World!" << std::endl;
      return 0;
  }
  ```

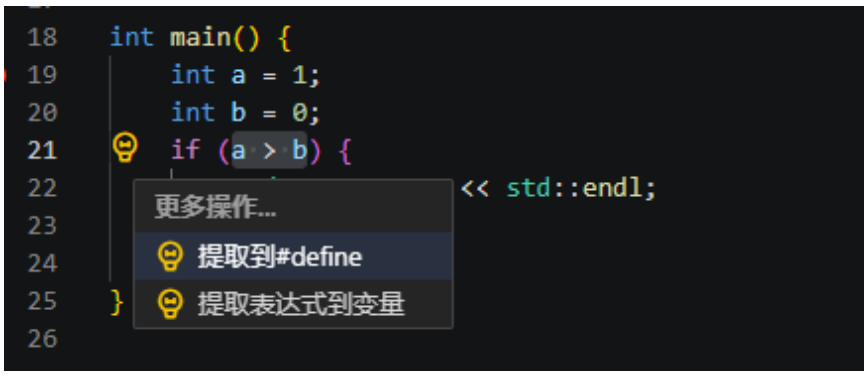
  重构后：

  ```
  void extracted()
  {
      std::cout << "Hello, World!" << std::endl;
  }
  int main()
  {
      extracted();
      return 0;
  }
  ```

- **提取表达式到变量（Extract subexpression to variable）**

  将选定的表达式提取为文件中的新变量。

  图 8-26 提取表达式到变量

  执行重构示例

重构前：

```
int main()
{
    int a = 1;
    int b = 0;
    if (a > b)
    std::cout << "Hello, World!" << std::endl;
}
```

重构后：

```
int main()
{
    int a = 1;
    int b = 0;
    auto comparisonResult = a > b;
    if (comparisonResult)
        std::cout << "Hello, World!" << std::endl;
}
```

定义构造函数（Define constructor)

在每次创建类时，可以自动定义类的构造函数，并且初始化成员。当单击或选中类名时，可以单击左侧黄色灯泡选择**定义构造函数**。

图 8-27 定义构造函数

执行重构示例

重构前：

```
class Bar {
    int value = 1;
}
```

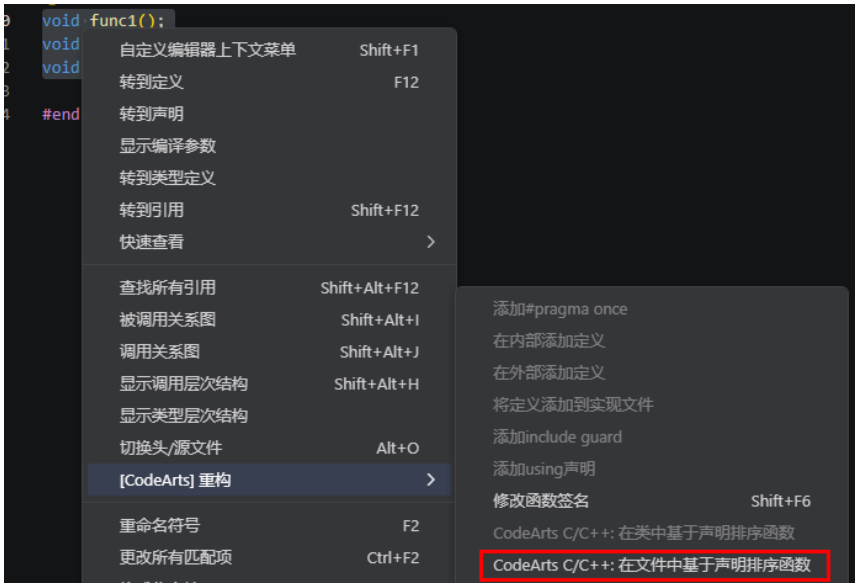
重构后：

```
class AbstractBar {
    int value = 1;
};
class Bar : public AbstractBar {}
```

基于声明排序函数（Sort functions to declarations）

根据头文件中的声明顺序，排序当前定义函数/方法的顺序。当单击或选中当前函数/方法定义时，重构选项可用。

图 8-28 基于声明排序函数



执行重构示例

重构前：

```
void func1();
void func2();

void func2()
{
    return 0;
}
void func1()
{
    return 0;
}
```

重构后：

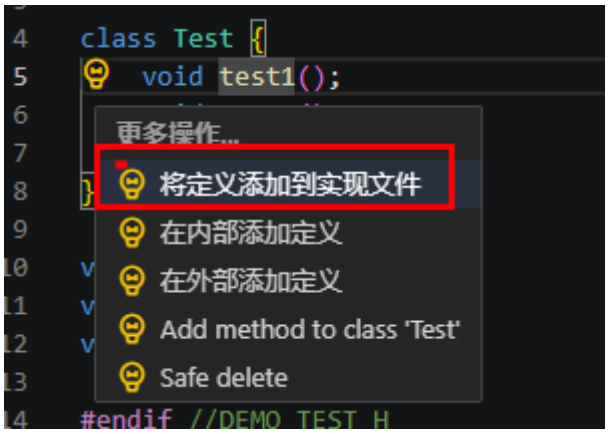
```
void func1();
void func2();

void func1()
{
    return 0;
}
void func2()
{
    return 0;
}
```

将定义添加到实现文件（Add definition to implementation file)

选中头文件，将定义添加到实现文件中。当单击或选中当前函数/方法时，重构选项可用。

图 8-29 将定义添加到实现文件



执行重构示例

重构前：

```
class test {
    int testAdd();
}
```

重构后：

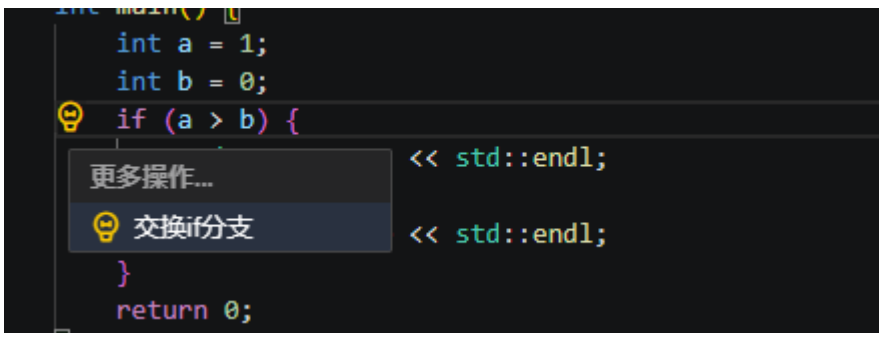
```
// test.h文件
class test {
    int testAdd();
}

// test.cpp文件
int test::testAdd()
{
    /* Not implemented yet */
    return {};
}
```

交换 if 分支（Swap if branches)

若当前条件只有if和else分支，选中代码片段后，选择**交换if分支**，可自动交换if和else分支。

图 8-30 交换 if 分支



执行重构示例

重构前：


```
int main()
{
    int a = 1;
    int b = 2;
    if (a < b) {
        std::cout << "Hello, World!" << std::endl;
    } else {
        std::cout << "Hello, Today!" << std::endl;
    }
}
```

重构后：

```
int main()
{
    int a = 1;
    int b = 2;
    if (a < b) {
        std::cout << "Hello, Today!" << std::endl;
    } else {
        std::cout << "Hello, World!" << std::endl;
    }
}
```

内联变量（Inline variable)

该功能可以用相应的值替换所有引用。假设计算值总是产生相同的结果。选中需要替换的内容，重构选项可用。

图 8-31 内联变量

执行重构示例

重构前：

```
int main()
{
    int a = func();
    int b = a;
}
```

重构后：

```
int main()
{
    int b = func();
}
```

内联函数（Inline function)

该功能尝试使用适当的代码内联所有函数用法。它只能处理简单的功能，不支持内联方法、函数模板、主函数和在系统头文件中声明的函数，该功能可以内联所有函数引用。

图 8-32 内联函数

执行重构示例

重构前：

```
int func(){
    return 1;
}
int main()
{
    int a = func();
    int b = a;
}
```

重构后：

```
int func(){
    return 1;
}
int main()
{
    int a = 1;
    int b = a;
}
```

生成 getter 和 setter（Generate getter and setter)

通过为**生成getter和setter**来封装选定的类属性。同时也可以选择只**生成getter**或者**生成setter**选项。

图 8-33 生成 getter 和 setter

执行重构示例

重构前：

```
class AbstractFoo {
    int value = 1;
};
```

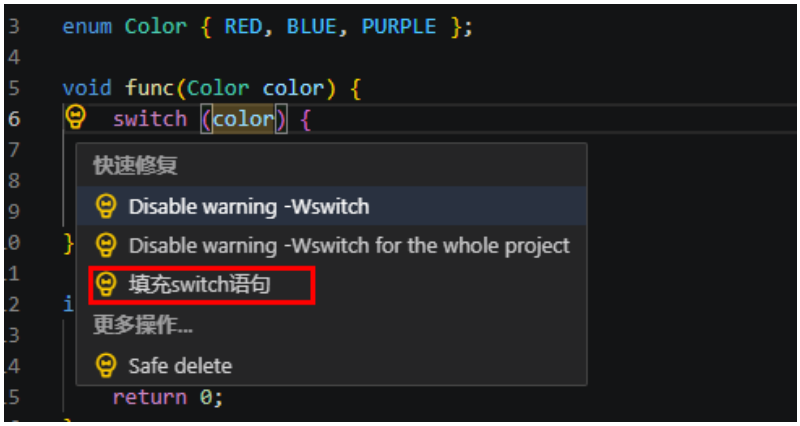
重构后：

```
class AbstractFoo {
public:
    int getValue() const;
    void setValue(int NewValue);
private:
    int value = 1;
};
int AbstractFoo::getValue() const
{
    return value;
}
void AbstractFoo::setValue(int NewValue)
{
    value = NewValue;
}
```

填充 switch 语句（Populate switch）

该功能可以自动填充switch语句。选中任意switch字段，并且单击黄色灯泡，选择**填充switch语句**。

图 8-34 填充 switch 语句



执行重构示例

重构前：

```
enum Color { RED, BLUE };
void getColor(Color color) {
    switch (color) {

    }
}
```

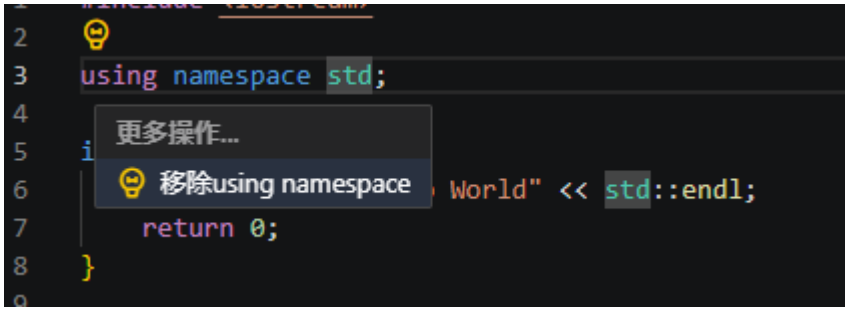
重构后：

```
enum Color { RED, BLUE };
void getColor(Color color) {
    switch (color) {
        case RED:
        case BLUE:
            break;
    }
}
```

移除 namespace（Remove using namespace）

移除namespace功能，会自动移除所有使用到的namespace。当光标单击或选中namesapace关键字时，重构选项可用。

图 8-35 移除 namespace



执行重构示例

重构前：

```
using namespace std;
int main() {
    cout << "Hello, World!" << endl;
}
```

重构后：

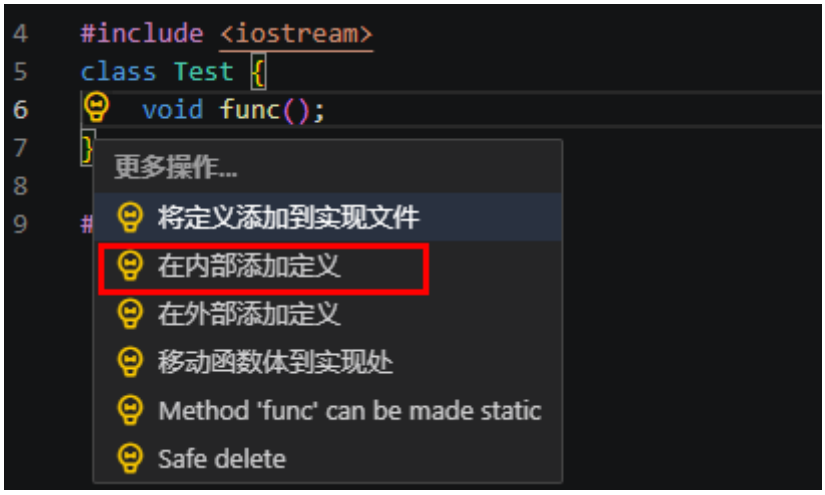
```
int main() {
    std::cout << "Hello, World!" << std::endl;
}
```

添加定义

- **在内部添加定义（Add definition in-place）**

  在类内部生成当前函数/方法的函数定义。当光标移动到函数/方法时，单击“重构”，重构选项可用。

图 8-36 在内部添加定义



执行重构示例

重构前：

```
class Test {
    void func();
}
```

重构后：

```
class Test {
    void func()
    {
        /* Not implemented yet */
    }
}
```

- **在外部添加定义（Add definition out-of-place）**

  在类外部生成当前函数/方法的函数定义。当光标移动到函数/方法时，单击重构按钮，重构选项可用。

  执行重构示例

  重构前：

  ```
  class Test {
      void func();
  }
  ```

  重构后：

  ```
  class Test {
      void func();
      void Test::func()
      {
          /* Not implemented yet */
      }
  }
  ```

展开宏（Expand macro）

在页面上添加展开宏（Expand macro），以便在可扩展/可折叠的部分提供内容。

图 8-37 展开宏

执行重构示例

重构前：

```
#define TEST 1
int main() {
    int a = TEST;
    std::cout std:: << "Hello, World!" << std::endl;
}
```

重构后：

```
#define TEST 1
int main() {
    int a = 1;
    std::cout std:: << "Hello, World!" << std::endl;
}
```

替换为推导类型（Replace with deduced type）

展开auto type所隐藏的变量类型，并替换为推导类型。

图 8-38 替换为推导类型

执行重构示例

重构前：

```
void replaceType()
{
    int b = 1;
    auto a = b;
}
```

重构后：

```
void replaceType()
{
    int b = 1;
    int a = b;
}
```

移动函数体到声明处（Move function body to declaration）

该功能会将函数/方法的定义移动到声明的位置。

图 8-39 移动函数体到声明处

执行重构示例

重构前：

在test.cpp文件中，定义test函数。

```
void test()
{
    std::cout << "Hello, World!" << std::endl;
}
```

重构后：

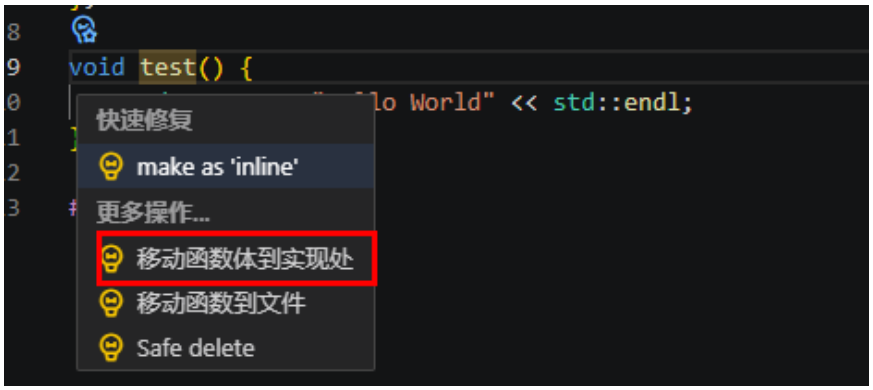
test.cpp文件中的test函数移动到test.h文件中。

```
inline void test()
{
    std::cout std:: << "Hello, World!" << std::endl;
}
```

移动函数体到实现处（Move function body to out-of-line）

该功能会将函数/方法的定义移动到对应的文件中。

图 8-40 移动函数体到实现处



执行重构示例

重构前：

在头文件中定义一个函数，光标放在函数名处。

```
void test() {
    std::cout << "Hello, World!" << std::endl;
}
```

重构后：

头文件中定义函数，函数体移动到实现处。

```
void test();
```

转为原始字符串（Convert to raw string）

该功能可以将转义后的字符串转换为原始的字符串，当单击或选择了当前字符串，单击重构图标，重构选项可用。

图 8-41 转为原始字符串

执行重构示例

重构前：

```
int myPrint() {
    std::cout << "\"a\"\n" << std::endl;
}
```

重构后：

```
int myPrint() {
    std::cout << R"("a"
)" << std::endl;
}
```

9 使用 CodeArts IDE for Java

9.1 在 CodeArts IDE for Java 创建 Java 项目

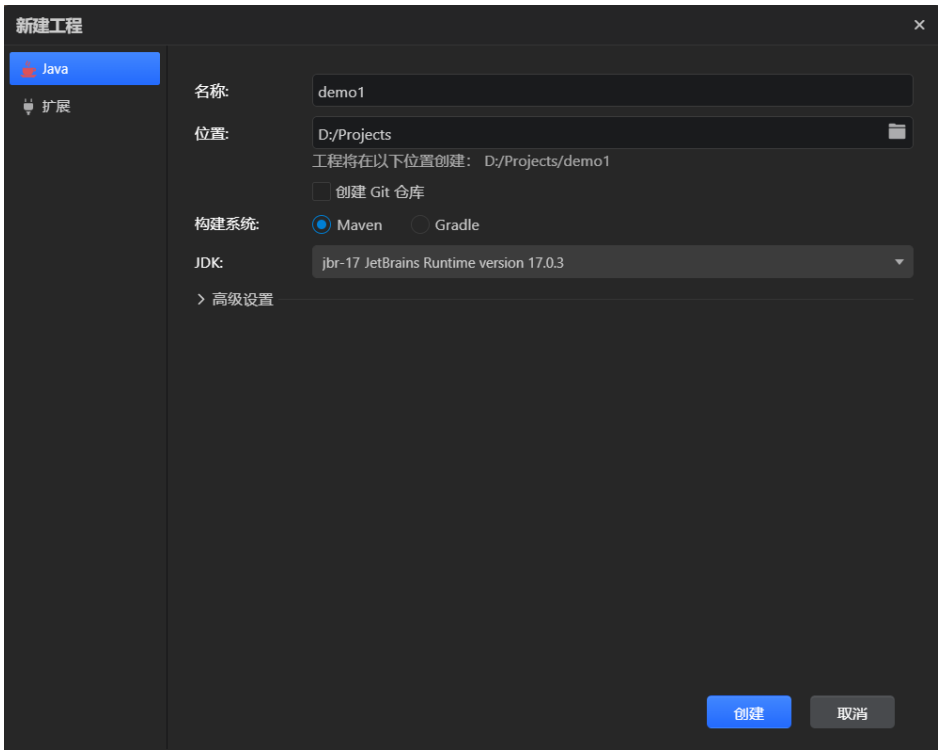
本章节介绍了如何通过CodeArts IDE for Java新建Java项目、创建Java类和查看项目依赖关系。

新建项目

步骤1 打开CodeArts IDE，单击“文件 > 新建 > 工程”。

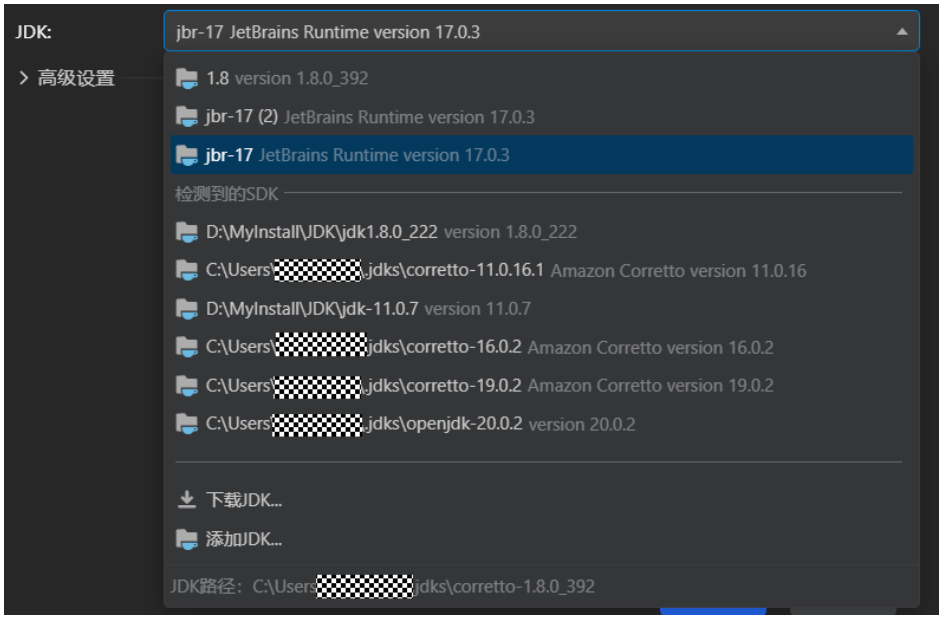
步骤2 选择“Java”，其中“名称”和“位置”需要用户自定义。其它参数可参考如下参数说明进行选择。如下图所示：

图 9-1 新建工程页面



- **“创建Git仓库”**：要在创建的项目内自动**初始化Git仓库**，勾选此复选框。
- “构建系统”：选择“Maven”或“Gradle”以创建对应的Maven或Gradle工程。
- “JDK”：CodeArts IDE会自动检测系统上安装的JDK，并在JDK列表中显示它们。
- 若未安装JDK，CodeArts IDE会在下拉框提示您“下载JDK...”。若已下载JDK，但是未配置JDK安装路径，也可以单击“添加JDK...”来添加本地已下载的JDK。如下图所示：

图 9-2 JDK 列表



说明

C:\Users*用户名*\.jdks文件夹下的JDK可以被CodeArts IDE自动检测到。若您想下载多个版本的JDK，优先考虑将下载的JDK解压到此路径下，路径中的用户名需要替换成系统的用户名。

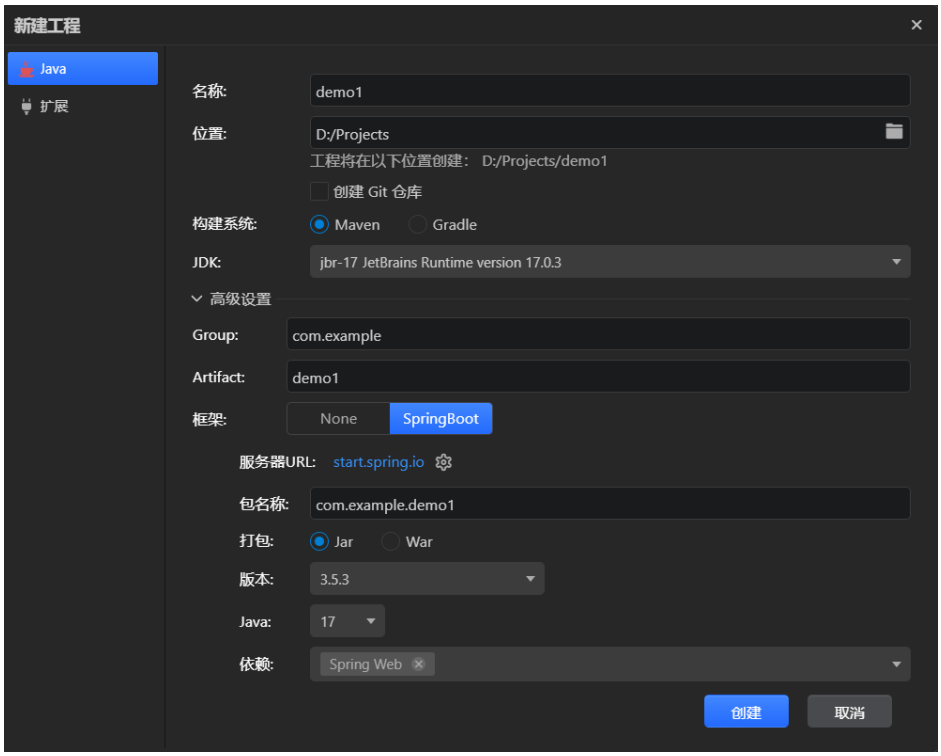
步骤3 在“高级设置”下，**Group**和**Artifact**信息会默认填充，其中Artifact与工程名称相同。

说明

Group和Artifact也支持用户自定义修改，必须以字母开头，可输入点、字母、数字、中划线、下划线，长度为1~64。

如果需要创建Spring Boot框架，参考此步骤操作。否则，直接跳过该步骤。

图 9-3 新建 SpringBoot 工程页面



1. 将“**框架**”设置为“**SpringBoot**”。
2. 在“**版本**”选项中，选择Spring Boot的版本。
3. 在“**Java**”选项中，选择项目的Java语言级别。语言级别定义了代码编辑器提供的一组编码辅助功能（如代码补全或错误突出显示）。
4. 在“依赖”列表中搜索并选择所需的项目依赖项。所选的项目依赖项将添加到**pom.xml**（Maven项目）或**build.gradle**（Gradle项目）文件中。

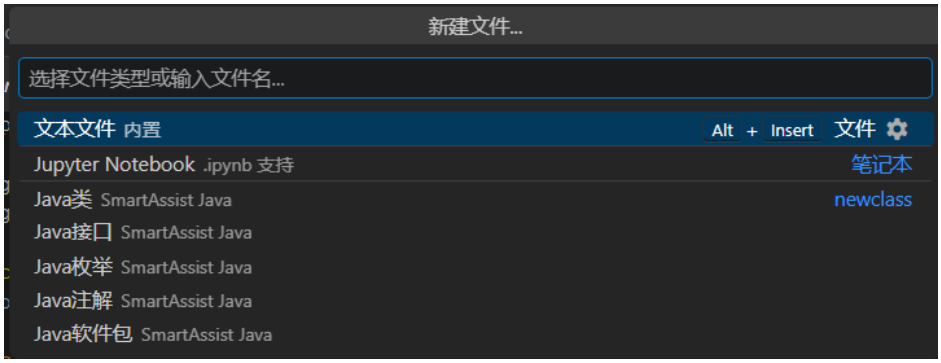
步骤4 单击“**创建**”按钮。CodeArts IDE会根据所选的模板创建对应的项目。

----结束

创建文件

步骤1 单击“文件 > 新建 > 文件...”或按下“Ctrl+Alt+Win+N”快捷键，打开的“新建文件...”弹出窗口中，根据需要选择创建对应的文件类型。如下图所示：

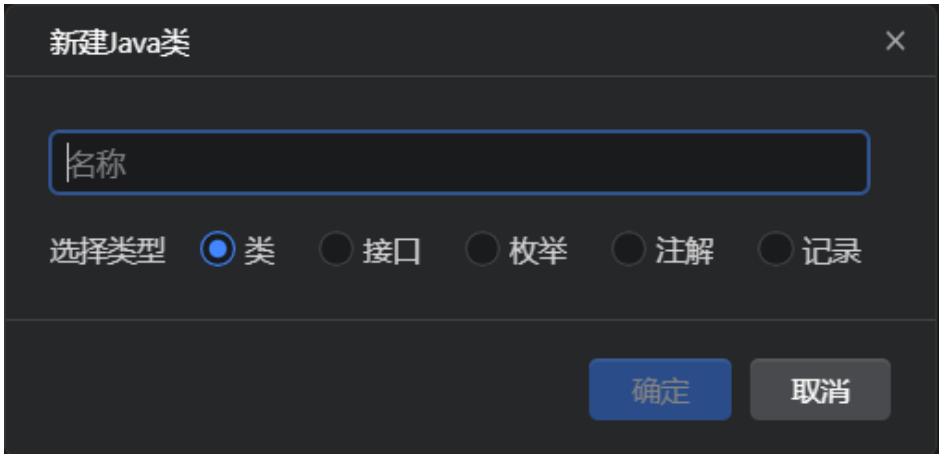
图 9-4 新建文件



步骤2 以新建Java类为例，在打开的“新建Java类”对话框中输入文件名。如图 **新建Java类** 所示：

- 类名可以包含字母、数字、下划线（ _ ）和美元符号（ $ ），但不能以数字开头。
- 类名不能是Java的关键字或保留字。例如，class、int、public等。

图 9-5 新建 Java 类



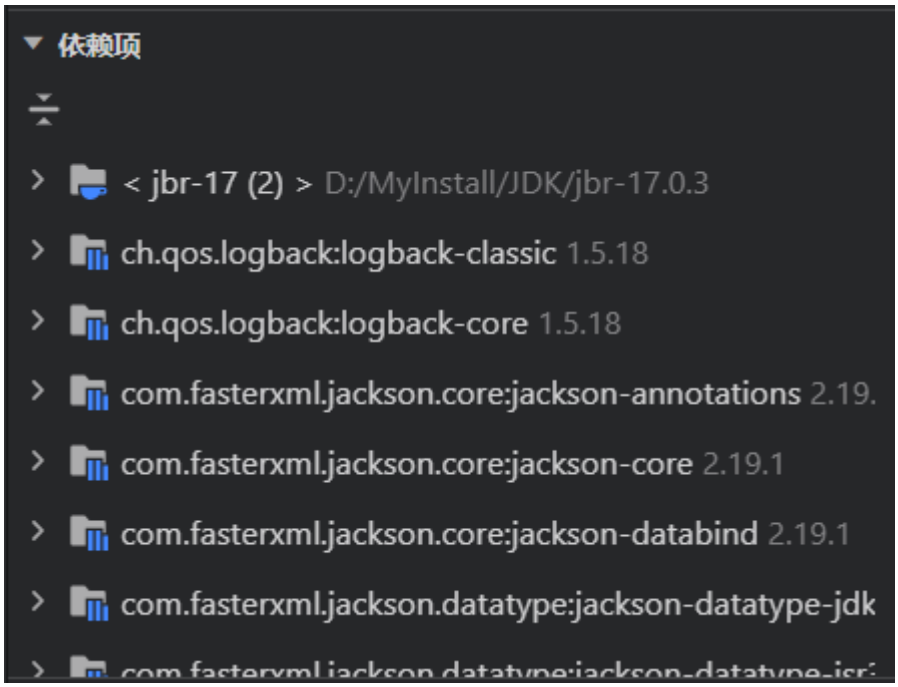
步骤3 单击**“确定”**。CodeArts IDE会在指定的目录中创建文件，并根据所选的文件模板填充它的内容。

----结束

查看项目依赖关系

如果项目在**pom.xml**（Maven项目）或**build.gradle**（Gradle项目）中声明了依赖项，CodeArts IDE会在“资源管理器”视图中的**“依赖项”**视图显示它们。配置的JDK的内容也会显示在**“依赖项”**子视图中。如下图所示：

图 9-6 新建工程页面新建普通工程依赖项示例



用户可以浏览依赖项列表，并以只读模式在代码编辑器中打开文件。对于Maven项目，若需要进一步分析依赖项之间的关系，可以参考分析Maven依赖关系章节。

说明

- 如果“依赖项”子视图除了“JDK”依赖项外，**pom.xml**（Maven项目）或**build.gradle**（Gradle项目）文件中配置的依赖项也能正常显示，则说明项目解析正常。
- 如果**“依赖项”**被隐藏了，可以在“资源管理器”中单击“视图和更多操作”按钮，并在弹出菜单中启用**“依赖项”**来显示它。

9.2 在 CodeArts IDE for Java 配置 Java 项目

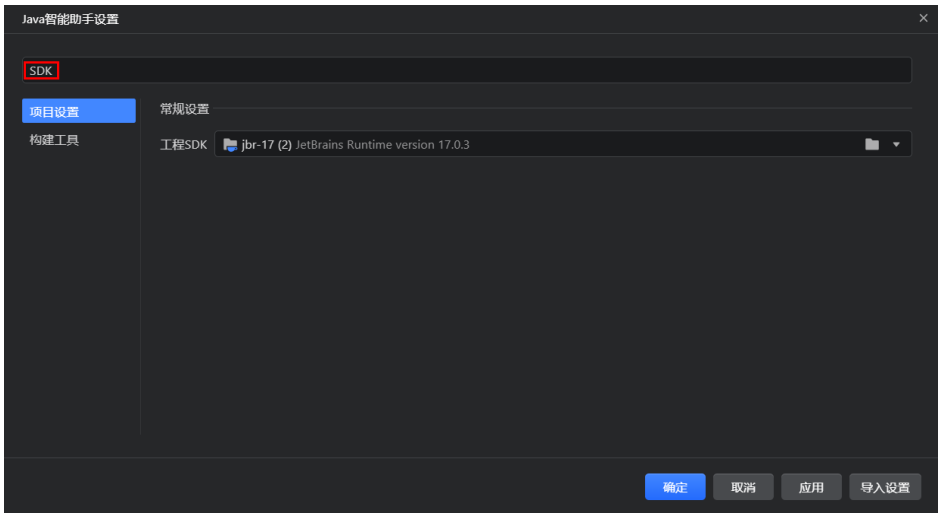
Java 智能助手设置

要配置Java项目，需要打开“Java智能助手设置”对话框，可以通过以下任意一种方式打开“Java智能助手设置”对话框：

- 通过“Ctrl+Shift+P”或者“Ctrl+Ctrl”打开“命令”面板，输入“SmartAssist:Java智能助手设置”命令。
- 单击CodeArts IDE左下角**“管理>Java智能助手设置”**。

打开“Java智能助手设置”对话框之后，可通过搜索关键字快速定位所需的设置。如下图所示：

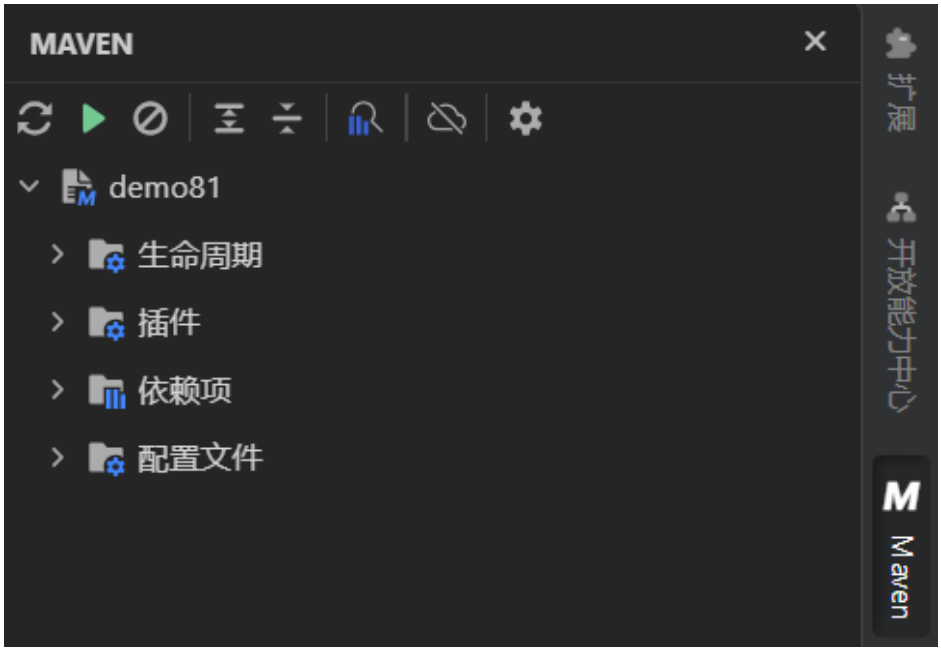
图 9-7 搜索 SDK 关键字示例



配置并运行 Maven 任务

Maven是一个用于管理Java项目和自动化应用程序构建的工具。CodeArts IDE集成了Maven支持。专用的**“Maven”**视图与**pom.xml**同步，并提供了一个可视化界面，用于查看项目依赖项或运行Maven任务。如下图所示：

图 9-8 Maven 视图



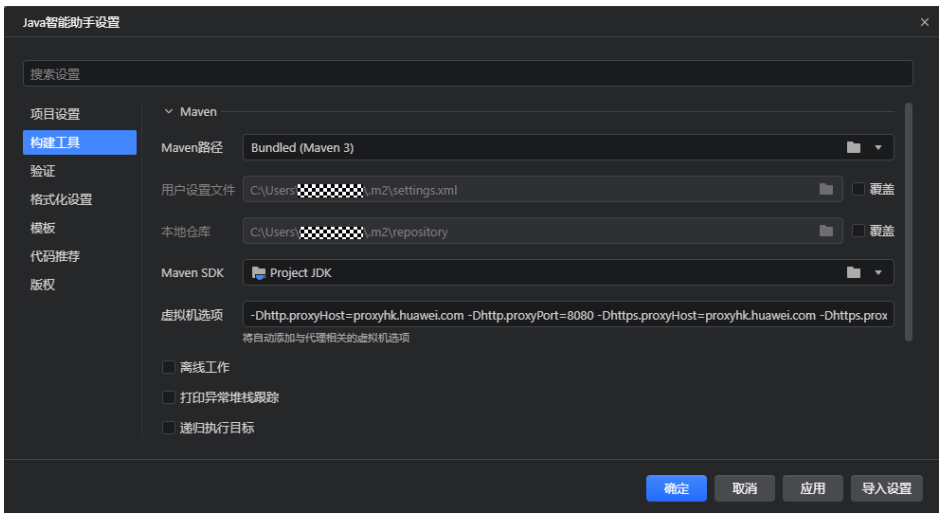
说明

在修改**pom.xml**文件后，例如添加新的依赖项，单击“Maven”视图工具栏上的**“重新加载所有Maven工程”**按钮（ ），以使CodeArts IDE应用新的更改。

- 自定义Maven配置选项

  打开**Java智能助手设置**，切换到**“构建工具”**页签，在**“Maven”**部分自定义配置选项。

  图 9-9 自定义配置选项

  a. “Maven路径”：在此字段中选择捆绑的Maven版本（Maven 3），或单击浏览按钮（ ）手动定位用户自己的**Maven**安装路径。

  b. “用户设置文件”：在此字段中指定包含用户特定配置的配置文件。

c. “本地仓库”：在此字段中指定存储下载内容和临时构建产物的本地目录路径。

d. “Maven SDK”：从此列表中选择要与Maven一起使用的JDK：绑定的JDK、项目级别的JDK或从系统变量（如**JAVA_HOME**）解析的JDK。

e. “虚拟机选项”：Java命令的CLI参数。例如：-ea -Xmx2048m。

f. “离线工作”：如果选择此项，Maven将在离线模式下工作。将无法连接到远程仓库，只使用本地可用的资源。此选项对应于**--offline**命令行选项。

g. “打印异常堆栈跟踪”：如果选择此项，将生成异常堆栈跟踪。此选项对应于**--errors**命令行选项。

h. “递归执行目标”：如果选择此项，将递归执行构建，包括嵌套项目。此选项对应于**--non-recursive**命令行选项。

- 分析Maven依赖关系

  在“Maven”视图中，单击“分析依赖关系”按钮（  ），打开“依赖分析器”弹窗中，在弹窗中可以根据关键字搜索对应的依赖项或查看并解决有冲突的依赖。如下图所示：

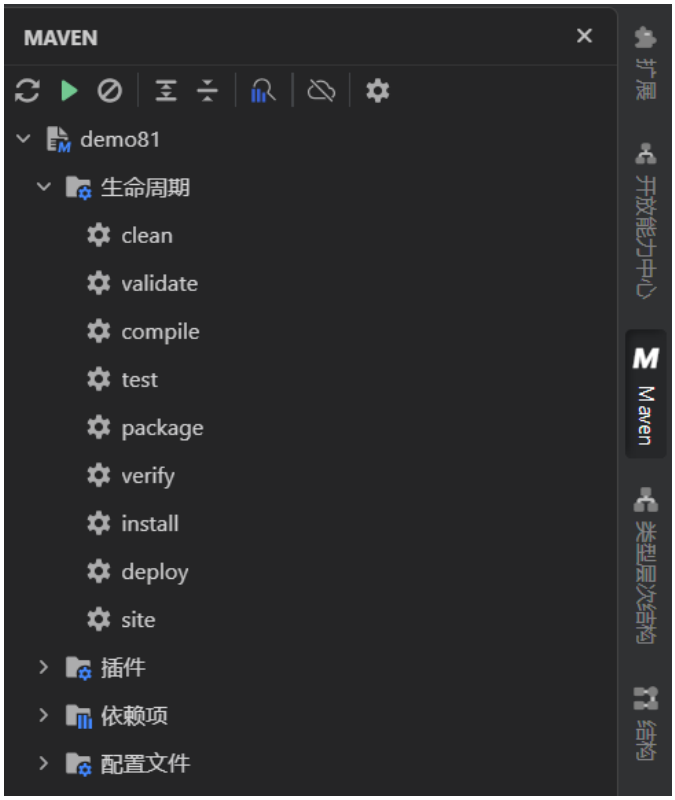
  图 9-10 依赖分析器

- 运行Maven任务

  在CodeArts IDE中打开一个Maven项目后，可在“Maven”视图的“生命周期”栏中找到Maven任务列表。如下图所示：

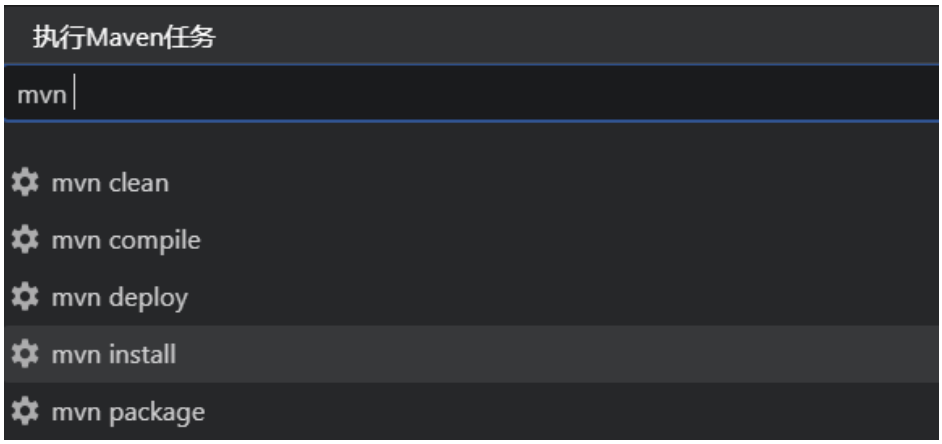
图 9-11 Maven 任务列表



要运行Maven任务，请执行以下任一操作：

- 双击Maven任务列表中的任务，如“application>run”。
- 在“Maven”视图工具栏上，单击**“执行Maven任务”**按钮（▶）然后在打开的“执行Maven任务”弹窗中选择所需的任务并双击运行。如下图所示：

图 9-12 可执行的 Maven 任务列表



说明

您还可以通过**专用的Maven启动配置**来运行Maven任务。

配置并运行 Gradle 任务

Gradle是一种用于管理Java项目和自动化应用程序构建的工具。CodeArts IDE提供了对Gradle Java项目的内置支持。专用的**Gradle**视图与build.gradle同步，并提供了一个可视化界面，用于查看项目依赖项或运行Gradle任务。如下图所示：

图 9-13 Gradle 视图

说明

在修改**build.gradle**文件后，例如添加新的依赖项，单击“Gradle”视图工具栏上的**“重新加载所有Gradle工程”**按钮（），以使CodeArts IDE应用新的更改。

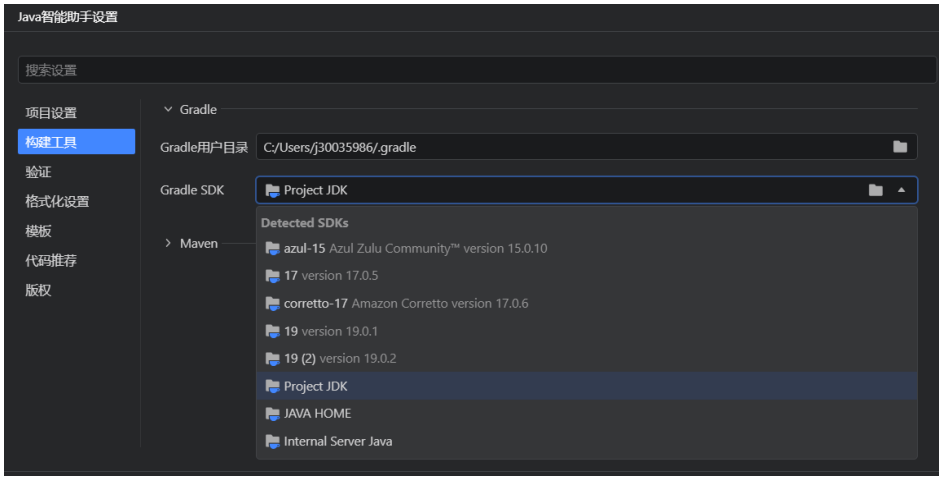
- 自定义Gradle配置选项

  打开Java智能助手设置，切换到**“构建工具”**页签，在**“Gradle”**部分自定义配置选项。

  a. **“Gradle用户目录”**：在此字段中，指定Gradle用户主目录的路径（默认为“$USER_HOME/.gradle”），该目录用于存储全局配置属性和初始化脚本、缓存和日志文件。默认值是基于**GRADLE_USER_HOME**环境变量的值提供的。要修改它，用户可以设置环境变量或单击**“浏览”**按钮（），并手动定位所需的Gradle用户主目录。

  b. **“Gradle SDK”**：从这个列表中选择一个与Gradle一起使用的JDK：捆绑的JDK、本地安装的JDK或从系统变量（如“JAVA_HOME”）解析的JDK。

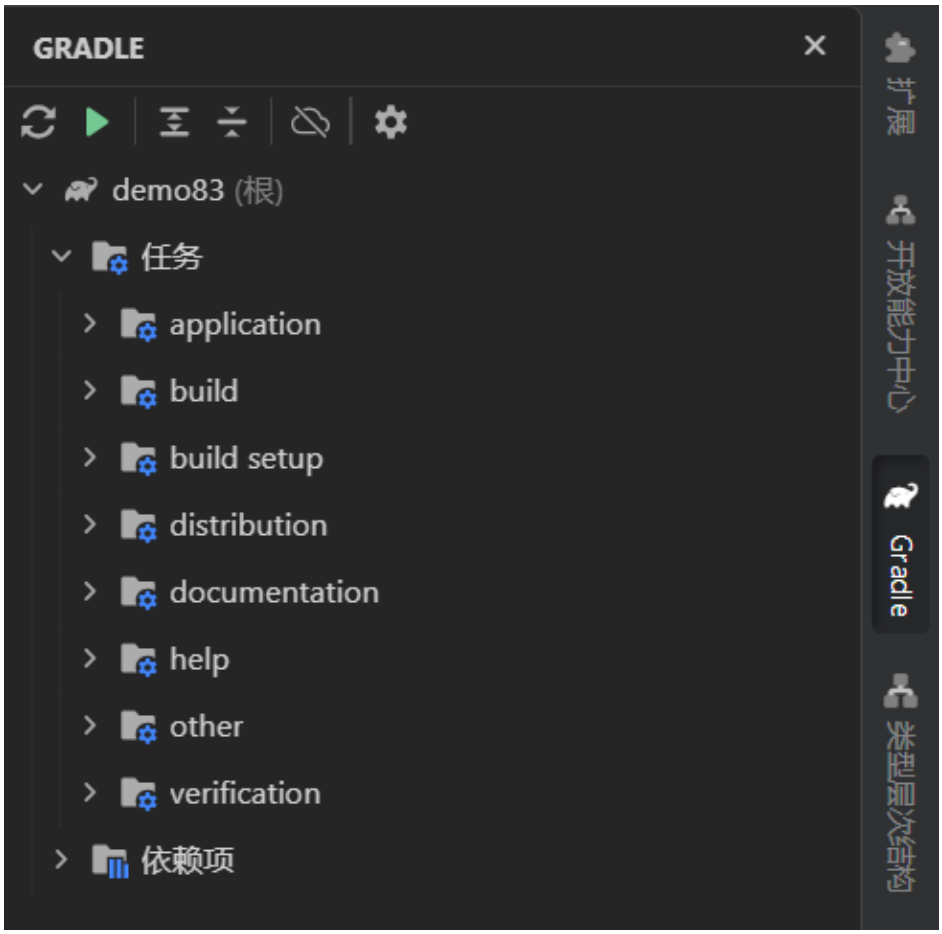
图 9-14 可配置的 JDK 列表示例



- 运行Gradle任务

  在CodeArts IDE中打开一个Gradle项目时，可以在Gradle视图中找到列出的**Gradle**任务。

  图 9-15 Gradle 任务列表

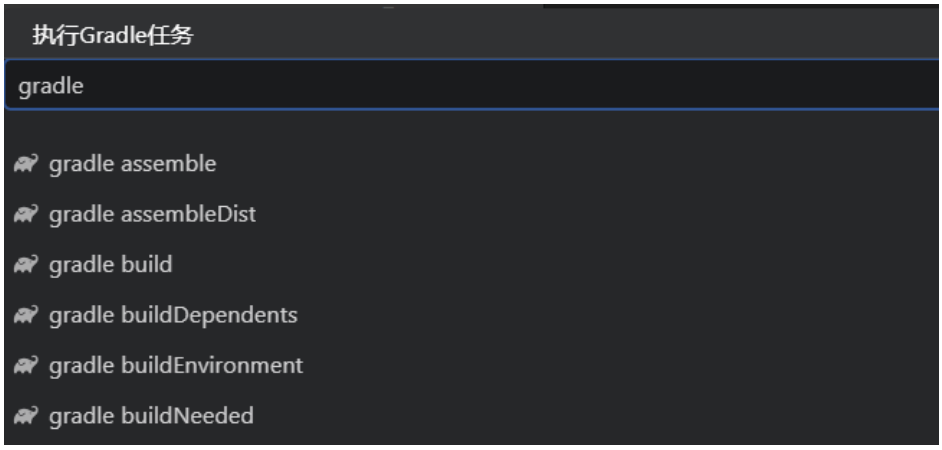
  

  要运行Gradle任务，请执行以下任一操作：

  - 双击Gradle任务列表中的任务，如“application>run”。

- 在“Gradle”视图工具栏上，单击**“执行Gradle任务”**按钮（▶）然后在打开的“执行Gradle任务”弹窗中选择所需的任务并双击运行。

  图 9-16 可执行的 Gradle 任务列表

  说明

  您还可以通过**专用的Gradle启动配置**来运行Gradle任务。

9.3 运行和调试 Java 项目启动配置

9.3.1 启动调试会话

CodeArts IDE允许用户直接从代码编辑器或通过启动配置启动调试会话。

从代码编辑器启动调试会话

如果不打算向程序传递任何参数，则可以直接从代码编辑器启动调试会话。

1. 在具有“main()”方法的类的代码编辑器中，单击或右键“main()”方法左侧**“运行/调试主类”**按钮（▶），
2. 弹出下拉列表，单击“在xxx中运行main方法”或“在xxx中调试main方法”项，**启动配置属性**将自动创建并运行。如下图所示：

   图 9-17 代码编辑区区域“main()”方法处启动调试会话入口

   说明

   创建的启动配置会自动保存，可以随时从CodeArts IDE右上角工具栏上的配置列表中选择它。

从启动配置启动调试会话

启动配置允许用户配置和保存各种调试方案的详细信息。如何使用启动配置的详细信息，请参见**Java启动配置简介**。

1. 从CodeArts IDE右上角工具栏上的配置列表中选择所需的启动配置，执行以下任一操作：
   - 在右上角主菜单中选择“调试>启动调试”，或按“**F5**”或者“**Shift+F9**”。
   - 在调试工具栏上，确保在启动配置列表中选择了所需的启动配置，然后单击“**开始执行(不调试)**”按钮（ ）或者“**开始调试**”按钮（ ）。

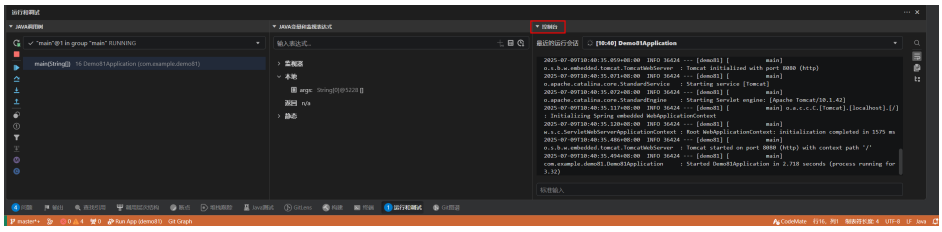
   如下图所示：

   图 9-18 启动配置下拉列表详情

2. 调试会话启动后，将立即显示“运行和调试”面板并在“**控制台**”子视图中显示调试输出。如下图所示：

   图 9-19 启动调试后“运行和调试”面板的“**控制台**”子视图输出示例

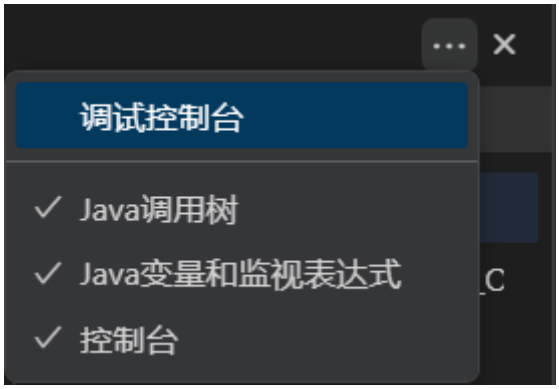
   

9.3.2 运行和调试程序

启动调试会话时，“运行和调试”视图将打开，以显示与运行和调试相关的所有信息。

- 手动打开“运行和调试”视图，单击底部活动栏中的“**运行和调试**”**选项卡**，或按“Ctrl+Shift+D”/“Shift+Alt+F9”/“Alt+5”/“Ctrl+Shift+F8”快捷键。
- 要自定义“运行和调试”视图内容，请单击右上角“**更多操作**”（**...**）然后选中要显示的视图选项。如下图所示：

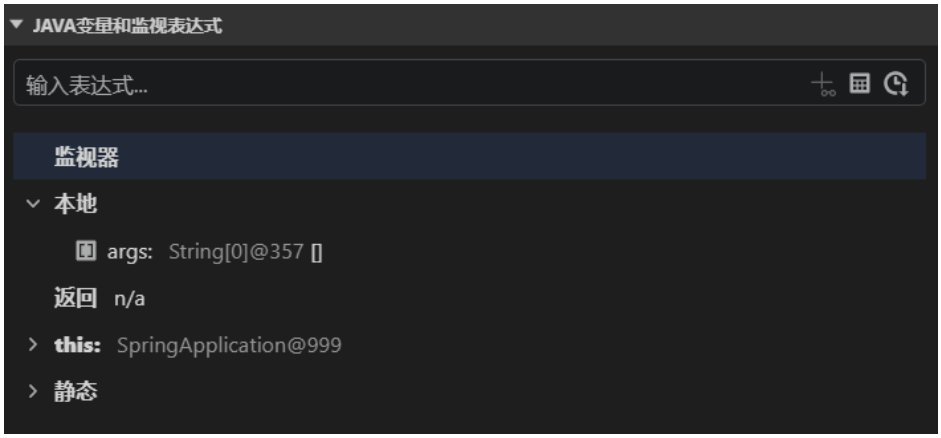
图 9-20 设置运行调试视图中子视图是否展示入口



查看 JAVA 变量和监视表达式

“JAVA变量和监视表达式”部分显示当前堆栈帧（即在“JAVA调用树”部分中选择的堆栈帧）中可访问的元素。如下图所示：

图 9-21 JAVA 变量和监视表达式视图



该视图有以下几部分：

- **“监视器”**：显示进行“将表达式添加到监视”操作之后的表达式和表达式的值。如下图所示：

图 9-22 输入相关表达式并添加到监视示例

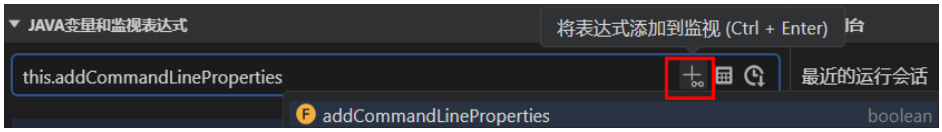
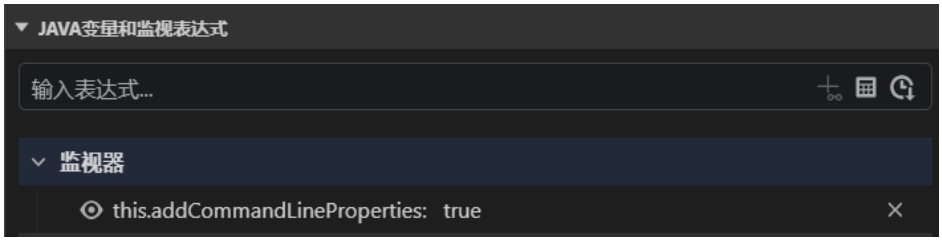


图 9-23 监视器显示已添加的表达式



- **“本地”**：显示被调用方法作用域内的局部变量。
- “返回 n/a”：当一个方法在调试会话期间被多次调用时，本节显示该方法在上一步返回的值。这使用户可以观察值在方法调用之间的变化。
- **“this”**：显示正在调用其方法的对象的内容。
- “静态”：列出静态类字段。

用户可以在变量上通过右键唤出上下文菜单，使用**“设置值”**操作来修改变量的值。此外，用户可以使用**“复制值”**操作复制变量的值。

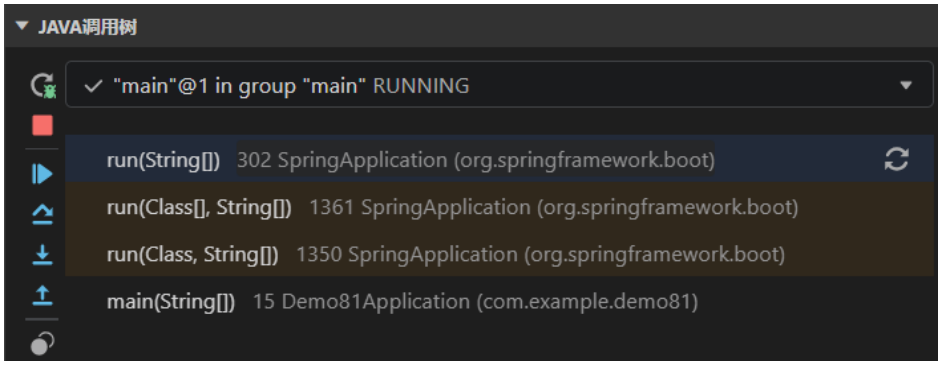
说明

用户还可以直接在CodeArts IDE代码编辑器中查看变量或表达式的值。为此，请在挂起的调试程序中将鼠标悬停在所需的变量、表达式上。

查看 JAVA 调用树

“JAVA调用树”部分列出了当前活动的堆栈帧，方法的调用堆栈分组在每个帧下。如下图所示：

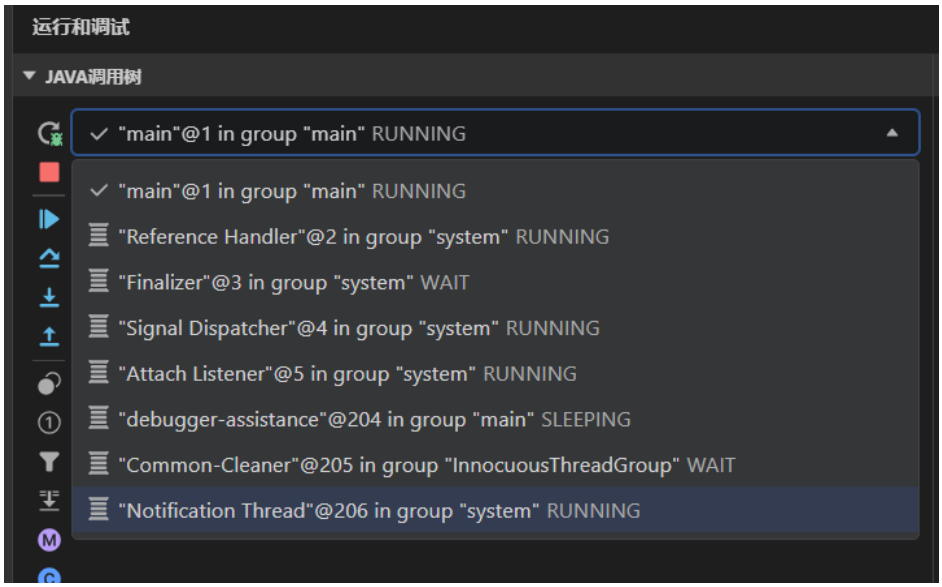
图 9-24 默认展示的主线程对应的堆栈帧



当前堆栈帧内可访问的元素列在**“JAVA变量和监视表达式”**部分中。

- 默认展示主线程的堆栈帧。要切换到其他线程，请在**“JAVA调用树”**的下拉框中选择对应的线程。如下图所示：

图 9-25 切换不同线程的堆栈帧入口



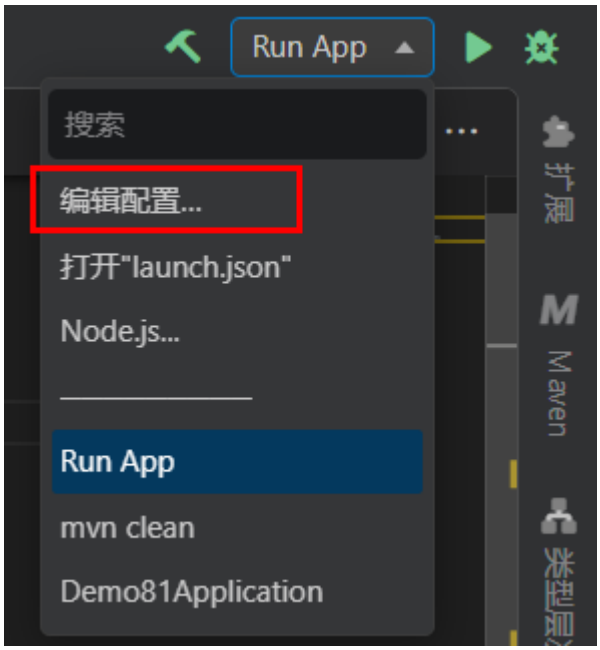
- 要快速导航到代码编辑器中的方法调用处，请单击**“JAVA调用树”**中的堆栈帧。

9.3.3 Java 启动配置使用简介

启动配置允许用户配置和保存各种场景的运行或调试设置详细信息。CodeArts IDE支持用户可视化的新增/修改启动配置，并将配置信息保存在项目根路径下“.arts”文件夹中的“launch.json”文件内。

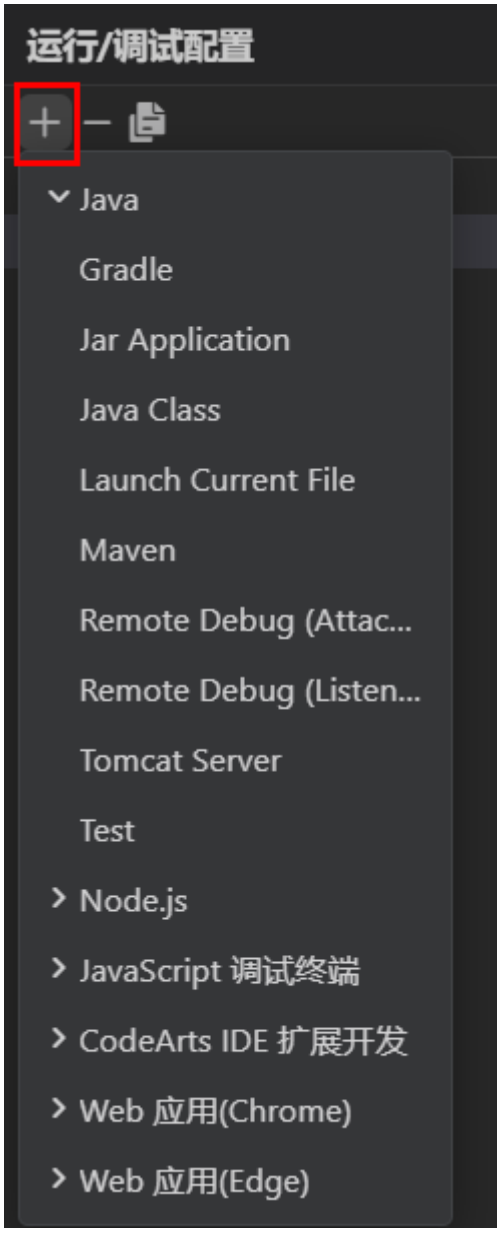
可视化的新增/修改启动配置

步骤1 在CodeArts IDE右上角工具栏上的配置列表中，选择**“编辑配置...”**，如下图所示。

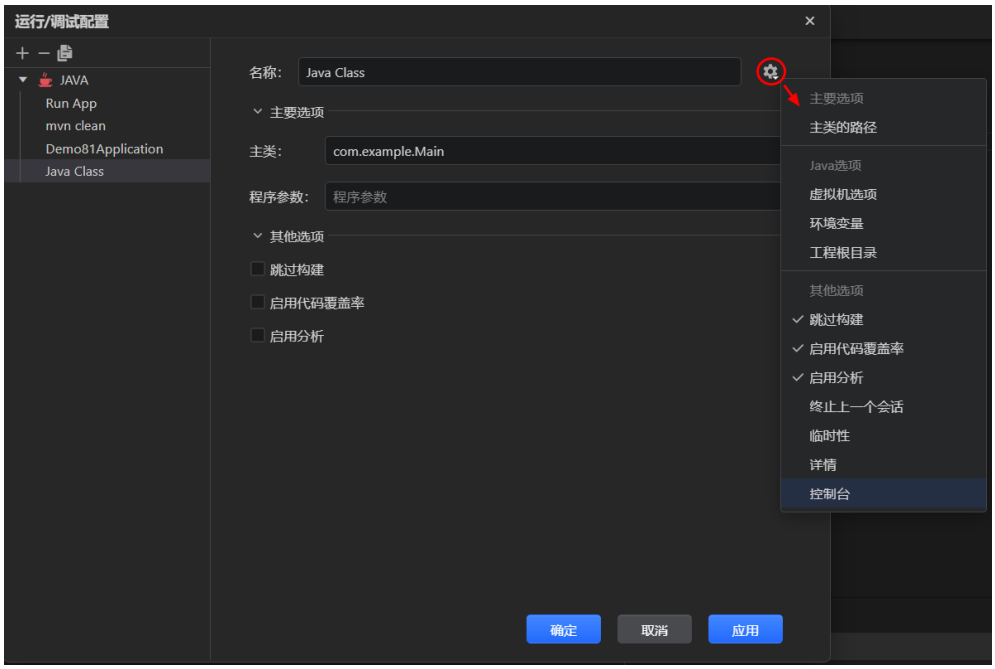


步骤2 在打开的“运行/调试配置”中，单击左侧工具栏的**“新增配置项”**（+）按钮，选择并添加对应类型的配置项。

图 9-26 可添加的 Java 启动配置示例



步骤3 在新增Java配置项（如Java Class）的编辑界面中，右侧的**“配置选项”**按钮可以选择更多的参数进行配置。



步骤4 单击“运行/调试配置”窗口底部的**“应用”**按钮可以保存当前的配置，并且可以在“运行/调试配置”窗口中继续添加新的配置。单击**“确定”**按钮会保存当前的配置，并关闭“运行/调试配置”窗口，单击**“取消”**按钮会取消本次的编辑并关闭“运行/调试配置”窗口。

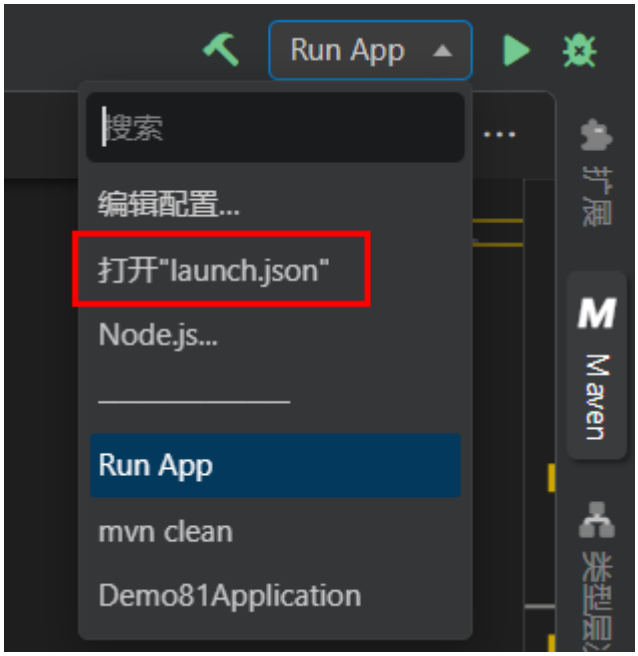
----结束

向现有的 launch.json 添加新的配置

由于可视化新增/编辑的启动配置和“launch.json”文件中的配置是一一对应的，用户也可以直接操作launch.json文件，如在launch.json文件中添加新的配置。

要打开“launch.json”文件，请在CodeArts IDE右上角工具栏上的列表中选择**“打开launch.json”**，如下图所示。

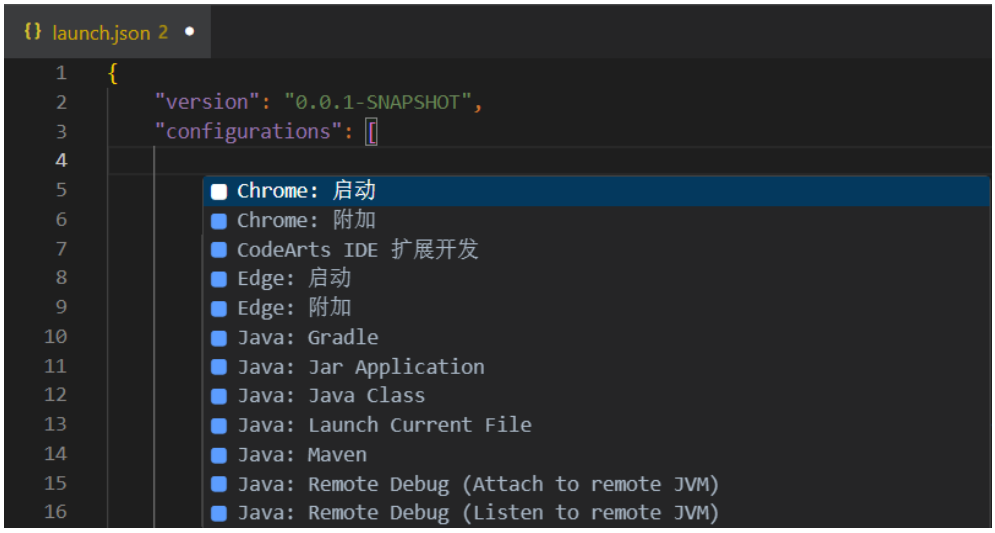
图 9-27 打开 launch.json 文件



请按如下步骤向现有的launch.json文件中添加新的配置：

步骤1 在打开的“launch.json”文件的编辑器中，单击编辑器右下角的 **“添加配置...”** 按钮，或将光标放置在“configurations”数组内，并使用代码推荐快捷键快速唤出可添加的配置列表（“Ctrl+I” / “Ctrl+Space” / “Ctrl+Shift+Space”（IDEA键盘映射））。

图 9-28 可添加的配置列表示例



步骤2 在弹出的配置列表中，选择要使用的启动配置模板。

在“launch.json”中，使用代码推荐快捷键（“Ctrl+I” / “Ctrl+Space” / “Ctrl+Shift+Space”（IDEA键盘映射））查看可用属性及其值的列表。

步骤3 配置完成后，右上角工具栏上的列表中可以选中新增的启动配置，然后单击**“开始执行(不调试)”**按钮（ ）或者**“开始调试”**按钮（ ）使用新增的启动配置发起运行或调试。

----结束

变量动态替换

CodeArts IDE将常用路径和其他值作为变量提供，并支持对“launch.json”中的字符串中进行变量动态替换，因此您不必在启动配置中使用绝对路径。

支持以下预定义变量：

- **${cwd}** - CodeArts IDE启动时任务运行器的当前工作目录。
- **${defaultBuildTask}** - 默认构建任务的名称。
- **${extensionInstallFolder}** - 指定扩展安装的路径。
- **${fileBasenameNoExtension}** - 当前打开文件的无扩展名的基本名称。
- **${fileBasename}** - 当前打开文件的基本名称。
- **${fileDirname}** - 当前打开文件的目录名。
- **${fileExtname}** - 当前打开文件的扩展名。
- **${file}** - 当前打开的文件。
- **${lineNumber}** - 活动文件中当前选定的行号。
- **${pathSeparator}** - 操作系统用于分隔文件路径组件的字符。
- **${relativeFileDirname}** - 相对于**workspaceFolder**的当前打开文件的目录名。
- **${relativeFile}** - 相对于**workspaceFolder**的当前打开文件。
- **${selectedText}** - 活动文件中当前选定的文本。
- **${workspaceFolderBasename}** - 在CodeArts IDE中打开的文件夹的名称，不包含任何斜杠（/）。
- **${workspaceFolder}** - 在CodeArts IDE中打开的文件夹的路径。

9.3.4 运行 Java 类的启动配置

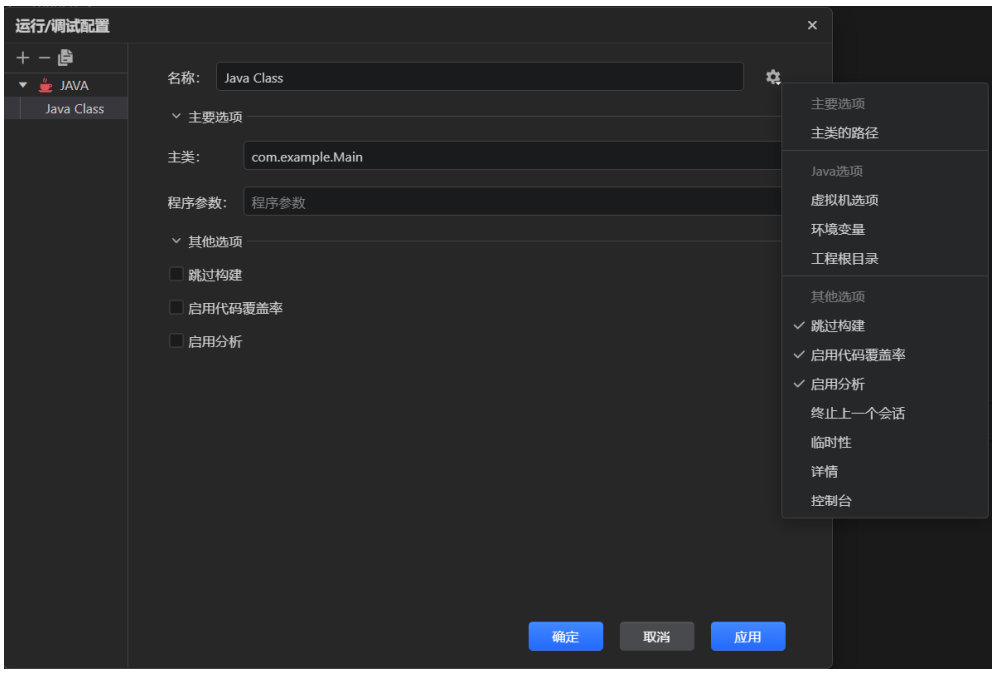
使用以下启动配置来运行应用程序的主类（包含“main()”方法的类）。

说明

要快速运行一个应用程序而不必手动创建一个启动配置，可参考**从代码编辑器启动调试会话**。

启动配置属性

图 9-29 Java 类启动配置示例



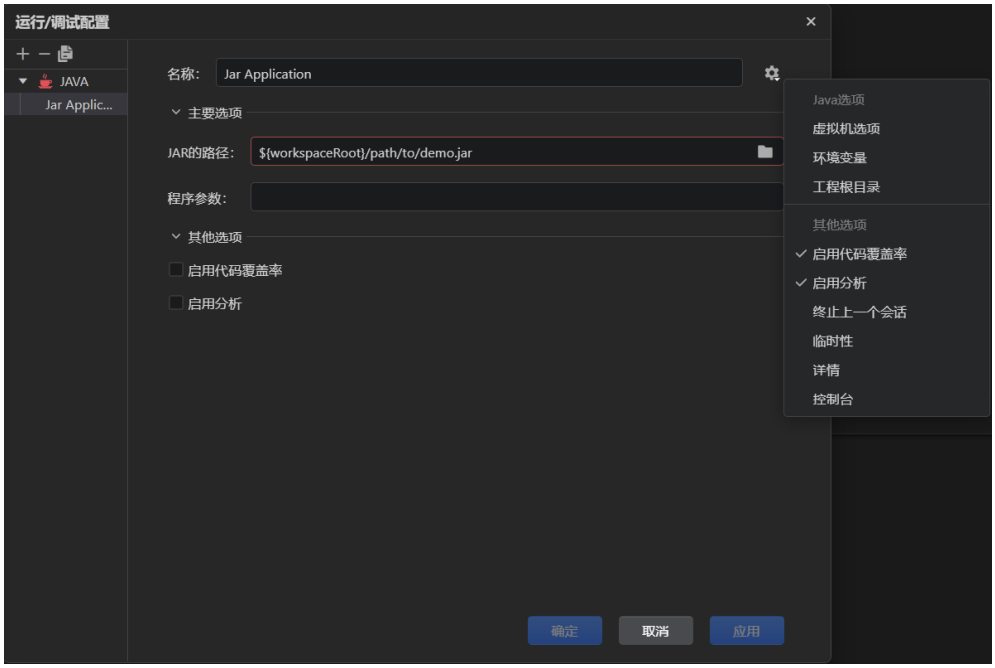
配置项	描述
主类	包含“main()”方法的类。
程序参数	传递给“main()”方法的参数。
跳过构建	跳过程序的构建过程。
启用代码覆盖率	使用代码覆盖率功能运行任务。
启用分析	启用用于Java堆内存使用情况的JMX分析。
虚拟机选项	JVM的额外选项。
环境变量	额外的环境变量。
终止上一个会话	终止具有相同名称的先前运行会话（勾选此选项），或中止启动（不勾选此选项）。
临时性	将配置标记为临时。默认情况下，CodeArts IDE根据指定的限制（默认为10）保留临时启动配置的数量，并在超过限制时自动删除最少使用的配置。如果需要，您可以通过“java.executedConfigurationsLimit”设置来调整此限制。
详情	将附加信息打印到调试控制台，如调试程序启动的命令行信息。
控制台	在“运行和调试”面板的“控制台”子视图（**internal-console**）或**集成终端**（**integrated-terminal**）中显示调试输出。

9.3.5 运行 JAR 应用的启动配置

使用此启动配置来运行打包在JAR文件中的应用程序。

启动配置属性

图 9-30 JAR 应用的启动配置



配置项	描述
JAR的路径	Jar包的路径。
程序参数	传递给“main()”方法的参数。
启用代码覆盖率	使用代码覆盖率功能运行任务。
启用分析	启用用于Java堆内存使用情况的JMX分析。
虚拟机选项	JVM的额外选项。
环境变量	额外的环境变量。
终止上一个会话	终止具有相同名称的先前运行会话（勾选此选项），或中止启动（不勾选此选项）。
临时性	将配置标记为临时。默认情况下，CodeArts IDE根据指定的限制（默认为10）保留临时启动配置的数量，并在超过限制时自动删除最少使用的配置。如果需要，您可以通过“java.executedConfigurationsLimit”设置来调整此限制。
详情	将附加信息打印到调试控制台，如调试程序启动的命令行信息。

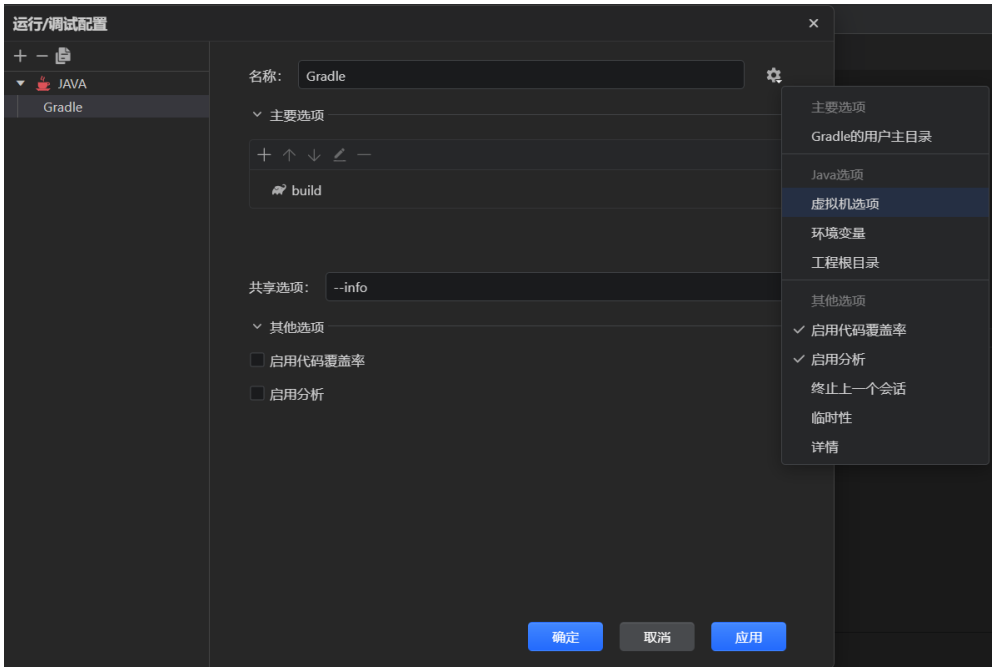
配置项	描述
控制台	在 “运行和调试”面板的“控制台”子视图（**internal-console**）或**集成终端**（**integrated-terminal**）中显示调试输出。

9.3.6 运行 Gradle 任务的启动配置

使用此启动配置来运行一个或多个Gradle任务。

启动配置属性

图 9-31 Gradle 任务的启动配置



配置项	描述
主要选项	要运行的Gradle任务。如“build”任务，可添加多个Gradle任务。
共享选项	传递给Gradle的参数， 例如 '--info' 或 '--debug'。
启用代码覆盖率	使用代码覆盖率功能运行任务。
启用分析	启用用于Java堆内存使用情况的JMX分析。
Gradle的用户主目录	覆盖Gradle的用户主目录的默认值。
虚拟机选项	JVM的额外选项。Gradle的情况下，这些选项将传递给Gradle的虚拟机。
环境变量	额外的环境变量。

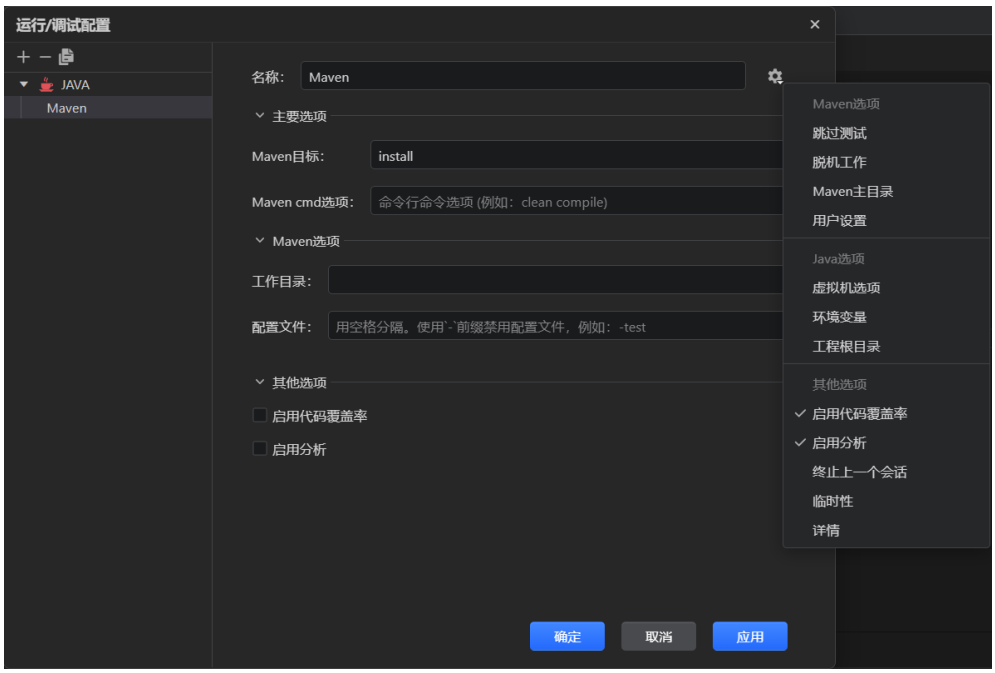
配置项	描述
终止上一个会话	终止具有相同名称的先前运行会话（勾选此选项），或中止启动（不勾选此选项）。
临时性	将配置标记为临时。默认情况下，CodeArts IDE根据指定的限制（默认为10）保留临时启动配置的数量，并在超过限制时自动删除最少使用的配置。如果需要，您可以通过“java.executedConfigurationsLimit”设置来调整此限制。
详情	将附加信息打印到调试控制台，如调试程序启动的命令行信息。
控制台	在“运行和调试”面板的“控制台”子视图（**internal-console**）或**集成终端**（**integrated-terminal**）中显示调试输出。

9.3.7 运行 Maven 任务的启动配置

使用此启动配置来运行一个或多个Maven任务。

启动配置属性

图 9-32 Maven 任务的启动配置



配置项	描述
Maven目标	要运行的Maven目标任务。
Maven cmd选项	要附加到maven调用命令的其他命令选项列表。（例如clean compile）。

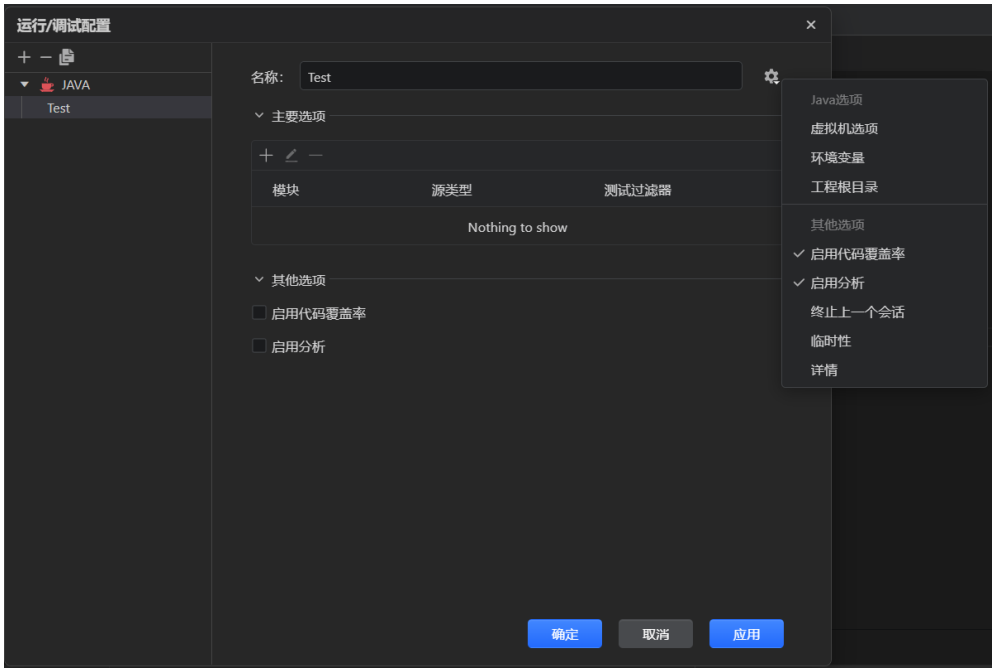
配置项	描述
工作目录	调用maven时的工作目录路径（例如项目的根目录）。
配置文件	要使用的配置文件列表。使用“-”前缀禁用配置文件，例如：-test。
启用代码覆盖率	使用代码覆盖率功能运行任务。
启用分析	启用用于Java堆内存使用情况的JMX分析。
跳过测试	运行Maven任务的时候跳过测试。
脱机工作	在执行maven任务时设置 --offline。
Maven主目录	Maven安装路径。覆盖MAVEN_HOME环境变量。
用户设置	Maven调用的用户设置文件路径。
虚拟机选项	JVM的额外选项。Maven的情况下，这些选项将传递给Maven的虚拟机。
环境变量	额外的环境变量。
终止上一个会话	终止具有相同名称的先前运行会话（勾选此选项），或中止启动（不勾选此选项）。
临时性	将配置标记为临时。默认情况下，CodeArts IDE根据指定的限制（默认为10）保留临时启动配置的数量，并在超过限制时自动删除最少使用的配置。如果需要，您可以通过“java.executedConfigurationsLimit”设置来调整此限制。
详情	将附加信息打印到调试控制台，如调试程序启动的命令行信息。
控制台	在 “运行和调试”面板的“控制台”子视图（**internal-console**）或**集成终端**（**integrated-terminal**）中显示调试输出。

9.3.8 运行 JUnit 测试的启动配置

使用此启动配置来运行JUnit测试。

启动配置属性

图 9-33 JUnit 测试的启动配置



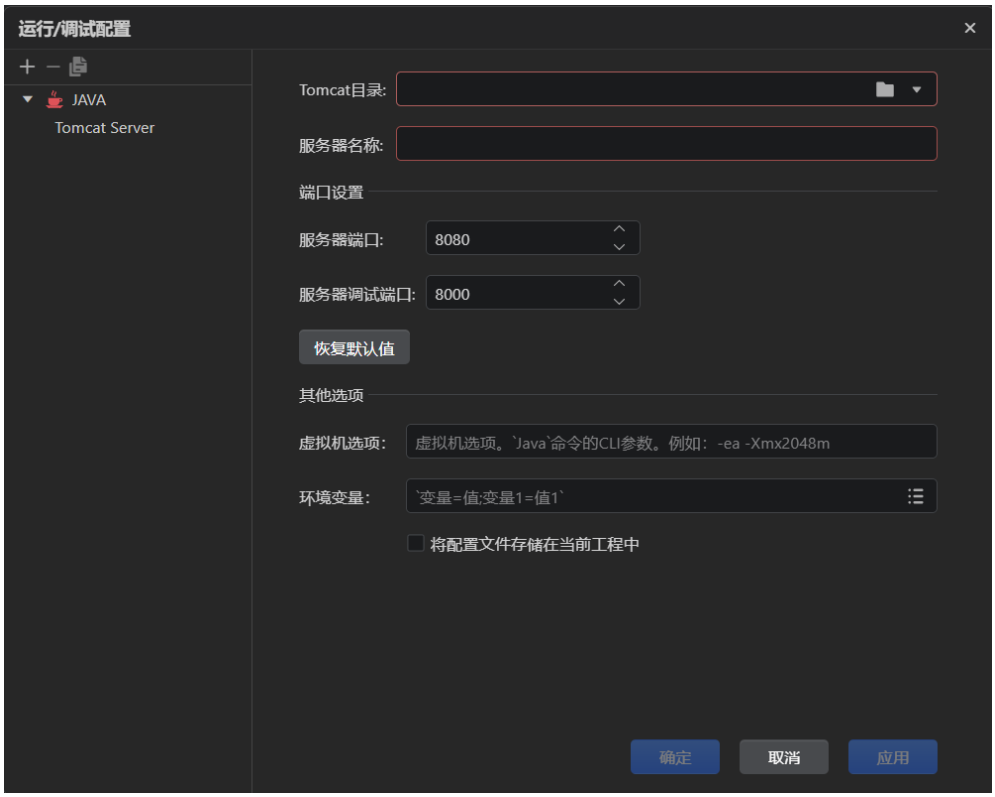
配置项	描述
主要选项	测试过滤器，过滤要执行的测试用例的模块和类型。
启用代码覆盖率	使用代码覆盖率功能运行任务。
启用分析	启用用于Java堆内存使用情况的JMX分析。
虚拟机选项	JVM的额外选项。
环境变量	额外的环境变量。
终止上一个会话	终止具有相同名称的先前运行会话（勾选此选项），或中止启动（不勾选此选项）。
临时性	将配置标记为临时。默认情况下，CodeArts IDE根据指定的限制（默认为10）保留临时启动配置的数量，并在超过限制时自动删除最少使用的配置。如果需要，您可以通过“java.executedConfigurationsLimit”设置来调整此限制。
详情	将附加信息打印到调试控制台，如调试程序启动的命令行信息。
控制台	在 “运行和调试”面板的“控制台”子视图（**internal-console**）或**集成终端**（**integrated-terminal**）中显示调试输出。

9.3.9 运行 Tomcat 服务的启动配置

使用此启动配置来运行Tomcat服务。

启动配置属性

图 9-34 Tomcat 服务启动配置



配置项	描述
Tomcat目录	Tomcat的安装目录。
服务器名称	自定义Tomcat服务器的名称。
服务器端口	Tomcat服务器的启动端口，默认是8080。
服务器调试端口	Tomcat服务器的调试端口。
虚拟机选项	JVM的额外选项。
环境变量	额外的环境变量。
将配置文件存储在当前工程中	将Tomcat配置文件存储在工程中。

9.3.10 调试远程连接的启动配置

使用此启动配置，可以通过连接到远程JVM或使调试器监测传入连接，来进行远程调试。

启动配置属性

远程调试模式，分为“Attach”模式（连接到远程JVM）和“Listen”模式（监听传入连接）。默认情况下，使用“Attach”模式，如下图所示。

图 9-35 远程调试启动配置：“Attach”模式

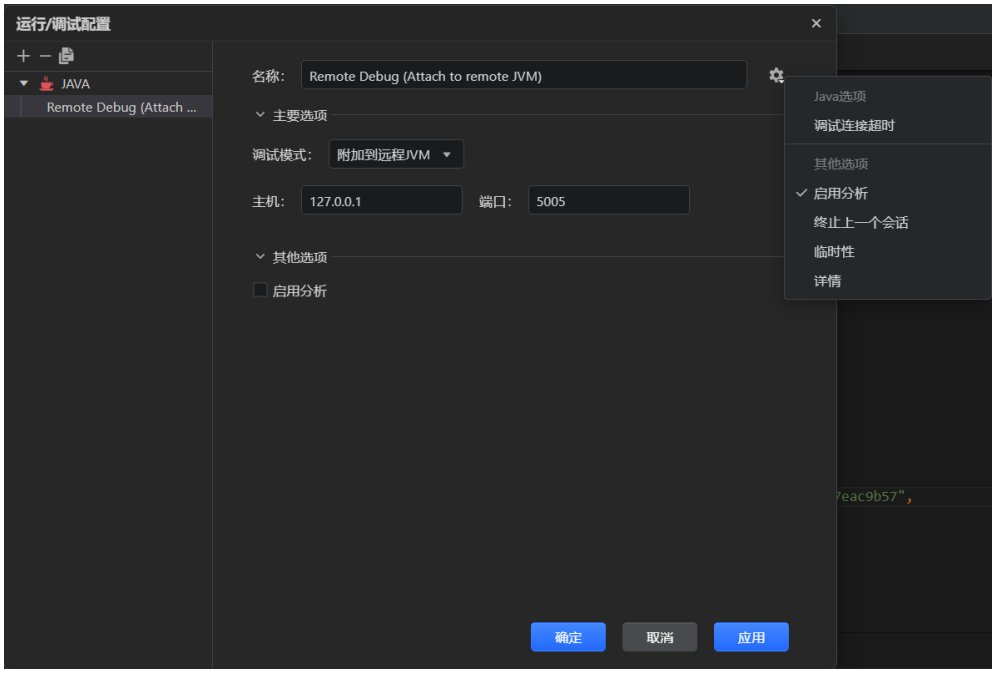
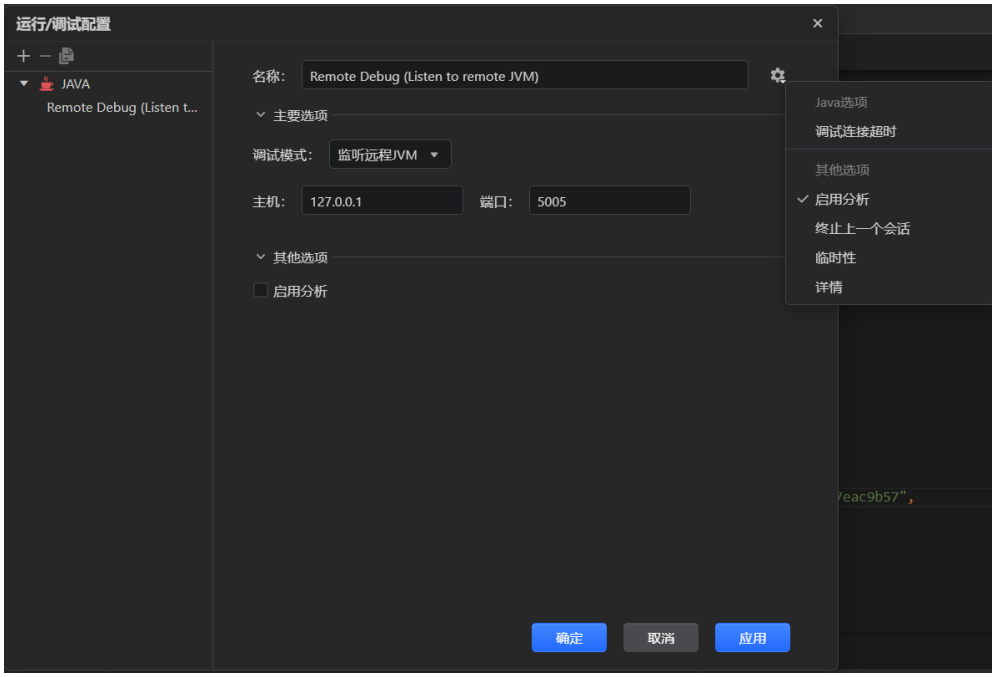


图 9-36 远程调试启动配置：“Listen”模式



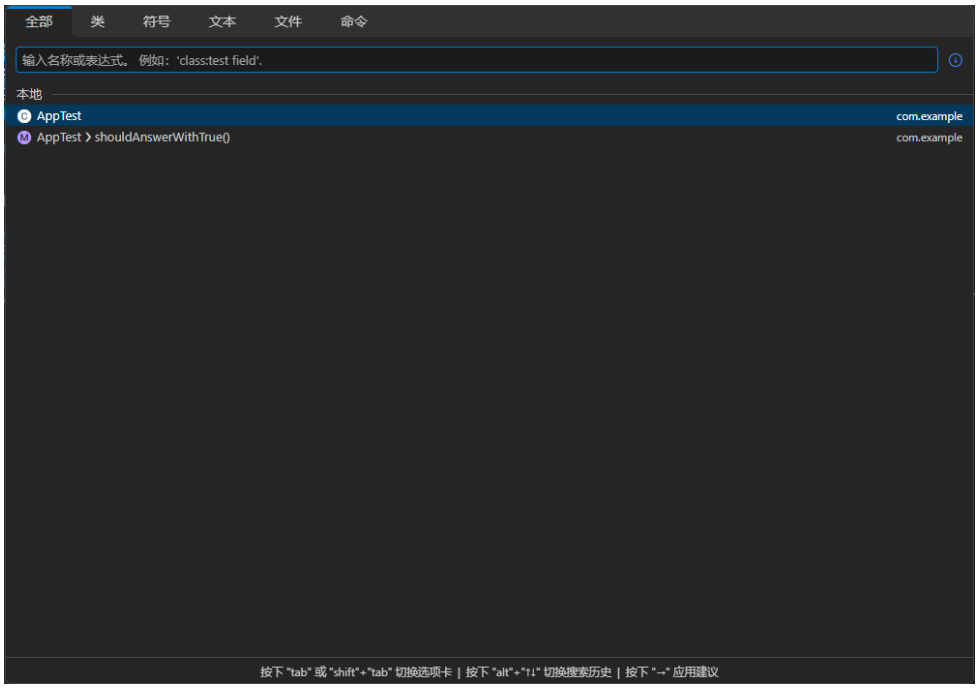
配置项	描述
调试模式	通过“Remote Debug (Attach to remote JVM)”类型创建的启动配置默认选中“附件到远程JVM”（即Attach模式）；通过“Remote Debug (Listen to remote JVM)”类型创建的启动配置默认选中“监听远程JVM”（即Listen模式）。
主机	主机应用程序运行的机器地址。
端口	远程连接的端口。
启用分析	启用用于Java堆内存使用情况的JMX分析。
调试连接超时	调试器连接的等待时间，以毫秒为单位。
终止上一个会话	终止具有相同名称的先前运行会话（勾选此选项），或中止启动（不勾选此选项）。
临时性	将配置标记为临时。默认情况下，CodeArts IDE根据指定的限制（默认为10）保留临时启动配置的数量，并在超过限制时自动删除最少使用的配置。如果需要，您可以通过“java.executedConfigurationsLimit”设置来调整此限制。
详情	将附加信息打印到调试控制台，如调试程序启动的命令行信息。

9.4 使用 Java 进行智能搜索

9.4.1 智能搜索基本用法

步骤1　可选: 使用快捷键“Ctrl+Shift+A”/“Shift Shift”启动**“智能搜索”**窗口。

图 9-37 智能搜索窗口

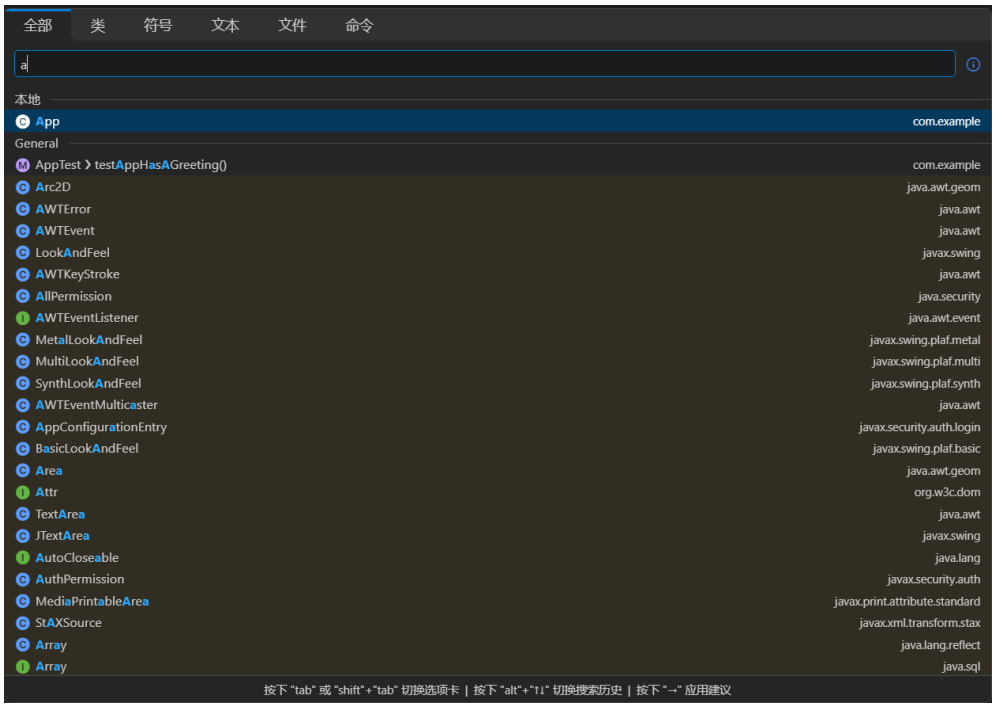


如果在代码编辑器中打开了一个Java文件，**“智能搜索”**窗口将自动显示其大纲，用户可以在代码条目（例如类成员）之间导航。

步骤2 输入关键字即可进行搜索。为了缩小搜索范围，可以在**“智能搜索”**窗口中切换选项卡，或参考**搜索查询语法**使用语法进行精确搜索。

步骤3 在**“智能搜索”**窗口中，打开文件的结果显示在**“本地”**区域；项目中其他位置的结果显示在**“General”**区域。

图 9-38 在**“智能搜索”**窗口中查看信息



步骤4 使用光标键在条目之间导航，按“Enter”键转到相应位置或执行命令，或者单击所需的条目。要关闭**“智能搜索”**窗口，按“Escape”键，或单击**“智能搜索”**窗口以外的区域。

----结束

9.4.2 搜索查询语法和运算符

搜索查询语法

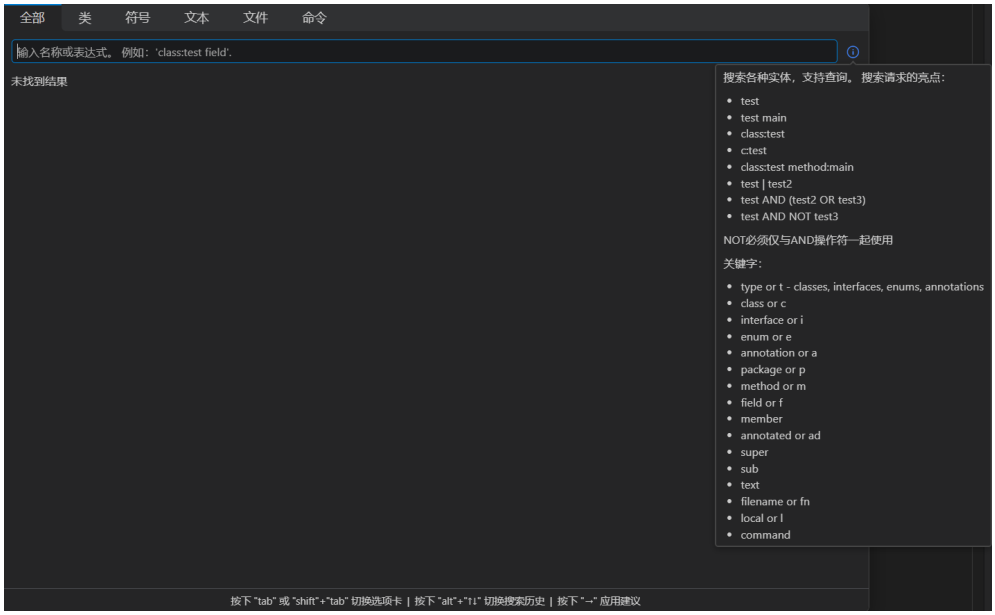
搜索查询是由**dataSource:stringToMatch**对组成的字符串，可以通过空格或运算符连接。如果查询中省略了**dataSource**，将在所有可用的数据源中进行搜索。也可以使用反向模式，即**stringToMatch:dataSource**。以下是可用数据源的列表。

表 9-1 数据源列表

数据源名称	数据源简码	描述
local	l	当前文件实体
class	c	类实体
interface	i	接口实体
enum	e	枚举实体
annotation	a	注解实体
annotated	ad	带注解实体
method	m	方法实体
field	f	字段实体
super	--	超类/接口实体
sub	--	子类/接口实体
type	--	类型化实体，即类、接口、枚举或注解实体
member	--	成员实体，即类方法或类字段实体
text	--	文本实体。只有文本文件会被文本搜索处理，jar文件会被忽略。
command	--	CodeArts IDE命令实体

要快速了解智能搜索查询语法，请将鼠标悬浮到“智能搜索”窗口右上角的按钮（ ）。

图 9-39 智能搜索查询语法



搜索运算符

通过使用**AND**和**OR**运算符，或它们的组合，来构建复杂的搜索查询，例如**class:foo AND(method:bar OR method:baz)**。

表 9-2 运算符列表

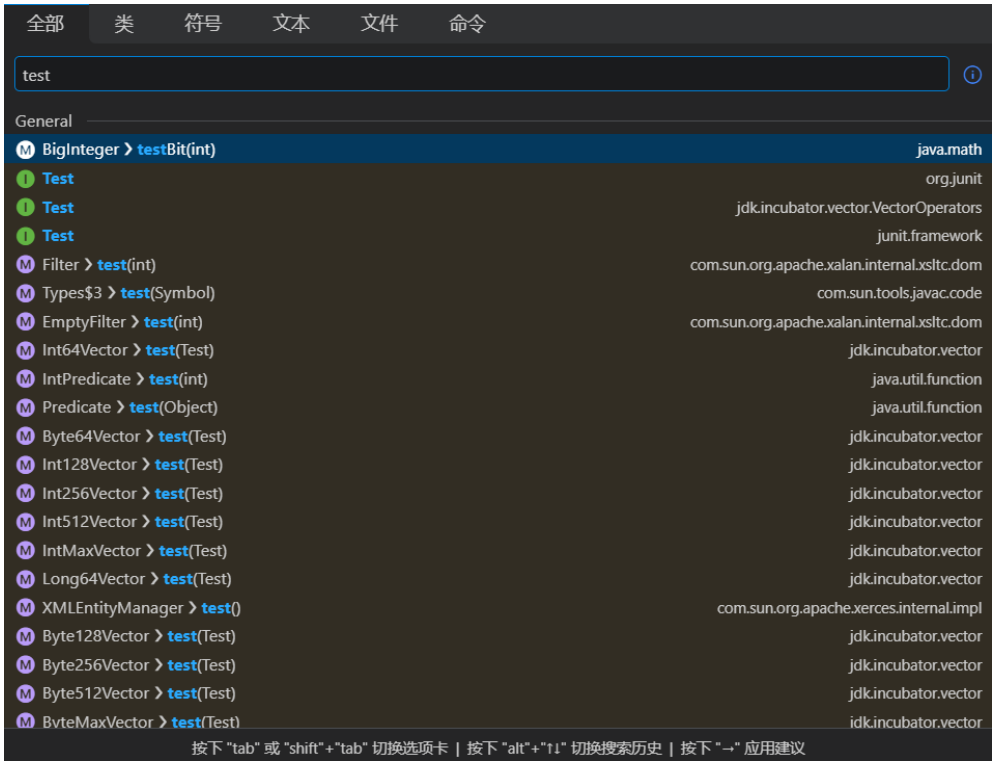
运算符	语法	描述
AND	AND，&，&&，(space character)	**智能搜索**将定位与每个查询匹配的条目，并仅返回与彼此相关的条目。
OR	OR，|，||	**智能搜索**将返回与任何提供的查询匹配的所有条目。

9.4.3 定位代码

定位任意实体

搜索查询**test**将匹配所有名称中带有**test**的实体，如下图所示。

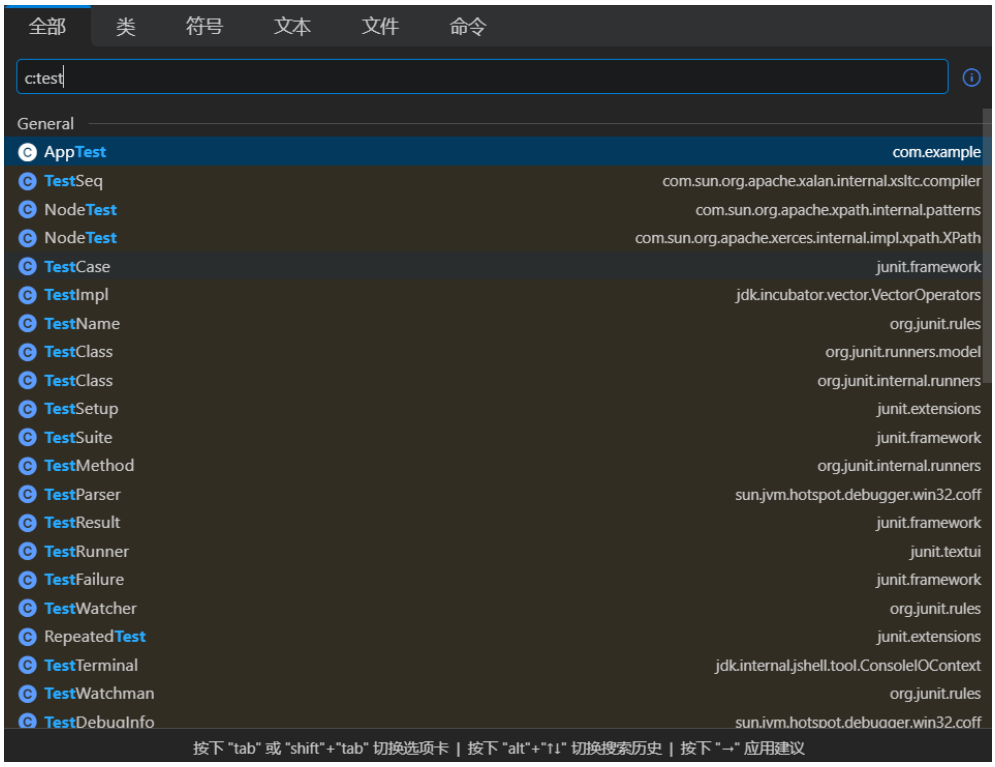
图 9-40 定位任意实体



定位类

搜索查询**class:test**将匹配所有名称中包含**test**的类。使用替代语法，这个查询也可以写作**c:test**、**test:class**或**test:c**，如下图所示。

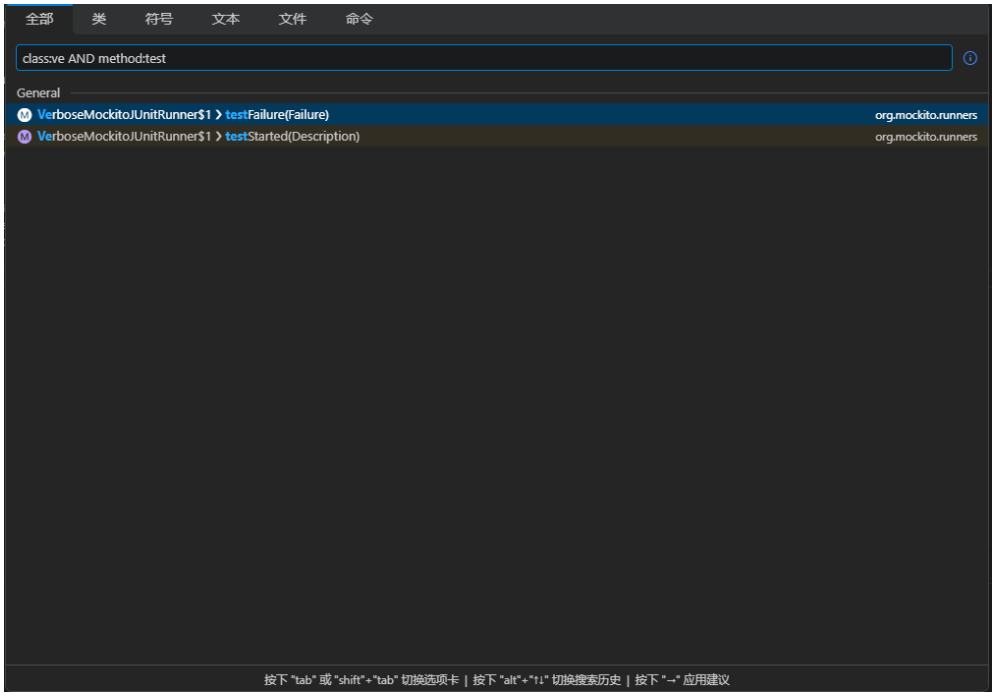
图 9-41 定位类



定位类中的方法

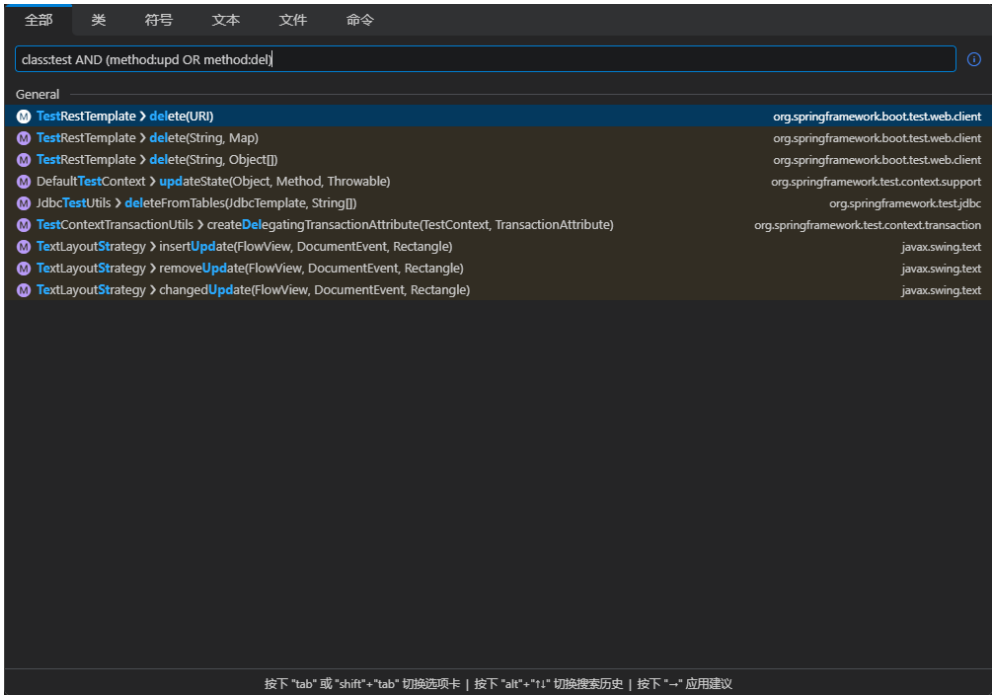
搜索查询**class:vet AND method:test**匹配所有名称中带有**test**的方法，并且名称中带有**vet**的类，如下图所示。

图 9-42 搜索查询 **class:vet AND method:test**



搜索查询**class:test AND (method:upd OR method:del)**将匹配所有名称中带有**upd**或**del**的方法，并且名称中带有**test**的类，如下图所示。

图 9-43 精确定位类中的方法



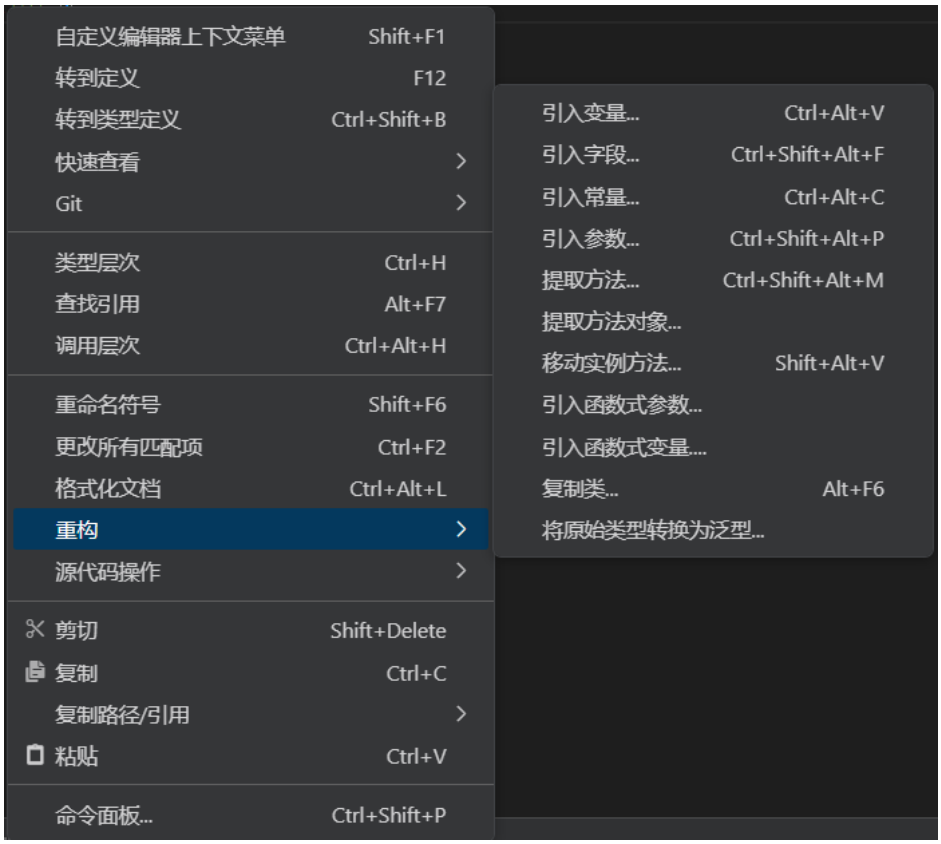
9.5 使用 Java 重构代码

9.5.1 Java 重构代码简介

Java代码重构的目标是在不影响程序行为的情况下进行系统范围的代码更改。SmartAssist扩展提供了许多易于访问的重构选项。

重构命令存在于编辑器的右键菜单中。选择用户要重构的元素，单击鼠标右键，然后从上下文菜单中选择“**重构**”。或者在主菜单中，选择“**重构**”，如下图所示。

图 9-44 重构菜单



Java代码重构包含多种重构方式，其中：

- **移动重构**支持在不同的包中创建类的副本，在整个类层次结构中移动类和类成员。
- 提取/引入重构允许将任意代码表达式转换为新变量、类字段、方法等，这与**内联重构**相反。
- 内联重构允许将变量、字段、方法等用法替换为其内容，这与**提取/引入重构**相反。

9.5.2 移动重构

9.5.2.1 复制类

此重构支持在不同的包中创建类的副本，维护正确的目录结构。

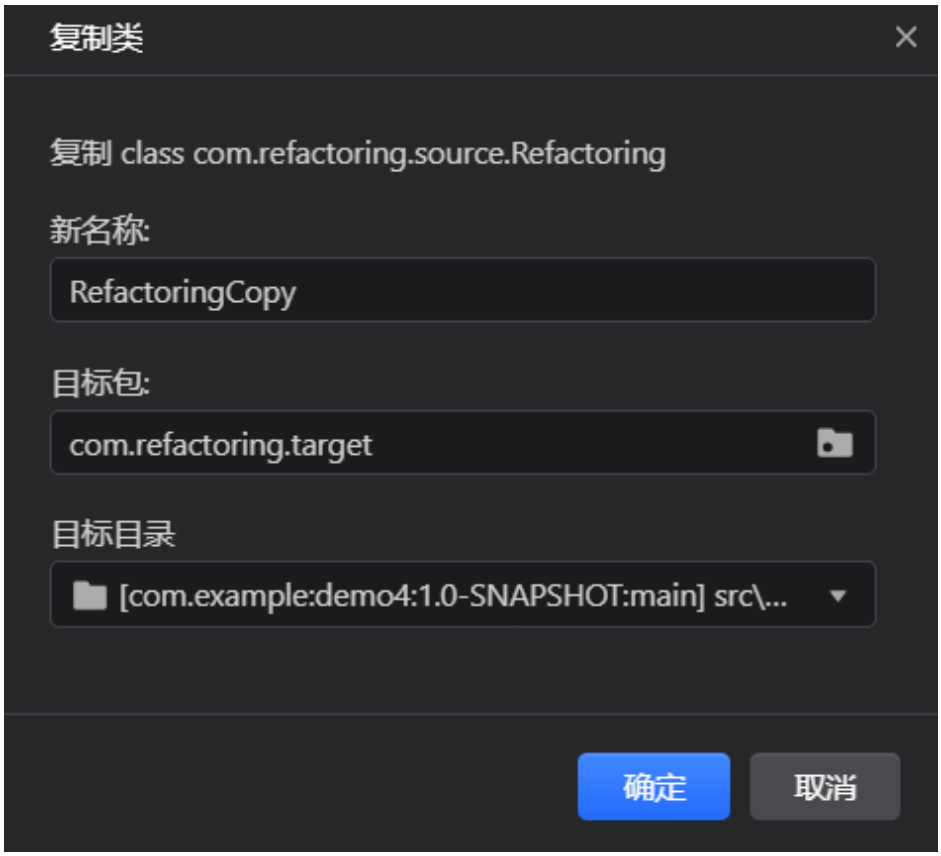
执行重构

步骤1 在代码编辑器中，将光标放置在要复制的类中的任何位置，单击右键。

步骤2 单击右键展示上下文菜单，选择“重构 > 复制类...” 或按“**Alt+F6**”或“**F5**”(仅适用于IDEA快捷键方案)。

步骤3 在打开的“复制类”对话框中，提供重构参数。如下图所示：

图 9-45 复制类



步骤4 单击“**确定**”以应用重构。

----结束

示例

作为示例，将创建一个位于com.refactoring.source包中的Refactoring类的副本，并将该副本类RefactoringCopy存储在com.refactoring.target包中。

重构前

“com\refactoring\source\Refactoring.java” 文件内容如下：

```
package com.refactoring.source;

public class Refactoring {
    public String testStr = "test";
    public void DoSomeThing() {
        System.out.println(testStr);
    }
}
```

重构后

“com\refactoring\source\Refactoring.java” 文件不变，新增了 “com\refactoring\target\RefactoringCopy.java” 文件，文件内容如下：

```
package com.refactoring.target;

public class RefactoringCopy {
    public String testStr = "test";
```

```
    public void DoSomeThing() {
        System.out.println(testStr);
    }
}
```

9.5.2.2 移动类

此重构允许移动不同包中的类，维护正确的目录结构。

执行重构

步骤1 在代码编辑器中，将光标放在用户想要移动的类上。

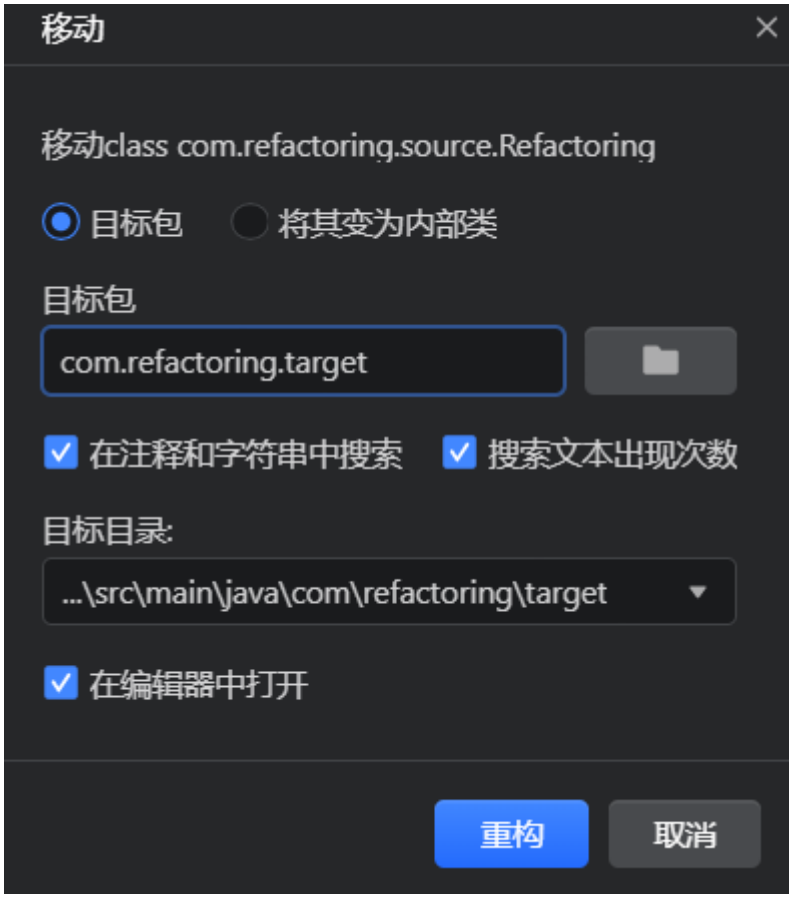
步骤2 单击右键展示上下文菜单，选择“重构 > 移动类...”或按“F6”。

步骤3 在打开的“移动”对话框中，提供重构参数。

- 要将类移动到不同的包中，请选择**“目标包”**并在**“目标包”**选择框中选择目标包。单击浏览按钮（ ），在打开的“选择目标包”对话框中，选择包或创建一个新包。
- 要将类移动到其他类中，使其成为内部类，请选择**“将其变为内部类”**并在**“将其变为内部类”**输入框中输入目标类的完全限定名称。
- 要在代码中搜索移动的类的出现情况，请勾选**“在注释和字符串中搜索”**和**“搜索文本出现次数”**复选框。

如下图所示：

图 9-46 移动类



步骤4 单击“**重构**”以应用重构。

----结束

示例

作为示例，将存储在包com.refactoring.source中的类Refactoring移动到包com.refactoring.target中。

重构前

“com\refactoring\source\Refactoring.java” 文件内容如下：

```
package com.refactoring.source;

public class Refactoring {
    public String testStr = "test";
    public void DoSomeThing() {
        System.out.println(testStr);
    }
}
```

重构后

“com\refactoring\source\Refactoring.java” 文件被删除，重新创建 “com\refactoring\target\Refactoring.java” 文件， 内容如下：

```
package com.refactoring.target;

public class Refactoring {
   public String testStr = "test";
   public void DoSomeThing() {
      System.out.println(testStr);
   }
}
```

9.5.2.3 移动包

此重构允许将包移动到不同的包中，以保持正确的目录结构。

执行重构

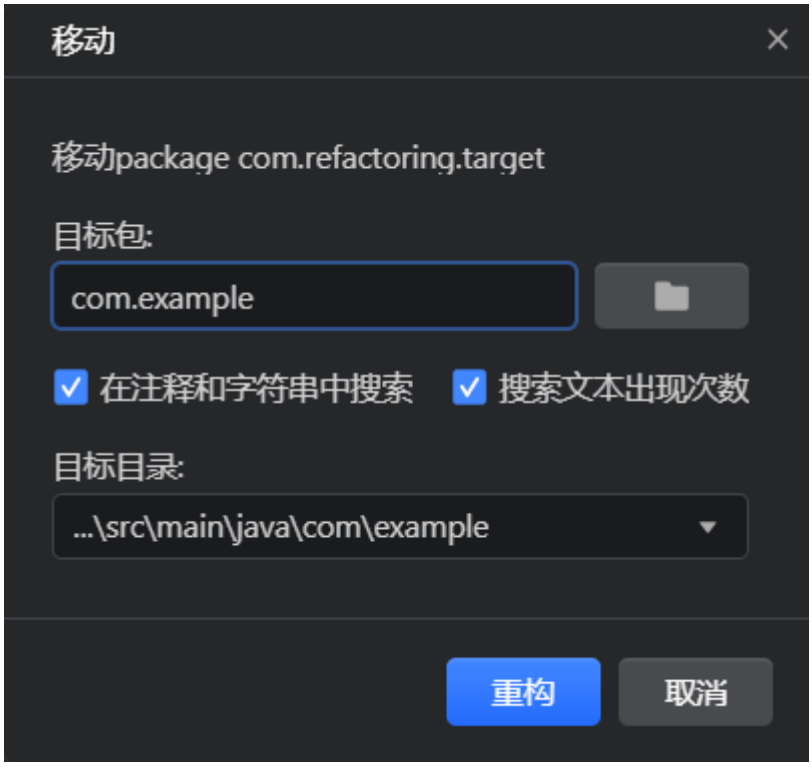
步骤1 在代码编辑器中，将光标放置在要移动的包声明上，或单击左侧任务栏的**Java工程**。

步骤2 单击右键展示上下文菜单，选择“重构 > 移动包...” 。

步骤3 在打开的“移动”对话框中，在“**目标包**”选择框中选择目标package。单击**浏览**按钮（  ），然后在打开的“选择目标包”对话框中，选择包或创建新包。要搜索代码中移动包的引用，请选中“**在注释和字符串中搜索**”和 “**搜索文本出现次数**”复选框。

如下图所示：

图 9-47 移动包



步骤4 单击“**重构**”以应用重构。

----结束

示例

作为示例，将包com.refactoring移动到包com.example中，并替换代码中出现的com.refactoring.target包的位置。

重构前

“com\refactoring\target\Refactoring.java”文件内容如下：

```
package com.refactoring.target;
public class Refactoring {
    public String testStr = "test";
    public void DoSomeThing() {
        System.out.println(testStr);
    }
}
```

重构后

“com.refactoring.target”包全部移动，改成“com.example.target”包。例如，包移动后的文件“com\example\target\Refactoring.java”，文件内容如下：

```
package com.example.target;
public class Refactoring {
    public String testStr = "test";
    public void DoSomeThing() {
        System.out.println(testStr);
    }
}
```

9.5.2.4 移动内部类到上一级

此重构支持将内部类移至上层，这个重构将包外的类、函数、变量、常量和命名空间移动到一个包中。

执行重构

步骤1 在代码编辑器中，将光标放在要移动到上层的内部类（非静态内部类）的声明位置。

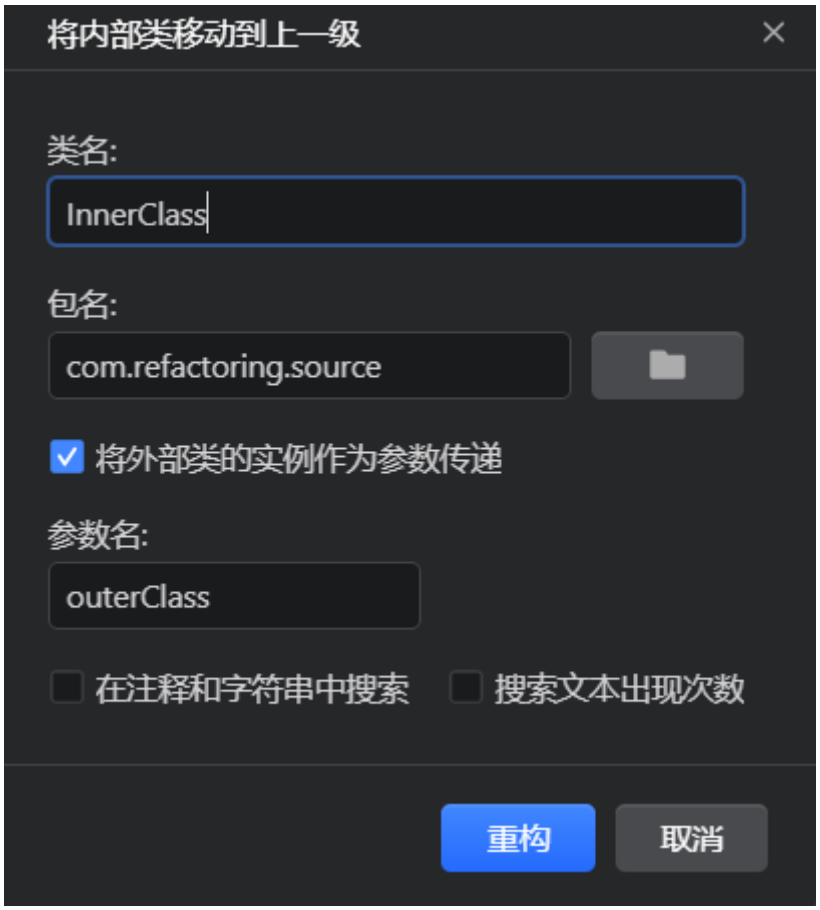
步骤2 单击右键展示上下文菜单，选择“重构 > 将内部类移动到上一级...”。

步骤3 在打开的“将内部类移动到上一级”对话框中，提供移动类的名称和其他重构选项。

- 要保留移动类对其以前的外部类的访问权限，请勾选复选框**“将外部类的实例作为参数传递”**。
- 要在搜索代码中移动类的引用，请勾选**“在注释和字符串中搜索”**和**“搜索文本出现次数”**复选框 。

  如下图所示：

图 9-48 将内部类移动到上一级



步骤4 单击**“重构”**以应用重构。

----结束

示例

例如，将InnerClass移动到上层。要保留从InnerClass到OuterClass的访问，OuterClass的实例将作为参数传递给InnerClass。

重构前

“com\refactoring\source\OuterClass.java” 文件内容如下：

```
class OuterClass {
    String str = "test";
    public void outermethod(){
        InnerClass ic = new InnerClass();
        ic.print();
    }

    class InnerClass {
        public void print(){
            System.out.println(str);
        }
    }
}
```

重构后

“com\refactoring\source\OuterClass.java” 文件删除内部类， 文件内容如下：

```
class OuterClass {
    String str = "test";
    public void outermethod(){
        InnerClass ic = new InnerClass(this);
        ic.print();
    }
}
```

新增“com\refactoring\source\InnerClass.java”文件，文件内容如下：

```
class InnerClass {
    private final OuterClass outerClass;

    public InnerClass(OuterClass outerClass) {
        this.outerClass = outerClass;
    }

    public void print() {
        System.out.println(outerClass.str);
    }
}
```

9.5.2.5 移动实例方法

如果方法在项目中具有类型参数，则此重构允许将实例（非静态）方法移动到其他类。

执行重构

步骤1 在代码编辑器中，将光标放在要移动到另一个类的实例方法的声明上。

步骤2 在单击右键展示上下文菜单，选择“重构 > 移动实例方法...”。

说明

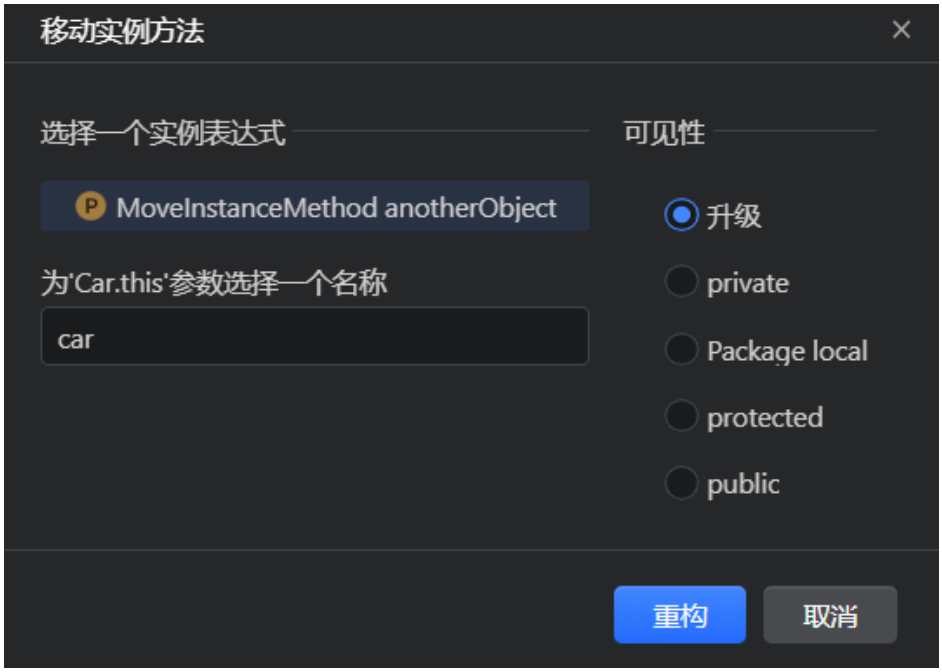
如果该方法在项目中没有类型参数，则需要将其设置为静态，然后将其移动到所需的目标类。有关详细信息，请参见**将内部类或实例转换为静态**和**移动静态成员**。

步骤3 在打开的“移动实例方法”对话框中，提供重构选项。

- 在**“选择一个实例表达式”**列表中，选择要将实例方法移动到的目标类。潜在移动目标的列表包括当前类中的方法参数的类和字段的类。
- 为将要移动的方法添加参数名称，并将替换对当前类所有参数的引用。
- 在**“可见性”**区域中，指定移动方法的可见性修改器，或选择**“升级”** 以自动将可见性设置为所需的级别。

  如下图所示：

  图 9-49 移动示例方法



步骤4 单击**“重构”**以应用重构。

----结束

示例

例如，将实例方法**getName**从**Car**类移动到**MoveInstanceMethod**类方法。

重构前

“com\refactoring\source\MoveInstanceMethod.java”文件内容如下：

```
public class MoveInstanceMethod {
    public static void main(String[] args) throws Exception {
        Car c = new Car();
        System.out.println(c.getName(new MoveInstanceMethod()));
    }
}

class Car {
    String name = "Default Car";

    String getName(MoveInstanceMethod anotherObject) {
        System.out.print(anotherObject.toString());
        return this.name;
    }
}
```


重构后

“com\refactoring\source\MoveInstanceMethod.java”文件的类“Car”将实例方法移动到“MoveInstanceMethod”类中，文件内容如下：

```
public class MoveInstanceMethod {
    public static void main(String[] args) throws Exception {
        Car c = new Car();
        System.out.println(new MoveInstanceMethod().getName(c));
    }

    String getName(Car car) {
        System.out.print(toString());
        return car.name;
    }
}

class Car {
    String name = "Default Car";
}
```

9.5.2.6 移动静态成员

此重构允许将类的静态成员移动到不同的类中。

执行重构

步骤1 在代码编辑器中，将光标放置在要移动到另一个类的静态成员（字段或方法）的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 移动静态成员...”。

步骤3 在打开的“移动静态成员”对话框中，提供重构选项。

- 提供有效的目标类名称。
- 在**“要移动的成员”**列表中，选中要移动的静态成员复选框。
- 在**“可见性”**区域中，指定移动的静态成员的可见性修改器，或选择**“升级”**以自动将可见性设置为所需的级别。
- 选中**“如果可能，作为枚举常量移动”**复选框，将常量（即**static final**字段）作为枚举常量移动到枚举类型。如果枚举类型的构造函数有类型参数，则可以作为枚举常量进行移动。

  如下图所示：

图 9-50 移动静态成员

步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，将类MoveStaticMembers的静态成员“STATICFINALSTR”移动到枚举类**MyEnum**。由于**MyEnum**有一个带有**String**类型参数的构造函数，因此可以使用“如果可能，作为枚举常量移动选项”将**STATICFINALSTR**和**staticStr**字段移动为枚举常量。

重构前

“com\refactoring\source\MoveStaticMembers.java”文件内容如下：

```
package com.refactoring.source;
import java.util.List;

class MoveStaticMembers {
    Boolean isTrue;
    public static final String STATICFINALSTR = "TEST";
    static List<Integer> staticList;
    static int[] staticN;
    private static final String staticStr = "test";
```

```
    public static void staticMethod() {
        System.out.println(staticStr);
    }

    private static Boolean staticMethod2() {
        return true;
    }

    void method() {
    }
}

enum MyEnum {
    ;
    String typeName;

    MyEnum(String name) {
        typeName = name;
    }
}
```

重构后

将“com\refactoring\source\MoveStaticMembers.java”文件中的静态成员“STATICFINALSTR”移动到MyEnum中，修改后的文件内容如下：

```
package com.refactoring.source;
import java.util.List;
class MoveStaticMembers {
    Boolean isTrue;
    static List<Integer> staticList;
    static int[] staticN;
    private static final String staticStr = "test";
    public static void staticMethod() {
        System.out.println(staticStr);
    }
    private static Boolean staticMethod2() {
        return true;
    }
    void method() {
    }
}
enum MyEnum {
    STATICFINALSTR("TEST");
    String typeName;
    MyEnum(String name) {
        typeName = name;
    }
}
```

9.5.2.7 上/下移成员

“上移成员”重构允许用户将类成员移动到超类或接口。**“下移成员”**重构的作用则相反，允许用户将类成员移动到子类。

执行重构

步骤1 在代码编辑器中，将光标放置在要向上拉或向下推类层次结构的字段或方法的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 上移成员 / 下移成员”。

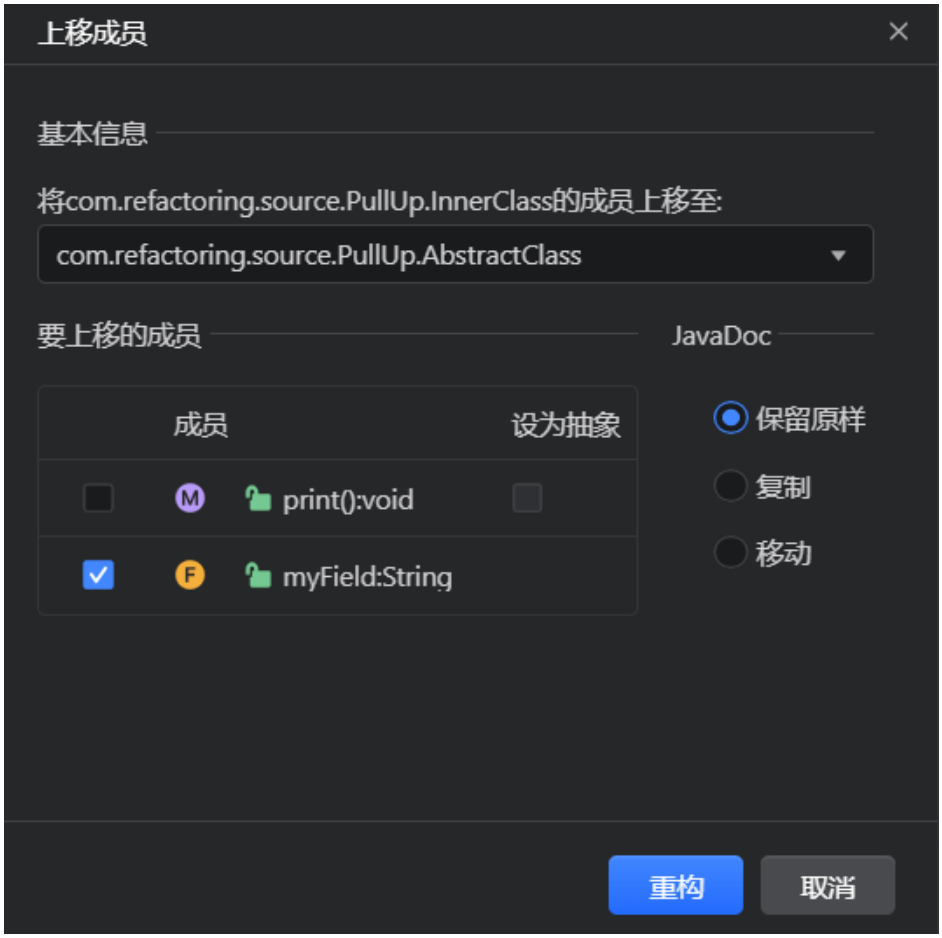
在打开的“上移成员”或“下移成员”对话框中，选择目标类并提供重构选项。

- 选中要向上（向下）移动的类成员的复选框。

- 对于方法，选中“**设为抽象**”复选框，将被移动的原始方法转换为抽象方法，并将其实现保留在原始类中。
- 在“**JavaDoc**”选项中，提供JavaDoc注释应与移动的类成员一起移动、复制还是保持原样的选择。

  如下图所示：

  图 9-51 上移成员

- 单击“**重构**”以应用重构。

----结束

示例

作为示例，将“InnerClass”中的字段**myField**和方法**print**移动到超类“AbstractClass”中。

重构前

“com\refactoring\source\PullUp.java”文件内容如下：

```
class PullUp {

  public static void main(String[] args) {
    new InnerClass().print();
  }

  private static class InnerClass extends AbstractClass {
```



```
    public String myField;
    public void print() {
      System.out.println("Hello World");
    }
  }

  private static abstract class AbstractClass {
  }
}
```

重构后

上移成员“myField”和“print”后，“com\refactoring\source\PullUp.java”文件内容如下：

```
class PullUp {

  public static void main(String[] args) {
    new InnerClass().print();
  }

  private static class InnerClass extends AbstractClass {
  }

  private static abstract class AbstractClass {
    public String myField;

    public void print() {
      System.out.println("Hello World");
    }
  }
}
```

9.5.3 提取/引入重构

9.5.3.1 引入变量

此重构允许创建新变量，通过选定的表达式进行初始化，并使用创建变量的引用替换原始表达式。这与**内联变量**相反。

执行重构

步骤1 在代码编辑器中，将光标放置在要提取到变量的表达式上。

步骤2 单击右键展示上下文菜单，选择“重构 > 引入变量...”。或者按“Ctrl+Alt+V”。

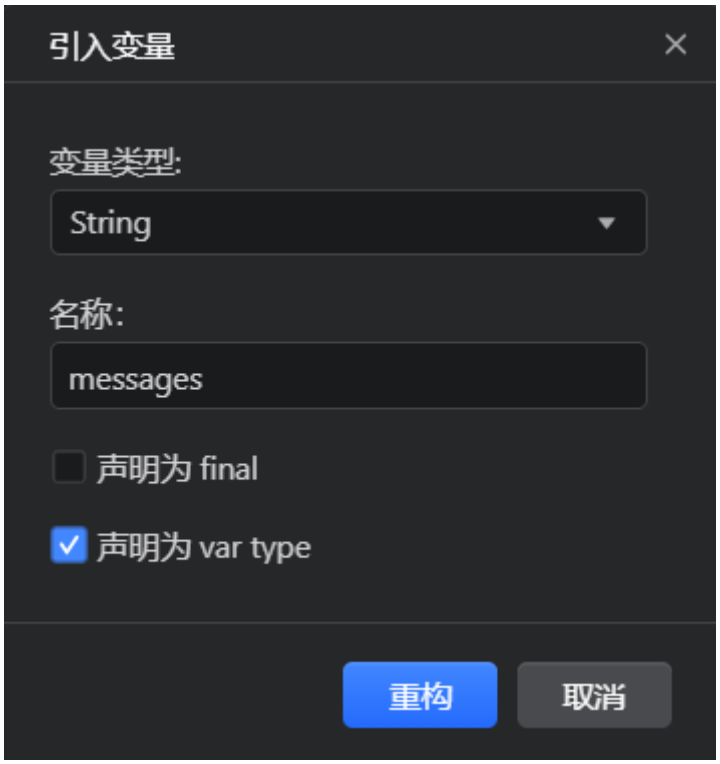
步骤3 如果多个表达式属于重构范围，请在弹窗中选择所需的表达式。

步骤4 在打开的“引入变量”对话框中，提供引入变量的类型和名称，并选择重构选项：

- 选择变量是否应“声明为final”。
- 如果项目的语言级别设置为Java 10以及更高的版本，则可以选择使用**var**标识符来声明变量，而不是显式提供其类型，这有助于提高代码的可读性。

  如下图所示：

图 9-52 引入变量



步骤5 单击“**重构**”以应用重构。

----结束

示例

例如，将表达式“Hello”+“ ”+“World!”提取到一个新的**message**变量中。

重构前

“com\refactoring\source\ExtractVariable.java”文件内容如下：

```
class ExtractVariable {

    public static void main(String[] args) {
        sayHello();
    }

    private static void sayHello() {
        System.out.println("Hello" + " " + "World!");
    }
}
```

重构后

将 "Hello" + " " + "World!" 提取变量，提取后，“com\refactoring\source\ExtractVariable.java”文件内容如下：

```
class ExtractVariable {

    public static void main(String[] args) {
        sayHello();
    }

    private static void sayHello() {
```



```
        String message = "Hello" + " " + "World!";
        System.out.println(message);
    }
}
```

9.5.3.2 引入参数

此重构允许为方法声明引入新参数，以便在方法调用中，将原始表达式作为方法参数传递。您还可以选择保留原始方法，或者使用创建的参数定义一个新方法。这与**内联参数**重构相反。

执行重构

步骤1 在代码编辑器中，将光标放置在要提取到参数的表达式上。

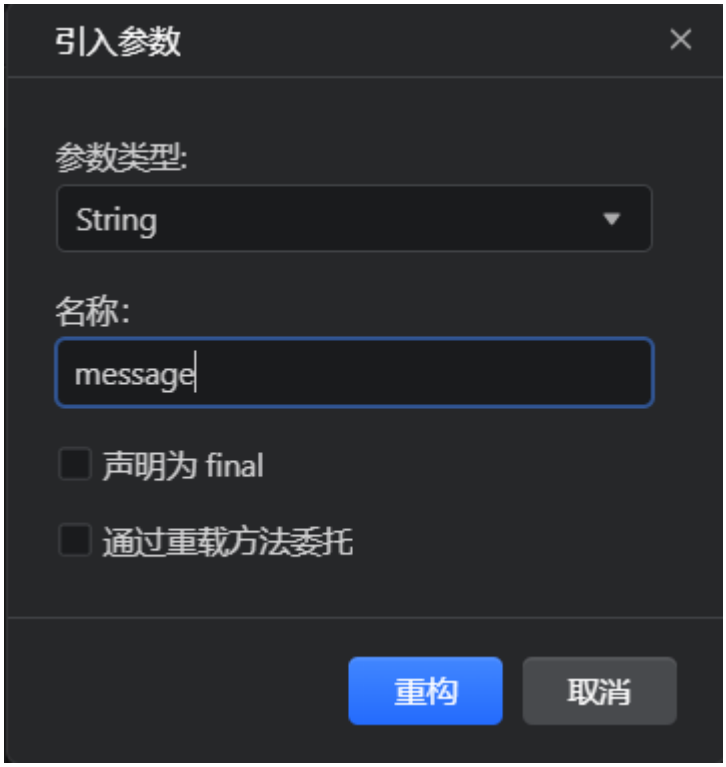
步骤2 单击右键展示上下文菜单，选择“重构 > 引入参数...”。或者按“Ctrl+Shift+Alt+P”。

步骤3 如果多个表达式属于重构范围，请在弹窗中选择所需的表达式。

步骤4 在打开的“引入参数”对话框中，提供引入参数的类型和名称，并选择是否应将参数“声明为final”参数。要保留原始方法并使用引入的参数定义新方法，请使用**“通过重载方法委托”**选项。

如下图所示：

图 9-53 引入参数



步骤5 单击**“重构”**以应用重构。

----结束

示例

作为示例，将表达式 "Hello" + " " + "World!" 提取到一个新的**message**参数中，并将其委托给一个重载的方法。

重构前

“com\refactoring\source\ExtractParameter.java”文件内容如下：

```
class ExtractParameter {

    public static void main(String[] args) {
        sayHello();
    }

    private static void sayHello() {
        System.out.println("Hello" + " " + "World!");
    }
}
```

重构后

将“println”的参数作为参数引入后，“com\refactoring\source\ExtractParameter.java”文件内容如下：

```
class ExtractParameter {

    public static void main(String[] args) {
        sayHello("Hello" + " " + "World!");
    }

    private static void sayHello(String message) {
        System.out.println(message);
    }
}
```

9.5.3.3 引入字段

此重构允许您创建一个新的类字段，使用选定的表达式初始化它，并使用创建的类字段的引用替换原始表达式。这与**内联字段**重构相反。

执行重构

步骤1 在代码编辑器中，将光标放置在要提取到类字段的表达式上。

步骤2 单击右键展示上下文菜单，选择“重构 > 引入字段...”。或者按“Ctrl+Shift+Alt+F”。

步骤3 如果多个表达式属于重构范围，请在出现的弹窗中选择所需的表达式。如下图所示：

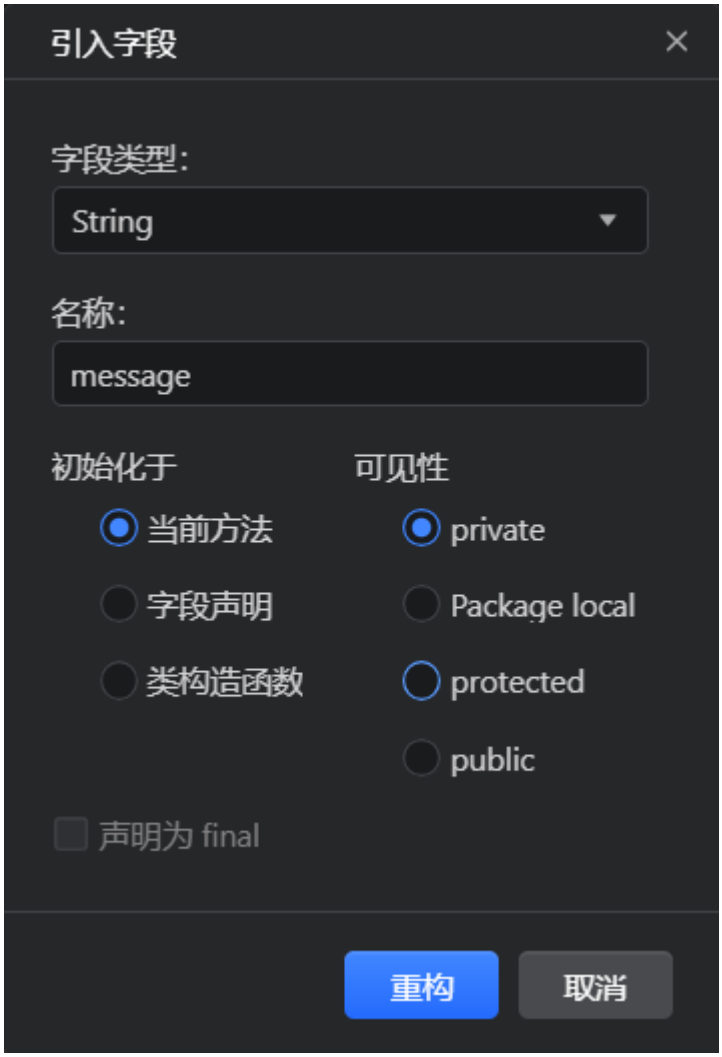
图 9-54 选择表达式

步骤4 在打开的“引入字段”对话框中，提供引入字段的**类型**和**名称**、**初始化于**和**可见性**选项。

说明

当“初始化于”勾选“字段声明”时，可以选择是否将字段“声明为final”字段。

图 9-55 引入字段



步骤5 单击“**重构**”以应用重构。

----结束

示例

例如，将表达式“Hello”+“ ”+“World!”提取到在类构造函数中初始化的新**message**私有字段。

重构前

“com\refactoring\source\ExtractField.java”文件内容如下：

```
class ExtractField {

    public static void main(String[] args) {
        sayHello();
    }
```



```
    private static void sayHello() {
        System.out.println("Hello" + " " + "World!");
    }
}
```

重构后

将表达式“Hello”+“ ”+“World!”提取到新的字段“message”后，“com\refactoring\source\ExtractField.java”文件内容如下：

```
class ExtractField {
    private static String messages;
    public static void main(String[] args) {
        sayHello();
    }
    private static void sayHello() {
        messages = "Hello" + " " + "World!";
        System.out.println(messages);
    }
}
```

9.5.3.4 引入常量

此重构允许创建新常量，通过使用选定的表达式进行初始化，并使用创建常量的引用替换原始表达式。

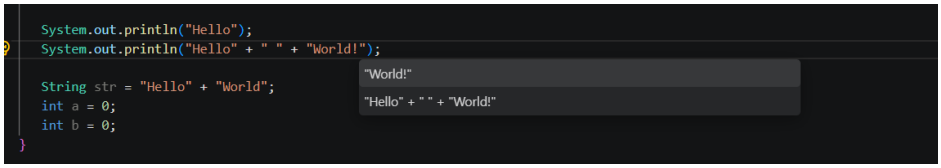
执行重构

步骤1 在代码编辑器中，将光标放置在要提取到常量的表达式上。

步骤2 单击右键展示上下文菜单，选择“重构 > 引入常量”。或者按“Ctrl+Alt+C”。

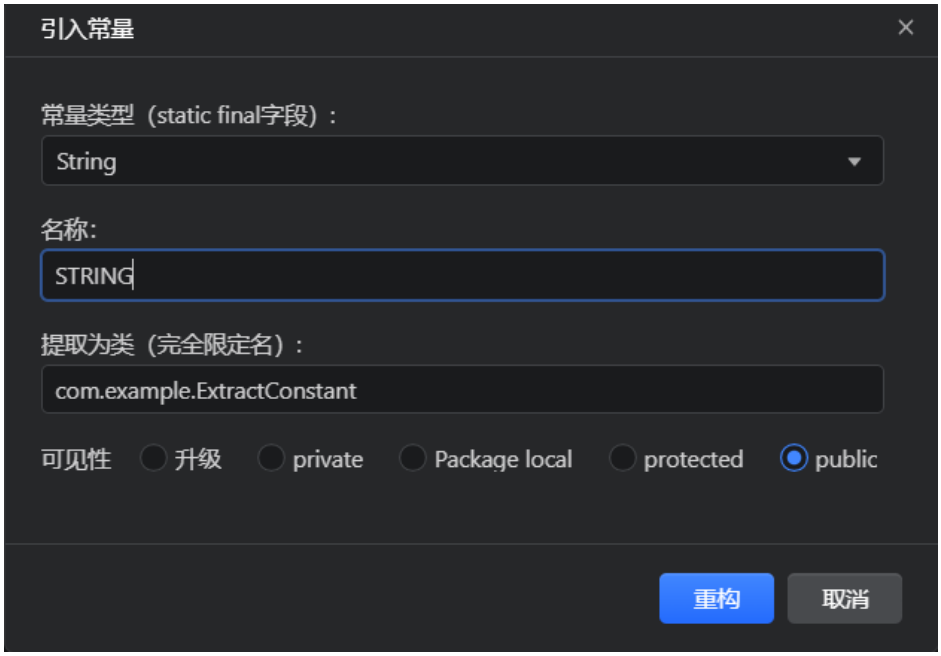
步骤3 如果多个表达式属于重构范围，请在出现的弹窗中选择所需的表达式。

图 9-56 选择表达式



步骤4 在打开的“引入常量”对话框中，提供引入常量的类型和名称，选择应声明常量的类和常量的可见性修饰符。

图 9-57 引入常量



步骤5 单击**“重构”**以应用重构。

----结束

示例

例如，将表达式“Hello”+“ ”+“World!”提取到一个新的“MESSAGE”常量中。

重构前

“com\refactoring\source\ExtractConstant.java”文件内容如下：

```
class ExtractConstant {

    public static void main(String[] args) {
        sayHello();
    }

    private static void sayHello() {
        System.out.println("Hello" + " " + "World!");
    }
}
```

重构后

将表达式“Hello”+“ ”+“World!”提取到新的常量“MESSAGE”后，“com\refactoring\source\ExtractConstant.java”文件内容如下：

```
class ExtractConstant {

    public static final String MESSAGE = "Hello" + " " + "World!";

    public static void main(String[] args) {
        sayHello();
    }

    private static void sayHello() {
        System.out.println(MESSAGE);
    }
}
```

9.5.3.5 提取方法

此重构允许将任意代码片段移动到单独的方法中，并将其替换为对此新创建的方法的调用。这与**内联方法**相反。

执行重构

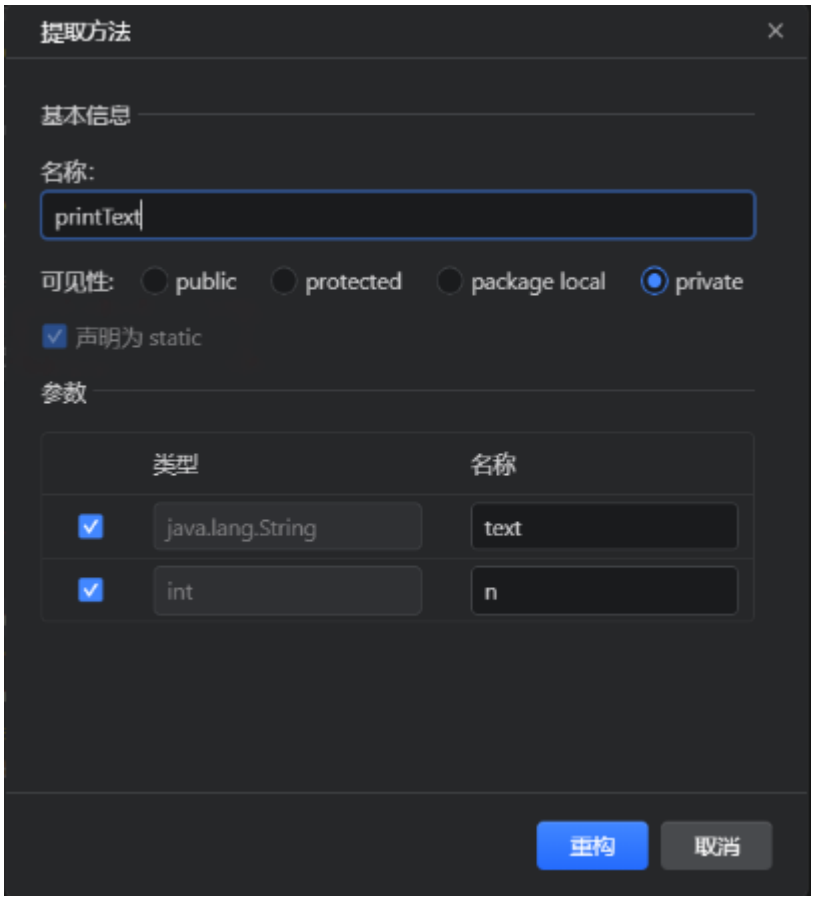
步骤1 在代码编辑器中，选择要提取到新方法的代码片段。

步骤2 单击右键展示上下文菜单，选择“重构 > 提取方法...”，或按“**Ctrl+Shift+Alt+M**”。

步骤3 在打开的“提取方法”对话框中，提供新方法的名称和可见性修饰符，并从选择范围中选择变量作为方法参数。

如下图所示：

图 9-58 提取方法



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，将包含println语句的for循环提取到一个新的printText方法中，其中text和n作为方法的参数。

重构前

“com\refactoring\source\ExtractMethod.java”文件内容如下：

```
class ExtractMethod {
  public static void main(String[] args) {
    String text = "Hello World!";
    int n = 5;

    for (int i = 0; i < n; i++) {
      System.out.println(text);
    }
  }
}
```

重构后

将for循环提取到新的方法“printText”后，“com\refactoring\source\ExtractMethod.java”文件内容如下：

```
class ExtractMethod {
  public static void main(String[] args) {
    String text = "Hello World!";
    int n = 5;

    printText(text, n);
  }

  public static void printText(String text, int n) {
    for (int i = 0; i < n; i++) {
      System.out.println(text);
    }
  }
}
```

9.5.3.6 提取接口

此重构允许选定现有类的成员来创建接口，以使它们可以被其他类继承。

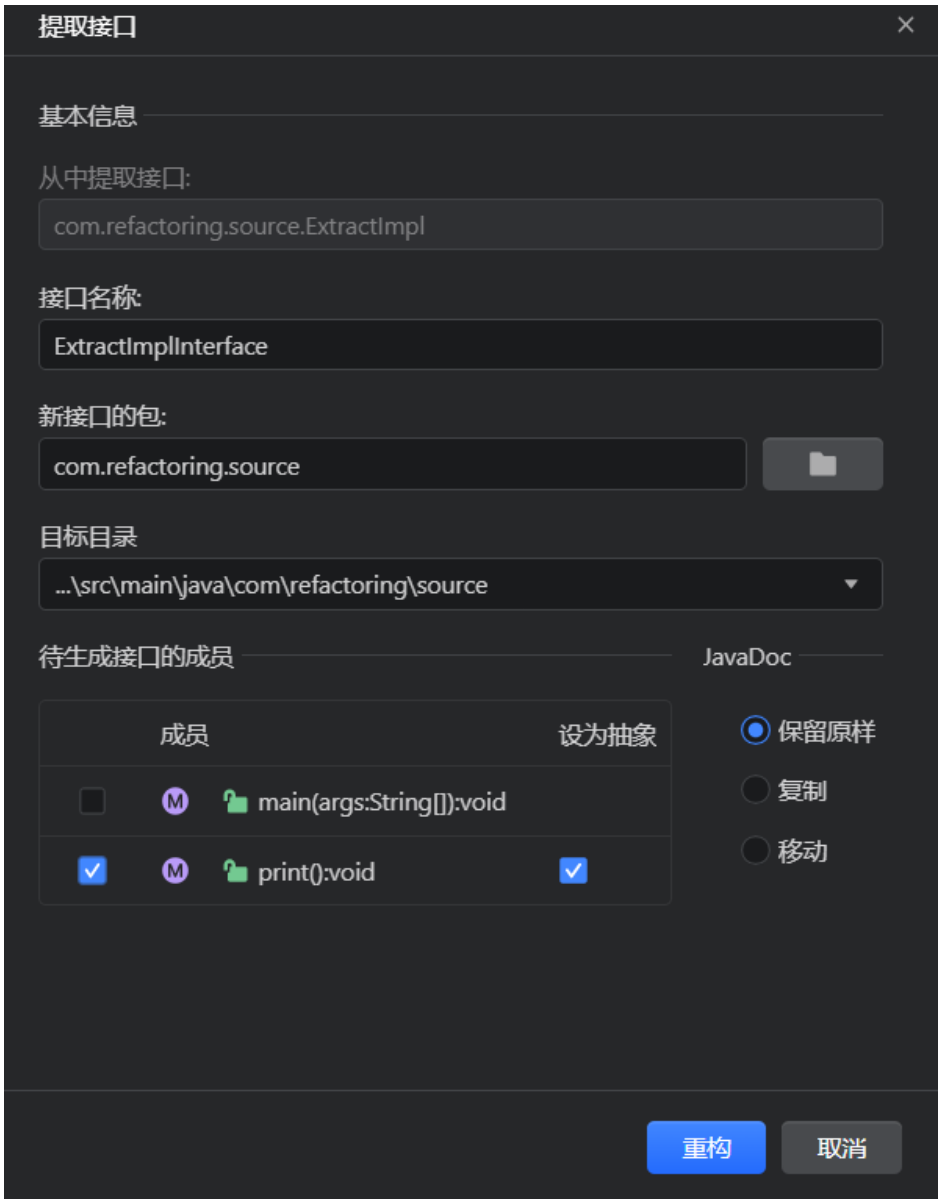
执行重构

步骤1 在代码编辑器中，将光标放置在要将其成员提取到接口的类中的任何位置。

步骤2 单击右键展示上下文菜单，选择“重构 > 提取接口...”。

步骤3 在打开的“提取接口”对话框中，提供提取接口的名称和包，选择要提取的类成员。在“**JavaDoc**”选项中，选择是将JavaDoc注释移动或复制到提取的接口，还是保持原样。如下图所示：

图 9-59 提取接口



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，基于提取**ExtractImpl**类的**print**方法创建一个新的提取**ExtractImplInterface**接口。

重构前

“com\refactoring\source\ExtractImpl.java”文件内容如下：

```
class ExtractImpl {
    public static void main(String[] args) {
        new ExtractImpl().print();
    }
```



```
    public void print() {
        System.out.println("Hello World!");
    }
}
```

重构后

“com\refactoring\source\ExtractImpl.java”文件内容如下：

```
class ExtractImpl implements ExtractImplInterface {
    public static void main(String[] args) {
        new ExtractImpl().print();
    }

    @Override
    public void print() {
        System.out.println("Hello World!");
    }
}
```

同时，生成一个“com\refactoring\source\ExtractImplInterface.java”文件，文件内容如下：

```
package com.refactoring.source;
public interface ExtractImplInterface {
    void print();
}
```

9.5.3.7 提取超类

此重构允许选定现有类的成员创建新的超类。这与**内联超类**相反。

执行重构

步骤1 在代码编辑器中，将光标放置在要将其成员提取到超类中的任何位置。

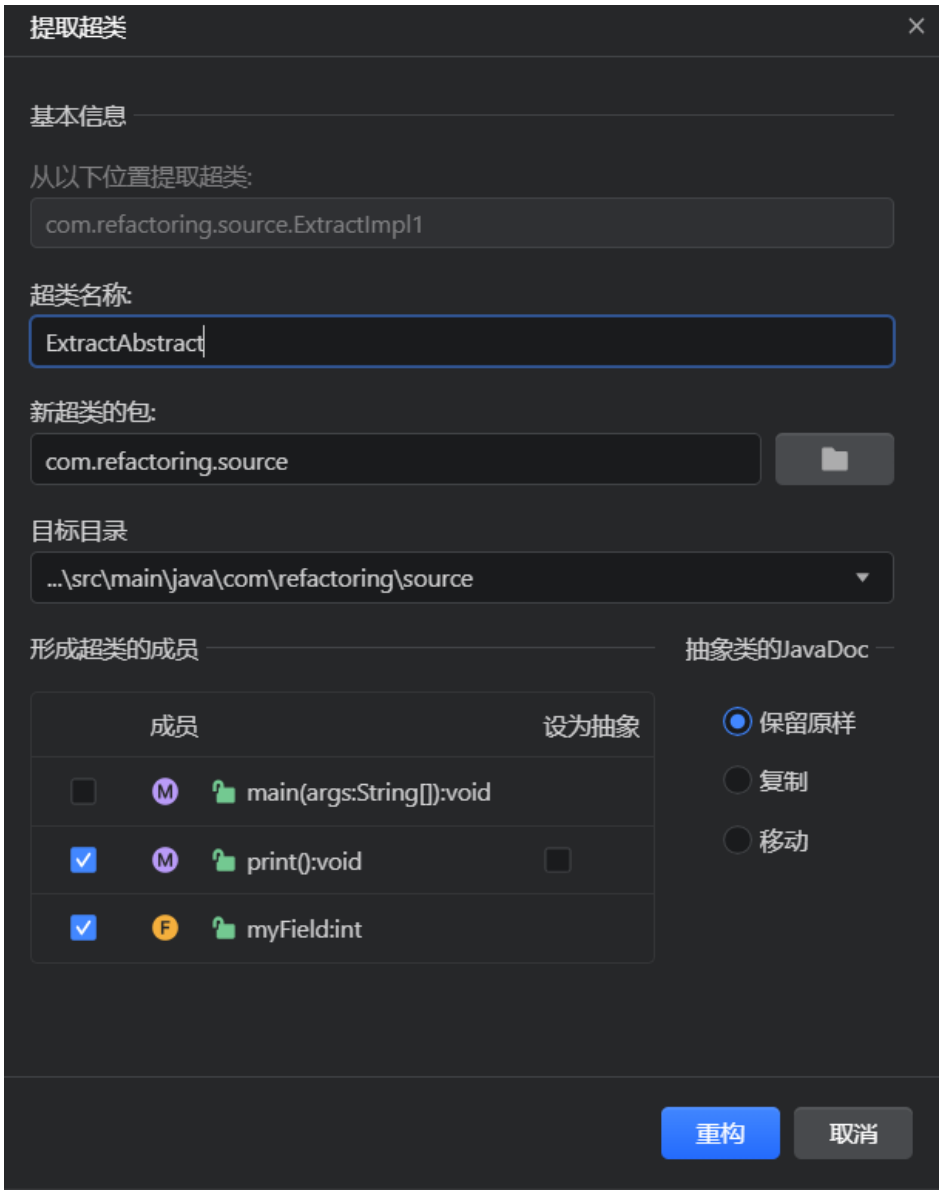
步骤2 单击右键展示上下文菜单，选择“重构 > 提取超类...”。

步骤3 在打开的“提取超类”对话框中，提供重构参数。

- 提供提取的超类名称和包。
- 在**“形成超类的成员”**区域中，选择要提取的类成员。对于方法，选中**“设为抽象”**复选框，将提取的方法声明为超类中的**abstract**方法，并将其实现保留在原始类中。
- 在**“抽象类的JavaDoc”**选项中，选择是将JavaDoc注释移动或复制到提取的超类，还是保持原样。

  如下图所示：

图 9-60 提取超类



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，基于提取**ExtractImpl**类的**print**方法和**myField**字段创建一个新的**ExtractAbstract**超类。在创建的超类中，**print**方法将被声明为**abstract**。

重构前

“com\refactoring\source\ExtractImpl.java”文件内容如下：

```
class ExtractImpl {
  public int myField;

  public static void main(String[] args) {
    new ExtractImpl().print();
```



```
  }

  public void print() {
    System.out.println("Hello World!");
  }
}
```

重构后

“com\refactoring\source\ExtractImpl.java”文件内容如下：

```
class ExtractImpl extends ExtractAbstract {
  public static void main(String[] args) {
    new ExtractImpl().print();
  }
}
```

同时，生成一个“com\refactoring\source\ExtractAbstract.java”文件，文件内容如下：

```
package com.refactoring.source;
public class ExtractAbstract {
    public int myField;
    public void print() {
        System.out.println("Hello World!");
    }
}
```

9.5.3.8 提取委托

此重构允许基于现有类的成员选定来创建新类。

执行重构

步骤1 在代码编辑器中，将光标放置在要将其成员提取到新类的类中的任何位置。

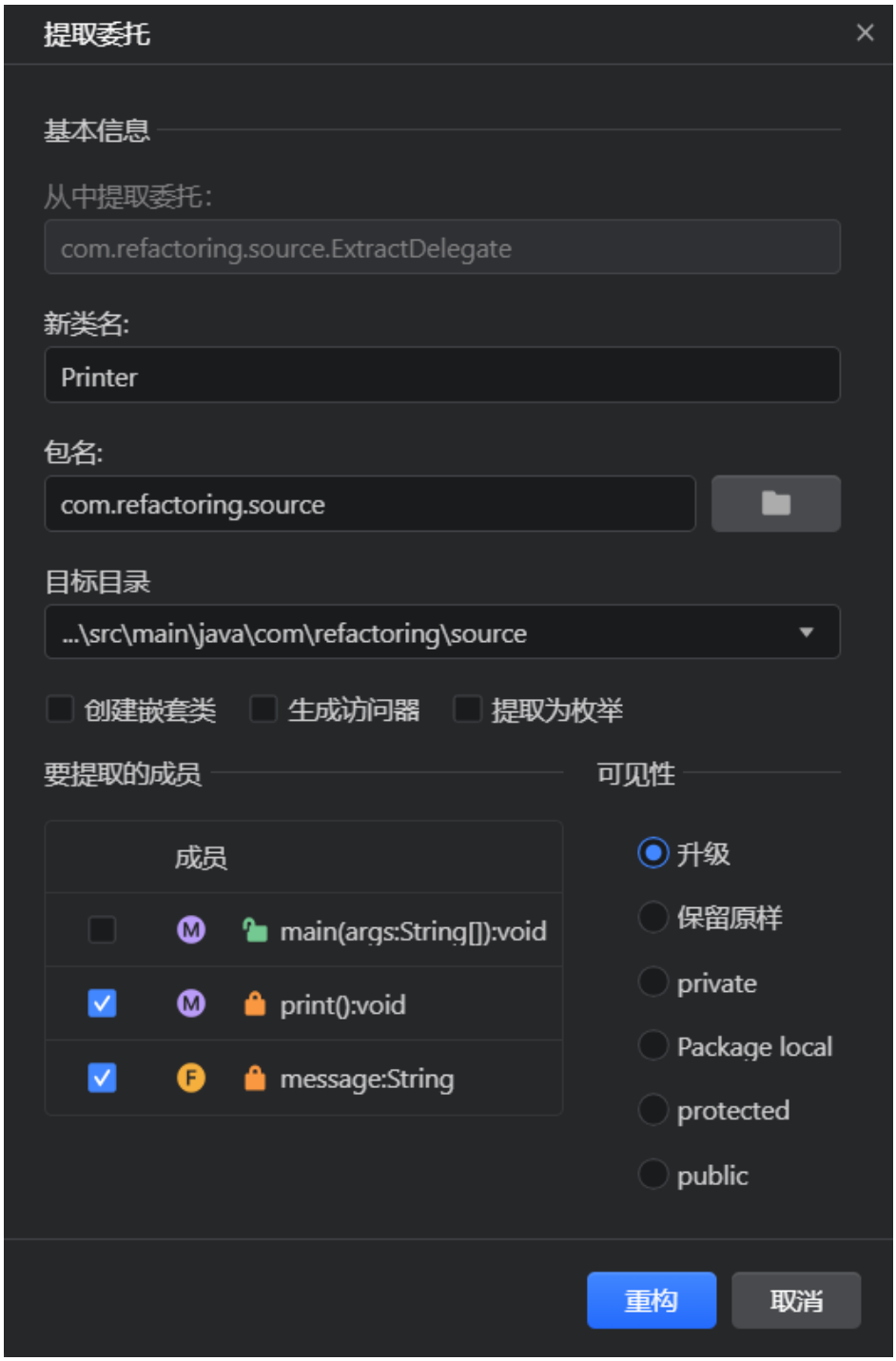
步骤2 单击右键展示上下文菜单，选择“重构 > 提取委托...”。

步骤3 在打开的“提取委托”对话框中，提供重构参数。

- 提供提取类的名称、包和目标目录。
- 选中**“创建嵌套类”**复选框以在当前类中创建新类。
- 选中**“生成访问器”**复选框，为提取的字段生成getter方法。
- 选中**“提取为枚举”**复选框，将提取的类创建为枚举类。如果源类包含静态最终字段**static final fields**，这是可能的。
- 在**“要提取的成员”**区域中，选择要提取的类成员。如果选择了**“提取为枚举”**，则还可以选择要作为枚举常量提取的字段旁边的**“作为枚举”**复选框。
- 在**“可见性”**选项中，选择要应用于提取的类成员的可见性修改器。要自动应用所需的可见性修改器，以便访问类成员，请选择**“升级”**选项。

  如下图所示：

图 9-61 提取委托



步骤4 单击“**重构**”以应用重构。

----结束

示例

作为示例，将**ExtractDelegate**类中的**print**方法提取到一个嵌套的**Printer**枚举类中。

重构前

“com\refactoring\source\ExtractDelegate.java”文件内容如下：

```
class ExtractDelegate {
    public static void main(String[] args) {
        new ExtractDelegate().print();
    }

    private static final String message = "Hello World!";

    private void print() {
        System.out.println(message);
    }
}
```

重构后

“com\refactoring\source\ExtractDelegate.java”文件内容如下：

```
class ExtractDelegate {
    private final Printer printer = new Printer();
    public static void main(String[] args) {
        new ExtractDelegate().print();
    }
    private void print() {
        printer.print();
    }
}
```

同时，生成一个“com\refactoring\source\Printer.java”文件，文件内容如下：

```
package com.refactoring.source;
public class Printer {
    static final String message = "Hello World!";
    public Printer() {
    }
    void print() {
        System.out.println(message);
    }
}
```

9.5.3.9 引入函数式参数

此重构允许基于适当的函数接口使用匿名类（或函数表达式）包装代码片段，并将其用作方法的参数。

执行重构

步骤1 在代码编辑器中，选择要转换为函数参数的表达式。

步骤2 单击右键展示上下文菜单，选择“重构 > 引入函数式参数...”。

步骤3 在选择适用的函数接口弹窗选择需要的接口。如下图所示：

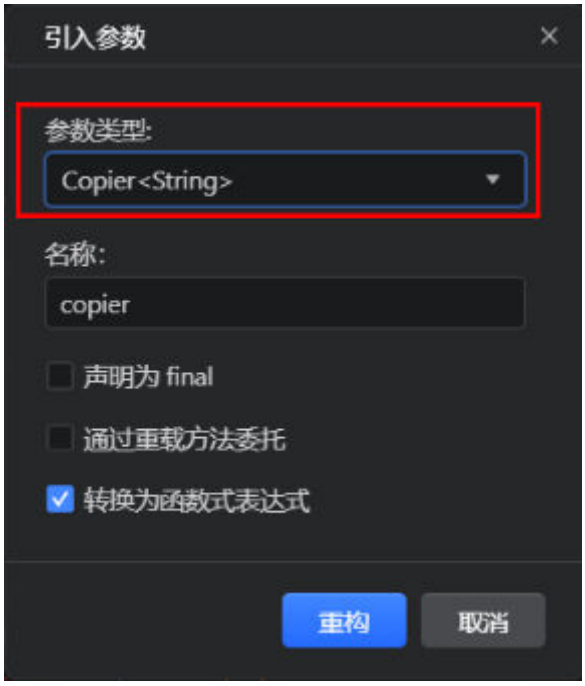
图 9-62 选择适用的函数接口

步骤4 选定接口后，会打开“引入参数”对话框，该弹窗提供引入参数的名称和其他重构选项。

- 在**“参数类型”**列表中，为提取的参数选择其中一种的类型。
- 选择是否应将提取的参数“声明为final”参数。
- 要保留原始方法并使用引入的参数定义新方法，请选中**“通过重载方法委托”**选项。
- 要让CodeArts IDE生成函数表达式而不是匿名类，请选中**“转换为函数表达式”**选项。

  如下图所示：

  图 9-63 引入参数

步骤5 单击**“重构”**以应用重构。

----结束

示例

例如，提取表达式“**Hello World!”.toUpperCase()**作为**generateText**方法的参数。

重构前

“com\refactoring\source\IntroduceFunctionalParameter.java”文件内容如下：

```
class IntroduceFunctionalParameter {
    public static void main(String[] args) {
        System.out.println(generateText());
    }

    private static String generateText() {
        return "Hello World!".toUpperCase();
    }
}
```

重构后

“com\refactoring\source\IntroduceFunctionalParameter.java”文件内容如下：

```
import java.util.function.Supplier;

class IntroduceFunctionalParameter {
    public static void main(String[] args) {
        System.out.println(generateText("Hello World!"::toUpperCase));
    }
    private static String generateText(Supplier<String> supplier) {
        return supplier.get();
    }
}
```

9.5.3.10 引入函数式变量

此重构允许将选定的表达式转换为新的函数类型变量或匿名类。

执行重构

步骤1 在代码编辑器中，选择要转换为函数变量的表达式。

步骤2 单击右键展示上下文菜单，选择“重构 > 引入函数式变量...”。

步骤3 在打开的引入函数式变量对话框中，选择**“将字段作为参数传递”**，以使字段作为参数传递到创建的函数表达式。如下图所示：

图 9-64 引入函数式变量

步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，将表达式“**Data**”**+ data.toString()**提取到函数变量中。

重构前

“com\refactoring\source\PrintAction.java”文件内容如下：

```
import java.util.List;

class PrintAction implements Runnable {
    private List<String> data;

    public PrintAction(List<String> data) {
        this.data = data;
    }

    @Override
    public void run() {
        System.out.println("Data" + data.toString());
    }
}
```

重构后

“com\refactoring\source\PrintAction.java”文件内容如下：

```
import java.util.List;
import java.util.function.Function;
```

```
class PrintAction implements Runnable {
    private List<String> data;

    public PrintAction(List<String> data) {
        this.data = data;
    }

    @Override
    public void run() {
        Function<List<String>, String> listStringFunction = data -> "Data" + data.toString();
        System.out.println(listStringFunction.apply(data));
    }
}
```

9.5.3.11 提取方法对象

此重构允许将任意代码片段单独移动到新类的方法中，以便进一步将该方法分解为同一对象上的其他方法。

执行重构

步骤1 在代码编辑器中，选择要提取到包装类的新方法的代码片段。

步骤2 单击右键展示上下文菜单，选择“重构 > 提取方法对象...”。

步骤3 在打开的“提取方法对象”对话框中，提供重构选项。

- **“创建内部类”**：选择创建新的内部类。所有局部变量都转换为此类的字段。提供类的名称及其可见性修饰符。
- **“创建匿名类”**：选择创建新对象并提供需要创建方法的名称。
- 在**“参数”**区域中，选择变量作为方法参数。

  如下图所示：

图 9-65 提取方法对象

步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，将包含 **println**语句的**for**循环提取到**Printer**包装类的新方法中。

重构前

“com\refactoring\source\ExtractMethodObject.java”文件内容如下：

```
class ExtractMethodObject {
  public static void main(String[] args) {
    String text = "Hello World!";
    int n = 5;

    for (int i = 0; i < n; i++) {
      System.out.println(text);
    }
```



```
  }
}
```

重构后

“com\refactoring\source\ExtractMethodObject.java”文件内容如下：

```
class ExtractMethodObject {
  public static void main(String[] args) {
    String text = "Hello World!";
    int n = 5;

    new Printer(text, n).invoke();
  }

  private static class Printer {
    private String text;
    private int n;

    public Printer(String text, int n) {
      this.text = text;
      this.n = n;
    }

    public void invoke() {
      for (int i = 0; i < n; i++) {
        System.out.println(text);
      }
    }
  }
}
```

9.5.3.12 引入参数对象

此重构允许将方法的参数移动到新的包装类或某些现有的包装类。所有参数的用法都将替换为对包装类的相应调用。

执行重构

步骤1 在代码编辑器中，将光标放置在要提取到包装类的参数上。

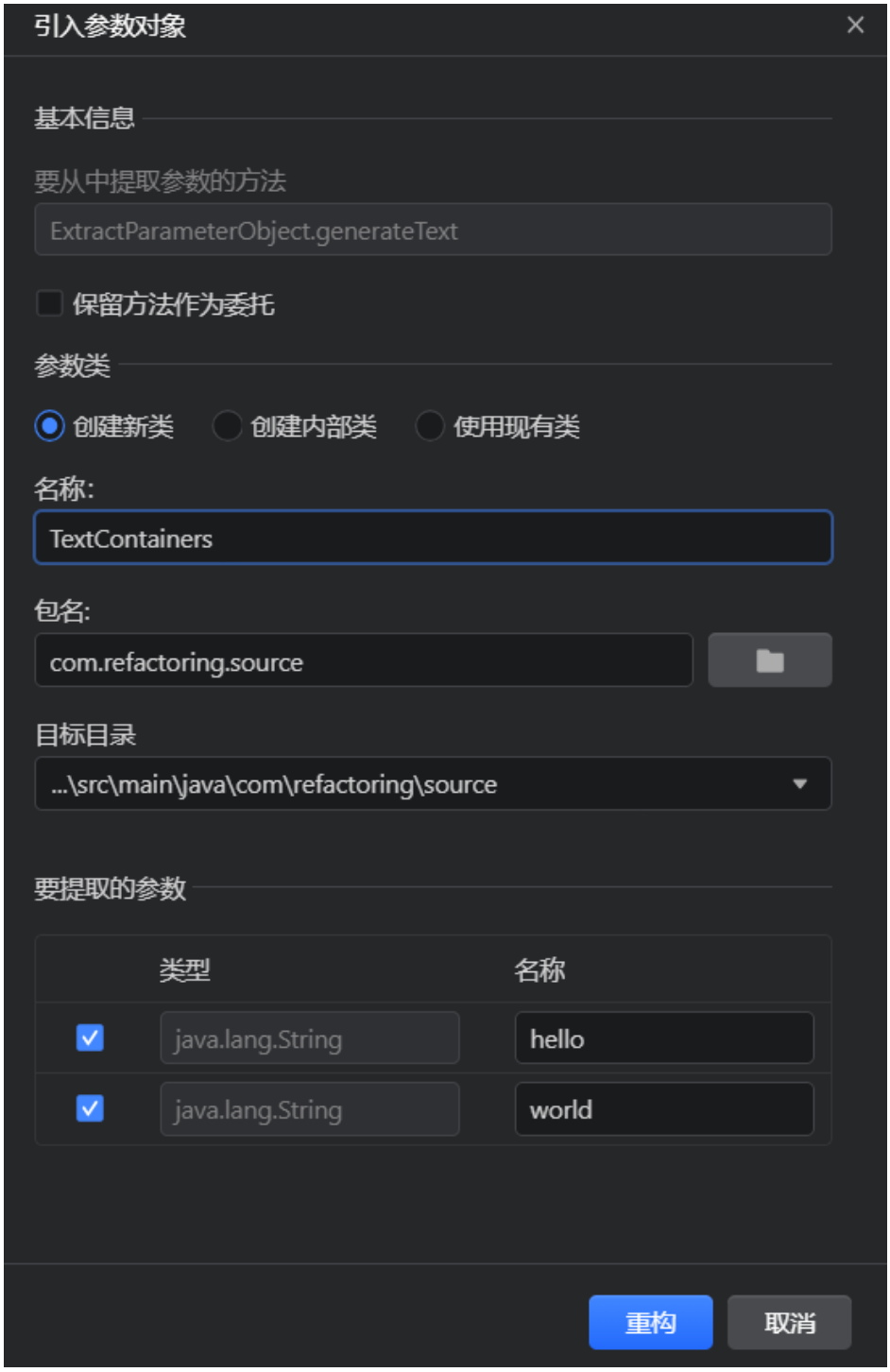
步骤2 单击右键展示上下文菜单，选择“重构 > 引入参数对象...”。

步骤3 在打开的“引入参数对象”对话框中，提供重构选项。

- **“保留方法作为委托”**：选择将原始方法保留为新创建方法的委托。
- **“参数类”**： 在此区域中，选择是要创建新的包装参数类、在当前包装参数类中创建内部类，还是使用某些现有类。
- **“名称”**：输入包装后参数的名称。
- **“要提取的参数”**： 在此区域中，选中要提取到包装参数类的参数旁边的复选框。

  如下图所示：

图 9-66 引入参数对象



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，将**hello**和**world**参数提取到**TextContainer**类，以使生成的**generateText**方法调用将包含对**TextContainer**对象的调用。

重构前

“com\refactoring\source\ExtractParameterObject.java”文件内容如下：

```
class ExtractParameterObject {
    public static void main(String[] args) {
        System.out.println(generateText("Hello", "World!"));
    }

    private static String generateText(String hello, String world) {
        return hello.toUpperCase() + world.toUpperCase();
    }
}
```

重构后

“com\refactoring\source\ExtractParameterObject.java”文件内容如下：

```
class ExtractParameterObject {
    public static void main(String[] args) {
        System.out.println(generateText(new TextContainers("Hello", "World!")));
    }
    private static String generateText(TextContainers textContainers) {
        return textContainers.getHello().toUpperCase() + textContainers.getWorld().toUpperCase();
    }
}
```

同时，生成一个“com\refactoring\source\TextContainers.java”文件，文件内容如下：

```
package com.refactoring.source;
public class TextContainers {
    private final String hello;
    private final String world;
    public TextContainers(String hello, String world) {
        this.hello = hello;
        this.world = world;
    }
    public String getHello() {
        return hello;
    }
    public String getWorld() {
        return world;
    }
}
```

9.5.4 内联重构

9.5.4.1 内联变量

此重构允许用变量的初始化器替换变量。

执行重构

步骤1 在代码编辑器中，将光标放置在要内联其值的变量的用法上。

步骤2 单击右键展示上下文菜单，选择“重构 > 内联变量...”，或按“Ctrl+Alt+N”。

步骤3 在打开的“内联变量”对话框中，选择是内联所有变量的引用，还是仅内联当前引用。如下图所示：

图 9-67 内联变量

说明

如果在代码中修改了变量的初始值，则仅内联修改之前的引用。

步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，内联变量**number**，用其初始化器**test.intValue()**替换它。请注意，由于变量在代码中被进一步修改，因此只有它在修改之前的一次引用会受到重构的影响。

重构前

“com\refactoring\source\InlineVariable.java”文件内容如下：

```
class InlineVariable {
    private int a;
    private Byte test;
    private int b;

    public void InlineVariable() {
        int number = test.intValue();
        int b = a + number;
        number = 42;
    }
}
```

重构后

“com\refactoring\source\InlineVariable.java”文件内容如下：

```
class InlineVariable {
    private int a;
    private Byte test;
    private int b;

    public void InlineVariable() {
        int number;
```

```
        int b = a + test.intValue();
        number = 42;
    }
}
```

9.5.4.2 内联参数

此重构允许使用方法调用中相应参数的值替换方法的参数。这与**引入参数**相反。

执行重构

步骤1 在代码编辑器中，将光标放置在要内联其值的方法参数的声明或用法上。

步骤2 单击右键展示上下文菜单，选择“重构 > 内联参数...”。

----结束

示例

例如，内联参数**pi**，将其替换为参数的值**Math.PI**。

重构前

“com\refactoring\source\InlineParameter.java”文件内容如下：

```
class InlineParameter {
    private double InlineParameter(double rad, double pi) {
        return pi * rad * rad;
    }

    public void Test() {
        double area = InlineParameter(10, Math.PI);
    }
}
```

重构后

“com\refactoring\source\InlineParameter.java”文件内容如下：

```
class InlineParameter {
    private double InlineParameter(double rad) {
        return Math.PI * rad * rad;
    }

    public void Test() {
        double area = InlineParameter(10);
    }
}
```

9.5.4.3 内联方法

此重构允许用方法的主体替换方法的用法。这与**提取方法**相反。

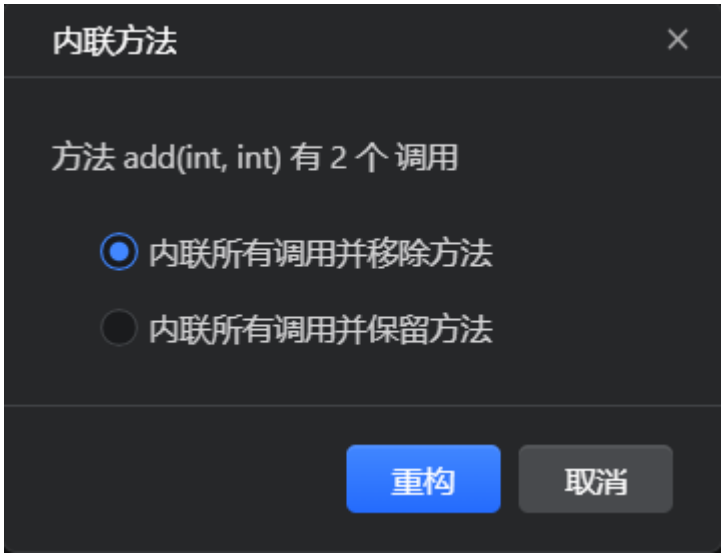
执行重构

步骤1 在代码编辑器中，将光标放置在要内联的方法的声明或调用上。

步骤2 单击右键展示上下文菜单，选择“重构 > 内联方法...”，或按“**Ctrl+Shift+Alt+L**”。

步骤3 在打开的“内联方法”对话框中，选择是否在方法的所有引用都内联后保留该方法。如下图所示：

图 9-68 内联方法



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，内联方法add，用方法的主体替换其调用。

重构前

“com\refactoring\source\InlineMethod.java”文件内容如下：

```
class InlineMethod {
  private int a;
  private int b;

  public void InlineMethod() {
    int c = add(a, b);
    int d = add(a, c);
  }

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

重构后

“com\refactoring\source\InlineMethod.java”文件内容如下：

```
class InlineMethod {
   private int a;
   private int b;

   public void InlineMethod() {
      int c = a + b;
      int d = a + c;
   }
}
```

9.5.4.4 内联字段

此重构操作允许将字段的使用替换为其值，并删除字段的声明。这与引入字段相反。

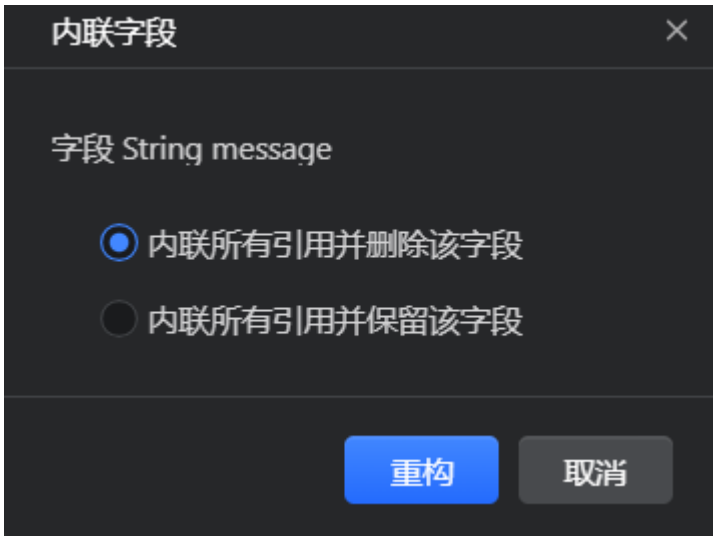
执行重构

步骤1 在代码编辑器中，将光标放在用户想要内联其值的字段的声明或使用位置。

步骤2 单击右键展示上下文菜单，选择“重构 > 内联字段...”。

步骤3 在打开的“内联字段”对话框中，选择是否在内联所有引用后删除该字段。如下图所示：

图 9-69 内联字段



步骤4 单击**“重构”**以应用重构。

----结束

示例

将字段**message**内联，将其使用位置替换为其初始化值**"Hello World!"**。

重构前

“com\refactoring\source\InlineField.java”文件内容如下：

```
class InlineField {
    private String message = "Hello World!";

    private void InlineField() {
        System.out.println(message);
    }
}
```

重构后

“com\refactoring\source\InlineField.java”文件内容如下：

```
class InlineField {
    private void InlineField() {
        System.out.println("Hello World!");
    }
}
```


9.5.4.5 内联超类

此重构操作允许将超类的成员移动到子类中，并删除超类。这与**提取超类**相反。

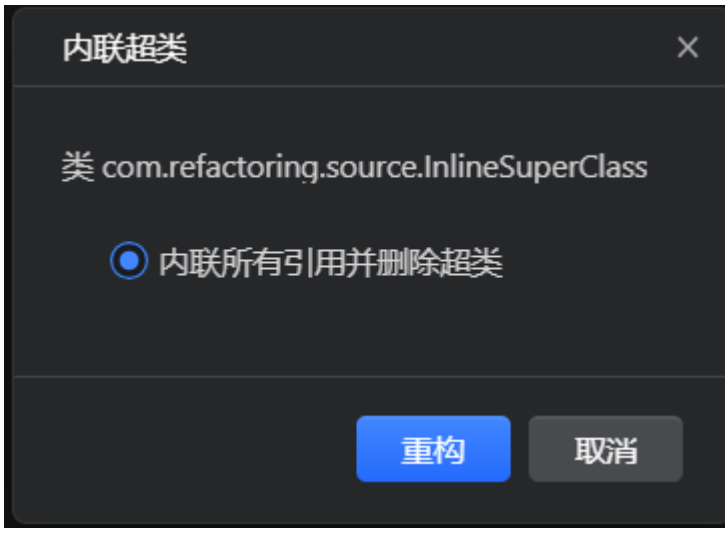
执行重构

步骤1 在代码编辑器中，将光标放在用户想要内联的超类的声明或引用位置。

步骤2 单击右键展示上下文菜单，选择“重构 > 内联超类...”。

步骤3 在打开的“内联超类”对话框中，选择内联所有引用并删除超类。如下图所示：

图 9-70 内联超类



步骤4 单击**“重构”**以应用重构操作。

----结束

示例

举个例子，将类**InlineSuperClass**内联并删除，将其成员移动到**SubClass**中。

重构前

“com\refactoring\source\SubClass.java”文件内容如下：

```
class InlineSuperClass {
    public int returnValue() { ... }
    public int returnNewValue() { ... }
}

class SubClass extends InlineSuperClass {
    private int myValue;

    int someMethod() {
        if (myValue > returnValue()) {
            return returnNewValue();
        }
        return 0;
    }
}
```

重构后

“com\refactoring\source\SubClass.java”文件内容如下：

```
class SubClass {
    private int myValue;

    int someMethod() {
        if (myValue > returnValue()) {
            return returnNewValue();
        }
        return 0;
    }

    public int returnValue() { ... }
    public int returnNewValue() { ... }
}
```

9.5.4.6 内联为匿名类

此重构操作允许用其内容替换多余的类。从Java 8开始，内联的匿名类可以自动转换为lambda表达式。

执行重构

步骤1 在代码编辑器中，将光标放在用户想要内联为匿名类的声明位置。

步骤2 单击右键展示上下文菜单，选择“重构 > 内联为匿名类...”。

步骤3 在打开的“内联为匿名类”对话框中，选择是否在所有内联位置完成后保留该类。如下图所示：

图 9-71 内联为匿名类

步骤4 单击“**重构**”以应用重构操作。

----结束

示例

举个例子，将类**MyComparator**内联并删除。生成的匿名类将自动转换为lambda表达式。

重构前

“com\refactoring\source\InlineAnonymousClazz.java”文件内容如下：


```
class InlineAnonymousClazz {
    public class MyComparator implements Comparator<String> {
        @Override
        public int compare(String s1, String s2) {
            return 0;
        }
    }

    void sort(List<String> scores) {
        scores.sort(new MyComparator());
    }
}
```

重构后

“com\refactoring\source\InlineAnonymousClazz.java”文件内容如下：

```
class InlineAnonymousClazz {
    void sort(List<String> scores) {
        scores.sort((s1, s2) -> 0);
    }
}
```

9.5.5 将内部类或实例转换为静态

通过此重构，可以将内部类转换为嵌套的静态类，或将实例方法转换为静态方法。

执行重构

步骤1 在代码编辑器中，将光标放在要转换为静态的类或方法的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 使静态...”。

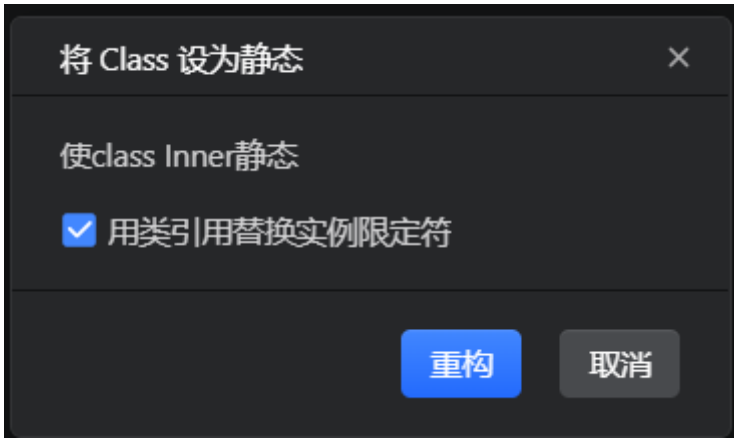
步骤3 在打开的“将Class设为静态”对话框中，提供重构参数。

- 如果类或方法包含对外部类字段的引用，则可以将被引用的对象作为参数传递给类构造函数，或者将被引用的字段作为方法的参数传递给类构造函数。如下图所示：

图 9-72 将 Class 设为静态

- 否则，如果类或方法不包含对外部类字段的引用，则可以用类引用替换实例限定符。如下图所示：

图 9-73 将 Class 设为静态



步骤4 单击“**重构**”以应用重构。

----结束

示例

作为一个例子，将**Inner**内部类转换为嵌套的静态类。由于**Inner**类包含对**Outer**类的**message**字段的引用，可以将**Outer**对象和**message**字段作为**Inner**类构造函数的参数添加进去。

重构前

“com\refactoring\source\Outer.java”文件内容如下：

```
class Outer {
  public String message;
  public static void main(String[] args) {

  }

  class Inner{
    public void print() {
      System.out.println(message);
    }
  }
}
```

重构后

“com\refactoring\source\Outer.java”文件内容如下：

```
class Outer {
  public String message;
  public static void main(String[] args) {

  }

  static class Inner {

    private String message;

    public Inner( String message) {
      this.message = message;
    }

    public void print() {
      System.out.println(message);
    }
  }
}
```

9.5.6 反转布尔值

通过此重构，可以反转布尔变量的值或方法的返回值。

执行重构

步骤1 在代码编辑器中，将光标放置在布尔变量或具有布尔返回值的方法的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 反转布尔值”。

步骤3 在打开的“反转布尔值”对话框中，为反转变量或方法提供新名称。如下图所示：

图 9-74 反转布尔值

步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，反转**condition**变量和**checkCondition**方法的值。

重构前

“com\refactoring\source\Invert.java”文件内容如下：

```
class Invert {
    private static double a;
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) {
            System.out.println("Hello World!");
        }
    }
    public static boolean checkCondition() {
        if (a > 15 && a < 100) {
            a = 5;
            return true;
        }
        return false;
    }
}
```

重构后

“com\refactoring\source\Invert.java”文件内容如下：

```
class Invert {
    private static double a;
    public static void main(String[] args) {
        boolean condition = false;
        if (!condition) {
            System.out.println("Hello World!");
        }
    }
    public static boolean checkCondition() {
        if (a > 15 && a < 100) {
            a = 5;
            return false;
```



```
        }
        return true;
    }
}
```

9.5.7 用委托替换继承

通过此重构，可以从继承层次结构中删除类，同时保留父类的功能。在重构过程中，会创建一个私有内部类来继承以前的超类或接口。通过新创建的内部类调用父类的选定方法。

执行重构

步骤1 在代码编辑器中，选择要重构的类，并将光标放置在要从其继承层次结构中删除继承的类中。

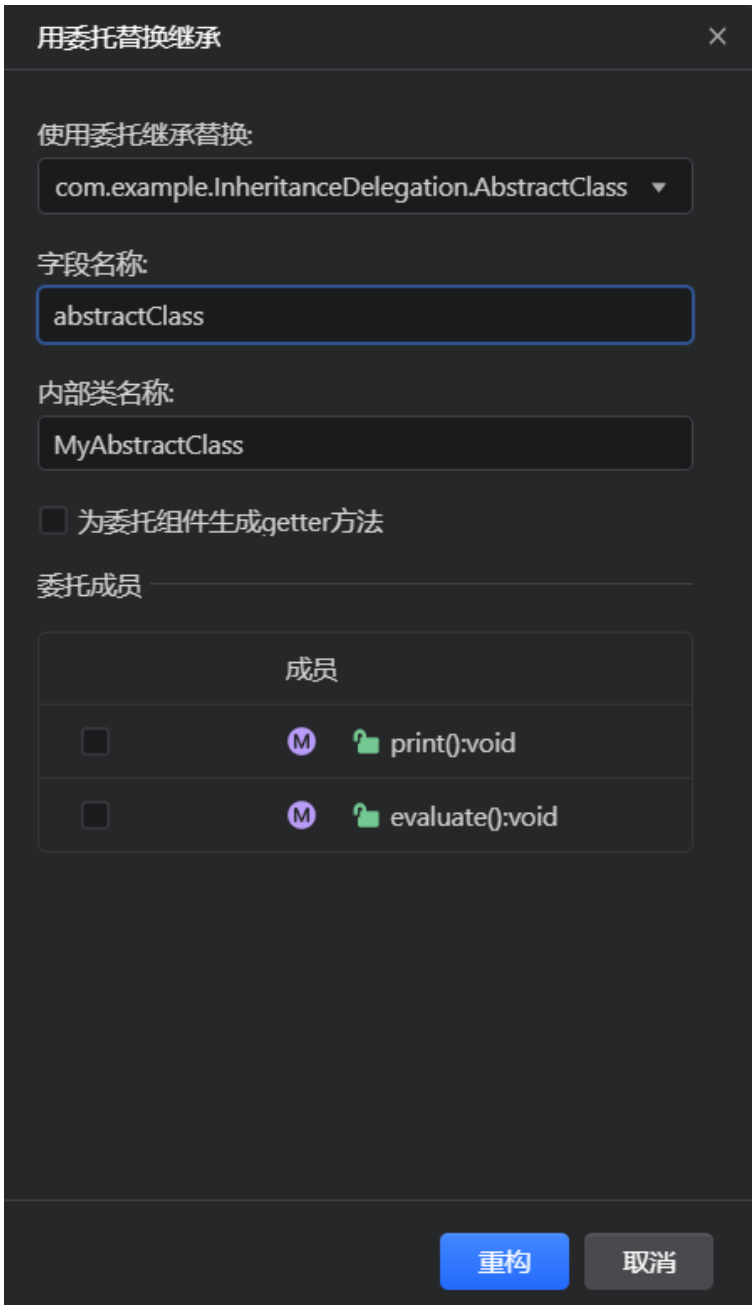
步骤2 单击右键展示上下文菜单，选择“重构 > 用委托替换继承”。

步骤3 在打开的“用委托替换继承”对话框中选择类，其中的继承将通过内部类替换为委托。提供重构选项：

- 在**“字段名称”**字段中，指定新创建的内部类的字段名称。
- 在**“内部类名称”**字段中，指定新创建的内部类的名称。
- 选中为**“委托组件生成Getter方法”**复选框，为新创建的内部类创建getter方法。
- 在**“委托成员”**选项中，选择要通过新创建的内部类委托的父类的成员。

  如下图所示：

图 9-75 用委托替换继承



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，用委托替换**InnerClass**类的继承，重构后将创建一个新的内部**MyAbstractClass**类，并通过**MyAbstractClass**调用**print**和**evaluate**方法。

重构前

“com\refactoring\source\InheritanceDelegation.java”文件内容如下：

```
class InheritanceDelegation {
```



```
    public static void main(String[] args) {
        InnerClass innerClass = new InnerClass();
        print(innerClass);
    }

    private static void print(InnerClass innerClass) {
        innerClass.print();
    }

    private static class InnerClass extends AbstractClass {
        public void evaluate() {
        }
    }

    private abstract static class AbstractClass {
        public void print() {
            System.out.println("Hello World!");
        }

        public abstract void evaluate();
    }
}
```

重构后

“com\refactoring\source\InheritanceDelegation.java”文件内容如下：

```
class InheritanceDelegation {

    public static void main(String[] args) {
        InnerClass innerClass = new InnerClass();
        print(innerClass);
    }

    private static void print(InnerClass innerClass) {
        innerClass.print();
    }

    private static class InnerClass {
        private final MyAbstractClass abstractClass = new MyAbstractClass();

        public AbstractClass getAbstractClass() {
            return abstractClass;
        }

        public void print() {
            abstractClass.print();
        }

        public void evaluate() {
            abstractClass.evaluate();
        }

        private class MyAbstractClass extends AbstractClass {
            public void evaluate() {
            }
        }
    }

    private abstract static class AbstractClass {
        public void print() {
            System.out.println("Hello World!");
        }

        public abstract void evaluate();
    }
}
```


9.5.8 用工厂方法替换构造函数

此重构允许用返回类实例的工厂方法替换类构造函数。

执行重构

步骤1 在代码编辑器中，将光标放置在要用工厂方法替换的类构造函数上。

步骤2 单击右键展示上下文菜单，选择“重构 > 用工厂方法替换构造函数”。

步骤3 在打开的“用工厂方法替换构造函数”对话框中，提供要创建的工厂方法的名称及其包含类。如下图所示：

图 9-76 用工厂方法替换构造函数

步骤4 单击**“重构”**以应用重构。

----结束

示例

作为示例，将**InnerClass**类构造函数替换为带有**newInnerClass**的工厂方法。

重构前

“com\refactoring\source\ReplaceConstructor.java”文件内容如下：

```
class ReplaceConstructor {
  public static void main(String[] args) {
    new InnerClass("Hello", "World").print();
  }

  private static class InnerClass {
    private String message;

    public InnerClass(String hello, String world) {
```

```
      message = hello + ", " + world;
    }

    public void print() {
      System.out.println(message);
    }
  }
}
```

重构后

“com\refactoring\source\ReplaceConstructor.java”文件内容如下：

```
class ReplaceConstructor {
  public static void main(String[] args) {
    InnerClass.newInnerClass("Hello", "World").print();
  }

  private static class InnerClass {
    private String message;

    private InnerClass(String hello, String world) {
      message = hello + ", " + world;
    }

    public static InnerClass newInnerClass(String hello, String world) {
      return new InnerClass(hello, world);
    }

    public void print() {
      System.out.println(message);
    }
  }
}
```

9.5.9 用构建器替换构造函数

通过此重构，可以将类构造函数的用法替换为对构建器类的引用。

执行重构

步骤1 在代码编辑器中，将光标放置在要将其调用替换为对构建器类的引用的类构造器的声明上。

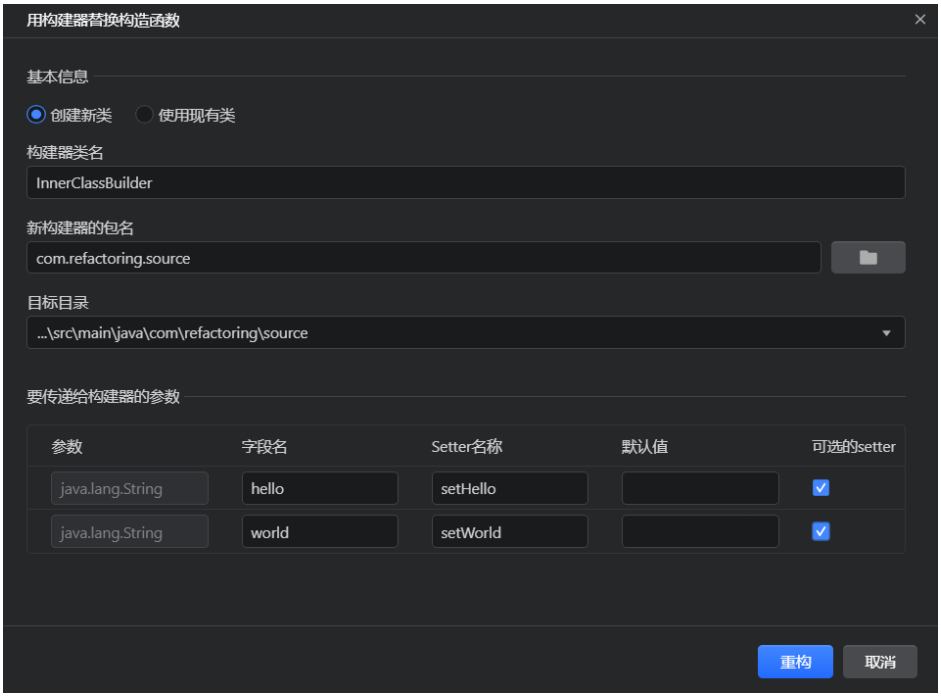
步骤2 单击右键展示上下文菜单，选择“重构 > 用构建器替换构造函数”。

步骤3 在打开的“用构建器替换构造函数”对话框中，提供重构参数。

- 选择是创建新的生成器类还是使用现有的生成器类。
- 提供构建器类名，如果创建新类，则提供其包名和目标目录。
- 在**“要传递给构建器的参数”**选项，配置作为生成器类字段传递的构造函数参数。如果指定的字段的默认值与作为构造函数参数传递的值匹配，则可以选中“可选的Setter”复选框跳过此类参数的setter方法。否则，如果未选中复选框，则始终调用字段设置器。

  如下图所示：

图 9-77 用构建器替换构造函数



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，用对新创建的**InnerClass**构建器类的引用替换**InnerClassBuilder**调用。请注意，由于将“**Hello**”和“**World**”作为**Hello**和**World**生成器类字段的默认值，因此可以使用**可选的Setter**选项，以便在**InnerClassBuilder**调用中跳过对**setHello**和**setWorld**的调用。

重构前

“com\refactoring\source\ReplaceConstructor.java”文件内容如下：

```
class ReplaceConstructor {
  public static void main(String[] args) {
    new InnerClass("Hello", "World").print();
  }

  private static class InnerClass {
    private String message;

    public InnerClass(String hello, String world) {
      message = hello + ", " + world;
    }

    public void print() {
      System.out.println(message);
    }
  }
}
```

重构后

“com\refactoring\source\ReplaceConstructor.java”文件内容如下：

```
class ReplaceConstructor1 {
    public static void main(String[] args) {
        new InnerClassBuilder().setHello("Hello").setWorld("World").createInnerClass().print();
    }
    static class InnerClass {
        private String message;
        public InnerClass(String hello, String world) {
            message = hello + ", " + world;
        }
        public void print() {
            System.out.println(message);
        }
    }
}
```

同时，生成“com\refactoring\source\InnerClassBuilder.java”文件，文件内容如下：

```
package com.refactoring.source;
public class InnerClassBuilder {
    private String hello;
    private String world;
    public InnerClassBuilder setHello(String hello) {
        this.hello = hello;
        return this;
    }
    public InnerClassBuilder setWorld(String world) {
        this.world = world;
        return this;
    }
    public ReplaceConstructor1.InnerClass createInnerClass() {
        return new ReplaceConstructor1.InnerClass(hello, world);
    }
}
```

9.5.10 封装字段

此重构允许限制类字段的可见性，并提供用于访问它们的getter和setter方法。

执行重构

步骤1 在代码编辑器中，将光标放置在要封装字段的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 封装字段”。

步骤3 在打开的“封装字段”对话框中，提供重构选项。

- 选择要封装的字段，并（可选）自定义getter和setter方法的名称。
- 在封装区域中，可以选择不同的选项达到同时创建或单独创建getter、setter的目的。例如，选中**“即使字段可访问，也使用访问器”**复选框，以将所有字段引用替换为对相应访问器方法的调用。
- 指定封装字段和访问器的可见性。

图 9-78 封装字段

步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，封装**name**变量，并为其生成getter和setter方法。

重构前

“com\refactoring\source\Person.java”文件内容如下：

```
class Person {
    public String name;

    public static void main(String[] args) {
        Person person = new Person();
        person.name = "John";
        System.out.println(person.name);
    }
}
```

重构后

“com\refactoring\source\Person.java”文件内容如下：


```
class Person {
    private String name;

    public static void main(String[] args) {
        Person person = new Person();
        person.setName("John");
        System.out.println(person.getName());
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

9.5.11 更改方法签名

此重构支持对方法进行以下修改：包括重命名方法、添加异常，以及对方法参数进行增删、重新排序和重命名。

执行重构

步骤1 在代码编辑器中，将光标放置在要更改其签名的方法的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 更改方法签名”或按“Ctrl+F6”。

步骤3 在打开的“更改方法签名”对话框中，提供重构选项。

- 指定方法的可见性、名称和返回类型。
- 在**“参数”**选项卡上，配置方法的参数：指定参数的名称和类型，并使用工具栏按钮添加、删除和重新排序参数。

  说明

  目前CodeArts IDE不支持为参数提供默认值。如果添加参数，则可能需要手动更新方法调用。
- 在**“抛出”**选项卡上，配置方法抛出的异常列表。使用工具栏按钮添加、删除和重新排序功能例外。
- 在**“方法调用”**区域中，选择是修改现有方法或通过重载方法委托。

  如下图所示：

图 9-79 更改方法签名

步骤4 单击“**重构**”以应用重构。

----结束

示例

作为示例，对**myMethod**方法通过添加参数**price**来更改方法的签名，使方法抛出异常**Exception**，并通过重载方法委托它。

重构前

“com\refactoring\source\Example.java”文件内容如下：

```
public class Example {
  public void myMethod(int value) {
  }

  public class AntherClass {
    public void myMethodCall(Example example) {
      example.myMethod(1);
    }
```



```
  }
}
```

重构后

“com\refactoring\source\Example.java”文件内容如下：

```
public class Example {
  public void myMethod(int value) {
    myMethod(value, 0);
  }

  public void myMethod(int value, double price) throws Exception {}

  public class AntherClass {
    public void myMethodCall(Example example) {
      example.myMethod(1);
    }
  }
}
```

9.5.12 更改类签名

此重构允许将类转换为泛型并操作其类型参数。

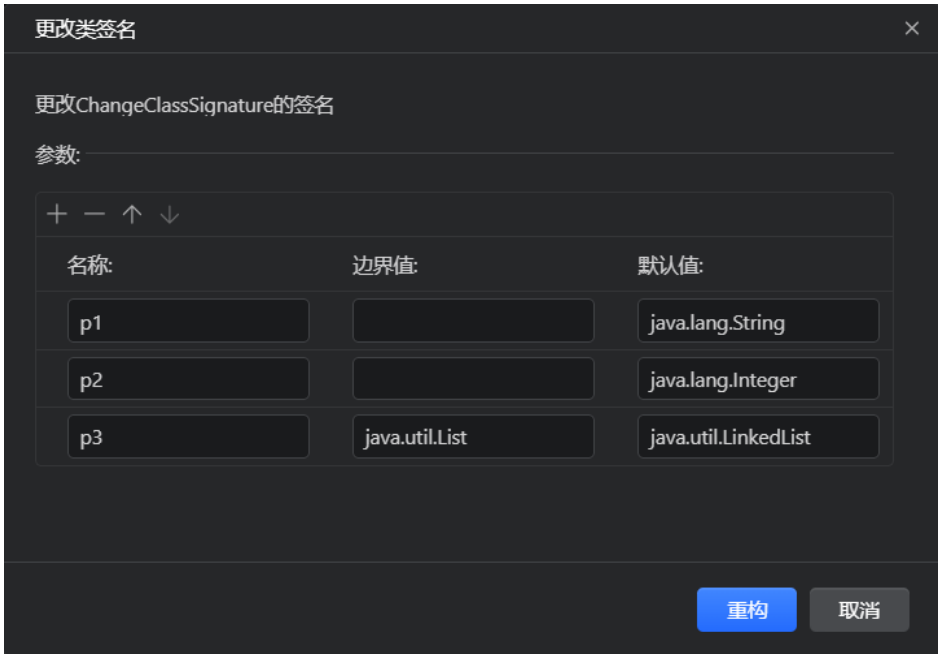
执行重构

步骤1 在代码编辑器中，将光标放置在要更改其签名的类的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 更改类签名”或按“Ctrl+F6”。

步骤3 在打开的“更改类签名”对话框中，配置类参数。使用工具栏按钮添加、删除和重新排序参数。对于每个参数，指定其名称和默认类型。在**“边界值”**字段中，用户可以选择提供限制传递给类型参数的值的有界值。如下图所示：

图 9-80 更改类签名



步骤4 单击**“重构”**以应用重构。

----结束

示例

例如，通过添加三个类型参数来更改类**ChangeClassSignature**的签名：**P1 (String)**， **P2 (Integer)**， 和 **P3 (LinkedList)**，其边界为**List**。

重构前

“com\refactoring\source\ChangeClassSignature.java”文件内容如下：

```
class ChangeClassSignature {

   public class MyOtherClass {
      ChangeClassSignature myClass;

      void myMethod(ChangeClassSignature myClass) {
      }
   }
}
```

重构后

“com\refactoring\source\ChangeClassSignature.java”文件内容如下：

```
class ChangeClassSignature<P1, P2, P3 extends List> {

   public class MyOtherClass {
      ChangeClassSignature<String, Integer, LinkedList> myClass;

      void myMethod(ChangeClassSignature<String, Integer, LinkedList> myClass) {
      }
   }
}
```

9.5.13 将匿名类转换为内部类

此重构允许将匿名类转换为重命名的内部类。

执行重构

步骤1 在编辑器中，将光标放置在要转换为内部类的匿名类表达式中的任何位置。

步骤2 单击右键展示上下文菜单，选择“重构 > 转换匿名类为内部类”。

步骤3 在打开的“转换匿名类为内部类”对话框中，提供重构参数。

- 提供内部类的名称并选择是否将其创建为静态类。
- 在**“构造函数参数”**区域中，提供要用作类构造函数参数的变量的名称。使用工具栏按钮对参数重新排序。

  如下图所示：

图 9-81 转换匿名类为内部类

步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，将用作**方法**返回值的匿名类表达式转换为内部静态类**MyTestClass**。

重构前

“com\refactoring\source\AnonymousToInner.java”文件内容如下：

```
class AnonymousToInner {
  public TestClass method() {
    final int i = 0;
    final String str = "string";
    return new TestClass() {
      public String str () {
        return str;
      }
      public int publicMethod() {
        return i;
      }
    };
  }
}
```

同时，在“com\refactoring\source\TestClass.java”文件中定义“TestClass”类，文件内容如下：

```
package com.refactoring.source;

public class TestClass {
}
```

重构后

“com\refactoring\source\TestClass.java”文件中定义“TestClass”类内容不变，“com\refactoring\source\AnonymousToInner.java”文件内容如下：

```
class AnonymousToInner {
  public TestClass method() {
    final int i = 0;
    final String str = "string";
    return new MyTestClass(str, i);
  }

  private static class MyTestClass extends TestClass {
    private final String str;
    private final int i;

    public MyTestClass(String str, int i) {
      this.str = str;
      this.i = i;
    }

    public String str () {
      return str;
    }

    public int publicMethod() {
      return i;
    }
  }
}
```

9.5.14 尽可能使用接口

此重构允许将从基类/接口派生的指定方法的执行委托给实现同一接口的父类或内部类的实例。

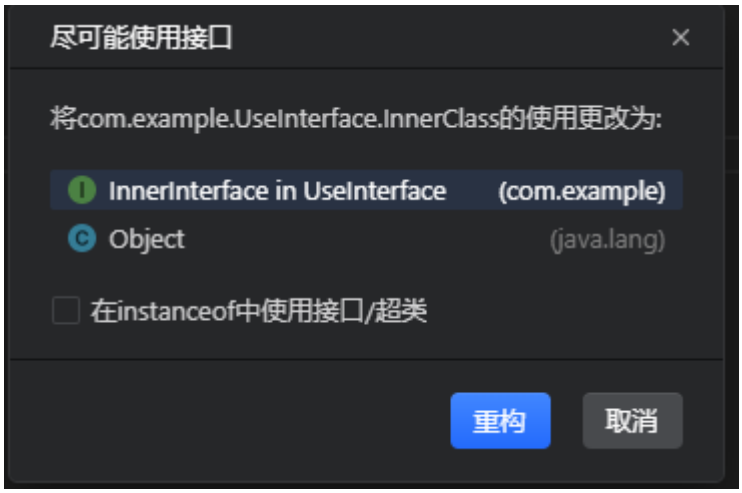
执行重构

步骤1 在代码编辑器中，将光标放在应通过父类/接口委托其方法的类的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 尽可能使用接口”。

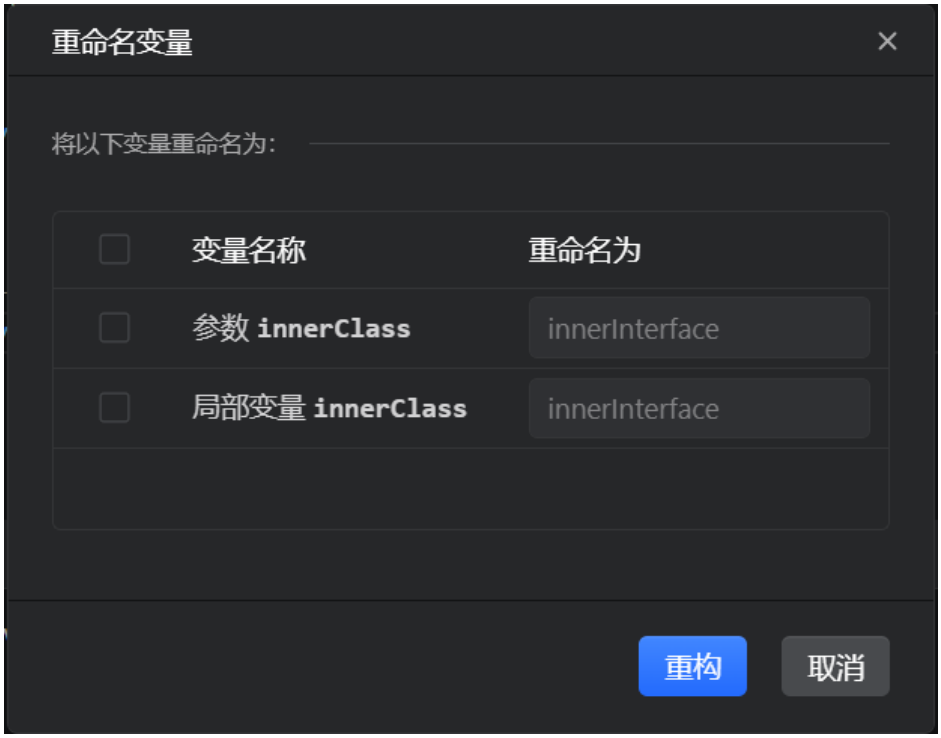
步骤3 在打开的“尽可能使用接口”对话框中，选择应替换当前类用法的父类/接口。想要同时替换当前类在**instanceof**语句中的用法，请选中在**“在instanceof中使用接口/超类”**复选框。如下图所示：

图 9-82 尽可能使用接口



CodeArts IDE将自动重命名变量，以匹配重构引入的更改。如有必要，请在“重命名变量”对话框中提供替代名称。如下图所示：

图 9-83 重命名变量



步骤4 单击**“重构”**以应用重构。

----结束

示例

例如，将**print**方法的使用从类**InnerClass**委托给它实现的接口**InnerInterface**。

重构前

“com\refactoring\source\UseInterface.java”文件内容如下：

```
class UseInterface {

    public static void main(String[] args) {
        InnerClass innerClass = new InnerClass();
        print(innerClass);
    }

    private static void print(InnerClass innerClass) {
        innerClass.print();
    }

    private static class InnerClass implements InnerInterface {
        @Override
        public void print() {
            System.out.println("Hello World!");
        }
    }

    private static interface InnerInterface{
        void print();
    }
}
```

重构后

“com\refactoring\source\UseInterface.java”文件内容如下：

```
class UseInterface {

    public static void main(String[] args) {
        InnerInterface innerInterface = new InnerClass();
        print(innerInterface);
    }

    private static void print(InnerInterface innerInterface) {
        innerInterface.print();
    }

    private static class InnerClass implements InnerInterface {
        @Override
        public void print() {
            System.out.println("Hello World!");
        }
    }

    private static interface InnerInterface{
        void print();
    }
}
```

9.5.15 类型迁移

此重构允许更改类成员、局部变量、参数、方法返回值等类型。还可以在数组、集合之间转换变量或方法返回值的类型。

执行重构

步骤1 在代码编辑器中，将光标放置在要迁移的类型上。

步骤2 单击右键展示上下文菜单，选择“重构 > 类型迁移”。

步骤3 在打开的“类型迁移”对话框中，提供要迁移到的类型。在**“选择可能发生变更签名的范围”**列表中，指定用于查找使用实例的新类型和范围：整个项目或仅根目录文件（即不包括库和SDK）。如下图所示：

图 9-84 类型迁移

步骤4 单击**“重构”**以应用重构。

----结束

示例

例如，将**myResult**字段的类型从集合**ArrayList<String>**迁移到数组**String[]**。

重构前

“com\refactoring\source\TypeMigration.java”文件内容如下：

```
class TypeMigration {

    private ArrayList<String> myResult;

    public String[] getResult() {
        return myResult.toArray(new String[myResult.size()]);
    }
}
```

重构后

“com\refactoring\source\TypeMigration.java”文件内容如下：

```
class TypeMigration {

    private String[] myResult;

    public String[] getResult() {
        return myResult;
    }
}
```


9.5.16 将原始类型转换为泛型

此重构允许为每个原始类型创建安全且一致的参数类型，将不使用的泛型代码转换为泛型感知代码。

执行重构

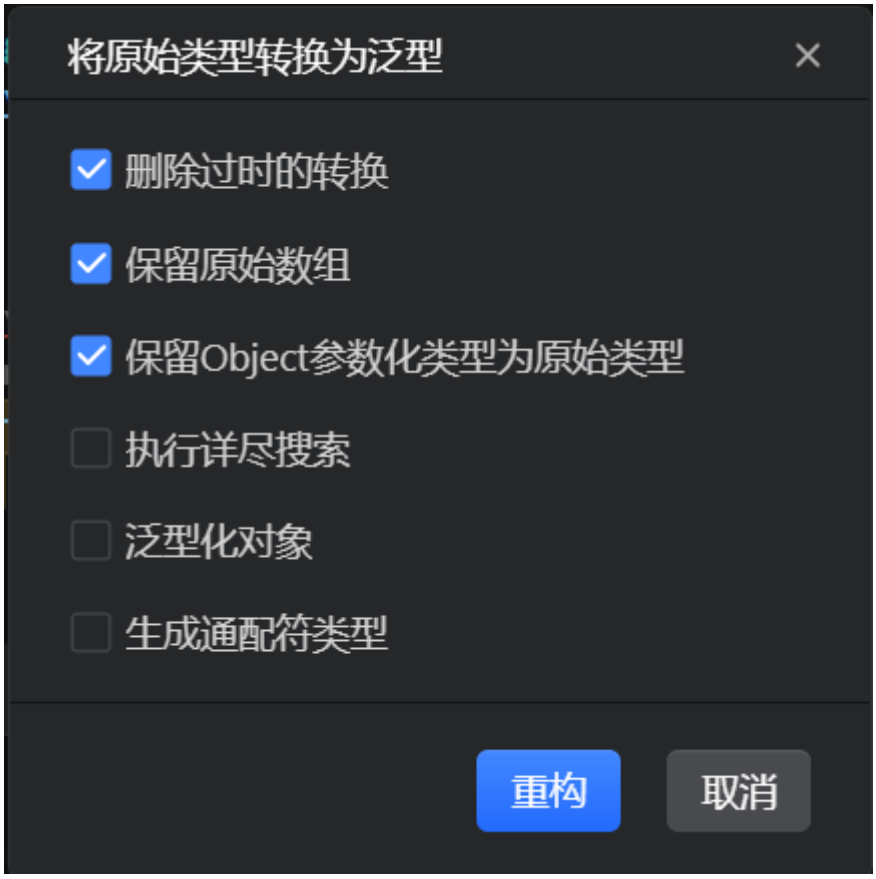
步骤1 选择要应用重构的实体。

步骤2 单击右键展示上下文菜单，选择“重构 > 将原始类型转换为泛型”。

步骤3 在打开的“将原始类型转换为泛型”对话框中，提供重构选项。

- **“删除过时的转换”**：如果选中，CodeArts IDE将分析参数强制转换案例是否会被重构而更改。如果生成的参数类型与过期的参数类型相似，则将删除强制转换语句。
- **“保留原始数组”**：如果选中，数组不会更改为具有参数化类型的数组。否则，数组将转换为参数化类型。清除此复选框可能会有风险，并导致无法编译的代码。
- **“保留Object参数化类型为原始类型”**： 如果选择了具有**java.lang.Object**作为参数的对象，它们将被设置为原始类型。
- **“执行详尽搜索”**： 如果选中，则在所有节点上执行搜索。
- **“泛型化对象”**： 如果选中，**java.lang.Object**对象将转换为它们实际使用的类型。
- **“生成通配符类型”：** 如果选中此选项，则尽可能生成通配符类型（即**List<? extends String>**等表达式）。

图 9-85 将原始类型转换为泛型



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，生成**List**和**LinkedList**类型。

重构前

“com\refactoring\source\ConvertTypes.java”文件内容如下：

```
package com.refactoring.source;
import java.util.LinkedList;
import java.util.List;

public class ConvertTypes {
    public void method() {
        List list = new LinkedList();
        list.add("string");
    }
}
```

重构后

“com\refactoring\source\ConvertTypes.java”文件内容如下：

```
package com.refactoring.source;
import java.util.LinkedList;
import java.util.List;
```



```
public class ConvertTypes {
    public void method() {
        List<String> list = new LinkedList<String>();
        list.add("string");
    }
}
```

9.5.17 转换为实例方法

此重构允许将类的静态方法转换为类实例的非静态方法。

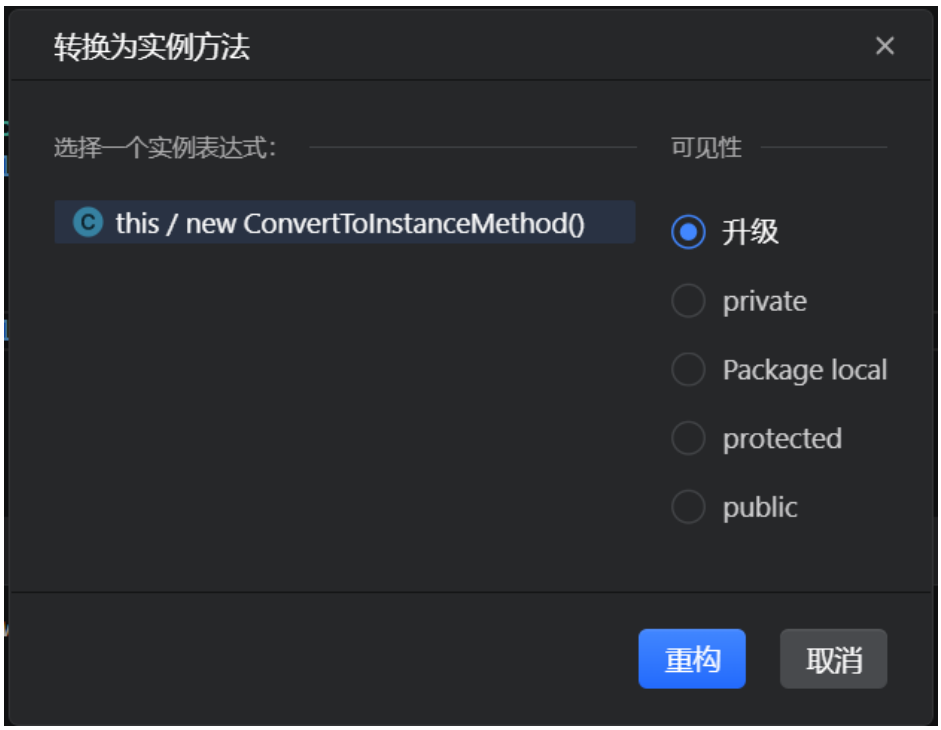
执行重构

步骤1 在代码编辑器中，将光标放置在要转换为实例方法的静态方法的声明上。

步骤2 单击右键展示上下文菜单，选择“重构 > 转换为实例方法”。

步骤3 在打开的“转换为实例方法”对话框中，选择方法所属的类。方法中该类的所有用法都将替换为this。如有必要，请更改转换方法的可见性修饰符。如下图所示：

图 9-86 转换为实例方法



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，将静态方法**sayHello**转换为实例方法。

重构前

“com\refactoring\source\ConvertToInstanceMethod.java”文件内容如下：

```
class ConvertToInstanceMethod {
    public static void main(String[] args) {
```

```
        sayHello();
    }

    public static void sayHello() {
        System.out.println("Hello World");
    }
}
```

重构后

“com\refactoring\source\ConvertToInstanceMethod.java”文件内容如下：

```
class ConvertToInstanceMethod {
    public static void main(String[] args) {
        new ConvertToInstanceMethod().sayHello();
    }

    public void sayHello() {
        System.out.println("Hello World");
    }
}
```

9.5.18 移除中间人

通过此重构，可以将类中的委托方法的调用替换为直接对委托字段的调用，并删除不再使用的委托方法。

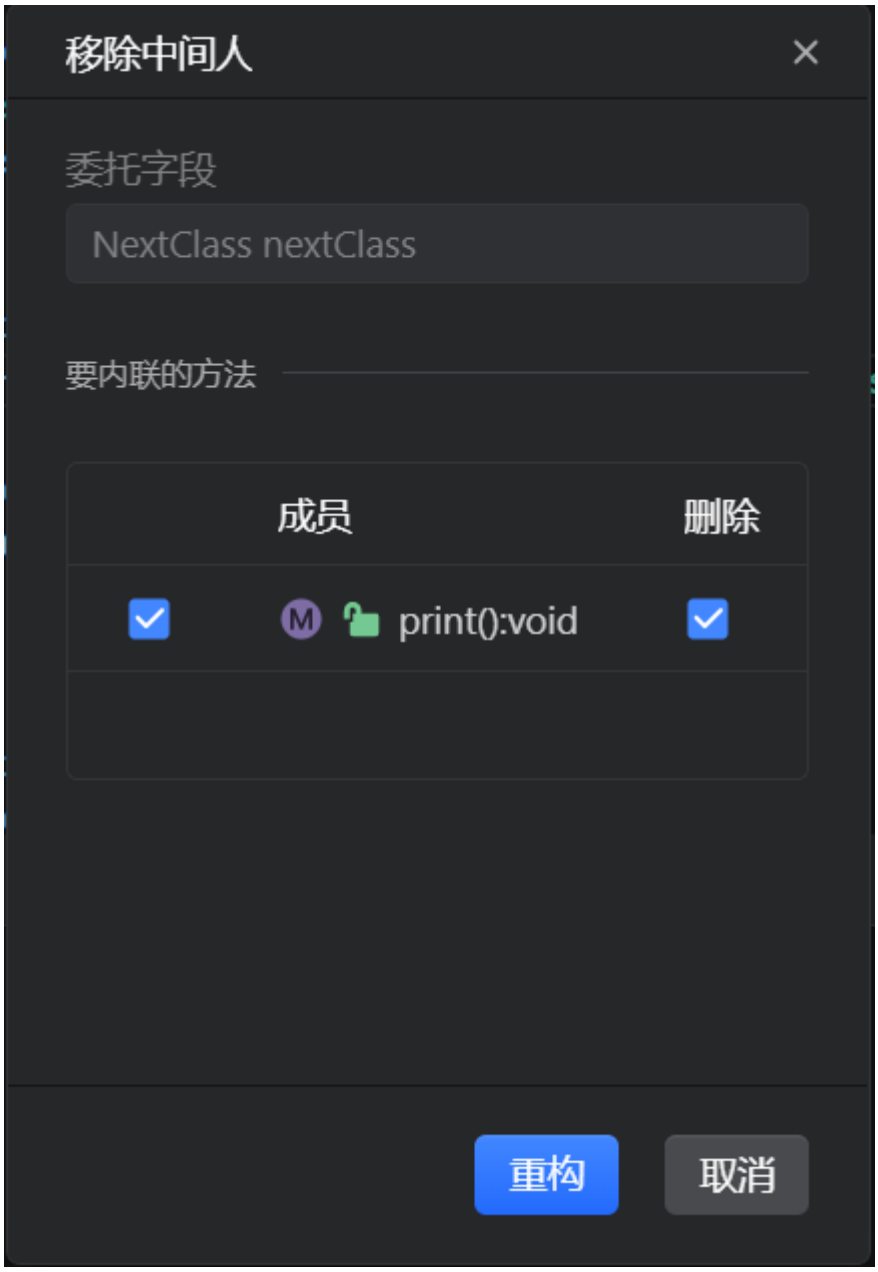
执行重构

步骤1 在代码编辑器中，将光标放置在其声明中委托字段的名称上。

步骤2 单击右键展示上下文菜单，选择“重构 > 移除中间人”。

步骤3 在打开的“移除中间人”对话框中，选择要内联的委托方法。要删除方法，请选中该方法旁边的复选框。如下图所示：

图 9-87 移除中间人



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，从类**InnerClass**中删除**print**委托方法，将其替换为对委托字段**nextClass**的调用。

重构前

“com\refactoring\source\Middleman.java”文件内容如下：

```
class Middleman {
```



```
    public static void main(String[] args) {
        InnerClass innerClass = new InnerClass();
        innerClass.print();
    }

    private static class InnerClass {
        private final NextClass nextClass = new NextClass();

        public void print() {
            nextClass.print();
        }
    }

    private static class NextClass {
        public void print() {
            System.out.println("Hello World!");
        }
    }
}
```

重构后

“com\refactoring\source\Middleman.java”文件内容如下：

```
class Middleman {

    public static void main(String[] args) {
        InnerClass innerClass = new InnerClass();
        innerClass.getNextClass().print();
    }

    private static class InnerClass {
        private final NextClass nextClass = new NextClass();

        public NextClass getNextClass() {
            return nextClass;
        }
    }

    private static class NextClass {
        public void print() {
            System.out.println("Hello World!");
        }
    }
}
```

9.5.19 安全删除

此重构允许安全地删除文件和代码符号。CodeArts IDE将验证所有受影响的用法，并相应地调整代码。

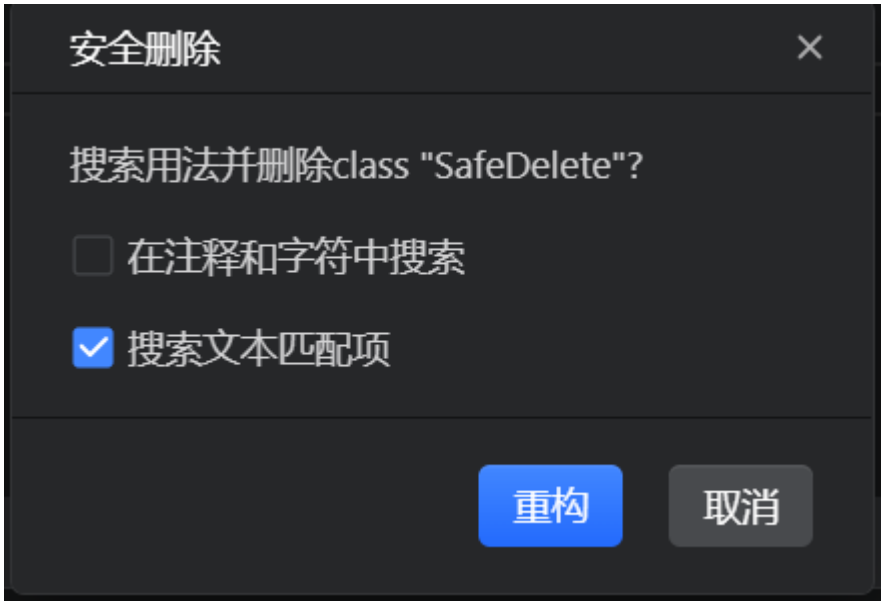
执行重构

步骤1 选择要应用重构的实体（资源管理器中的文件或代码编辑器中的符号）。

步骤2 单击右键展示上下文菜单，选择“重构 > 安全删除”。

步骤3 在打开的“安全删除”对话框中，选择CodeArts IDE是否要搜索代码中被选定符号的引用。如下图所示：

图 9-88 安全删除



步骤4 单击“**重构**”以应用重构。

----结束

示例

例如，在整个方法调用层次结构中删除未使用的参数**i**。

重构前

“com\refactoring\source\SafeDelete.java”文件内容如下：

```
class SafeDelete {
    private void foo(int i) { bar(i);}
    private void bar(int i) { baz(i);}
    private void baz(int i) { }
}
```

重构后

“com\refactoring\source\SafeDelete.java”文件内容如下：

```
class SafeDelete {
    private void foo() { bar();}
    private void bar() { baz();}
    private void baz() { }
}
```

9.6 配置和运行 Java 项目单元测试

9.6.1 集成测试框架

CodeArts IDE提供了**JUnit**和**TestNG**测试框架的集成，让用户轻松运行和调试Java测试用例。在开始之前，请确保项目已设置了JDK，详情请参考**在CodeArts IDE创建Java项目需选择JDK**。

JUnit 3/4

对于Maven项目，请在**pom.xml**中添加以下配置，以JUnit 4.11版本为例：

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>
```

对于Gradle项目，请将以下行添加到**build.gradle**中：

```
dependencies {
    testImplementation 'junit:junit:4.11'
}
test {
    useJUnitPlatform()
}
```

JUnit 5

对于Maven项目，请在**pom.xml**中添加以下配置，以JUnit 5.8.1版本为例：

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
```

对于Gradle项目，请将以下行添加到**build.gradle**中：

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.1'
}
test {
    useJUnitPlatform()
}
```

TestNG

对于Maven项目，请在**pom.xml**中添加以下配置，以testng 7.7.0版本为例：

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.7.0</version>
    <scope>test</scope>
</dependency>
```

对于Gradle项目，请将以下行添加到**build.gradle**中：

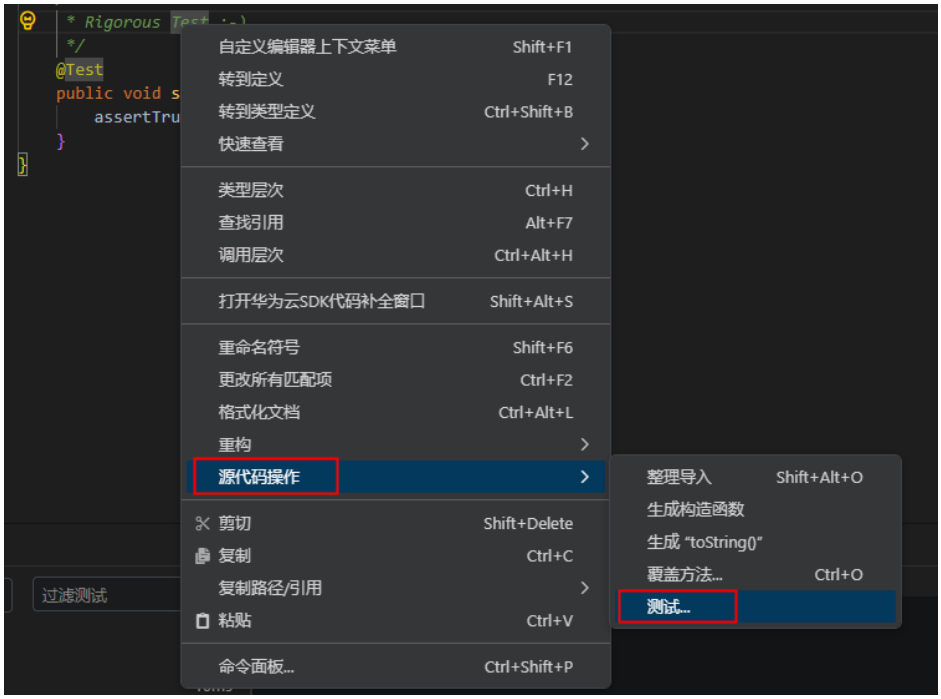
```
dependencies {
    testImplementation 'org.testng:testng:7.7.0'
}
```

9.6.2 创建测试用例

CodeArts IDE提供了专门的“测试”入口，帮助用户创建测试用例。

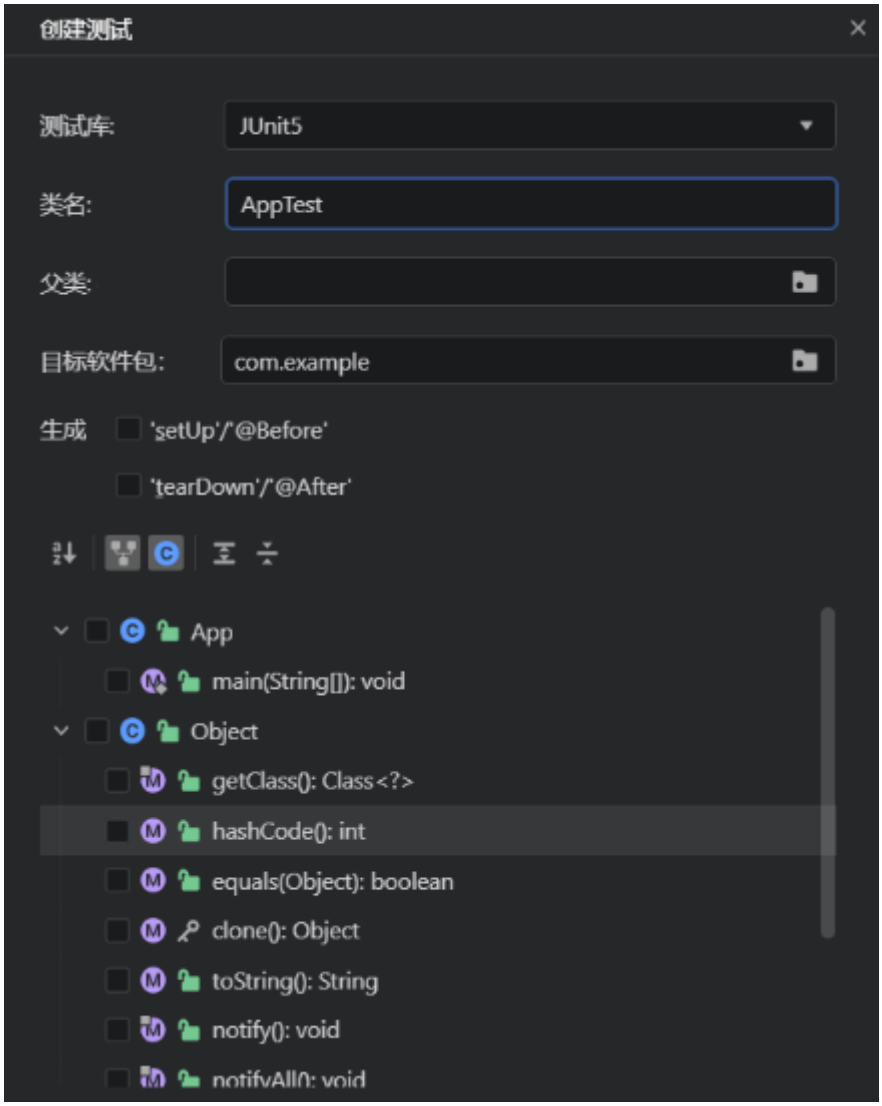
步骤1 在代码编辑器中，右键单击要为其创建测试的类的声明处，然后从右键菜单中选择“源代码操作 > 测试”。或在“命令面板”（“Ctrl+Shift+P”或双击“Ctrl”）中搜索并运行**源代码操作**命令后单击“**测试**”。

图 9-89 “测试”入口



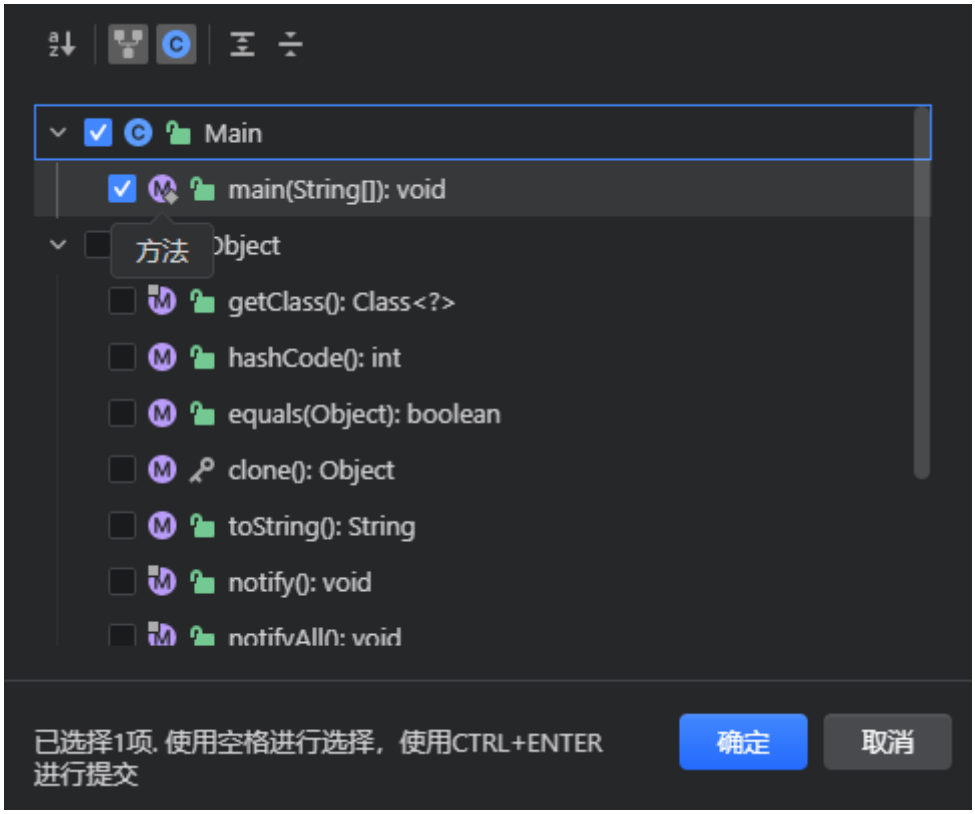
步骤2 在打开的“创建测试”对话框中，提供要创建的测试类的详细信息。

图 9-90 创建测试



- **测试库：**选择要使用的测试框架。
- **类名：**提供测试类的名称，或保留默认值。
- **父类**：对于JUnit3，在“**父类**”字段中提供**junit.framework.TestCase**。对于其他框架，请将该字段留空。
- **目标软件包：**在“目标软件包”中，选择测试类存储在其中的包。
- **生成**：如有必要，请选择“**'setUp'/'@Before'**”或“**'tearDown'/'@After'**”复选框，以将测试装置和注释的存根方法包括到生成的测试类中。
- 如果需要查看并选择从超类中继承的方法，请单击“**显示继承成员**”  按钮，勾选显示继承的“**方法**”复选框。

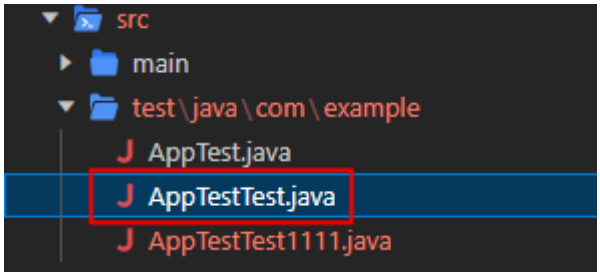
图 9-91 勾选显示继承的“方法”



- 在下方**成员**区域中，选中要为其创建测试方法的方法旁边的复选框。

步骤3 单击“**确定**”即可完成测试用例的创建。

图 9-92 测试用例创建完成



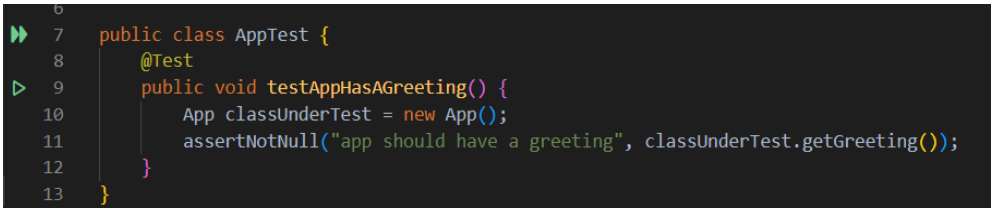
----结束

9.6.3 运行与调试测试用例

CodeArts IDE提供了多个选项来运行和调试测试：

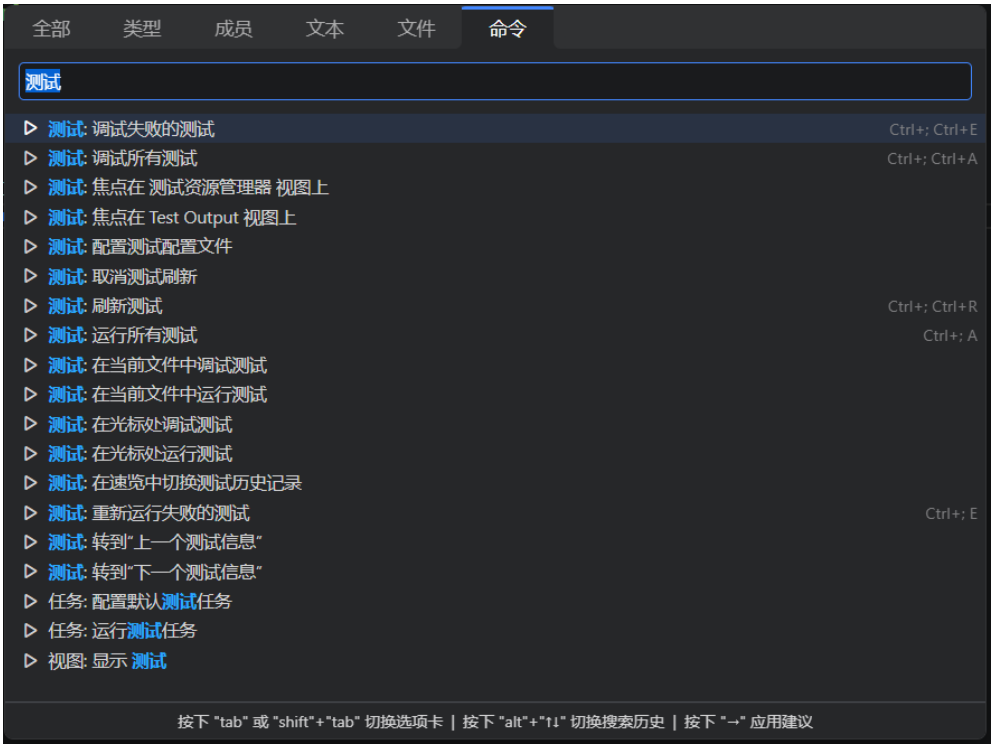
- 在测试类的代码编辑器中，单击测试类声明旁边的“**运行/调试测试用例**”按钮（）（运行或调试类中的所有测试），或者单击单个测试用例的方法声明旁边的运行按钮（）（仅运行或调试单个测试）。

图 9-93 运行/调试测试用例



- 参考**运行Java测试视图中的测试用例**管理和运行测试。
- 参考**运行JUnit测试的启动配置**，使用“运行/调试启动配置”来运行Junit测试。
- 在“命令”面板（按“Ctrl+Shift+P”或双击“Ctrl”）中，搜索**测试**并使用与测试相关的命令，如**“在当前文件中运行测试”**或**“在光标处运行测试”**。

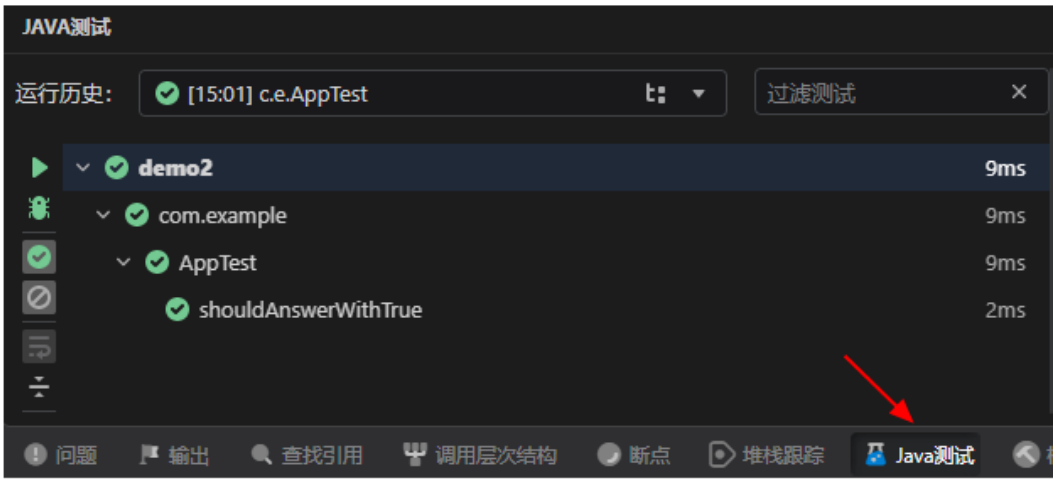
图 9-94 通过“命令”面板运行与调试测试



9.6.4 运行 Java 测试视图中的测试用例

“Java测试”视图列出了项目中的所有测试用例，让用户可以运行它们并检查结果。要打开“Java测试”视图，请单击CodeArts IDE底部面板中的**“Java测试”**，如下图所示。

图 9-95 打开“Java 测试”视图



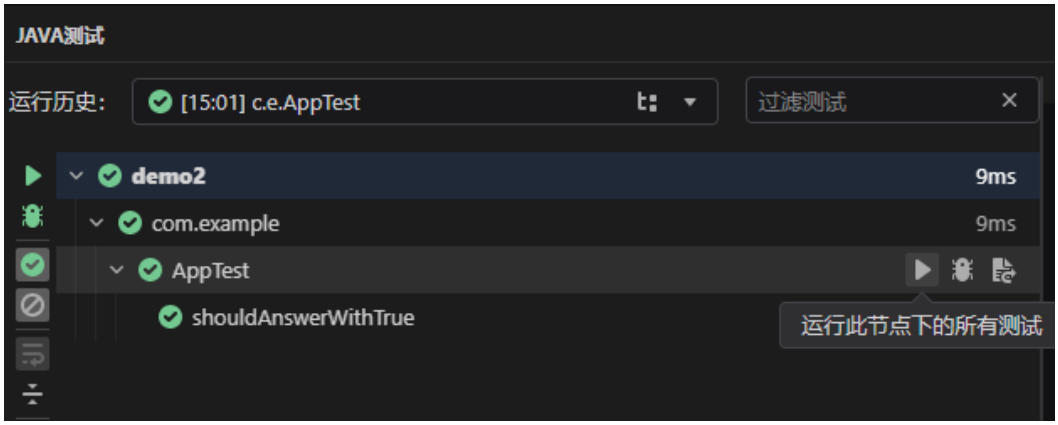
“Java测试”视图提供了多种功能。测试视图顶部的“过滤测试”输入框允许按关键字筛选测试用例列表。

请参考以下步骤运行和调试测试。

步骤1 将鼠标悬停在与包含要运行的测试的包、类或方法对应的树节点上。

步骤2 单击所在行右侧的**“运行此节点下的所有测试”**按钮（ ）运行测试，或单击**“调试此节点下的所有测试”**按钮（ ）调试测试。

图 9-96 “运行此节点下的所有测试按钮”位置示意图



步骤3 要运行或调试所有可用的测试，请单击“JAVA测试”视图左侧工具栏上的**“运行视图中所有测试”**（ ）或**“调试视图中所有测试”**（ ）按钮。

----结束

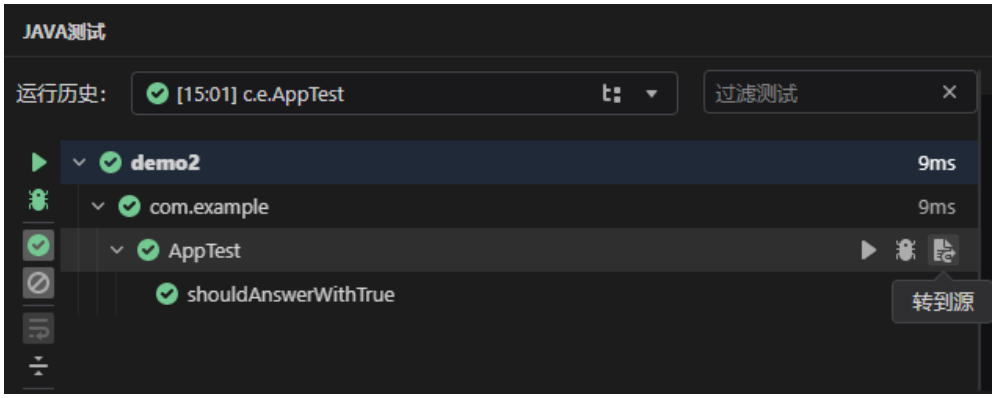
导航到测试类或方法

在测试视图中，可以直接从测试用例导航到相应的测试类或方法。

- **双击树节点**：直接双击你想要导航到的测试类或测试方法的树节点。将打开相应的文件，并将光标定位到该类或方法的位置。

- **使用“转到源”按钮**：将鼠标悬停在你想要导航的测试类或测试方法的树节点上，并单击**“转到源”**按钮（ ），将打开相应的文件，并将光标定位到该类或方法的位置。

  图 9-97 “转到源”按钮位置示意图

10 使用 CodeArts IDE for Python

10.1 在 CodeArts IDE for Python 创建 Python 项目

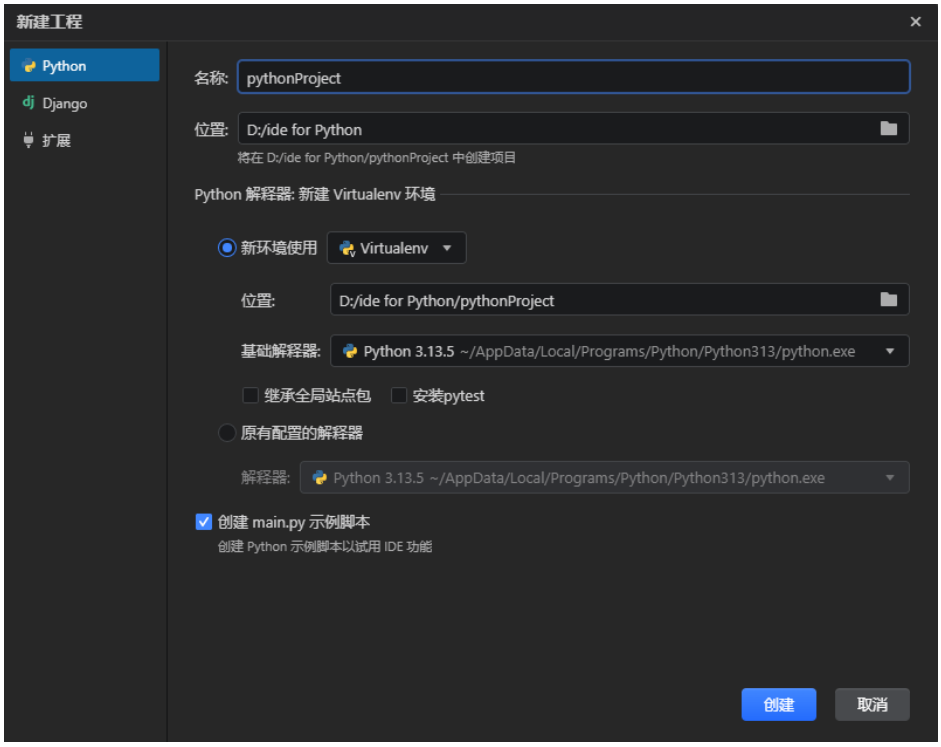
CodeArts IDE附带了Python扩展，它提供了广泛的Python语言支持。Python扩展可与各种Python解释器配合使用，并利用CodeArts IDE功能，如代码完成、验证、调试和单元测试，以及在Python环境之间轻松切换的能力。

本章节简要概述了使用CodeArts IDE for Python创建一个Python项目。具体操作如下：

步骤1 在主菜单中，选择“**文件 > 新建 > 工程**”。

步骤2 在打开的“**新建工程**”对话框中，从左侧列表选择“**Python**”，填入项目参数。

图 10-1 新建 Python 工程



- 填写项目名称和路径。
- 若要使用已配置好的解释器，勾选“**新环境使用**”。
  - **Virtualenv**：不需要复杂的依赖管理，适合轻量级的项目。
  - **Pipenv**：用来管理依赖和虚拟环境，适合中等规模的项目。
  - **Poetry**：支持复杂的项目管理和依赖解析，适合大型项目和团队协作。

  说明

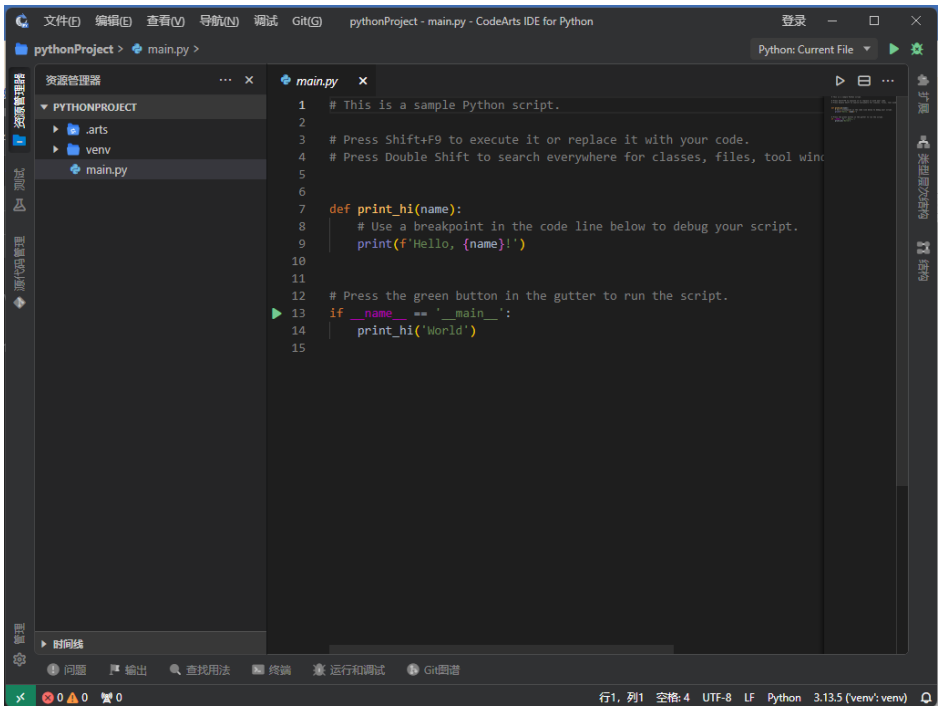
  选择使用“**Virtualenv**”选项，CodeArts IDE会为您创建一个隔离的、特定于项目的Virtualenv Python环境。这样就可以使您在项目级别安装包，不会污染全局Python。
- **位置**：保留创建环境的默认位置。
- **基础解释器**：列表中会选择一个解释器。通常会自动从标准安装位置检测解释器位置并展示在此处。

  要配置新的解释器，从列表中选择所需的环境类型。根据所选环境，提供对应工具的路径、系统范围的基础Python安装路径或要创建的目标虚拟环境路径。
- **继承全局站点包**：勾选“**继承全局站点包**”复选框，以便CodeArts IDE将全局Python中可用的所有包安装到创建的虚拟环境中。
- **安装pytest**：勾选“**安装pytest**”复选框，以便CodeArts IDE安装pytest包。
- **原有配置的解释器**：若要使用另一个项目中已配置的解释器，请选择“**原有配置的解释器**”选项，并从列表中选择所需的解释器。

步骤3 勾选“**创建 main.py 示例脚本**”复选框，以便CodeArts IDE使用示例内容填充项目，让您快速试用IDE的主要功能。

步骤4 单击“**创建**”。CodeArts IDE将创建并打开项目，在项目根目录下的“venv”文件夹中创建一个新环境，并将其设置为项目解释器。

图 10-2 打开 Python 项目



----结束

10.2 配置 Python 工程环境

在Python中，“环境”由解释器和所有已安装的包组成，定义了程序运行的上下文。CodeArts IDE能够自动检测标准位置安装的Python解释器和工作区文件夹中的虚拟环境。

默认情况下，Python解释器在**全局环境**中运行，不会对特定项目进行额外操作。因此，任何安装或卸载的包都会影响全局环境及其上运行的所有程序。随着时间的推移，全局环境可能会变得拥挤，不利于应用程序的测试和管理。

为了避免这种混乱和不便，您可以为每个项目创建一个**虚拟环境**。虚拟环境是一个包含特定Python解释器副本的独立文件夹。安装到虚拟环境中的包仅存在于该文件夹中，不会影响全局Python解释器。当在虚拟环境中运行程序时，它只会使用该环境中安装的特定包。这样可以确保项目的依赖关系清晰且独立，提高开发和测试的效率。

CodeArts IDE会自动在以下位置查找解释器：

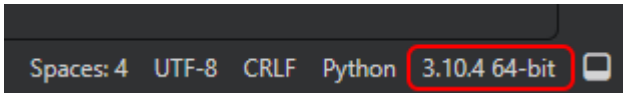
- 标准安装位置，例如 /usr/local/bin 、/usr/sbin、/sbin、c:\\python27、c:\\python36等。
- 工作环境直接位于工作区（项目）文件夹下的虚拟环境。
- 位于由python.venvPath设置标识的文件夹中的虚拟环境，该文件夹可以包含多个虚拟环境。扩展程序会在venvPath的一级子文件夹中查找虚拟环境。

如果CodeArts IDE无法自动定位您的解释器，您可以手动指定它。CodeArts IDE还会加载由python.envFile设置标识的环境变量定义文件。此设置的默认值为${workspaceFolder}/.env。

创建虚拟环境

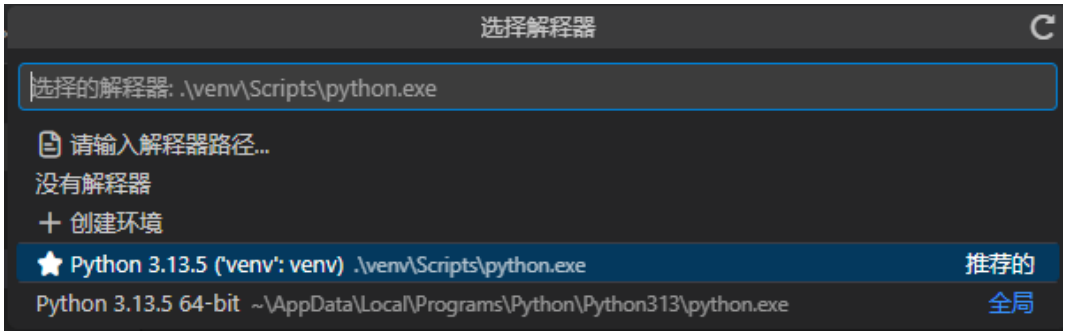
步骤1 在**命令面板**（“Ctrl+Ctrl”/“Ctrl+Shift+P”）中，搜索并运行“**Python: 选择解释器**”命令，或者单击状态栏右侧的按钮，如图10-3所示。

图 10-3 运行解释器



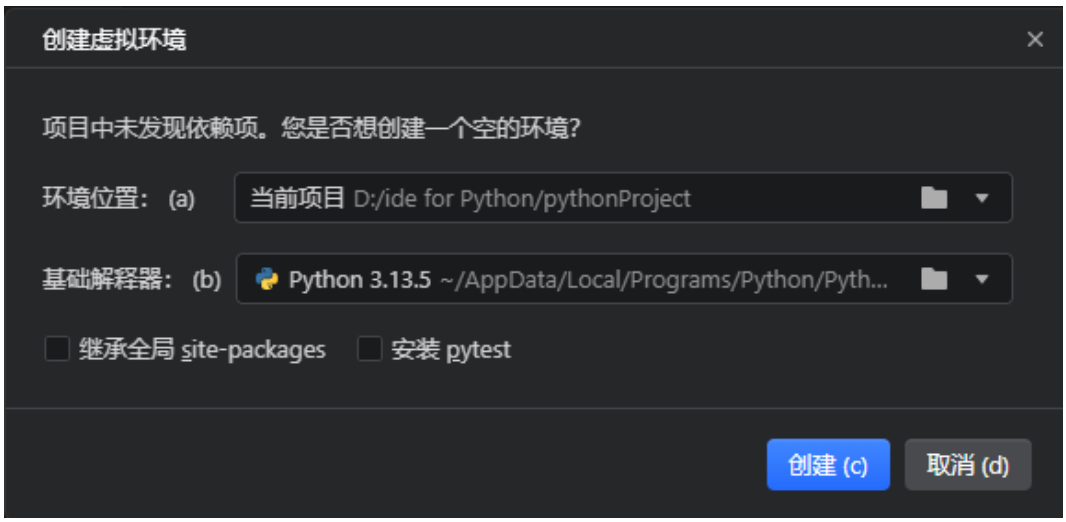
步骤2 在“**选择解释器**”对话框中选择“**创建环境**”。

图 10-4 创建环境



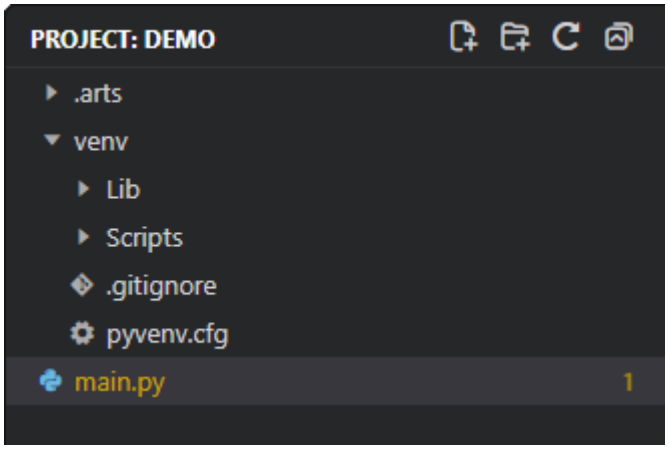
步骤3 在打开的“**创建虚拟环境**”对话框中，选择需要应用的环境创建选项。

图 10-5 创建虚拟环境



1. 在“**环境位置**”中选择环境新建的目标路径，可以选择项目路径、默认虚拟环境路径或是任意路径。单击浏览按钮（ ）手动指定路径。
2. 在“**基础解释器**”列表中选择一个已安装的Python解释器，或者单击浏览按钮（ ）来手动指定解释器路径。
3. 勾选“**继承全局site-packages**”会将全局Python中可用的所有包安装到创建的虚拟环境中。
4. 勾选“**安装pytest**”将会安装pytest包。

步骤4 单击“**创建**”。CodeArts IDE将在指定的文件夹中创建一个新的环境，并将其设置为项目的解释器。该环境将包含全局Python解释器的一个副本，并且特定于项目（安装到该环境中的所有包只应用于项目内部）。



----结束

设置默认的项目环境

如果您想要手动指定首次打开项目时使用的默认解释器，可以使用Python可执行文件的完整路径创建或修改 “python.defaultInterpreterPath” 设置的条目，如：

- Windows
  ```
  {
    "python.defaultInterpreterPath": "c:/python39/python.exe",
  }
  ```
- Linux
  ```
  {
    "python.defaultInterpreterPath": "/home/python39/python",
  }
  ```

您也可以将“python.defaultInterpreterPath”指向虚拟环境，如：

- Windows
  ```
  {
    "python.defaultInterpreterPath": "c:/dev/ala/venv/Scripts/python.exe",
  }
  ```
- Linux
  ```
  {
    "python.defaultInterpreterPath": "/home/abc/dev/ala/venv/bin/python",
  }
  ```

说明

在为工作区选择解释器后，不会应用“python.defaultInterpreterPath”设置的更改；一旦为工作区选择了初始解释器，后续对设置的任何更改都将被忽略。

还可以使用语法“${env:VARIABLE}”在路径设置中使用环境变量。如果您创建了一个名为“PYTHON_INSTALL_LOC”的变量及解释器的路径，则可以使用以下设置值：

```
"python.defaultInterpreterPath": "${env:PYTHON_INSTALL_LOC}",
```

选择调试环境

除非将“python.terminal.activateEnvironment”设置设为“false”，否则当右键单击一个文件并选择“**运行Python文件**”和使用“**Python: 创建新终端**”命令时，将会自动激活项目选择的环境。

一旦从Shell中启动CodeArts IDE，并且该Shell已经激活了特定Python环境，CodeArts IDE将不会自动在默认的集成终端中激活环境。要想在CodeArts IDE中激活环境，需要在一个正在运行的CodeArts IDE实例的命令面板中使用“**Python: 创建新终端**”命令。

在终端中对已激活的环境所做的任何更改都是持久的。如在激活了虚拟环境的终端中使用“pip install”命令，将会永久地将该包添加到该虚拟环境中。

使用“**Python: 选择解释器**”命令更改解释器不会影响已经打开的终端面板。因此，可以在分割的终端中激活不同的环境：选择第一个解释器，为它创建一个终端，选择另一个解释器，然后在终端标题栏中使用“**拆分终端**”按钮（“Ctrl+Shift+5”）（  ）。

默认情况下，调试器将使用CodeArts IDE用户界面选择的Python解释器。

如果您在 “launch.json” 启动配置中定义了“python”属性，则会使用该解释器。

如果未定义相关属性，CodeArts IDE将使用项目设置的Python解释器。

说明

有关启动配置的更多详细信息，请参阅**运行Python工程启动配置**。

10.3 使用 Python 编辑代码

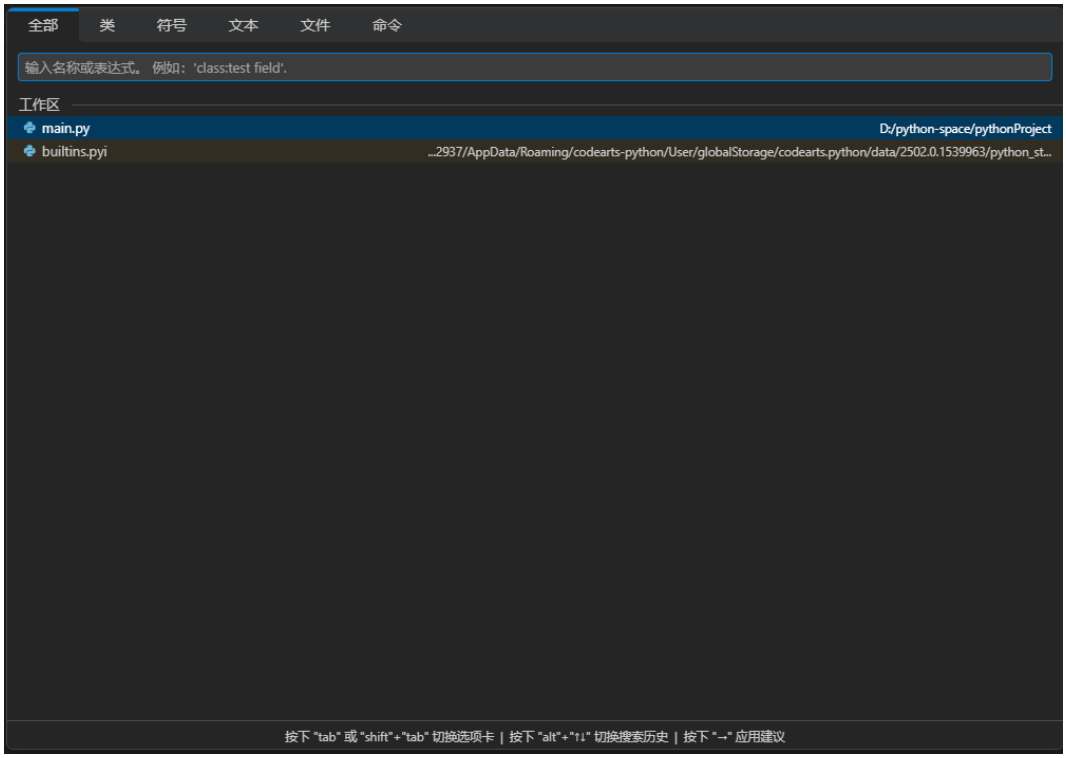
10.3.1 搜索与定位代码

CodeArts IDE的智能搜索功能让您能够即时搜索并导航到项目中的任意位置，同时轻松查找和执行任何CodeArts IDE命令。

搜索代码

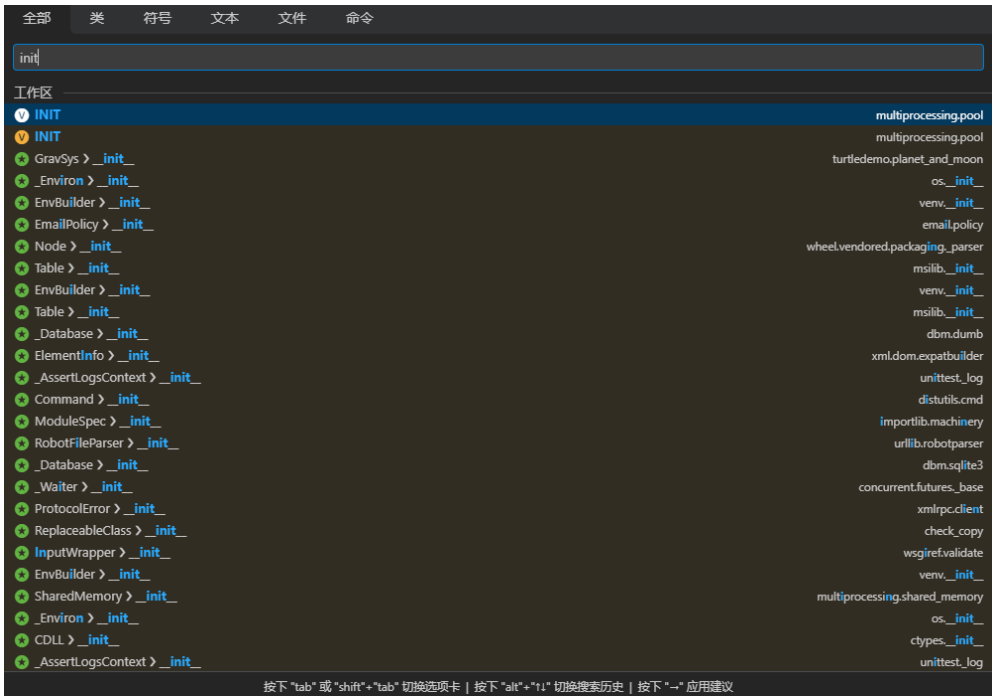
步骤1 通过按下 “Shift+Shift” / “Ctrl+Shift+A” 来启动**智能搜索**窗口。

图 10-6 启动智能搜索窗口



步骤2 输入搜索请求。要缩小搜索范围，例如仅搜索类成员或CodeArts IDE命令，可以在**智能搜索**窗口中通过按下“Tab”/“Shift+Tab”来切换标签页，或者使用**搜索查询语法和运算符**。

图 10-7 输入搜索请求



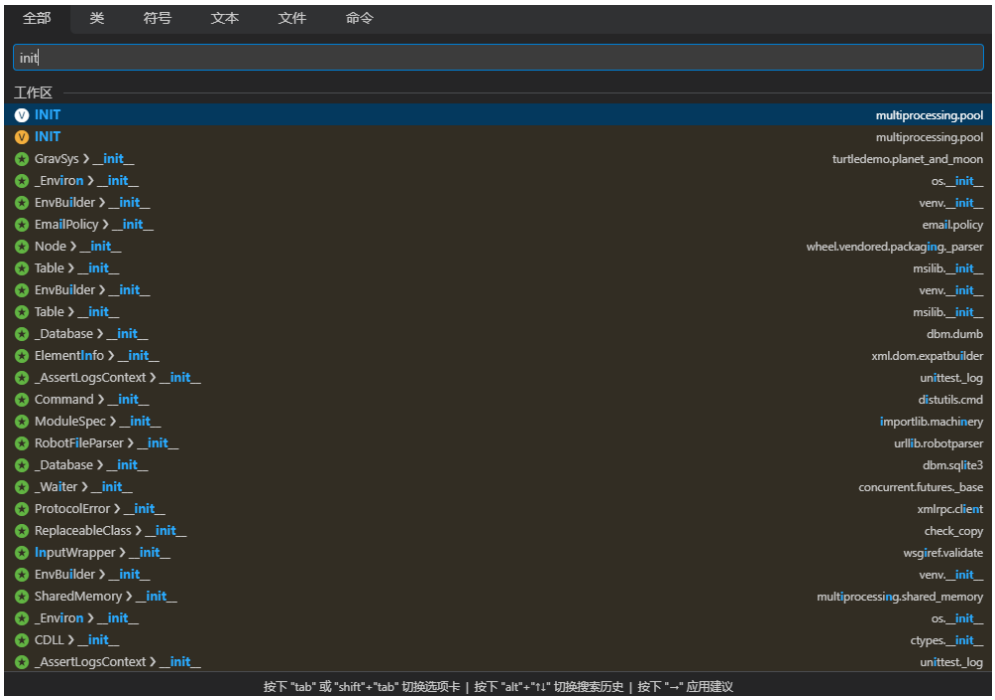
步骤3 使用光标在条目之间导航，并按下“Enter”键跳转到相应的位置或执行命令。另外，也可以双击所需的条目。要关闭**智能搜索**窗口，请按下“Escape”键。

----结束

定位任意实体

搜索查询“init”将匹配名称中包含“init”的所有实体。

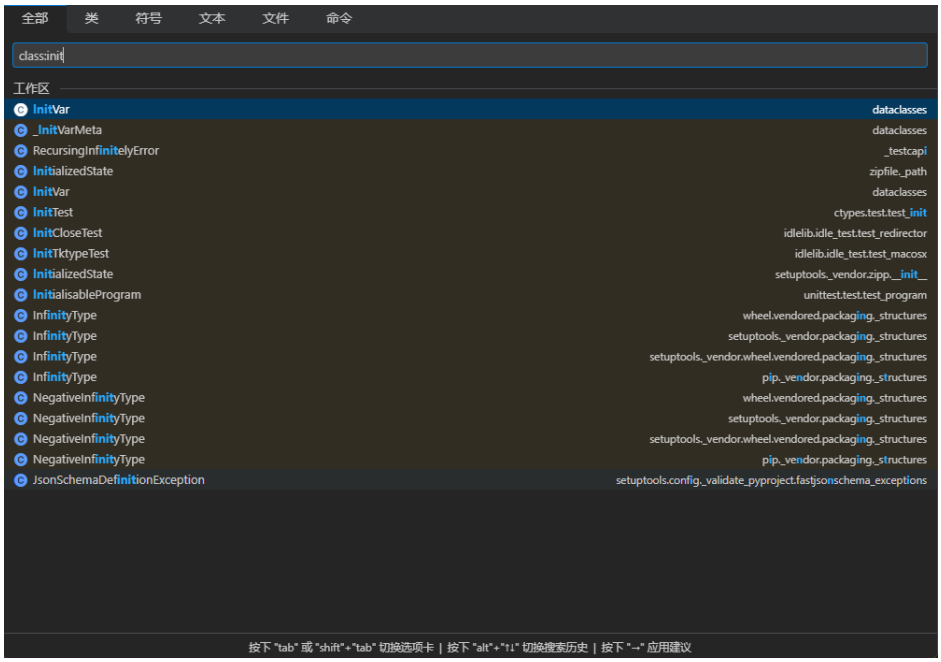
图 10-8 搜索包含“init”的所有实体



定位类

搜索查询“class:init”将匹配名称中包含“init”的所有类。使用替代语法，此查询也可以写为“type:init”、“init:class” 或“init:c”。

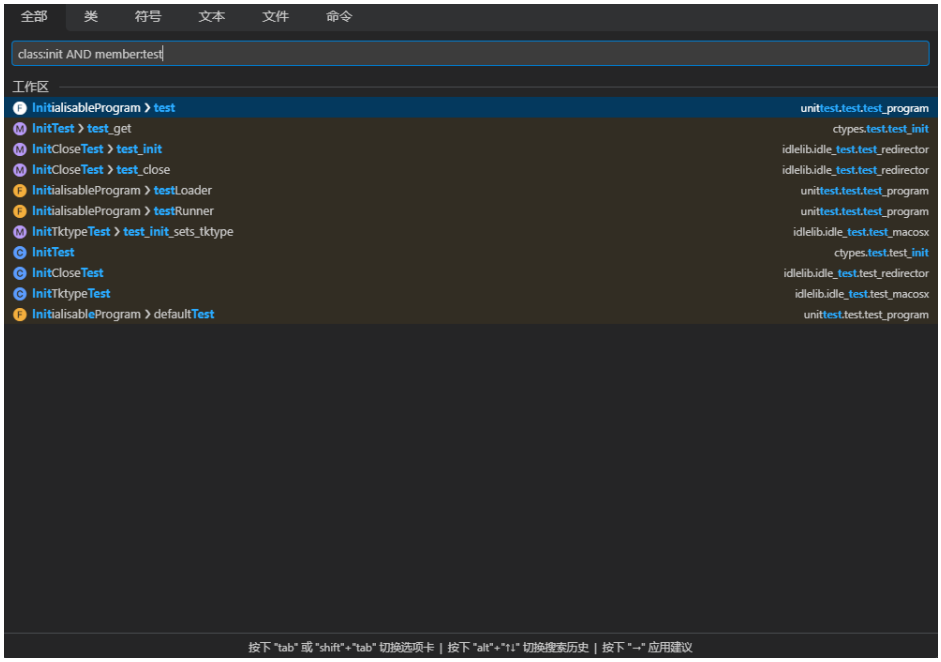
图 10-9 搜索包含“init”的所有类



查询某个类的成员

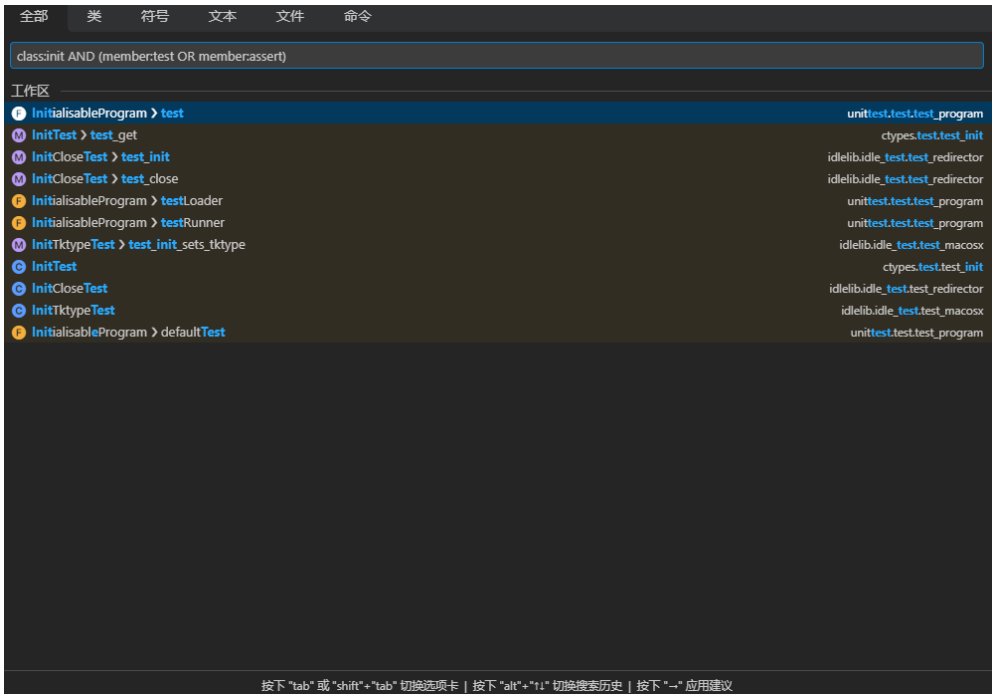
搜索查询“class:init AND member:test”将匹配类名称包含“init”的类中，类成员名称中包含“test”的所有类成员。

图 10-10 搜索包含“test”的所有类成员



搜索查询“class:init AND (member:test OR member:assert)”将匹配名称包含“init”的类中，类成员名称中包含“test”或“assert”的所有类成员。

图 10-11 搜索类成员名称中包含“test”或“assert”的所有类成员



10.3.2 搜索查询语法和运算符

搜索查询语法

搜索查询是一个字符串，用于在**智能搜索**窗口（“Shift+Shift”/“Ctrl+Shift+A”）中查询对应条目，由“dataSource:stringToMatch”对组成，这些对可以通过空格或运算符连接。如果查询中省略了“dataSource”，则搜索将在所有可用的数据源中进行。使用“stringToMatch:dataSource”，即反向模式，也是可能的。

以下是可用的数据源列表：

表 10-1 数据源列表

数据源名称	数据源缩写	描述
“class”/“type”	“c”/“t”	类实体
“member”	-	成员实体，即类方法或类字段实体
“text”	-	文本实体
“file”	“fn”	文件和文件夹实体
“command”	-	命令实体

搜索运算符

您可以使用AND和OR运算符或其组合来组成复杂的搜索查询，例如class:foo AND (method:bar OR method:baz)。

表 10-2 搜索运算符

运算符	语法	描述
“AND”	“AND”，“&”，“&&”，(“空格”字符)	**SmartSearch**将定位与每个查询匹配的条目，并仅返回所有条件的交集条目。
“OR”	“OR”，“|”，“||”	**SmartSearch**将返回与提供的任何查询匹配的所有条目。

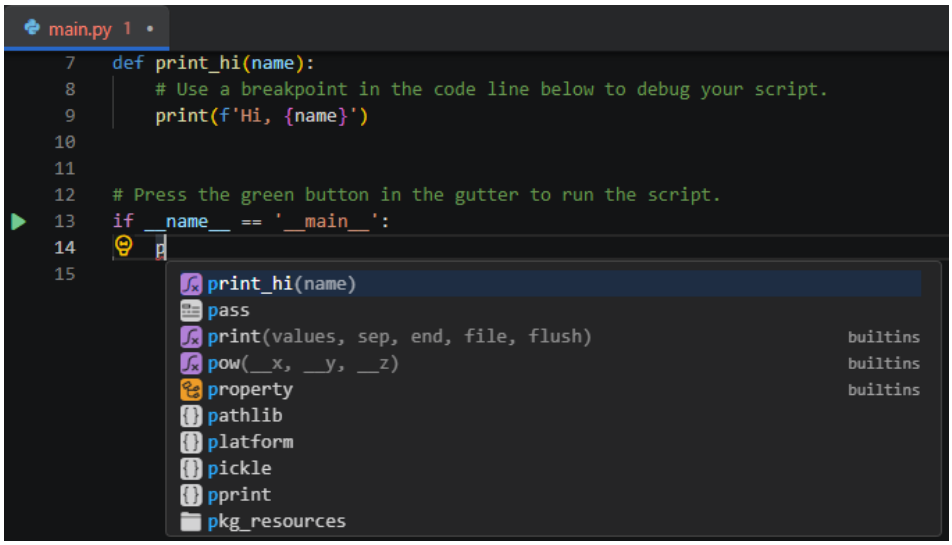
10.4 使用 Python 补全代码

CodeArts IDE为当前项目和已安装包中的文件中的Python关键字和所有符号提供代码补全。

触发代码补全

- 要手动触发代码补全，请按“Ctrl+I” / “Ctrl+Space” / “Ctrl+Shift+Space”或输入触发字符（如点字符 “.”）。

  图 10-12 手动触发代码补全

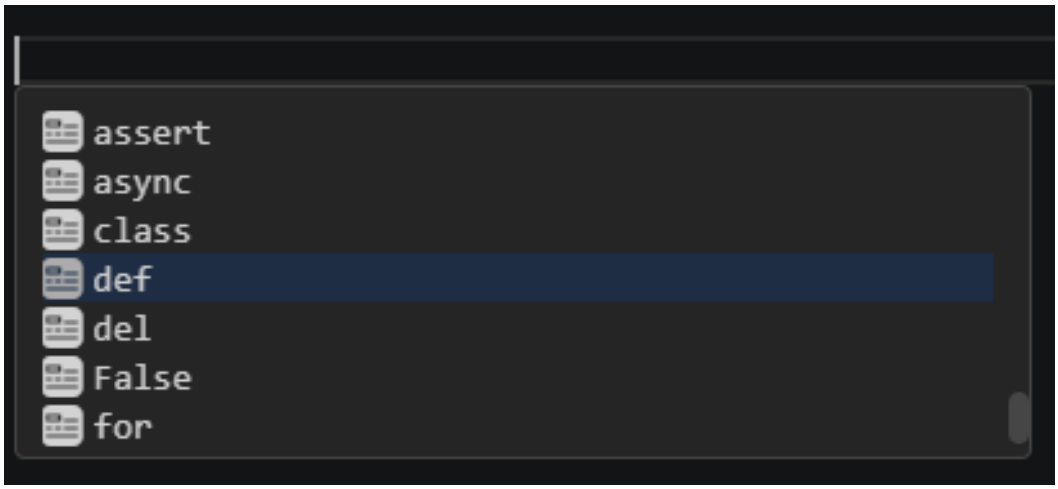
  

  - 要插入选择的符号，请按“Enter” 。
  - 要插入选定的符号并替换当前光标位置处的符号，请按“Tab”键。
  - 要关闭建议列表而不插入建议，请按“Esc”键。

关键词补全

CodeArts IDE为Python保留关键字（如“assert”、“class”、“if”、“def”等）提供代码补全。

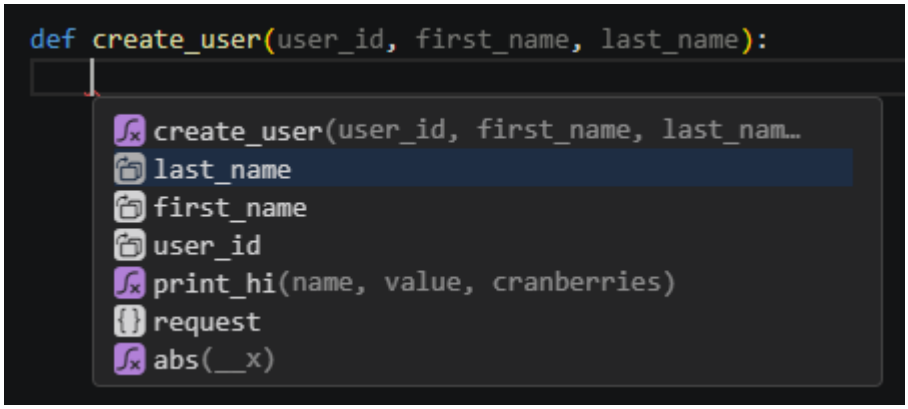
图 10-13 关键词补全



参数补全

代码补全建议列表中会按优先级排列项目中定义好的方法参数。

图 10-14 参数补全



折叠区域

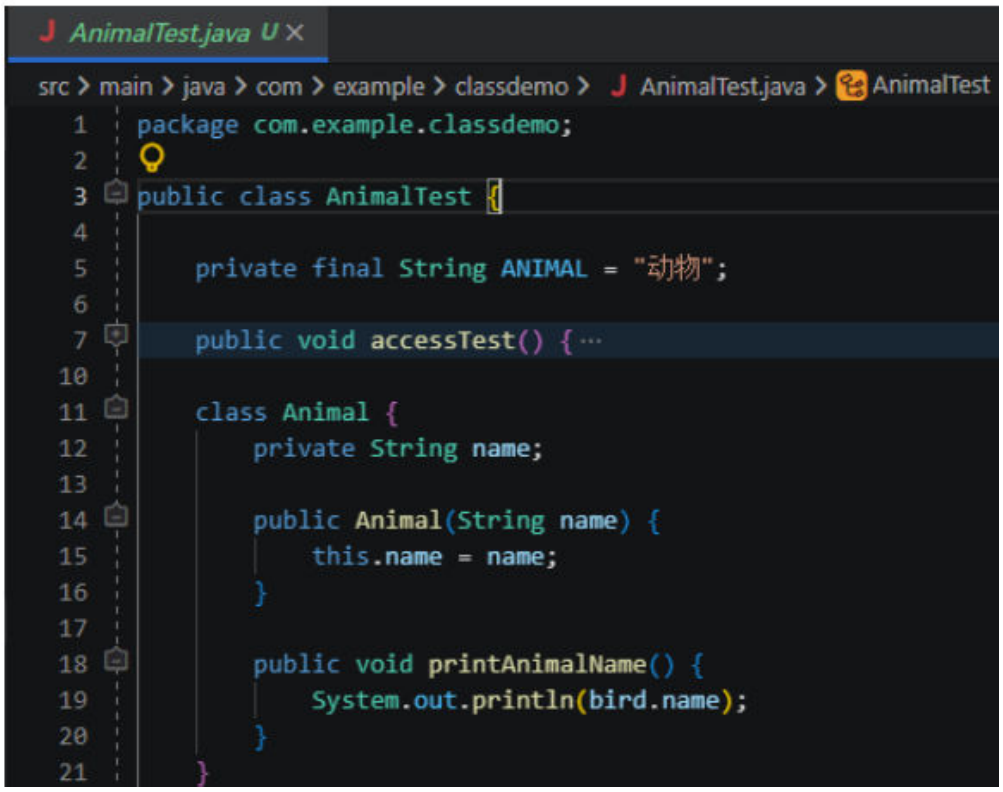
折叠区域允许您折叠或展开代码片段，以便更好地查看源代码。在Python上下文中，使用以下字符来标记折叠区域：

- 开始区域：“#region” 或 “# region”
- 结束区域：“#endregion”或 # endregion

然后就可以使用 “Ctrl+Shift+[” / “Ctrl+-”（IDEA快捷键）/ “Ctrl+Numpad-”（IDEA快捷键）来折叠光标处最内部的未折叠区域，以及 “Ctrl+Shift+]” / “Ctrl+=”（IDEA快捷键）/ “Ctrl+Numpad+”（IDEA快捷键）来展开光标处的折叠区域。

使用行号和行开始之间的折叠图标来折叠代码区域。将鼠标移动到折叠符号上，然后单击图标实现折叠和展开区域。使用“**Shift + 单击**”折叠图标实现折叠或展开区域和内部的所有区域。

图 10-15 折叠区域



您还可以使用以下操作：

- 折叠（**Ctrl+Shift+[/ Ctrl+-(IDEA键盘映射)**）：折叠光标处最里面的未折叠区域。
- 展开（**Ctrl+Shift+] / Ctrl+=(IDEA键盘映射)**）：在光标处展开折叠区域。
- 切换折叠（**Ctrl+K Ctrl+L**）：折叠或展开光标处的区域。
- 递归折叠（**Ctrl+K Ctrl+[/ Ctrl+Alt+-(IDEA键盘映射)**）：折叠光标处最里面的未折叠区域以及该区域内的所有区域。
- 递归展开（**Ctrl+K Ctrl+] / Ctrl+Alt+=(IDEA键盘映射)** ）：展开光标处的区域以及该区域内的所有区域。
- 全部折叠（**Ctrl+K Ctrl+0 / Ctrl+Shift+-(IDEA键盘映射)**）：折叠编辑器中的所有区域。
- 全部展开（**Ctrl+K Ctrl+J / Ctrl+Shift+=(IDEA键盘映射)**）：展开编辑器中的所有区域。
- 折叠级别X（**对于级别2，Ctrl+K Ctrl+2**）：折叠级别X的所有区域，但当前光标位置的区域除外。
- 折叠所有块注释（**Ctrl+K Ctrl+/**）：折叠以块注释标记开头的所有区域。

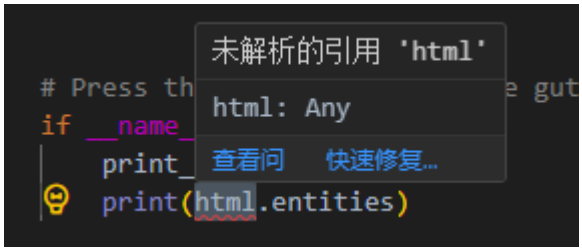
默认情况下，使用基于缩进的折叠策略。

添加导入

当您使用代码补全（ Ctrl+I / Ctrl+Space / Ctrl+Shift+Space ）插入引用尚未导入的元素时，CodeArts IDE会自动插入缺少的导入语句。CodeArts IDE还会突出显示当前缺少导入语句的符号，并提供源操作来自动插入导入。

步骤1 在代码编辑器中，将光标置于强调显示的未解析符号处。

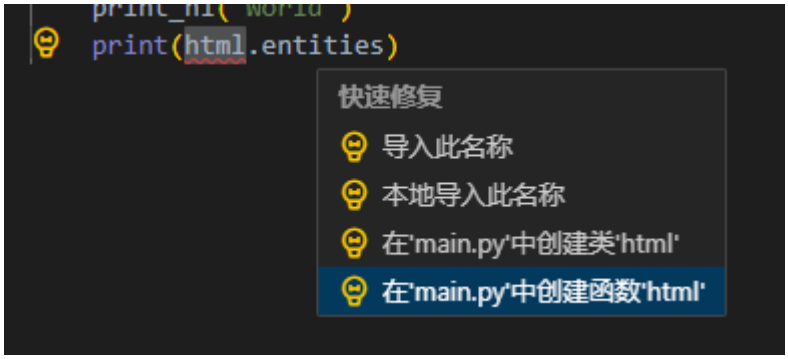
图 10-16 光标置于未解析符号处



步骤2 按“Ctrl+.” / “Ctrl+1” / “Alt+Enter”（IDEA快捷键），然后在弹出菜单中选择“**快速修复...**”。

如果有多个可能的导入声明，请选择“**导入此名称**”，然后在弹出菜单中选择所需的声明。

图 10-17 导入此名称



----结束

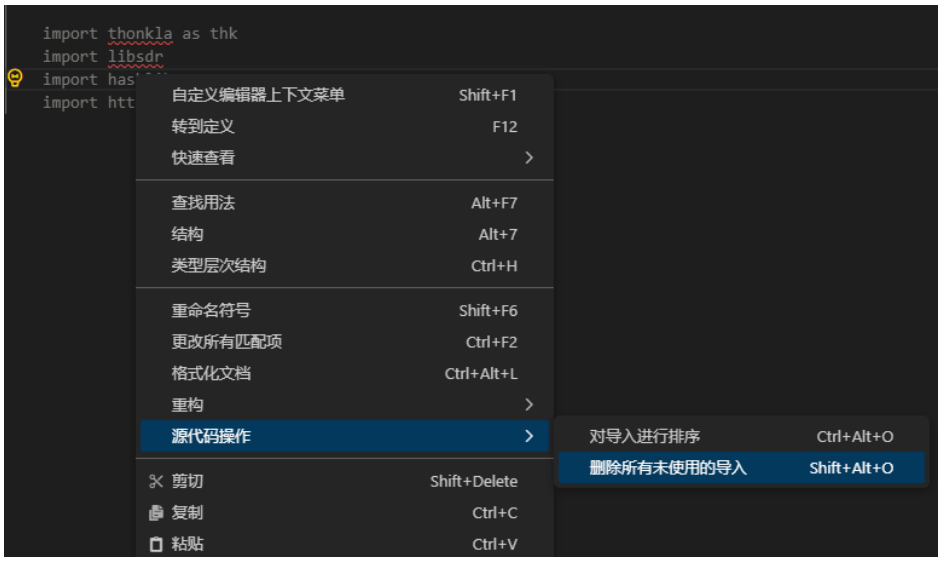
导入排序

CodeArts IDE提供了自动按字母顺序排序导入语句并移除不明确导入的“源代码操作”。

步骤1 在代码编辑器中，右键单击并选择上下文菜单中的“**源代码操作**”。或者，按“Shift+Alt+S” /“Alt+Insert”（IDEA快捷键）。

步骤2 在弹出菜单中，选择“对导入进行排序”。CodeArts IDE会删除不明确的导入，并按字母顺序对导入语句进行排序。

图 10-18 对导入进行排序



----结束

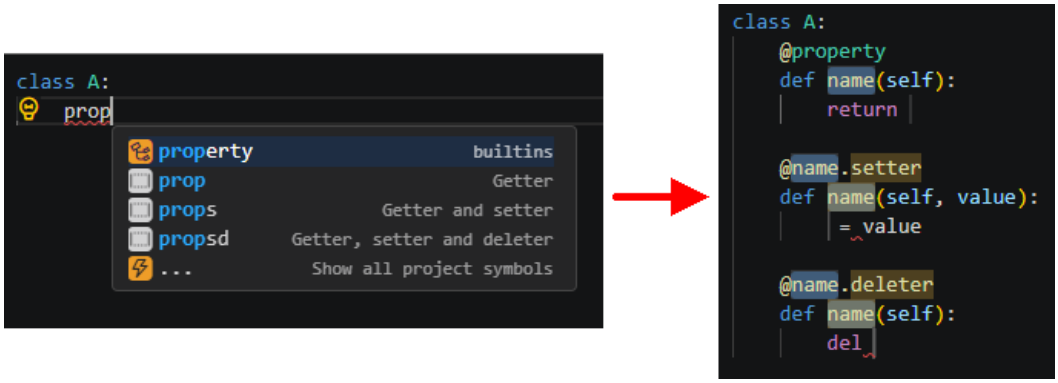
10.5 使用 Python 代码片段

代码片段是模板，可以简化如循环或条件语句等重复代码模式的输入。CodeArts IDE为Python语言提供了多个内置的代码片段，这些片段和其他建议一起出现在代码补全（“Ctrl+I” / “Ctrl+Space” / “Ctrl+Shift+Space”）中。代码片段通常放置在代码补全建议列表底部。要快速访问它们，请触发代码补全，然后按“Ctrl+Up” / “Up”。

常规片段

常规代码片段用于快速输入常见的代码结构。例如，使用“propsd”代码片段，您可以快速为类属性创建getter、setter和deleter方法。

图 10-19 使用“propsd”代码片段



有些代码片段初始化时是包含占位符的不完整片段，需要填充对应占位符来使代码片段成为完整的可执行代码。您可以通过按“Tab”键在这些占位符之间跳转。

表 10-3 常规片段语句

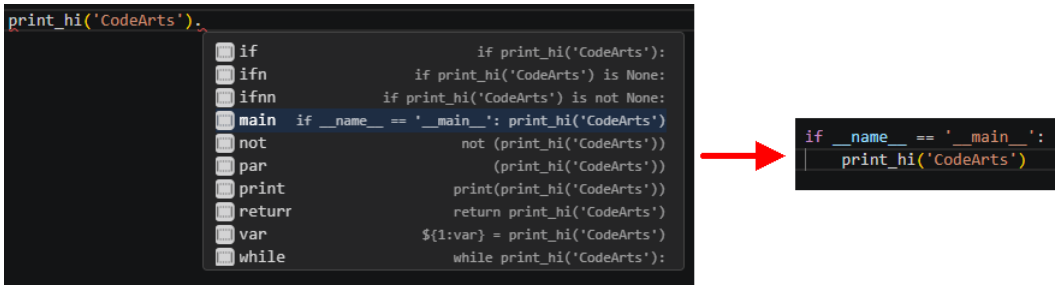
类型	代码片段描述	缩写	扩展内容
条件语句	创建一个“__name__”变量的主函数守卫 或者主程序入口。	“main”	if __name__ == '__main__': pass
循环语句	用“for”循环迭代一个可迭代对象。	“iter”	for i in <iterable>: pass
	用“for”循环迭代一个可迭代对象的索引和键值。	“itere”	for index, value in enumerate(iterable): pass
	迭代生成的数字范围。	“iterr”	for i in range(0): pass
列表解析	字典解析。	compd	{item: item for item in <iterable>}
	带过滤的字典解析。	compdi	{item: item for item in <iterable> if <condition>}
	生成器解析。	compg	(item for item in <iterable>)
	带过滤的生成器解析。	compgi	(item for item in <iterable> if <condition>)
	列表解析。	compl	[item for item in <iterable>]
	带过滤的列表解析。	compli	[item for item in <iterable> if <condition>]
	集合解析。	comps	{item for item in <iterable>}
	带过滤的集合解析。	compsi	{item for item in <iterable> if <condition>}
类成员	从父类初始化调用方法的实现。	“super”	super().<super_method_name>()
	为类属性创建getter方法。	“prop”	@property def name(self): return

	为类属性创建setter和getter方法。	“props”	@property
			def name(self):
			  return
			
			@name.setter
			def name(self, value):
			  = value

	为类属性创建setter、getter和deleter方法。	“propsd”	@property
			def name(self):
			  return
			
			@name.setter
			def name(self, value):
			  = value
			
			@name.deleter
			def name(self):
			  del

后缀片段

后缀片段（Postfix snippets）是用于将一个现有的表达式转换为另一个表达式的工具。要使用后缀片段，只需在表达式后面添加一个点（“.”），然后从代码补全建议列表中选择所需的片段。例如，通过使用“main”后缀片段，你可以将一个表达式包装成一个条件性的名为main的表达式。



有些代码片段初始化时是包含占位符的不完整片段，需要填充对应占位符来使代码片段成为完整的可执行代码。您可以通过按Tab键在这些占位符之间跳转。

表 10-4 后缀片段语句

类型	代码片段描述	缩写	扩展内容
一般语句	为表达式引入变量。	“var”	var my_expression
	从封闭方法返回表达式的值。	“return”	return my_expression
	将表达式用括号包围。	“par”	(my_expression)
	给表达式取反。	“not”	not (my_expression)
	返回表达式的长度（项目数）。	“len”	len(my_expression)
条件语句	创建“if”语句。	“if”	if my_expression: <cursor>
	创建if语句判断表达式的值是否为“None”。	“ifn”	if my_expression is None: <cursor>
	创建if语句判断表达式的值是否不为“None”。	“null”	if my_expression is not None: <cursor>
	为表达式创建一个“__name__”主函数守卫。	“main”	if __name__ == '__main__': my_expression
循环语句	为表达式创建“while”循环。	“while”	while my_expression: <cursor>
程序输出	将表达式发送到标准输出。	“print”	print(my_expression)

10.6 使用 Python 校验代码

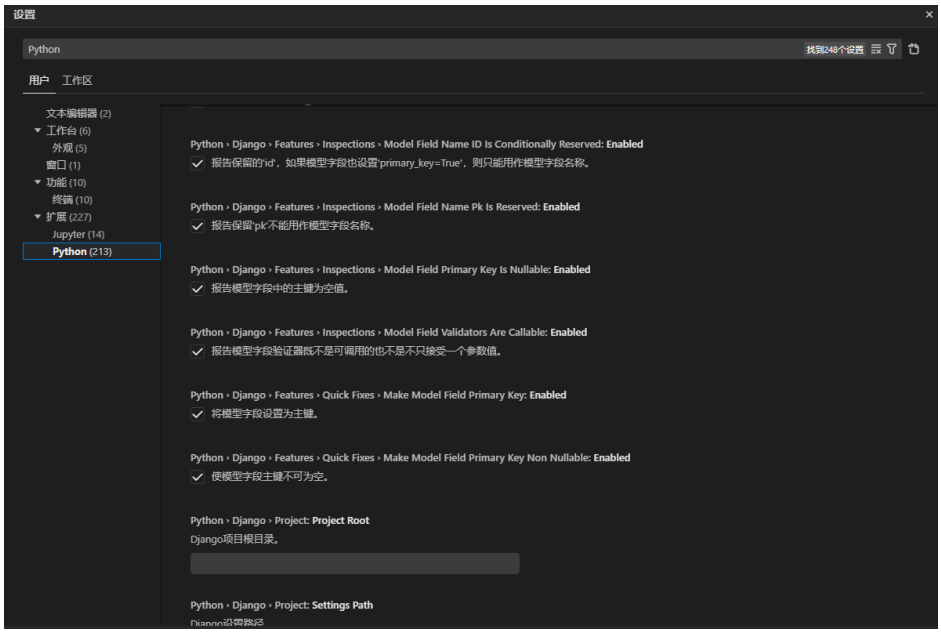
编写代码时，CodeArts IDE会在后台自动根据一组预定义的验证规则对其进行分析，使其能够发现各种问题，如拼写错误和其他潜在的错误。这有助于您在运行代码之前检测并修正问题。CodeArts IDE对很多问题提供了快速修复功能，方便您迅速解决问题。

您可以自定义应用于代码的验证规则集。

步骤1 在CodeArts IDE设置中（“Ctrl+,”），输入python关键字，转到“**扩展**”>“**Python**”。

步骤2 在“**Features: Inspections**”或“**Features: Quick Fixes**”设置组下找到所需的验证规则或快速修复，或者使用搜索框快速定位。

图 10-20 快速定位



要启用或禁用某个规则或快速修复，请在其名称旁边的复选框中进行选择。

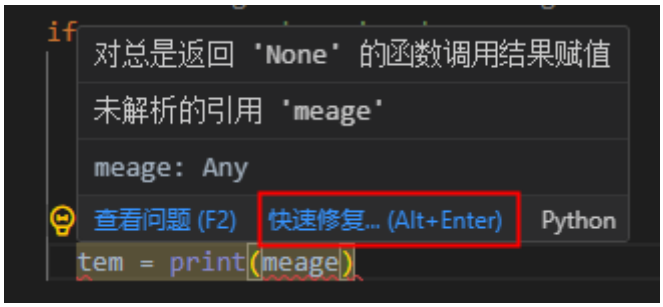
----结束

应用快速修复

如果检测到的问题有快速修复可用，您可以即时修复它。

1. 单击问题描述中的“快速修复”链接。或将光标定位在高亮显示的位置，然后按Alt+Enter键。

图 10-21 快速修复



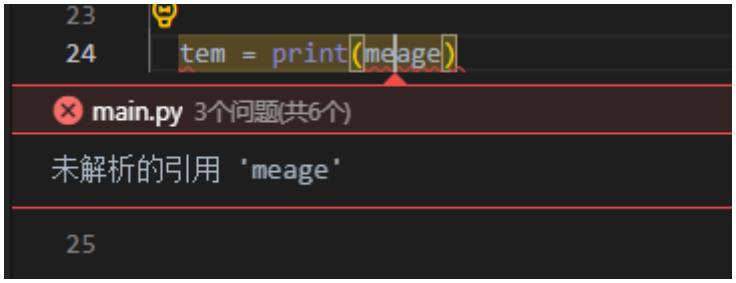
2. 在弹出菜单中，选择所需的快速修复类型。

图 10-22 选择快速修复的类型

查看问题详情

- 要在快速查看器中打开问题说明，请单击“查看问题”链接。或将光标定位在高亮显示的位置，然后按F2。

图 10-23 查看问题详情



- 在快速查看器中，单击“转到下一个问题”（↓）和“转到上一个问题”（↑），或按**“Alt+F8”** / **“F2”(IDEA键盘映射)** / **“Shift+Alt+F8”** / **“Shift+F2”(IDEA键盘映射)**在当前文件中的错误之间跳转。

10.7 使用 Python 重构代码

10.7.1 Python 代码重构简介

Python程序重构的目标是进行系统级的代码更改，同时不影响程序的行为。CodeArts IDE提供了许多易于访问的重构选项。

重构命令可以从编辑器的上下文菜单中获取。选择您想要重构的元素，右键单击它，并从上下文菜单中选择“重构”。

以下是一些可用的重构选项：

- **内联变量重构**
  这种重构允许您用变量的初值替换变量本身。这是引入变量重构的相反操作。
- **引入变量重构**
  这种重构允许您创建一个新变量，用选定的表达式初始化它，并将原始表达式替换为对新创建变量的引用。
- **变量重命名重构**
  这种重构允许您在整个项目文件中重命名一个符号及其所有使用的地方。

10.7.2 内联变量重构

通过此重构，您可以用变量的初始值设定项替换变量。这与引入变量重构相反。

执行重构

步骤1 在代码编辑器中，将光标放在要内联变量使用的地方上。

步骤2 在编辑器上下文菜单中，选择“**重构**”>“**内联变量**”或按 “Ctrl+Alt+N” 。

步骤3 在打开的对话框中，提供内联变量的名称。

----结束

执行重构案例

作为示例，将内联变量“message”替换为其初始值设定项“Hello!”。

重构前：

```
message = "Hello!"

print(message)
```

重构后：

```
print("Hello!")
```

10.7.3 引入变量重构

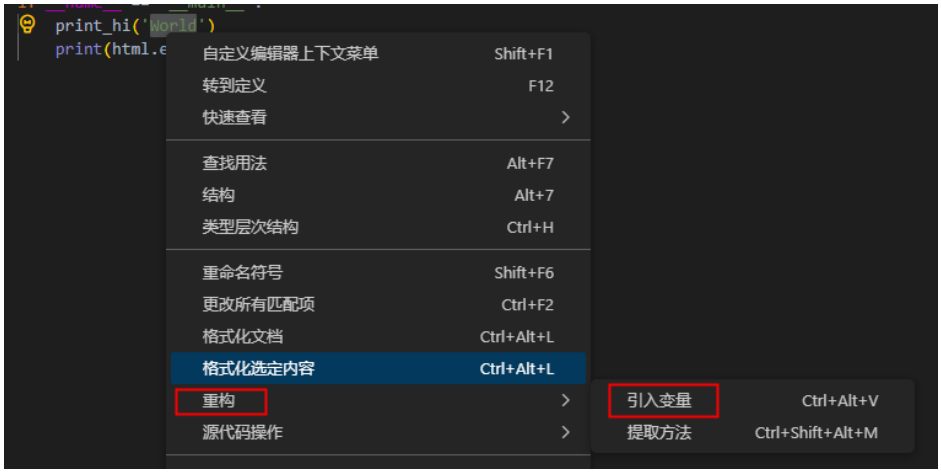
通过此重构，您可以创建一个新变量，使用所选表达式对其进行初始化，然后使用对所创建变量的引用替换原始表达式。这与内联变量重构相反。

执行重构

步骤1 在代码编辑器上，将光标放在要提取的表达式上。

步骤2 在编辑器上下文菜单中，选择“**重构**”>“**引入变量**”或按 “Ctrl+Alt+V” / “Shift+Alt+L” 。

图 10-24 引入变量重构



步骤3 如果多个表达式属于重构范围，请在出现的弹出窗口中选择所需的表达式。

步骤4 在打开的对话框中，提供引入变量的名称。

----结束

执行重构案例

作为示例，提取字符串“Hello!”到一个新的消息变量中。

重构前：

```
print("Hello!")
```

重构后：

```
message = "Hello!"

print(message)
```

10.7.4 变量重命名重构

通过此重构，您可以在项目文件中重命名符号及用法。

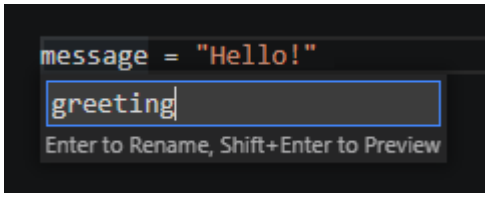
执行重构

步骤1 在代码编辑器中，将光标放在要重命名的符号的用法或声明上。

步骤2 在编辑器上下文菜单中，选择“**重命名符号**”，或按 “F2”/ “Shift+Alt+R”/ “Shift+F6”（IDEA快捷键）。

步骤3 在出现的弹出窗口中，为符号提供所需的新名称。

图 10-25 变量重命名重构



----结束

执行重构案例

作为示例，将变量 “message” 重命名为“greeting”。

重构前：

```
message = "Hello!"

print(message)
```

重构后：

```
greeting = "Hello!"

print(greeting)
```

10.8 配置 Python 工程测试框架

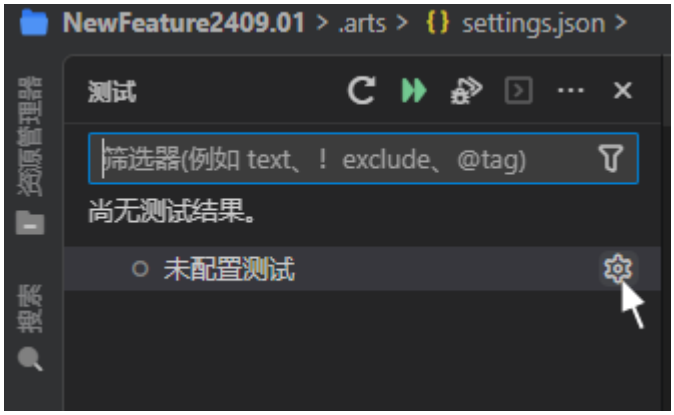
10.8.1 配置测试框架

CodeArts IDE集成了pytest和unittest测试框架，让您可以轻松运行和调试Python测试用例。

步骤1 单击CodeArts IDE左边活动栏的**“测试”**（ ）按钮来打开**测试**视图。

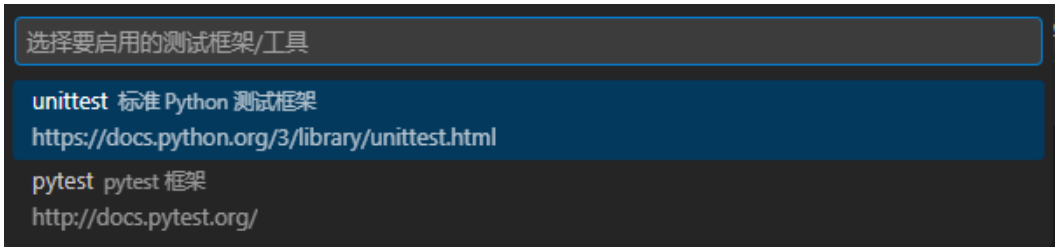
步骤2 在测试视图中，将鼠标悬停在未配置的测试记录上，然后单击配置测试按钮（ ）。

图 10-26 配置测试



步骤3 在弹出的窗口中选择测试框架来启动对应集成。

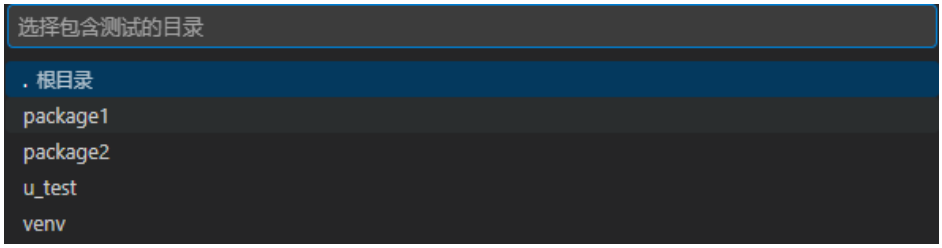
图 10-27 选择测试框架



- 如果您选择“**pytest**”，CodeArts会根据**pytest的测试识别规范**自动发现测试用例。

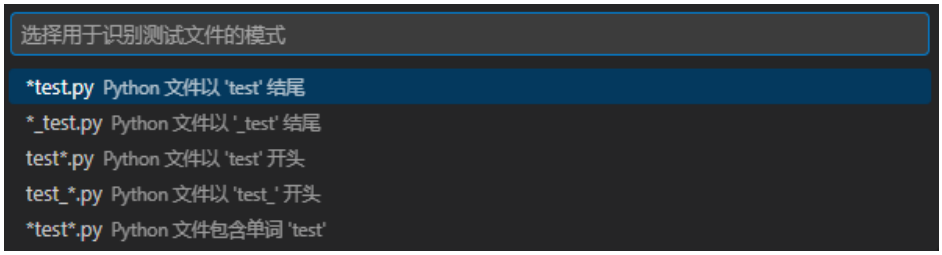
- 如果您选择“**unittest**”，您需要执行以下步骤来识别测试用例。
  - 在打开的对话框中，选择包含测试源文件的项目文件夹。

    图 10-28 选择项目文件夹

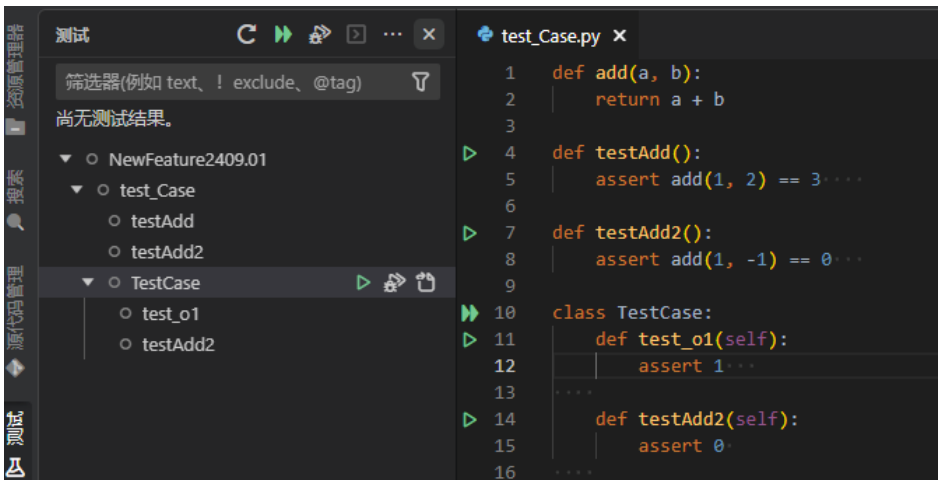
  - 在接着打开的对话框中，选择用于识别您的测试文件的文件通配符模式。

    图 10-29 选择通配符模式

步骤4 测试框架集成配置完成后，CodeArts IDE会在测试视图中展示测试用例。

图 10-30 在测试视图中展示测试用例



----结束

10.8.2 运行和调试测试用例

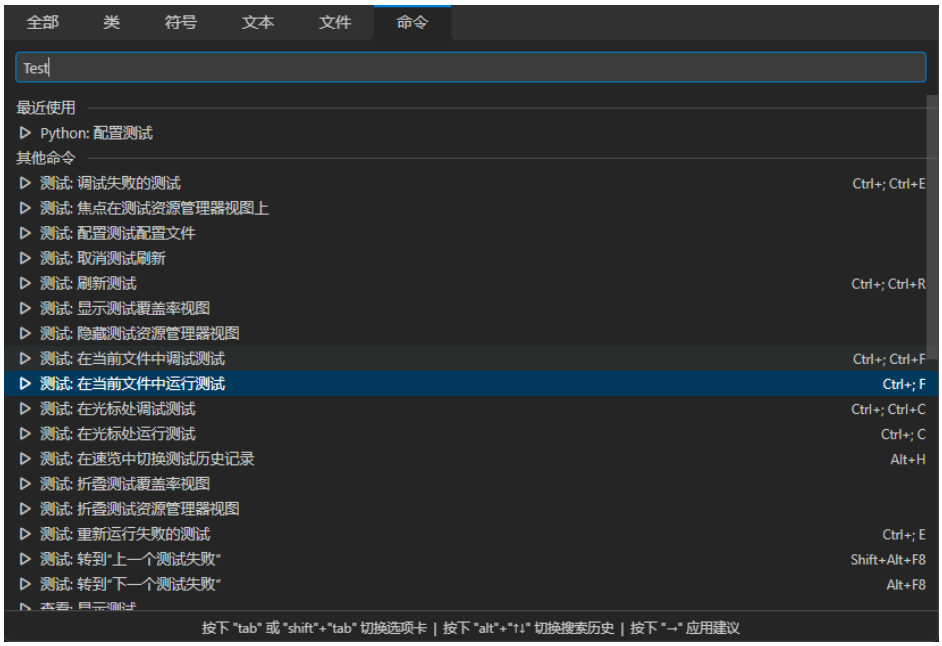
CodeArts IDE为运行和调试您的测试用例提供了多个选项：

- 在测试类的代码编辑器中，单击测试类声明旁的**运行**按钮（▶），运行该类中的所有测试。或者单击某个测试方法旁边的运行按钮来仅运行单个测试。要调试测试，请右键单击**运行**按钮（▶），并从上下文菜单中选择“**调试测试**”。

图 10-31 调试测试

- 配置**pytest**和**unittest**启动配置。
  在命令面板（“Ctrl+Ctrl”/“Ctrl+Shift+P”）中，搜索“Test”并使用与测试相关的命令，例如“**在当前文件中运行测试**”或“**在光标处运行测试**”。

图 10-32 运行测试



10.9 设置断点和调试 Python 工程

10.9.1 调试 Python 工程

CodeArts IDE内置调试器有助于加快编辑、运行和调试循环。调试器提供了所有基本功能，例如通过**启动配置**自定义应用程序启动、在代码中**设置断点**、**检查程序的挂起状态**并**单步执行**、动态评估表达式等等。以下为调试的具体步骤：

步骤1 在代码中**设置断点**，以定义程序应停止的位置。

步骤2 在调试模式下**运行程序**。

步骤3 当程序暂停时，在“运行和调试”视图中**检查其状态**。

步骤4 定位错误，进行修复，并重新运行程序。

----结束

10.9.2 设置不同的断点

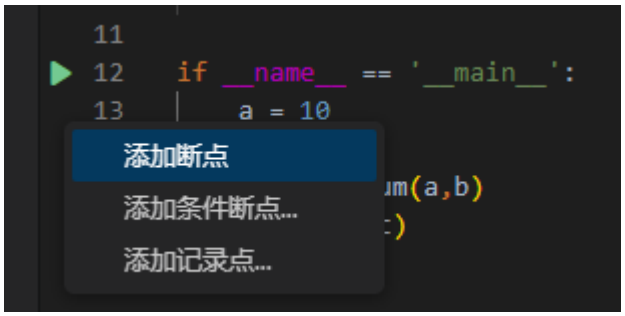
断点定义了源代码中程序执行应停止的位置。CodeArts IDE支持多种类型的断点，可以通过单击编辑器行号边缘、使用边缘的上下文菜单或在“**运行和调试**”视图的“**断点**”部分中进行切换。

行断点

行断点是常规的断点类型，当程序执行到设置有断点的行时，程序会暂停执行。

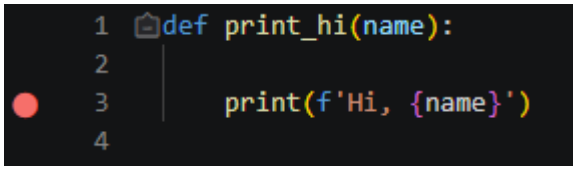
步骤1 在编辑器行号边缘单击需要设置断点的行。或右键单击，从上下文菜单中选择“**添加断点**”。

图 10-33 添加断点



步骤2 行断点添加完成后在编辑器边缘以圆形图标 (🔴) 表示：

图 10-34 行断点添加完成后



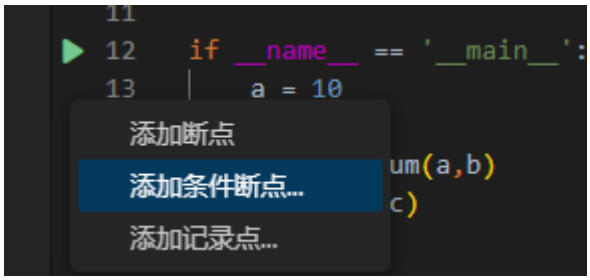
----结束

条件断点

CodeArts IDE调试器允许您根据任意表达式或命中次数设置条件断点。

步骤1 在代码编辑器中，右键单击所需行边缘，从上下文菜单里选择“**添加条件断点**”。

图 10-35 添加条件断点



步骤2 在打开的行内编辑器中，在列表里选择条件类型。

- **表达式**：当表达式计算结果为“true”时命中断点。
- **命中次数**：断点需要命中指定的次数才能暂停程序执行。

图 10-36 选择条件类型

步骤3 输入条件并按下**Enter**键。

----结束

说明

也可以向常规行断点添加条件或命中计数。右键单击编辑器边缘中的断点，然后从上下文菜单中选择所需的操作。

记录点

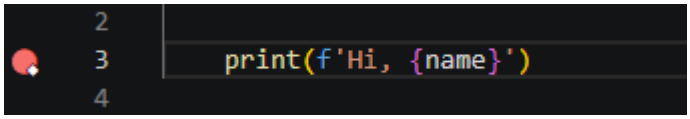
记录点也是一种断点，但被触发时不会暂停程序执行，而是将一条消息记录到控制台。

步骤1 在代码编辑器中，通过右键单击想要设置记录点的行的编辑器边缘，并从上下文菜单中选择“**添加记录点**”。

另外，也可以在主菜单中选择“**调试**”>“**新建断点**”>“**内联记录点**”。

在编辑器边缘，记录点用一个圆形图标（ ）表示。

图 10-37 添加记录点



步骤2 随后会打开一个预览编辑器，在其中输入当记录点被触发时应该记录的消息。日志消息可以是纯文本，也可以是包含在大括号（“{}”）中需要求值的表达式。

----结束

说明

与常规断点一样，记录点可以被启用或禁用，也可以由条件或触发次数来控制。如果设置了条件或触发次数，则只有当条件为真或达到触发次数时，才会记录消息。

函数断点

除了直接在源代码中放置断点外，还可以通过指定函数/方法名来创建断点，程序执行在进入指定的函数时将会暂停。

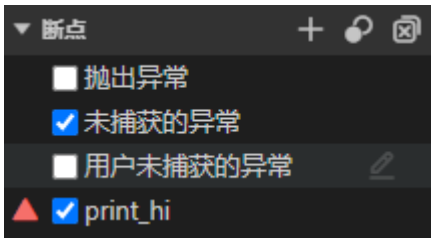
步骤1 单击CodeArts IDE底部面板中的“**运行和调试**”按钮（），或者按下 “Ctrl+Shift+D” / “Shift+Alt+F9”（IDEA快捷键）/ “Alt+5”（IDEA快捷键）/ “Ctrl+Shift+F8”（IDEA快捷键），打开“**运行和调试**”视图。

步骤2 在“**断点**”部分视图的工具栏中，单击“**添加函数断点**”按钮（），或在主菜单中选择“**调试**”>“**新建断点**”>“**函数断点**”。

步骤3 输入所限定函数的完整名称，按“Enter” 键。

函数断点将在“**运行和调试**”视图的“**断点**”部分视图里以三角形图标（）表示。

图 10-38 添加函数断点



----结束

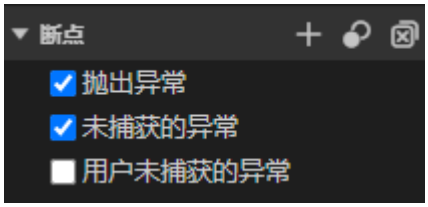
异常断点

CodeArts IDE调试器支持异常断点，每当抛出异常时，都会暂停程序执行。异常断点是应用于全局的，不需要特定的源代码引用。

步骤1 单击CodeArts IDE底部面板中的“**运行和调试**”按钮（），或按下 “Ctrl+Shift+D” / “Shift+Alt+F9”（IDEA快捷键）/ “Alt+5”（IDEA快捷键）/ “Ctrl+Shift+F8”（IDEA快捷键）来打开“**运行和调试**”视图。

步骤2 展开“**断点**”部分，并勾选你想要设置的异常断点旁边的复选框。

图 10-39 添加异常断点



说明

CodeArts IDE提供了几种类型的异常断点，这些断点定义了抛出时会导致程序执行暂停的特定异常。

- **抛出异常**：任何抛出的异常，无论是否被捕获。
- **未捕获的异常**：任何被抛出且未被捕获的异常。
- **用户未捕获的异常**：源自用户代码（而非库）的任何未捕获异常。

----结束

内联断点

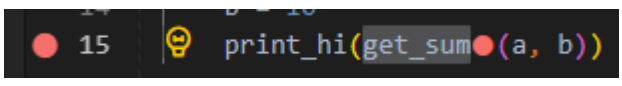
内联断点仅在执行到达与内联断点关联的列时触发。这在调试压缩代码时特别有用，因为压缩代码可能包含单行中的多个语句。

步骤1 在主菜单中选择“**调试**”>“**新建断点**”>“**内联断点**”，或者在调试会话期间使用上下文菜单。

图 10-40 添加内联断点

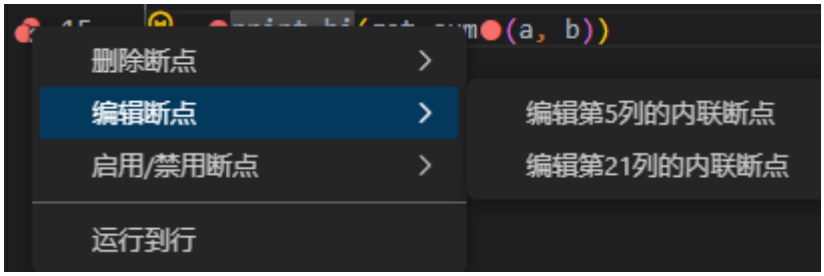
步骤2 内联断点会直接在编辑器中内联显示。

图 10-41 内联显示



步骤3 对内联断点设置条件，编辑一行上的多个断点，使用编辑器边缘的上下文菜单。

图 10-42 编辑断点



----结束

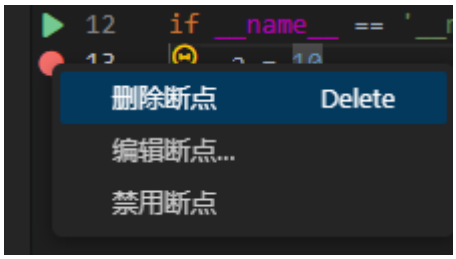
禁用断点

您可以禁用单个断点，或者一次性禁用所有断点。

要禁用单个断点，请执行以下操作之一：

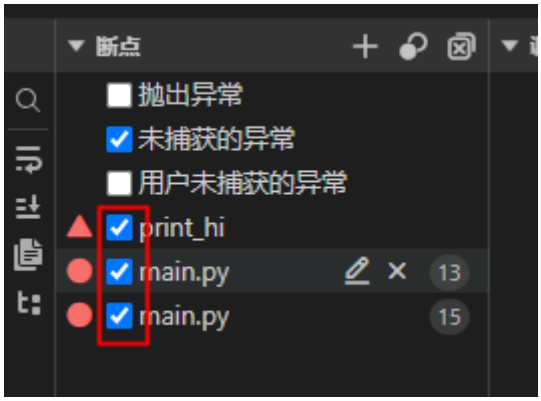
- 在编辑器边缘中，右键单击断点，并从上下文菜单中选择“**禁用断点**”。

  图 10-43 禁用断点

- 在“**运行和调试**”视图的“**断点**”部分视图，取消勾选您想禁用的断点旁边的复选框。

  图 10-44 勾选禁用的断点

要一次性禁用所有断点：

在“**运行和调试**”视图的“**断点**”部分，单击“**切换激活断点**”工具栏按钮（ ）。

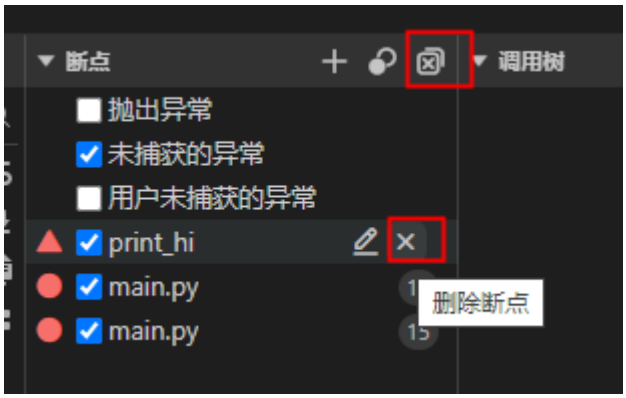
图 10-45 禁用所有断点

删除断点

您可以删除单个断点，或一次删除所有断点。

要删除单个断点，请单击编辑器边缘中的断点。要一次删除所有断点，请在“**运行和调试**”视图的“**断点**”部分视图的工具栏中，单击“**删除所有断点**”按钮 ( )。

图 10-46 删除断点



10.9.3 运行调试模式下的程序

CodeArts IDE允许直接从代码编辑器或通过启动配置启动调试会话。

一旦调试会话开始，就会显示“**调试控制台**”面板并显示调试输出，状态栏的颜色也会发生变化（默认为橙色）。

状态栏中会显示**调试状态**，**活动的调试配置**。单击调试状态可以更改活动的启动配置并开始调试，无需重新打开“**运行和调试**”视图。

图 10-47 调试控制台

从代码编辑器启动调试会话

如果不需要向程序传递任何参数，可以直接从代码编辑器开始一个调试会话。

在Python文件的代码编辑器中，单击编辑器边缘中的**运行**按钮（▶），并从弹出菜单中选择“**调试**”，或者右键单击代码编辑器，从上下文菜单中选择“**调试 Python 文件**”。**Python文件启动配置**将被创建并自动运行。

图 10-48 调试 Python 文件

说明

创建的启动配置会自动保存，之后你可以在任何时候从CodeArts IDE主工具栏上的配置列表中选择它。

通过启动配置启动调试会话

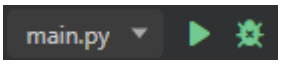
启动配置允许您配置并保存各种调试场景的调试细节设置。有关使用启动配置的更多信息，请参阅**运行Python工程启动配置**。

步骤1 从CodeArts IDE主工具栏上的配置列表中选择所需的启动配置。

步骤2 执行以下操作之一：

- 在主菜单中，选择“**调试**”>“**调试**”，或按 “F5” / “F11” / “Shift+F9”（IDEA快捷键）。

- 在调试工具栏上，确保在启动配置列表中选中了所需的启动配置，然后单击“**开始调试**”按钮（）。

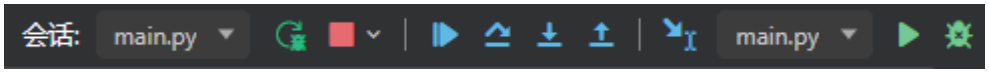


----结束

10.9.4 控制程序执行

运行调试模式下的程序后，您可以使用调试工具栏操作控制程序执行。

图 10-49 调试工具栏



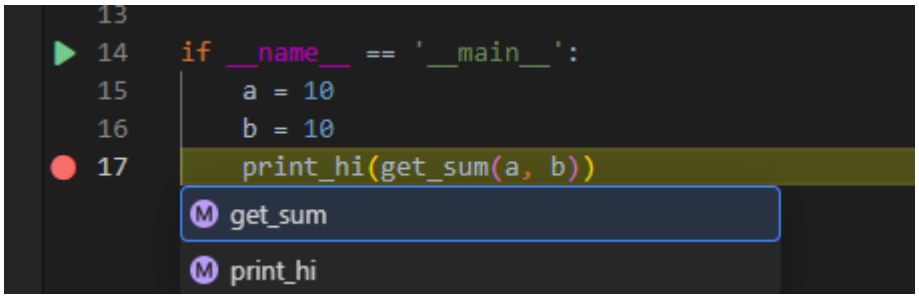
图标	对应动作	快捷键	描述
/|	暂停/继续	“F5” / “F8” / “F9” （IDEA快捷键）	暂停/恢复调试会话。
	逐过程	“F10” / “F6” / “F8” （IDEA快捷键）	跳过当前代码行到下一行。如果当前行中有方法调用，则会跳过它们的实现，以便您移至调用者方法的下一行。
	单步调试	“F11” / “F5” / “F7” （IDEA快捷键）	进入方法里展示实现代码。
	单步跳出	“Shift+F11” / “F7” / “Shift+F8” (IDEA快捷键)	跳出当前方法并跳转到调用者方法。
	重启	“Ctrl+Shift+F5” / “Shift+F9” （IDEA快捷键）	重启调试会话。
	停止	“Shift+F5” / “Ctrl+F2”	停止调试会话。
	运行到光标处	“Alt+F9” （IDEA快捷键）	恢复调试会话，在光标处暂停。

- 当程序暂停时，您可以继续执行到光标位置。在代码编辑器中，右键单击所需的行，然后从上下文菜单中选择“**运行到光标处**”或按 “Alt+F9” （IDEA快捷键）。
- 当一行中有多个方法调用时，“单步执行目标”功能可让您选择要单步执行的方法调用。

步骤1 右键单击代码编辑器边缘并从上下文菜单中选择**单步执行目标**，或按 “Ctrl+F11” 。

步骤2 在弹出菜单中，选择要单步执行的方法。

图 10-50 选择单步执行的方法



----结束

10.9.5 检查暂停的程序

当您启动调试会话时，“**运行和调试**”视图将自动打开并显示与运行和调试相关的所有信息。

要手动打开“**运行和调试**”视图，请单击CodeArts IDE底部面板中的“**运行和调试**”按钮（ ），或按 “Ctrl+Shift+D” / “Shift+Alt+F9”（IDEA快捷键）/ “Alt+5”（IDEA快捷键）/ “Ctrl +Shift+F8”（IDEA快捷键）。

图 10-51 打开“**运行和调试**”视图

要自定义“**运行和调试**”视图内容，单击右上角的“**视图和更多操作**”按钮（ ），然后在上下文菜单中勾选要显示的部分。或者可以右键单击“**运行和调试**”视图中任意部分的标题栏，然后在上下文菜单中选择。

“**运行和调试**”视图包含以下部分：

- 检查变量
- 检查调用堆栈
- 监视
- 断点

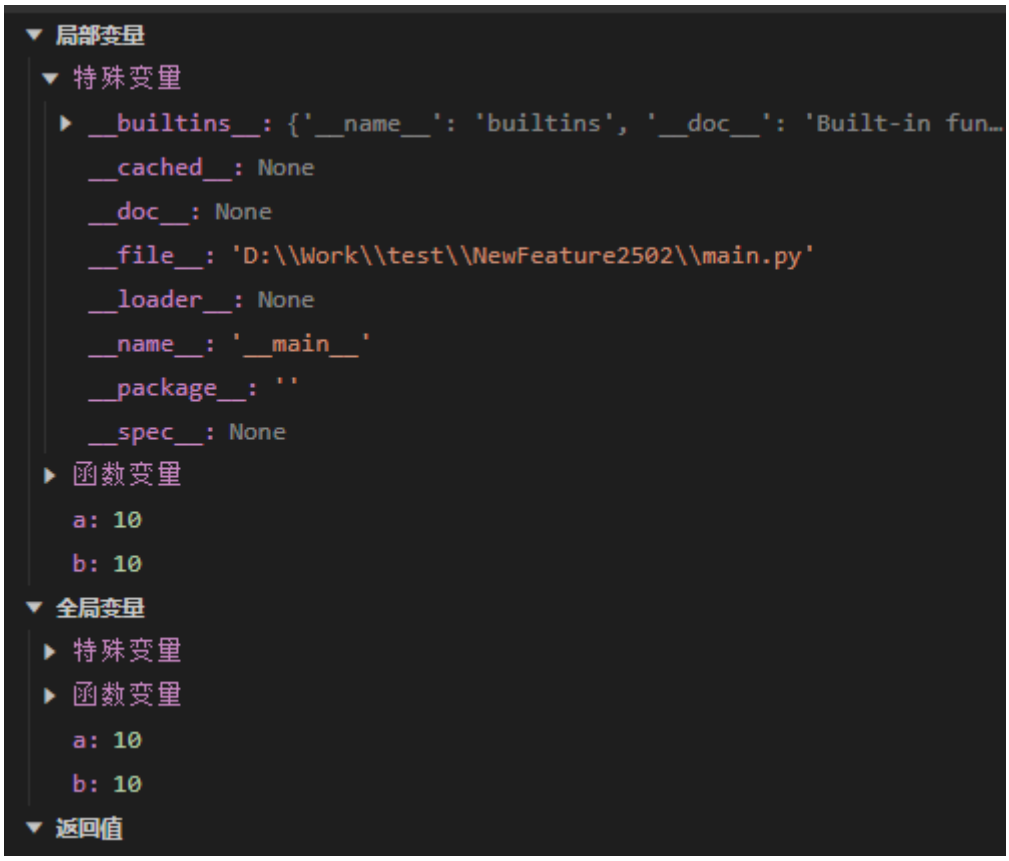
检查变量

“**变量**”部分显示在当前堆栈帧（即在“**调用堆栈**”部分中选定的堆栈帧）中可访问的元素，并包含以下部分：

- **局部变量**：列出局部变量。
- **全局变量**：列出全局变量。
- **特性变量**：列出特殊变量。

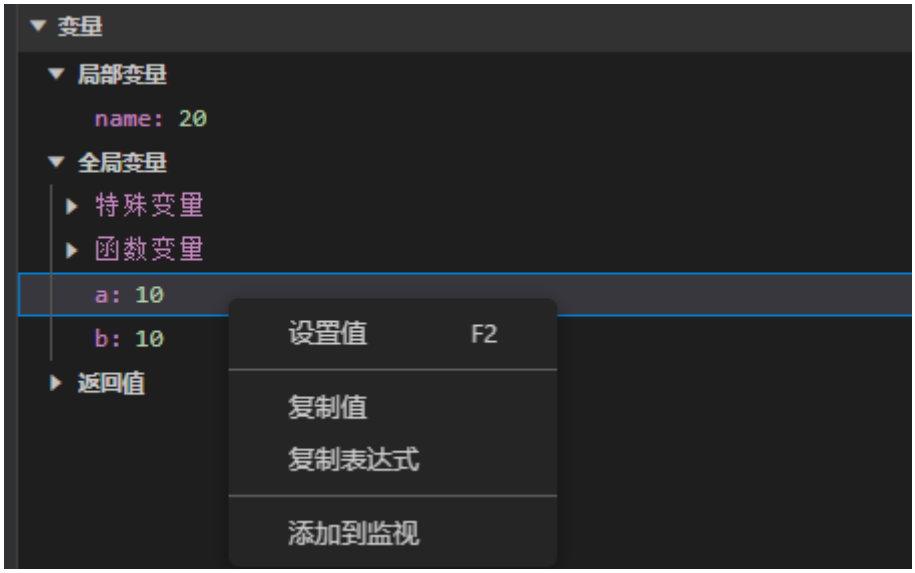
- **返回值**：在调试会话期间，当方法被多次调用时，该部分显示方法在上一步返回的值。这允许您观察值在方法调用之间如何变化。

图 10-52 检查变量



通过在变量上右键单击并从上下文菜单中选择“**设置值**”来修改变量的值。此外，您还可以使用“**复制值**”操作来复制变量的值，或者使用“**复制为表达式**”操作来复制一个用于访问该变量的表达式。

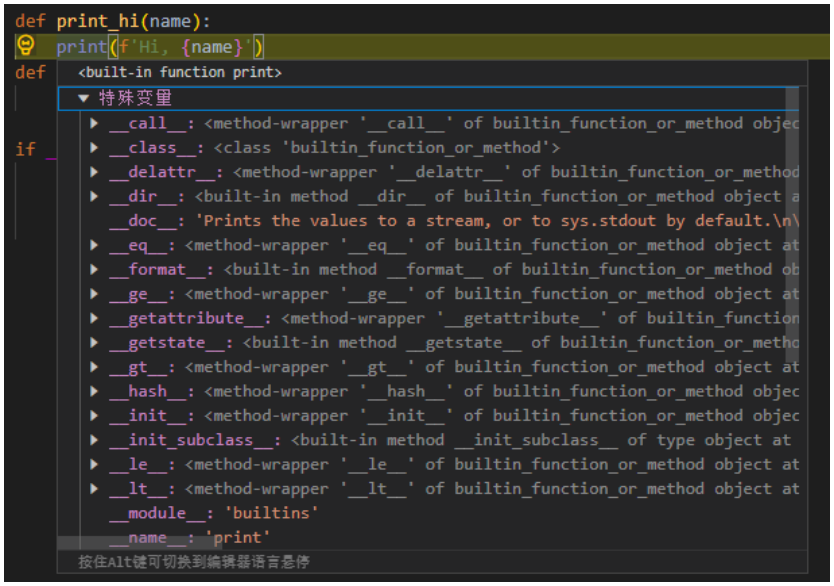
图 10-53 选择“**设置值**”来修改变量的值



还可以在“**运行和调试**”视图的“**监视**”部分中评估和监视变量和表达式。

在CodeArts IDE代码编辑器中直接评估和检查表达式的值，是当程序处于暂停状态时，将鼠标悬停在所需的表达式、变量或方法调用上。

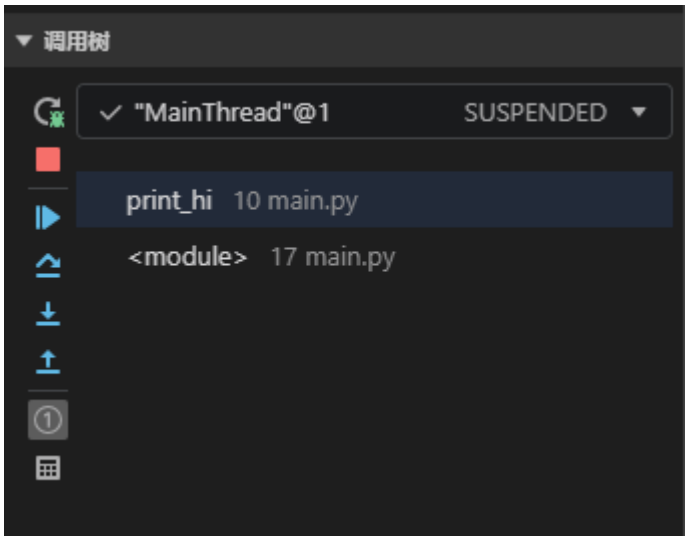
图 10-54 评估和检查表达式的值



检查调用堆栈

“**调用堆栈**”部分列出了当前活动的堆栈帧，每个帧下都分组列出了方法的调用堆栈。

图 10-55 检查调用堆栈



在堆栈帧内可访问的元素会在“**变量**”部分中列出。

- 要在线程之间切换，请使用“调用树”部分顶部的列表。
- 要切换到其他框架，请在“调用树”部分中选择它。这还会在代码编辑器中打开相应的方法调用。

监视

“**监视**”部分允许您在程序运行时跟踪变量或任意表达式的求值结果。

图 10-56 “**监视**”视图

要添加一个表达式，您可以执行以下操作之一：

- 在“**监视**”部分的任意位置双击，或者单击“**添加表达式**”按钮（ ），并在出现的输入框中输入您想要监视的表达式。
- 如要快速为某变量添加监视，请在“**变量**”部分中右键单击变量名，并在上下文菜单中选择“**添加到监视**”。

要删除一个表达式，只需选择它并按“Delete” 键。若要一次性删除所有表达式，请单击“**删除所有表达式**”按钮（ ）。

断点

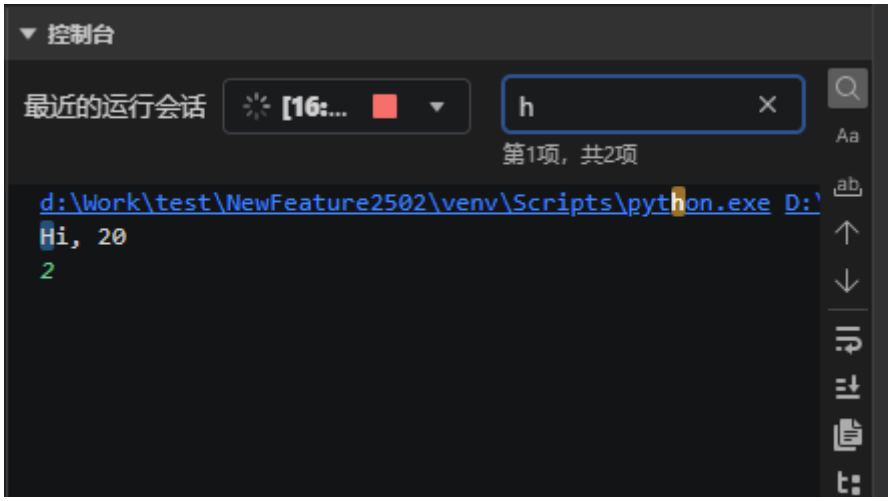
断点部分允许您管理断点，详情信息请参阅**设置不同的断点**。

10.9.6 查看程序输出

在调试会话期间，可以在**控制台**中查看正在运行的程序的输出。

- 要打开**控制台**，请在CodeArts IDE窗口的底部单击“运行和调试”按钮（ ），或者按“Ctrl+Shift+D”/“Shift+Alt+F9”（IDEA快捷键）/”Ctrl+Shift+F8”（IDEA快捷键）。然后展开Console部分。
- 要搜索控制台日志，请单击控制台工具栏上的“在输出中搜索”按钮（ ）。

图 10-57 查看程序输出

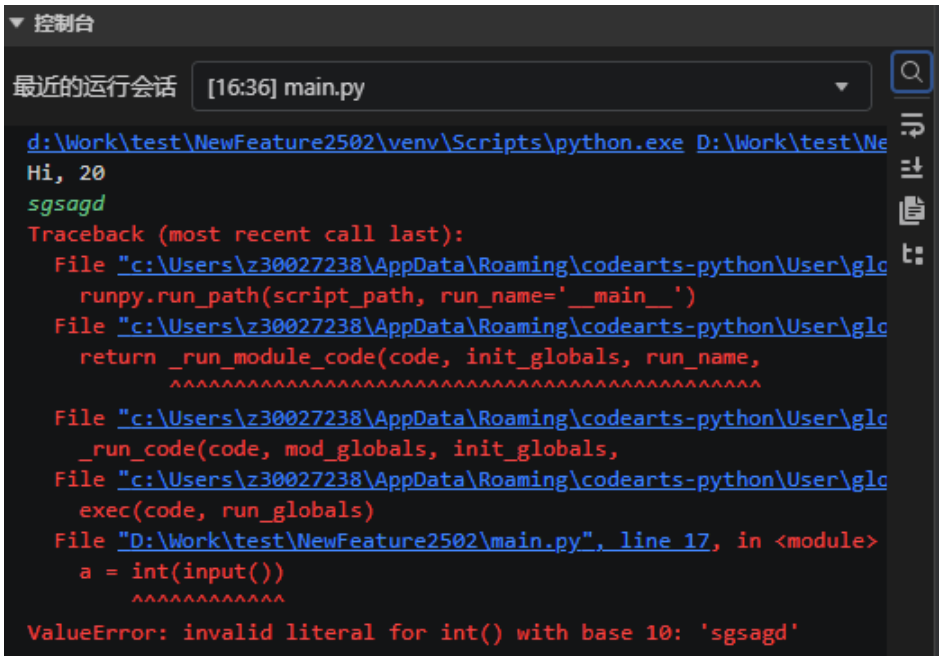


使用控制台工具栏按钮来使用常见的搜索选项：

- 区分大小写( )

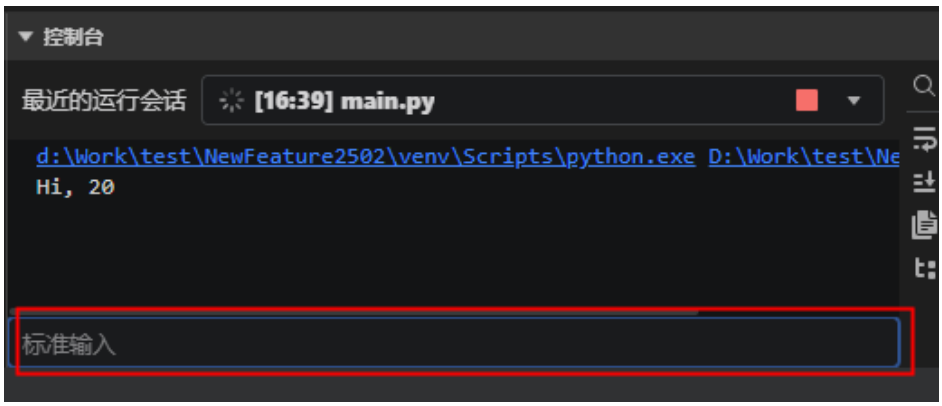
  - 全字匹配(ab)
  - 下一个匹配项/上一个匹配项(/)。
- 如果程序运行过程中出现错误，CodeArts IDE也会在**控制台**中显示错误信息。如果错误涉及项目代码，Ctrl+Click错误中的链接可以快速定位到相应的项目文件。

图 10-58 显示错误信息



- 对于python的输入，在**控制台**的**标准输入**中输入值，然后按”Enter”输入。

图 10-59 标准输入

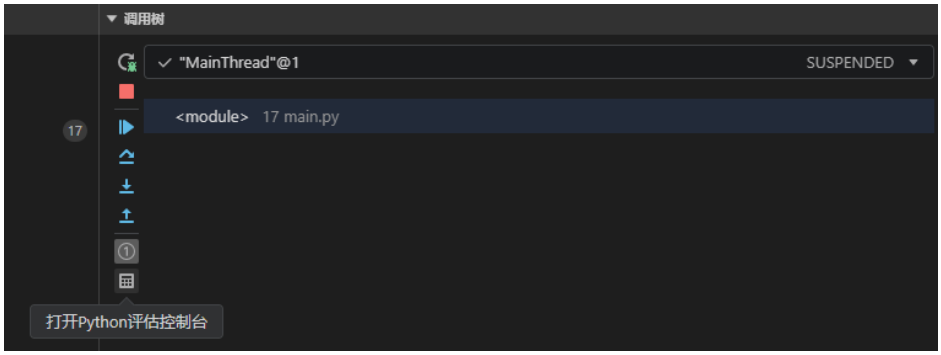


10.9.7 评估表达式

在调试会话期间，使用调试控制台来评估任意表达式。

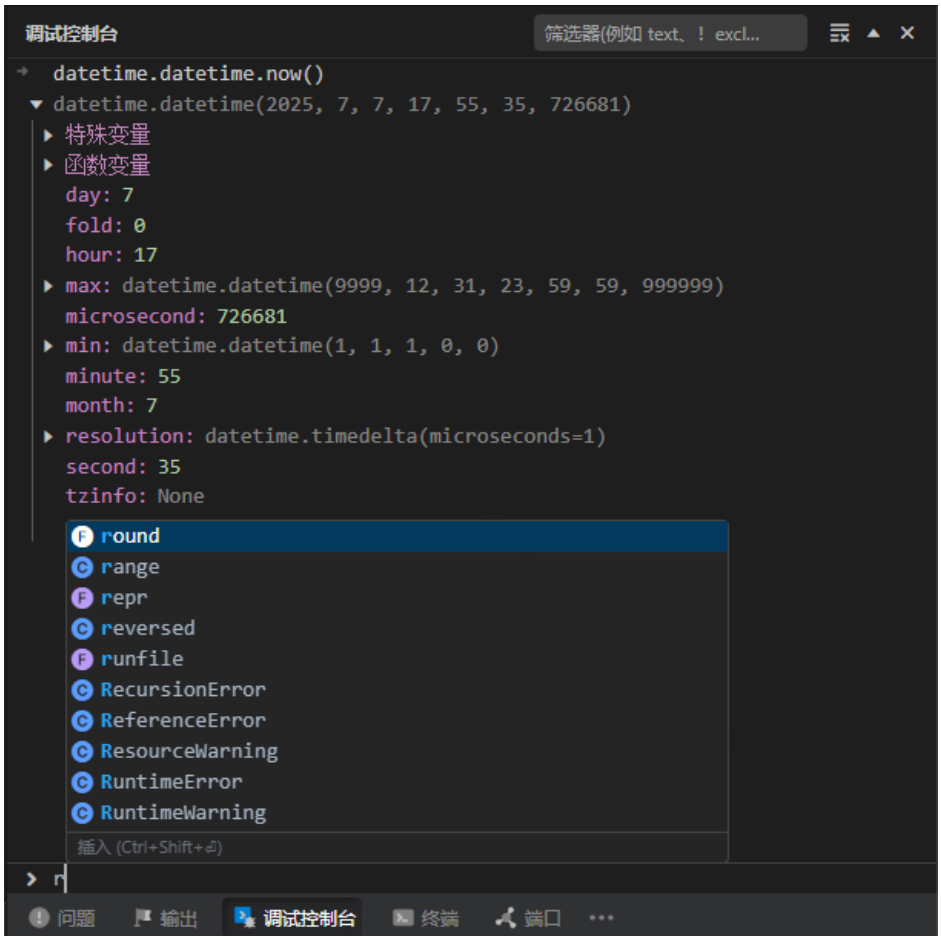
从CodeArts IDE底部面板打开“运行和调试”视图（或按Ctrl+Shift+D），然后单击调用树部分中的“打开Python评估控制台”按钮。

图 10-60 打开 Python 评估控制台



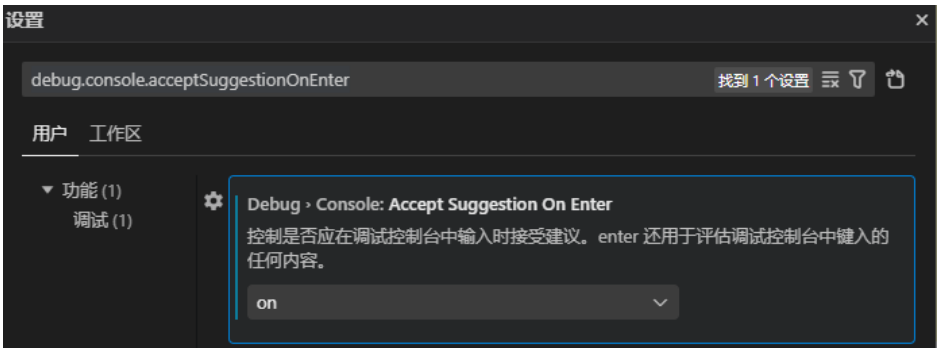
在调试控制台中，在输入的字段中输入一个表达式。

- 要使用代码完成，请按 “Ctrl+I” /“Ctrl+Space”/“Ctrl+Shift+Space” 。
- 要接受完成建议或提交表达式，请按“Ctrl+Shift+Enter”。



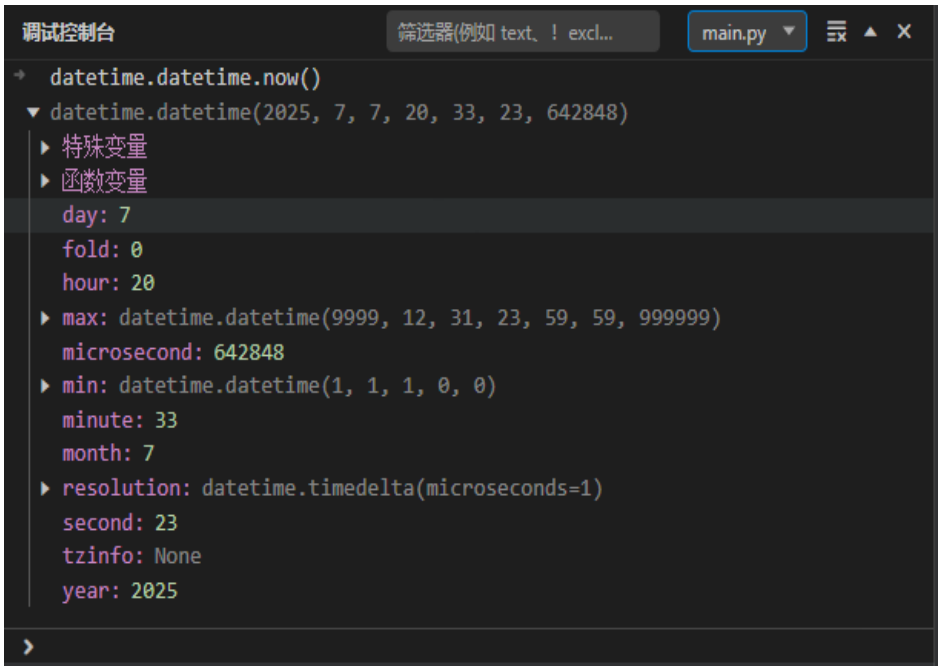
说明

如有需要，可以将接受补全建议和提交表达式的键盘快捷键切换为Enter。在CodeArts IDE的**设置**模块(使用快捷键Ctrl+,打开)，找到并启用debug.console.acceptSuggestionOnEnter选项。



调试控制台会在当前正在运行的调试会话上下文中计算表达式的值。如果有多个调试会话处于活动状态，请使用调试控制台视图顶部的列表在它们之间切换。

图 10-61 计算表达式的值



说明

请注意，**调试**控制台或集成终端中可能会出现中文字符显示不正确的问题。

您可以尝试以下解决方法来修复终端输出：

1. 在 **Windows 控制面板**中，转到“**时钟和区域**”>“**区域**”。
2. 在“**管理**”选项卡上，单击“**更改系统区域设置**”。
3. 在打开的“**区域设置**”对话框中，勾选“**Beta：使用 Unicode UTF-8 提供全球语言支持**”。

然后调整您的启动配置，可以通过将“console”属性设置为“integrated”来在控制台输出里使用集成终端。这样，中文输出可能会正常显示在集成终端中。

10.10 运行 Python 工程启动配置

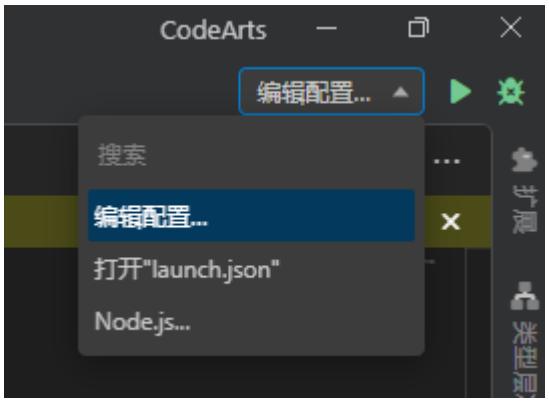
10.10.1 启动配置简介

启动配置允许您配置和保存各种场景下运行或调试的设置细节。

Python 启动配置

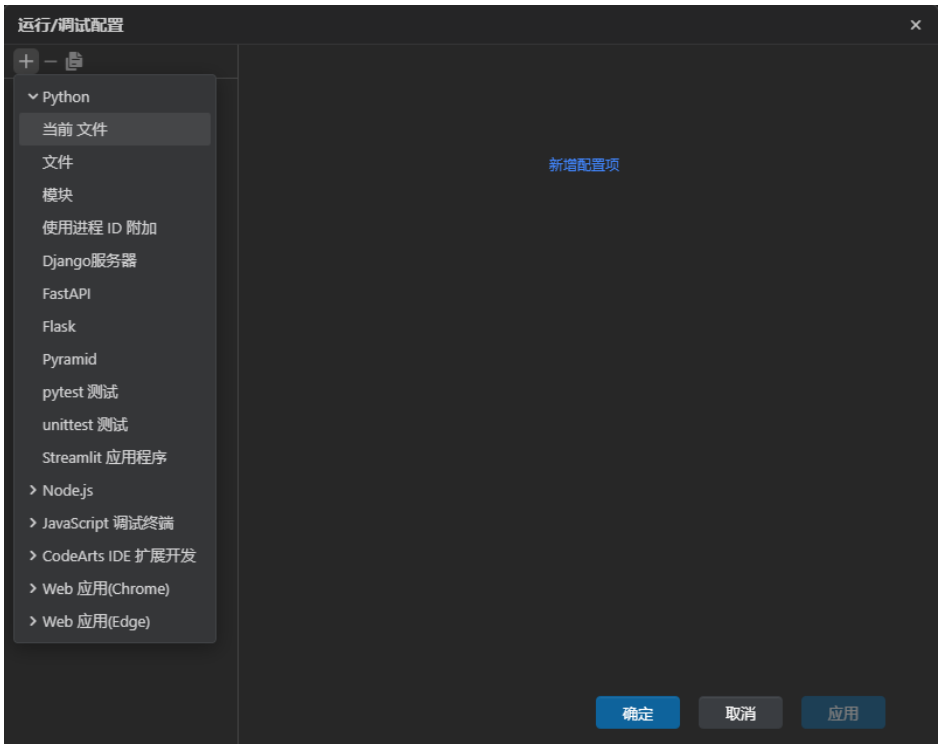
步骤1 在CodeArts IDE主工具栏上，从列表中选择“**编辑配置...**”。

图 10-62 编辑配置



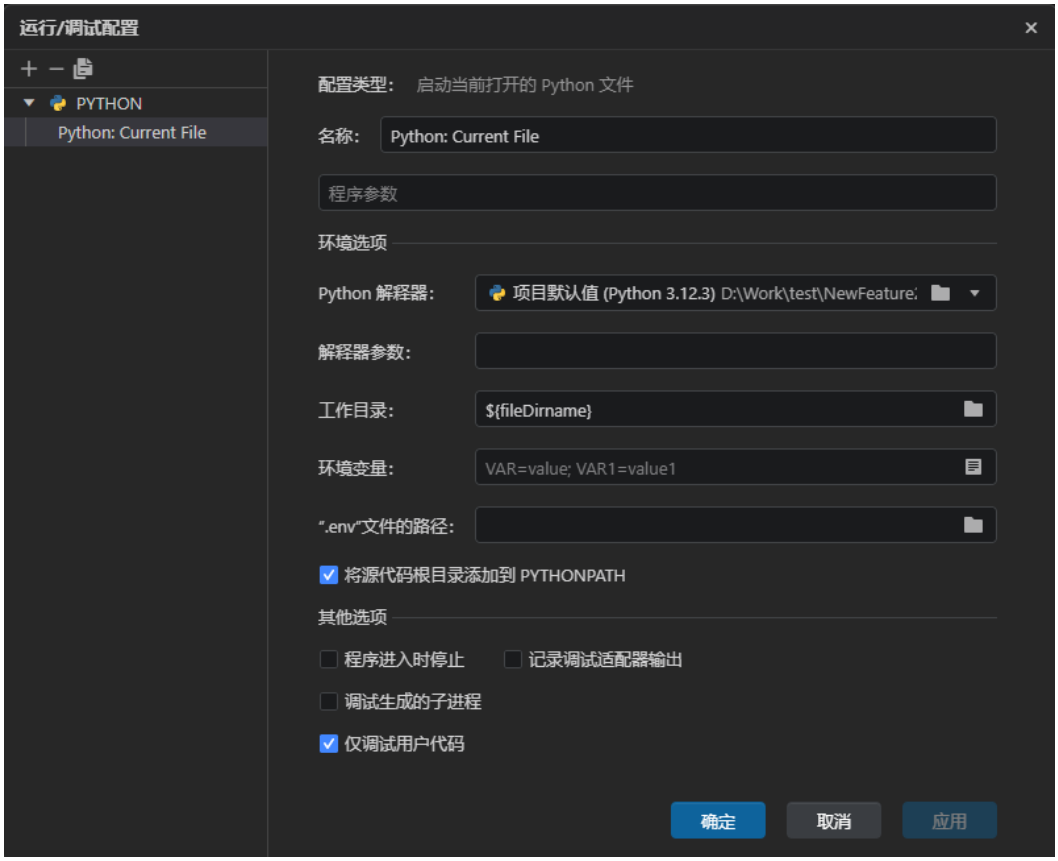
步骤2 在打开的“**运行/调试配置**”对话框中，单击工具栏上的“**新增配置项**”按钮（ ）或使用“**新增配置项**”链接。在出现的列表中，选择“**Python**”条目下所需的启动配置模板。

图 10-63 新增配置项



步骤3 在参数区域中填写启动配置参数。

图 10-64 输入启动配置参数



步骤4 单击“**确定**”以应用更改并关闭对话框。

步骤5 如果要删除启动配置，请在“**运行/调试配置**”对话框中单击工具栏上的“**删除选中的配置项**”按钮（ ）。

----结束

动态启动配置

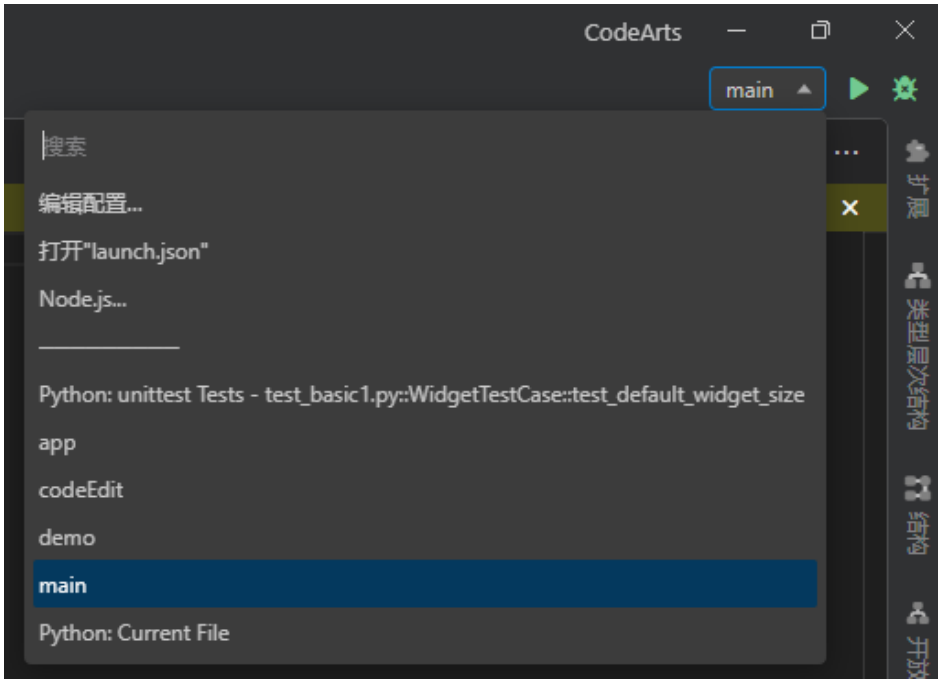
右键单击“资源管理器”或代码编辑器中的任意Python文件，然后从上下文菜单中选择“调试 Python 文件”/“运行 Python 文件”来运行该文件。

此外，您也可以直接通过单击编辑器边栏中的按钮，在代码编辑器中运行Python测试。

在这些情况下，CodeArts IDE会根据运行的文件自动创建**Python文件**、**pytest**或**unittest**启动配置。

然后，从CodeArts IDE主工具栏中选择并运行创建的启动配置。

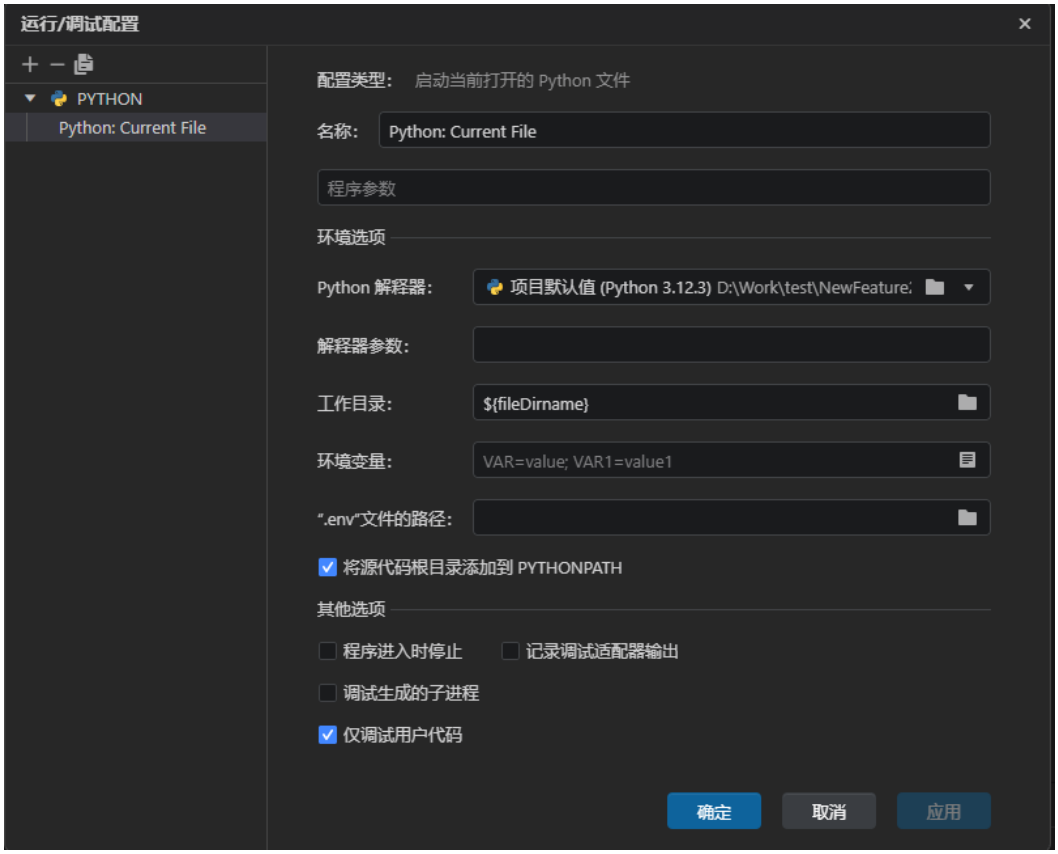
图 10-65 运行创建的启动配置



10.10.2 运行当前的 Python 文件

使用该启动配置运行当前在代码编辑器中打开的Python文件。

图 10-66 运行当前的 Python 文件



说明

若是在没有创建启动配置时快速运行Python文件，可以在资源管理器右键单击该文件或其代码编辑器中右键单击，从上下文菜单中选择**“调试 Python 文件”/“运行 Python 文件”**。CodeArts IDE会自动为此文件创建 **Python文件**启动配置。之后就可以从CodeArts IDE主工具栏选择并运行创建的启动配置。

表 10-5 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于Python文件启动配置，此选项始终设置为“launch”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。

名称	描述
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。
“currentFileConfiguration”	指示启动配置是否为当前文件配置。对于当前文件配置，它始终设置为true。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“program”	被调试文件的路径，默认值${file}解析为当前打开的文件。您可以使用变量来提供。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给被调试程序的命令行参数。
“cwd”	调试程序工作目录的绝对路径。默认值${fileDirname}解析为包含调试文件的文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，此选项设置为“false”。
“justMyCode”	如果设置为“true”（默认），则仅显示和调试用户编写的代码。否则，显示和调试所有代码，包括库调用。
“pythonArgs”	传递给Python解释器的命令行参数。要将参数传递给被调试程序，请使用“args”。
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

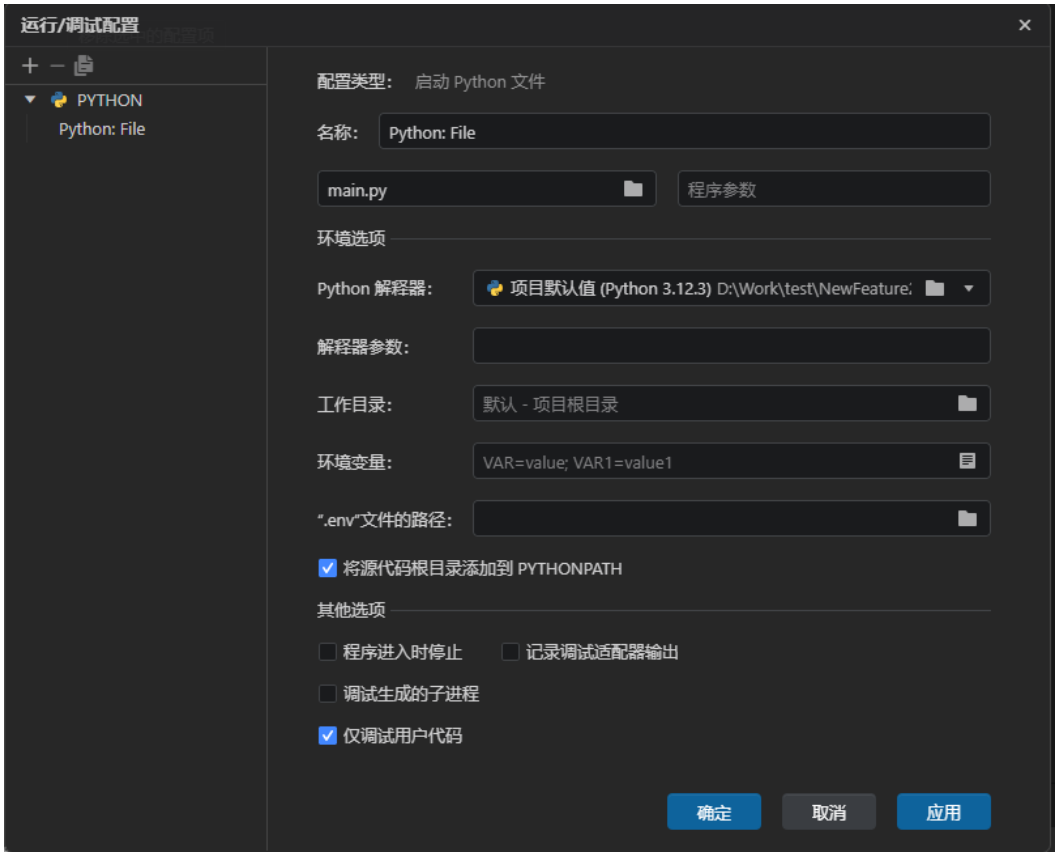
```
{
    "request": "launch",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "currentFileConfiguration": true,
    "stopOnEntry": false,
    "program": "${file}",
    "env": {},
    "type": "python",
    "logToFile": false,
```

```
    "args": [],
    "cwd": "${fileDirname}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: Current File",
    "showReturnValue": true
}
```

10.10.3 运行任意的 Python 文件

使用该配置来运行任意Python文件。

图 10-67 运行任意的 Python 文件



说明

要在没有手动创建启动配置的时候快速运行Python文件，可以在资源管理器右键单击该文件或其代码编辑器中右键单击，从上下文菜单中选择**“调试 Python 文件”/“运行 Python 文件”**。CodeArts IDE会自动为此文件创建 Python文件启动配置。之后您就可以从CodeArts IDE主工具栏选择并运行创建的启动配置。

表 10-6 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。

名称	描述
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于Python文件启动配置，此选项始终设置为“launch”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“program”	被调试文件的路径，默认值${file}解析为当前打开的文件。您可以使用变量来提供。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给被调试程序的命令行参数。
“cwd”	调试程序工作目录的绝对路径。默认值${workspaceFolder}解析为包含调试文件的文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，此选项设置为“false”。
“justMyCode”	如果设置为 “true”（默认），则仅显示和调试用户编写的代码。 否则，显示和调试所有代码，包括库调用。
“pythonArgs”	传递给Python解释器的命令行参数。要将参数传递给被调试程序，请使用“args”。
“showReturnValue”	如果设置为 “true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。


```
{
    "request": "launch",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "program": "example.py",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: File",
    "showReturnValue": true
}
```

10.10.4 运行 Python 模块

使用该配置来运行任意Python模块。

图 10-68 运行 Python 模块

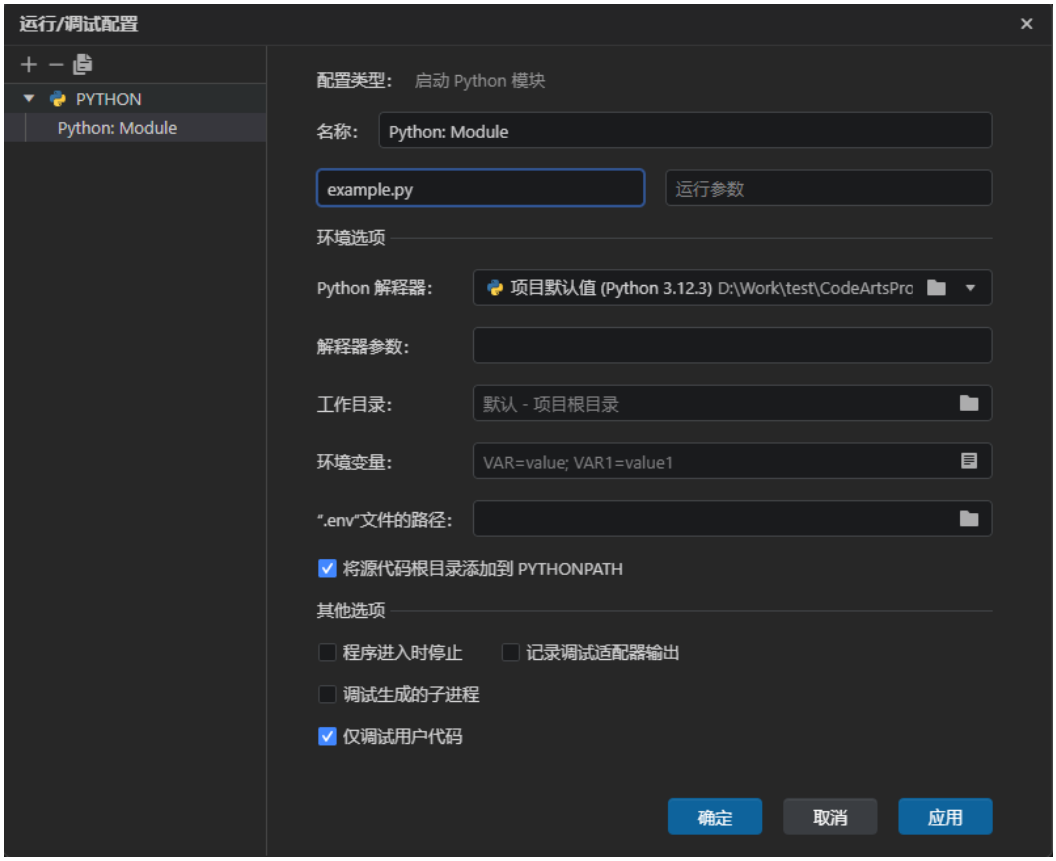


表 10-7 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。

名称	描述
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于Python文件启动配置，此选项始终设置为“launch”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“module”	被调试模块的路径。您可以使用变量来提供路径。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给被调试程序的命令行参数。
“cwd”	调试程序工作目录的绝对路径。默认值${workspaceFolder}解析为包含调试文件的文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，此选项设置为“false”。
“justMyCode”	如果设置为 “true”（默认），则仅显示和调试用户编写的代码。 否则，显示和调试所有代码，包括库调用。
“pythonArgs”	传递给Python解释器的命令行参数。要将参数传递给被调试程序，请使用“args”。
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

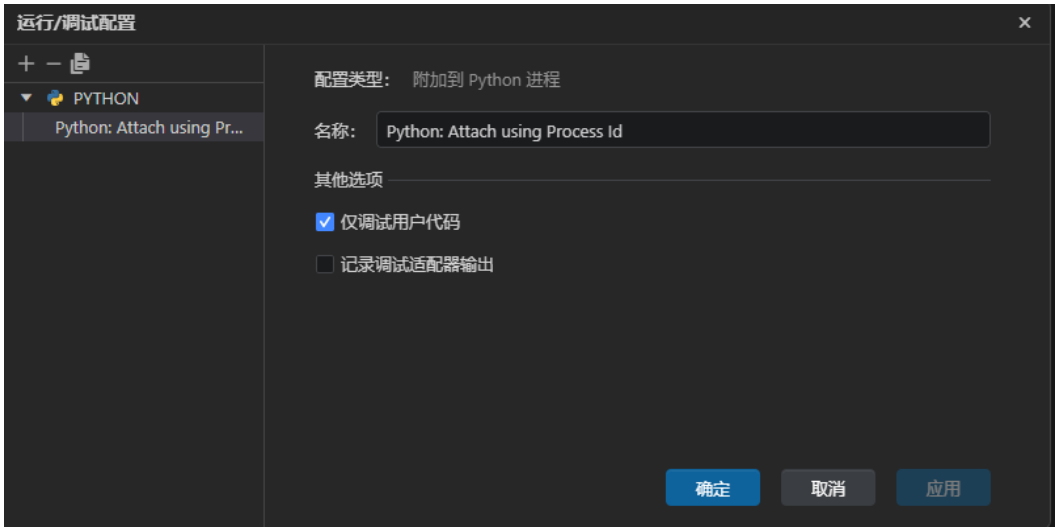
```
{
    "request": "launch",
```

```
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "module": "example.py",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: Module",
    "showReturnValue": true
}
```

10.10.5 运行附加进程

使用该启动配置将调试器附加到已运行的Python程序。

图 10-69 运行附加进程



当启动该配置时，CodeArts IDE会提示您选择要附加的进程。

图 10-70 选择附加的进程

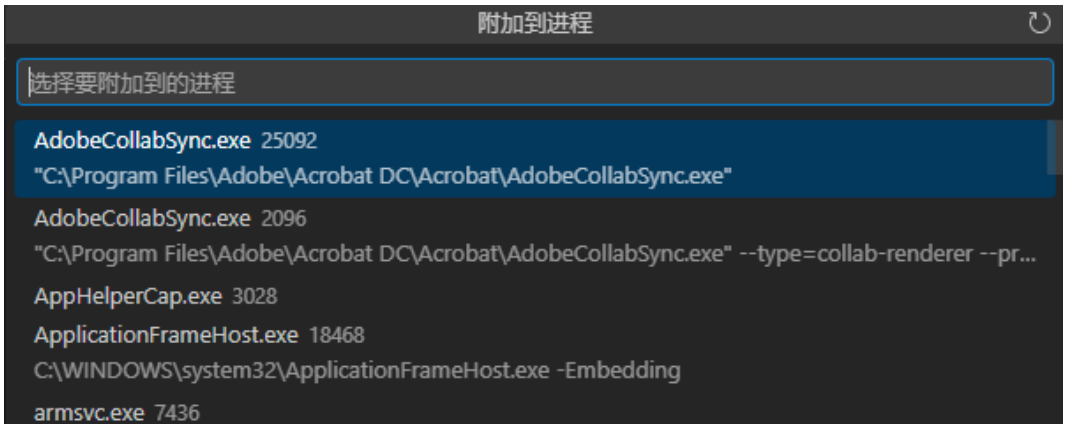


表 10-8 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于附加到进程启动配置，此选项始终设置为“attach”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“processId”	正在运行的Python程序的进程标识符 (PID)。设置默认值“${command:pickProcess}” 时，CodeArts IDE会提示您选择要附加到的进程。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“justMyCode”	如果设置为“true”（默认），则仅显示和调试用户编写的代码。否则将显示和调试所有包括库调用的代码。
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "request": "attach",
    "jinja": true,
    "justMyCode": true,
    "processId": "${command:pickProcess}",
    "name": "Python: Attach using Process Id",
    "type": "python",
    "logToFile": false,
    "showReturnValue": true
}
```

10.10.6 运行 Django 应用

该启动配置执行“python manage.py runserver”命令来启动内置的Django开发服务器。

图 10-71 运行 Django 应用

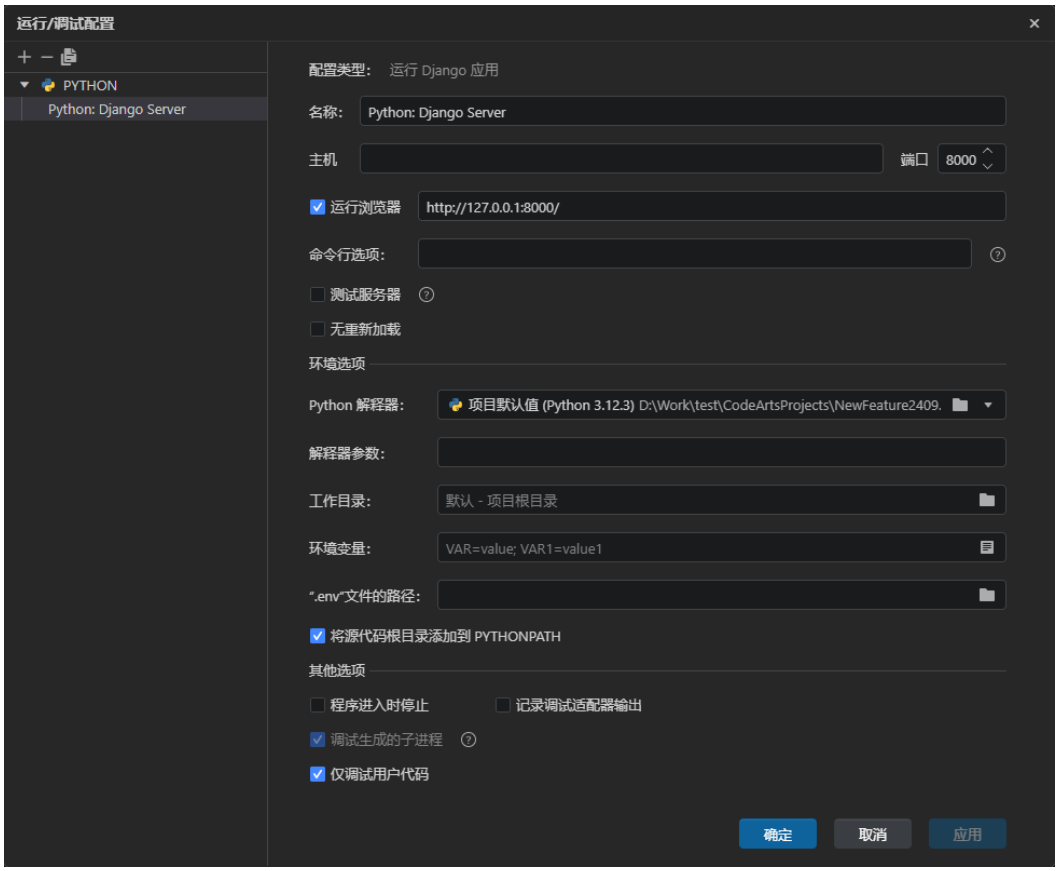


表 10-9 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“addSourceRootsToPythonpath”	如果设置为true，项目的源代码根文件夹将被添加到PYTHONPATH环境变量中，从而扩展模块文件的默认搜索路径。有关使用项目源根目录的详细信息，请参阅**配置项目结构**。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于Python文件启动配置，此选项始终设置为“launch”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。

名称	描述
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“program”	管理 `manage.py` 的路径，这是 Django 的命令行管理工具。默认情况下，该路径设置为 `${workspaceFolder}\\manage.py`，以定位项目根目录下的 `manage.py`。您可以使用变量来提供路径。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给manage.py的命令行参数。默认情况下，设置为runserver，它将启动内置的 Django 开发服务器。
“cwd”	调试程序工作目录的绝对路径。默认值${workspaceFolder}解析为包含调试文件的文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。由于 Django 严重依赖子进程，因此该选项始终设置为true。
“justMyCode”	如果设置为 “true”（默认），则仅显示和调试用户编写的代码。否则，显示和调试所有代码，包括库调用。
“django”	在此节点下，提供了Django开发服务器的参数设置。 - host/port：指定运行Django服务器的主机和端口。 - runBrowser：若设置为true，将在默认浏览器中打开runBrowserUri下提供的URI。 - commandLineOptions：为manage.py工具提供的命令行选项。 - testServer：若设置为true，将使用fixtures下提供的固定数据集，以测试数据库启动Django开发服务器。（更多关于测试服务器使用的详情，请参阅**Django文档**）。 - noReload：若设置为true，当服务器运行时，对Python代码的任何更改若相关模块已加载至内存中，则不会生效（此选项在testServer设置为false时可用）。 - runBrowserUri：在默认浏览器中打开的应用页面URI（此选项在runBrowser设置为true时可用）。 - fixtures：提供给测试数据库的数据固定数据集列表。
“pythonArgs”	传递给Python解释器的命令行参数。要将参数传递给被调试程序，请使用“args”。
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "addSourceRootsToPythonpath": true,
    "request": "launch",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "envFile": "${workspaceFolder}/.env",
    "stopOnEntry": false,
    "program": "${workspaceFolder}/manage.py",
    "env": {},
    "type": "python",
    "logToFile": false,
    "cwd": "${workspaceFolder}",
    "subProcess": true,
    "justMyCode": true,
    "django": {
        "port": 8000,
        "host": "127.0.0.1",
        "runBrowser": true,
        "commandLineOptions": [],
        "testServer": false,
        "noReload": false,
        "runBrowserUri": "http://127.0.0.1:8000/",
        "fixtures": []
    },
    "pythonArgs": [],
    "name": "Python: Django Server",
    "showReturnValue": true
}
```

10.10.7 运行 FastAPI 应用

使用此启动配置来运行FastAPI应用程序。该配置将执行“uvicorn”命令，启动一个运行应用程序的**uvicorn**服务器。

图 10-72 运行 FastAPI 应用

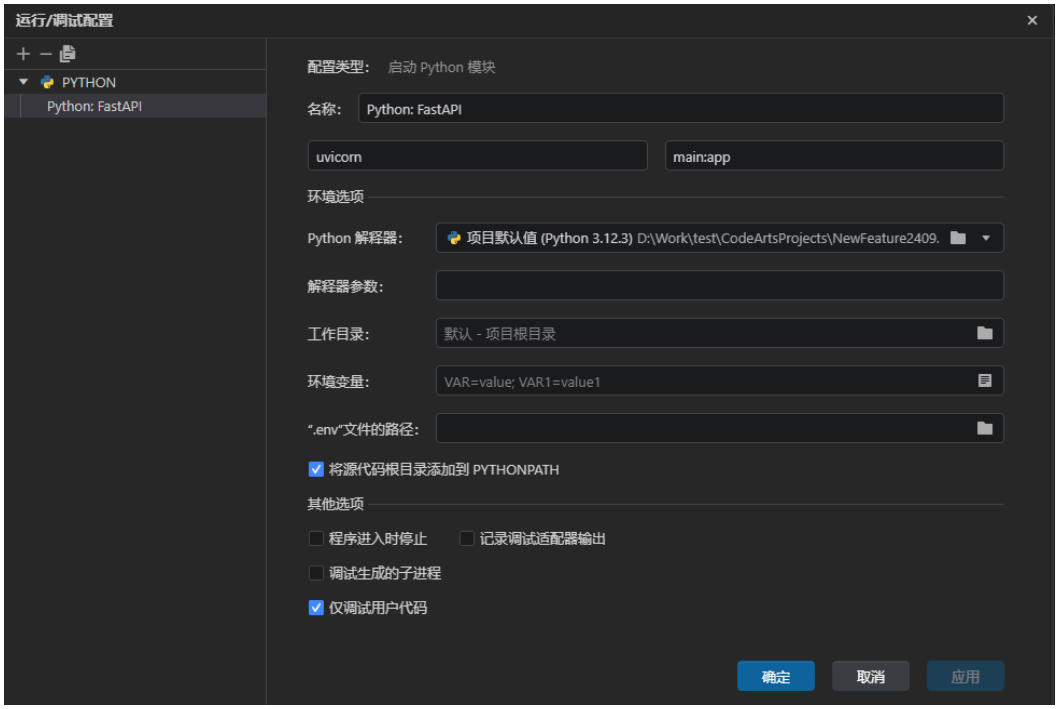


表 10-10 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于FastAPI应用的启动配置，此选项始终设置为“launch”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“module”	用于运行应用程序服务器的“uvicorn”模块的路径，默认情况下设置为“uvicorn”。您可以使用变量来提供路径。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。

名称	描述
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给被调试程序的命令行参数。
“cwd”	调试程序工作目录的绝对路径。默认值${workspaceFolder}解析为包含调试文件的文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，此选项设置为“false”。
“justMyCode”	如果设置为 “true”（默认），则仅显示和调试用户编写的代码。否则，显示和调试所有代码，包括库调用。
“pythonArgs”	传递给Python解释器的命令行参数。要将参数传递给被调试程序，请使用“args”。
“showReturnValue”	如果设置为 “true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "request": "launch",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "module": "uvicorn",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [
        "main:app"
    ],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: FastAPI",
    "showReturnValue": true
}
```

10.10.8 运行 Flask 应用

使用此启动配置来运行Flask应用程序。该配置将执行“python Flask run”命令，启动内置Flask开发服务器。

图 10-73 运行 Flask 应用

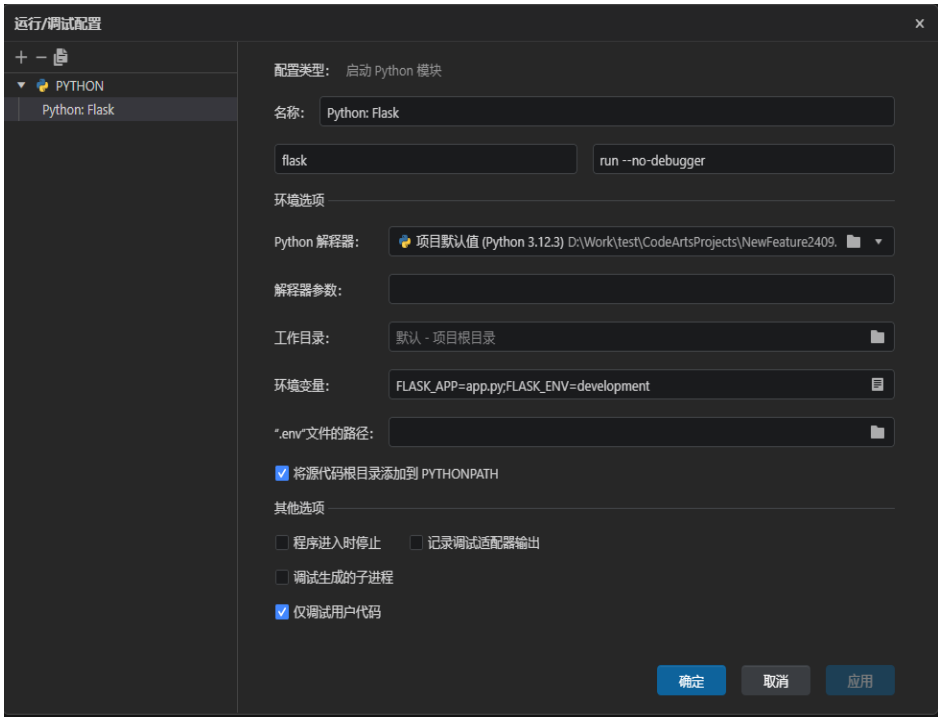


表 10-11 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于Flask应用的启动配置，此选项始终设置为“launch”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“module”	用于运行Flask应用服务器的模块的名称，默认情况下设置为flask。您可以使用变量来提供路径。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。默认设置为“{"FLASK_APP": "app.py","FLASK_ENV": "development"}”。

名称	描述
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给Flask的命令行参数。默认情况设置为“["run","--no-debugger"]”，这会在启动Flask应用程序服务器的同时禁用内置Flask调试器。
“cwd”	调试程序工作目录的绝对路径。默认值${workspaceFolder}解析为包含调试文件的文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，此选项设置为“false”。
“justMyCode”	如果设置为“true”（默认），则仅显示和调试用户编写的代码。否则，显示和调试所有代码，包括库调用。
“pythonArgs”	传递给Python解释器的命令行参数。要将参数传递给被调试程序，请使用“args”。
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "request": "launch",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "module": "flask",
    "env": {
        "FLASK_APP": "app.py",
        "FLASK_ENV": "development"
    },
    "type": "python",
    "logToFile": false,
    "args": [
        "run",
        "--no-debugger"
    ],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: Flask",
    "showReturnValue": true
}
```

10.10.9 运行 Pyramid 应用

使用此启动配置来运行Pyramid应用程序。该配置将执行“python pserve”命令启动内置Pyramid开发服务器。

图 10-74 运行 Pyramid 应用

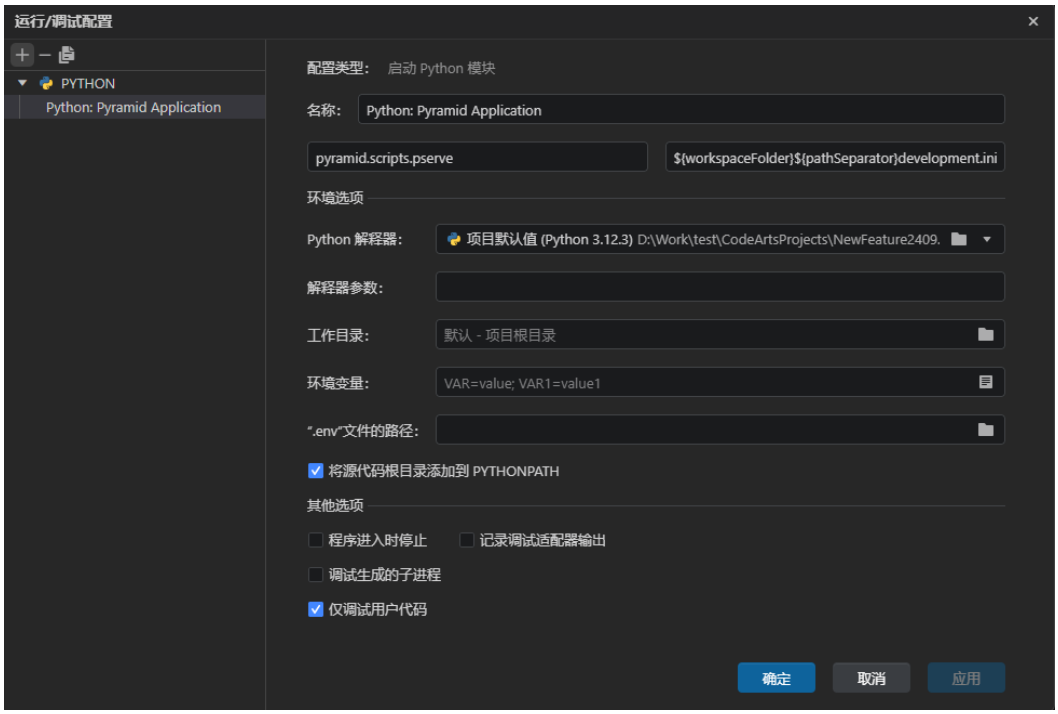


表 10-12 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于Pyramid应用的启动配置，此选项始终设置为“launch”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。

名称	描述
“module”	用于运行Pyramid应用程序服务器的模块的名称。默认设置为“pyramid.scripts.pserve”。您可以使用变量来提供路径。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。默认设置为 “{"FLASK_APP": "app.py","FLASK_ENV": "development"}”。
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给pserve的命令行参数。默认情况下，此参数设置为【${workspaceFolder}${pathSeparator}development.ini】，这将解析为项目根目录下包含应用程序配置参数的development.ini文件。
“cwd”	调试程序工作目录的绝对路径。默认值 “${workspaceFolder}” 解析为项目根文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，此选项设置为“false”。
“justMyCode”	如果设置为“true”（默认），则仅显示和调试用户编写的代码。否则，显示和调试所有代码，包括库调用。
“pythonArgs”	传递给Python解释器的命令行参数。要将参数传递给被调试程序，请使用“args”。
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

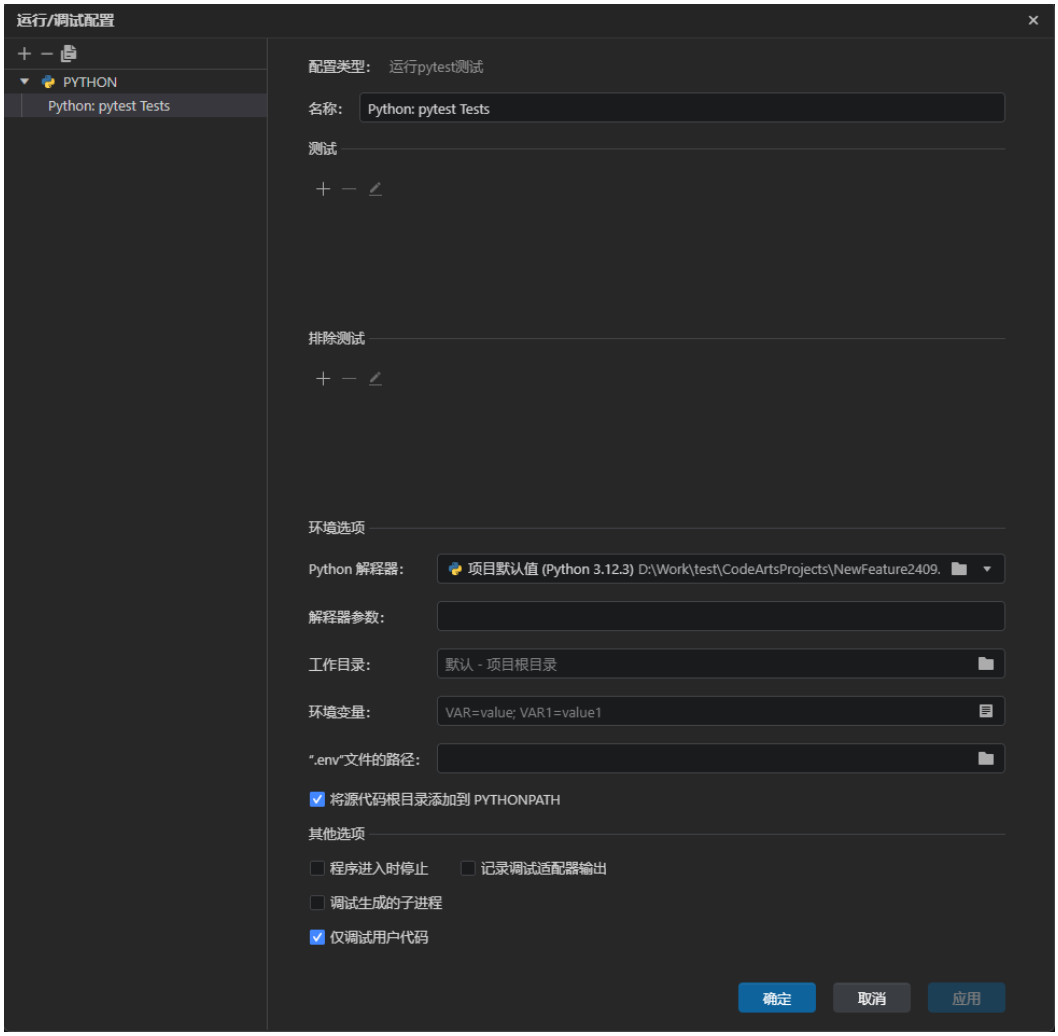
```
{
    "request": "launch",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "module": "pyramid.scripts.pserve",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [
        "${workspaceFolder}\\development.ini"
    ],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
```

```
    "pythonArgs": [],
    "name": "Python: Pyramid Application",
    "showReturnValue": true
}
```

10.10.10 运行 pytest 测试

使用该启动配置来运行pytest测试。

图 10-75 运行 pytest 测试



说明

若是没有手动创建启动配置时快速运行pytest测试，请在测试文件的代码编辑器中，单击测试类声明旁边的“**全部运行**”按钮（ ）（以运行类中的所有测试），或测试方法旁边的“**运行方法**”按钮（ ）（仅运行单个测试）。

CodeArts IDE会自动创建相应的pytest启动配置并将其显示在配置列表中。

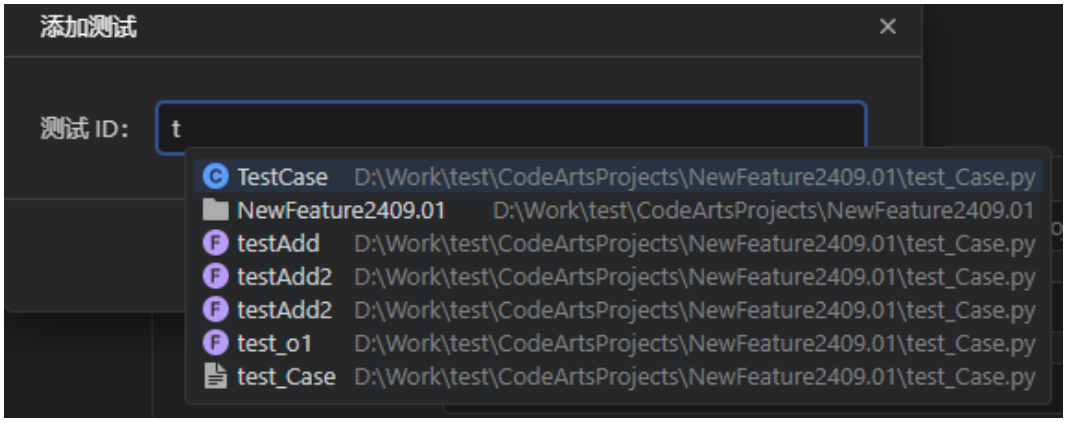
启动配置中包含测试/排除测试

在“**测试/排除测试**”区域中，您可以列出要包含在启动配置范围内的测试或要排除的测试。

步骤1 要向列表中添加测试，请单击“**添加测试**”按钮（ ）。

步骤2 在打开的“**添加测试**”窗口中，找到所需的测试。使用代码完成功能（“Ctrl+I” / “Ctrl+空格键” / “Ctrl+Shift+空格键”）让CodeArts IDE列出可用的测试。

图 10-76 添加测试



步骤3 在“**添加测试**”窗口中，单击“保存”以将所选测试添加到列表中。

步骤4 要从列表中删除测试，请选择它并单击“删除选定项”按钮（ ）。

----结束

表 10-13 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于pytest启动配置，此选项始终设置为“test”。
“testIds”	要包含在启动配置范围中的测试ID列表。ID的格式如下：“test_file_name::test_class_name::test_method_name”。
“excludeTestIds”	要从启动配置范围中排除的测试ID列表。ID的格式如下：“test_file_name::test_class_name::test_method_name”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。

名称	描述
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“cwd”	调试程序工作目录的绝对路径。默认值“${workspaceFolder}” 解析为项目根文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，此选项设置为“false”。
“justMyCode”	如果设置为“true”（默认），则仅显示和调试用户编写的代码。否则，显示和调试所有代码，包括库调用。
“provider”	测试框架。对于pytest启动配置，此选项始终设置为“PYTEST”。
“pythonArgs”	传递给Python解释器的命令行参数。
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

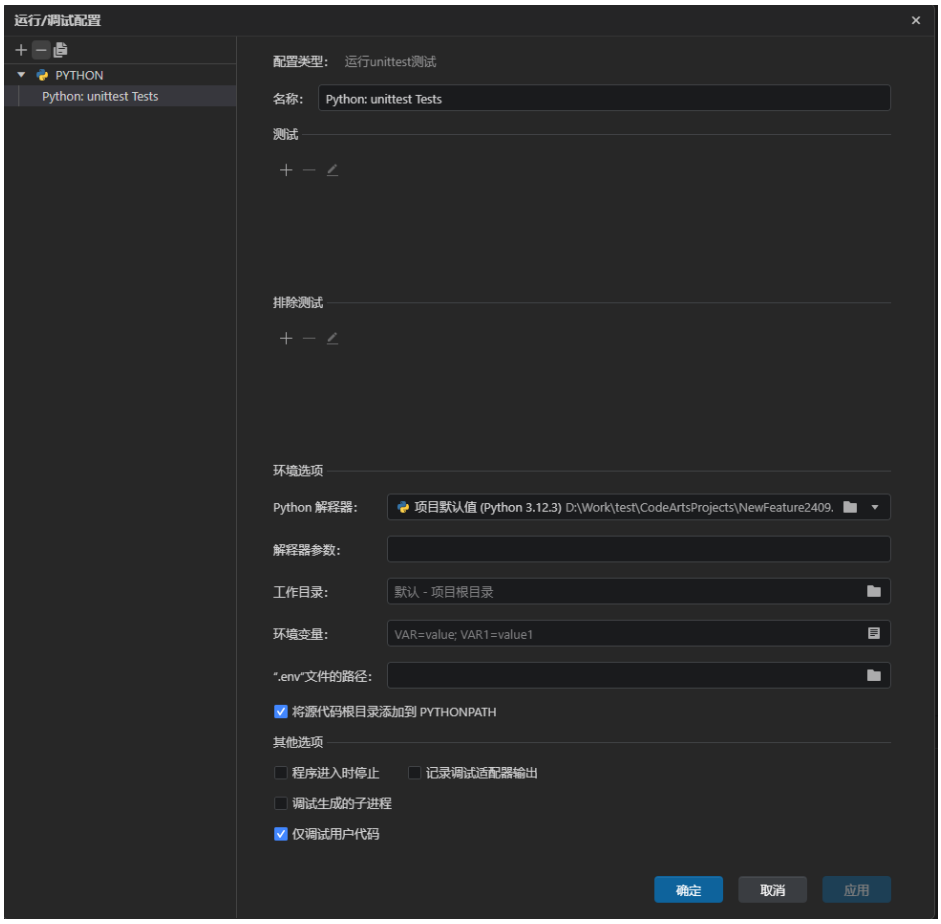
以下是一个可运行的启动配置示例。

```
{
  "excludeTestIds": [],
  "request": "test",
  "jinja": true,
  "python": "${command:python.interpreterPath}",
  "stopOnEntry": false,
  "env": {},
  "type": "python",
  "logToFile": false,
  "testIds": [
     "test_file_name::test_class_name::test_method_name"
  ],
  "cwd": "${workspaceFolder}",
  "subProcess": false,
  "justMyCode": true,
  "provider": "PYTEST",
  "pythonArgs": [],
  "name": "Python pytest tests",
  "showReturnValue": true
}
```

10.10.11 运行 unittest 测试

使用该启动配置来运行**unittest**测试。

图 10-77 运行 unittest 测试



说明

要在没有手动创建启动配置的时候快速运行unittest测试，请在测试文件的代码编辑器中，单击测试类声明旁边的“**全部运行**”按钮（）（以运行类中的所有测试），或测试方法旁边的“**运行方法**”按钮（）（仅运行单个测试）。

```
1   import inc_dec    # The code to test
2   import unittest   # The test framework
3
    超类 (2)
4   class Test_TestIncrementDecrement(unittest.TestCase):
5       def test_increment(self):
6           self.assertEqual(inc_dec.increment(3), 4)
7
8       def test_decrement(self):
9           self.assertEqual(inc_dec.decrement(3), 4)
10
11  if __name__ == '__main__':
12      unittest.main()
```

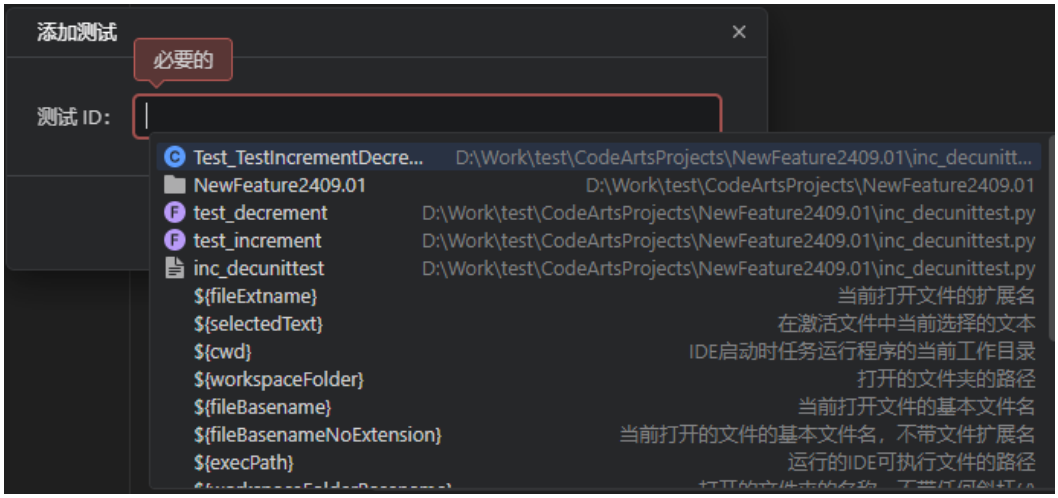
CodeArts IDE会自动创建相应的unittest启动配置并将其显示在配置列表中。

启动配置中包含测试/排除测试

在“**测试/排除测试**”区域中，您可以列出要包含在启动配置范围内的测试或要排除的测试。

步骤1 要向列表中添加测试，请单击“**Add New**”按钮（）。

步骤2 在打开的“**添加测试**”窗口中，找到所需的测试。**使用Python补全代码**（“Ctrl+I”/“Ctrl+空格键”/“Ctrl+Shift+空格键”）让CodeArts IDE列出可用的测试。



步骤3 在“**添加测试**”窗口中，单击“**保存**”以将所选测试添加到列表中。

步骤4 要从列表中删除测试，请选择它并单击“删除选定项”按钮（）。

----结束

表 10-14 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于unittest启动配置，此选项始终设置为“test”。
“testIds”	要包含在启动配置范围中的测试ID列表。ID的格式如下：“test_file_name::test_class_name::test_method_name”。
“excludeTestIds”	要从启动配置范围中排除的测试ID列表。ID的格式如下：“test_file_name::test_class_name::test_method_name”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试，例如在**Flask**应用程序中。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“cwd”	调试程序工作目录的绝对路径。默认值 “${workspaceFolder}” 解析为项目根文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，此选项设置为“false”。
“justMyCode”	如果设置为“true”（默认），则仅显示和调试用户编写的代码。否则，显示和调试所有代码，包括库调用。
“provider”	测试框架。对于unittest启动配置，此选项始终设置为“UNITTEST”。
“pythonArgs”	传递给Python解释器的命令行参数。

名称	描述
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

```
{
  "excludeTestIds": [],
  "request": "test",
  "jinja": true,
  "python": "${command:python.interpreterPath}",
  "stopOnEntry": false,
  "env": {},
  "type": "python",
  "logToFile": false,
  "testIds": [
    "test_file_name::test_class_name::test_method_name"
  ],
  "cwd": "${workspaceFolder}",
  "subProcess": false,
  "justMyCode": true,
  "provider": "UNITTEST",
  "pythonArgs": [],
  "name": "Python unittest tests",
  "showReturnValue": true
}
```

10.10.12 运行 Streamlit 应用

使用此启动配置来启动使用Streamlit框架创建的应用程序。该配置会执行带有指定参数的`streamlit run`命令。

图 10-78 运行 Streamlit 应用

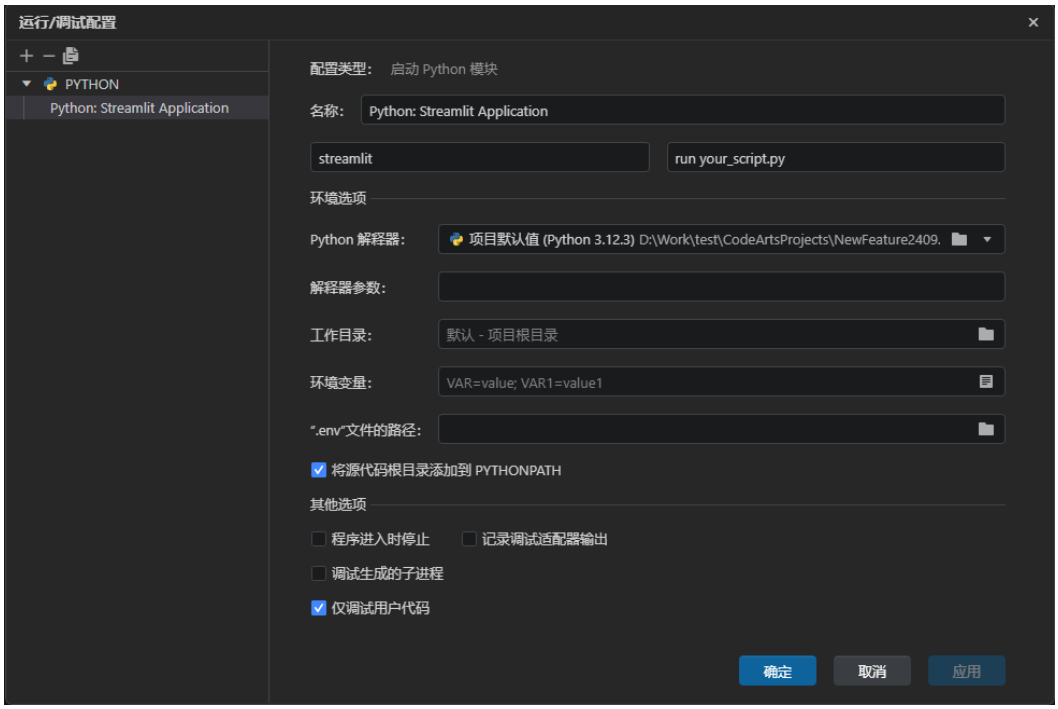


表 10-15 启动配置属性

名称	描述
“type”	调试器的类型。对于运行和调试Python代码，应将其设置为“python”。
“name”	启动配置的名称。
“addSourceRootsToPythonpath”	如果设置为true，项目的源代码根文件夹将被添加到PYTHONPATH环境变量中，从而扩展模块文件的默认搜索路径。有关使用项目源根目录的详细信息，请参阅**配置项目结构**。
“request”	调试模式，可以是“launch”（在program中指定的文件或当前文件上启动调试器）、“attach”（将调试器附加到已经运行的进程）或“test”（运行单元测试）。对于*Streamlit 应用程序*启动配置，此选项始终设置为“launch”。
“jinja”	当设置为“true”（默认）时，启用对**Jinja**模板的调试。
“python”	Python可执行文件的路径。默认值“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**Python环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“module”	用于运行Streamlit应用服务器的模块名称。默认情况下，此名称设置为streamlit。您可以使用变量来提供路径。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“envFile”	环境变量定义文件的路径。默认情况下，此路径设置为${workspaceFolder}/.env指向.env项目根目录下的文件。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给Streamlit的命令行参数。默认情况下，此参数设置为【"run","your_script.py"】，这将使用提供的脚本执行Streamlit运行命令
“cwd”	调试程序工作目录的绝对路径。默认值${workspaceFolder}解析为包含调试文件的文件夹。您可以使用变量来提供路径。
“subProcess”	指定是否启用子进程调试。默认情况下，该选项设置为false。
“justMyCode”	如果设置为 “true”（默认），则仅显示和调试用户编写的代码。否则，显示和调试所有代码，包括库调用。

名称	描述
“pythonArgs”	传递给Python解释器的命令行参数。要将参数传递给被调试程序，请使用“args”。
“showReturnValue”	如果设置为“true”（默认），则在“**运行和调试**”视图中逐步执行时显示函数的返回值。

启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "addSourceRootsToPythonpath": true,
    "request": "launch",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "envFile": "${workspaceFolder}/.env",
    "stopOnEntry": false,
    "module": "streamlit",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [
        "run",
        "your_script.py"
    ],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: Streamlit Application",
    "showReturnValue": true
}
```

11 使用 CodeArts IDE for RemoteShell

11.1 CodeArts IDE for RemoteShell 概述

CodeArts IDE for RemoteShell提供了基于SSH协议访问已绑定EIP的华为云ECS主机的文件系统和终端的能力，以及基于kubectl访问已绑定EIP的华为云容器集群的能力，便于用户访问和使用资源。

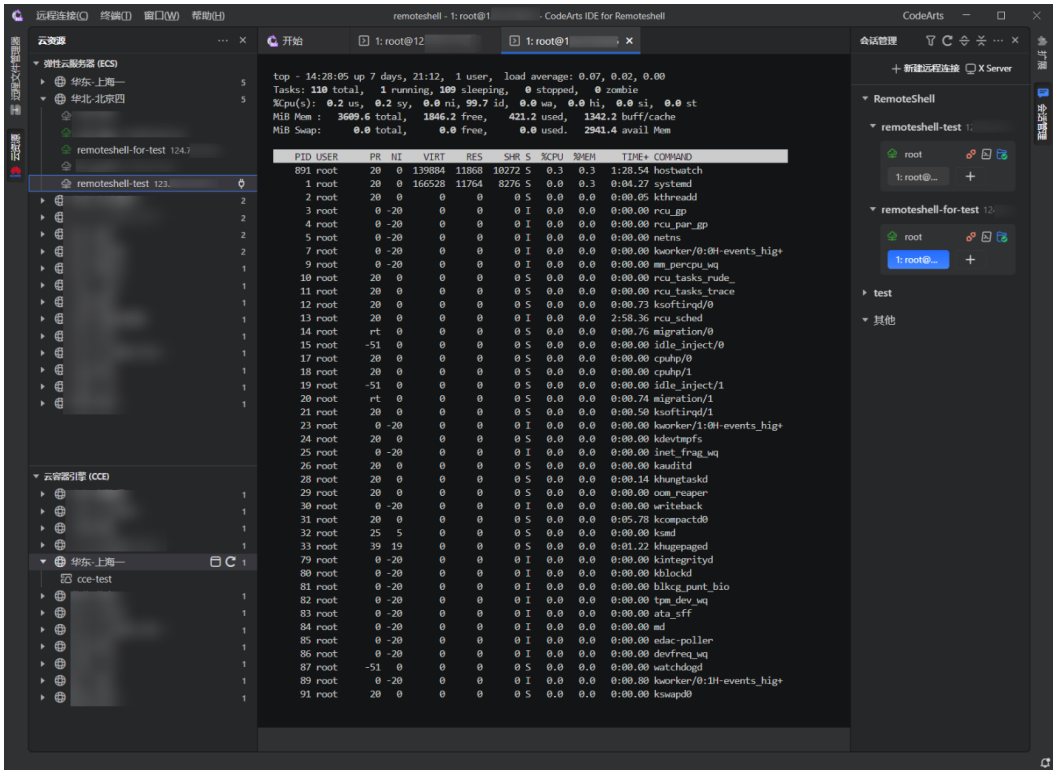
约束与限制

RemoteShell目前仅可用于Windows。

用户界面

参考**登录CodeArts IDE客户端**，登录CodeArts IDE for RemoteShell客户端，如下图所示：

图 11-1 CodeArts IDE for RemoteShell 客户端



RemoteShell的用户界面主要由以下模块组成：

- **“云资源”**区域，列出与华为云账户关联的所有云资源：弹性云服务器（ECS）、云容器引擎（CCE）。
- **“远程文件管理器”**区域，提供对已连接主机的文件系统的访问。
- **“编辑器”**区域，保存当前打开的远程终端会话和文件的选项卡。
- **“会话管理”**区域，用于管理主机和连接。

配置代理

如果您需要通过代理连接远程主机（例如，在局域网内访问公共网络的IP地址），您可以参考本章节配置网络代理后使用CodeArts IDE for RemoteShell。

RemoteShell支持通过代理连接主机。可以配置多个代理，并在每台主机上单独使用它们，或使用应用程序代理。代理管理帮助用户更加便捷地管理代理配置。

步骤1 打开CodeArts IDE for RemoteShell客户端。

步骤2 使用以下任意方式打开代理管理。

- 单击侧边栏的“会话管理”，打开“会话管理”区域，单击右上角，选择“打开代理管理”
- 单击左上角主菜单的“远程连接”，选择“代理管理”。

步骤3 在“代理管理”弹窗中，在左侧列表选择已有代理进行修改，或单击新增代理，参考表11-1填写代理参数信息。

图 11-2 代理管理

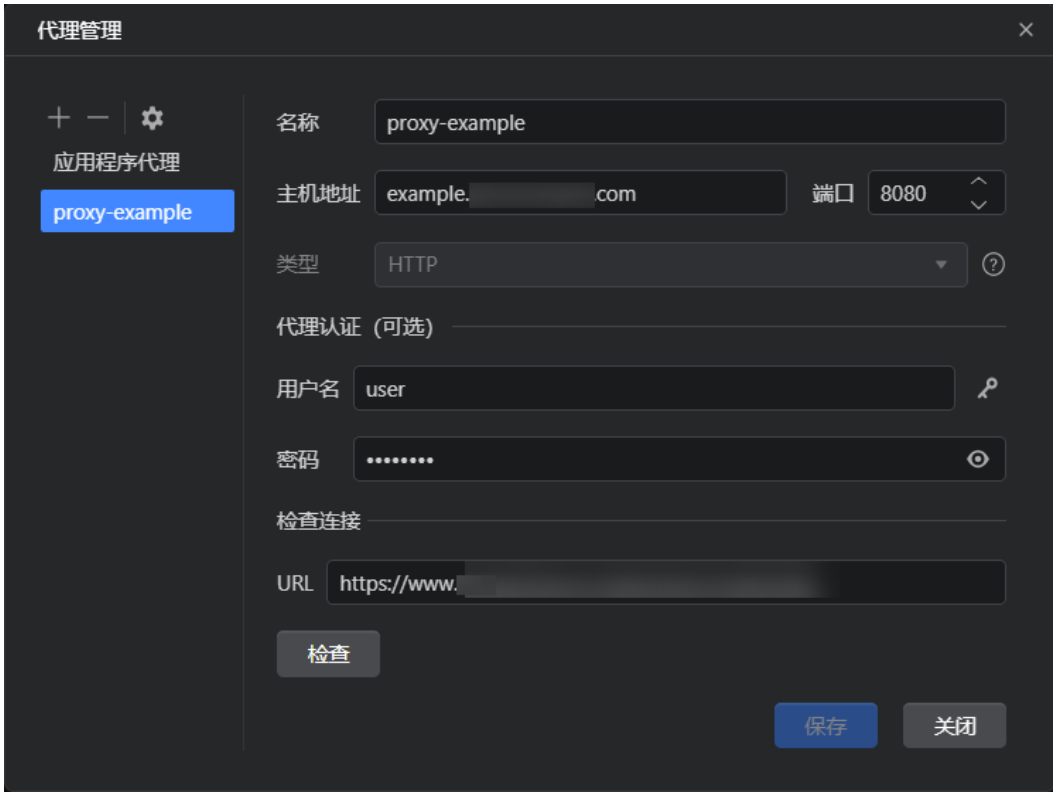


表 11-1 代理参数

参数项	描述	样例值
名称	代理的名称，用户自定义。	proxy-sample
主机地址	代理服务器的IP地址。	example.proxy.com
端口	代理服务器的端口。	3128
用户名	可选参数，登录代理服务器的用户名。	user
密码	可选参数，登录代理服务器的用户名密码。	******
URL	可选参数，验证代理信息是否可用。 输入任意有效的URL地址，然后单击**“检查”**，提示代理“连接成功”时，表示当前代理信息可用。	https://www.huaweicloud.com/product/ide.html

步骤4 单击**“保存”**，完成代理配置。

----结束

11.2 配置 RemoteShell 连接主机

11.2.1 管理主机

通过RemoteShell，可以连接任意主机，确保网络畅通即可。

对于每个已配置的主机，可以创建和维护多个用户连接。

连接主机

步骤1 执行以下操作之一连接到主机：

- 要连接华为云服务器，请在**“云资源”**>**“弹性云服务器(ECS)”**区域中选择要连接的服务器，单击打开**“新建远程连接”**窗口。
- 要连接到任意主机，请在**“会话管理”**区域中，单击**“新建远程连接”**。

步骤2 在打开的**“新建远程连接”**窗口中，需提供主机连接参数。

在**“通用”**标签页，指定连接参数：主机地址、端口、用户名、认证方式和用户凭证。

图 11-3 新建远程连接

如果必须通过代理连接，请在“代理”列表中选择一个已配置的代理，或单击 并按照**配置代理**中所述配置。

表 11-2 主机连接参数

参数项	描述	样例值
主机名称	（可选）用户自定义的主机名称。	主机1
主机地址	要连接的主机的IP地址。	100.100.100.100
主机分组	用户自定义的主机分组，默认为<其他>。	<其他>
代理服务器	连接该主机是否要使用代理服务器，可根据实际情况选择。	<不使用代理>
端口	通过SSH连接的端口。	22

参数项	描述	样例值
连接名称	（可选）用户自定义的当前连接的名称。	连接1
颜色	默认无颜色，可选择其他颜色主题。	默认
用户名	登录主机的用户。	root
认证方式	通过密码、密钥或双因子认证。	密码
密码	选择“密码”、“双因子认证”方式时的用户凭据。	******
私钥	选择“密钥”认证方式时的私钥文件的路径。	C:\Users\用户名\.ssh\id_rsa
密码短语	选择“密钥”认证方式时，如果当前密钥为加密密钥时，则需要填写密码短语。	******

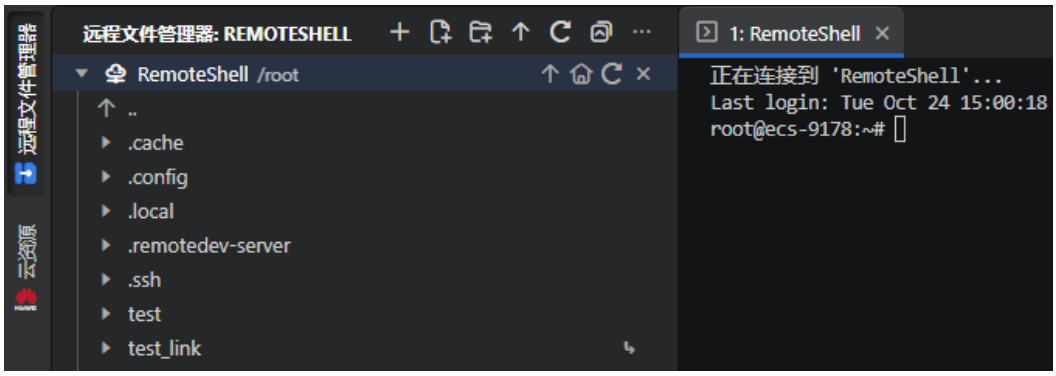
步骤3 在**“高级”**标签页，可以选择RemoteShell是否周期性地发送保持活动状态的消息来让连接保持活动状态。如果需要连接的主机提供X11转发以通过SSH启动图形应用程序，可以通过选中**“X11转发”**在客户端侧启用它，单击**“配置X11转发”**可以按照**配置X Server转发功能**中所述配置指定X11转发的目的地址。

图 11-4 新建远程连接“高级”标签页

步骤4 单击**“连接”**。主机连接记录将添加到**“会话管理”**区域。

RemoteShell自动建立与主机的新连接，在**“编辑器”**区域的单独选项卡中打开命令行会话，并在**“远程文件管理器”**区域中显示主机的文件系统。

图 11-5 连接成功界面



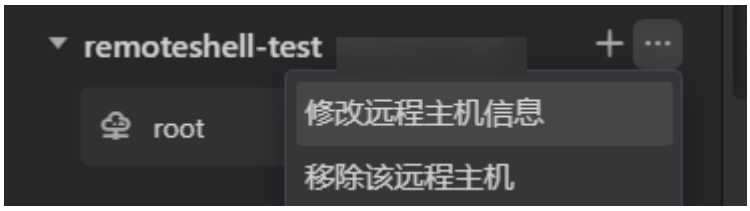
----结束

编辑主机

步骤1 在“**会话管理**”区域中，单击要编辑的主机右侧的…或者单击鼠标右键。

步骤2 在弹出菜单中，选择“**修改远程主机信息**”。

图 11-6 修改远程主机信息



步骤3 在打开的“**修改远程主机信息**”窗口中，根据需要修改主机参数。

步骤4 单击“**保存**”以应用更改。

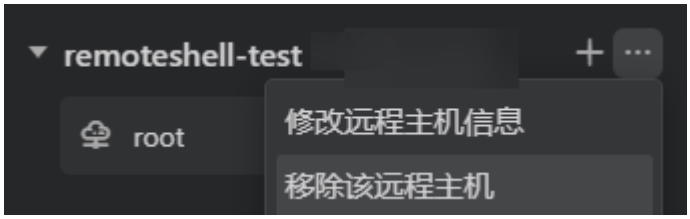
----结束

删除主机

步骤1 在“**会话管理**”区域中，单击要删除的主机右边的…或者单击鼠标右键。

步骤2 在弹出菜单中，选择“**移除该远程主机**”。

图 11-7 移除该远程主机



----结束

11.2.2 管理连接

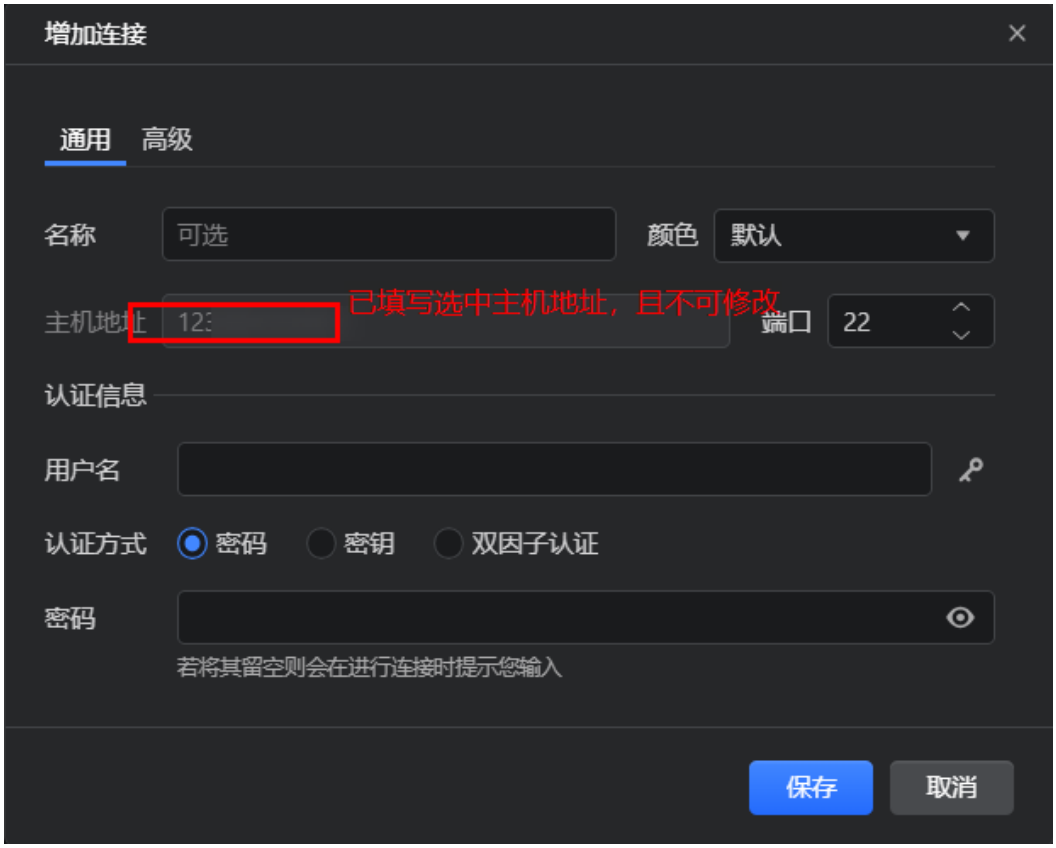
通过RemoteShell（基于SSH协议）管理连接到单个主机的多个终端会话。如何添加需要连接的新主机请参考**管理主机**。

添加连接

步骤1 在“**会话管理**”区域中，单击要连接的主机右边的+。

步骤2 在打开的“**增加连接**”窗口中，会默认填写在**步骤1**中选中的主机地址（不可修改），需要指定其他连接参数。参考**表11-2**完成参数填写。

图 11-8 增加连接



- 在“**通用**”标签页，指定连接名称、端口、用户名、认证方式和用户凭据。
- 在“**高级**”标签页，通过勾选“**X11转发**”复选框选择是否启用X11转发。

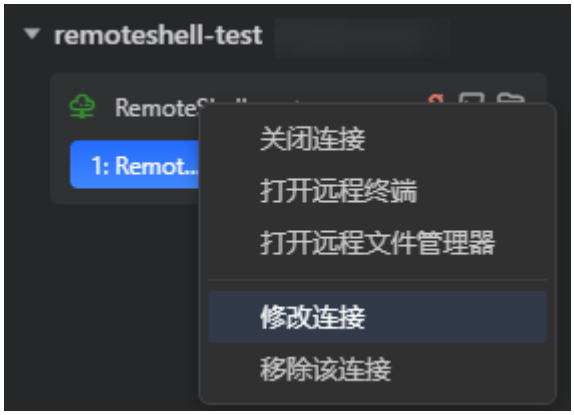
步骤3 单击“**保存**”。连接成功后，新的连接记录将添加到所选主机下面。

----结束

修改连接

步骤1 在“**会话管理**”区域中，右键单击要编辑的用户连接记录，在弹出菜单中选择“**修改连接**”。

图 11-9 修改连接



步骤2 在打开的“**编辑连接**”窗口中，根据需要修改连接参数。

- 在“**通用**”标签页，可以修改连接名称、端口、用户名、认证方式和用户凭据。
- 在“**高级**”标签页，通过勾选“**X11转发**”复选框选择是否启用X11转发。

步骤3 单击“**保存**”即可完成更改。

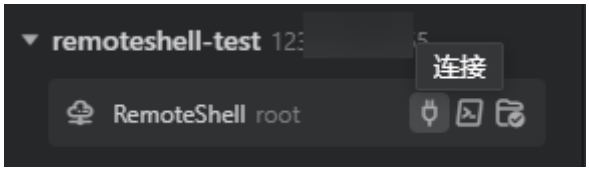
----结束

建立连接

执行以下操作之一建立连接：

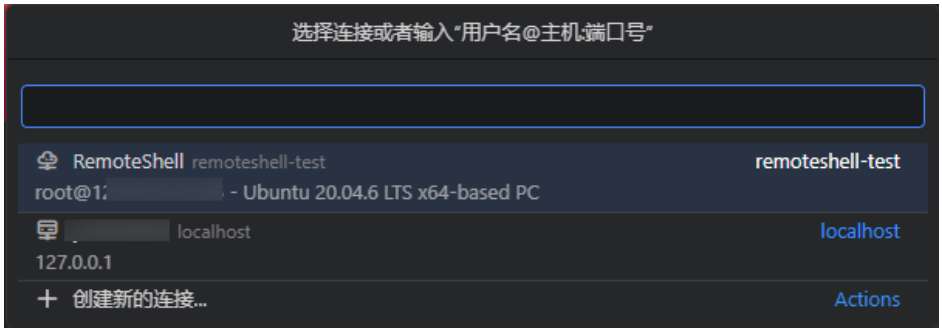
- 在“**会话管理**”区域，单击所需用户连接右边的“**连接**”按钮（ ）建立连接。

图 11-10 连接



- 在主菜单栏选择“远程连接 >打开远程连接”，或者按下**Ctrl+Alt+O**，选择连接的弹窗将被打开。

图 11-11 选择连接



执行以下操作之一：

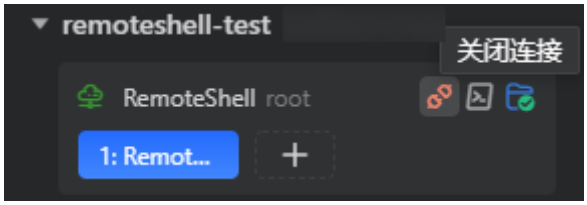
- 选择一个现有主机连接，与其建立连接。
- 选择“**创建新的连接...**”，从创建主机开始。
- 输入*user@host:port*格式的连接字符串，以继续创建主机并预填充其主要连接参数。

关闭连接

执行以下操作之一关闭连接：

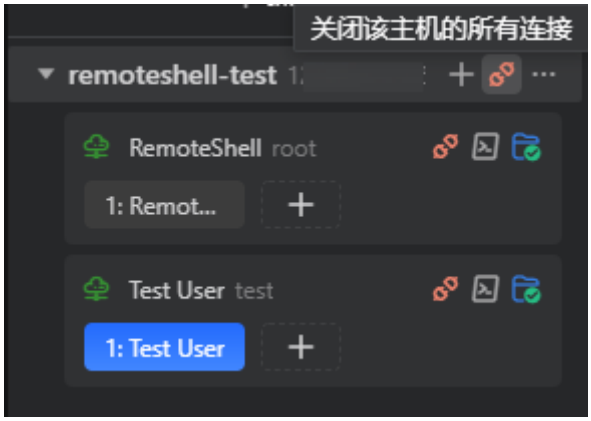
- 要终止与主机的单个连接，请在“**会话管理**”区域中，单击要终止的连接旁边的“**关闭连接**”按钮（）。

  图 11-12 关闭连接

- 要终止与主机的所有连接，请在“**会话管理**”区域中，单击要断开的主机旁边的“**关闭该主机的所有连接**”按钮（）。

  图 11-13 关闭该主机的所有连接

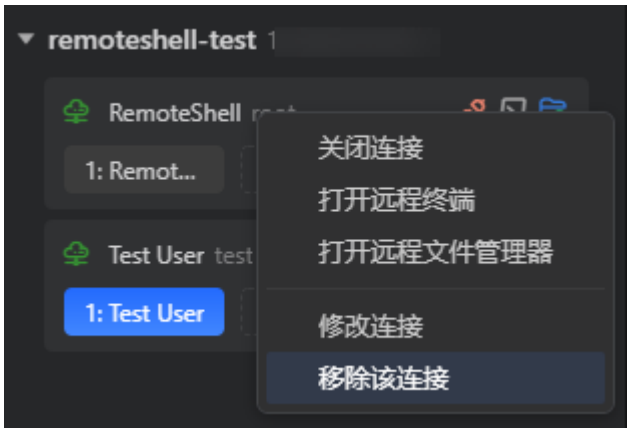
  

移除连接

步骤1 在“**会话管理**”区域中，右键单击要删除的连接记录行。

步骤2 在弹出菜单中，选择“**移除该连接**”。

图 11-14 移除该连接



----结束

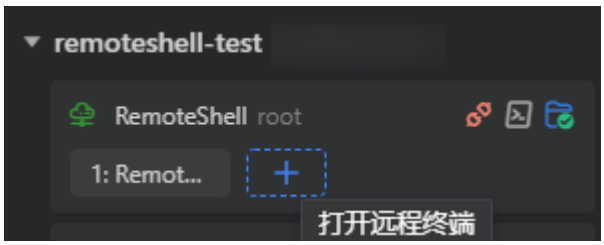
11.2.3 管理终端会话

当与主机建立连接时，RemoteShell会自动启动一个终端会话。如有必要，可以为每个已建立的连接打开多个单独的终端会话。

启动终端会话

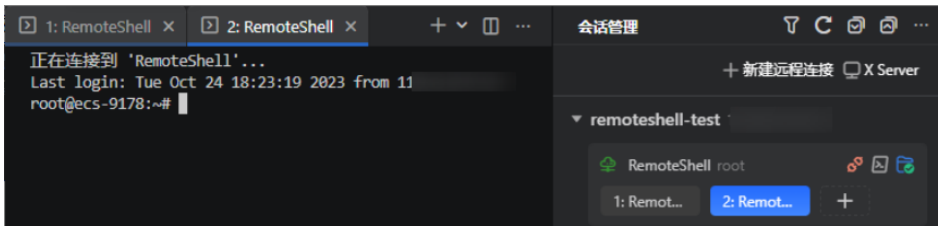
步骤1 在**“会话管理”**区域中，单击**“打开远程终端”**按钮（＋），或者头部导航中的**“打开远程终端”**按钮（）。或者使用快捷键**Ctrl+Alt+T**。

图 11-15 打开远程终端



步骤2 在**“编辑器”**区域的新选项卡中打开新会话，会话记录将添加到**“会话管理”**区域的相应主机下。

图 11-16 打开远程终端成功界面



----结束

说明

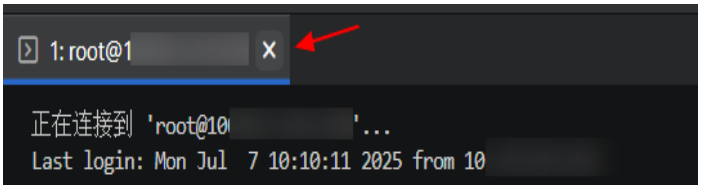
要复制当前终端会话并在单独的选项卡中打开它，请在主菜单中选择“终端>复制远程终端”或按“Ctrl+Alt+D”。

关闭终端会话

执行以下任一操作可以关闭终端会话：

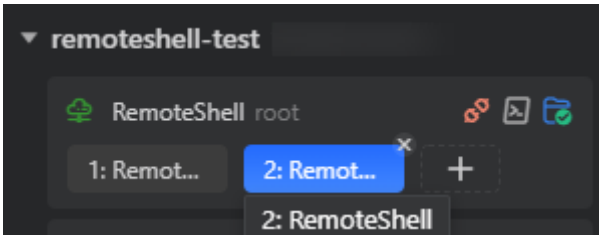
- 在“编辑器”区域中，单击要关闭的选项卡右侧的“关闭”按钮（×），或使用快捷键Ctrl+F4。

  图 11-17 选项卡关闭终端

- 在“会话管理”区域中，单击要关闭的会话记录旁边的“关闭”按钮（×）。

  图 11-18 会话管理页面关闭终端

布局编辑器区域

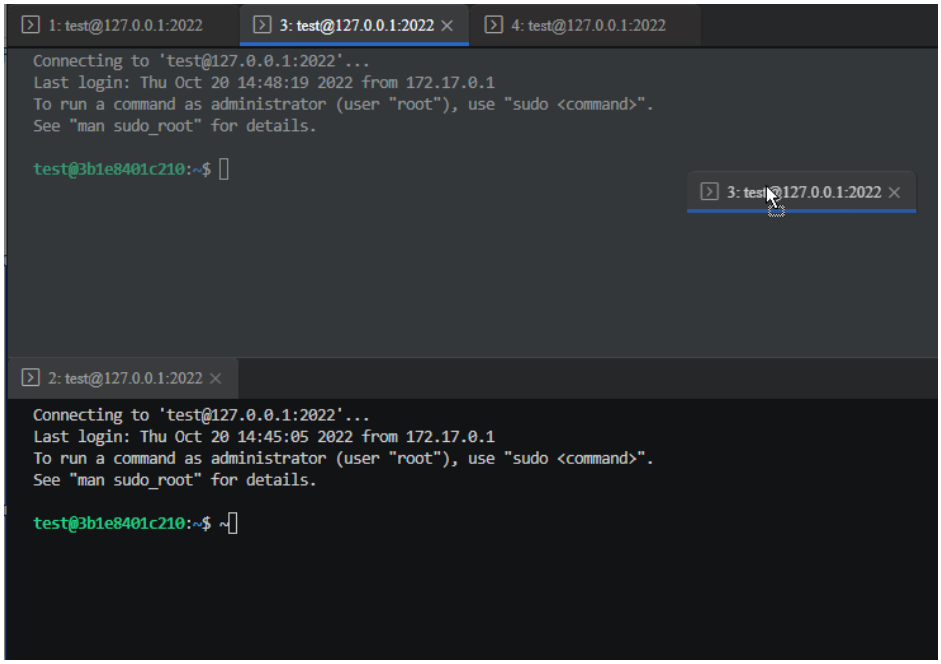
为了便于使用打开的选项卡，可以重新组织“编辑器”区域布局。

步骤1 在“编辑器”区域中，开始拖动要重新定位的选项卡。

步骤2 选择要放置终端会话选项卡的“编辑器”区域（顶部、底部、左侧或右侧）。

步骤3 占位符出现后，立即释放鼠标，使其在选定的“编辑器”区域的部分中打开。

图 11-19 布局编辑器区域



----结束

11.2.4 管理凭据

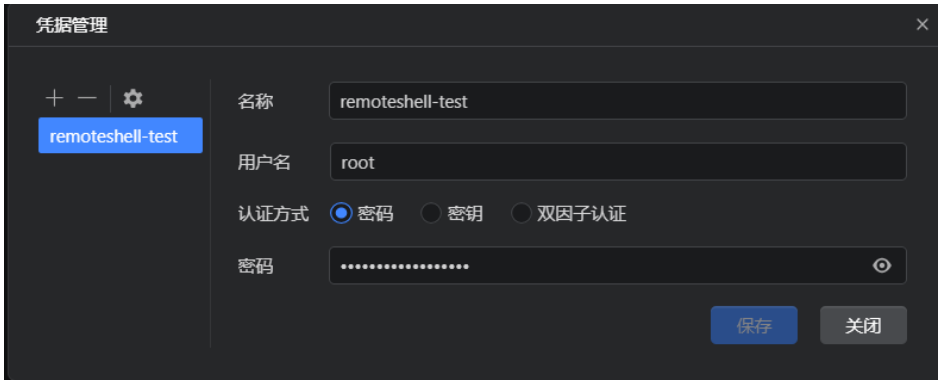
RemoteShell提供“凭据管理”，可以轻松管理所有提供的凭据。

步骤1 在**“会话管理”**区域中单击**“视图和更多操作...”**按钮（ ），然后从弹出菜单中选择**“打开凭据管理”**。

图 11-20 打开凭据管理

步骤2 在“凭据管理”窗口会展示所有存在的凭据记录。

图 11-21 凭据管理



步骤3 通过以下方式管理凭据：

- 要添加新的凭据记录，请在“凭据管理”的工具栏上单击**“新增凭据”**按钮（ ）。
- 要删除凭据记录，请在“凭据管理”的工具栏上单击**“移除凭据”**按钮（ ）。
- 要配置凭据管理，请在“凭据管理”的工具栏上单击设置按钮（ ）。在打开的弹出窗口中，选择控制是否在“凭据管理”中自动保存成功使用后的新的连接和代理服务器的凭据。

----结束

11.2.5 管理文件系统

管理远程文件系统

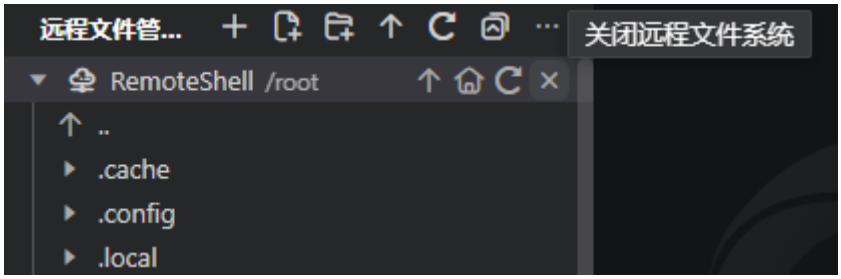
当与主机建立连接时，RemoteShell会自动在**“远程文件管理器”**区域中打开其文件系统。

- 要手动打开远程文件系统，请在**“会话管理”**区域中，单击**“打开远程文件管理器”**按钮（ ）。

图 11-22 打开远程文件管理器

- 要关闭远程文件系统，请在**“远程文件管理器”**区域中，单击要关闭的文件系统旁边的**“关闭远程文件系统”**按钮（ ）。

图 11-23 关闭远程文件系统

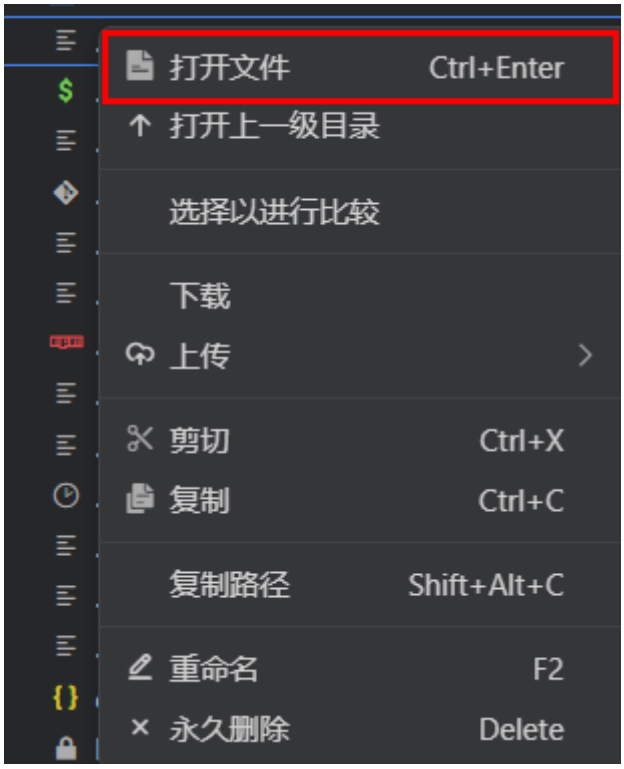


编辑文件

RemoteShell允许编辑远程主机的文件。

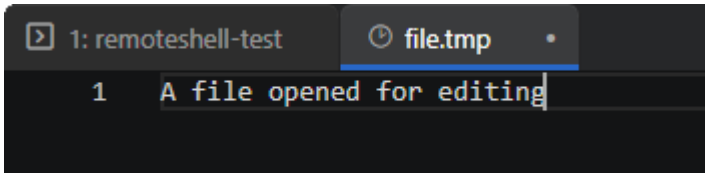
步骤1 右键单击文件并从上下文菜单中选择**“打开文件”**，或选择它并按**Ctrl+Enter**键。

图 11-24 打开文件



步骤2 该文件将在“**编辑器**”区域的单独选项卡中打开。要保存文件，在主菜单中选择“文件>保存”或按**Ctrl+S**。如果编辑了多个文件，可以通过选择“**远程连接>全部保存**”或按**Ctrl+K S**来保存它们。

图 11-25 打开文件后的编辑器区域选项卡



----结束

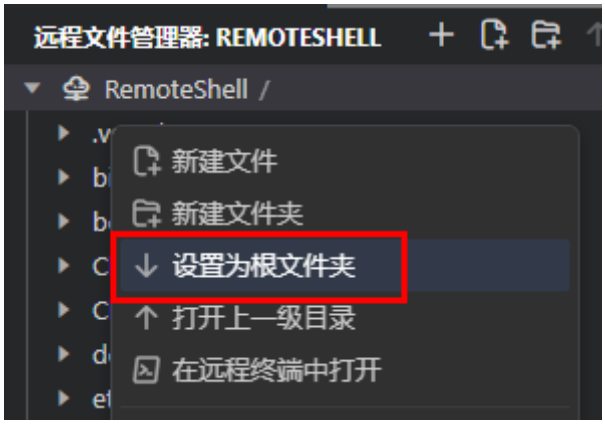
设置根文件夹

为了方便浏览远程主机文件系统，可以将远程主机上的任何文件夹设置为根目录。稍后重新连接到主机时，选定的文件夹将作为起始位置打开。

执行以下操作之一：

- 在“**远程文件管理器**”区域中，右键单击要设置为根的文件夹。
- 在上下文菜单中，选择“**设置为根文件夹**”。

图 11-26 设置为根文件夹



上传文件和文件夹

RemoteShell允许将文件和文件夹从本地计算机上传到远程主机。

步骤1 在“**远程文件管理器**”区域中，右键单击要上传的文件或文件夹的目标文件夹。

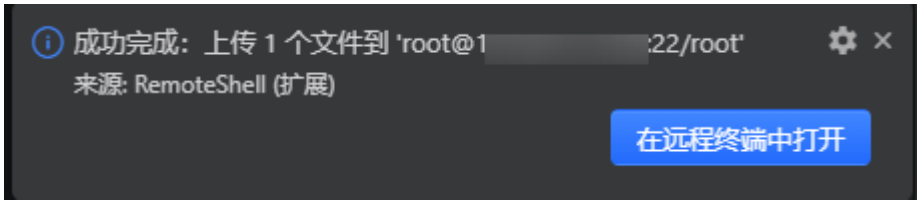
步骤2 在上下文菜单中，单击“**上传文件**”或“**上传文件夹**”。

图 11-27 上传文件

步骤3 在打开的文件选择器对话框中，选择所需的文件或文件夹。

步骤4 上传完成后，显示相应的通知信息。

图 11-28 上传成功提示信息



----结束

下载文件和文件夹

RemoteShell允许将文件和文件夹从远程主机下载到本地计算机。

步骤1 在**“远程文件管理器”**区域中，右键单击要下载的文件或文件夹。

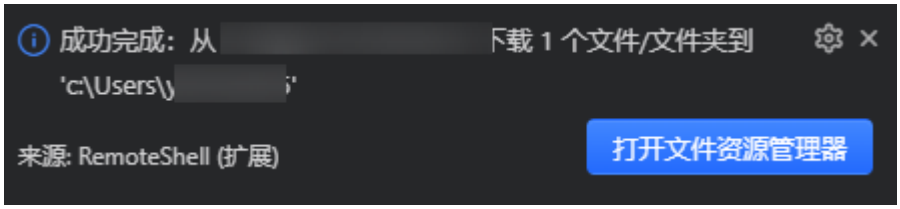
步骤2 在上下文菜单中，选择**“下载”**。

图 11-29 下载

步骤3 在打开的文件选择器对话框中，选择下载位置。

步骤4 下载完成后，显示相应的通知信息。

图 11-30 下载成功提示信息



----结束

创建文件和文件夹

步骤1 在“**远程文件管理器**”区域中，选择要新建的文件或文件夹。

步骤2 执行以下任一操作：

- 在“**远程文件管理器**”区域工具栏上，单击**新建文件**（）/**新建文件夹**（）按钮。
- 右键单击选定的文件夹，然后从上下文菜单中选择**新建文件**/**新建文件夹**。

步骤3 在出现的输入框中，输入要新建的文件/文件夹的名称，按**Enter**确认。

----结束

复制、移动文件和文件夹

可以在同一主机上以及不同主机上的位置之间复制和移动文件和文件夹。

步骤1 在“**远程文件管理器**”区域中，右键单击要复制或要移动到其他位置的文件/文件夹。

- 要复制文件/文件夹，请在上下文菜单中选择“**复制**”，或按**Ctrl+C**。
- 要移动文件/文件夹，请在上下文菜单中选择“**剪切**”，或按**Ctrl+X**。

步骤2 右键单击要将文件/文件夹移动到的文件夹，然后在上下文菜单中选择“**粘贴**”，或按**Ctrl+V**。

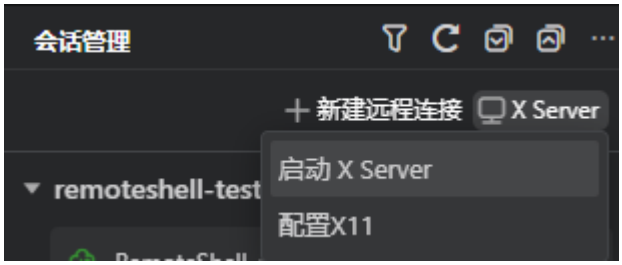
----结束

11.2.6 配置 X Server 转发功能

RemoteShell提供X11转发功能，可以通过SSH启动图形应用程序。 X11连接可以转发到RemoteShell内置的X Server或者第三方X Server。

在“**会话管理**”区域中，单击“**X Server**”按钮（）并从弹出列表中选择所需的选项。

图 11-31 X Server



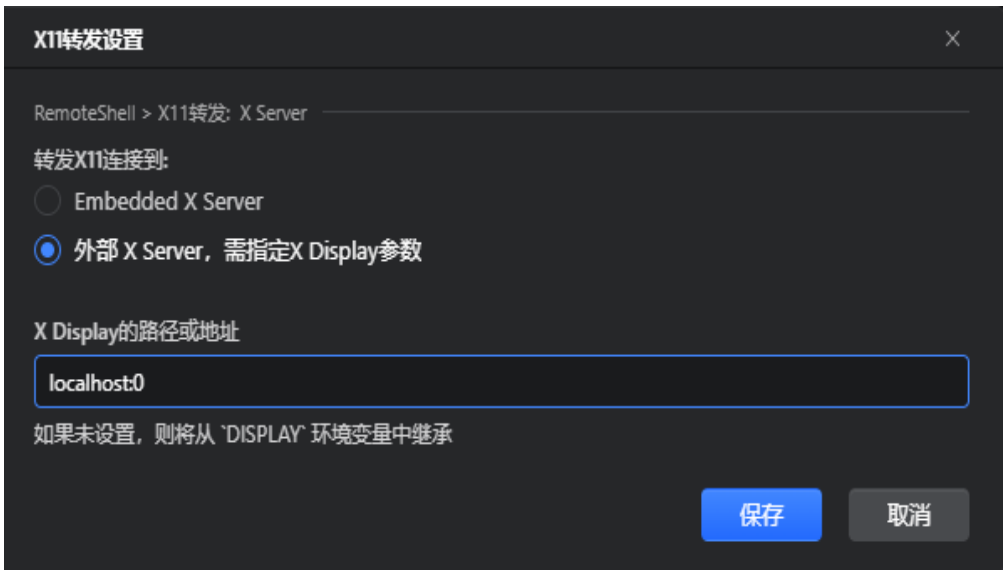
- 使用“**启动X Server**”/“**停止X Server**”命令来启动/停止内置X Server。

  说明

  如果想要使用第三方X Server来使用X11转发，其与内置X Server会有冲突，可能需要停止内置X Server。

- 使用“**配置 X11**”命令可以配置X11转发。如果选择将X11连接转发到外部 X Server，请在“**X Display的路径或地址**”字段中提供服务器路径及其display。

图 11-32 X11 转发设置



11.3 配置 RemoteShell 访问 Kubernetes 集群

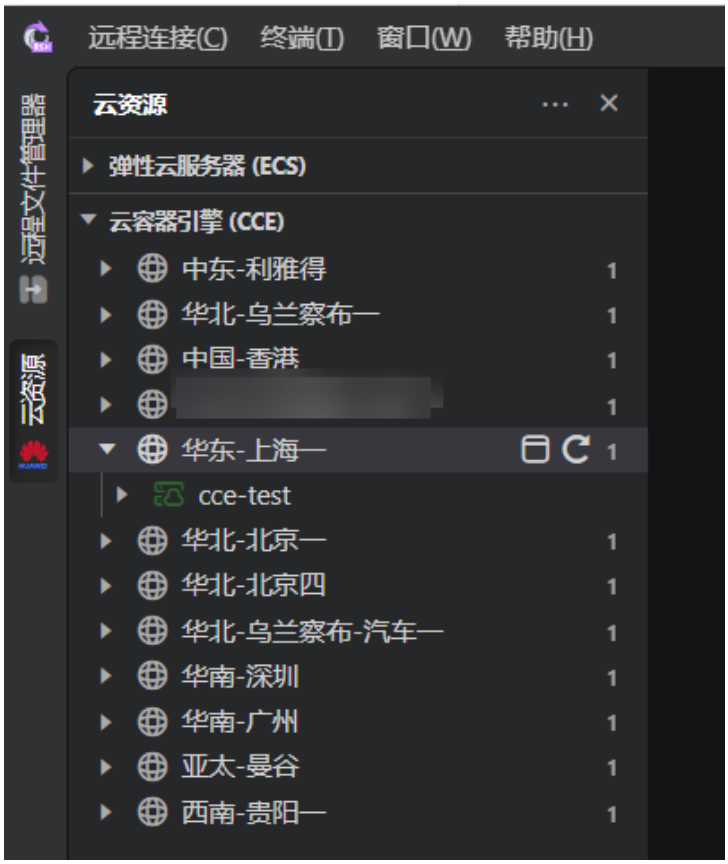
约束与限制

- 仅支持v1.27，v1.28，v1.29，v1.30，v1.31版本的集群。
- 在连接前检查与集群之间的网络连通。

打开 Kubectl 命令行

步骤1 在“**云资源>云容器引擎(CCE)**”区域中选择要访问的集群。

图 11-33 云容器引擎（CCE）



步骤2 单击**“打开Kubectl命令行”**按钮（ ），打开Kubectl命令行。

----结束

通过 Kubectl 连接到集群 Pod

步骤1 在**“云资源>云容器引擎(CCE)”**区域中选择要访问的集群。

步骤2 单击集群，依次选择对应的命名空间、工作负载类型，选择要连接的集群Pod。

步骤3 单击**“连接到Pod”**按钮（ ）。

----结束

12 使用集成终端运行命令

12.1 集成终端简介

CodeArts IDE包括一个功能齐全的集成终端，其工作目录为当前工作区根目录。集成终端可以使用安装在计算机上的各种终端，默认为PowerShell。用户可以从**“启动配置文件...”**列表（ ▾ ）中选择其他可用的终端类型（如Command Prompt或Git Bash）。

要打开终端，请执行以下任一操作：

- 按“Shift+Esc”或“Alt+F12”。
- 在主菜单中，选择**“查看 > 终端”**。
- 从“命令”面板（“Ctrl+Shift+P”）中，运行**“查看: 切换终端”**命令。

集成终端shell以CodeArts IDE的权限运行时，如果需要以管理员或不同的权限运行shell命令，请在终端中使用平台应用程序，如runas.exe。

说明

- 如果想在CodeArts IDE的外部打开终端，可以通过按“Ctrl+Shift+C”。
- 在终端中右键单击时，如果有选中的文本，将会复制该文本并取消选中；如果没有选中的文本，则会执行粘贴操作。此行为可通过设置中的terminal.integrated.rightClickBehavior 选项进行配置。

设置终端外观

可以通过“Ctrl+,”打开“设置”编辑器，在搜索框输入**terminal.integrated.xxx**来设置自定义终端的外观，如：terminal.integrated.Font。

- Font：字体系列、字号和字体粗细。
- Line Height：行高。
- Letter Spacing：字母间距。
- Cursor：样式、宽度和闪烁。

重连终端进程

本地和远程终端进程在窗口重新加载时恢复（例如，当扩展安装需要重新加载时）。终端将重新连接，终端的UI状态将恢复，包括活动选项卡和拆分终端相对尺寸。

实验设置**terminal.integrated.persistentSessionReviveProcess**允许您定义在终端进程关闭后（例如，在窗口或应用程序关闭时）应恢复以前的终端会话内容并重新创建进程的时间。恢复进程的当前工作目录取决于shell是否支持它。

打开工作目录

默认情况下，终端在资源管理器中当前打开的文件夹中打开。使用**terminal.integrated.cwd**设置，您可以指定要打开的自定义路径。

须知

在Windows上，反斜杠符号\必须转义为\\。

```
{
  "terminal.integrated.cwd": "D:\\CodeArtsProjects"
}
```

拆分终端从父终端启动的目录中启动。可以使用**terminal.integrated.splitCwd**设置更改此行为，以便拆分终端在当前工作区根中启动。

```
{
  "terminal.integrated.splitCwd": "workspaceRoot"
}
```

12.2 配置终端快捷键

当“终端”视图聚焦时，许多在CodeArts IDE中绑定的快捷键将失效，因为此时触发的快捷键行为将传递到终端本身并由终端处理。
“terminal.integrated.commandsToSkipShell”设置定义一个命令列表，用于指定哪些命令在集成终端中执行时会绕过“shell”直接运行。要替代默认值并将相关命令的快捷绑定传给“shell”，可以加“-”字符为前缀的命令。例如，添加“-workbench.action.quickOpen”可以“Ctrl+P”到达“shell”。

```
{
  "terminal.integrated.commandsToSkipShell": [
    // Ensure the toggle sidebar visibility keybinding skips the shell
    "workbench.action.toggleSidebarVisibility",
    // Send quick open's keybinding to the shell
    "-workbench.action.quickOpen",
  ]
}
```

要覆盖“**terminal.integrated.commandsToSkipShell**”设置并将键绑定发送到终端，请启用“**terminal.integrated.sendKeybindingsToShell**”设置。

终端中的组合按键

默认情况下，当按下组合按键时，组合按键是最高优先级的，它将始终跳过终端（绕过“**terminal.integrated.commandsToSkipShell**”设置）转到CodeArts IDE。如果

用户希望终端使用“Ctrl+K”（对于bash终端，这将剪切光标之后的行），可以将“**terminal.integrated.allowChords**”设置禁用：

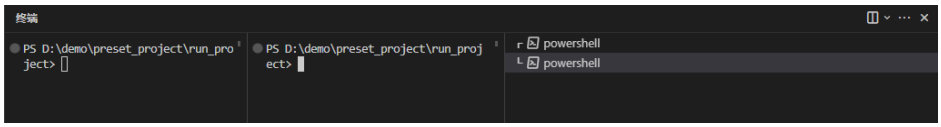
```
{
  "terminal.integrated.allowChords": false
}
```

要覆盖**terminal.integrated.commandsToSkipShell**并将键绑定发送到shell而不是工作台，请设置**terminal.integrated.sendKeybindingsToShell**。

12.3 管理终端实例

终端实例以选项卡的形式显示，这些选项卡展示在“终端”视图的右侧或左侧。每个实例都有一个条目，包含其名称、图标、颜色和组合装饰（对于分组实例）。如下图所示：

图 12-1 终端示例



说明

用户可以通过终端设置项terminal.integrated.tabs.location更改选项卡位置。

添加终端实例

执行以下任一操作：

- 单击“终端”视图右上角的**“新建终端”**按钮（＋），或按“Ctrl+Shift+`”。
- 单击**“启动配置文件...”**按钮（▾），然后从列表中选择所需的终端类型。

删除终端实例

在选项卡列表中执行如下操作：

悬停一个选项卡，然后右键单击选择**“终止终端”**菜单，或选择一个选项卡并按**“终止 (Delete)”**（🗑）键。如下图所示：

图 12-2 终止终端

终端实例分组

“终端”提供了多种功能，让用户可以自定义其布局。

要将当前终端实例拆分为两个，从而创建组，请执行以下任一操作：

- 在选项卡列表中，悬停选项卡，然后单击“**拆分**”按钮（ ）。
- 在选项卡列表中，右键单击选择“**拆分终端**”。
- 在选项卡列表中，选中选项卡，按“Ctrl+Shift+5”。

要将终端实例添加到组，请将对应选项卡拖入当前显示终端区域。要重新排列组中的选项卡，可以拖拽选项卡调整其在组内的顺序。

要取消拆分终端，请在选项卡列表中右键单击该终端，然后从上下文菜单中选择“**取消拆分终端**”。

要在终端组内导航，请使用以下键盘快捷键。

- 按“Ctrl+PageDown”或“Alt+→”键聚焦下一组。
- 按“Ctrl+PageUp”或“Alt+←”键聚焦上一组。

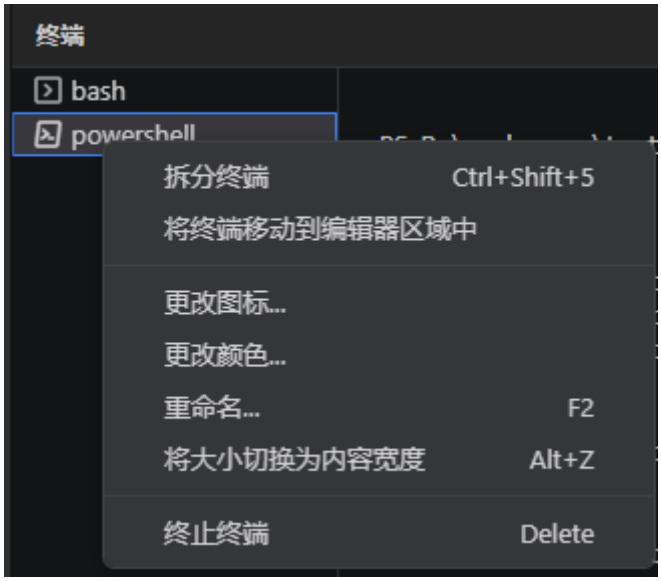
在组中，通过使用“Alt+↑”聚焦上一个窗格，使用“Alt+↓”聚焦下一个窗格，在终端之间导航。

自定义选项卡

更改终端的名称、图标或选项卡颜色，执行以下任一操作。

- 右键单击其选项卡，从上下文菜单中选择相应的操作。

图 12-3 终端右键菜单



- 通过单击左下角的“管理>命令面板”，参考**表12-1**修改终端信息。

图 12-4 修改终端信息

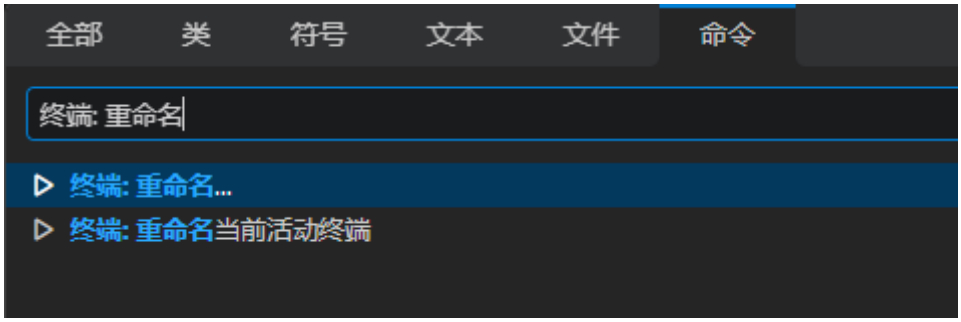


表 12-1 终端相关命令

命令	描述
终端: 重命名	更改终端实例的名称。
终端: 更改图标	更改终端实例的图标。
终端: 更改颜色	更改终端实例的选项卡颜色。

12.4 编辑终端配置文件

要创建新的配置文件，请运行**“终端: 选择默认配置文件”**命令，打开弹窗，选择每一项右侧**“配置终端配置文件”**按钮（ ）以将其作为基础。这将向设置中添加一个新条目，该条目可以在配置文件**settings.json**文件中手动调整。

您可以使用路径或源以及一组可选参数来创建配置文件。源仅在Windows上可用，可用于让CodeArts IDE检测PowerShell或Git Bash的安装。或者使用直接指向shell可执行文件的路径。以下是一些配置文件配置示例：

```
{
  "terminal.integrated.profiles.windows": {
    "PowerShell -NoProfile": {
      "source": "PowerShell",
      "args": ["-NoProfile"]
    }
  },
  "terminal.integrated.profiles.linux": {
    "zsh (login)": {
      "path": "zsh",
      "args": ["-l"]
    }
  }
}
```

终端配置文件是特定于平台的终端配置，由可执行路径、参数和其他自定义项组成。

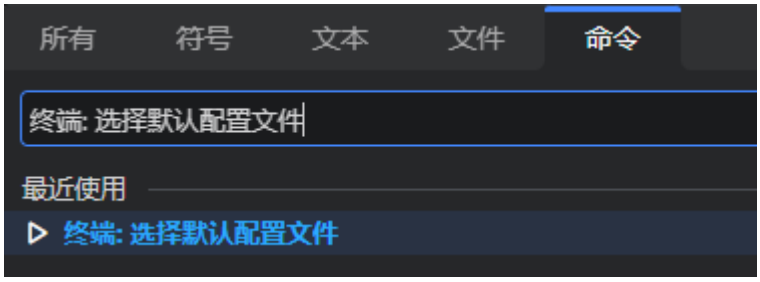
配置文件示例如下：

```
{
  "terminal.integrated.profiles.windows": {
    "My PowerShell": {
      "path": "pwsh.exe",
      "args": [
        "-noexit",
        "-file",
```

```
        "${env:APPDATA}\PowerShell\my-init-script.ps1"
      ]
    }
  },
  "terminal.integrated.defaultProfile.windows": "My PowerShell"
}
```

用户可以在终端配置文件中使用变量（例如上面示例中的APPDATA环境变量），也可以通过运行“**终端: 选择默认配置文件**”命令选择默认的集成终端。如下图所示：

图 12-5 选择默认配置文件命令



要从“**启动配置文件...**”列表（ ）中删除条目，请将配置文件的名称设置为**null**。

例如，要删除**Git Bash**配置文件，请在**settings.json**文件（所在路径为“%AppData%\codearts-java\User\settings.json”或“%AppData%\codearts-cpp\User\settings.json”）使用以下设置。

```
{
  "terminal.integrated.profiles.windows": {
    "Git Bash": null
  }
}
```

配置文件中支持的其他参数包括：

- **overrideName**：一个布尔值，指示是否将根据运行的程序检测到的动态终端标题替换为静态配置文件名称。
- **env**：定义环境变量及其值的映射，将变量设置为**null**以将其从环境中删除。这可以使用**terminal.integrated.env.<platform>**设置为所有配置文件配置。
- **icon**：用于配置文件的图标ID。
- **color**：用于设置图标样式的主题颜色ID。

默认配置文件可以使用**terminal.integated.defaultProfile.***设置手动定义。这应设置为现有配置文件的名称。

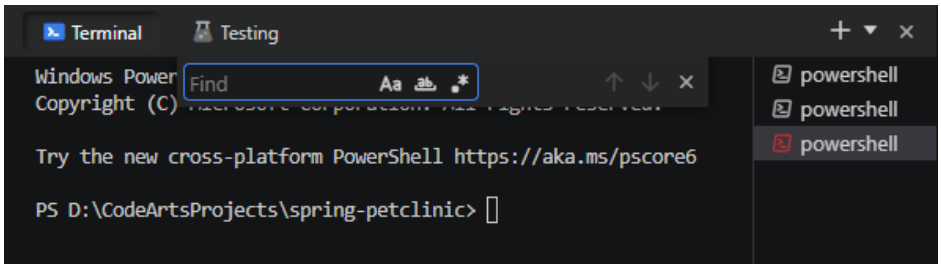
```
{
  "terminal.integrated.profiles.windows": {
    "my-pwsh": {
      "source": "PowerShell",
      "args": ["-NoProfile"]
    }
  },
  "terminal.integrated.defaultProfile.windows": "my-pwsh"
}
```

12.5 查找和运行文本

查找文本

集成终端具有查找功能，可使用**Ctrl+F**触发。如下图所示：

图 12-6 搜索框



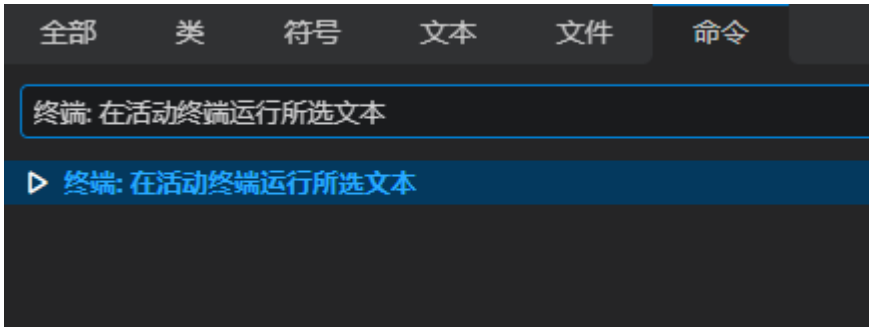
如果您希望“Ctrl+F”转到shell而不是启动**Find**控件，请将以下内容添加到**settings.json**中，这将告诉终端不要跳过与**workbench.action.terminal.focusFind**命令匹配的键绑定。焦点查找命令匹配的键绑定的shell：

```
{
  "terminal.integrated.commandsToSkipShell": [
    "-workbench.action.terminal.focusFind"
  ],
}
```

运行文本

要通过终端执行某些文本，如脚本的一部分，请在编辑器中选中它，然后通过**“命令”**面板（按**“Ctrl+Shift+P”**或**“双击Ctrl”**）运行命令**“终端: 在活动终端运行所选文本”**。如下图所示：

图 12-7 在活动终端执行所选文本



终端尝试运行选定的文本。如下图所示：

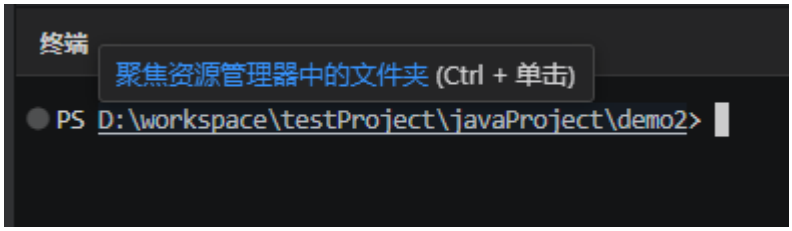
图 12-8 终端运行选的文本

如果在活动编辑器中没有选择文本，则光标下的行将在终端中运行。您也可以通过**workbench.action.terminal.runActiveFile**命令运行活动文件。

检测终端链接

当悬停文件或URL时，会显示下划线，即为终端检测链接。您可以“Ctrl+Click”链接以转到其目标。鼠标悬停工作目录展示终端检测链接，如下图所示：

图 12-9 终端检测链接



根据链路类型，激活它将执行以下操作之一。

- 在编辑器中打开文件。
- 聚焦工作区中的文件夹。
- 打开一个新窗口，其中包含工作区外的文件夹。

13 使用命令行运行文件

13.1 命令行使用简介

CodeArts IDE提供了一个强大的命令行界面，允许您控制如何启动编辑器。您可以使用命令行选项打开文件、安装扩展名、更改显示语言和输出诊断。

要使CodeArts IDE命令行实用程序正常工作，必须将CodeArts IDE二进制位置（以CodeArts IDE for Java产品为例，默认情况下是C:\Program Files\CodeArts IDE for Java\bin）添加到系统路径中。通常，这在安装期间自动执行。否则，您可以手动将位置添加到Path环境变量中。

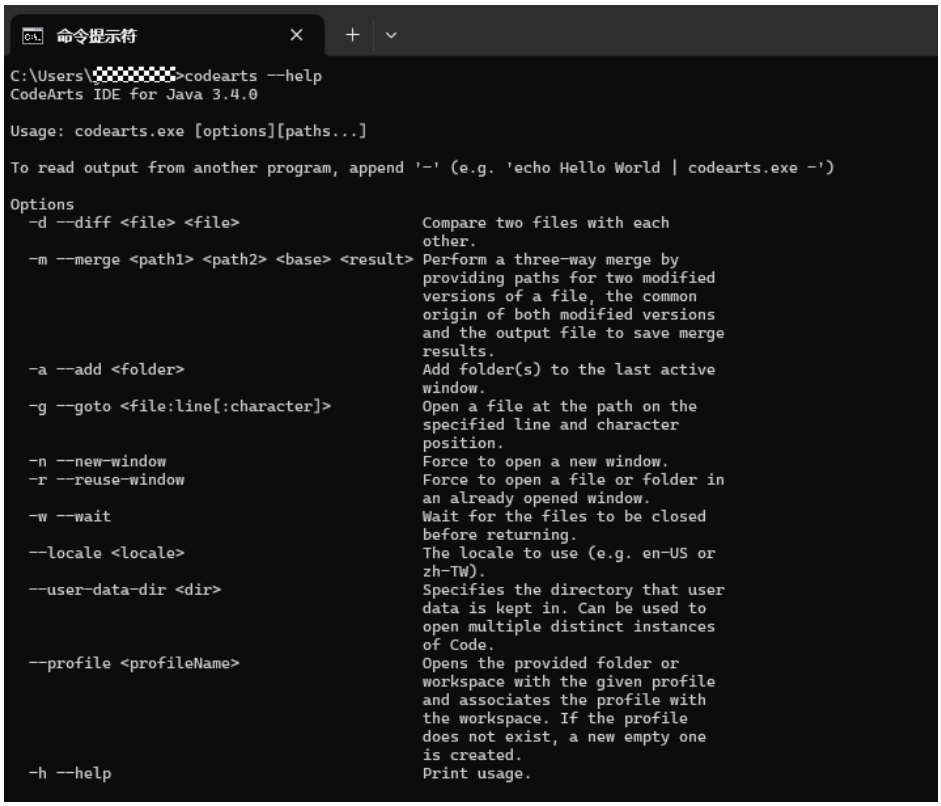
说明

如果您正在寻找如何在CodeArts IDE中运行命令行工具，请参见**使用集成终端运行命令**。

命令行帮助

要了解CodeArts IDE命令行界面的概述，请打开终端或命令提示符，然后键入**codearts --help**。如下图所示：

图 13-1 终端查询 CodeArts IDE 命令

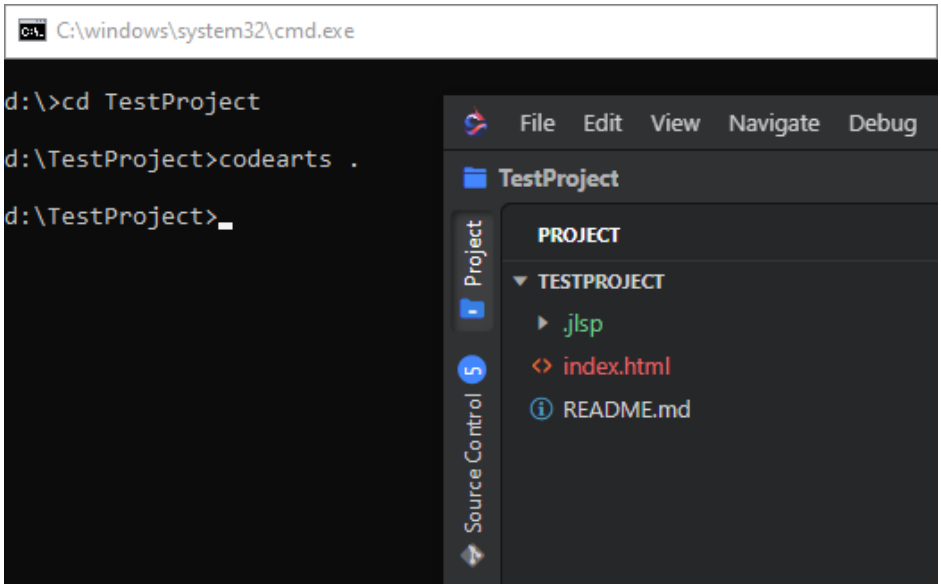


命令行启动

您可以从命令行启动CodeArts IDE以快速打开文件、文件夹或项目。

通常，您可以在文件夹的上下文中打开CodeArts IDE。要执行此操作，请导航到项目文件夹并运行**codearts .**命令。如下图所示：

图 13-2 使用命令打开 IDE



命令行界面参数

以下是通过codearts命令从命令行启动CodeArts IDE时可以使用的可选参数。

表 13-1 启动 CodeArts IDE 可选参数

参数	描述
-h或**--help**	打印命令行参数的内置帮助。
-v或**--version**	打印CodeArts IDE版本（例如1.22.2）、提交ID和体系结构（例如x64）。
-n或**--new-window**	打开CodeArts IDE的新会话，而不是恢复上一个会话（默认值）。
-g或**--goto**	与**file:line{:character}**一起使用时，在特定行和可选字符位置打开文件。之所以提供此参数，是因为某些操作系统允许：在文件名中。
-d或**--diff**	打开差异查看器。需要两个文件路径作为参数。
-w或--wait	等待文件关闭后再返回。
--locale <locale>	设置CodeArts IDE会话的显示语言（区域设置）（例如，**en**或**zh-cn**）。
-s,--status	打印进程使用情况和诊断信息。
--disable-gpu	禁用GPU硬件加速。
--verbose	打印详细输出（意味着**--wait**）。
--prof-startup	在启动过程中运行CPU探查器。
--install-extension <ext>	安装扩展。提供完整的**publisher.extension**作为参数。使用**--force**参数来避免提示。

13.2 使用命令行打开文件

您可以通过CodeArts CLI打开或创建文件。如果指定的文件不存在，CodeArts IDE将创建该文件以及任何新的中间文件夹：

```
codearts index.html style.css documentation\readme.md
```

对于文件和文件夹，您可以使用绝对路径或相对路径。相对路径是相对于运行**codearts**命令的命令提示符的当前目录。

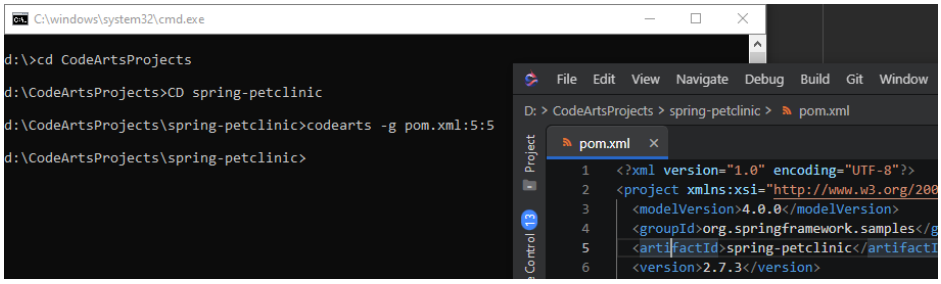
如果在命令行中指定多个文件，CodeArts IDE将仅打开一个实例。

如果在命令行中指定多个文件夹，CodeArts IDE将创建一个包括每个文件夹的多根工作区。如下图所示：

表 13-2 启动 CodeArts IDE 指定目录或文件

参数	描述
file	要打开的文件的名称。如果文件不存在，则将创建并标记为已编辑。您可以通过用空格分隔每个文件名来指定多个文件。
file:line[:character]	与**-g**参数一起使用。要在指定行和可选字符位置打开的文件的名称。您可以以这种方式指定多个文件，但在使用**file:line[: character]**说明符之前，必须使用**-g**参数（一次）。
folder	要打开的文件夹的名称。您可以指定多个文件夹，并创建新的多根工作区。

图 13-3 使用命令打开文件



13.3 使用 URLs 打开项目和文件

您还可以使用操作系统的URL处理机制打开项目和文件。

使用以下URL格式：

- 打开项目
  codearts://file/{full path to project}/
  codearts://file/c:/myProject/
- 打开文件
  codearts://file/{full path to file}
  codearts://file/c:/myProject/package.json
- 在特定行和列上打开文件
  codearts://file/{full path to file}:line:column
  codearts://file/c:/myProject/package.json:5:10

您可以在浏览器或文件资源管理器等应用程序中使用URL，这些应用程序可以解析和重定向URL。例如，您可以将**codearts://URL**直接传递给Windows资源管理器，或作为**codearts://{full path to file}**传递给命令行。

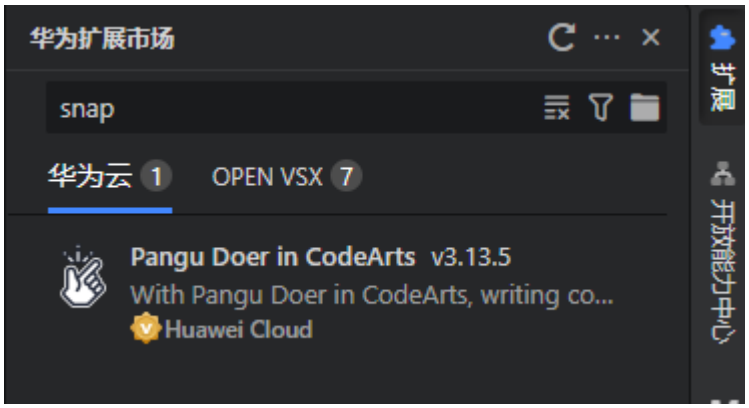
14 使用 CodeArts IDE 第三方扩展

14.1 安装第三方扩展

扩展搜索

CodeArts IDE提供内置的扩展市场，通过切换标签页搜索华为云和Open VSX上的扩展。如下图所示：

图 14-1 在华为云扩展市场查询



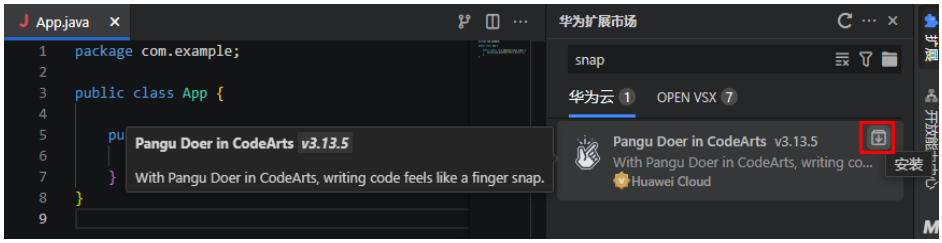
单击右上角的按钮可刷新页面，单击按钮可一键清空搜索框。

扩展安装

- **通过扩展市场安装**

  下载安装华为扩展市场的扩展时，请先登录华为云。单击按钮可通过扩展市场进行安装。如下图所示：

图 14-2 安装扩展



- **本地扩展安装**

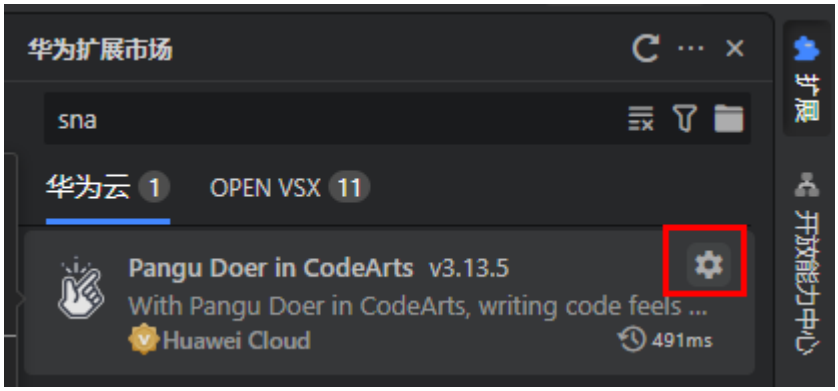
  如果需要安装本地的扩展包可以通过单击搜索框右侧的按钮，选择本地文件进行安装。

扩展管理

扩展启用、禁用及卸载

已安装列表显示了所有当前已安装的扩展，单击按钮可以对扩展进行启用、禁用或卸载等操作。CodeArts IDE的内置扩展无法卸载。如下图所示：

图 14-3 管理按钮



扩展详情

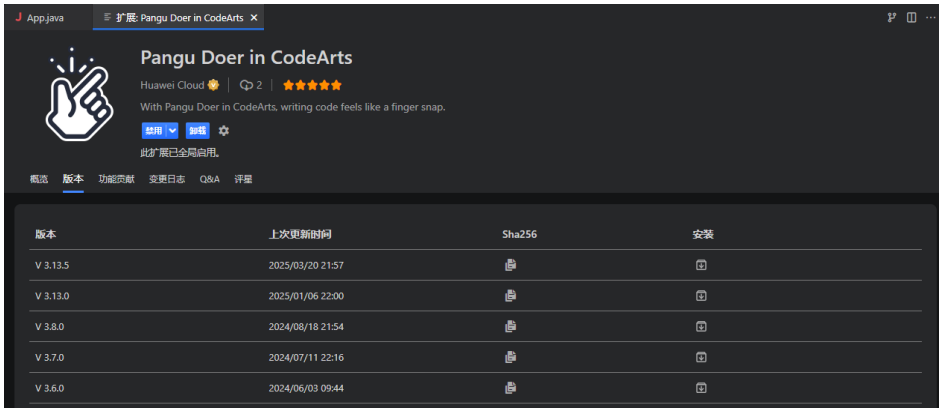
单击扩展可打开扩展详情页，详情页中包含扩展的概览、版本、功能贡献、变更日志、运行时状态，对于已发布至扩展市场上的扩展，则还会包含Q&A和评星。

- 概览：包含扩展的介绍，相关链接，类别和标签信息。如下图所示：

图 14-4 扩展详情概览

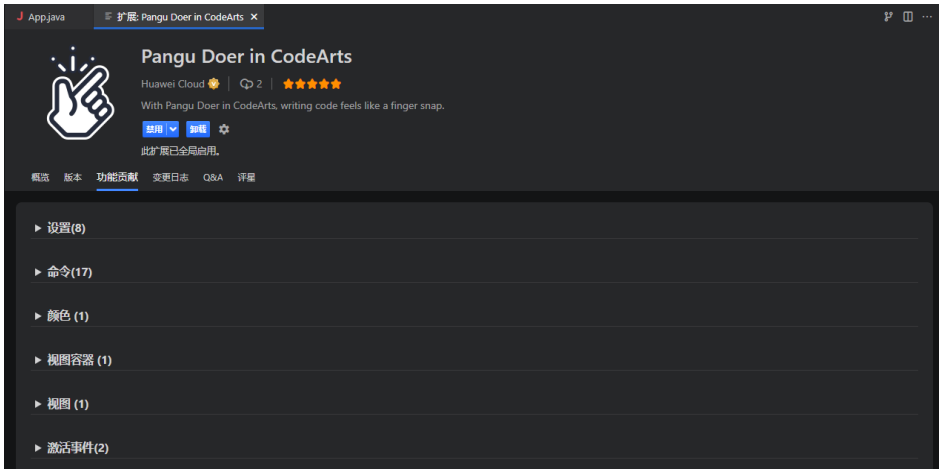
- 版本：扩展发布的历史版本。如下图所示：

图 14-5 扩展详情版本



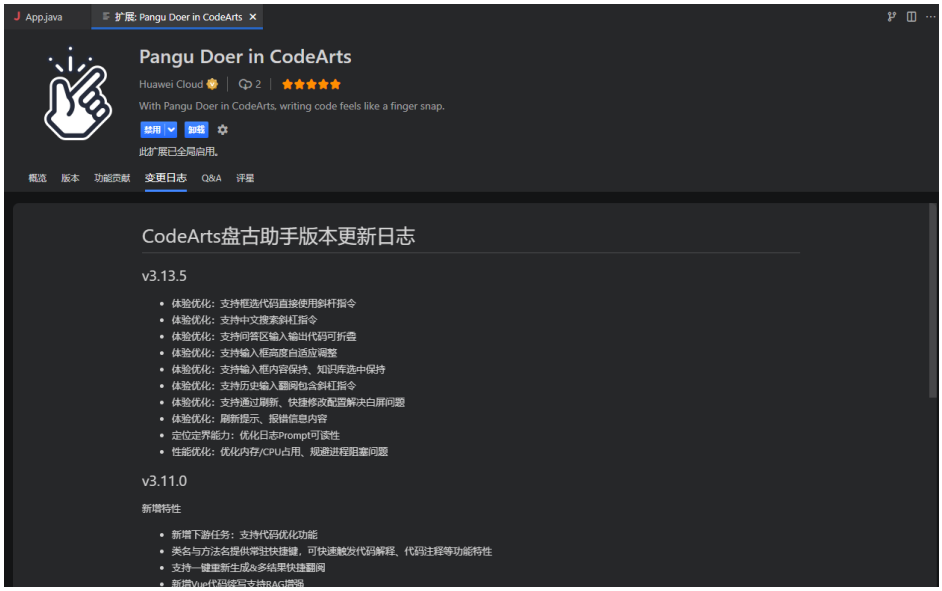
- 功能贡献：包含扩展注册的命令，设置，颜色，视图等信息。如下图所示：

图 14-6 扩展详情功能贡献信息



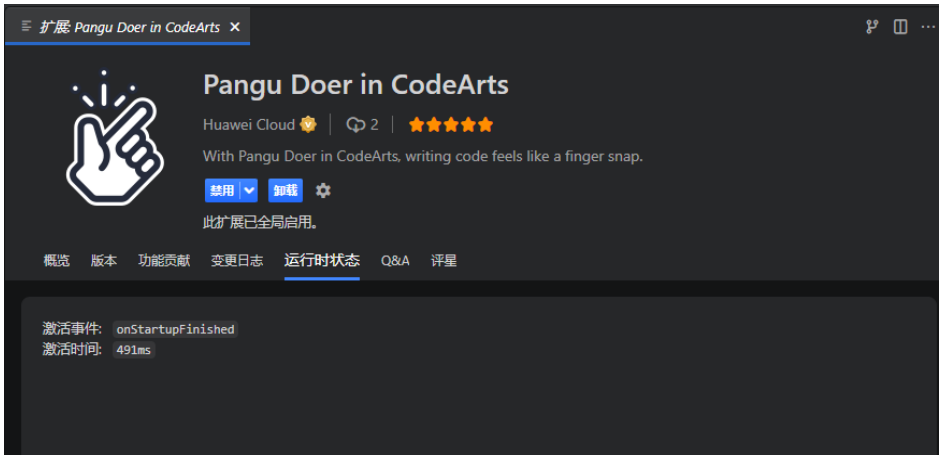
- 变更日志：展示扩展每次更新的内容。如下图所示：

图 14-7 扩展详情变更日志



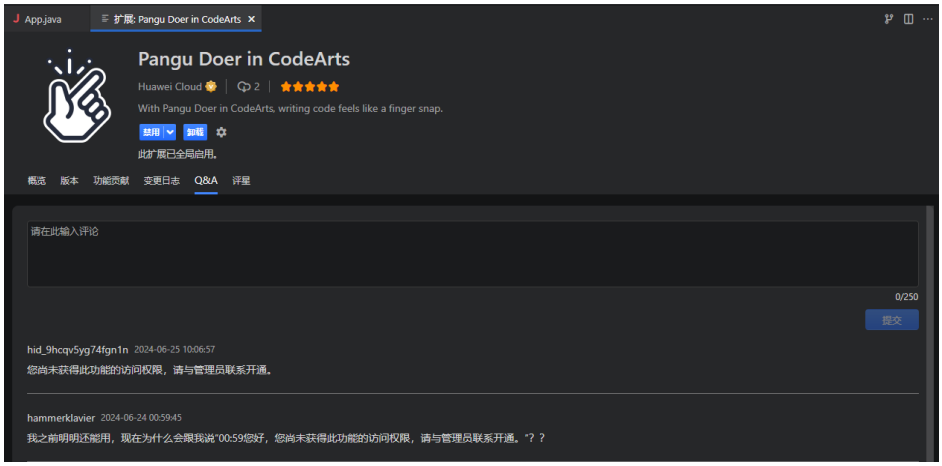
- 运行时状态：展示扩展状态、激活事件和激活时间。如下图所示：

图 14-8 扩展详情运行时状态



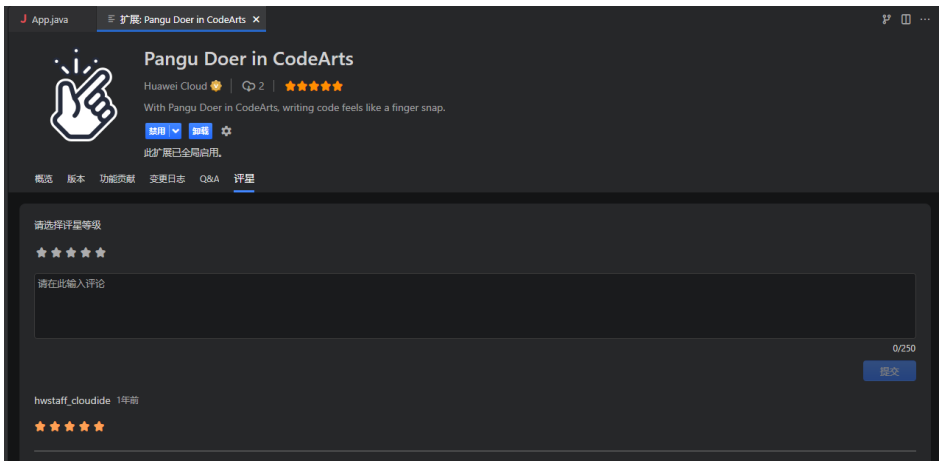
- Q&A：华为云扩展市场上此扩展的Q&A，在登录后，可以直接在 IDE 中发表评论或回复评论。OpenVSX扩展暂不支持功能。如下图所示：

图 14-9 扩展详情 Q&A



- 评星：华为扩展市场上此扩展的用户评星，在登录后，可以直接在CodeArts IDE中发表评星及评价。OpenVSX插件暂不支持发表评星功能。如下图所示：

图 14-10 插件详情评星



14.2 开发第三方扩展

CodeArts IDE为扩展开发者提供了扩展创建、开发调试以及打包发布的完整能力。

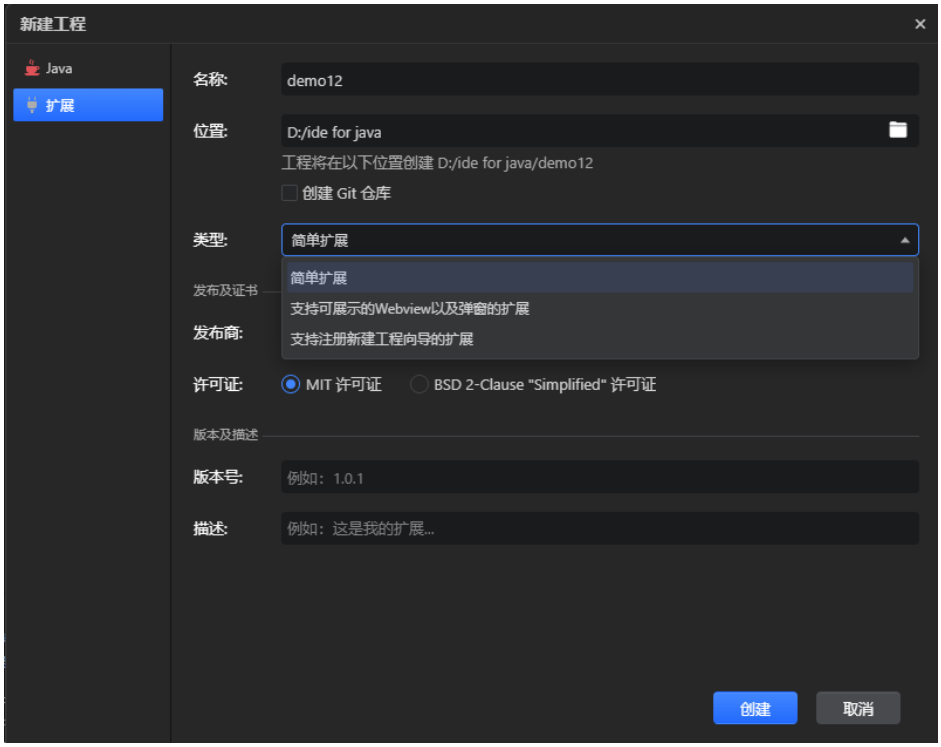
安装环境

在开始扩展开发之前，请在本地命令行客户端（打开cmd）或CodeArts IDE终端中检查是否已安装Node.js 16.10.0或以上版本 (**https://nodejs.org/en/**)。若已安装，可使用命令行node -v以及npm -v查看相应的安装版本。

扩展创建

步骤1 打开CodeArts IDE，单击菜单“文件 >新建 >工程”，选择“扩展”。如下图所示：

图 14-11 创建扩展



- 类型：支持以下三种类型。
  - 简单扩展：不包含后端模板。
  - 支持可展示的Webview以及弹窗的扩展：包含前端和后端的模板。
  - 支持注册创建项目向导的扩展：包含前端和后端的模板。
- 发布商：发布商必须为扩展市场中已创建的发布商，否则将无法在扩展市场上发布扩展，可在发布前在已创建扩展工程中的package.json 文件中修改"publisher"字段。

步骤2 单击“创建”，等待扩展工程创建完成，选择是否在当前窗口打开新建的扩展工程。如下图所示：

图 14-12 选择文件夹打开方式

----结束

后端调试

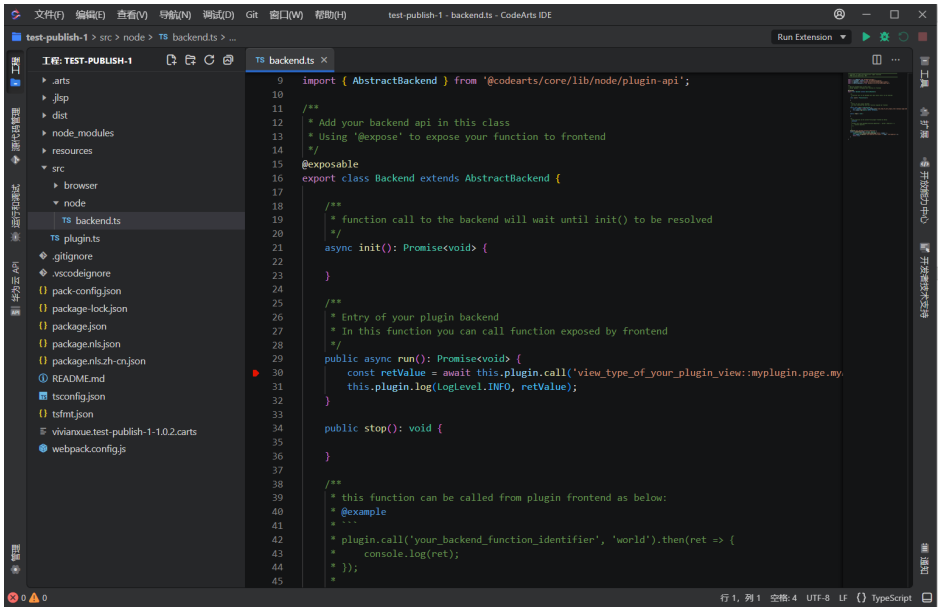
在扩展的src/node/目录下存放的是扩展的后端代码，后端代码运行在nodejs环境中，扩展工程在创建的时候已经默认生成了一个后端文件backend.ts，对于轻量级的扩展，只需要在该文件中添加自己想要实现的业务功能即可，该文件包含了三个默认的方法 init()、run()、stop()。另外还默认添加了一个doSomething方法，这个方法仅仅是作为示例使用，开发者可以根据需要进行修改或删除。

此处简单介绍init，run和stop方法：

- init 函数：作为后端实例的初始化方法，可以在扩展启动的时候进行一些初始化操作，写在该函数中的代码一定会先于 run 和其他函数被调用，这里需要注意的是，对于前端暴露给后端的函数不能在 init 函数中进行调用，也就是不能在 init 方法中执行 this.plugin.call 调用。
- run 函数：作为后端实例的主逻辑函数，承担着业务功能入口的作用，在该函数中可以方便地调用 CodeArts IDE 的 API，比如 codearts.window.showInformationMessage(`hello world!`); 也可以调用前端暴露出来的函数，也就是可以在该方法中执行 this.plugin.call 调用。
- stop 函数：将会在扩展被停止前被调用，如有需要可以进行一些资源清理的操作。

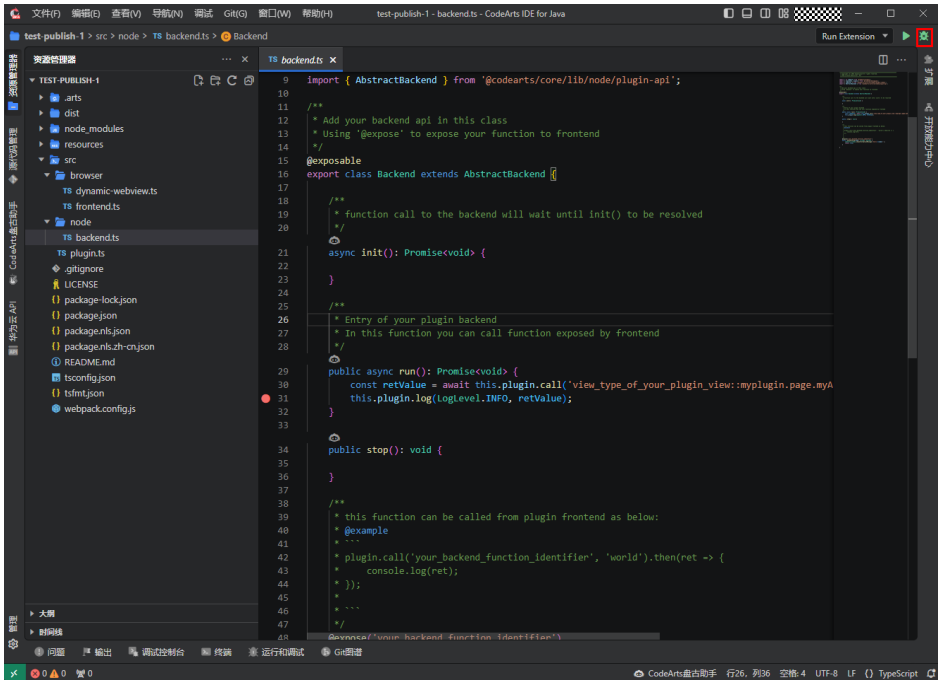
步骤1 添加断点：在backend.ts 的 run() 函数中添加一个断点。如下图所示：

图 14-13 添加断点



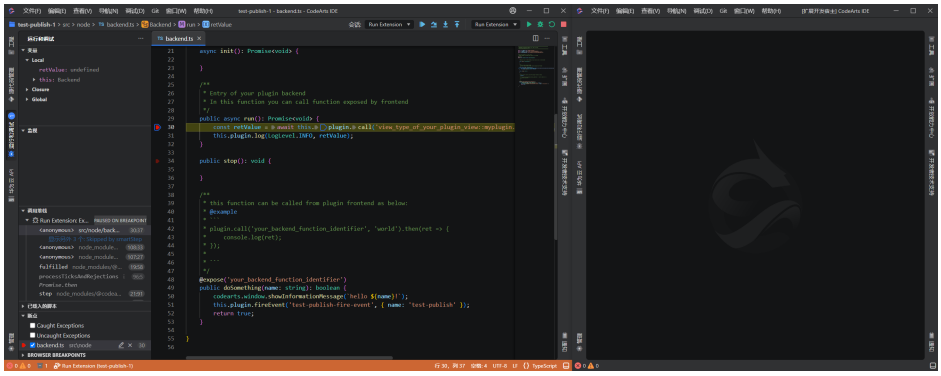
步骤2 打开调试窗口：按 F5 或者单击右上角调试工具栏中的开始调试按钮，打开“扩展开发宿主”窗口。如下图所示：

图 14-14 单击调试按钮



步骤3 进入断点，开始调试。如下图所示：

图 14-15 进入断点，开始调试



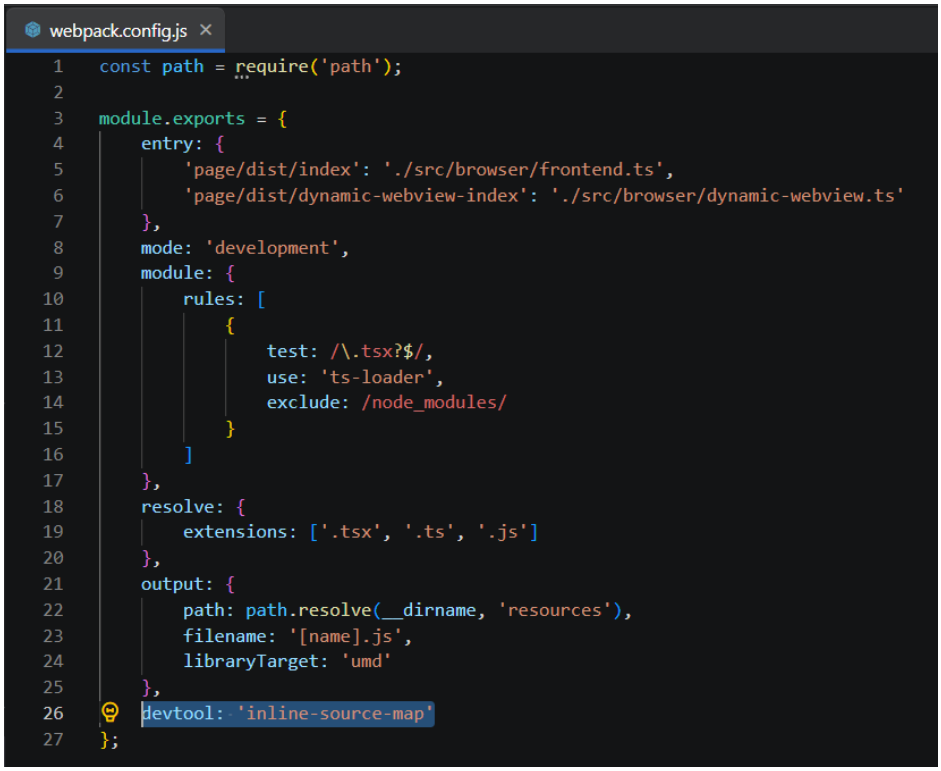
----结束

前端调试

与扩展的后端不同，前端的代码最终将被编译并运行于浏览器环境中，前端的代码存放于 src/browser 目录中，扩展工程在创建的时候会默认生成两个前端源码文件 frontend.ts 和 dynamic-webview.ts。这两个文件的内容与后端 backend.ts 的结构非常相似，只不过运行的环境不同而已，这里就不再重复对这两个文件中 init()、run()、stop() 方法进行介绍。由于前端运行在浏览器环境中，代码调试将借助于浏览器自带的调试功能。如果需要自动重新编译前端代码，可以在终端中执行命令npm run watch-browser，然后再运行调试。在启动调试后如果修改了代码，只需在调试窗口按 Ctrl+R 重新加载窗口即可看到修改后的效果。

步骤1 前端调试前，需要先把 webpack.config.js 文件中的 devtool 配置为 'inline-source-map'，然后在命令行执行 npm run prepare。如下图所示：

图 14-16 修改 webpack 文件

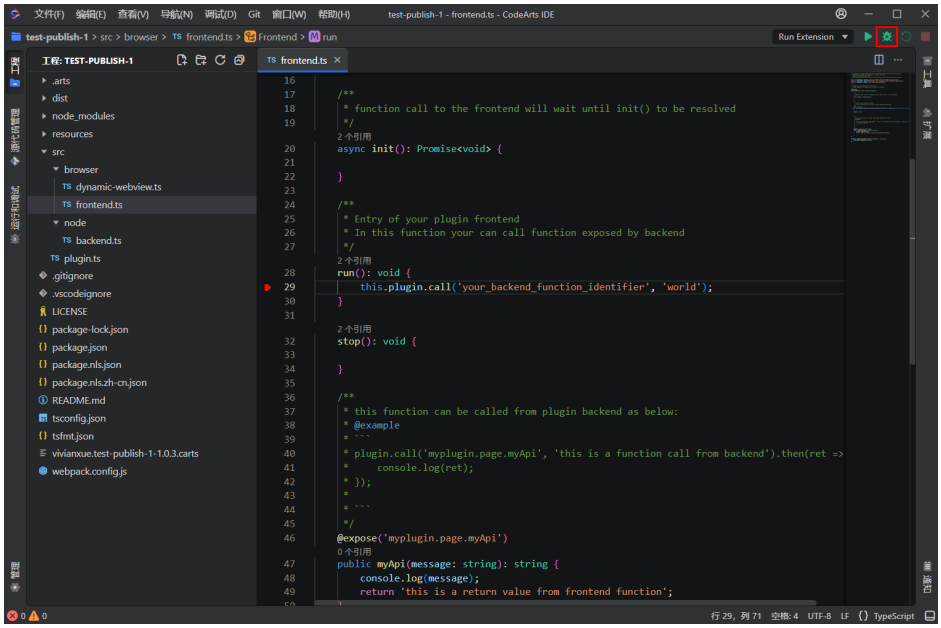


```
webpack.config.js
1  const path = require('path');
2
3  module.exports = {
4      entry: {
5          'page/dist/index': './src/browser/frontend.ts',
6          'page/dist/dynamic-webview-index': './src/browser/dynamic-webview.ts'
7      },
8      mode: 'development',
9      module: {
10         rules: [
11             {
12                 test: /\.tsx?$/,
13                 use: 'ts-loader',
14                 exclude: /node_modules/
15             }
16         ]
17     },
18     resolve: {
19         extensions: ['.tsx', '.ts', '.js']
20     },
21     output: {
22         path: path.resolve(__dirname, 'resources'),
23         filename: '[name].js',
24         libraryTarget: 'umd'
25     },
26     devtool: 'inline-source-map'
27 };
```

步骤2 添加断点：在frontend.ts的run() 函数中添加一个断点。

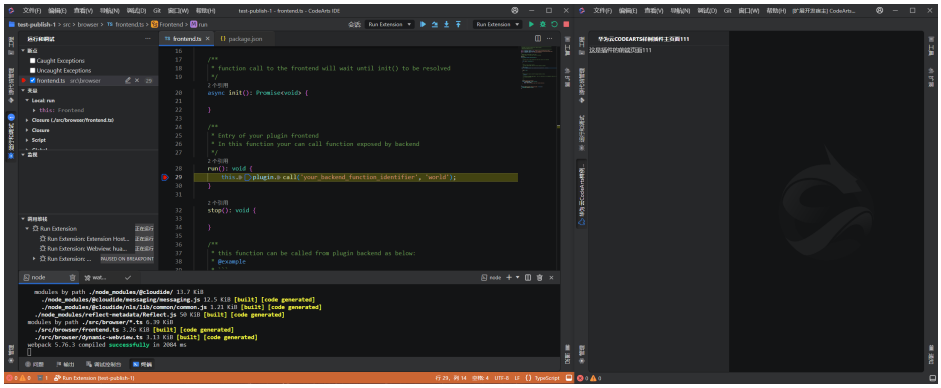
步骤3 打开调试窗口：按 F5 或者单击右上角调试工具栏中的开始调试按钮，打开“扩展开发宿主”窗口。如下图所示：

图 14-17 添加断点，单击调试按钮



步骤4 打开扩展注册的视图，进入断点，进行前端的调试，若无法进入断点，可以使用“Ctrl + Shift + I”打开“开发人员工具”，再“Ctrl + R”重新加载当前窗口。如下图所示：

图 14-18 进入断点



----结束

后端调用前端

- **在前端定义暴露给后端的方法**

  打开 src/browser/frontend.ts 文件，其中 Frontend 类继承自 AbstractFrontend，除了需要实现的 init()、run()、stop() 这三个方法，自定义了一个 myApi(message: string) 方法，如果想要把 myApi 方法暴露给后端去调用，只需要在函数上添加 @expose('function_id') 修饰器。

  注意：多个expose修饰器中的function_id不能重复。

```
@expose('myplugin.page.myApi')public myApi(message: string): string {
    console.log(message);
    return 'this is a return value from frontend function';
}
```

- **在后端调用前端暴露的方法**

  打开src/node/backend.ts 文件，其中Backend类继承自AbstractBackend，需要实现 init()， run()， stop() 这三个方法，可以在run() 方法中通过this.plugin.call() 调用在前端定义的 myApi 方法并获取到返回值。

```
public async run(): Promise<void> {
const=awaitthis.plugin.call('view_type_of_your_plugin_view::myplugin.page.myApi','this is a function call from backend');    const retValue = await this.plugin.call('view_type_of_your_plugin_view::myplugin.page.myApi', 'this is a function call from backend');
this.log(.INFO,);
this.plugin.log(LogLevel.INFO, retValue);
}}
```

前端调用后端

类似的，可以在后端定义自己的方法并将方法暴露给前端调用。

- **在后端定义暴露给前端的方法**

  打开src/node/backend.ts文件，自定义一个 doSomething(name: string) 方法。

```
@expose('your_backend_function_identifier')public doSomething(name: string): boolean {
    codearts.window.showInformationMessage(`hello ${name}!`);
    return true;
}
```

- **在前端调用后端暴露的方法**

  打开src/browser/frontend.ts文件，在 run() 方法中通过 this.plugin.call() 调用在后端定义的 doSomething 方法。

```
run(): void {
    this.plugin.call('your_backend_function_identifier', 'world');
}
```

事件订阅：发布和监听事件

- **在扩展后端监听事件**

  打开src/node目录下的backend.ts文件，在Backend类的run() 方法中添加如下代码注册监听一个文件删除的事件。

```
const registeredEvent = codearts.workspace.onDidDeleteFiles((event) => {
    codearts.window.showInformationMessage(`${event.files.join(',')} deleted.`);
});
this.plugin.context.subscriptions.push(registeredEvent);
```

  如果想要删除这个事件的监听可以直接调用registeredEvent的dispose() 方法即可。

  大家可以尝试注册一些其他的事件并测试效果。

- **在扩展前端监听事件**

  打开src/browser下的frontend.ts文件，通过在Frontend类的run() 方法中添加如下代码注册监听一个改变当前活动的编辑器的事件。

```
const eventHandler = (eventType: any, evt: any) => {
    // do something
};
this.plugin.subscribeEvent(EventType.WINDOW_ONDIDCHANGEACTIVETEXTEDITOR, eventHandler);
```

  前端取消事件注册的方式和后端并不相同，需要使用 plugin 对象的 unsubscribeEvent 方法取消注册的事件处理句柄。


```
this.plugin.unsubscribeEvent(EventType.WINDOW_ONDIDCHANGEACTIVETEXTEDITOR, eventHandler);
```

国际化

扩展创建完后，在根目录下默认生成了package.nls.json和package.nls.zh-cn.json文件，package.nls.json文件用来记录默认情况下的翻译词条，比如没有找到对应语言的翻译文件扩展框架将默认采用该文件中的词条。package.nls.zh-cn.json则是中文简体的翻译词条文件，如果扩展需要支持其他语言也可以自行添加翻译文件。

localize方法需要提供了一个key参数来指定使用国际化文件中的词条索引键值，后续的不定参数用来对翻译词条中的占位符进行替换，词条中支持使用"{0} {1} {2}"这样的格式进行占位，localize 方法的第二个参数开始会被依次替换到占位符中。

```
localize(key: string, ...args: any[]): string;
```

示例如下：

```
{
   "plugin.welcome": "Welcome!",
   "plugin.hello": "Hello {0}!"
}
```

- **内置成员plugin的localize方法**

  还在前后端内置的 plugin 成员变量中实现了localize方法。Frontend类 (src/browser/frontend.ts) 和Backend类 (src/browser/backend.ts) 分别继承了AbstractFrontend前端类和AbstractBackend后端类，可以直接使用this.plugin.localize 方法进行本地化翻译。

  ```
  // 不带参数
  this.plugin.localize('plugin.welcome');
  // 带参数
  this.plugin.localize('plugin.hello', 'world');
  ```

- **直接引入localize方法**

  ```
  import { localize } from '@cloudide/nls';
  ```

  使用如下代码就可以将词条填充为： Hello World!

  ```
  localize('plugin.hello', 'World');
  ```

- **页面文件中的国际化方法**

  通用扩展可以使用ejs和pug引擎来渲染界面，无论是ejs还是pug引擎扩展框架都为开发者提供了一个 l10n 内置对象，里面存储了当前所选语言的翻译词条列表。

  对于选择ejs引擎来做界面渲染的开发者可以在ejs文件中使用如下方式来对需要本地化的文案进行翻译：

  ```
  <%= l10n['plugin.hello'] %>
  ```

  对于使用 pug 引擎的开发者可以使用如下方式：

  ```
  #{l10n['plugin.hello']}
  ```

14.3 发布第三方扩展

扩展打包安装

在终端执行命令“npm run package”可以对扩展进行打包，打包文件后缀为“.carts”。

在 IDE 安装打包的扩展后即可使用。如下图所示：

图 14-19 安装打包后的扩展

扩展发布

通过IDE直接发布到扩展市场

步骤1 若还没有创建发布商，请参考**《CodeArts IDE插件市场帮助文档》**，前往扩展市场创建一个发布商。若已创建发布商但还未创建发布商凭证，请前往 **插件市场发布商管理** 创建，以下是创建凭证的步骤：

1. 单击新增凭证。如下图所示：

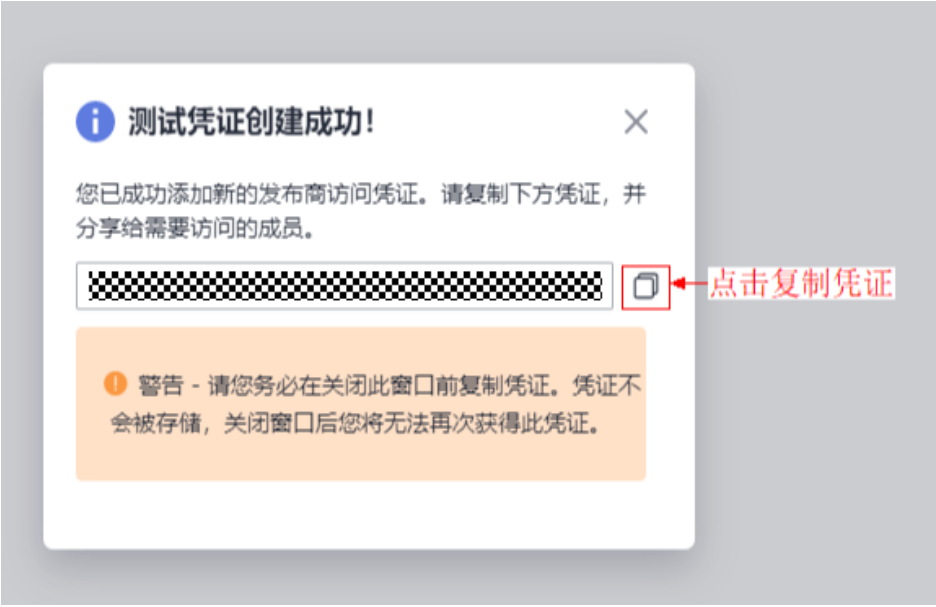
图 14-20 单击新增凭证

2. 输入凭证名称，并设置过期时间。如下图所示：

图 14-21 创建发布商凭证

3. 创建成功，请妥善保存发布商凭证，关闭窗口后将无法再次获得此凭证。如下图所示：

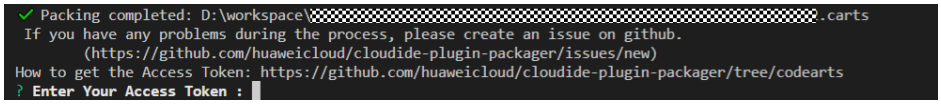
图 14-22 凭证创建成功



步骤2 在发布前，请确认 package.json 中的 publisher 与发布商凭证对应的发布商的唯一标识相符。

在 IDE 终端执行命令“npm run publish”，等待打包完成后输入发布商凭证，按回车确认。如下图所示：

图 14-23 在终端发布扩展



步骤3 等待扩展上传并发布。

步骤4 发布成功后，可以在扩展市场的扩展管理中看到已上传的扩展。如下图所示：

图 14-24 扩展市场查看已上传扩展

----结束

通过扩展市场进行发布

请参考**《CodeArts IDE插件市场帮助文档》**，按照步骤在扩展市场上传和管理自己的扩展。