CodeArts IDE

用户指南

文档版本 01

发布日期 2024-04-12

华为技术有限公司

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。

安全声明

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

目 录

1 激活 CodeArts IDE 客户端

1.1 购买激活码

步骤1 在CodeArts IDE客户端完成登录并且单击“购买或管理”激活码后，将会自动跳转到CodeArts IDE控制台页面。若此时您已有未绑定的有效激活码，可跳转至**绑定激活码**。在该页面右上角单击“购买激活码”即可开始购买流程。也可以直接访问**购买CodeArts IDE激活码**进行购买。

步骤2 进入到激活码购买页面后，您可选择是否自动续费，激活码数量（单次购买数量最多为10），以及是否绑定当前用户（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。

----结束

1.2 绑定激活码

步骤1 如果您在**购买激活码**时勾选了“绑定当前用户”，可以省略该步骤。如果未勾选，则您需要在购买后返回CodeArts IDE控制台。在您购买的激活码右侧选择“绑定”，单击后将会帮您绑定到当前账号。

步骤2 完成绑定后，您可以回到CodeArts IDE弹出的提示页面，或者单击右上角的“立即激活”按钮（在试用期内），完成激活。

----结束

1.3 解绑激活码

打开CodeArts IDE控制台页面，选择需要解绑的激活码，单击右侧“解绑”后再单击“确定”完成解绑。解绑后当前用户将不可再使用CodeArts IDE。

2 管理权限

2.1 授权 IAM 用户

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 为用户授权 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

   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 自定义权限策略

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 基本操作

3.1 代码编辑

3.1.1 基本信息

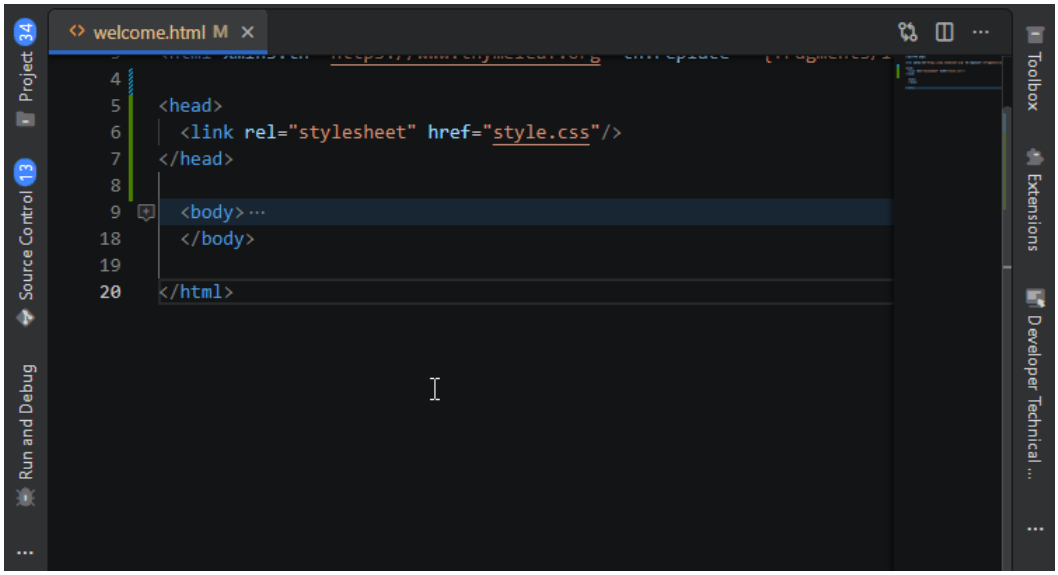
CodeArts IDE作为一个代码编辑器，包含了您所需要的高效的代码编辑功能。本节将带您了解使用CodeArts IDE编辑代码的基本知识。

创建或打开文件

步骤1 鼠标移动至文件链接处style.css。

步骤2 按下“Ctrl”，鼠标单击style.css快速打开文件或图像。

----结束

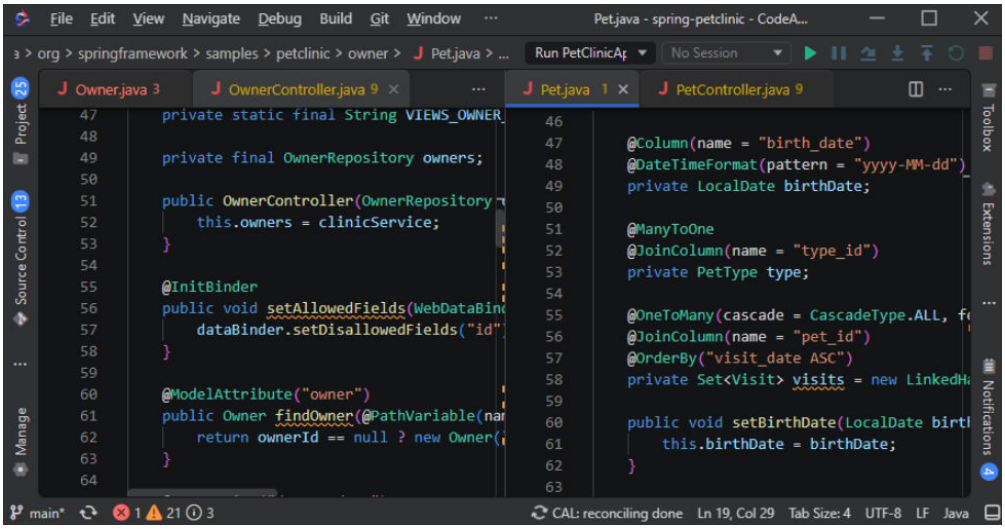


并排编辑

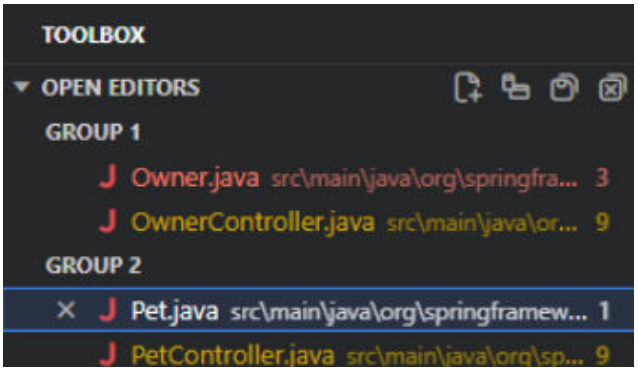
您可以在垂直和水平方向打开编辑器。如果您已经打开了一个编辑器，您可以通过以下方式在现有编辑器的一侧打开另一个编辑器：

- 按住“**Alt**”或“**Ctrl**”键双击**资源管理器**中的一个文件。
- 按“**Ctrl+**”将现有的编辑器一分为二。
- **资源管理器**中右键单击文件，在上下文菜单中选择“在侧边打开”（“Ctrl +Enter”）。
- 单击编辑器右上角的。
- 可以将文件拖动到编辑器区域的任意一侧。

拆分编辑器时，将创建编辑器组，这些编辑器组可以容纳多个选项卡。

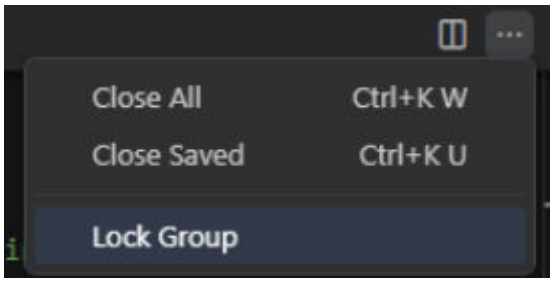


您可以通过“打开的编辑器”视图查看所有创建的编辑器组，您可以通过单击右侧活动栏中的打开该视图。



要快速聚焦编辑器组，请使用键盘快捷键“**Ctrl+1**”（聚焦第一组）、“**Ctrl+2**”（聚焦第二组）、“**Ctrl+3**”（聚焦第三组），以此类推。

新打开的文件将在当前聚焦组内打开选项卡。如有需要，您可以锁定组，以防止在其中打开新选项卡。要执行此操作，请单击组标题中的“更多操作”按钮（···），然后从弹出菜单中选择“锁定组”。



快速滚动

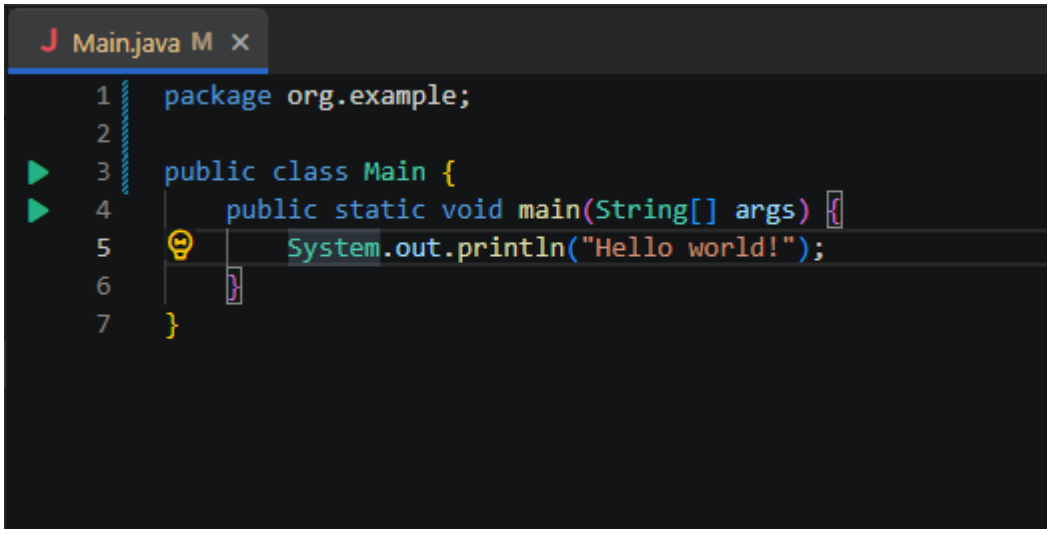
按**Alt**键的同时滚动鼠标滚轮可在编辑器和**资源管理器**中快速滚动。默认情况下，快速滚动速度倍增为5，但您可以使用**Editor: Fast Scroll Sensitivity**设置项来调整它。

向上/向下复制行

步骤1 光标定位到想要复制的行

步骤2 使用快捷键“**Shift+Alt+Up**”/“**Shift+Alt+Down**”向上复制一行/向下复制一行。

----结束



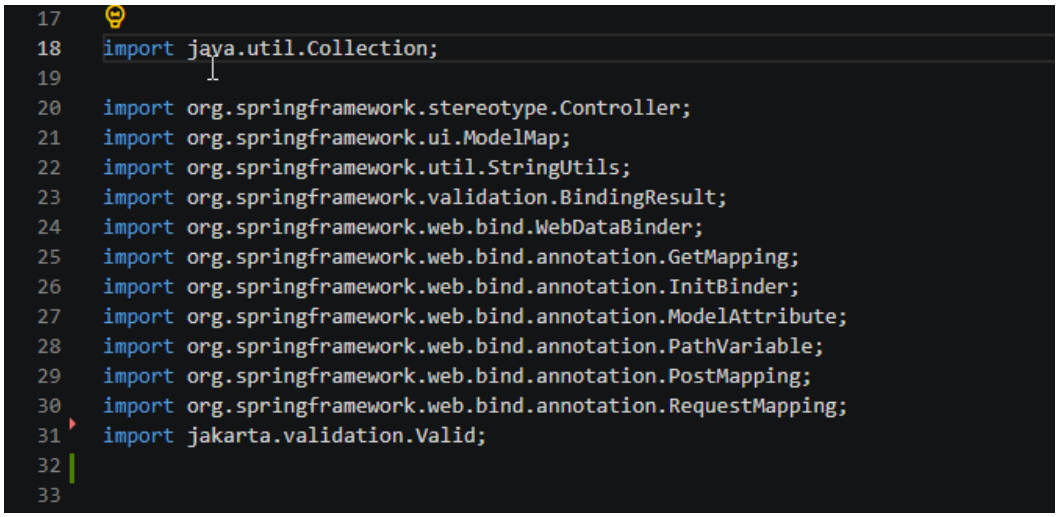
向上/向下移动行

步骤1 光标定位到想要移动的行上面

步骤2 使用快捷键“**Alt+Up**”/“**Shift+Alt+Up**”/“**Ctrl+Shift+Up**”(IDEA键盘映射)向上移动一行，

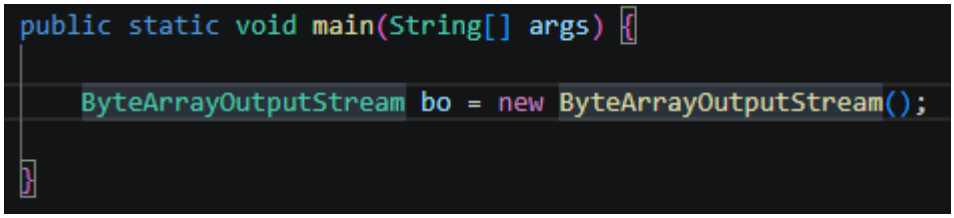
步骤3 使用快捷键“**Alt+Down**”/“**Shift+Alt+Down**”/“**Ctrl+Shift+Down**”(IDEA键盘映射)向下移动一行

----结束

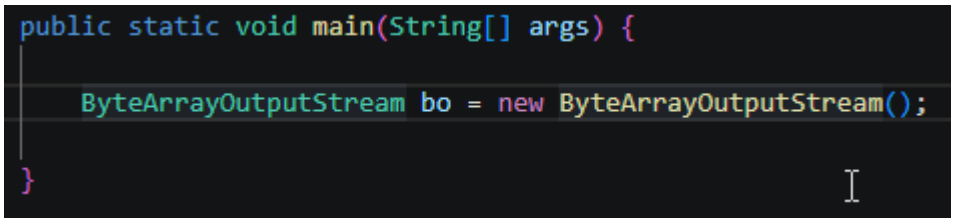


使用 CamelHumps 选择和导航代码

CodeArts IDE可以使用CamelHumps简化代码导航。如果**editor.useCamelHumpsWords**配置已启用，按下“**Ctrl+Arrow**”在单词之间导航会带您在CamelHump单词的各个部分之间进行导航，而不是在整个单词之间进行导航。

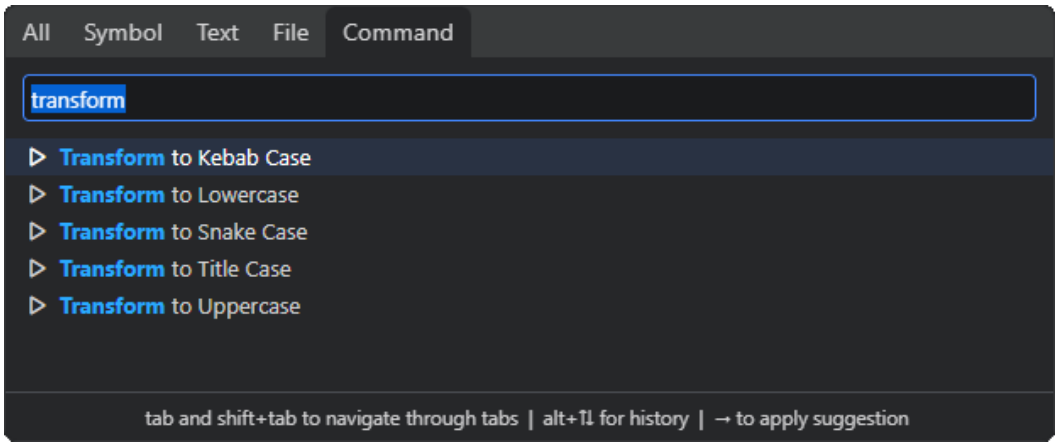


如果**editor.honorCamelHumpsWords**配置启用后，双击CamelHump单词只会选择其双击的部分，而不是整个单词。



转换文本

您可以使用**命令面板**（“**Ctrl+Shift+P**”/“**Ctrl+Ctrl**”）中的**转换**命令将选定的首字母自动转换为大写、小写、首字母大小写、串式命名和蛇形命名。



提示:

- Kebab Case（串式命名）：又称破折号方式(dash-case)，每个单词全小写或全大写，多单词使用中划线隔开。
- Snake Case（蛇形命名）：每个单词全小写或全大写，多单词使用下划线隔开。

保存/自动保存

默认情况下，CodeArts IDE需要手动操作来保存更改，键盘快捷键：“**Ctrl+S**”。

同时，您可以打开自动保存，这将在配置的指定延迟后或焦点离开编辑器时保存更改。启用此选项后，无需手动保存文件。打开自动保存的最简单方法是单击“文件>自动保存”开关，在延迟后打开或关闭保存。

有关对自动保存的更多控制，请打开用户设置并查找相关设置：

- files.autoSave：可以设置为以下值：
  - off - 禁用自动保存。
  - afterDelay - 在配置的延迟后自动保存文件（默认1000毫秒）。
  - onFocusChange - 当焦点移出未保存的编辑器时自动保存文件。
  - onWindowChange - 当焦点移出CodeArts IDE窗口时自动保存文件。
- files.autoSaveDelay：当file.autoSave被配置为afterDelay时，配置延迟以毫秒为单位。默认值为1000毫秒。

说明

您可以通过**本地历史记录**查看文件的历史，或者如果您的项目关联了Git仓库，可以使用CodeArts IDE内置的**Git支持**。

热退出

默认情况下退出CodeArts IDE时，CodeArts IDE将记住对文件的未保存更改。当通过**文件>退出**关闭应用程序或关闭最后一个窗口时，将触发热退出。

您可以通过将files.hotExit设置为以下值来配置热退出：

- off：禁用热退出。
- onExit：当应用程序关闭时，即最后一个窗口关闭或触发workbench.action.quit命令时（从命令选项板、键盘快捷键或菜单），将触发热退出。所有未打开文件夹的窗口将在下次启动时恢复。

- onExitAndWindowClose：当应用程序关闭时，即最后一个窗口关闭或触发workbench.action.quit命令时（从**命令面板**、键盘快捷键或菜单），以及任何打开文件夹的窗口，无论它是否是最后一个窗口，都将触发热退出。所有未打开文件夹的窗口将在下次启动时恢复。要将文件夹窗口恢复为关闭前的状态，请将window.restoreWindows设置为all。

如果热退出出现问题，所有备份都存储在以下文件夹中，用于标准安装位置：

- Windows：**%APPDATA%\CodeArts\Backups**。

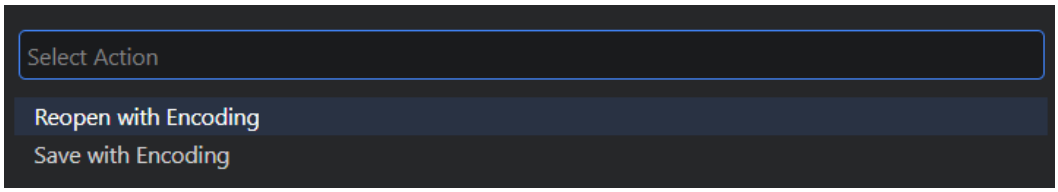
文件编码设置

通过用户设置或工作区设置中的file.encode设置来设置全局或每个工作区的文件编码。

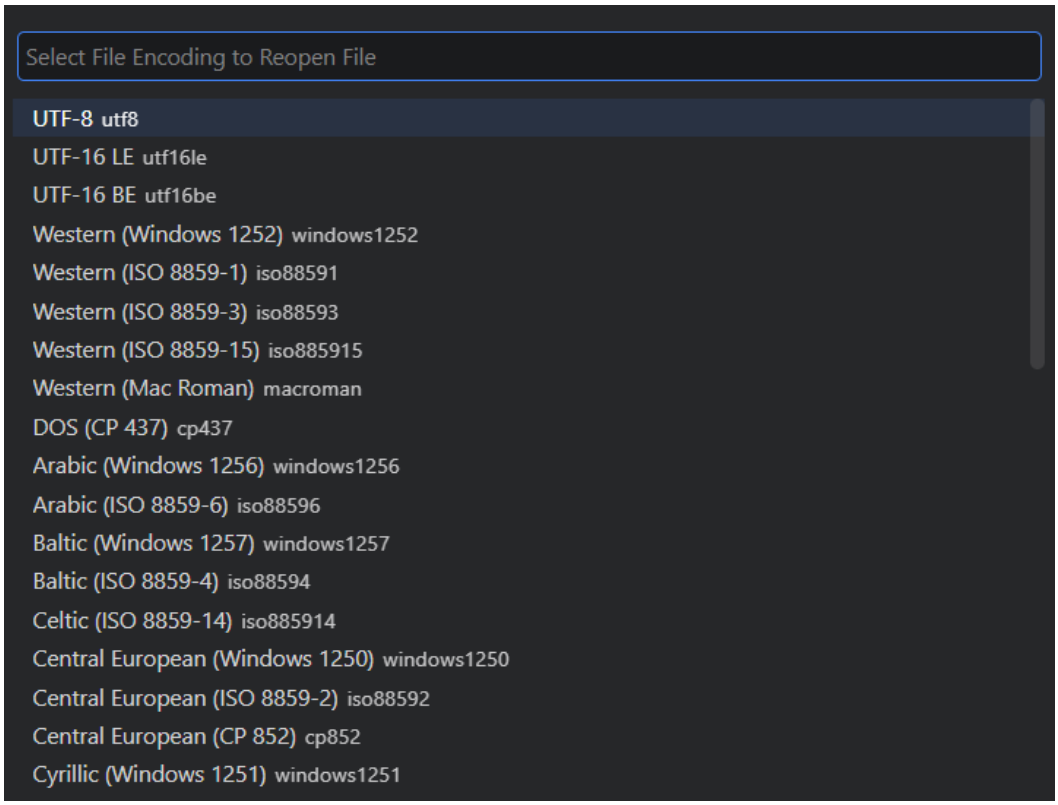
```
//-------- Files configuration --------

// The default character set encoding to use when reading and writing files.
"files.encoding": "utf8",
```

您可以在CodeArts IDE状态栏中查看文件编码。

单击状态栏中的编码按钮可使用不同编码重新打开或保存活动文件。



然后选择编码类型。



重新恢复关闭的页面

当您关闭了一个文件时，如果想要重新打开，可以按“Ctrl+Shift+T”重新恢复该页面。

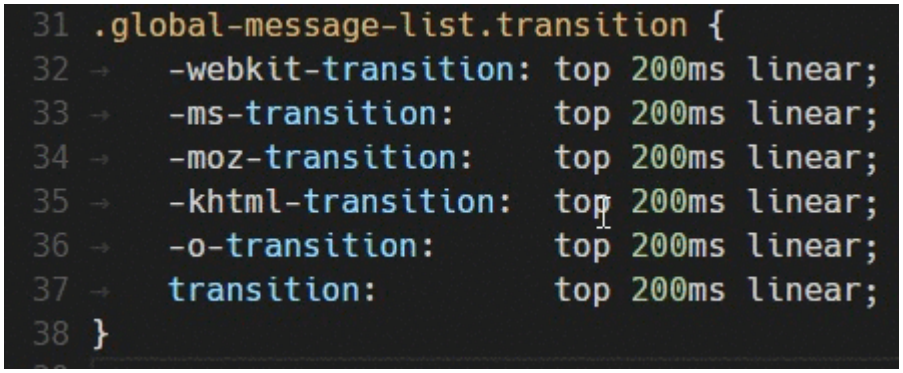
3.1.2 选择代码

选择当前行

键盘快捷键：**Ctrl+L**

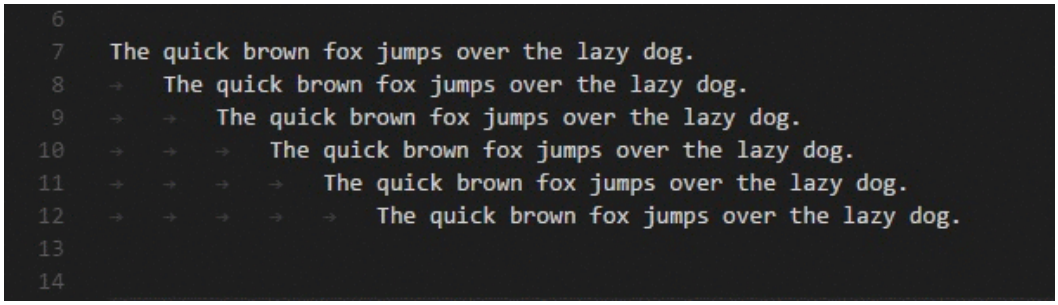
多重选择（多光标）

CodeArts IDE支持多个光标以实现快速的同步编辑。您可以使用**“Alt+单击”**添加二级光标。每个光标根据其所在的上下文独立运行。添加更多光标的常见方法是使用**Ctrl+Alt+Down**或**Ctrl+Alt+Up**将光标插入下方或上方。



注意：您的显卡驱动程序可能会覆盖这些默认的快捷方式。

按“**Ctrl+D**”/“**Alt+J**”**(IDEA键盘映射)**选择光标处的单词，或选择当前出现的下一个匹配的单词。



您还可以使用“**Ctrl+Shift+L**”/“**Ctrl+Shift+Alt+J**”（**IDEA键盘映射**）同时添加更多光标，这将在当前选中的文本的每一个匹配项添加一个光标。

多光标修改器

如果您想将通过鼠标添加多个光标时使用的修改键改为“**Ctrl+Click**”，您可以通过editor.multiCursorModifier设置项来实现。这让来自其他编辑器（如Sublime Text或Atom）的用户可以继续使用他们熟悉的键盘修改器。

设置项可以被设置为：

- ctrlCmd - 映射到Windows上的“**Ctrl**”。
- alt - 现有的默认“**Alt**”。

在"**编辑**"菜单中，使用菜单**Switch to Ctrl+Click for Multi-Cursor**可快速切换此设置。

“转到定义”和“打开链接”功能所需的鼠标动作将会相应调整，并且不会与多光标修改键冲突。例如，当设置为ctrlCmd时，可以使用“**Ctrl+Click**”添加多个光标，打开链接或转到定义可以使用“**Alt+Click**”调用。

缩小/扩大选区

快速缩小或展开当前选定内容。可使用“**Shift+Alt+Left**”/“**Ctrl+Shift+W**”（**IDEA键盘映射**）和“**Shift+Alt+Right**”触发。

下面是一个用“**Shift+Alt+Right**”扩大选区的例子：

```
int main(int argc,char *argv[])
{
   int decompress = 0;
   int level = 9;
   char *fn_r = NULL;
   char *fn_w = NULL;

#ifdef _WIN32
   if(BZ2DLLLoadLibrary()<0){
      fprintf(stderr,"Loading of %s failed.  Giving up.\n", BZ2_LIBNAME);
      exit(1);
   }
   printf("Loading of %s succeeded.  Library version is %s.\n",
          BZ2_LIBNAME, BZ2_bzlibVersion() );
#endif
   while(++argv,--argc){
      if(**argv =='-' || **argv=='/'){
         char *p;
         for(p=*argv+1;*p;p++){
            if(*p=='d'){
               decompress = 1;
            }else if('1'<=*p && *p<='9'){
               level = *p - '0';
            }else{
               usage();
               exit(1);
            }
         }
      }else{
         break;
      }
   }
```

列（框）选择模式

将光标放置在第一行的右上角，然后按住“**Shift+Alt**”，同时拖动光标到最后一行的右下角，即可选中所有行中的指定内容。当“**Ctrl**”设置为多光标功能（**多光标修改器**）时，请使用“**Shift+Ctrl**”。

```
s->blockNo           = 0;
s->state             = BZ_S_INPUT;
s->mode              = BZ_M_RUNNING;
s->combinedCRC       = 0;
s->blockSize100k     = blockSize100k;
s->nblockMAX         = 100000 * blockSize100k - 19;
s->verbosity         = verbosity;
s->workFactor        = workFactor;

s->block             = (UChar*)s->arr2;
s->mtfv              = (UInt16*)s->arr1;
s->zbits             = NULL;
s->ptr               = (UInt32*)s->arr1;
```

还有一些默认的键绑定用于列选择。

Key	Command	Command ID
Ctrl+Shift+Alt+Down，Shift+Down	**Column Select Down**	cursorColumnSelectDown

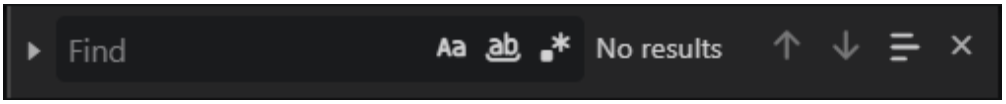
Key	Command	Command ID
Ctrl+Shift+Alt+Up，Shift+Up	**Column Select Up**	cursorColumnSelectUp
Ctrl+Shift+Alt+Left，Shift+Left	**Column Select Left**	cursorColumnSelectLeft
Ctrl+Shift+Alt+Right，Shift+Right	**Column Select Right**	cursorColumnSelectRight
Ctrl+Shift+Alt+Pagedown，Shift+Pagedown	**Column Select Page Down**	cursorColumnSelectPageDown
Ctrl+Shift+Alt+Pageup，Shift+Pageup	**Column Select Page Up**	cursorColumnSelectPageUp

3.1.3 代码搜索

3.1.3.1 查找和替换

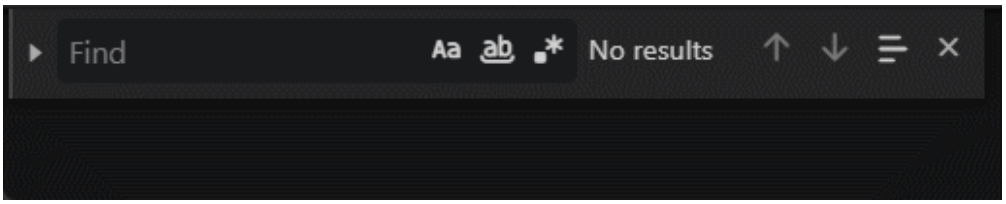
CodeArts IDE支持您快速查找和替换当前打开的文件文本。

- 按“**Ctrl+F**”在编辑器中打开查找小组件，搜索结果将在编辑器和右侧缩略图突出显示。



如果当前打开的文件中存在多个匹配结果，可以按“**Enter**”/“**Shift+Enter**”和“**F3**”/“**Shift+F3**”导航到下一个或上一个结果。

- 要将小组件切换到替换模式，请单击展开箭头，或按“**Ctrl+H**”/“**Ctrl+R**”（**IDEA键盘映射**）。



从选定内容中搜索种子字符串

光标定位在任意文本或者选中内容，打开查找小组件时，它会自动将编辑器中选中的文本填充到查找输入框中。如果选择为空，则将插入光标下的单词。如果想关闭此功能，可以通过搜索字符串选择项：editor.find.seedSearchStringFromSelection设置为false，将此功能关闭。

```
/*---------------------------------------------*/
static
int bz_config_ok ( void )
{
   if (sizeof(int)   != 4) return 0;
   if (sizeof(short) != 2) return 0;
   if (sizeof(char)  != 1) return 0;
   return 1;
}


/*---------------------------------------------*/
static
void* default_bzalloc ( void* opaque, Int32 items, Int32 size )
{
   void* v = malloc ( items * size );
   return v;
}
```

在选定内容中查找

默认情况下，可以在编辑器的整个文件内查找，也可以在选定的文本上查找。您可以通过单击查找小组件中的“在选择中查找”按钮（ ）来打开此功能。如果您希望它成为查找小组件的默认行为，则可以将editor.find.autoFindInSelection选择项设置为always，如果您希望仅在选择的多行内容上查找，则可以将其设置为multipleline。

```
n       = 100000 * blockSize100k;
s->arr1 = BZALLOC( n                   * sizeof(UInt32) );
s->arr2 = BZALLOC( (n+BZ_N_OVERSHOOT) * sizeof(UInt32) );
s->ftab = BZALLOC( 65537               * sizeof(UInt32) );

if (s->arr1 == NULL || s->arr2 == NULL || s->ftab == NULL) {
   if (s->arr1 != NULL) BZFREE(s->arr1);
   if (s->arr2 != NULL) BZFREE(s->arr2);
   if (s->ftab != NULL) BZFREE(s->ftab);
   if (s       != NULL) BZFREE(s);
   return BZ_MEM_ERROR;
}
```

多行支持和查找小部件调整大小

您可以通过将文本粘贴到“查找”字段和“替换”字段中来搜索多行文本。

- 要在字段中插入新行，请按**“Ctrl+Enter”**键。

```
/*---------------------------------------------*/
static
void prepare_new_block ( EState* s )
{
   Int32 i;
   s->nblock = 0;
   s->numZ = 0;
   s->state_out_pos = 0;
   BZ_INITIALISE_CRC ( s->blockCRC );
   for (i = 0; i < 256; i++) s->inUse[i] = False;
   s->blockNo++;
}


/*---------------------------------------------*/
static
void init_RL ( EState* s )
{
   s->state_in_ch  = 256;
   s->state_in_len = 0;
}
```

搜索长文本时，您可能会发现查找小组件的默认输入框宽度太小。

- 拖动左侧窗框以放大查找小部件.
- 双击左侧窗框以最大化查找小部件或将其缩小到默认大小。

高级查找和替换选项

除了查找和替换为纯文本外，查找小组件还有三个高级搜索选项：

- 区分大小写(Aa)“Alt+C”
- 全字匹配(Ab|)“Alt+W”
- 使用正则表达式(.*)“Alt+R”

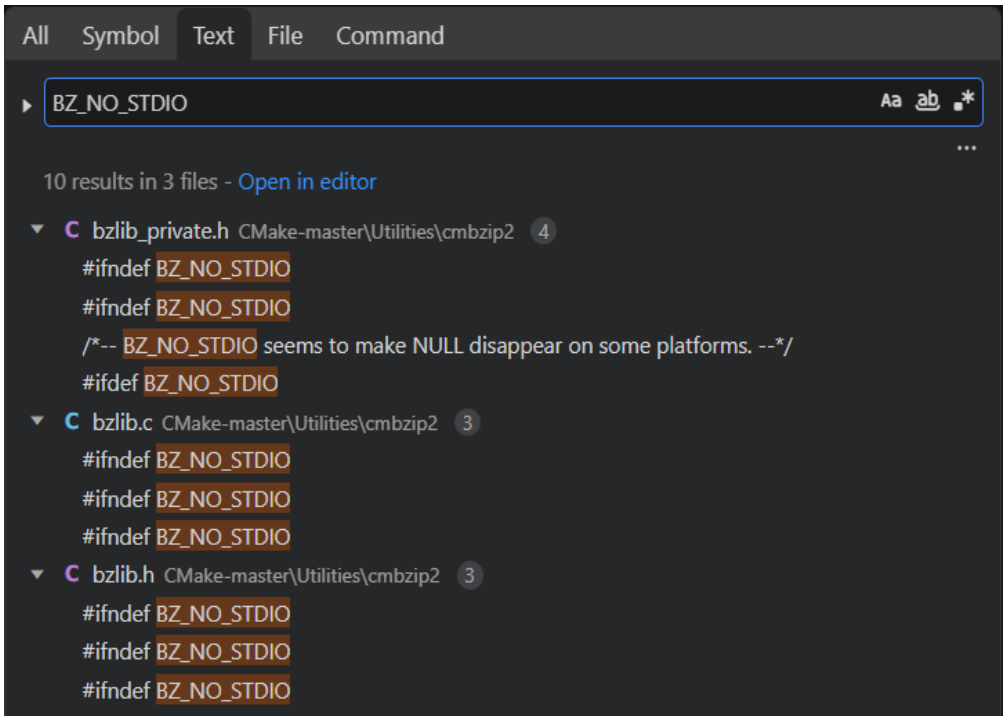
您可以保留被替换文本中的大小写（即全大写、全小写和标题大小写）。要做到这一点，您可以通过单击**Preserve Case**(AB)按钮或者快捷键“ALT+P”来使用此功能。

3.1.3.2 跨文件搜索

CodeArts IDE允许您快速搜索当前打开的文件夹中的所有文件。

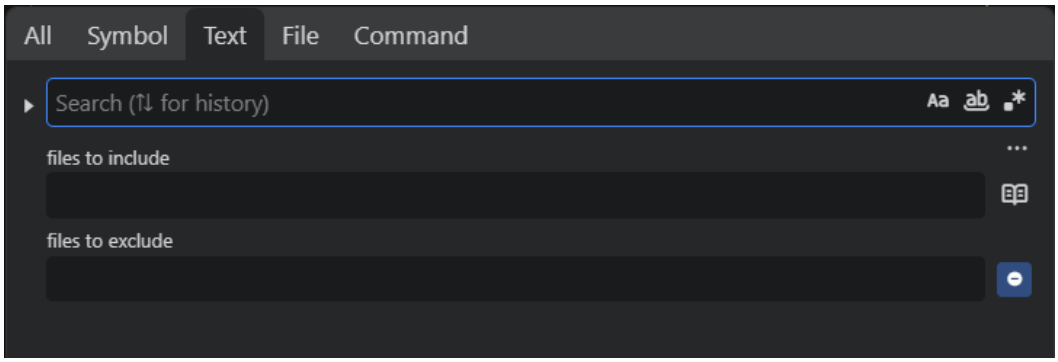
- 按“Ctrl+Shift+F”并输入搜索词。

  搜索结果被分组到包含搜索词的文件中，并指示每个文件中的匹配项及其位置。
- 展开文件可查看该文件中所有选中的预览，然后单击其中一个搜索结果可在编辑器中查看它。



高级搜索选项

您可以通过单击搜索字段下面的"**切换搜索详细信息**"按钮（ ）来提供高级搜索选项，并在**要包括的文件/要排除的文件**字段中输入要从搜索中包括或排除的模式。



如果您输入example，这将匹配工作区中每个名为example的文件夹和文件。如果您输入./example，这将匹配工作区顶层的文件夹example/。使用逗号（,）来分隔多个模式。路径必须使用正斜线。

您也可以使用glob语法来提供样式：

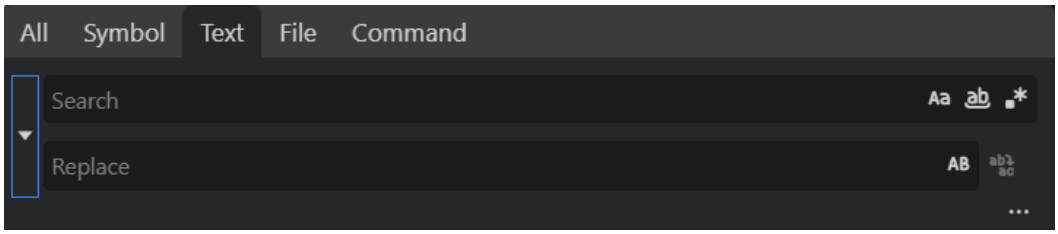
- *匹配一个路径段中的零个或多个字符。
- ?匹配一个路径段中的单个字符。
- **匹配任意数量的路径段，包括无。
- {}用于分组条件（例如，{**/*.html，**/*.txt}匹配所有HTML和文本文件）。
- []用于**声明**要匹配的字符范围（example.[0-9]匹配example.0，example.1，等等）。
- [!...]否定要匹配的字符范围（example.[!0-9]匹配example.a，example.b，但不匹配example.0）。

CodeArts IDE默认排除了一些文件夹，以减少您可能不感兴趣的搜索结果（例如，node_modules）。您可以打开设置，并在files.exclude和search.exclude部分改变这些默认选项。若要快速包含或排除被.gitignore文件忽略的文件或被files.exclude和search.exclude设置匹配的文件，请在**排除的文件**栏中单击**使用“排除设置”与“忽略文件”**按钮（  ）。搜索视图中的glob模式与files.exclude和search.exclude等设置中的工作方式不同。在设置中，glob模式总是相对于工作区文件夹的路径进行评估，您必须使用**/example来匹配子文件夹folder1/example中名为example的文件夹。在搜索视图中，**前缀是假定的。

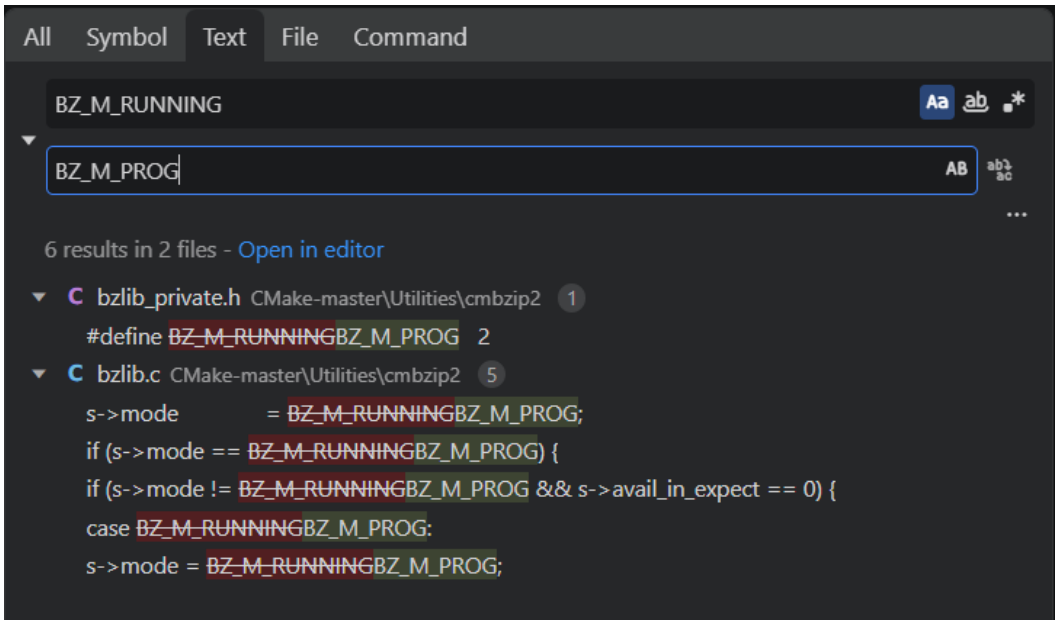
若您要将搜索范围限制在当前打开的文件，请在**包含的文件**栏中单击**仅在打开的编辑器中搜索**按钮（  ）。

查找和替换

您可以跨文件搜索和替换。若要显示替换字段，单击展开搜索部件，或按**“Ctrl+Shift+H”/“Ctrl+Shift+R”（IDEA键盘映射）**。



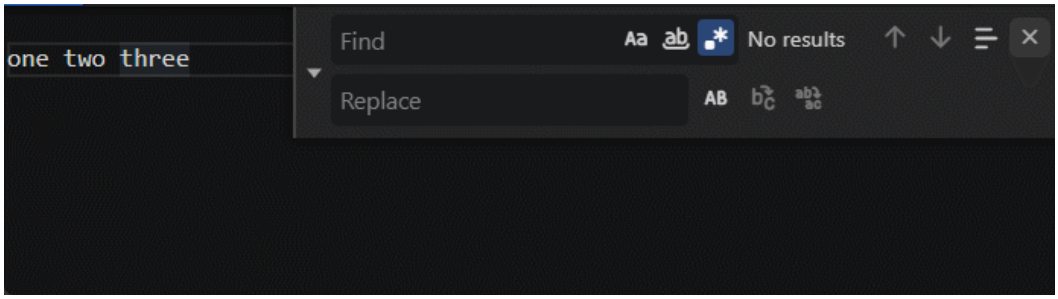
当您在**“替换”**字段中键入文本时，CodeArts IDE会显示一个待定修改的差异视图。您可以选择跨所有文件**替换**“Ctrl+Alt+Enter”、在一个文件中全部替换或单个替换。也可以通过使用 **Down** 和 **Up** 可在搜索词历史记录中进行导航，快速重用以前的搜索词。



3.1.3.3 搜索并替换为正则表达式

除了搜索和替换表达式外，您还可以通过将正则表达式与捕获组一起搜索和重用匹配的部分内容。通过单击“使用正则表达式”按钮（  ）或按**“Alt+R”**，在搜索框中启用正则表达式，然后编写正则表达式并使用括号定义组。

然后，您可以通过在替换字符串中使用$n引用每个组中匹配的内容，其中n是捕获组的顺序（例如，$1、$2等）。

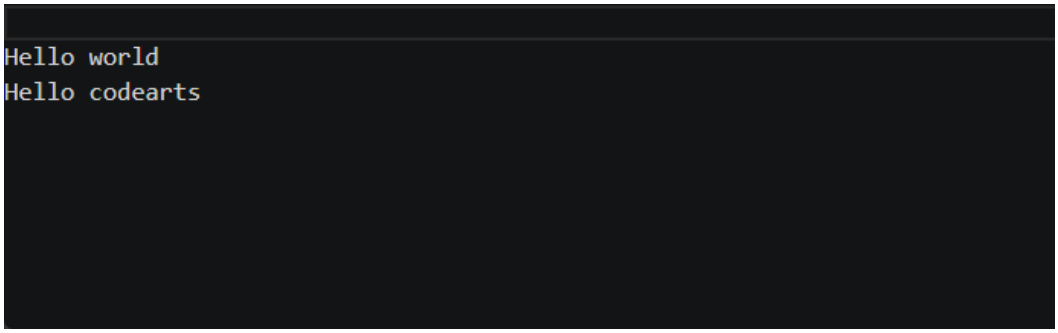


正则表达式替换中的大小写变化

CodeArts IDE支持在编辑器或全局中执行搜索和替换时改变正则表达式匹配组的大小写。

这可以通过使用修改符来实现：

- \u：大写单个字符。
- \U：将匹配组的其余部分大写。
- \l：小写单个字符。
- \L：将匹配组的其余部分小写。



修饰符可以堆叠，例如，\u\u\u$1将大写组的前三个字符，\l\U$1将小写第一个字符，大写其余字符。

3.1.4 格式化代码

3.1.4.1 概述

CodeArts IDE提供了许多代码格式化功能。编辑器有两个显式格式操作：

- 格式化文档（**“Shift+Alt+F” / “Ctrl+Alt+L” (IDEA键盘映射)**）-格式化整个活动文件。
- 设置选定内容的格式化（**“Ctrl+K Ctrl+F” / “Ctrl+Alt+L” (IDEA键盘映射)**）-设置选定文本的格式。

您可以从命令面板（**“Ctrl+Shift+P” / “Ctrl Ctrl”**）或编辑器上下文菜单调用这些操作。

CodeArts IDE具有对JavaScript、TypeScript、JSON、HTML和Java的默认格式化程序。每种语言都有特定的格式选项（例如，html.format.indentInnerHtml），您可以根据用户或工作区设置中的首选项进行调整。如果您安装了另一个为同一语言提供格式化的扩展，您也可以禁用默认语言格式化程序。

除了手动调用代码格式化外，您还可以根据用户操作触发格式化，如键入、保存或粘贴。默认情况下，这些功能处于关闭状态，但您可以通过以下设置启用这些功能：

- editor.formatOnType - 键入一行后自动格式化该行。
- editor.formatOnSave - 保存时格式化文件。
- editor.formatOnPaste - 格式化粘贴的内容。

除了默认的格式化工具外，您可以在插件市场上找到支持其他语言或格式化工具的扩展插件。

约束与限制

并非所有格式化程序都支持粘贴时的格式，它们必须支持格式化文本的选定内容。

3.1.4.2 缩进

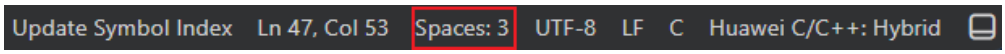
CodeArts IDE允许您使用空格或制表符来控制文本缩进。默认情况下，CodeArts IDE插入空格，并且每个“**Tab**”键使用4个空格。如果要使用其他默认值，您可以修改编辑器选项editor.insertSpaces和editor.tabSize设置。

"editor.insertSpaces": true,

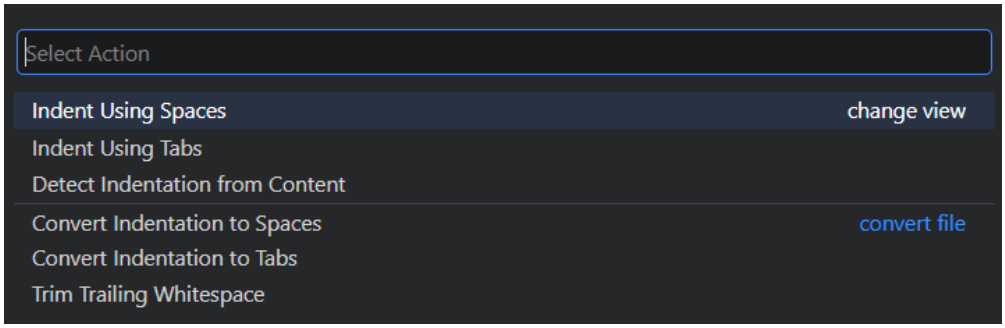
"editor.tabSize": 4

自动检测

CodeArts IDE会根据您所打开的文件来决定文档中使用的缩进。自动检测的缩进将覆盖默认的缩进设置。检测到的设置将显示在底部状态栏的右侧。



您可以通过单击状态栏中的缩进设置打开缩进命令列表，然后更改打开文件的默认缩进设置或在制表符和空格之间转换。

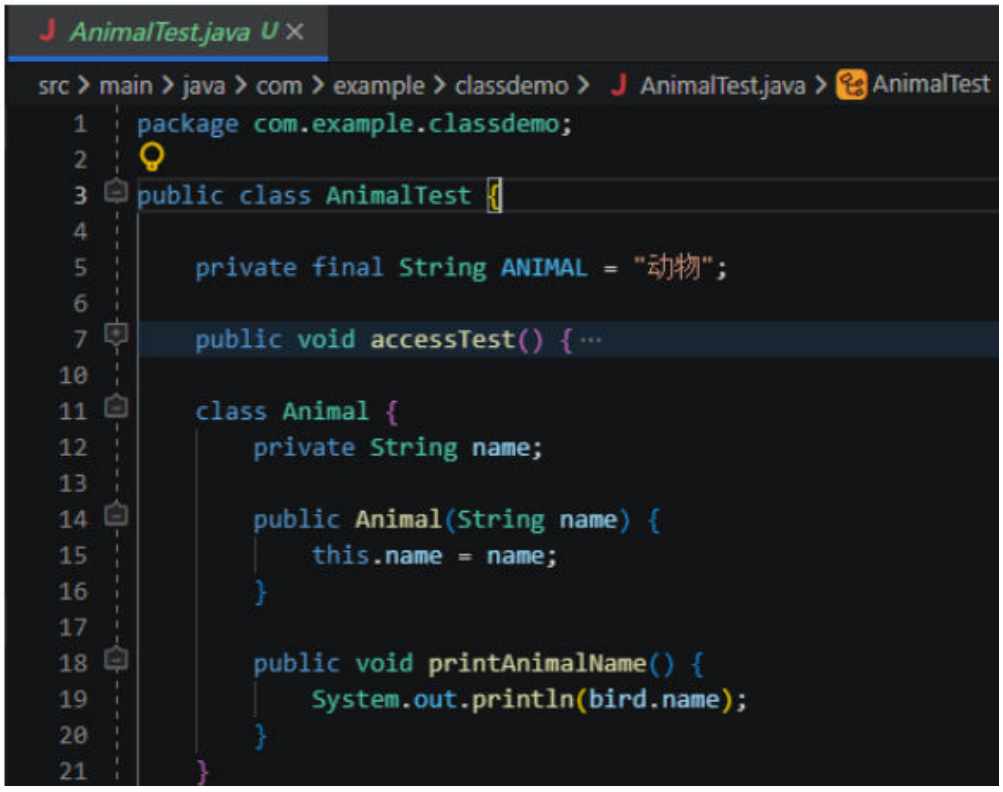


CodeArts IDE自动检测检查2、4、6或8空格的缩进。如果文件使用不同数量的空格，则可能无法正确检测缩进。例如，如果您需使用3个空格缩进，则您需关闭editor.detectIndentation选项，并将选项卡大小显式设置为3。

```
"editor.detectIndentation": false,
"editor.tabSize": 3,
```

3.1.5 折叠代码

您可以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。



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。

特定语言的折叠区域

折叠区域可以根据编辑器配置的语言的语法标记计算。以下语言已经提供了语法感知折叠：Markdown、HTML、CSS、LESS、SCSS和JSON。

如果您想为上述一种（或所有）语言切换回基于缩进的折叠，请使用：

```
"[html]": {
    "editor.foldingStrategy": "indentation"
},
```

区域也可以由每种语言定义的标记定义。以下语言当前定义了标记：

Language	Start region	End region
Bat	::#region or REM #region	::#endregion or REM #endregion
C#	#region	#endregion
C/C++	#pragma region	#pragma endregion
CSS/Less/SCSS	/*#region*/	/*#endregion*/
Coffeescript	#region	#endregion
F#	//#region or (#region)	//#endregion or (#endregion)
Java	//#region or //<editor-fold>	// #endregion or //</editor-fold>
Markdown	<!-- #region -->	<!-- #endregion -->
Perl5	#region or =pod	#endregion or =cut
PHP	#region	#endregion
PowerShell	#region	#endregion
Python	#region or # region	#endregion or # endregion
TypeScript/JavaScript	//#region	//#endregion
Visual Basic	#Region	#End Region

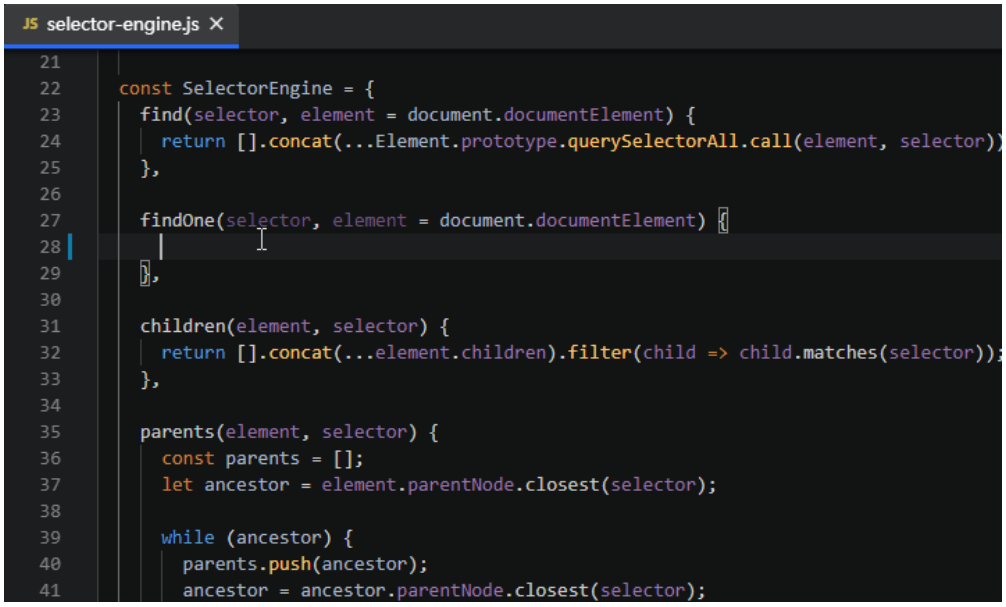
仅折叠和展开标记定义的区域，请您使用以下操作：

- 折叠所有区域（“**Ctrl+K Ctrl+8**”）折叠所有标记区域。
- 展开所有区域（“**Ctrl+K Ctrl+9**”）展开所有标记区域。

3.2 代码补全

3.2.1 编程语言的代码补全

智能代码补全是说各种代码编辑功能的总称，包括：代码补全、参数信息、快速信息和成员列表。代码补全功能有时被称为“内容辅助”或“代码提示”。



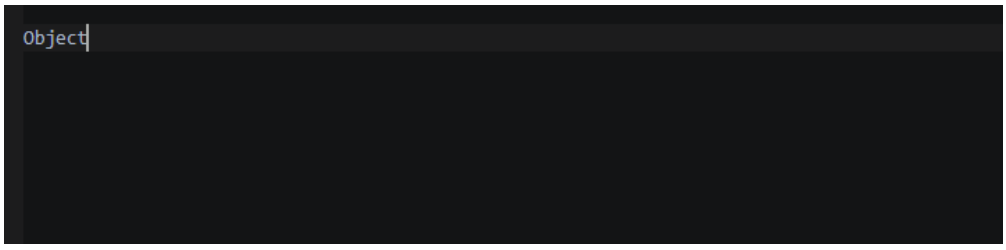
CodeArts IDE为JavaScript、TypeScript、JSON、HTML、CSS、SCSS和Less编程语言提供代码补全。CodeArts IDE支持任何编程语言的基于单词的补全，但也可以通过安装语言扩展来提供更丰富的代码补全。

3.2.2 代码补全功能

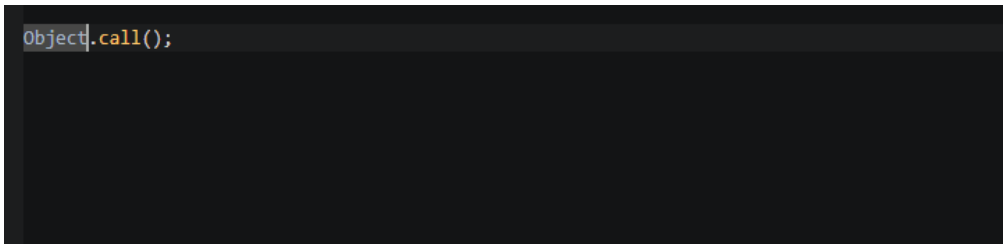
3.2.2.1 概述

CodeArts IDE代码补全功能由语言服务提供支持，该服务基于语言语义和代码分析提供智能代码补全功能。如果语言服务识别，则在您键入代码时自动弹出补全推荐列表。如果继续键入字符，则将过滤符号列表（变量、方法等），直到包含键入字符的符号列表。补全推荐小部件还支持驼峰过滤：如要限制补全推荐的代码数量，请在符号名称中键入大写字母。例如，cra将快速调出createApplication。

- 要手动触发代码补全，请按“**Ctrl+Shift+Space**”（**IDEA键盘映射**）或键入触发字符（如JavaScript中的点字符(.)）。
- 要插入选定的符号，请按“**Enter**”键。



- 要插入选定的符号并替换当前位于光标位置的符号，请按“**Tab**”键。

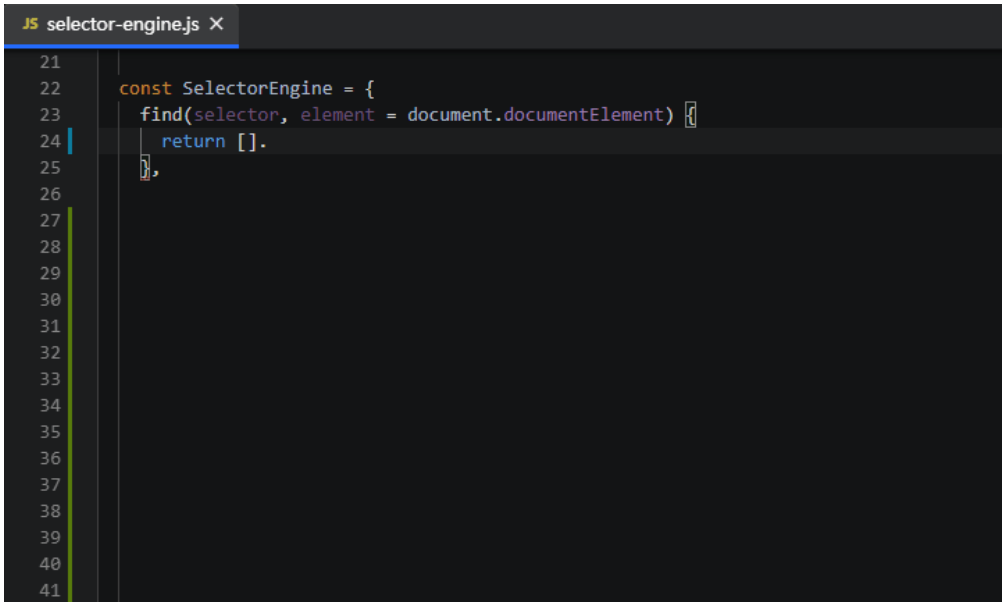


- 要关闭推荐列表而不插入推荐代码，请按**退出“Escape”**键。

3.2.2.2 快速信息

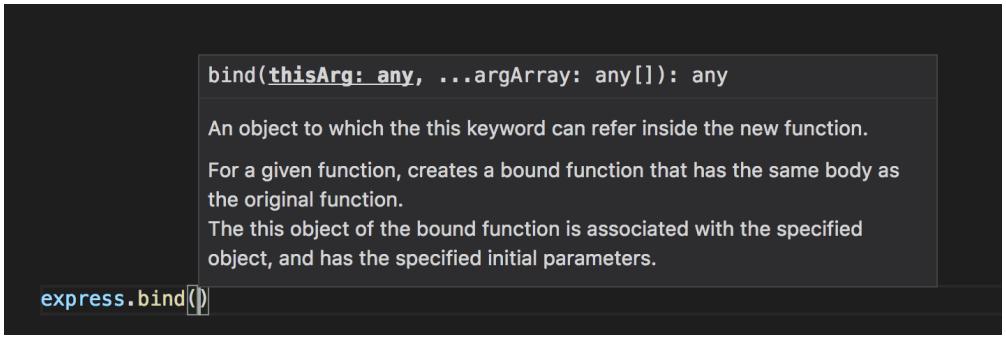
依靠语言服务提供的功能，您可以通过在推荐列表中再次按**“Ctrl+I”** / **“Ctrl+Space”** / **“Ctrl+Shift+Space”**（**IDEA键盘映射**）来查看方法的快速信息。该方法的文档弹出窗口将扩展到侧面，并在您导航列表时自动更新。您还可以通过按**“Ctrl+K Ctrl+I”** / **“Ctrl+Q” (IDEA键盘映射)**打开插入符号处任何符号的快速信息弹窗。

要隐藏快速信息的弹窗，请再次按**“Ctrl+空格”**键或单击**关闭**图标。



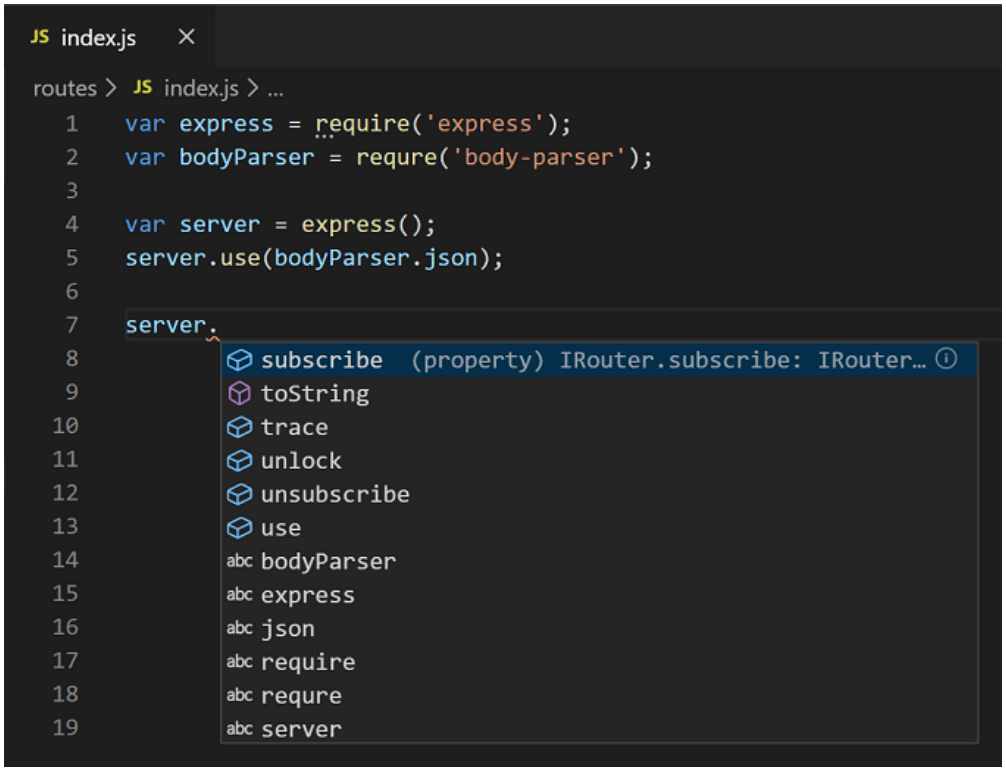
3.2.2.3 参数信息

在编辑区选择方法后，CodeArts IDE将显示参数信息。如果适用，语言服务将在快速信息和方法签名中显示基础类型。需要随时打开参数信息弹窗，请按**“Ctrl+P”**。



3.2.3 补全类型

代码补全可以提供补全推荐和项目的全局标识符。在代码推荐列表中，首先显示推荐的符号，然后是全局标识符（由Word图标显示）。



CodeArts IDE代码补全包括基于语法语义的补全、片段推荐补全和简单的基于单词的文本补全。SmartAssist套件补全推荐的代码会额外标记为⊕图标。

图标	描述
	Methods and Functions
	Variables
	Fields
	Type parameters
	Constants
	Classes
	Interfaces
	Structures
	Events
	Operators

图标	描述
	Modules
	Properties and Attributes
	Values and Enumerations
	References
	Keywords
	Files
	Folders
	Colors
	Unit
	Snippet prefixes
abc	Words
	SmartAssist suggestions

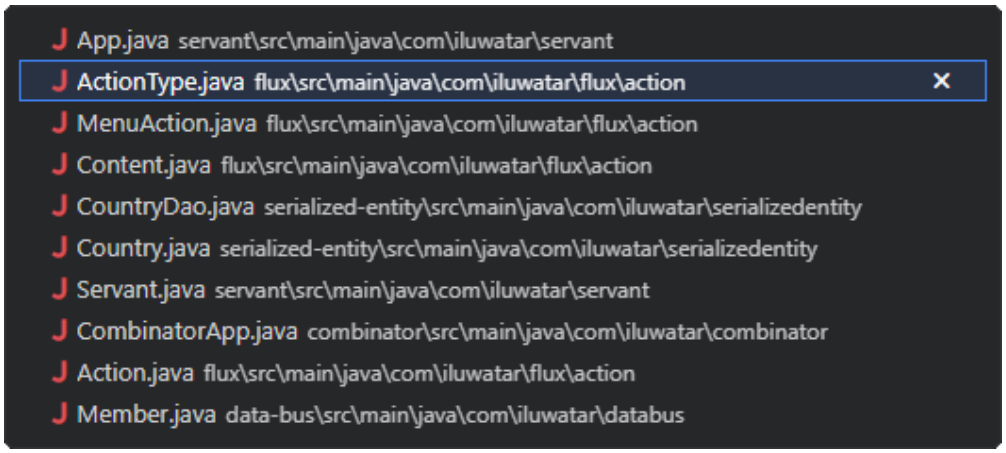
3.3 代码导航

3.3.1 快速文件导航

当浏览项目时，资源管理器支持您在文件之间导航。但是，当您处理任务时，您会发现自己在同一组文件之间快速跳转。CodeArts IDE提供了功能强大的命令，可以通过易于使用的键绑定在文件中导航和跨文件导航。

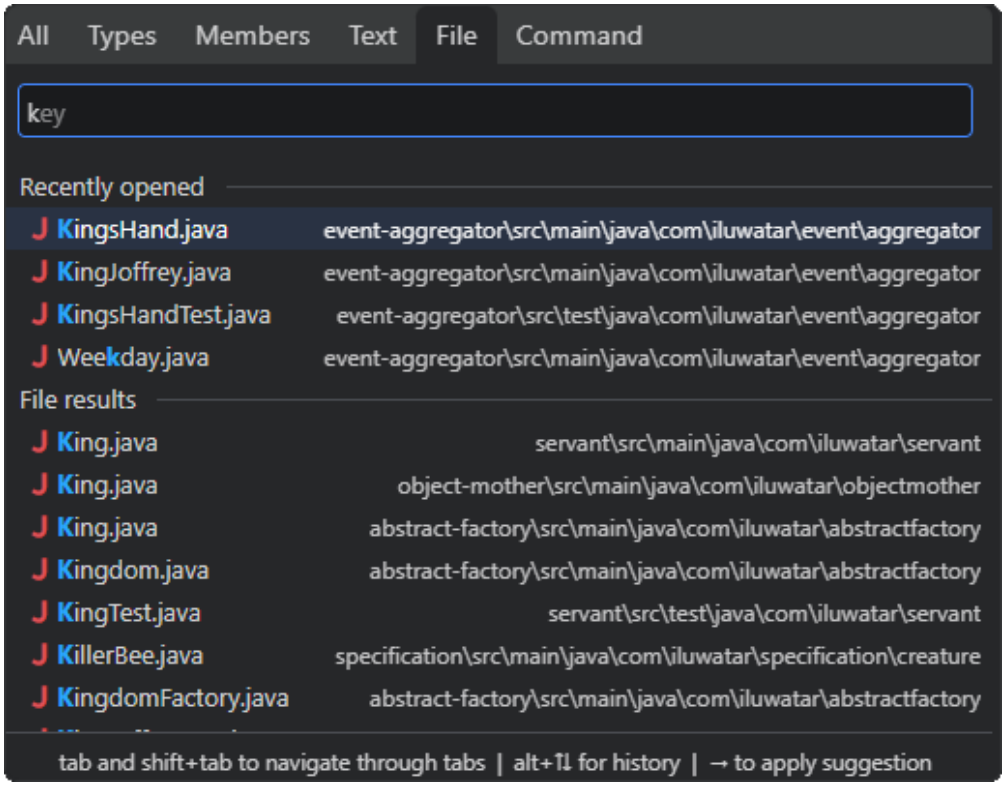
文件切换器

按**Ctrl+Tab**键打开文件切换器，列出**编辑器组**中当前打开的所有文件，并按“**Tab**”键循环文件。然后松开“**Ctrl**”**键**切换到选定的文件。



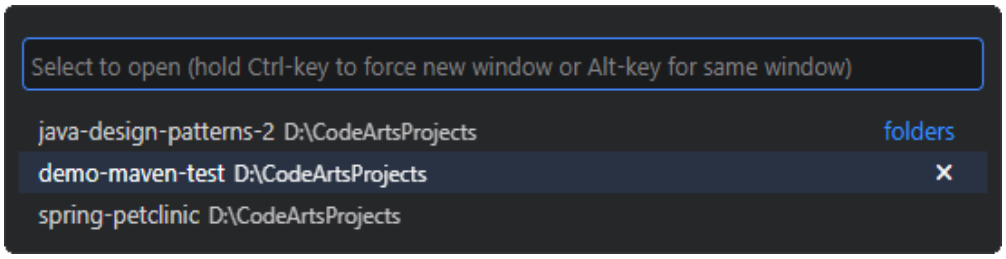
在最近打开的文件之间导航

按“**Ctrl+Shift+N**”打开SmartSearch窗口的“文件”选项卡，其中列出了最近打开的文件。您可以使用**向上**和**向下**光标键选择所需的文件，然后按“**Enter**”键打开它。要缩小列表范围，请键入要打开的文件的名称。



在最近打开的文件夹之间导航

按“**Ctrl+E**”，或在主菜单中选择“文件”>“打开最近的文件”，查看最近打开的文件夹列表。使用**向上**和**向下**光标键选择所需的文件夹，然后按“**Enter**”键打开该文件夹。



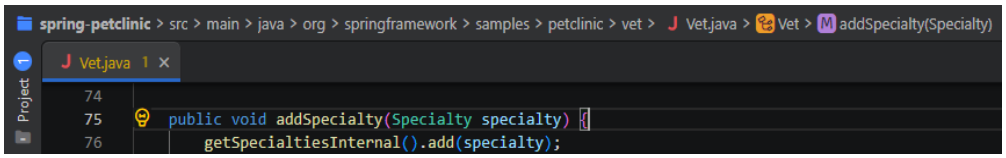
导航到文件的开头和结尾

按“**Ctrl+Home**”键导航到文件的开头，按“**Ctrl+End**”键导航到结尾。

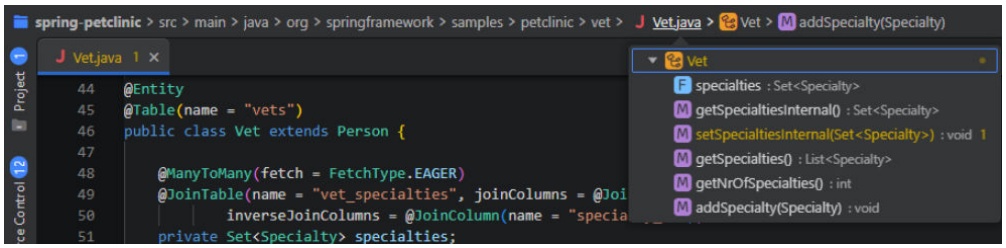
3.3.2 使用面包屑导航路径

编辑器内容上方有一个导航栏。它显示当前位置，并允许您在文件夹、文件和符号之间快速导航。您可以使用查看>显示导航路径主菜单或通过breadcrumbs.enabled **setting**，启用设置关闭面包屑。

导航栏始终显示文件路径，并在语言扩展的帮助下显示光标所在位置的符号路径。显示的符号与**大纲视图**和**转到编辑器中的符号**中的相同。



在路径中选择面包屑将显示一个包含该级别同级的列表，以便您可以快速导航到其他文件夹、文件和符号。



自定义面包屑

导航栏的外观可以自定义。如果显示路径很长，或者只对文件路径或符号路径感兴趣，则可以通过breadcrumbs.filePath和breadcrumbs.symbolPath设置项来配置。两者都支持on、off和last，它们定义了您是否能看到路径或看到哪一部分的路径。默认情况下，导航痕迹在导航栏的左侧显示文件和符号图标，但您可以通过将breadcrumbs.icons设置为false来删除图标。

您可以通过breadcrumbs.symbolSortOrder设置"控制“导航路径”大纲视图中符号的排序方式。

支持的排序方式为：

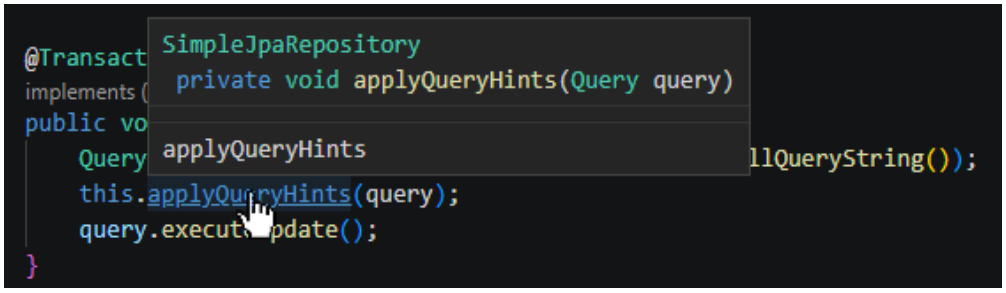
- position：以文件位置顺序显示符号大纲（默认）。
- name：以字母顺序显示符号大纲。
- type：以符号类型显示符号大纲。

面包屑键盘导航

要与面包屑进行交互，请按“**Ctrl+Shift**”+.。这将选择该元素并打开一个列表，允许您导航到兄弟文件或符号。使用“**Ctrl+Left**” / “**Ctrl+Right**” / “**Right**” 键盘快捷键，以前往当前元素之前或之后的元素。当列表打开时，开始键入元素的名称以快速选择进行导航。

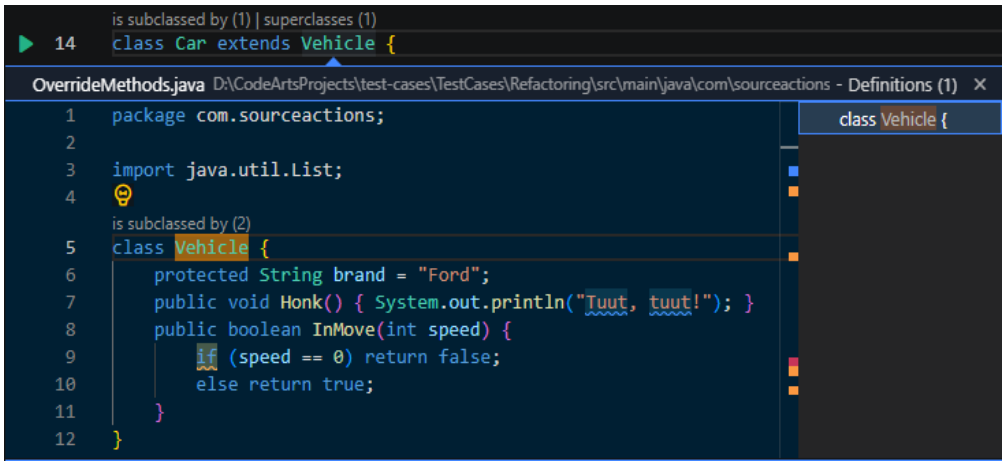
3.3.3 转到定义

语言服务加载成功后，可以通过按“**F12**”或在主菜单中选择**导航>转到定义**来转到符号的定义。如果按“**Ctrl**”键并将鼠标悬停在符号上，将显示声明的预览。



您可以使用“**Ctrl+单击**”跳转到定义，也可以使用“**Ctrl+Alt+单击**”将定义打开在侧边打开。

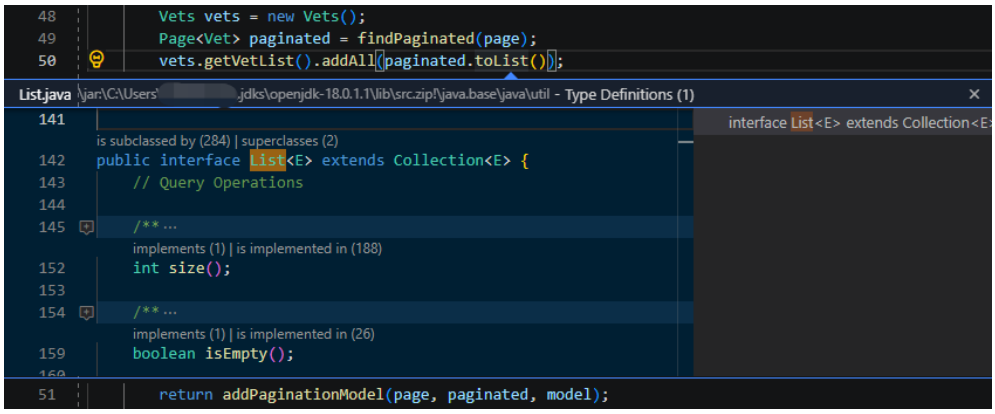
还可以通过Peek视图使用此功能，该视图显示在当前编辑器中，因此您无需切换上下文。在Peek视图中查看符号的定义，右键单击符号，然后在上下文菜单中选择**Peek>Peek定义**。



3.3.4 转到类型定义

某些语言服务还支持通过从编辑器上下文菜单或**命令面板**运行**转到类型定义**命令，或按“**Ctrl+Shift+B**”跳转到符号的类型声明。

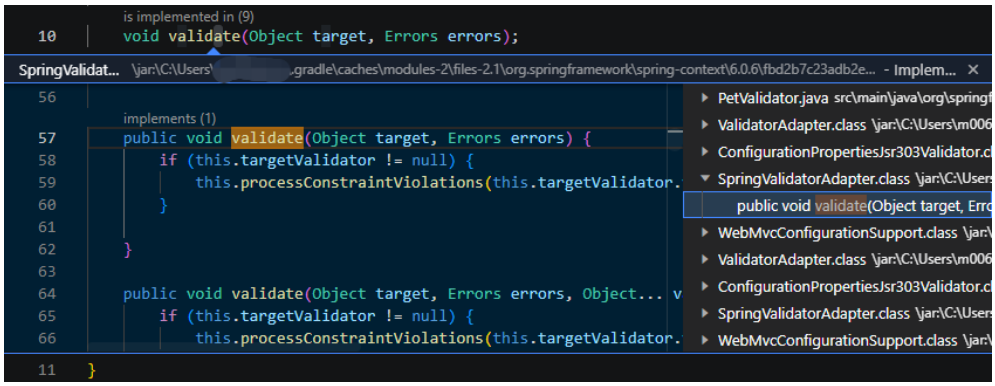
还可以通过Peek视图使用此功能，该视图显示在当前编辑器中，因此您不需要切换上下文。要在Peek视图中查看符号类型的定义，右键单击符号，然后在上下文菜单中选择Peek>Peek类型定义。



3.3.5 转到实现

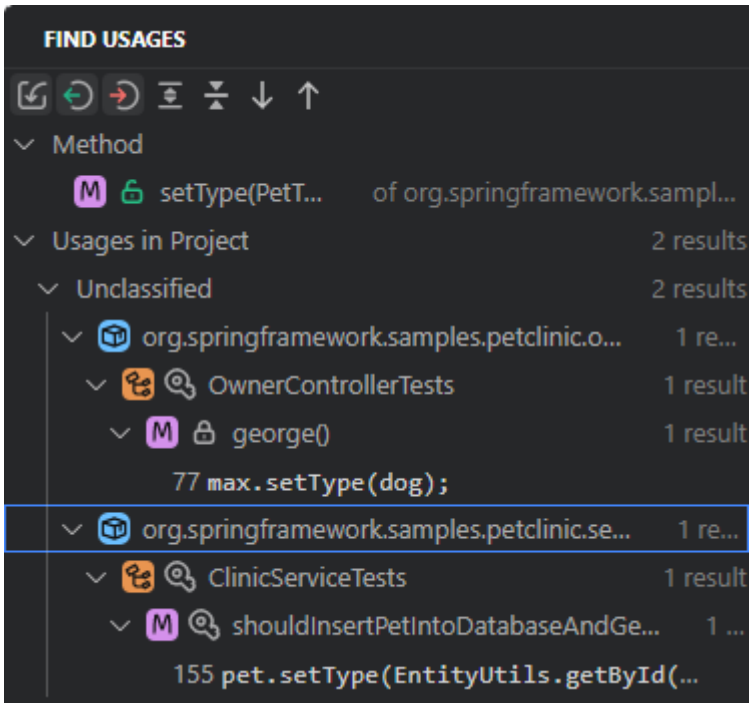
语言服务还支持通过按“**Ctrl+Alt+B**”跳转到符号的实现。对于接口，这显示了该接口的所有实现者，对于抽象方法，这显示了该方法的所有具体实现。

还可以通过Peek视图使用此功能，该视图显示在当前编辑器中，因此您不需要切换上下文。要在Peek视图中查看方法的实现，右键单击符号，然后在上下文菜单中选择Peek>Peek类型定义。



3.3.6 查找所有引用

选择一个符号，然后按“**Alt+F7**”在“**查找用法**”视图中打开对它的所有引用。

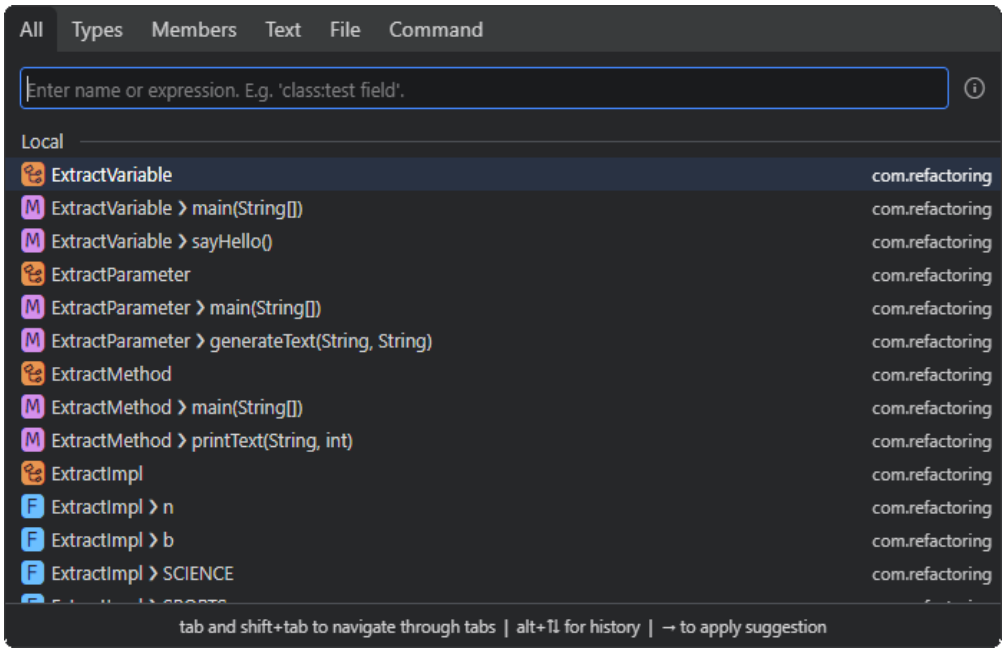


使用“查找用法”视图的工具栏按钮可以隐藏或显示导入语句（）、读取访问权限（）和写入访问权限（）中的符号引用。

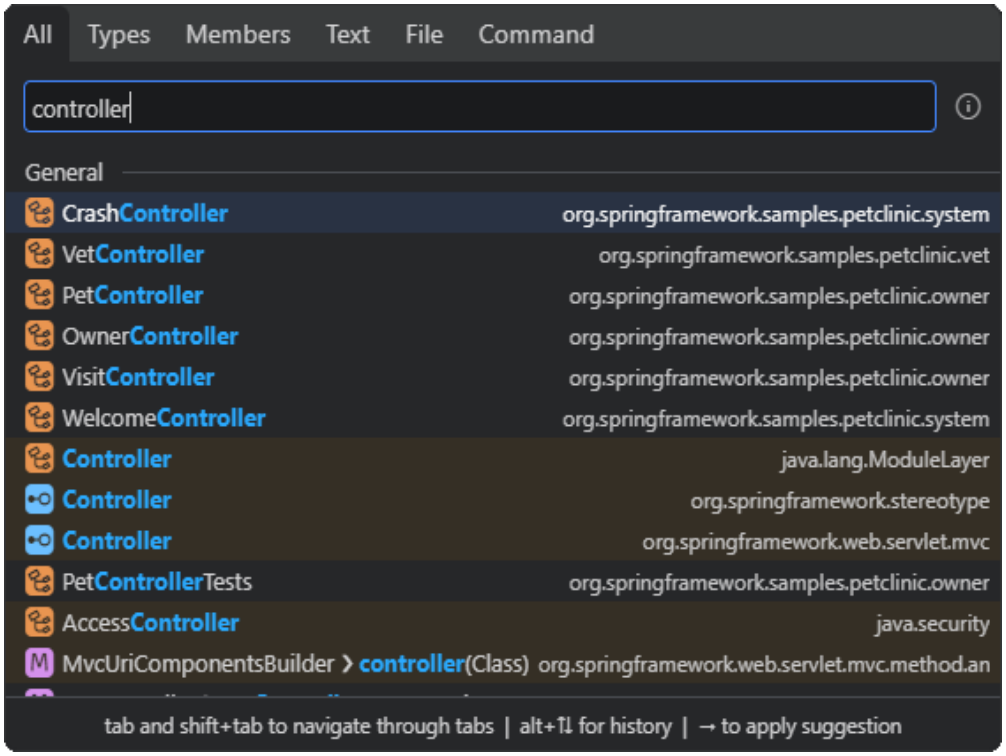
还可以通过Peek视图使用此功能，该视图显示在当前编辑器中，因此您不需要切换上下文。要在Peek视图中查看所有引用，右键单击符号，然后在上下文菜单中选择Peek>Peek类型定义。

3.3.7 转到编辑器中的符号

您可以使用“**Ctrl+Shift+O**”/“**Ctrl+Shift+Alt+N**”（**IDEA键盘映射**）/“**Ctrl+F12**”（**IDEA键盘映射**）导航到文件中的符号。按**向上**或**向下**箭头键导航到所需的位置。



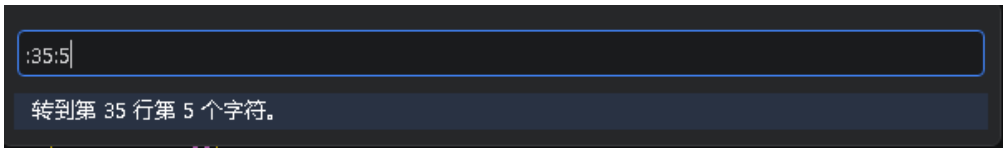
要导航到项目中的符号，请按“**Ctrl+T**”，开始键入符号的名称，然后使用**向上**或**向下**箭头键导航到它。



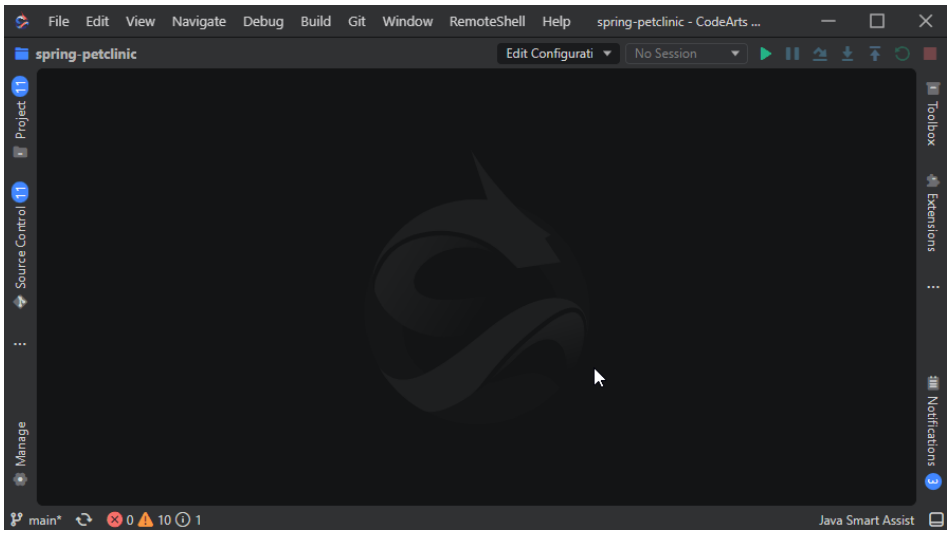
3.3.8 转到行

您可以跳转到当前打开的文件中的特定行，也可以跳转到任意文件中的特定行。

- 要跳到当前打开的文件中的一行，请按Ctrl+G，然后在打开的弹出窗口中键入所需的行。要额外指定要转至的列，请将其附加在冒号：之后，以便整个查询格式为：行：字符。

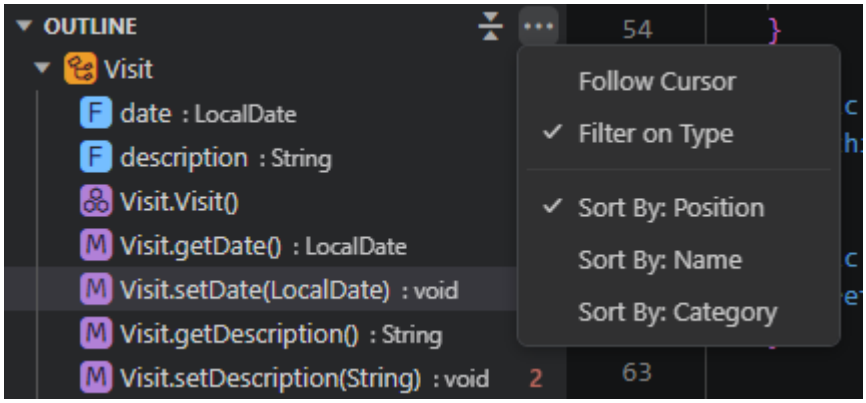


- 要跳转到任意文件中的行，请执行以下操作：
  a. 请按“**Ctrl+G**”。
  b. 在打开的弹窗中，删除前导冒号：，然后开始键入所需的文件名。
  c. 键入冒号：后跟所需的行。要额外指定要转至的列，请将其附加在第二个冒号：之后，以便整个查询格式为文件名：行：字符。
  d. 使用向上和向下光标键选择所需的文件，然后按“Enter”键打开文件。



3.3.9 大纲视图

大纲视图展开时，它将显示当前活动编辑器的符号树。大纲视图提供了几种排序模式、可选的光标跟踪，并允许您在键入时查找或筛选符号。错误和警告（如果有）将显示在受影响符号旁边的“大纲”视图中。



以下几项“大纲”视图设置，允许您启用/禁用图标并控制错误和警告的显示（默认情况下都启用）：

- outline.icons： 显示大纲图标。
- outline.problems.enabled： 显示大纲元素上的错误和警告。

- outline.problems.badges： 对错误和警告使用徽章。
- outline.problems.colors： 对错误和警告添加颜色。

3.3.10 括号匹配

当光标靠近其中一个括号，匹配的括号就会高亮显示。您可以使用“**Ctrl+Shift+**”跳转到匹配的括号。

```
private static String initProperty(String key) {
    String v = System.getProperty(key);
    if (v == null) {
        throw new InternalError("null property: " + key);
    }
    return v;
}
```

3.3.11 错误和警告

要循环浏览当前文件中的错误或警告，请按“F2”或“Shift+F2”，这将显示一个内联区域，详细说明问题和可能的代码操作（如果可用）。

```
5   class PrintAction implements Runnable {
Refactoring.java 1 of 3 problems
Class 'PrintAction' is never used
6       private List<String> data;
7
8       public PrintAction(List<String> data) {
9           this.data = data;
10      }
```

须知

有关CodeArts IDE中代码验证的更多详细信息，请参见**代码校验**。

3.4 代码校验

3.4.1 简介

编写代码时，CodeArts IDE会根据预定义的验证规则自动在后台分析代码。CodeArts IDE可以发现各种问题，识别可能的错误、拼写问题等。这有助于您在运行代码之前检测和更正代码中的问题。对于许多问题，CodeArts IDE提供了快速修复功能，让您可以迅速修复它们。

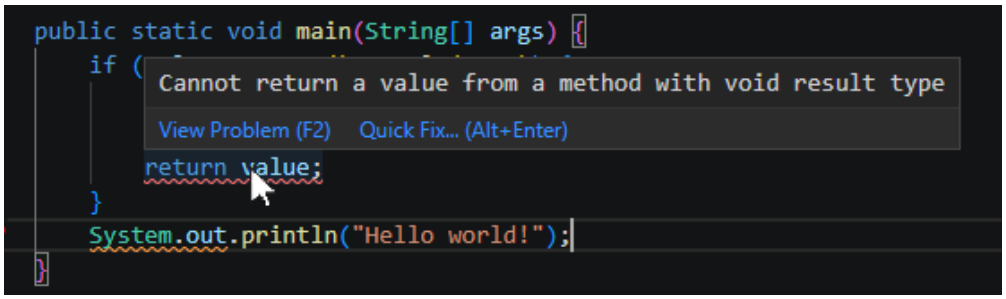
警告和错误显示在CodeArts IDE用户界面的几个位置：

- 状态栏显示所有错误和警告的摘要。

- “问题”视图列出了当前打开的文件中的所有问题。
- 打开文件时，所有错误和警告都将直接在代码编辑器中呈现，与文本内联，并在概述标尺中呈现。

3.4.2 在代码编辑器中查看问题

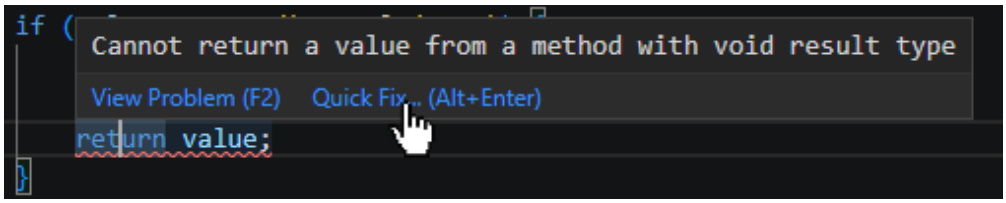
CodeArts IDE自动高亮显示代码中检测到的所有问题。要查看问题详细信息，请将鼠标悬停在代码编辑器中高亮显示的位置。



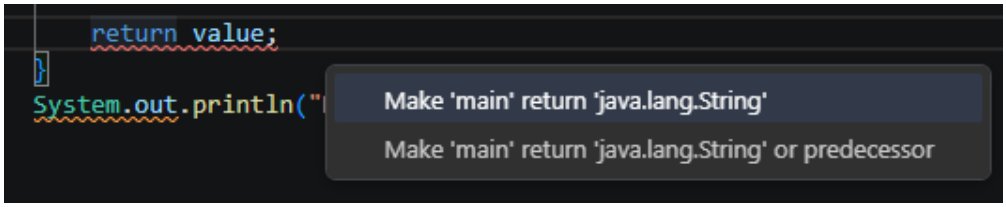
应用快速修复

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。

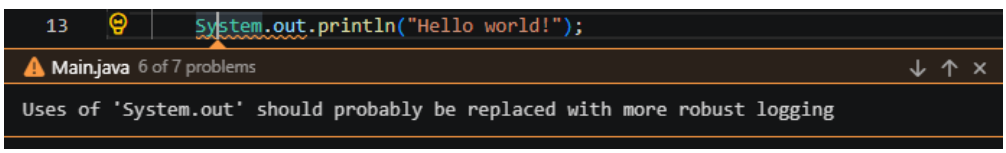


2. 在弹出菜单中，选择所需的快速修复。



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。

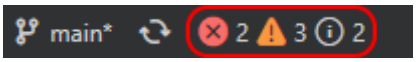


- 在快速查看器中，单击“转到下一个问题”（↓）和“转到上一个问题”（↑），或按**“Alt+F8” / “F2”(IDEA键盘映射) / “Shift+Alt+F8” / “Shift+F2”(IDEA键盘映射)**在当前文件中的错误之间跳转。

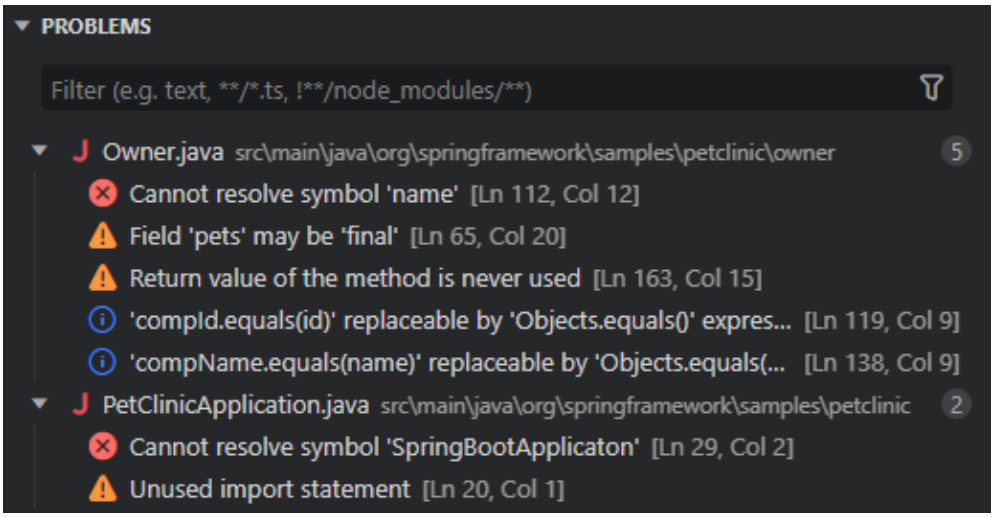
3.4.3 使用“问题”视图

“问题”视图列出了当前打开的文件中的所有问题。要打开它，请执行以下任何操作：

1. 单击CodeArts IDE状态栏中的问题摘要。



2. 在主菜单中，选择“查看>问题”。
3. 按**“Ctrl+Shift+M”** / **“Shift+Escape”**（**IDEA键盘映射**）/ **“Alt+0”**（**IDEA键盘映射**）。

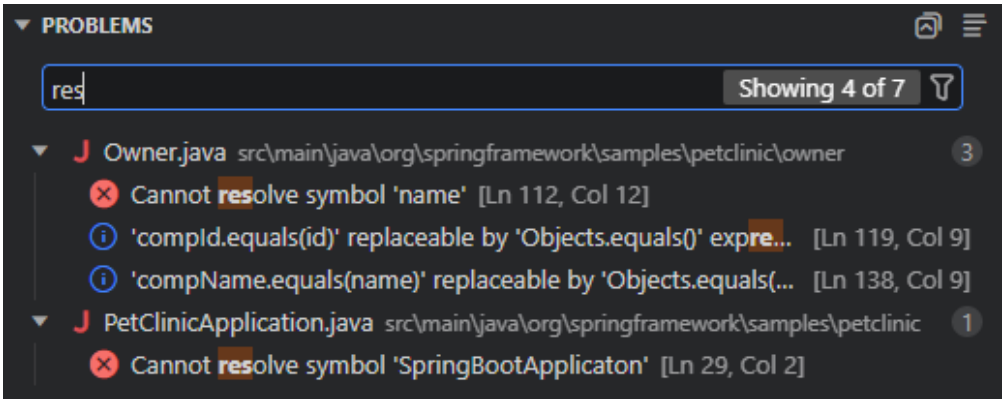


默认情况下，**“问题”**视图显示按其所属文件分组的问题。要在树视图和平面表视图之间切换，请单击“问题”视图标题栏中的**“作为表查看”**（ ）/**“作为树查看”**（ ）按钮。

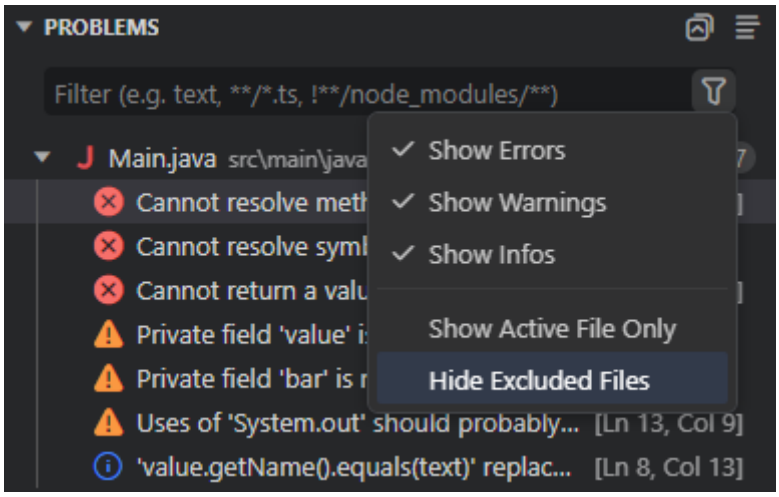
过滤问题列表

要限制显示的问题数量，您可以过滤列表。隐藏排除的文件筛选器可通过file.exclude setting设置配置为排除的文件。

- 要按任意查询进行筛选，请在“筛选器”字段中键入该查询。支持**/*.java等通配符模式。



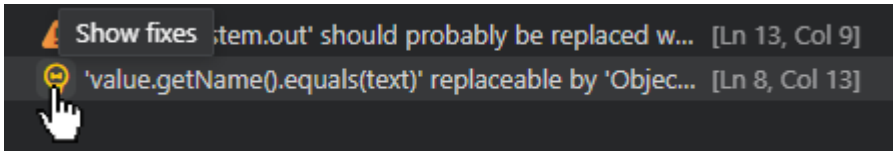
- 要使用预定义的过滤器，请单击“过滤器”字段中的“过滤器”按钮（ ），然后在弹出菜单中选择所需的过滤器。



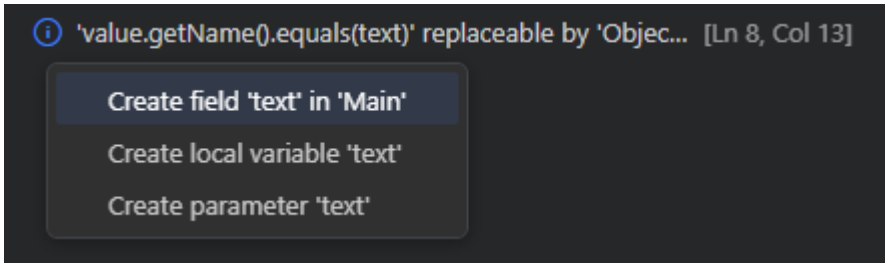
应用快速修复

如果检测到的问题有快速修复可用，您可以直接从“问题”视图应用它们。

步骤1 鼠标悬停到问题图标，使其更改为，然后单击它。



步骤2 在弹出式菜单中，选择要应用的快速修复项。

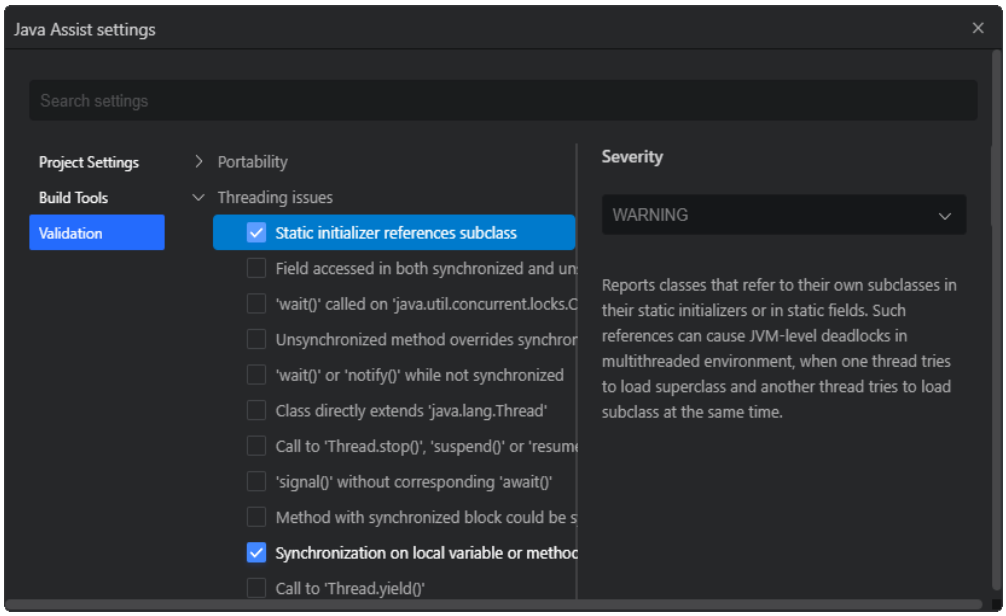


----结束

3.4.4 配置校验规则

您可以自定义应用于代码的验证规则集。

1. 通过执行以下任一操作打开**“Java助手设置”**对话框。
   - 单击CodeArts IDE状态栏中的**“Java智能助手”**。
   - 在“命令选项板”中运行**“SmartAssist: Open Settings”**命令（“Ctrl+Shift+P”）。
2. 使用搜索字段快速定位验证规则。然后配置如下：
   - 要启用或禁用规则，请使用其名称旁边的复选框。
   - 要调整相应代码问题在代码编辑器中突出显示的方式，请在**“严重性”**列表中选择所需的严重性级别。



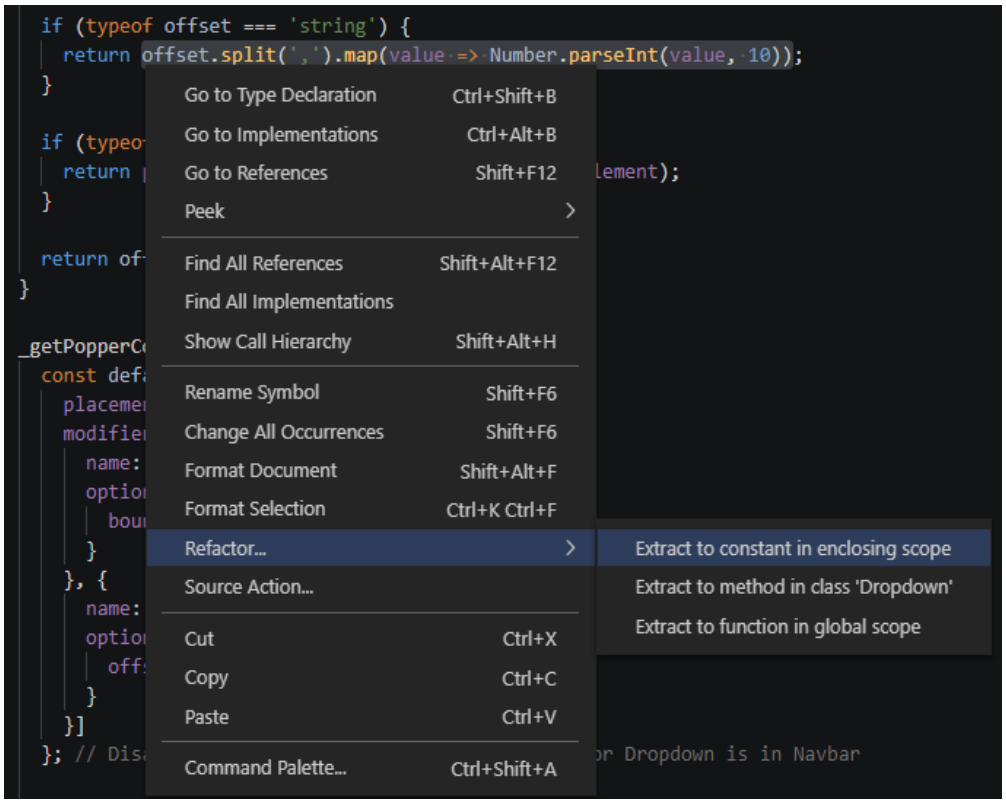
须知

有关**配置Java项目**的更多详细信息，请参见配置项目。

3.5 重构

3.5.1 简介

代码重构可以通过重构代码而不修改其运行行为来提高项目的质量和可维护性。CodeArts IDE支持重构操作（重构），以在编辑器中改进代码库。



例如，用于避免重复代码的常见重构是提取方法重构，在这种重构中，您可以将希望重用的代码拉到其自己的共享方法中。

重构由语言服务提供，CodeArts IDE内置了对TypeScript、JavaScript和Java的重构支持。

3.5.2 代码操作

在CodeArts IDE中，代码操作可以为检测到的问题提供重构和快速修复（以绿色曲线突出显示）。如果代码操作可用，则当光标位于曲线或选定文本区域上时，灯泡图标将显示在代码附近。单击代码操作灯泡图标或使用**快速修复**命令“**Alt+Enter**”将显示快速修复和重构建议。如果您只想查看没有快速修复的重构，请使用**重构**命令。

约束与限制

要禁用代码编辑器中的代码操作灯泡图标，请调整editor.lightbulb.enable设置。您仍然可以通过快速修复命令和“**Alt+Enter**”键盘快捷键打开快速修复。

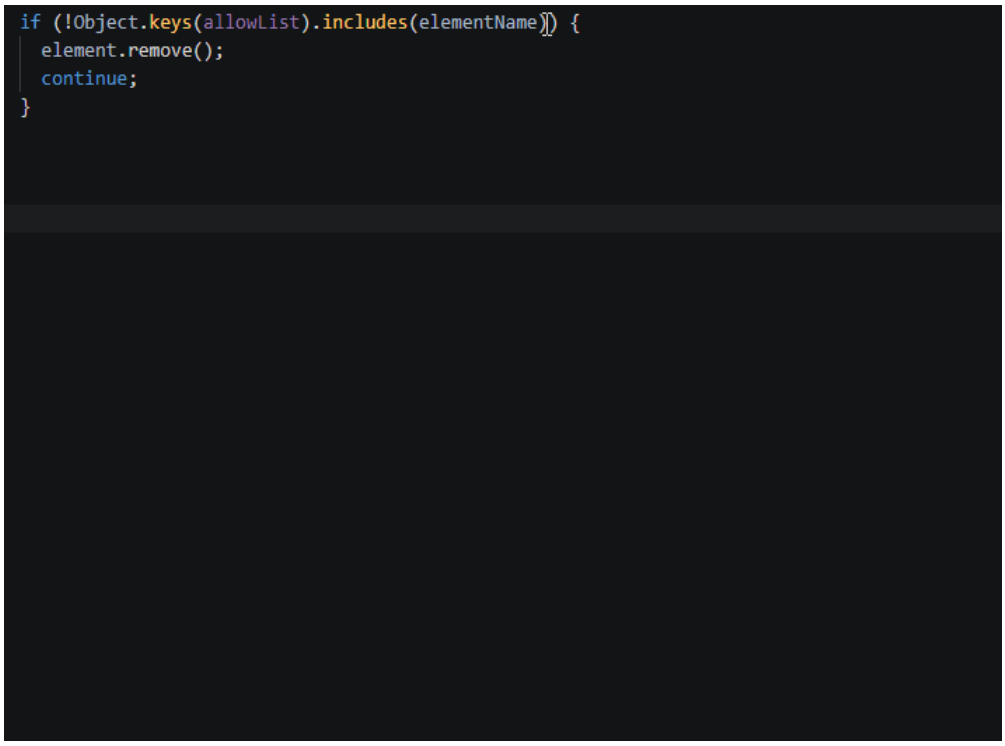
3.5.3 重构操作

3.5.3.1 提取方法

选择要提取的代码，然后单击装订线中的灯泡图标，或按“**Alt+Enter**”键查看可用的重构。源代码片段可以提取到新方法中，也可以提取到不同范围的新函数中。在提取重构期间，系统将提示您提供有意义的名称。

3.5.3.2 提取变量

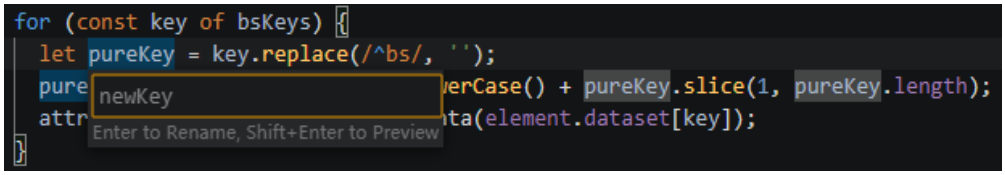
TypeScript语言服务提供**Extract to constant** 重构，为当前选定的表达式创建新的局部变量：



使用类时，还可以将值提取到新属性中。

3.5.4 重命名符号

重命名是与重构源代码相关的常见操作，CodeArts IDE有一个单独的**“重命名符号”**命令(“Shift+F6”)。某些语言支持跨文件重命名符号。按“Shift+F6”，键入新的所需名称，然后按Enter键。文件中符号的所有用法都将重命名。



3.5.5 代码操作的键绑定

editor.action.codeAction命令允许您为特定代码操作配置键绑定。例如，此键绑定触发提取函数重构代码操作：

```
{
  "key": "ctrl+shift+r ctrl+e",
  "command": "editor.action.codeAction",
  "args": {
    "kind": "refactor.extract.function"
  }
}
```

代码操作类型由扩展使用增强的CodeActionProvided API指定。种类是分层的，因此 "kind": "refactor" 将显示所有重构代码操作，而“ "kind": "refactor.extract.function"”将仅显示提取函数重构。

使用上述键绑定，如果只有单个“refactor.extract.function”代码操作可用，则将自动应用该代码操作。如果有多个提取函数代码操作可用，您可以从上下文菜单中选择所需的代码操作。

您还可以使用应用参数控制自动应用代码操作的方式和时间：

```
{
  "key": "ctrl+shift+r ctrl+e",
  "command": "editor.action.codeAction",
  "args": {
    "kind": "refactor.extract.function",
    "apply": "first"
  }
}
```

“apply”的有效值：

- first - 始终自动应用第一个可用的代码操作。
- ifSingle - 默认情况下。如果只有一个代码操作可用，则自动应用代码操作。否则，显示上下文菜单。
- never - 始终显示代码操作上下文菜单，即使只有一个代码操作可用。

当代码操作键绑定配置为 ’preferred": true时，仅显示首选的快速修复和重构。首选的快速修复解决了基础错误，而首选的重构是最常见的重构选择。例如，虽然可能存在多个refactor.extract.const重构，但每个重构都提取到文件中的不同作用域，但首选的refactor.extract.constant重构项是提取到局部变量的重构。

此键绑定使用 ’preferred": true创建始终尝试将选定源代码提取到本地作用域中的常量的重构：

```
{
  "key": "shift+ctrl+e",
  "command": "editor.action.codeAction",
  "args": {
    "kind": "refactor.extract.constant",
    "preferred": true,
    "apply": "ifSingle"
  }
}
```


4 C/C++

4.1 创建 C/C++工程

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

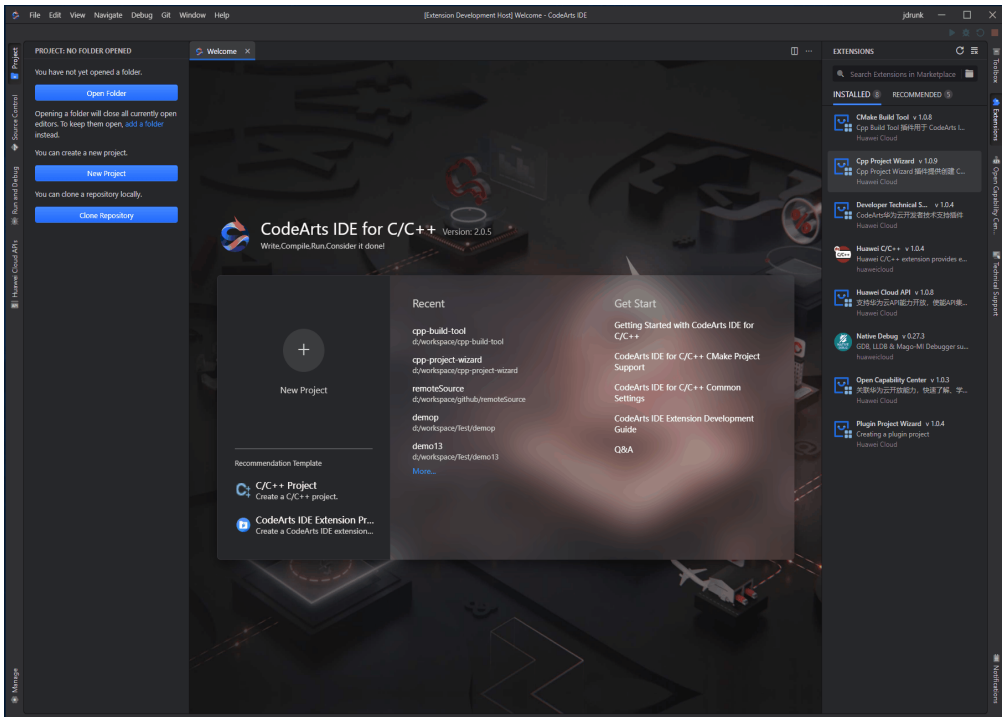
步骤1 单击顶部菜单 **File** ->**New** ->**Project...**。

步骤2 选择 C/C++。

步骤3 填写表单并单击创建按钮。

步骤4 等待工程创建完成并打开项目。

----结束



4.2 C/C++代码编写

4.2.1 编码基础操作

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，调用关系图等一些编码基础功能。

- **语法着色**

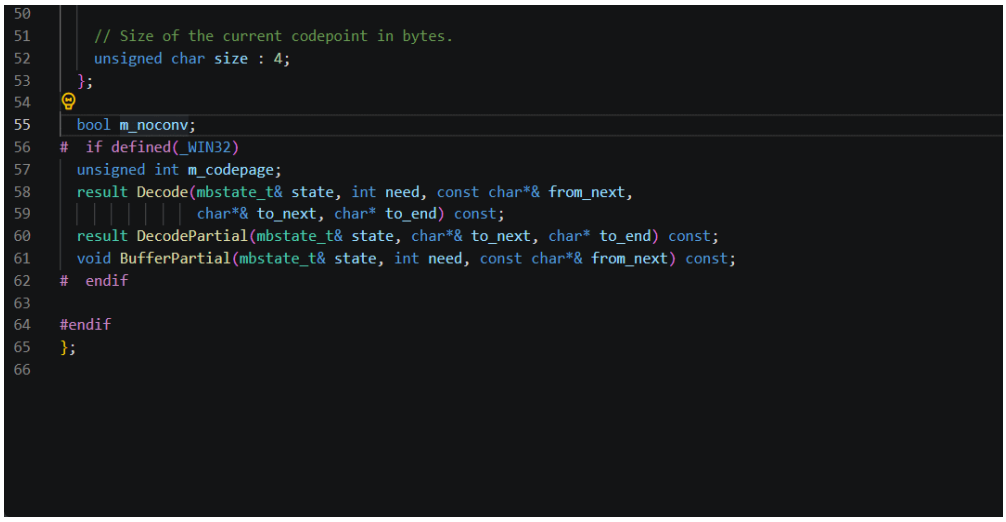
  该功能可对函数，类型，局部变量，全部变量，宏，枚举，成员变量等上色。

```
    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);
```

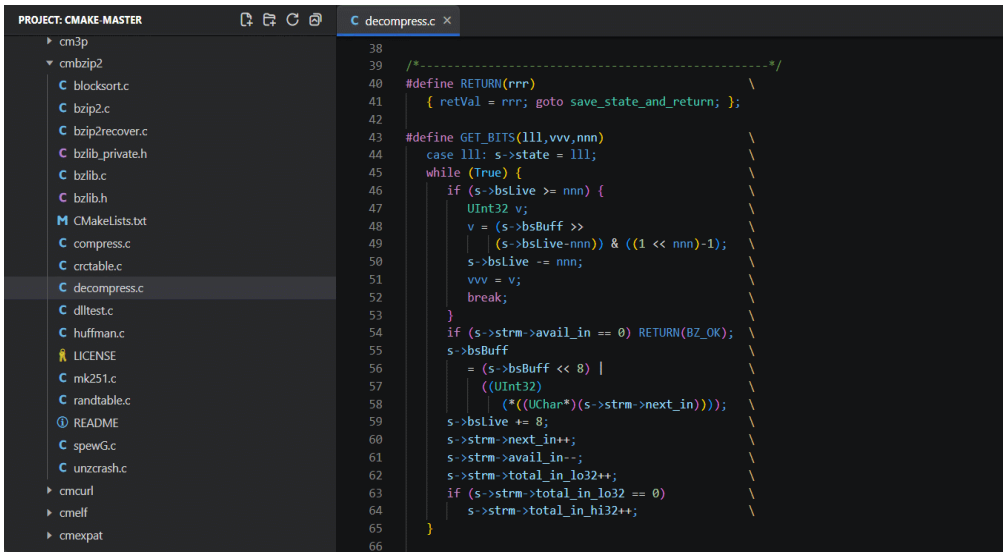
- **跳转定义** - 当光标放在代码处，“**Ctrl+单击**”或者“**F12**”跳转到定义，或者使用“**Ctrl+Alt+单击**”会打开定义到旁边。

```
for (left = ones - 1; left && first != last; --left) {
  c = (unsigned char)*first++;
  if (cm_utf8_ones[c] != 1) {
    return 0;
  }
  uc = (uc << 6) | (c & cm_utf8_mask[1]);
}

if (left > 0 || uc < cm_utf8_min[ones]) {
  return 0;
}

/* UTF-16 surrogate halves. */
if (0xD800 <= uc && uc <= 0xDFFF) {
  return 0;
}

/* Invalid codepoints. */
if (0x10FFFF < uc) {
  return 0;
}

*pc = uc;
return first;
```

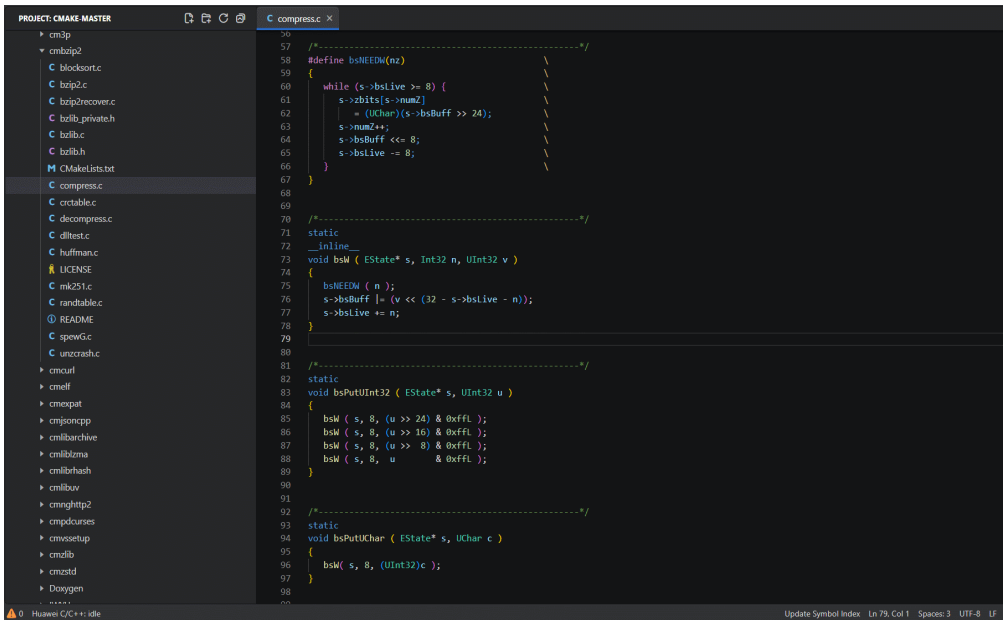
- **定义预览** - 当光标移至符号处，则会有符号定义的悬停预览，也可以用“**alt+F12**”的快捷键进行文件内的符号预览。



- **查找所有引用** - 当光标单击或者选择到需要查找的符号，**右键菜单->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。

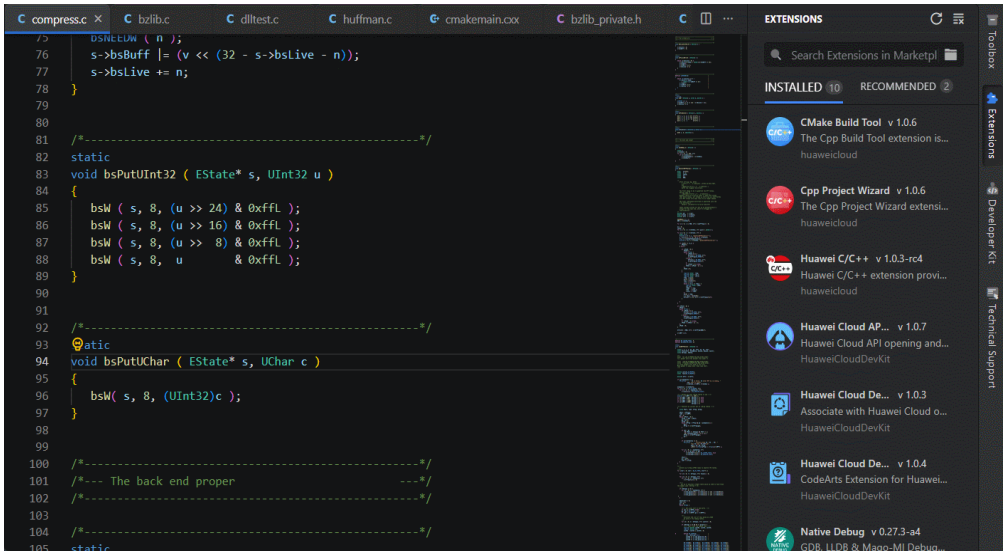


- **调用关系图** - 当光标单击或选中需要调用关系图的函数时，**右键菜单->调用关系图**，或可以使用快捷键“**Shift+Alt+H**”调出。在关系图中，也可以单击需要查看的函数并导航到该函数，同时也能够查看子类和基类。



- **符号大纲**

  左侧工具->右上角三个点->大纲即可打开符号大纲，或者使用快捷键“**Ctrl+Shift+B**”打开工具栏。打开大纲后，双击函数即可到达函数定义的位置，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）。



4.2.2 代码编写操作

CodeArts IDE for C/C++ 包含了内置的符号重命名，提取重构，代码补全/提示，实时语法检查等一些高级代码编写功能。

- **符号重命名（Rename symbol）**

  最基础的重构之一，但是变量或方法名字的可读性非常重要。在光标选中某个变量或方法后，右键单击以调出编辑器上下文菜单并且选择重命名符号或直接按“**F2**”，来重命名整个 C/C++ 项目中所有用到该命名的地方。

```
int maximum(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}

int main (){
    int a = 100;
    int b = 200;
    int ret;
    ret = maximum(a, b);
    printf( "Max value is : %d\n", ret );
    return 0;
}
```

- **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。

  可用的 C/C++ 重构类型包括：

  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。

```
void bubble_sort(int arr[], int len)
{
    int i, j, temp;
    for (i = 0; i < len - 1; i++)
        for (j = 0; j < len - 1 - i; j++)
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
}
int main() {
    int arr[] = { 22, 34, 3, 32, 82, 55, 89, 50, 37, 5, 64, 35, 9, 70 };
    int len = (int) sizeof(arr) / sizeof(*arr);
    bubble_sort(arr, len);
    int i;
    for (i = 0; i < len; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

  在选择**提取方法（Extract method）**重构后，输入提取的的方法/函数的名称。

  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。

```
    originalNumber = number;

    while (originalNumber != 0)
    {
        remainder = originalNumber%10;
        result += pow(remainder, n);
        originalNumber /= 10;
    }

    if(result == number)
        flag = 1;
    else
        flag = 0;

    return flag;
}
```

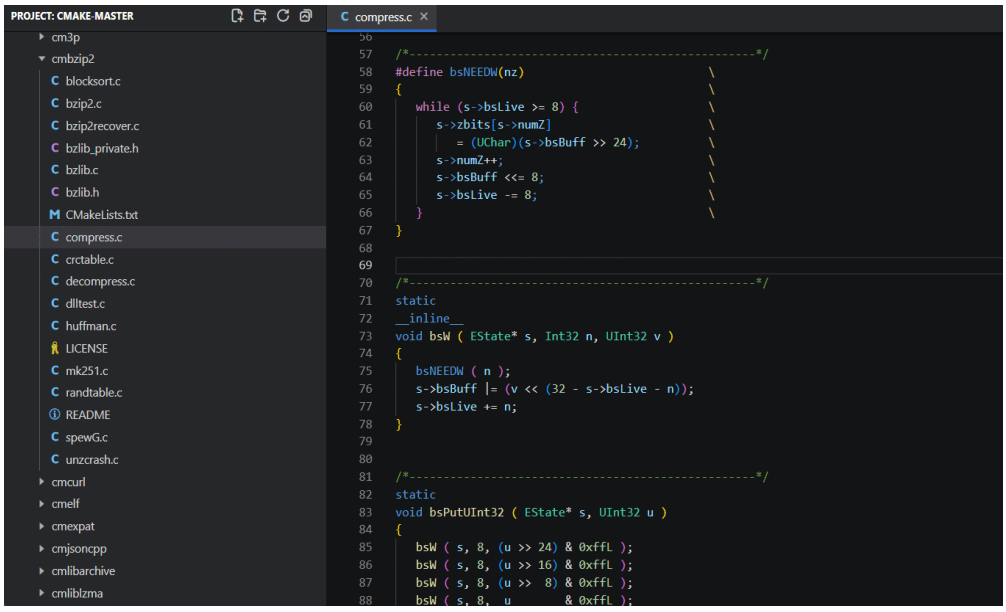
- **代码补全/提示（Code Completion/Hinting）**

  CodeArts IDE for C/C++ 代码补全包含了各种代码编辑功能，包括：代码完成，快速信息，成员列表以及参数信息。当您输入字符时，代码补全若知道可能的补全选项，则会自动弹出成员列表。如果您继续输入字符，成员列表（变量，方法等）将被过滤为仅包含您输入字符的成员。您可通过光标单击或者按“**Enter**”或“**Tab**”键插入选定的成员名称。该功能会提供各种提示信息帮助您更加方便快速的编辑代码。

```
/*-------------------------------------------*/
static void tooManyBlocks ( Int32 max_handled_blocks )
{
    fprintf ( stderr,
              "%s: `%s' appears to contain more than %d blocks\n", );

    exit ( 1 );
}
```

- **全局符号搜索（Global Symbol Search）**

  Ctrl+T导出搜索框，输入需要查找的符号，页面会显示出当前文件夹所有包含此符号的文件，单击即可跳转。或者按**向上**或**向下**选择并按**Enter**导航到您想要的位置。



- **实时检查编译错误（该功能依赖compile_commands.json文件）**

  实时检查编译错误是解决编码错误的建议编辑，包括自动补全，实时语法检查等。

  当编译错误时，会在错误处出现波浪线。可将光标移动或单击到C/C++的代码错误上时，会显示黄色灯泡，表示可以使用快速修复。单击灯泡或按**Ctrl+.**会显示可用的快速修复和重构列表。

```
        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);
     }
  }
#ifdef _WIN32
   BZ2DLLFreeLibrary();
#endif
```

- **Compile_commands.json 管理功能**

  Compiler 模式功能全面，但需要**compile_commands.json**文件编译数据库才能正常工作，可使用三种方式获取该文件。

- 使用内置 CMake Build Tool 插件（**推荐**）。构建 CMake 项目，会自动生成**cmake-build-debug/compile_commands.json**文件， 并且插件会自动将该文件导入到 **.arts**文件夹。
- 使用 CMake 生成。 如果当前工程是 CMake 工程，可以通过添加参数**-DCMAKE_EXPORT_COMPILE_COMMANDS=1**生成**compile_commands.json**，并通过**帮助->显示所有命令->Huawei C/C++：导入编译数据库文件**命令导入。

- 使用 Huawei C/C++ 提供的**Generate**命令。可通过**帮助->显示所有命令->Huawei C/C++：生成编译数据库文件**，并选择存放源文件的文件夹，该方法分析头文件生成对应的编译数据库。

同时 Huawei C/C++也支持以下功能：

- 通过命令或 API 导入**compile_commands.json**文件（**帮助->显示所有命令->Huawei C/C++：导入编译数据库文件**）
- 合并多个**compile_commands.json**文件.
- 移除**compile_command.json**文件中重复的命令.
- 导入时为**clangd**提供额外的参数设置.
- 索引更新命令
  - 同步工程索引（**帮助->显示所有命令->Huawei C/C++：同步工程索引**）
  - 同步文件夹索引（**资源管理器右键菜单->Huawei C/C++：同步文件夹索引**）
  - 同步文件索引（**资源管理器右键菜单->Huawei C/C++：同步当前文件索引**）
  - 重置工程索引（**帮助->显示所有命令->Huawei C/C++：重建全项目索引**）
  - 编辑源文件的编译选项并刷新索引（**右键菜单->编辑编译参数**）

以上命令和功能在 Compiler 模式或 Hybrid 模式均有效。

4.2.3 代码重构操作

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 支持重构操作，提供了多种重要的重构类型，来改变编辑器中的代码库。CodeArts IDE for C/C++ 内置了对 C/C++ 重构的支持，在本专题中，我们将展示 C/C++ 语言服务的重构支持。

- **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。

```
int main( )
{
    Line line;
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() <<endl;
    return 0;
}

class Foo {
    int bar;
};
```

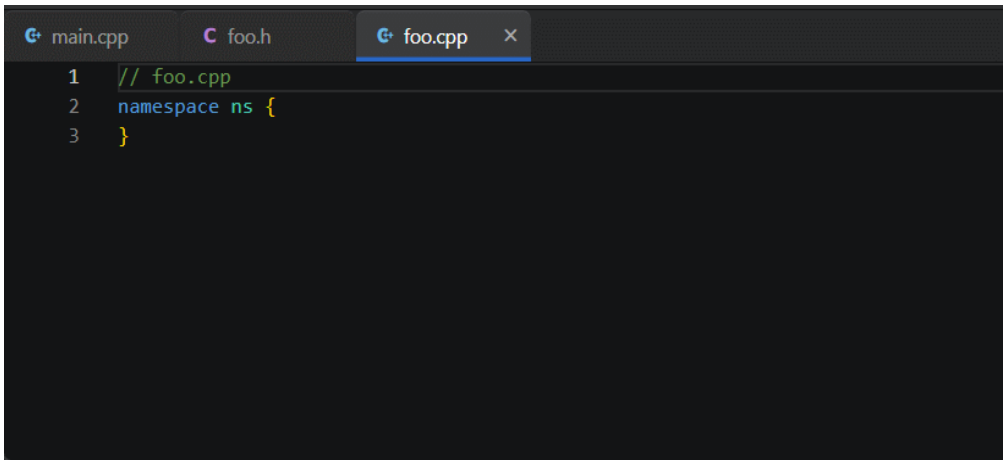
- **根据声明顺序排序函数（Sort functions to declarations）**

  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。


```cpp
// foo.cpp
#include "foo.h"
void Foo::foo3(){}
void Foo::foo2(){}
void Foo::foo1(){}

void foo3(){}
void foo1(){}
void foo2(){}

void test()
{
  Foo foo = getFoo();
  auto var1 = f^oo;
  auto var2 = foo;
  foo = getAnotherFoo();
  doWork(foo);
}
```

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

  将头文件的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。



- **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（Swap if branches）**，可自动交换if和else分支。

```cpp
int f(bool b) {
    int ret = 1;
    if (b) {
        ret += 4;
    } else {
        ret = 3;
    }
    return ret;
}
```

- **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。

```
originalNumber = number;
while (originalNumber != 0)
{
    remainder = originalNumber%10;
    result += pow(remainder, n);
    originalNumber /= 10;
}

if(result == number)
    flag = 1;
else
    flag = 0;
```

- **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。

```
int foo(int A) {
  std::cout << "A: " << A << std::endl;
  if (A > 2)
    return 1;
  else
    return 0;
}

int test() { return foo(5); }

int main() { return foo(1); }
```

- **生成 getter 和 setter（Generate getter and setter)**

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

```
class Foo {
  int Test;
};
```

- **声明隐式成员（Declare implicit members）**
  此选项会将类的隐式成员在类中声明，当选中类名时，重构选项可用。

```
class Foo {
    void foo1();
};
```

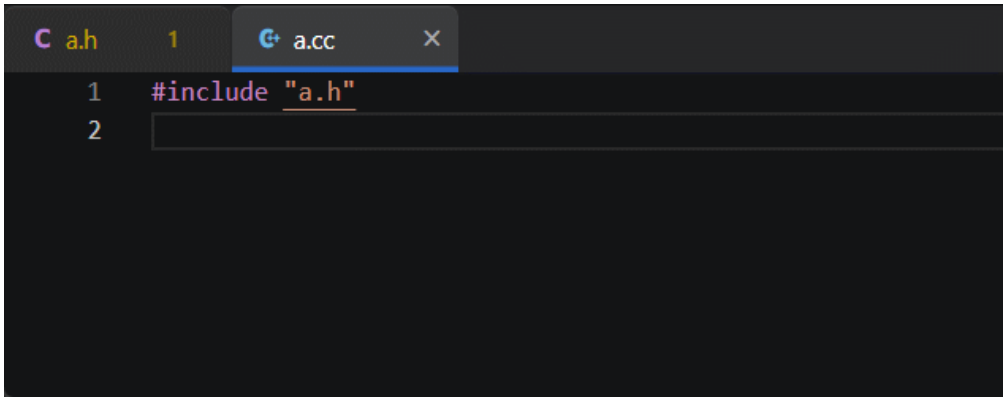
- **填充 switch 语句（Populate switch）**
  该功能可以自动填充switch语句。选中任意switch字段，并且单击黄色灯泡，选择**填充switch语句**。

```
enum Color { RED, GREEN, BLUE };

void f(Color color) {
  switch (color) {}
}
```

- **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。

```
#include <iostream>

using namespace std;
void f()
{
  cout << 1 << endl;
}
```

- **移动函数体到声明处（Move function body to out-of-line）**
  将函数/方法定义移动到它声明的位置。



- **在内部添加定义（Add definition in-place）**
  在当前函数/方法并且在类内部生成函数定义。当光标移动到函数/方法时，单击黄色灯泡，重构选项可用。

```
class Foo {
  bool test(int P) const;
};
```

- **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，单击黄色灯泡，重构选项可用。

```
class Baz {
  void foo();
  void test();
  void bar();
};

void Baz::foo() {}
void Baz::bar() {}
```

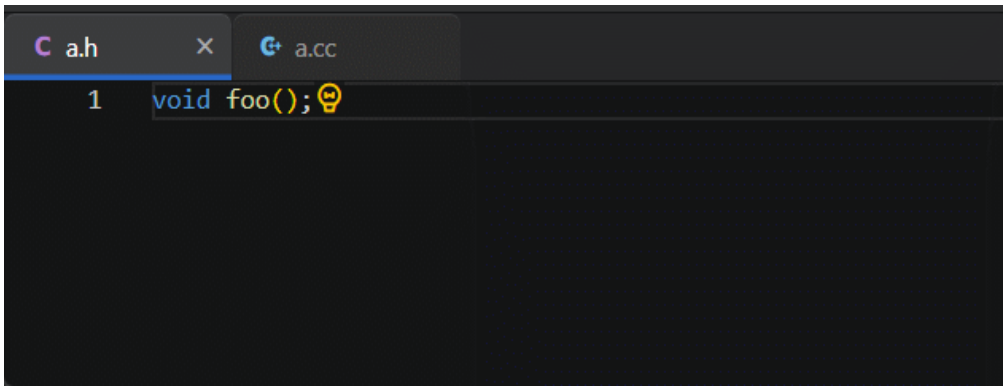
- **展开宏（Expand macro）**
  在页面上添加**展开宏（Expand macro）**，以便在可扩展/可折叠的部分提供内容。

```
#define AAA 1

int f() { return AAA; }
```

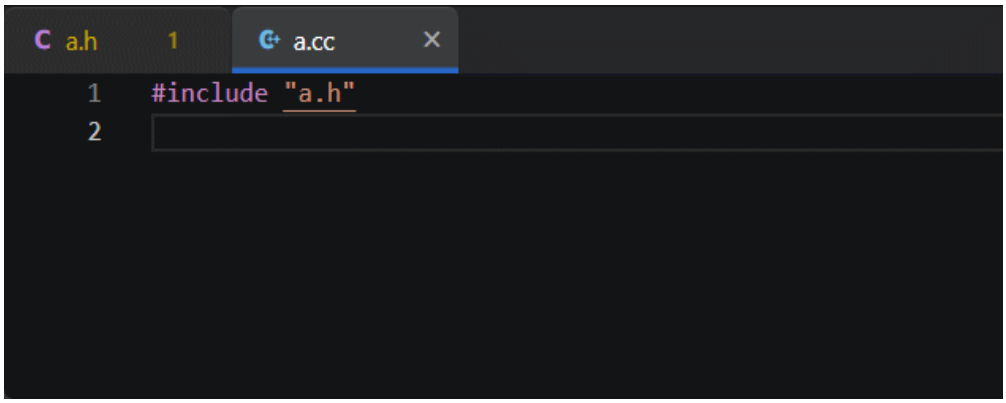
- **展开 auto（Expand auto type）**
  展开 auto type所隐藏的变量类型。

```
auto i = 1;
```

- **函数定义外移（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。



- **函数定义内移（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。



- **转为原始字符串（Convert to raw string）**
  此方法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，单击黄色灯泡，重构选项可用。

```
void test() {
  Foo foo = getFoo();
  auto var1 = foo;
  auto var2 = foo;
  foo = getAnotherFoo();

  printf("\"a\"\nb");

  doWork(foo);
}
```

- **快速修复（Quick fixes)**
  快速修复是解决简单编码错误的建议编辑，包括自动补全，实时语法检查等。当光标移动或单击到C/C++的代码错误上时，会显示黄色灯泡，表示可以使用快速修复。单击灯泡或按Ctrl+.会显示可用的快速修复和重构列表。

4.3 Cmake 工程支持

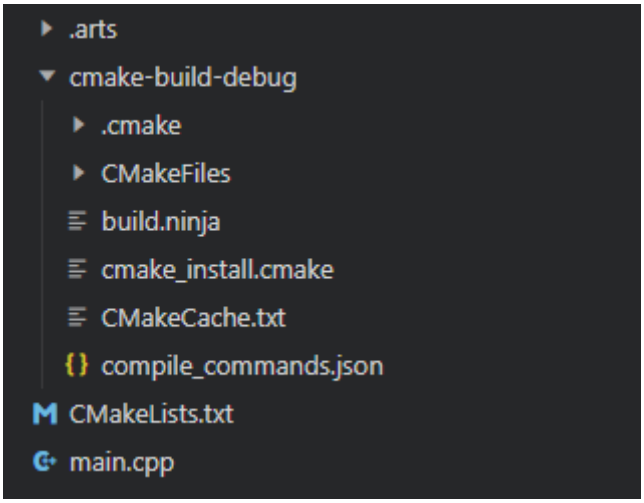
4.3.1 简介

CodeArts IDE for C/C++中基于CMake工程提供了功能齐全、方便和强大的工作流，让用户可以轻松配置、构建和调试CMake工程。

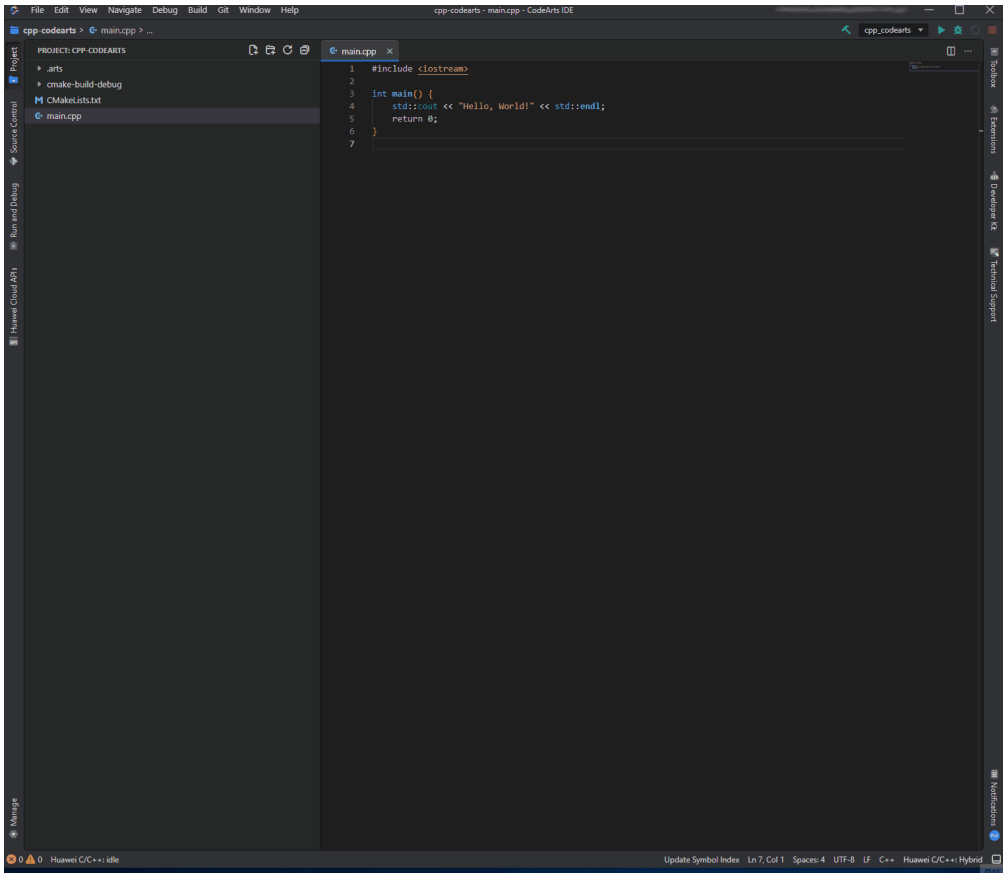
4.3.2 CMake 工程加载

CodeArts IDE for C/C++识别到打开的工程为CMake工程时（默认为Debug模式），将自动完成以下操作：

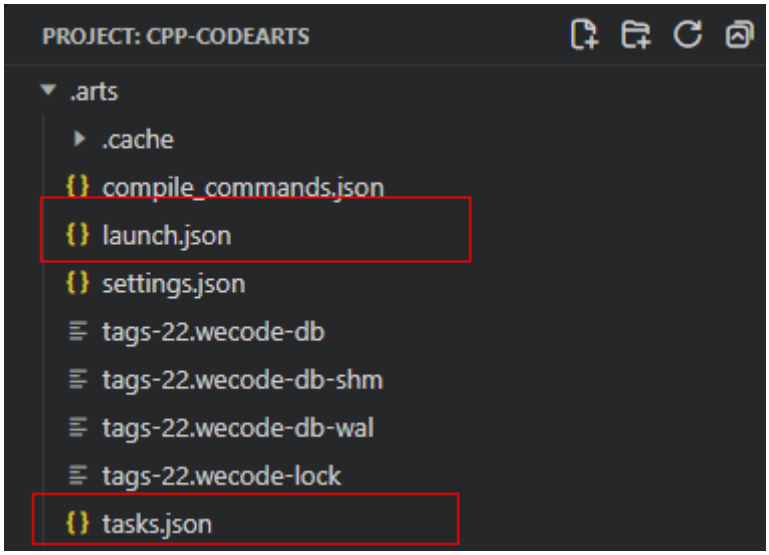
1. 生成cmake-build-debug目录，该目录中有compile_commands.json 文件，.cmake目录，CMakeFiles目录和CMakeCache.txt等文件。



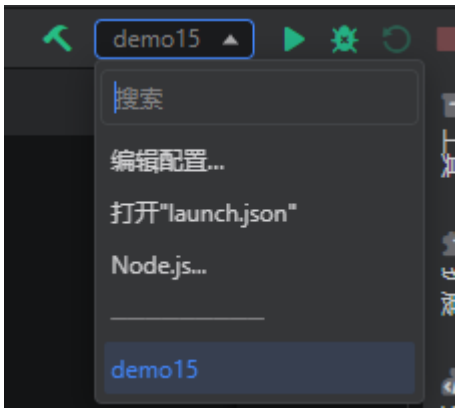
2. 生成compile_commands.json文件并导入，CMake工程获得Huawei C/C++ 插件的代码检测与提示的功能。



3. 生成tasks.json和launch.json文件。



- tasks.json是配置构建任务的文件。
- launch.json是启动程序的配置文件，该文件中的configurations会在运行和调试下拉框展示。

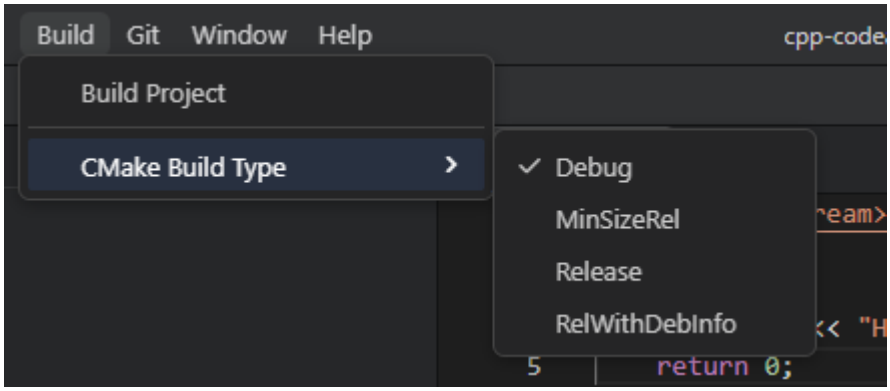


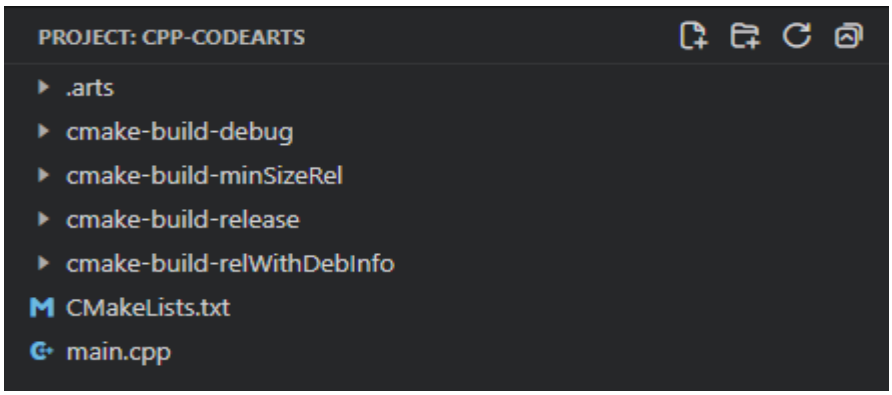
在此过程中，状态栏显示加载过程，单击可以查看具体的加载日志。

4.3.3 多种构建类型

CMake工程一共有四种构建类型，分别是 Debug，MimSizeRel，Release，RelWithDebInfo。选择需要的构建类型，工程构建之后， 生成对应的目录。

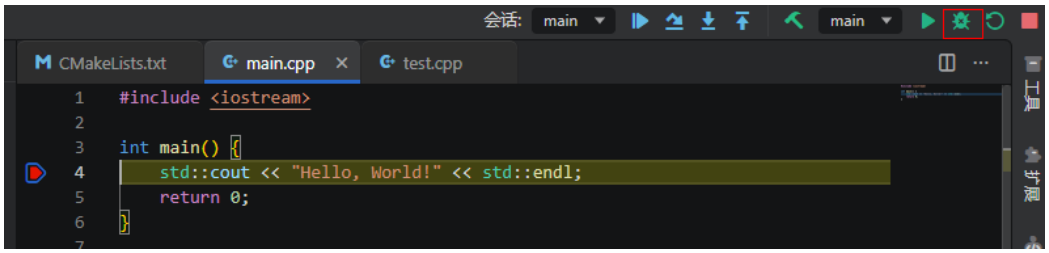
- Debug：禁用优化并包含调试信息。
- Release：包括优化但没有调试信息。
- MinRelSize：优化大小。没有调试信息。
- RelWithDebInfo：优化速度并包含调试信息。





4.3.4 CMake 工程调试

打开cpp文件，选择所需断点，然后单击调试图标。

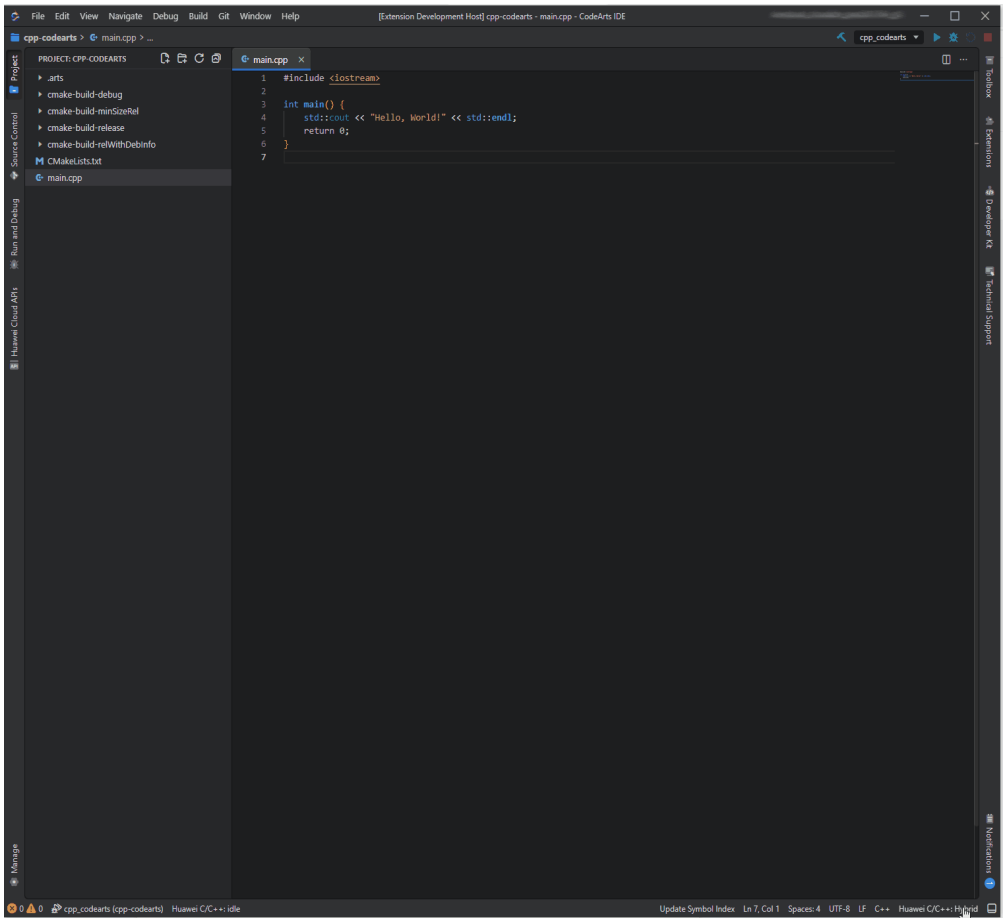


调试时，遇到需输入的场景，参考**CodeArts IDE for C/C++ 常见问题**文档进行配置。

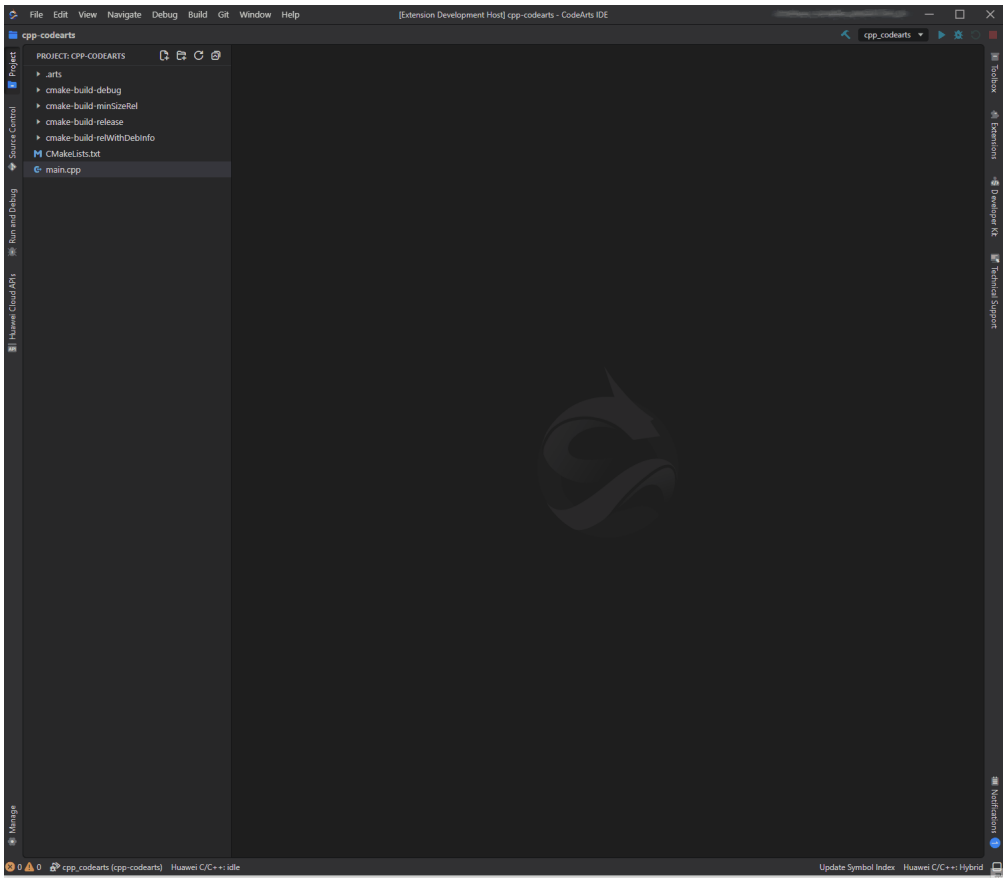
4.3.5 CMake 工程运行

可通过以下几种方式运行：

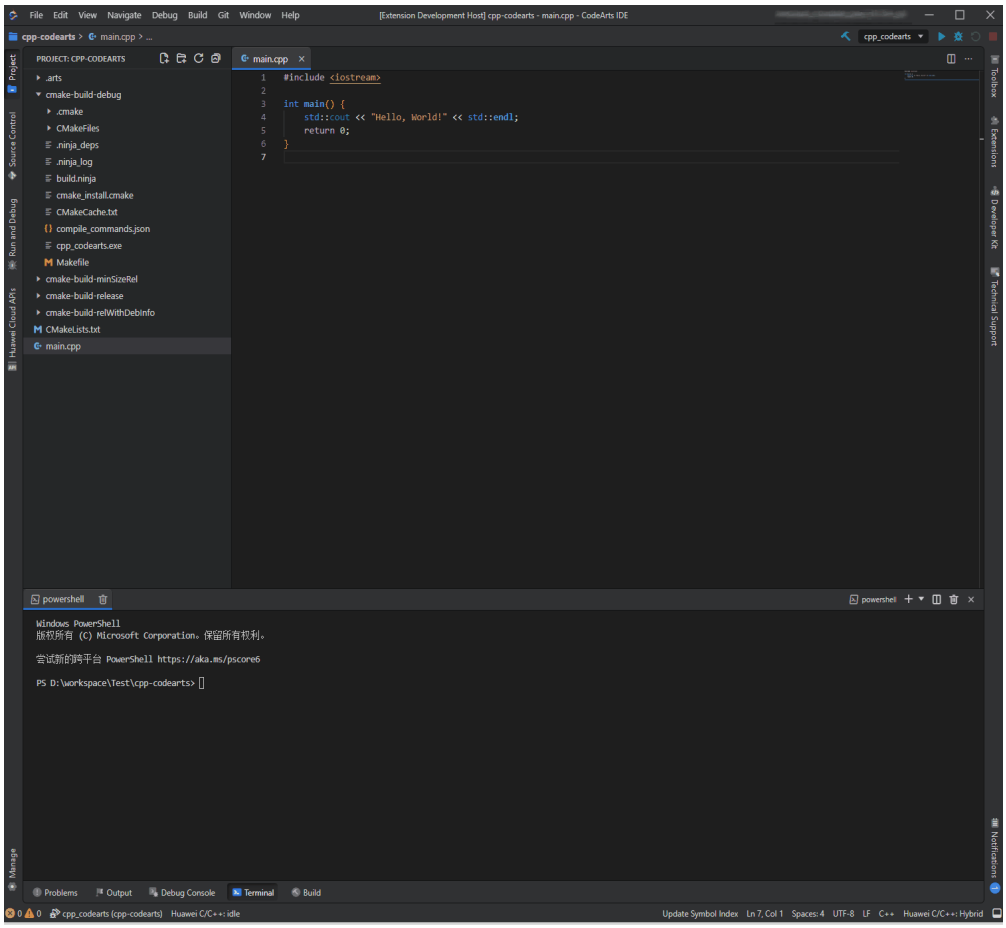
- 执行launch.json文件



- 右键运行可执行文件



- 控制台输入需要运行的文件路径



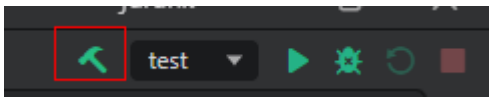
运行时，遇到需输入的场景，参考**CodeArts IDE for C/C++ 常见问题**文档进行配置。

4.3.6 CMake 工程构建

可从以下任一渠道构建：

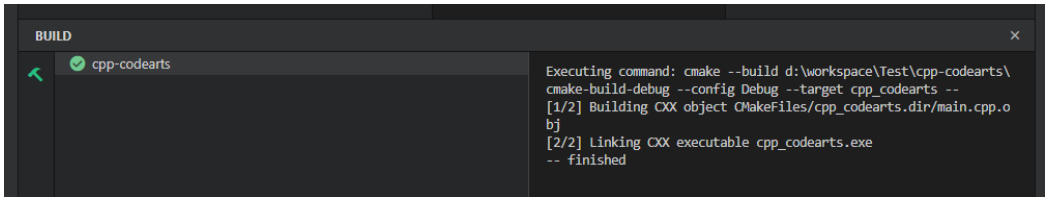
- 打开命令面板并运行CMake Build Tool: CMake Targets命令。
- 单击Build菜单。

- 单击build图标。



- Build面板。

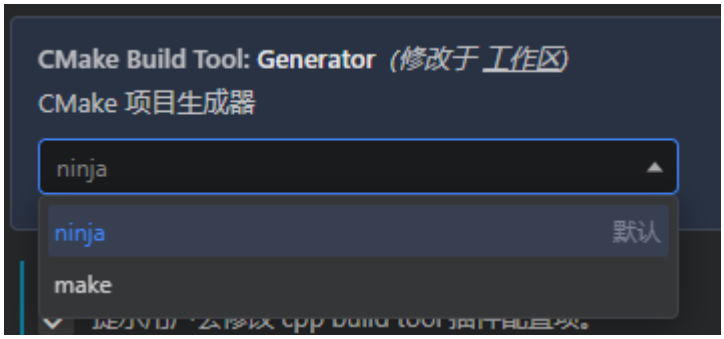
构建后，构建日志展示在build面板右侧。



4.3.7 多种生成器类型

CodeArts IDE for C/C++为CMake工程提供了两种生成器类型：ninja和make。IDE构建项目时，默认使用ninja生成器。

打开设置页面，搜索 cpp-build-tool.CMakeBuildTool.generator，在工作区修改生成器类型。



4.4 常用设置项

1. **排除或包含某些文件夹**

   Tag 或 Hybrid 模式下：

- 排除某些目录 设置项中搜索huawei-cpp.wecodeDb.excludePaths，默认值为：

```
**/.mm/**
**/.git/**
**/build/**
**/output/**
```

- 包含文件夹 设置项中搜索huawei-cpp.wecodeDb.includeFolders，将文件夹绝对路径填入即可。

Compiler 模式下：

- 排除某些目录： 设置项搜索：huawei-cpp.codebase.generator.pathsExclude，使用 Glob 通配符排除一些路径，然后重新生成 compile_commands.json 才会生效。

2. **开启/关闭问题窗口中的诊断信息**

   设置项中搜索huawei-cpp.clangd.ignoreDiagnostics：

   - none： 显示所有诊断信息。
   - all： 隐藏所有诊断信息。
   - not_indexed：仅当当前文件有编译选项或已经索引时显示诊断信息。

3. **修改系统头文件提供方**

   Huawei C/C++默认从 compile_commands.json 中的编译器提取系统头文件，如果无法提取则使用自带的 RTOS 头文件，可通过修改设置项改变默认规则： 设置项中搜索huawei-cpp.codebase.systemHeaderProvider：

   - Compiler： 仅根据 compile_commands.json 中提取系统头文件。
   - None： 从环境变量中获取系统头文件

4. **开启内联提示/高亮不活跃代码，开启/关闭/修改语义高亮颜色**

   开启或关闭内联提示： huawei-cpp.clangd.enableInlayHints

   开启或关闭高亮不活跃代码： huawei-cpp.syntaxColor.enableInactiveCode

   开启或关闭语义高亮： huawei-cpp.syntaxColor.enable

5. **cmake工程构建工具的路径**

   CodeArts IDE for C/C++提供了CMake工程构建、调试所需要的相关工具， 用户可以直接构建、调试CMake工程， 不必手动配置相关环境变量。用户目录下.codearts下面内置了cmake、MinGW、ninja工具CMake Build Tool插件默认先读取内置工具路径。

   - cpp-build-tool.CMakeBuildTool.CMake获取cmake工具的路径。
   - cpp-build-tool.CMakeBuildTool.debugger获取MinGW工具的路径。
   - cpp-build-tool.CMakeBuildTool.buildTool获取ninja工具的路径。

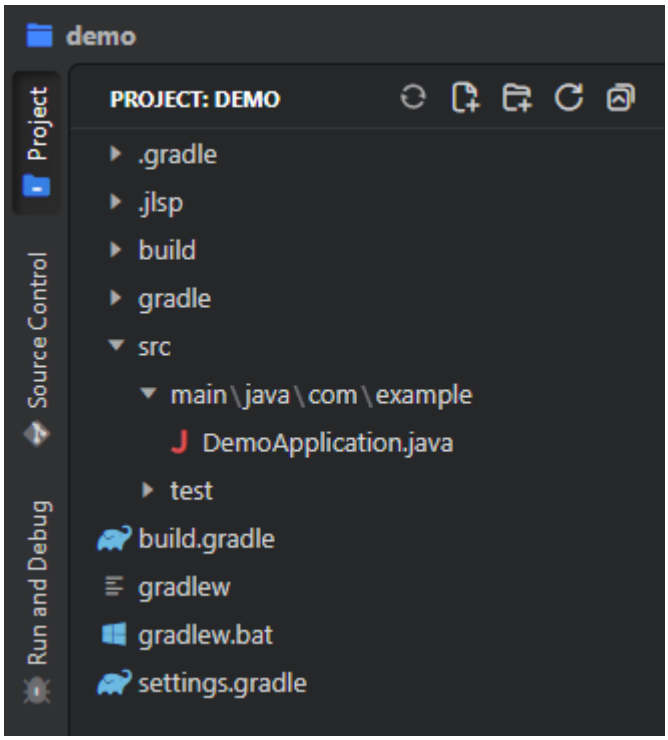
5 Java

5.1 使用 Java 项目

5.1.1 简介

当您打开包含源代码文件的任意文件或文件夹时，CodeArts IDE仅提供基本的文本编辑功能。要获得完整的Java编码帮助，CodeArts IDE会自动通过分析文件内容并生成项目的元数据来初始化项目。所有项目的元数据都存储在项目根目录下的**.jlsp**文件夹中。

在CodeArts IDE中，您的Java项目的内容会显示在资源管理器视图中（“Ctrl+Shift+E”/“Alt+1”(IDEA键盘映射)）中，该视图提供了常见的文件管理功能。



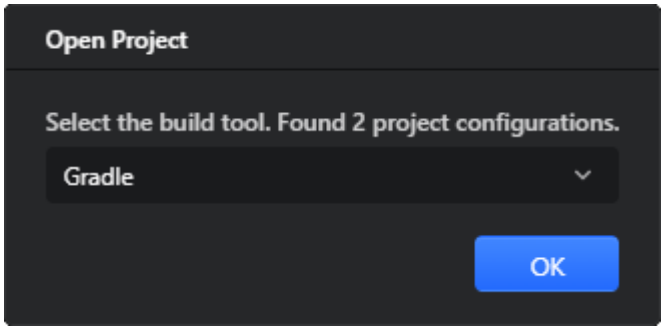
5.1.2 管理 Java 项目

5.1.2.1 打开文件夹或现有 CodeArts IDE 项目

步骤1 在主菜单中选择**File**>**Open Project**。

步骤2 在打开的**Open Project**对话框中，找到所需的文件夹，然后单击**Open**。

CodeArts IDE会自动检测项目文件夹中的**pom.xml**或**build.gradle**文件，并自动加载相应的项目（Maven或Gradle）。如果两个文件都存在，CodeArts IDE会提示您指定项目的构建系统。



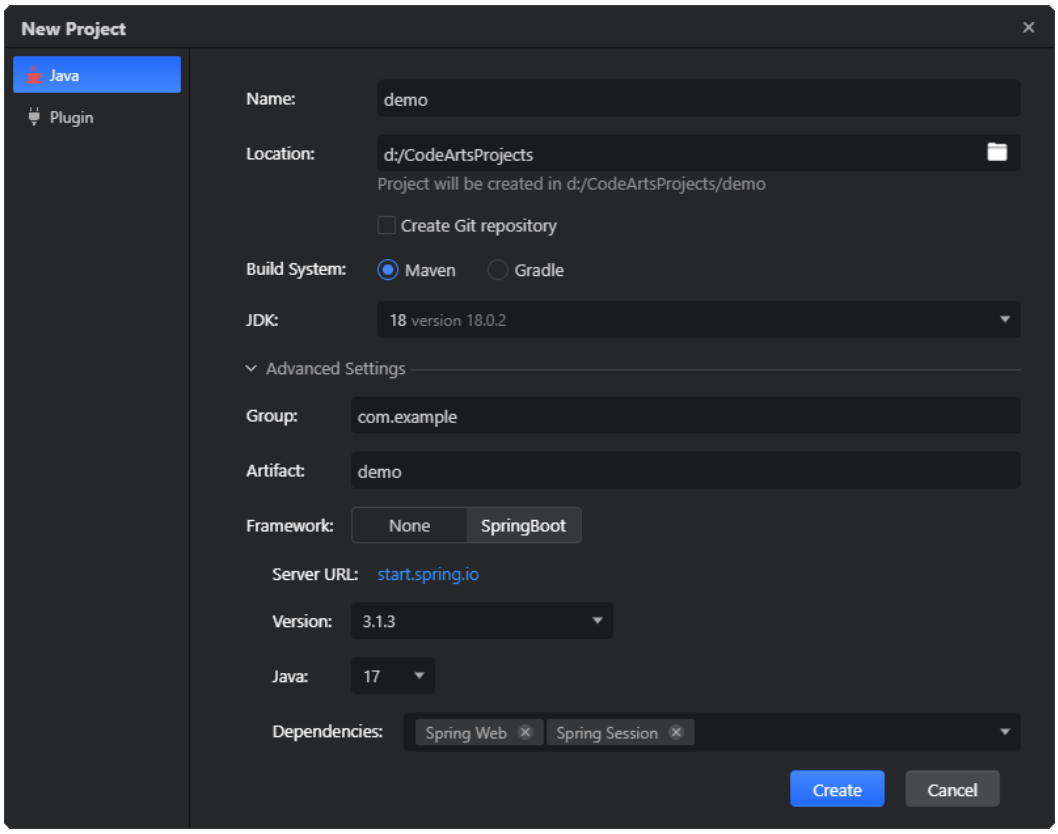
----结束

如果**.jlsp**文件夹尚不存在，CodeArts IDE会创建该文件夹，并扫描项目的内容。一旦此过程完成并且Java编码辅助功能可用，CodeArts IDE会显示相应的通知。

5.1.2.2 创建新项目

步骤1 在主菜单中，选择**File**>**New**>**Project**。

步骤2 在打开的**New Project**向导中，提供常见的项目参数：项目名称、位置、使用的构建系统和项目JDK。CodeArts IDE会自动检测您系统上安装的JDK，并在**JDK**列表中显示它们。要在创建的项目内自动**初始化Git存储库**，请选中**Create Git repository**复选框。



步骤3 要基于Spring Boot框架创建项目，请按照以下步骤操作：

a. 展开**Advanced Settings**部分，并将框架**Framework**设置为**SpringBoot**。

b.在版本**Version**列表中，选择Spring Boot的版本。

c.在**Java**列表中，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）。

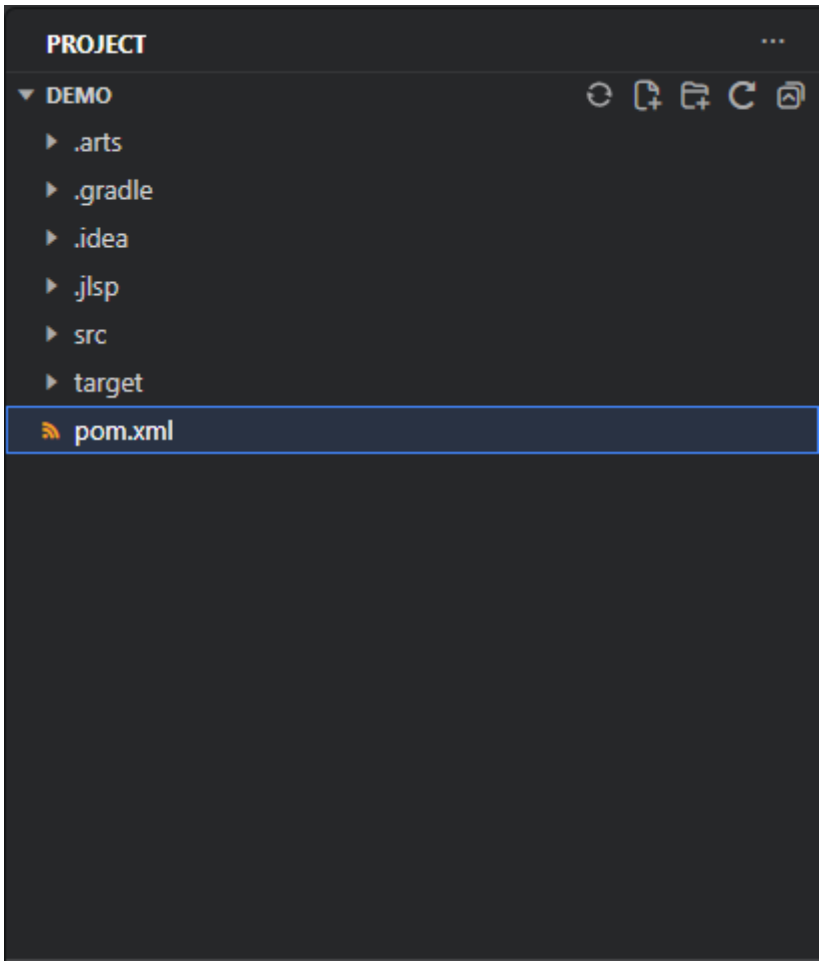
d.在依赖项**Dependencies**列表中搜索并选择所需的项目依赖项。所选的项目依赖项将在**build.gradle**（对于Gradle）或**pom.xml**（对于Maven）中注册。

步骤4 单击创建**Create**。CodeArts IDE会根据所选的模板创建项目结构。

----结束

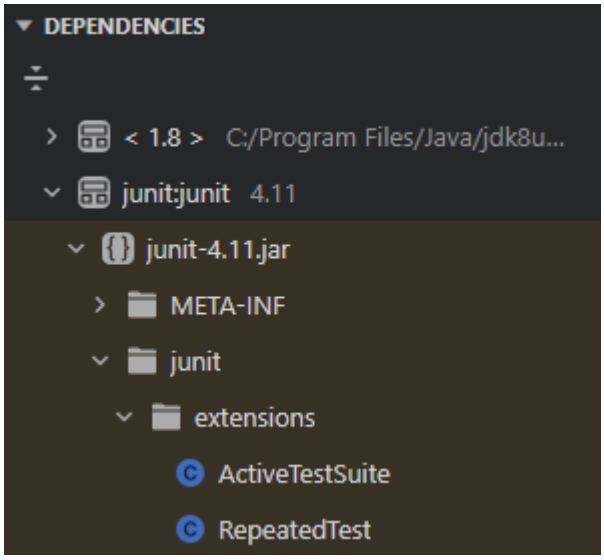
5.1.2.3 重新加载项目

如果您修改了项目的构建脚本（Gradle的**build.gradle**或Maven的**pom.xml**），例如添加了一个依赖项，您需要重新加载项目以使CodeArts IDE解析项目的结构。要执行此操作，请在资源管理器工具栏上单击**Reload Project**按钮( )。



5.1.2.4 查看项目依赖关系

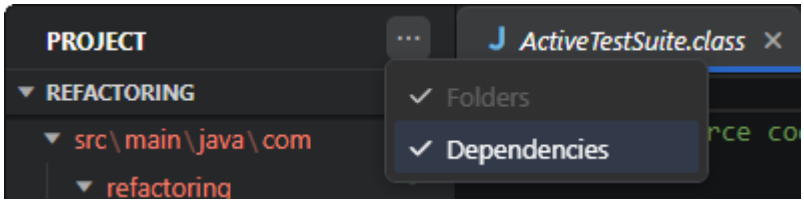
如果您的项目在**build.gradle**（对于Gradle项目）或**pom.xml**（对于Maven项目）中声明了依赖项，CodeArts IDE会在资源管理器中的**DEPENDENCIES**部分显示它们。**配置的JDK**的内容也会显示在**DEPENDENCIES**部分中。



您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。

约束与限制

如果**Dependencies**被隐藏了，您可以通过在资源管理器中单击**Views and More Actions**按钮(...)并在弹出菜单中启用**Dependencies**来显示它。



5.1.2.5 创建文件和文件夹

CodeArts IDE支持在创建Java源文件时应用模板。当您在资源管理器中创建一个**.java**文件时，语言服务器会自动生成类的主体，并为您填充包的信息。

创建文件夹

步骤1 在资源管理器中，右键单击要在其中创建新文件夹的文件夹，然后从上下文菜单中选择**New Folder**。

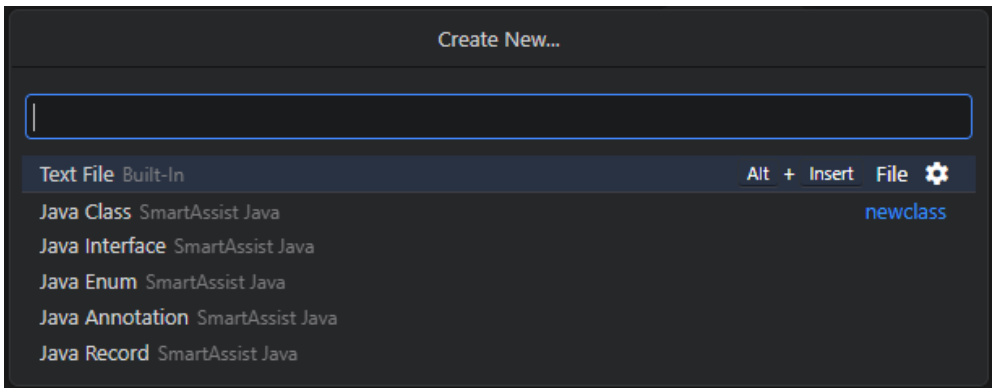
步骤2 键入文件夹的名称，然后按“Enter”键。

----结束

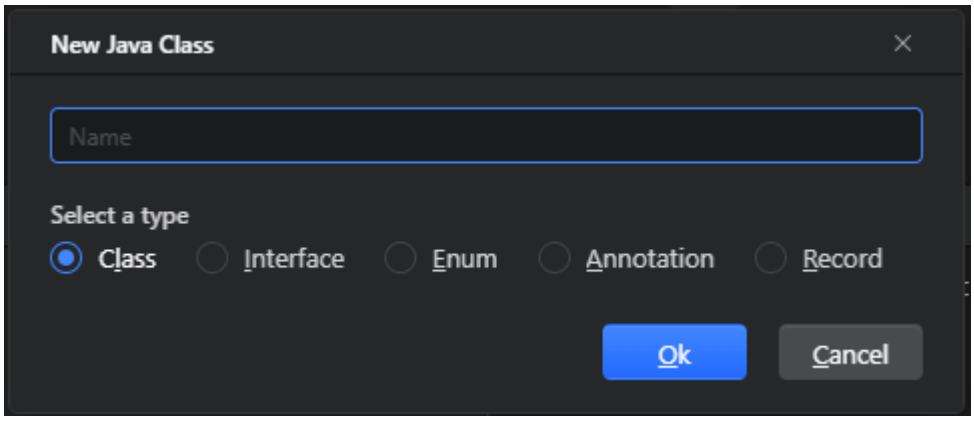
创建文件

步骤1 在资源管理器中，右键单击要在其中创建新文件的文件夹，然后从上下文菜单中选择**New File**或按下“Ctrl+Alt+Win+N”。或者，在主菜单中选择**File>New>File...**

步骤2 在打开的**Create New...**弹出窗口中，根据需要选择相应的选项。



步骤3 假如选择Java Class，在打开的**New Java Class**对话框中，提供文件名。



步骤4 单击**Ok**。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。

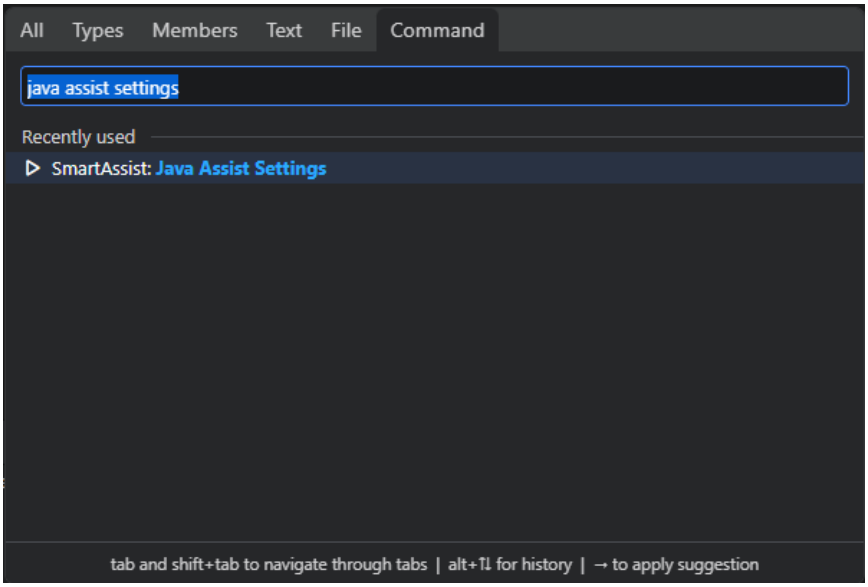
----结束

5.1.3 配置项目

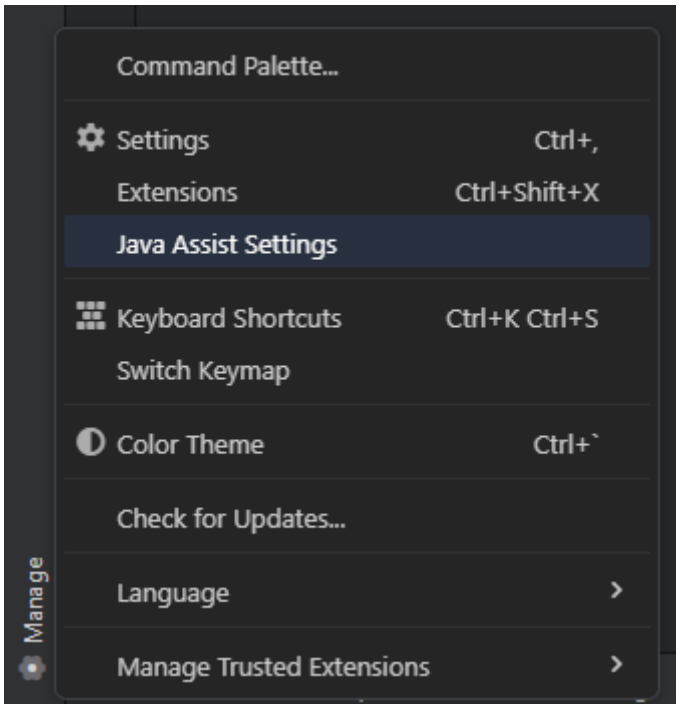
5.1.3.1 简介

要配置Java项目，先打开**Java Assist Settings** 对话框，您可以通过以下任意一种方式打开**Java Assist Settings** 对话框：

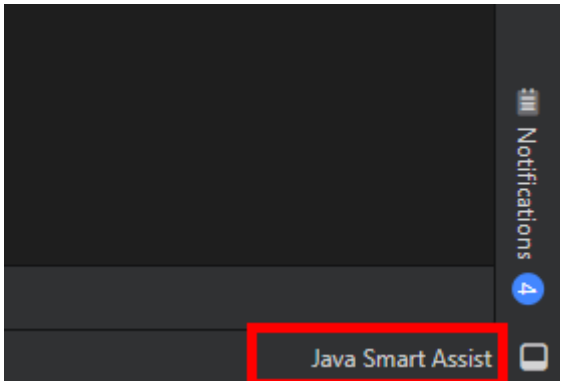
- 在命令面板中运行**SmartAssist: Java Assist Settings**命令（“Ctrl+Shift+P”/“Ctrl Ctrl”）来打开**Java Assist Settings** 对话框。



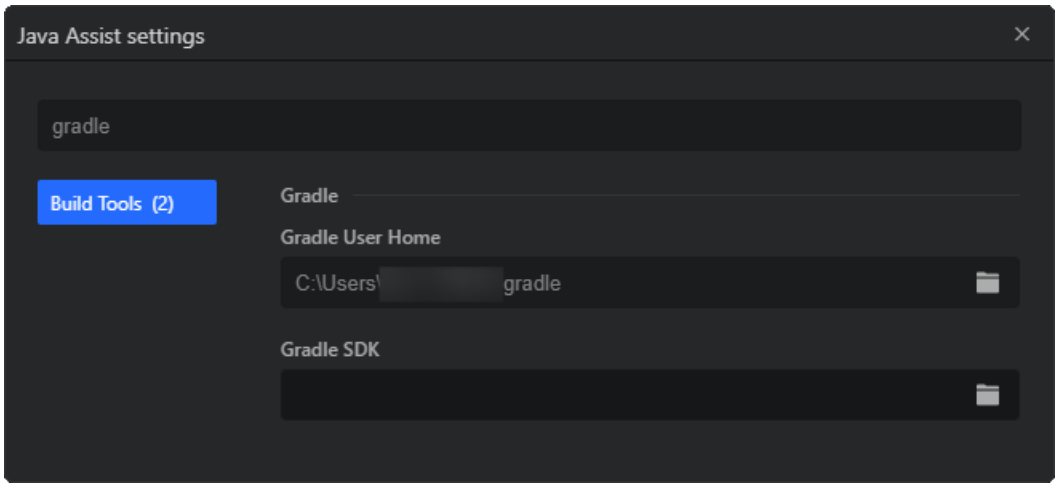
- 单击CodeArts IDE左下角Manage选项卡，选择**Java Assist Settings**，打开**Java Assist Settings** 对话框。



- 单击CodeArts IDE 右下角的Java Smart Assist，打开**Java Assist Settings** 对话框。

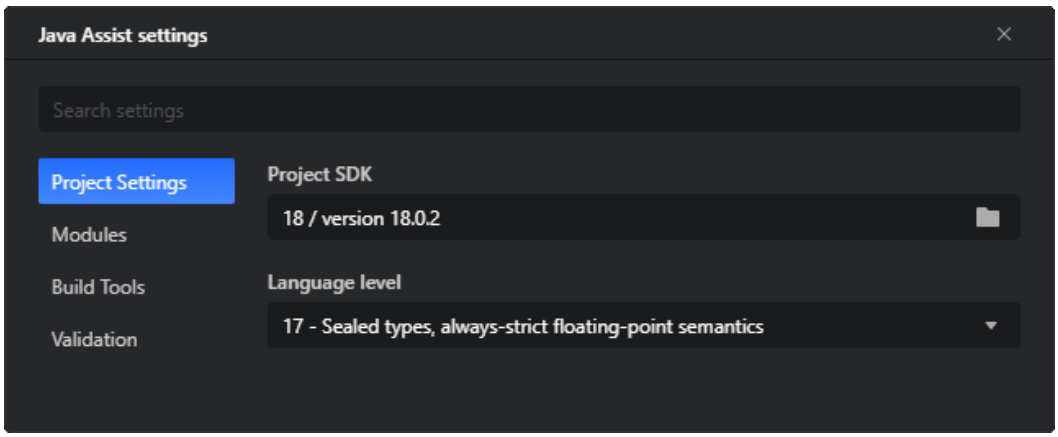


打开**Java Assist Settings** 对话框之后，在对话框中，使用搜索字段快速定位所需的设置。



5.1.3.2 项目设置

在**Project Settings**页面，您可以提供Java SDK（JDK）的路径并配置项目的语言级别，该级别定义了代码编辑器提供的一组编码辅助功能（例如代码完成或错误突出显示）。

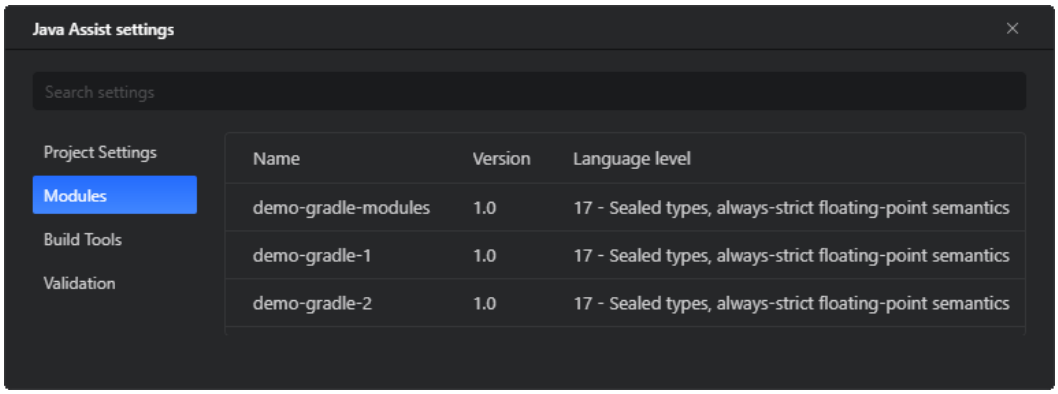


选定的语言级别将与项目生成配置文件同步。如果在设置中调整它，CodeArts IDE会相应地更新相应的值：

- 对于Gradle在**build.gradle**文件中，**sourceCompatibility**和**targetCompatibility**的值已经更新。
- 对于Maven在**pom.xml**文件中，Maven的**source**和**target**已经更新。

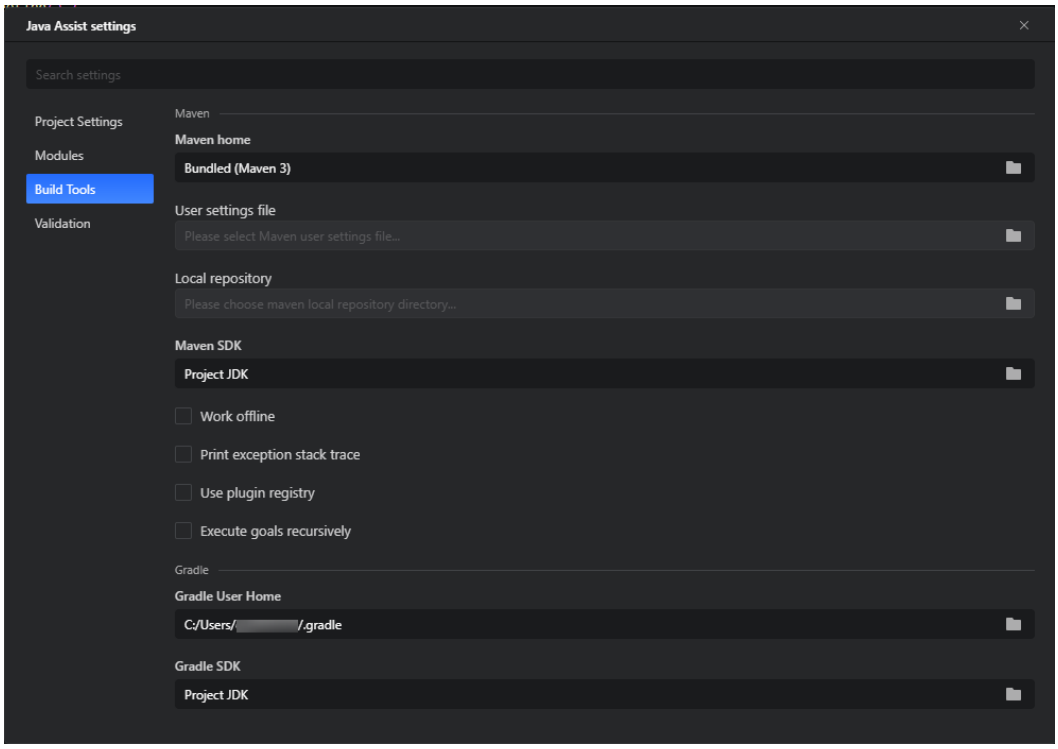
5.1.3.3 模块设置

在**Modules**页面列出了您项目中定义的模块。



5.1.3.4 构建工具设置

在**Build Tools**页面上，您可以配置项目使用的构建工具（Maven或Gradle）。



Maven 设置

- **Maven home**：请使用此字段选择捆绑的Maven版本（Maven 3）或者单击**Browse**按钮，手动定位您自己的Maven安装位置。
- **User settings file**：在此字段中，指定包含Maven用户特定配置的文件。
- **Local repository**：在此字段中，指定用户主目录下的本地目录的路径，该目录存储下载并包含临时生成工件。
- **Maven SDK**：从此列表中，选择要与Maven一起使用的JDK：捆绑的JDK、项目级JDK或从系统变量（如**JAVA_HOME**）解析的JDK。
- **Work offline**：如果选中，Maven将在脱机模式下工作。它不连接到远程存储库，只使用本地可用的资源。此选项对应于**--offline**命令行选项。
- **Print exception stack traces**：如果选中，则生成异常堆栈跟踪。此选项对应于**--errors**命令行选项。
- **Use plugin registry**：如果选中，则可以引用Maven的插件注册表。此选项对应于**--no-plugin-registry**命令行选项。
- **Execute goals recursively**：如果选中，则将递归执行生成，包括嵌套项目。此选项对应于**--non-recursive**命令行选项。

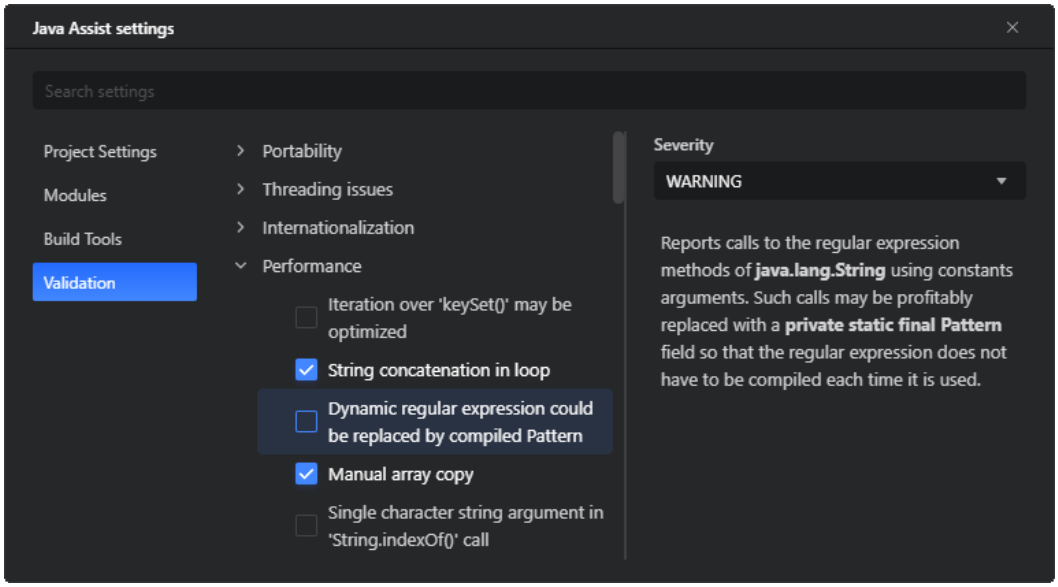
Gradle 设置

- **Gradle User Home**：在此字段中，指定Gradle用户主目录的路径（默认为**$USER_HOME/.gradle**），用于存储全局配置属性和初始化脚本、缓存和日志文件。默认值基于**GRADLE_USER_HOME**环境变量的值提供。要修改它，您可以设置环境变量或单击“”按钮并手动定位所需的Gradle用户主目录。
- **Gradle SDK**：从此列表中选择要与Gradle一起使用的JDK：捆绑的JDK、项目级别的JDK或从系统变量（如**JAVA_HOME**）解析的JDK。

5.1.3.5 代码校验规则

代码验证规则可以在编译之前检测和纠正项目中的错误代码。当您输入代码时，检测到的问题会在代码编辑器中实时显示出来。有关CodeArts IDE中代码验证的更多详细信息，请参阅**代码校验**。

- 要启用或禁用验证规则，请使用其名称旁边的复选框。
- 要调整相应代码问题在代码编辑器中突出显示的方式，请在**Severity**列表中选择所需的严重性级别。



5.2 代码编辑

5.2.1 简介

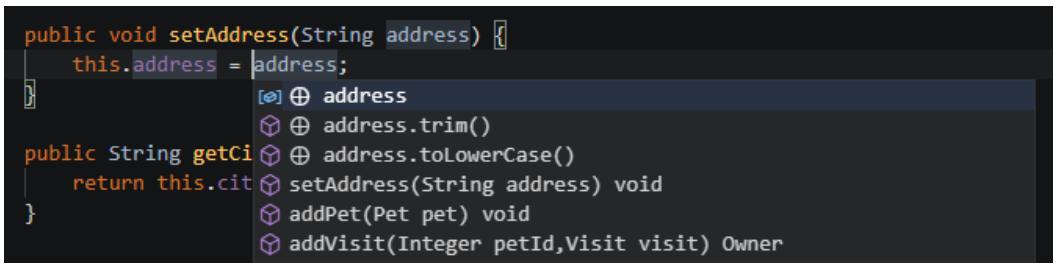
CodeArts IDE是一个具有丰富编辑功能的源代码编辑器。您可以使用代码片段、各种代码操作（如生成Getter/Setter和组织导入）以及重构来优化您的代码库。

在**Java重构**和**Java源代码操作**中了解更多信息。

5.2.2 代码补全

5.2.2.1 简介

CodeArts IDE for Java中的代码完成由SmartAssist提供支持。它的代码补全推荐基于数千个开源项目，旨在准确匹配当前代码上下文。在补全推荐列表中，SmartAssist提供的补全推荐列表会用⊕图标表示。

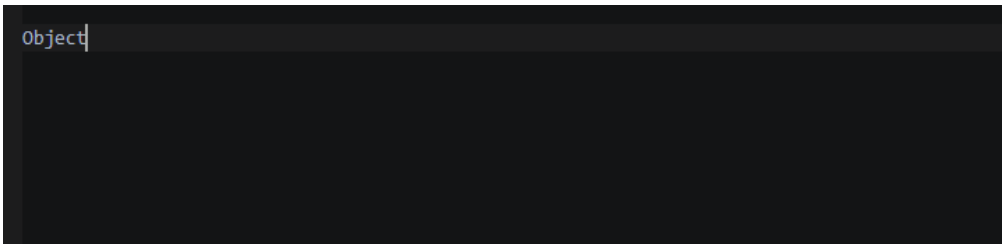


说明

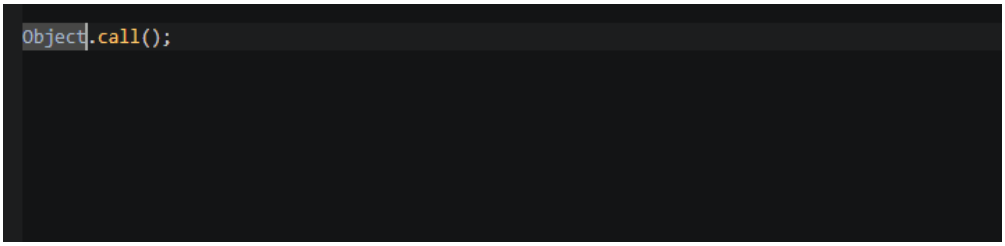
有关CodeArts IDE代码完成功能的一般信息，请参见**代码补全**。

5.2.2.2 触发代码补全

- 要手动触发代码补全，请按“Ctrl+Shift+Space”（IDEA键盘映射）或键入触发字符（例如点字符'.'）。
- 要插入所选符号，请按“Enter”键。



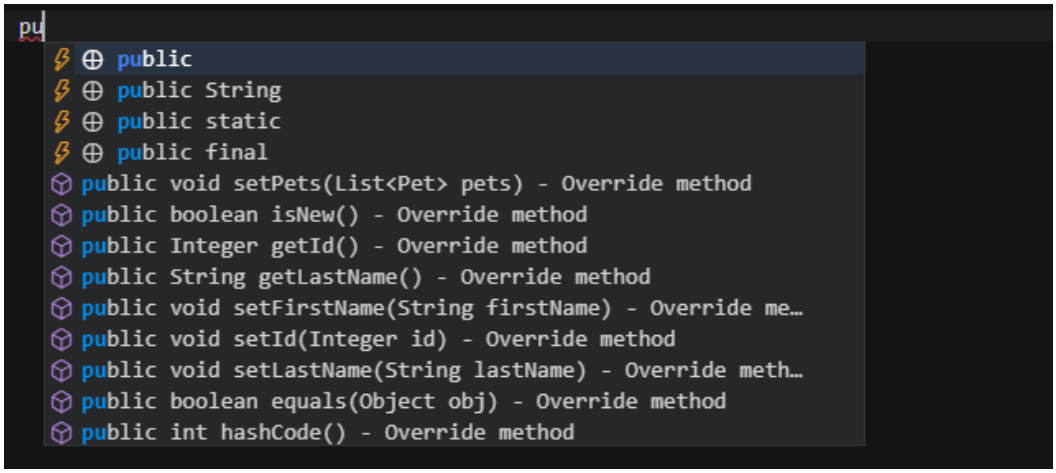
- 要插入选定的符号并替换当前位于光标位置的符号，请按“Tab”键。



- 要关闭代码推荐列表而不插入推荐代码，请按“Escape”。

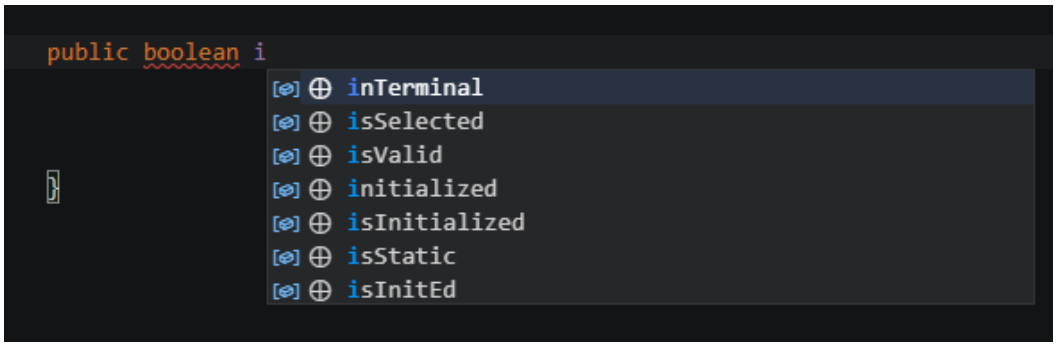
5.2.2.3 关键字补全

CodeArts IDE为Java保留关键字（如**public**、**void**、**if**、**while**、**boolean**等）提供代码补全推荐。

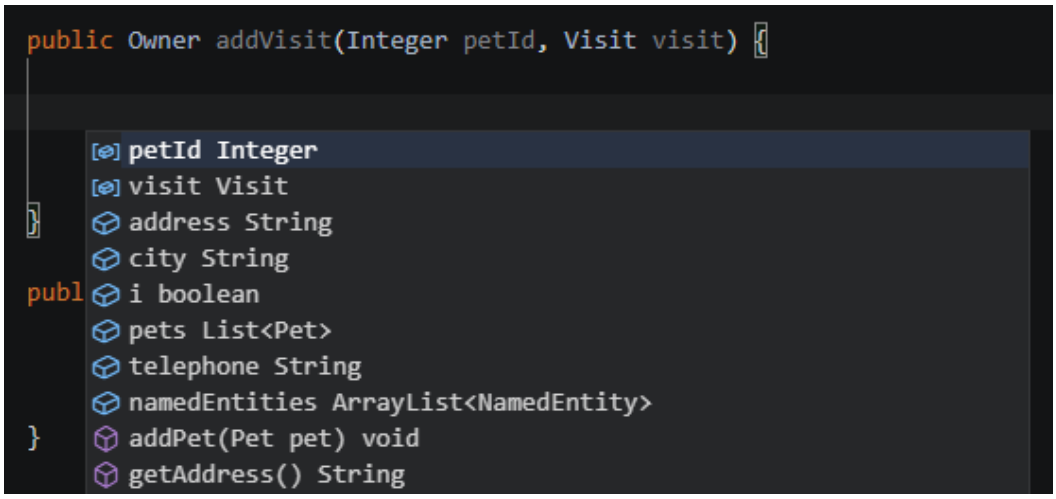


5.2.2.4 名字补全

基于当前上下文，CodeArts IDE为变量、参数、方法、类等提供了代码补全推荐。



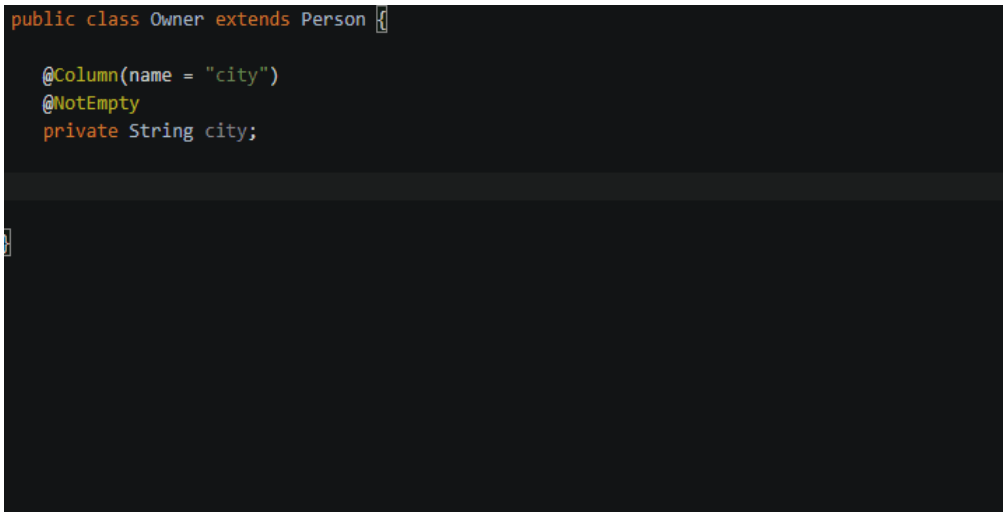
定义方法参数时，它们在代码补全推荐列表中被优先排序。



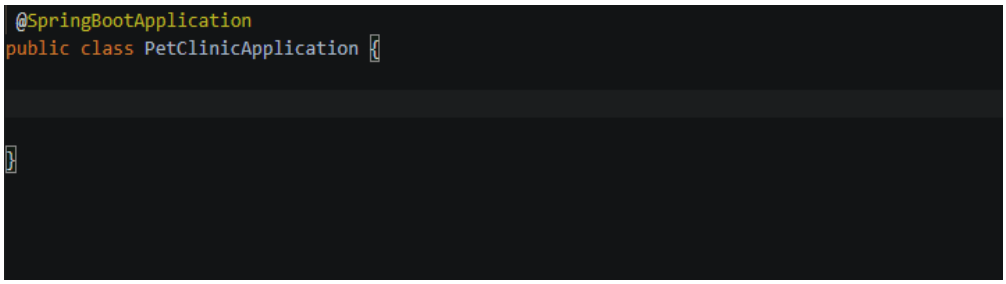
5.2.2.5 方法补全

CodeArts IDE为所需方法的元素提供代码补全：方法名称、返回值类型、参数和方法体。

- 在类内部，使用代码补全会根据类变量提供与变量相关方法（即getters/setters）的声明和主体。



- 在主项目类中，键入**m**并使用代码补全快速提供**main**的声明。



5.2.2.6 片段补全

CodeArts IDE为常用的代码结构提供上下文相结合的补全，包括条件语句和循环、折叠区域等。

5.2.2.7 智能类型匹配补全

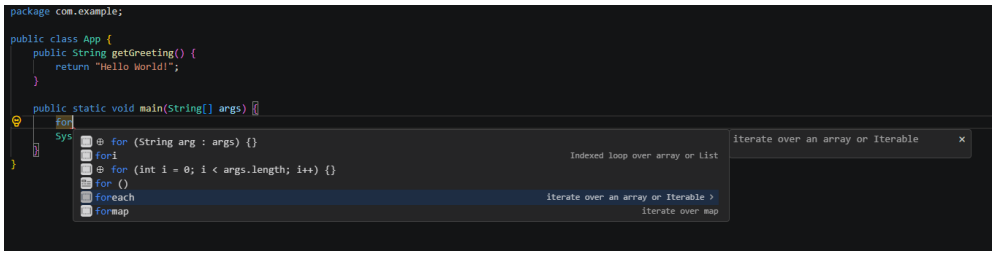
CodeArts IDE会自动过滤推荐列表，直到仅包括与当前上下文匹配的类型。

这在以下情况中有效：

- 在赋值语句的右侧。
- 在变量初始化中。
- 在**return**语句中。
- 在方法调用的参数列表中。
- 在对象实例化中**new**关键字后。
- 在链式表达式中。

5.2.3 代码片段

CodeArts IDE提供了多个Java代码片段补全，帮助您提高工作效率，如**class**/**interface**，**syserr**，**sysout**，**if**/**else**，**try**/**catch**，静态**main**方法等。此功能使用来自Java语言服务器的信息，您可以在选择代码期间预览代码段。



5.2.4 折叠区域

折叠区域允许您折叠或展开代码片段，以更好地查看源代码。在Java上下文中，使用以下折叠区域：

- 开始区域：**//#region**或**//<editor-fold>**。
- 结束区域：**//#endregion**或**//</editor-fold>**。

然后使用“Ctrl+Shift+[”/“Ctrl+-”/“Ctrl+Numpad-”（IDEA键盘映射）来折叠光标处最内层未折叠的区域，使用“Ctrl+Shift+]”/“”“Ctrl+=”/“Ctrl+Numpad+”（IDEA键盘映射）来展开光标处折叠的区域。有关代码折叠的更多详细信息，请参阅**代码折叠**。

5.2.5 智能选择

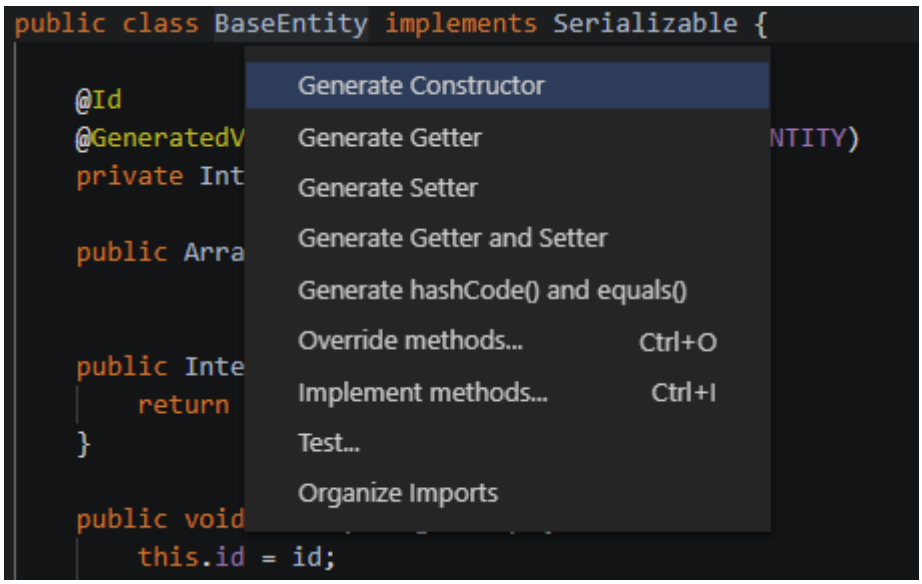
使用智能选择（语义选择），您可以根据代码中符号插入位置的语义信息扩大或缩小选择范围。

- 要扩大选区，请使用“Shift+Alt+right”。
- 要缩小选区，请使用“Shift+Alt+Left”/“Ctrl+Shift+W”(IDEA键盘映射)。

5.3 代码生成

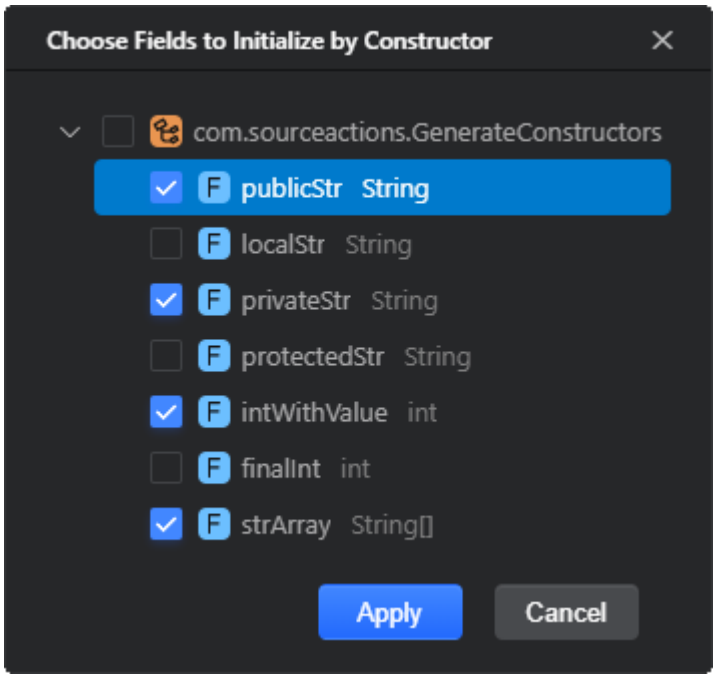
5.3.1 简介

CodeArts IDE提供了多个**Source Action**来生成公共代码结构和循环元素。要访问它们，请右键单击代码编辑器中的符号，然后从上下文菜单中选择**Source Action**。

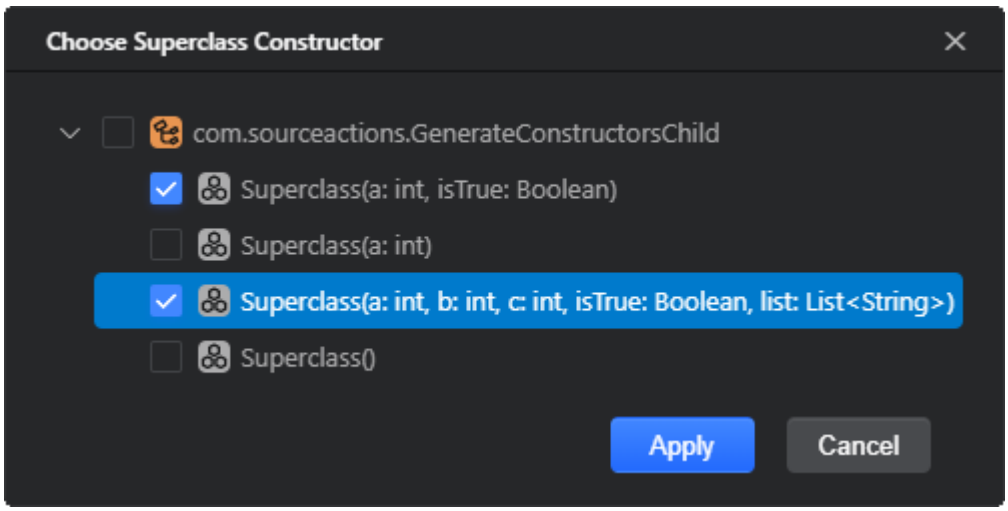


5.3.2 构造函数生成

使用此**Source Action**添加初始化选定类字段的类构造函数。

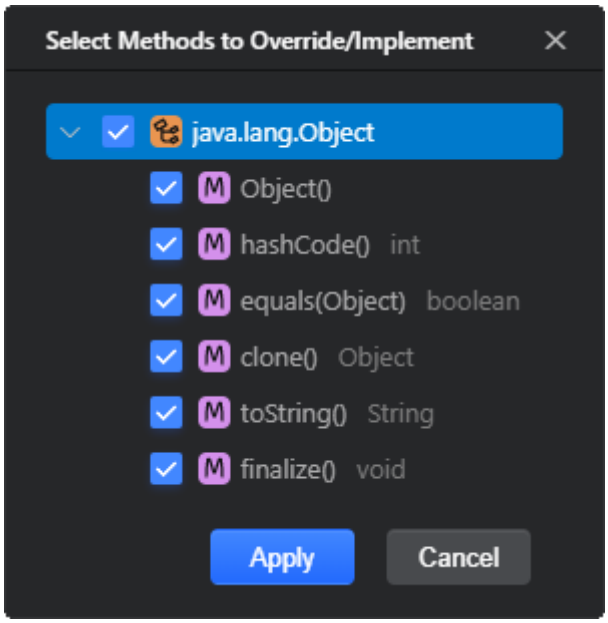


CodeArts IDE还将提示您选择超类的构造函数（如果有），以便在生成的构造函数中调用它。超类构造函数的参数与当前类的选定字段组合。



5.3.3 Override/implement 方法

使用这些源操作可以覆盖父类的选定方法（“Ctrl+O”）/实现接口或抽象类的方法（“Ctrl+I”）。因此，CodeArts IDE为Override/implement方法生成存根。

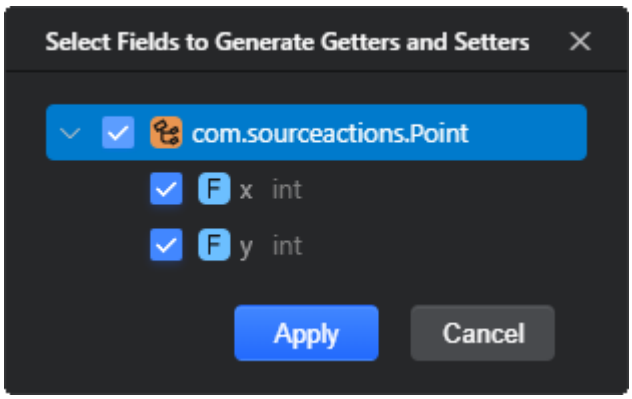


5.3.4 组织 imports

使用此**Source Action**可清理未使用的导入并自动排列**import**语句（“Shift+Alt+O”/“Ctrl+Alt+O”（IDEA键盘映射））。

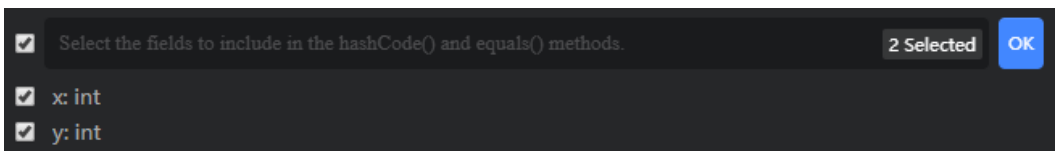
5.3.5 生成 getters 和 setters

使用此**Source Action**为类的选定字段生成getter和setter方法。



5.3.6 生成 hashCode()和 equals()

使用此**Source Action**生成具有默认实现的**hashCode()**和**equals()**方法存根。CodeArts IDE提示您选择类的非静态字段，这些字段用于确定对象相等/计算哈希代码值。



5.3.7 测试

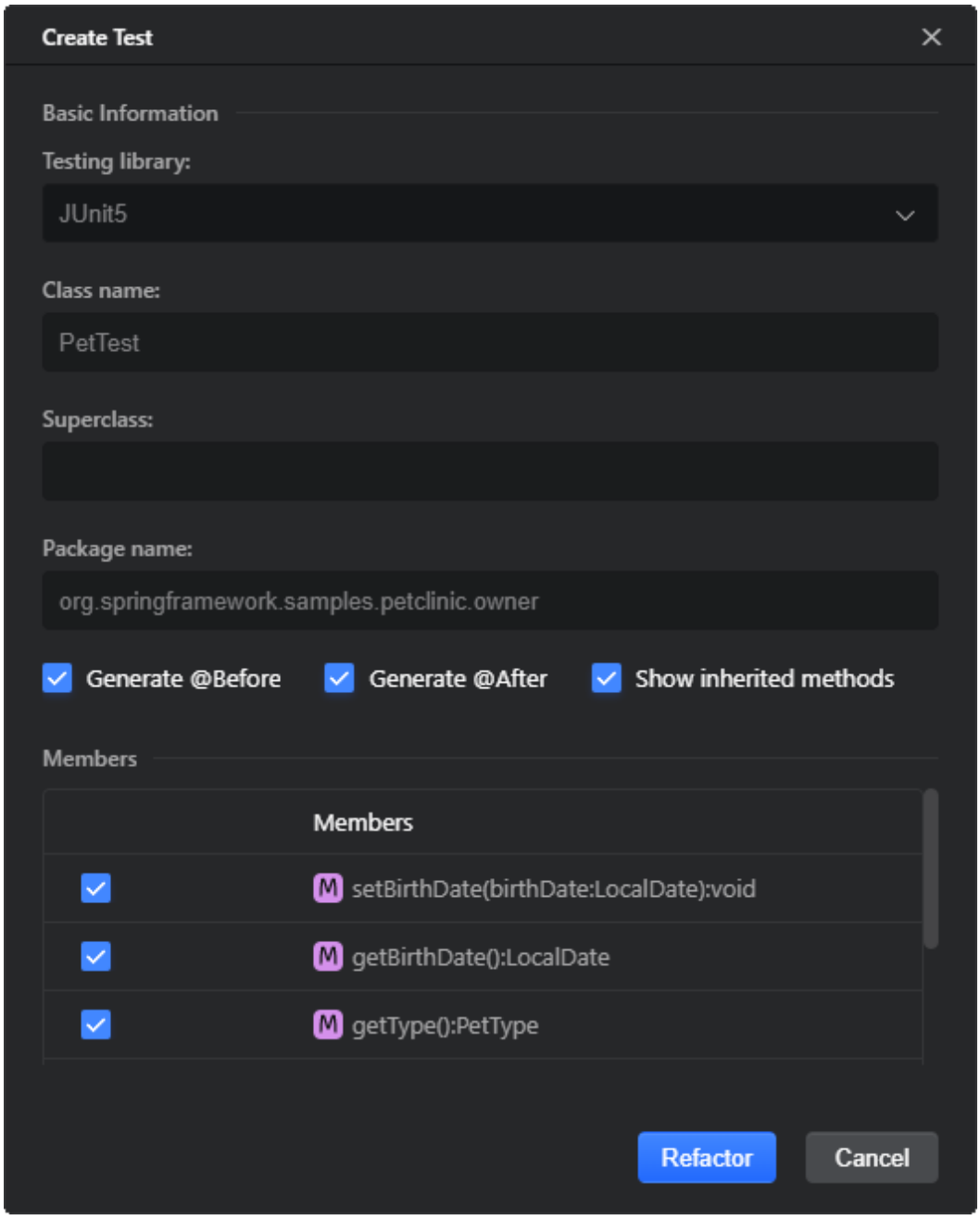
使用此**Source Action**为具有选定测试框架的生产类生成测试类。

说明

有关测试Java代码的更多详细信息，请参阅**调试**。

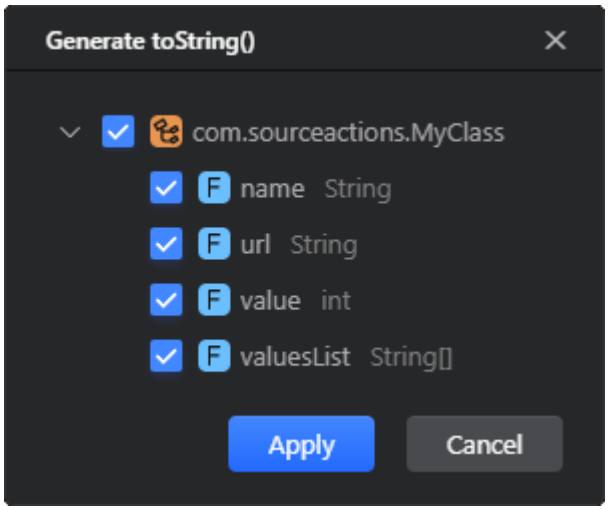
在**Create Test**对话框中，提供测试类参数：

- **Testing library**：选择要使用的测试库。
- **Class name**：提供测试类的名称，并根据选定的框架选择其超类。
- **Superclass**：指定存储生成的测试类的包。
- 选中**Generate @Before**/**Generate @After**复选框，让CodeArts IDE为测试夹具生成注释和存根方法。
- 在**Members**区域中，选择要为其生成测试方法的方法。要查看所有方法，包括继承的方法，请**Show inherited methods**复选框。



5.3.8 生成 toString()

使用此**Source Action**生成返回类的字符串表示形式的**toString()**方法。在**Generate toString()**对话框中，选择要在生成的**toString()**方法中返回的字段。



5.4 自动导入

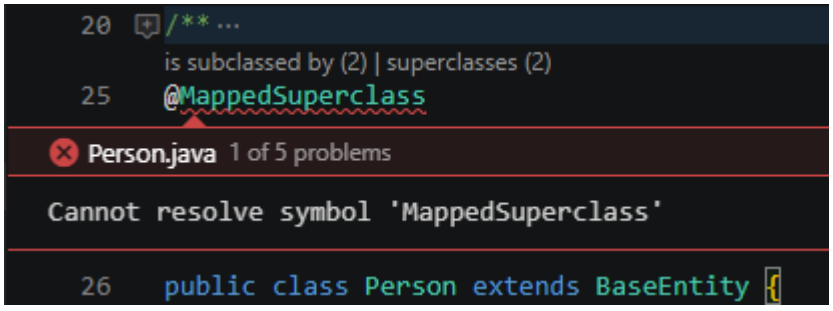
5.4.1 简介

如果您使用的是非导入类、静态成员、注释等，CodeArts IDE将帮助您添加相应的导入语句。此外，CodeArts IDE还可以重组和验证代码中的导入。

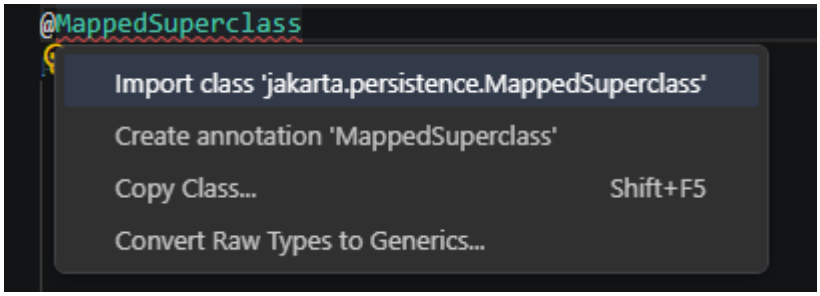
5.4.2 添加导入

当您键入包含对尚未导入元素的引用的代码块时，CodeArts IDE会自动插入缺少的导入语句。CodeArts IDE还突出显示当前缺少导入语句的符号，并提供了自动插入导入的**Source Action**。

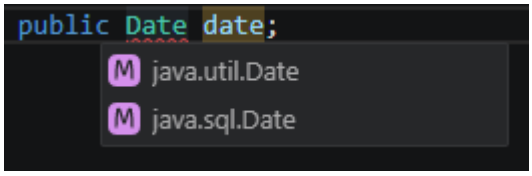
步骤1 在代码编辑器中，将光标定位在突出显示的未解析符号上。



步骤2 按“Ctrl+” /“Alt+Enter”键，然后在弹出菜单中选择**Import <symbol>**。



如果有多个可能的导入声明，请选择**Import class**，然后在弹出菜单中选择要导入的所需类。



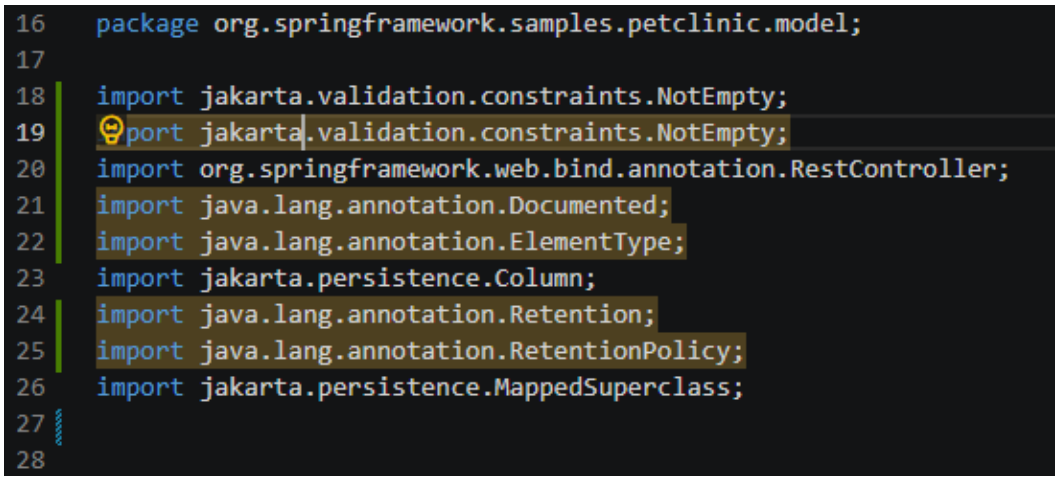
----结束

5.4.3 组织导入

CodeArts IDE突出显示模糊和未使用的导入，并提供**Source Action**来自动组织它们。

步骤1 在代码编辑器中，将光标定位在突出显示的import语句处。

步骤2 按“Ctrl+.”/“Alt+Enter”键，然后在弹出菜单中选择**Optimize imports**。CodeArts IDE删除不明确和未使用的导入，并按字母顺序对导入语句进行排序。

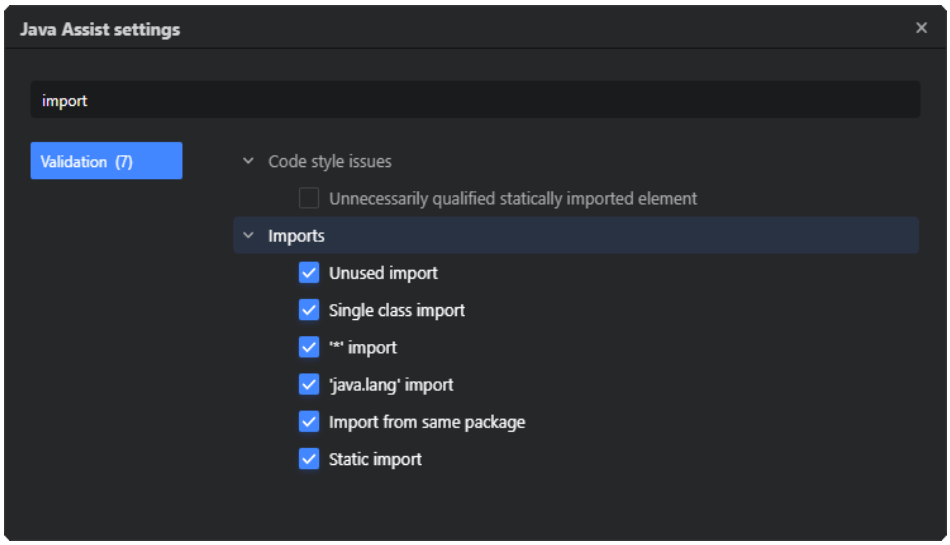


----结束

5.4.4 验证导入

CodeArts IDE提供了几个导入验证规则，您可以根据需要配置这些规则。

1. 通过执行以下任一操作打开**Java Assist Settings**对话框：
   - 单击CodeArts IDE状态栏中的**Java Smart Assist**。
   - 在命令面板中运行**SmartAssist:Open Settings**命令（“Ctrl+Shift+P”/“Ctrl Ctrl”）。
2. 在搜索字段中键入**import**以快速找到导入验证规则。然后配置如下：
   - 要启用或禁用规则，请使用其名称旁边的复选框。
   - 要调整相应代码问题在代码编辑器中突出显示的方式，请在**Severity**列表中选择所需的严重性级别。



说明

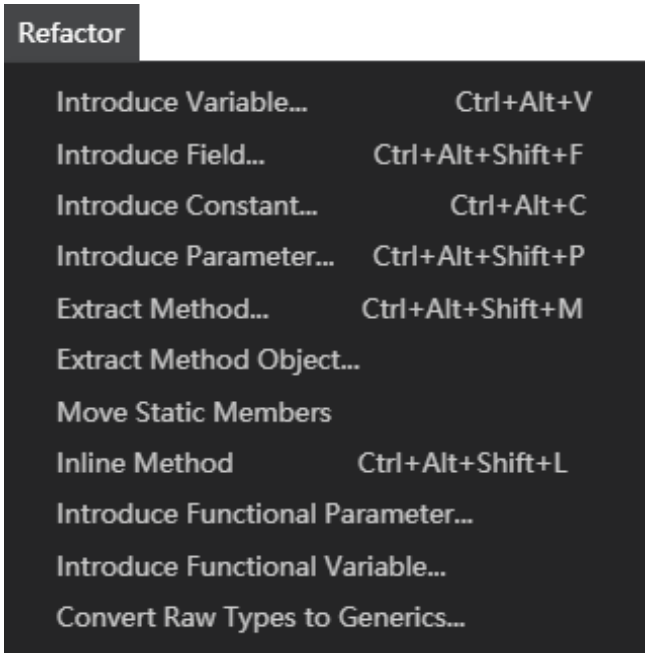
有关验证规则的更多详细信息，请参见**代码校验**。

5.5 重构

5.5.1 简介

Java程序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。选择您要重构的元素，右键单击它，然后从上下文菜单中选择**Refactor**。或者，在主菜单中，选择**Refactor**。



5.5.2 移动重构

5.5.2.1 简介

这些重构支持您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。

5.5.2.2 复制 Class

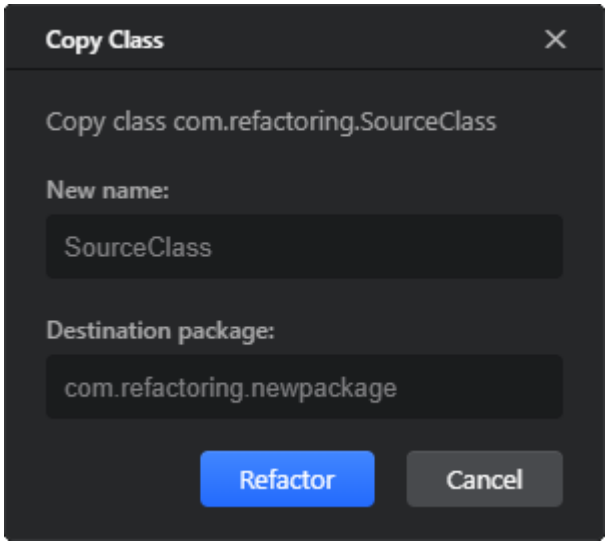
此重构支持您在不同的包中创建类的副本，维护正确的目录结构。

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。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor**>**Copy Class**或按“Alt+F6”/“F5”。

步骤3 在打开的**Copy Class**对话框中，提供重构参数。



步骤4 单击**Refactor**以应用重构。

----结束

示例

作为一个例子，让我们创建一个**Refactoring**类的副本，该类存储在包**com.refactoring.source**中。复制的类**RefactoringCopy**将存储在包**com.refactoring.target**中。

重构前

```
package com.refactoring.source;

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

```
    }
}
```

重构后

```
package com.refactoring.source;

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

package com.refactoring.target;

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

5.5.2.3 移动 Class

此重构允许您移动不同包中的类，维护正确的目录结构。

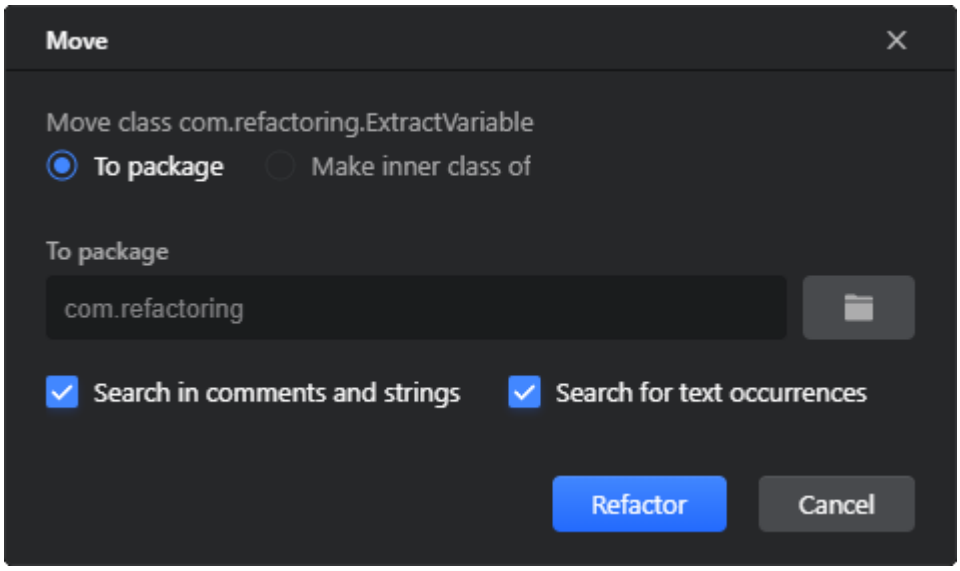
执行重构

步骤1 在代码编辑器中，将光标放在您想要移动的类上。

步骤2 在主菜单或上下文菜单中，选择**Refactor**>**Move Class** 或按“F6”。

步骤3 在打开的**Move**对话框中，提供重构参数。

- 要将类移动到不同的包中，请选择**To package**并在**To package**字段中提供目标包。单击浏览按钮（ ），在打开的**Choose destination package**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**Make inner class of**并在**Make inner class of**字段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，请选择**Search in comments and strings**和**Search for text occurrences**复选框。



步骤4 单击**Refactor**以应用重构。

----结束

示例

作为一个例子，让我们将存储在包**com.refactoring.source**中的类**Refactoring**移动到包**com.refactoring.target**中。

重构前

```
package com.refactoring.source;

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

重构后

```
package com.refactoring.target;

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

5.5.2.4 移动包

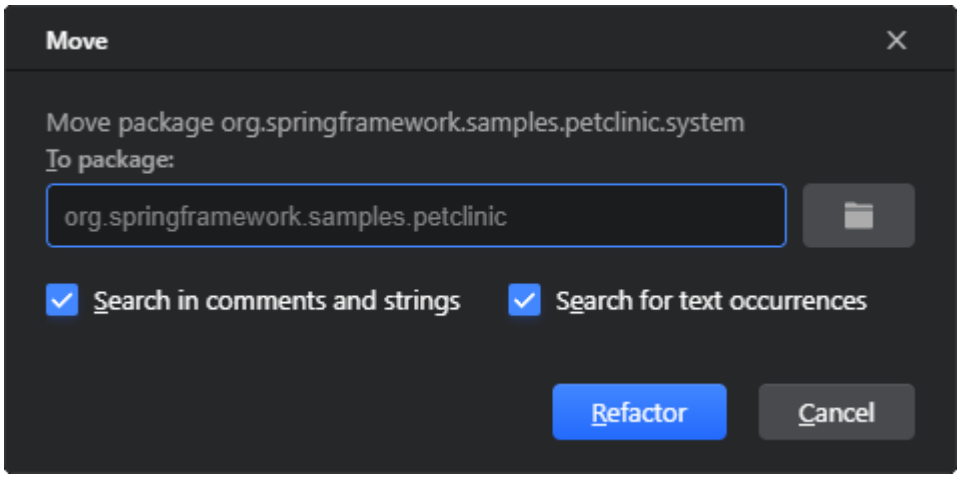
此重构允许您将包移动到不同的包中，以保持正确的目录结构。

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。或者，在**资源管理器**中，选择与所需软件包对应的目录。

步骤2 在主菜单或上下文菜单中，选择**Refactor**>**Move Package**。

步骤3 在打开的**Move** 对话框中，在**To package**字段中提供目标package。单击**浏览**按钮（ ），然后在打开的**Choose destination package**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**Search in comments and strings**和 **Search for text occurrences**复选框。



步骤4 单击**Refactor**以应用重构。

----结束

示例

作为一个例子，让我们将包**com.source.feature**移动到包**com.target**中，并替换代码中出现的**com.source.feature**包的位置。

重构前

```
package com.source.feature;

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

重构后

```
package com.target.feature;

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

5.5.2.5 移动内部类到上层

此重构支持您将内部类移至上层，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。

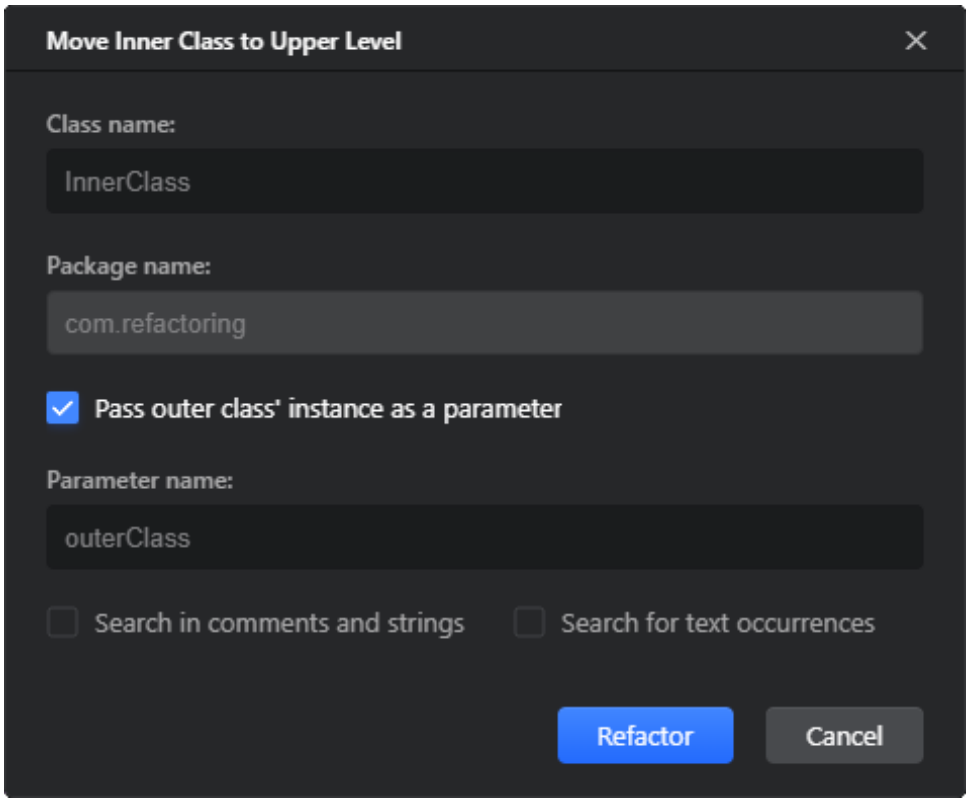
执行重构

步骤1 在代码编辑器中，将光标放在要移动到上层的类的声明位置。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Move Inner Class To Upper Level**。

步骤3 在打开的**Move Inner Class To Upper Level**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**Pass outer class' instance as a parameter**。
- 要在搜索代码中移动类的引用，请勾选**Search in comments and strings**和**Search for text occurrences**复选框 。



步骤4 单击 **Refactor**以应用重构。

----结束

示例

例如，让我们将**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

```
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

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

class InnerClass {
    private final OuterClass outerClass;

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

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

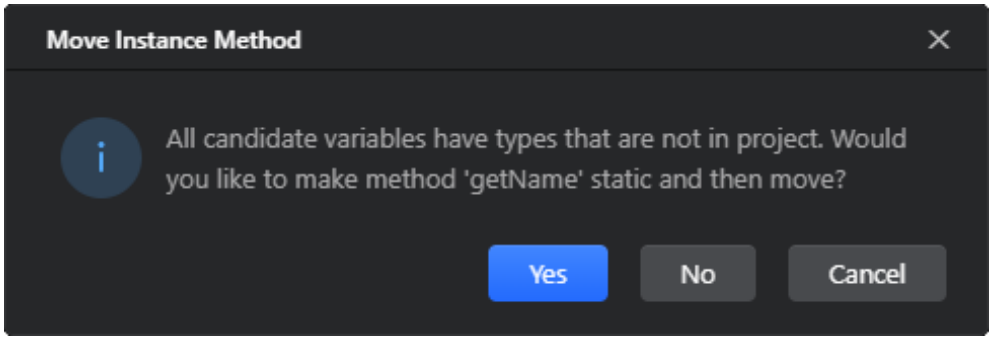
5.5.2.6 移动实例方法

如果此方法在项目中具有类型参数，则此重构允许您将实例（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。

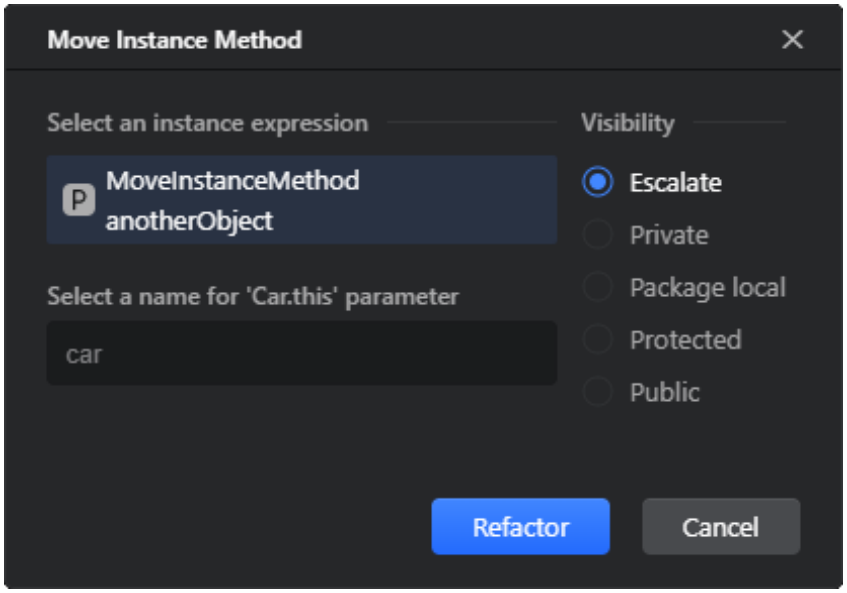
步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Move Instance Method**。



步骤3 在打开的**Move Instance Method**对话框中，提供重构选项。

- 在**Select an instance expression**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**Visibility**区域中，指定移动方法的可见性修改器，或选择**Escalate** 以自动将可见性设置为所需的级别。



步骤4 单击**Refactor**以应用重构。

----结束

约束与限制

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类。有关详细信息，请参见**使方法静态**和**移动静态成员**。

示例

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

重构前

```
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

```
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";
}
```

5.5.2.7 移动静态成员

此重构允许您将类的静态成员移动到不同的类中。

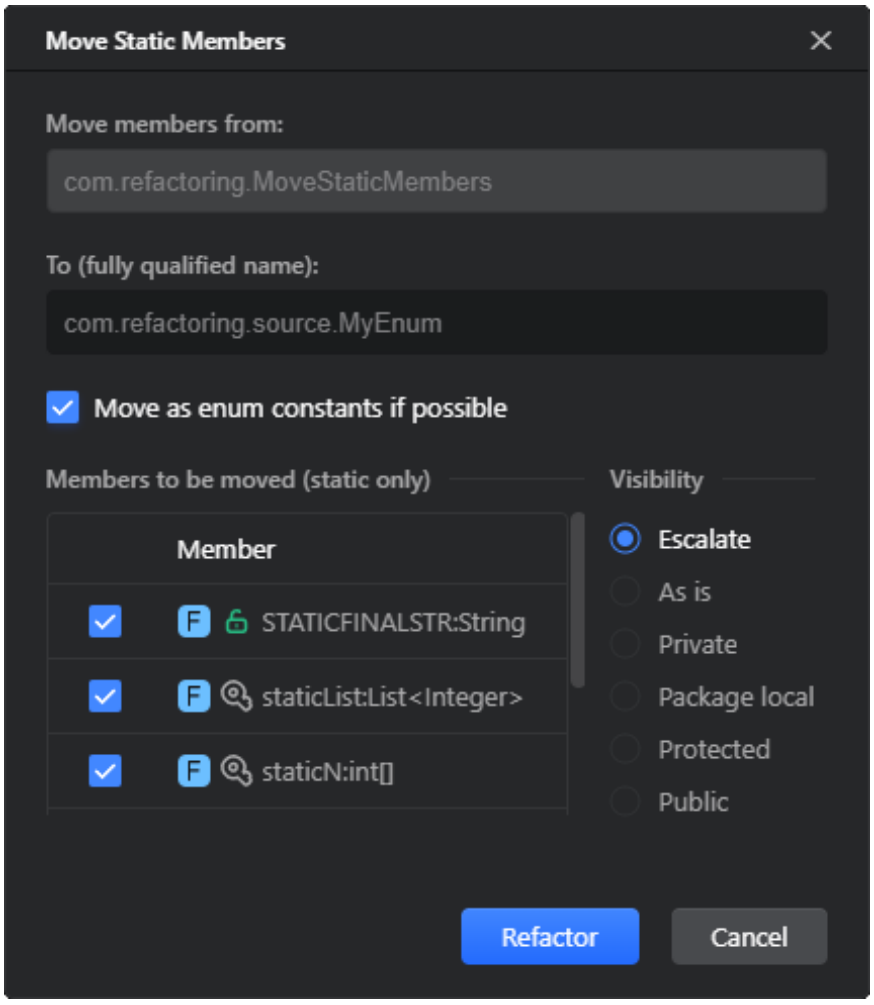
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Move Static Members**。

步骤3 在打开的**Move Static Members**对话框中，提供重构选项。

- 提供有效的目标类名称。
- 在**Members to be moved**列表中，选中要移动的静态成员复选框。
- 在**Visibility**区域中，指定移动的静态成员的可见性修改器，或选择**Escalate**以自动将可见性设置为所需的级别。
- 选中**Move as enum constant if possible**复选框，将常量（即**static final**字段）作为枚举常量移动到枚举类型。如果枚举类型具有类型参数的构造函数，则这是可能的。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们将类**MoveStaticMembers**的所有静态成员移动到枚举类**MyEnum**。由于**MyEnum**有一个带有**String**类型参数的构造函数，因此我们可以使用**Move as enum constant if possible**选项将**STATICFINALSTR**和**staticStr**字段移动为枚举常量。

重构前

```
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

```
class MoveStaticMembers {
    Boolean isTrue;

    void method() {
    }
}

enum MyEnum {
    STATICFINALSTR("TEST"), staticStr("test");
    static List<Integer> staticList;
    static int[] staticN;
    String typeName;

    MyEnum(String name) {
        typeName = name;
    }

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

    private static Boolean staticMethod2() {
        return true;
    }
}
```

5.5.2.8 向上/向下移动成员

Pull Members Up重构允许您将类成员移动到超类或接口。**Push Members Down**重构的作用恰恰相反，允许您将类成员移动到子类。

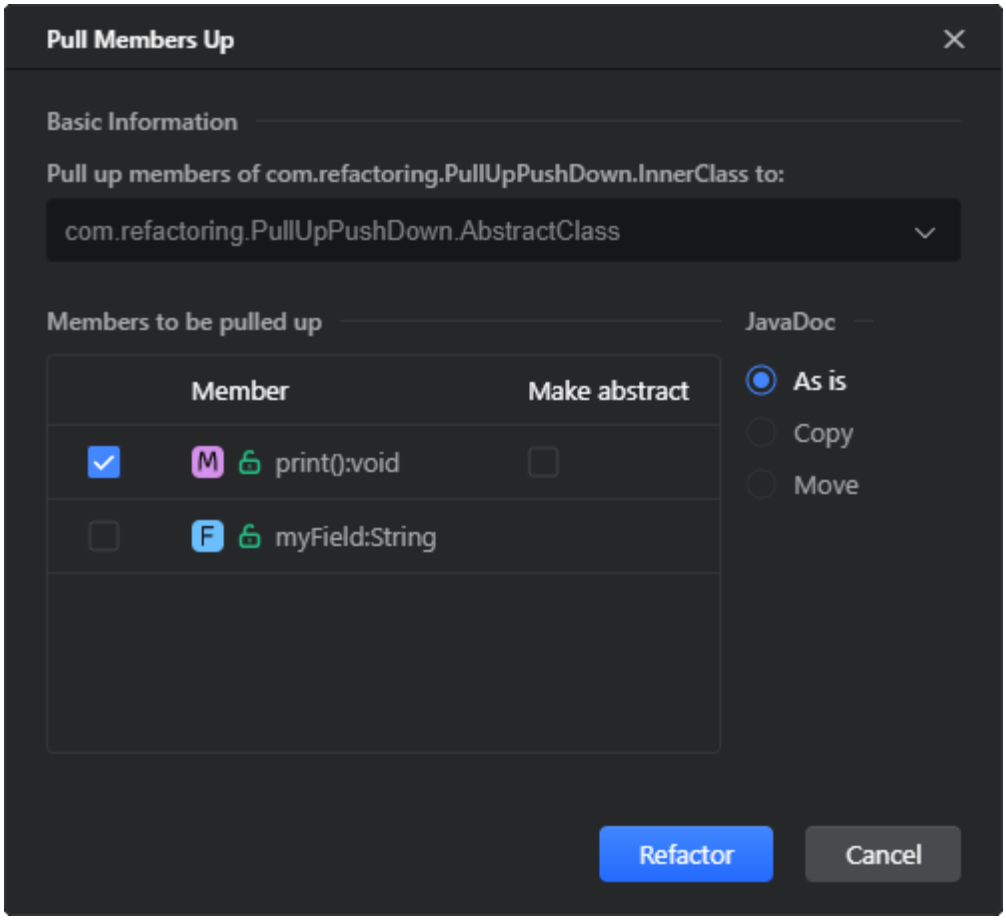
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Pull Members Up** / **Push Members Down**。

在打开的**Pull Members Up** / **Push Members Down**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**Make abstract**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。



- 单击**Refactor**以应用重构。

----结束

示例

作为一个例子，让我们从超类**AbstractClass**中提取字段**myField**和方法**print**的类层次结构。

重构前

```
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

```
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");
    }
  }
}
```

5.5.3 提取/引入重构

5.5.3.1 简介

这些重构允许您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。

5.5.3.2 引入变量

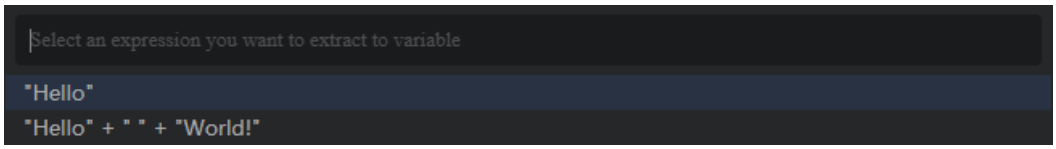
此重构允许您创建新变量，通过选定的表达式进行初始化，并使用创建变量的引用替换原始表达式。这与**内联变量**相反。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Introduce Variable**。或者按“Ctrl+Alt+V”。

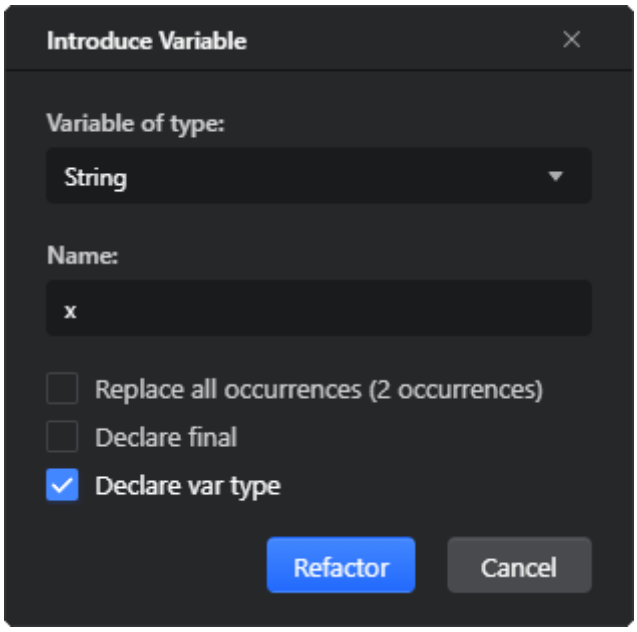
步骤3 如果多个表达式属于重构范围，请在弹窗中选择所需的表达式。



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

- 选择重构是否应用于所有找到的表达式，还是仅适用于当前表达式。

- 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及更高版本，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。这可能有助于提高代码的可读性。



步骤5 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
class ExtractVariable {

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

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

重构后

```
class ExtractVariable {

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

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



```
    }
}
```

5.5.3.3 引入参数

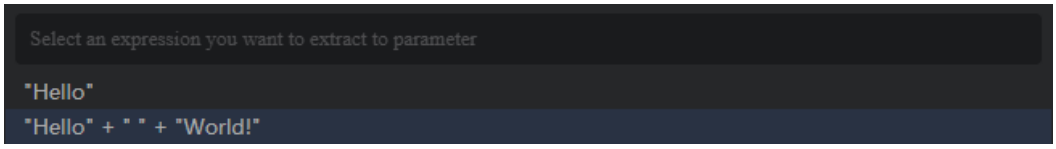
此重构允许您为方法声明引入新参数，以便在方法调用中，原始表达式被提供为方法参数。您还可以选择保留原始方法，或者使用创建的参数定义一个新方法。这与**内联参数**重构相反。

执行重构

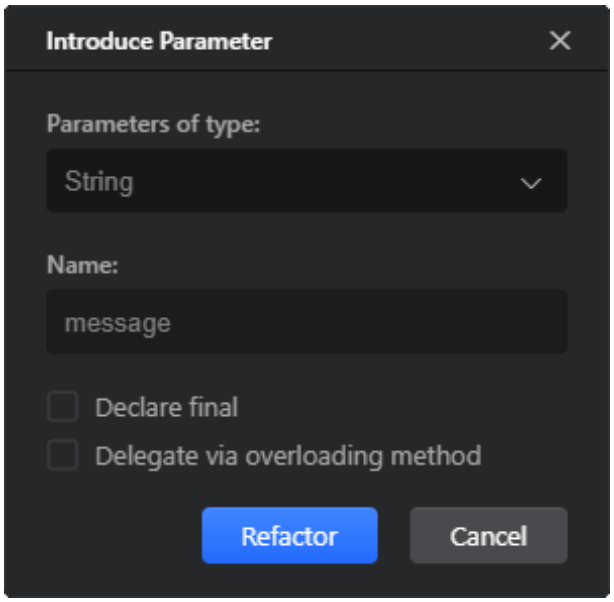
步骤1 在代码编辑器中，将光标放置在要提取到参数的表达式上。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Introduce Parameter**。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**Introduce Parameter**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**Delegate via overloading method**选项。



步骤5 单击**Refactor** 以应用重构。

----结束

示例

作为一个例子，让我们将表达式 "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

```
class ExtractParameter {

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

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

重构后

```
class ExtractParameter {

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

    private static void sayHello() {
        sayHello("Hello" + " " + "World!");
    }

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

5.5.3.4 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，并使用对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。

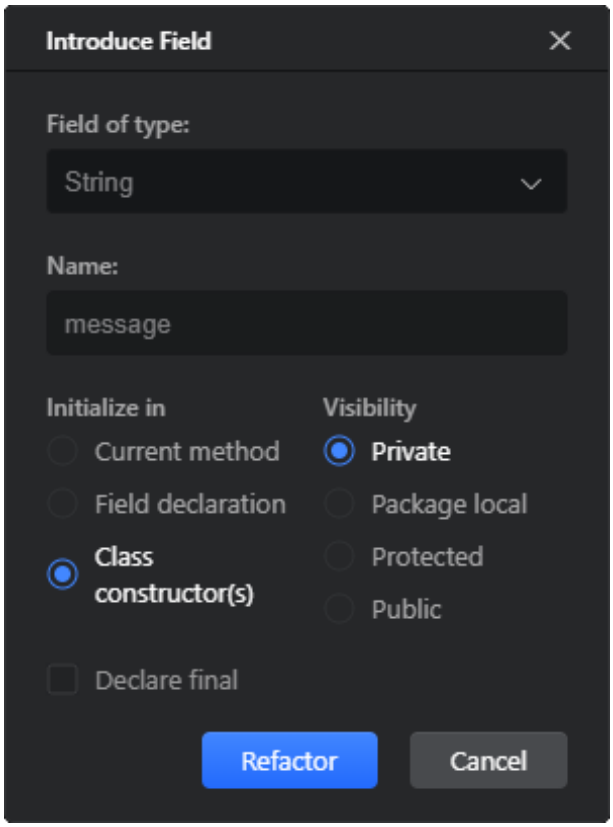
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor**>**Introduce Field**。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。

步骤4 在打开的**Introduce Field**对话框中，提供引入字段的类型和名称、其初始化和可见性选项，并选择是否应将字段声明为**final**字段。



步骤5 单击**Refactor**以应用重构。

----结束

示例

例如，让我们将表达式“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

```
class ExtractField {

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

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

重构后

```
class ExtractField {

    private static String message;

    public ExtractField() {
        message = "Hello" + " " + "World!";
    }
```



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

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

5.5.3.5 引入常量

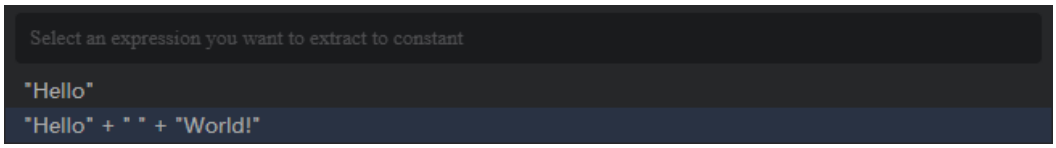
此重构允许您创建新常量，通过使用选定的表达式进行初始化，并使用创建常量的引用替换原始表达式。

执行重构

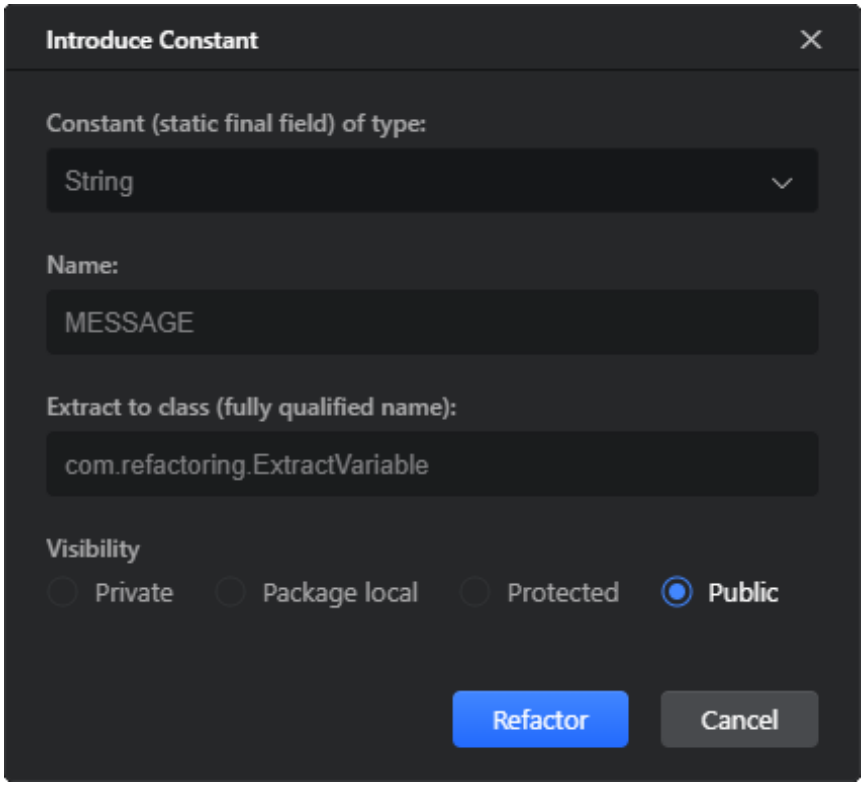
步骤1 在代码编辑器中，将光标放置在要提取到常量的表达式上。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Introduce Constant**。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。



步骤4 在打开的**Introduce Constant**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。



步骤5 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
class ExtractConstant {

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

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

重构后

```
class ExtractConstant {

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

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

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

5.5.3.6 提取方法

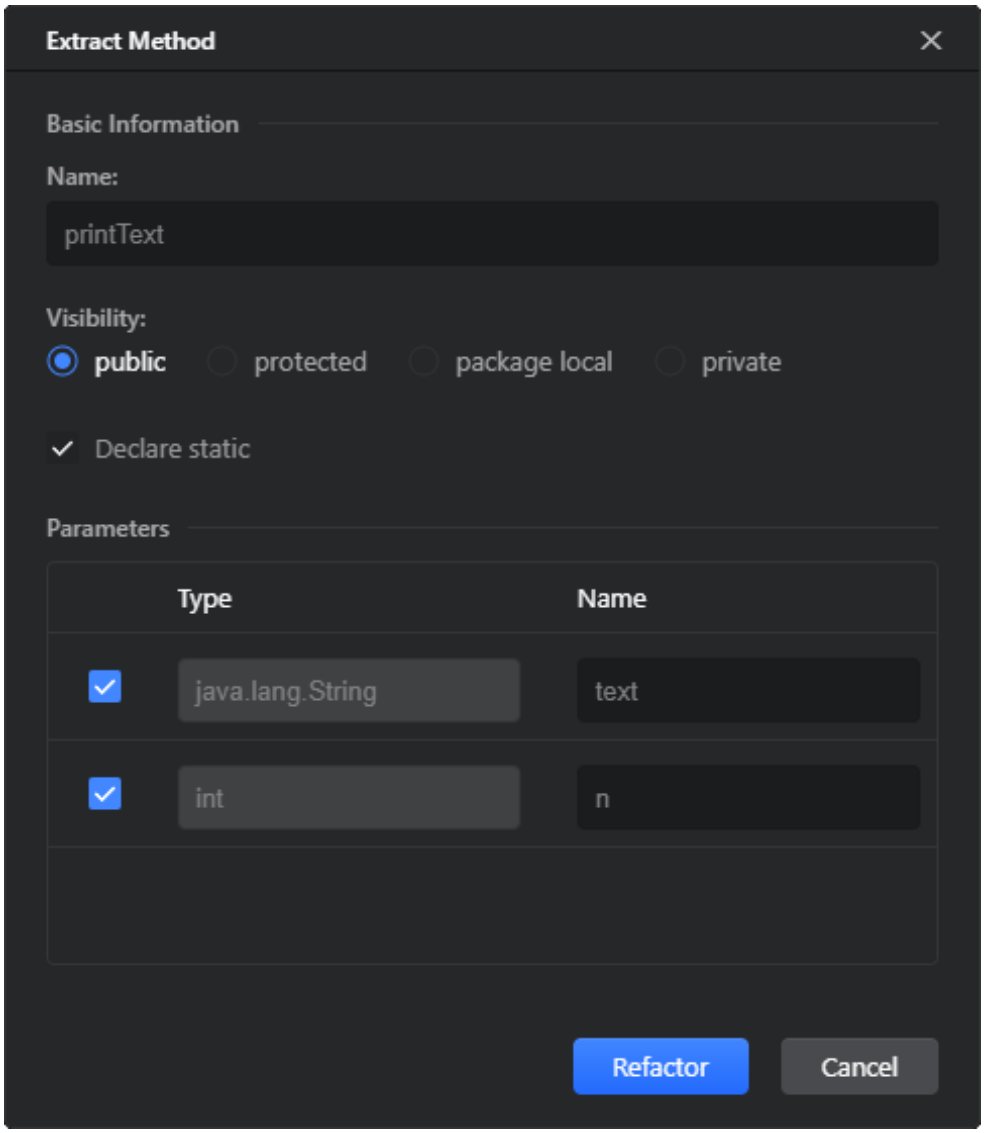
此重构允许您将任意代码片段移动到单独的方法中，并将其替换为对此新创建的方法的调用。这与**内联方法**相反。

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 在主菜单或编辑器上下文菜单中，选择**Refactor**>**Extract Method**，或按“Ctrl+Shift+Alt+M”。

步骤3 在打开的**Extract Method**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。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们将包含**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

```
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

```
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);
    }
  }
}
```

5.5.3.7 提取接口

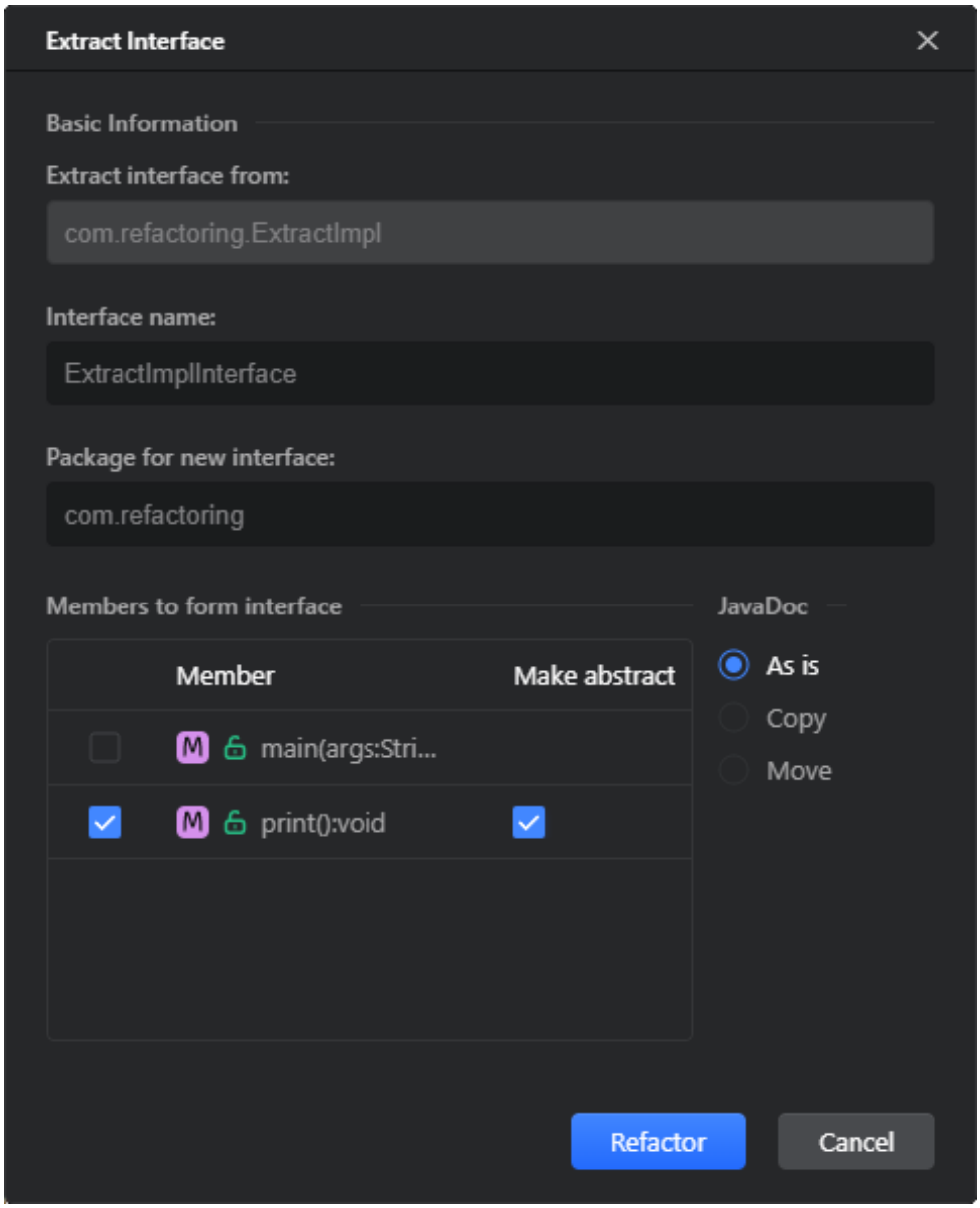
此重构允许您选定现有类的成员来创建接口，以使它们可以被其他类继承。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Extract Interface**。

步骤3 在打开的**Extract Interface**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。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们基于提取**ExtractImpl**类的**print**方法创建一个新的提取**ImplInterface**接口。

重构前

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

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



```
    }
}
```

重构后

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

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

public interface ExtractImplInterface {
  void print();
}
```

5.5.3.8 提取超类

此重构允许您选定现有类的成员创建新的超类。这与**内联超类**相反。

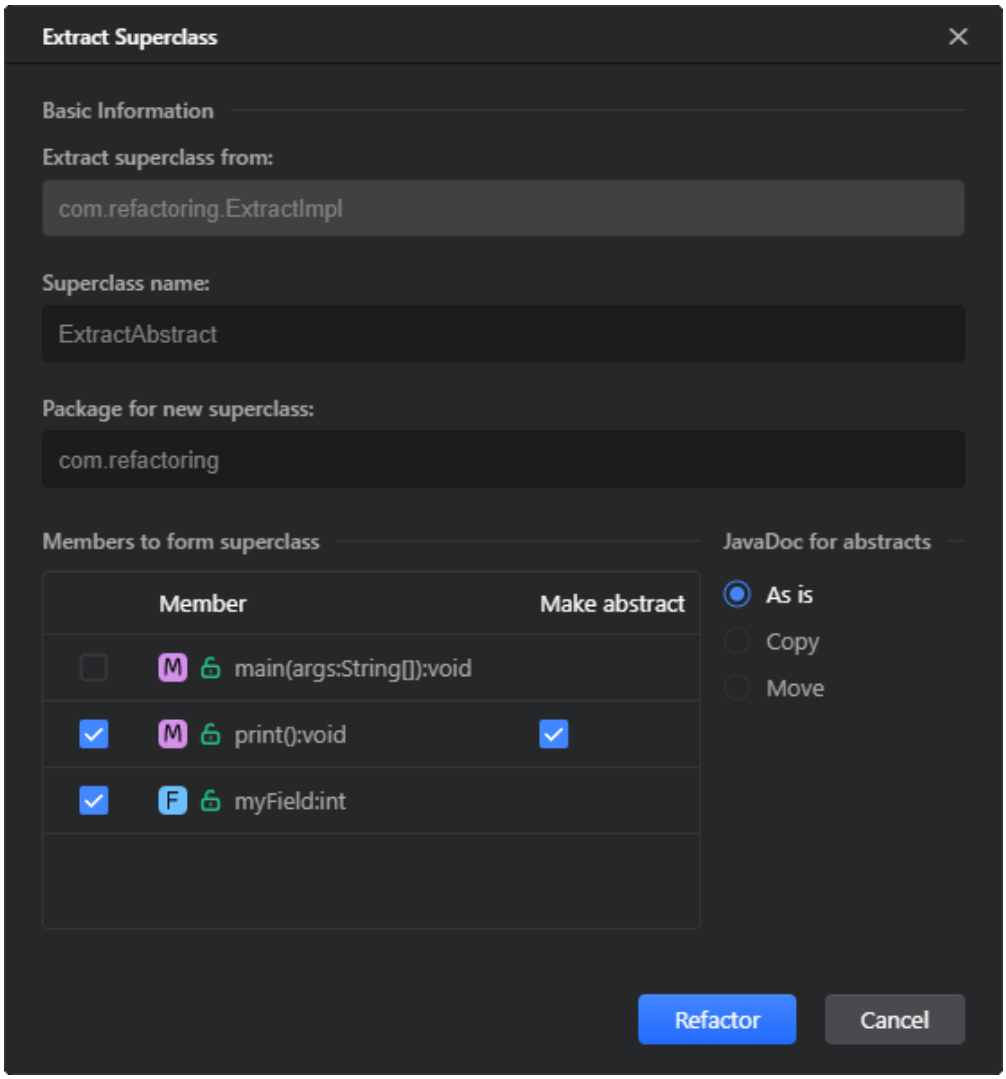
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Extract Superclass**。

步骤3 在打开的**Extract Superclass**对话框中，提供重构参数。

- 提供提取的超类名称和包。
- 在**Members to form superclass**区域中，选择要提取的类成员。对于方法，选中**Make abstract**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**JavaDoc for abstracts** 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。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们基于提取**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

```
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

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

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

public abstract class ExtractAbstract {
  public int myField;

  public abstract void print();
}
```

5.5.3.9 提取委托

此重构允许您基于现有类的成员选定来创建新类。

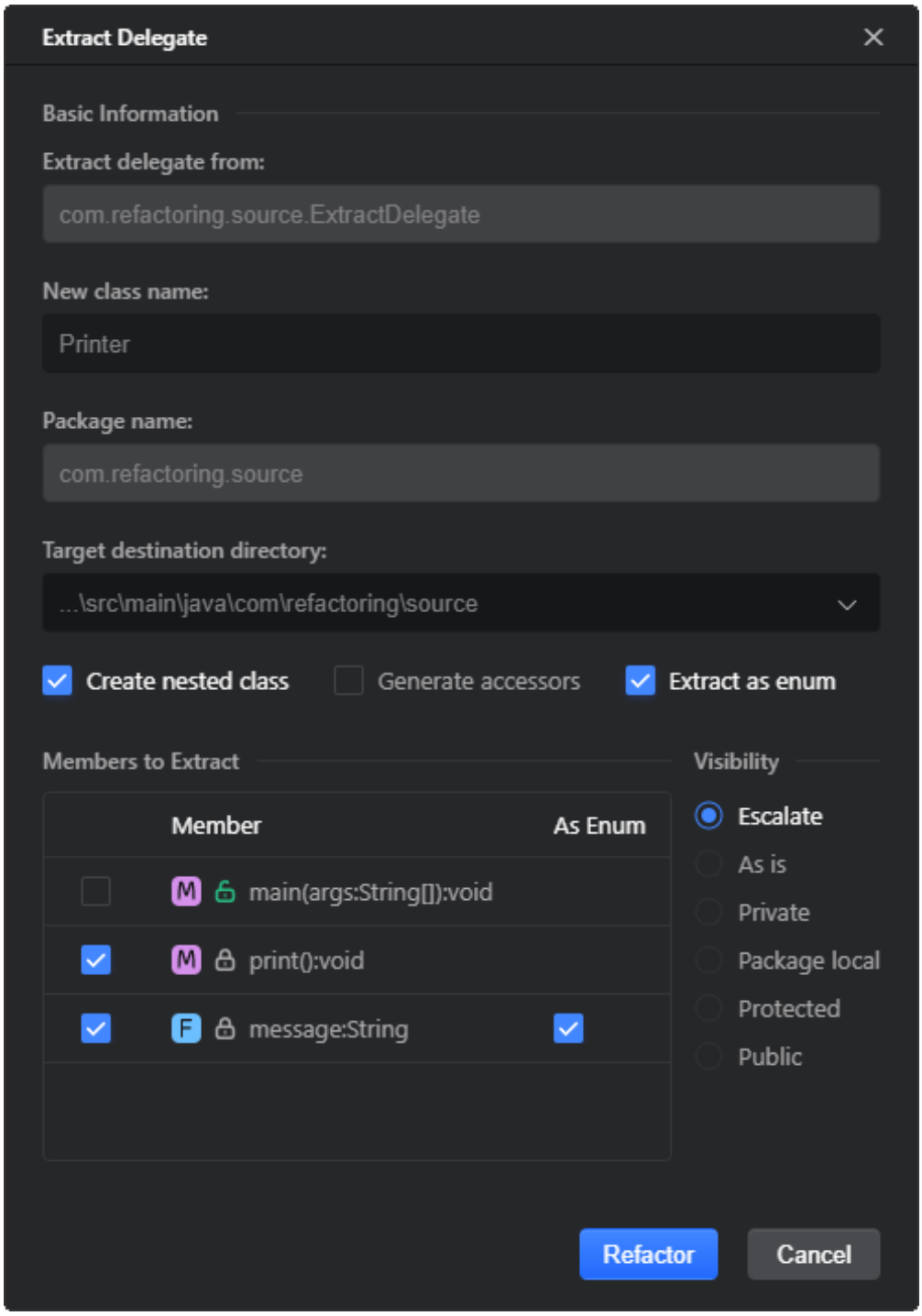
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Extract Delegate**。

步骤3 在打开的**Extract Delegate**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**Create nested class**复选框以在当前类中创建新类。
- 选中**Generate accessors**复选框，为提取的字段生成getter方法。
- 选中**Extract as enum**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 **Members to Extract**区域中，选择要提取的类成员。如果选择了**Extract as enum**，则还可以选择要作为枚举常量提取的字段旁边的**As Enum**复选框。
- 在**Visibility**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**Escalate**选项。



步骤4 单击**Refactor**以应用重构。

----结束

示例

作为一个例子，让我们将**ExtractDelegate**类中的**print**方法提取到一个嵌套的**Printer**枚举类中。

重构前

```
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

```
class ExtractDelegate {
    private final Printer printer = new Printer();

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

    private void print() {
        printer.print();
    }

    public enum Printer {
        message("Hello World!");
        private String value;

        public String getValue() {
            return value;
        }

        Printer(String value) {
            this.value = value;
        }

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

5.5.3.10 引入功能参数

此重构允许您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。

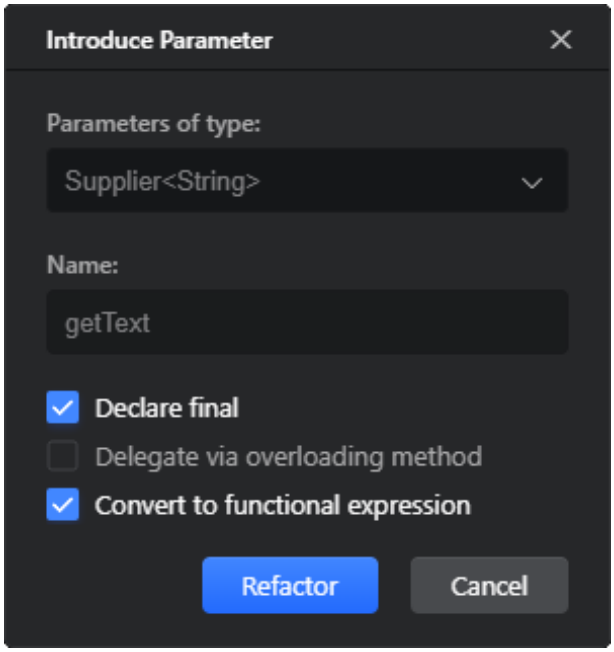
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Introduce Functional Parameter**。

步骤3 在打开的**Introduce Functional Parameter**对话框中，提供引入参数的名称和其他重构选项。

- 在**Parameters of type**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**Delegate via overloading method**选项。
- 要让CodeArts IDE生成函数表达式而不是匿名类，请选中**Convert to functional expression**复选框。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们提取表达式“**Hello World!**”**.toUpperCase()**作为**generateText**方法的参数。通过使用**Convert to functional expression**选项，我们可以通过两种方式完成：函数表达式或匿名类。

重构前

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

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

重构后(函数表达式)

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

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

```
        System.out.println(generateText(() -> "Hello World!"));
    }

    private static String generateText(Supplier<String> supplier) {
        return supplier.get().toUpperCase();
    }
}
```

重构后(匿名类)

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

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

    private static String generateText(Supplier<String> supplier) {
        return supplier.get();
    }
}
```

5.5.3.11 引入功能变量

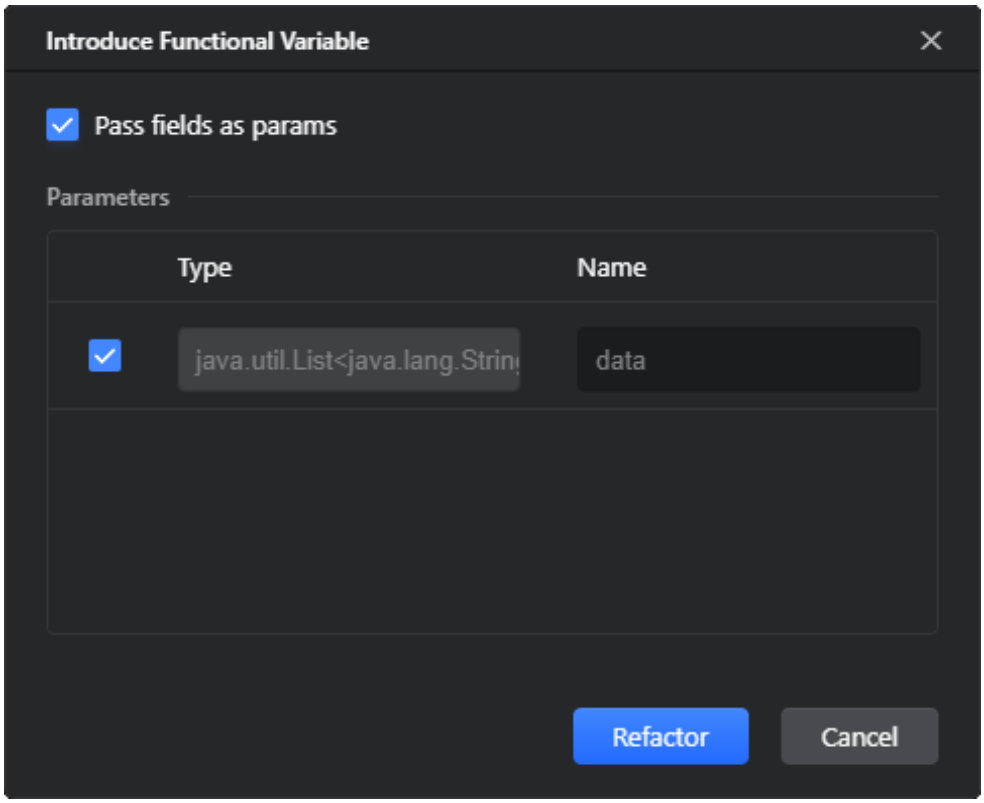
此重构允许将选定的表达式转换为新的函数类型变量或匿名类。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Introduce Functional Variable**。

步骤3 在打开的**Introduce Functional Variable**对话框中，选择**Pass fields as params**，以使实例字段作为参数传递到创建的函数表达式。



步骤4 单击 **Refactor**以应用重构。

----结束

示例

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

重构前

```
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

```
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));
    }
}
```

5.5.3.12 提取方法对象

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

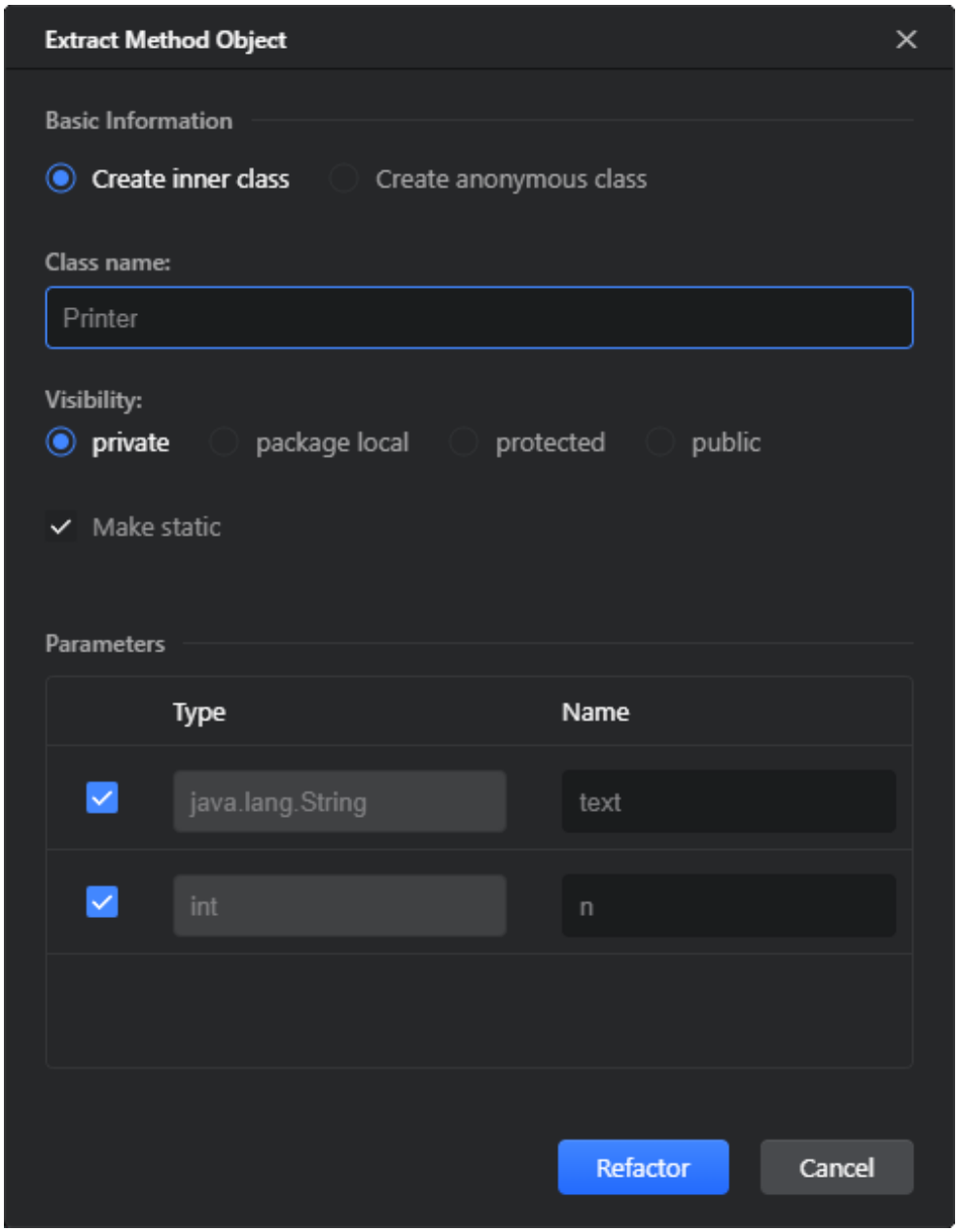
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Extract Method Object**。

步骤3 在打开的**Extract Method Object**对话框中，提供重构选项。

- **Create inner class**：选择以创建新的内部类。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。
- **Create anonymous class**：选择以创建新对象并提供要创建的方法的名称。
- 在**Parameters**区域中，在范围内选择变量作为方法参数。



步骤4 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
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

```
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);
      }
    }
  }
}
```

5.5.3.13 引入参数对象

此重构允许您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。

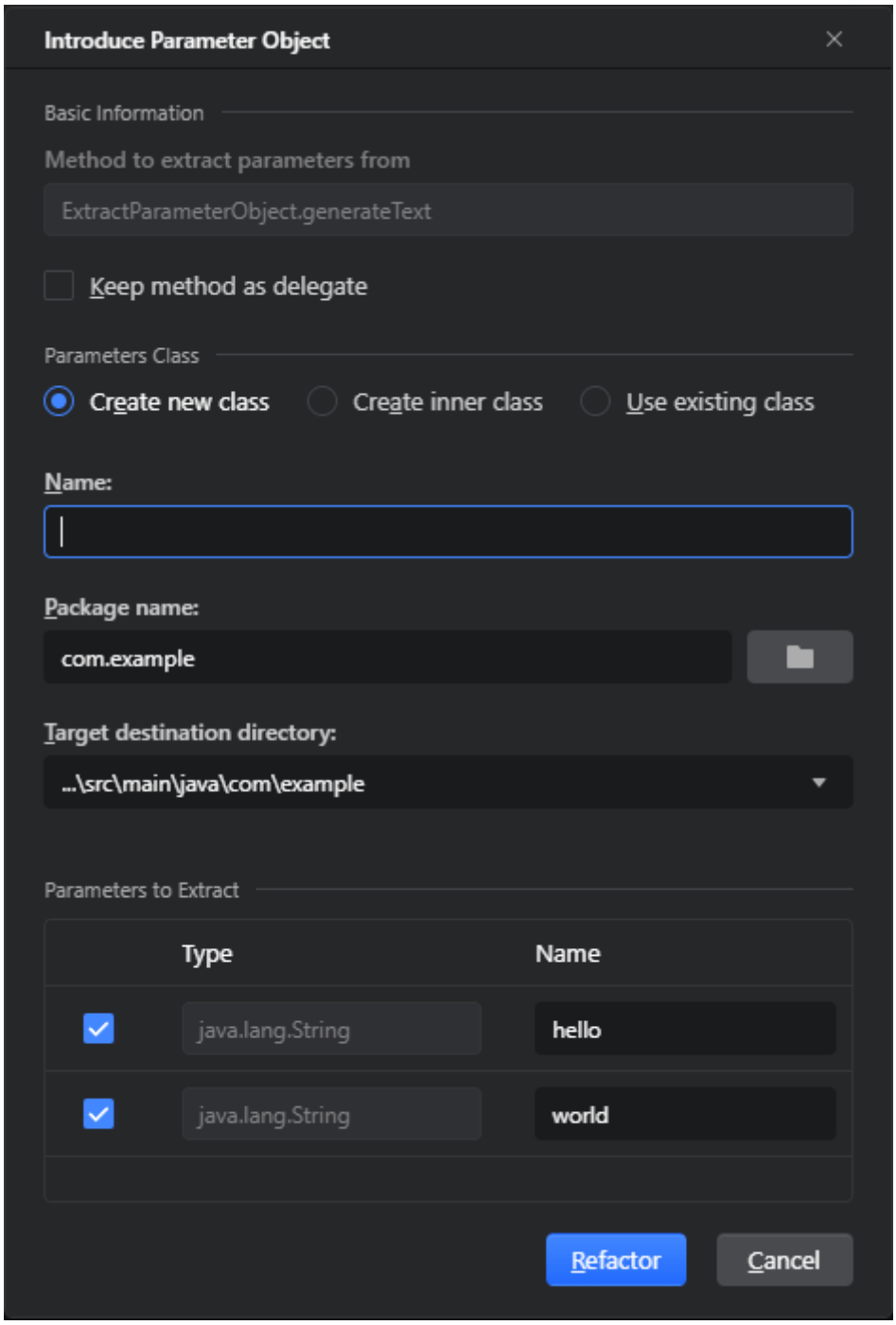
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Introduce Parameter Object**。

步骤3 在打开的**Introduce Parameter Object**对话框中，提供重构选项。

- **Keep method as delegate**：选择将原始方法保留为新创建方法的委托。
- **Parameters Class**： 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。
- **Name**：输入包装后参数的名称。
- **Parameters to Extract**： 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。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们将**hello**和**world**参数提取到**TextContainer**内部类，以使生成的**generateText**方法调用将包含对**TextContainer**对象的调用。

重构前

```
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

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

    private static String generateText(TextContainer textContainer) {
        return textContainer.getHello().toUpperCase() + textContainer.getWorld().toUpperCase();
    }

    private static class TextContainer {
        private final String hello;
        private final String world;

        private TextContainer(String hello, String world) {
            this.hello = hello;
            this.world = world;
        }

        public String getHello() {
            return hello;
        }

        public String getWorld() {
            return world;
        }
    }
}
```

5.5.4 内联重构

5.5.4.1 简介

这些重构允许您将变量、字段、方法等用法替换为其内容。这与**提取/引入重构**相反。

5.5.4.2 内联变量

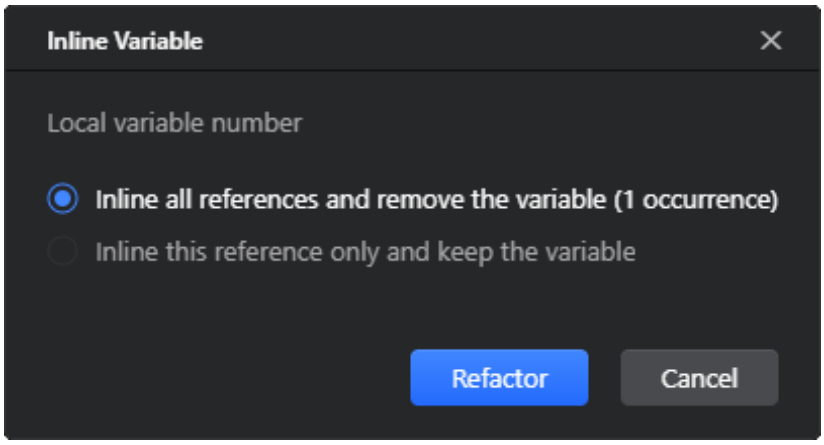
此重构允许您用变量的初始化器替换变量。这与**引入变量**相反。

执行重构

步骤1 在代码编辑器中，将光标放置在要内联其值的变量的用法上。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Inline Variable**，或按“Ctrl+Alt+N”。

步骤3 在打开的**Inline Variable**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。



步骤4 单击**Refactor**以应用重构。

----结束

约束与限制

如果在代码中修改了变量的初始值，则仅内联修改之前的用法。

示例

例如，让我们内联变量**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出现会受到重构的影响。

重构前

```
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

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

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

5.5.4.3 内联参数

此重构允许您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 在主菜单或编辑器上下文菜单中，选择**Refactor>Inline Parameter**。

----结束

示例

例如，让我们内联参数**pi**，将其替换为参数的值**Math.PI**。

重构前

```
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

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

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

5.5.4.4 内联方法

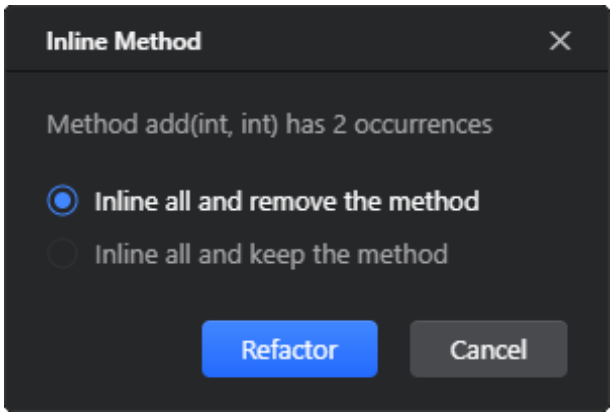
此重构允许您用方法的主体替换方法的用法。这与**提取方法**相反。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Inline Method**，或按“Ctrl+Shift+Alt+L”。

步骤3 在打开的 **Inline Method**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。



步骤4 单击**Refactor** 以应用重构。

----结束

示例

例如，让我们内联方法**add**，用方法的主体替换其调用。

重构前

```
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

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

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

5.5.4.5 内联字段

这个重构操作允许您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，将光标放在您想要内联其值的字段的声明或使用位置。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor**>**Inline Field**。

步骤3 在打开的**Inline Field**对话框中，选择是要内联所有字段的出现还是只内联当前位置的字段。

步骤4 单击 **Refactor** 以应用重构。

----结束

示例

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

重构前

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

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

重构后

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

5.5.4.6 内联超类

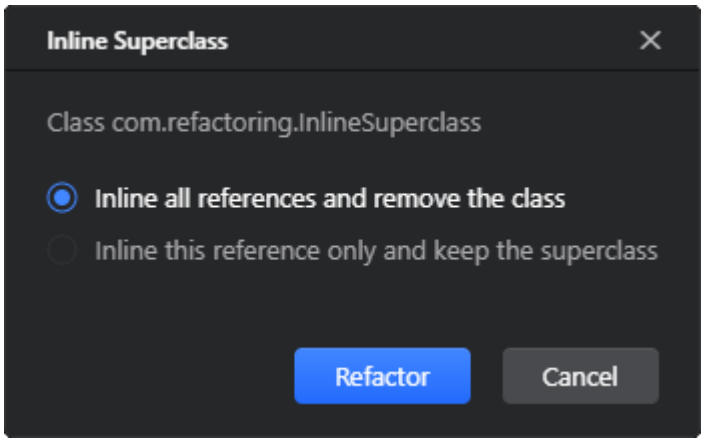
这个重构操作允许您将超类的成员移动到子类中，并删除超类。这与**提取超类**相反。

执行重构

步骤1 在代码编辑器中，将光标放在您想要内联的超类的声明或引用位置。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Inline Superclass**。

步骤3 在打开的 **Inline Superclass** 对话框中，选择是否在所有内联位置完成后保留超类。



步骤4 单击**Refactor**以应用重构操作。

----结束

示例

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

重构前

```
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

```
class SubClass {
    private int myValue;

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

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

5.5.4.7 内联到匿名类

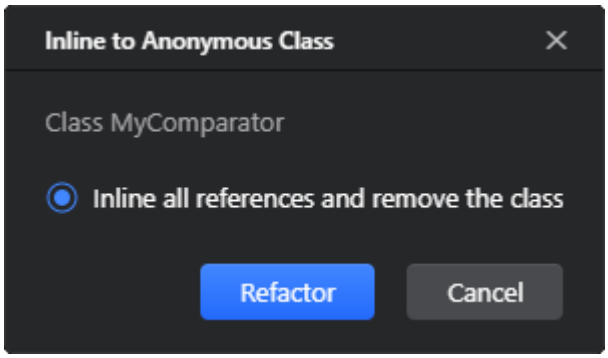
这个重构操作允许您用其内容替换多余的类。从Java 8开始，内联的匿名类可以自动转换为lambda表达式。

执行重构

步骤1 在代码编辑器中，将光标放在您想要内联为匿名类的类的声明位置。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor**>**Inline to Anonymous Class**。

步骤3 在打开的**Inline to Anonymous Class**对话框中，选择是否在所有内联位置完成后保留该类。



步骤4 单击 **Refactor**以应用重构操作。

----结束

示例

举个例子，让我们将类**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

```
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

```
class InlineAnonymousClazz {
    void sort(List<String> scores) {
```

```
        scores.sort((s1, s2) -> 0);
    }
}
```

5.5.5 使方法静态

此重构允许您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。

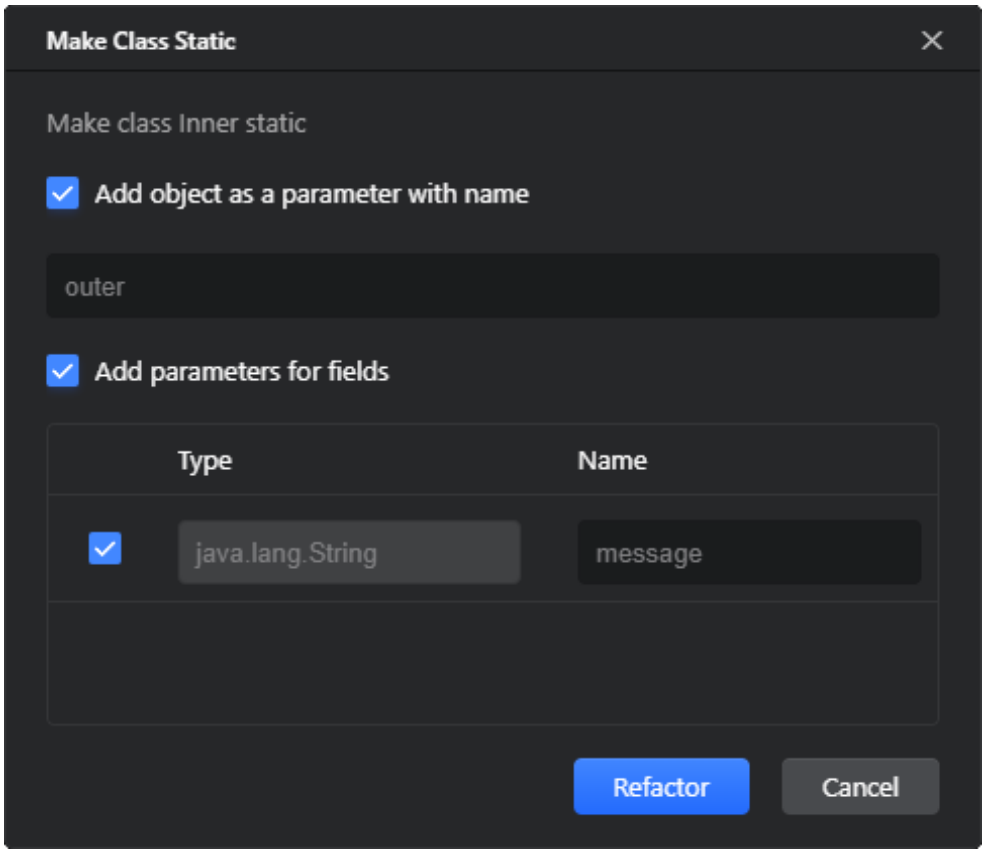
执行重构

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。

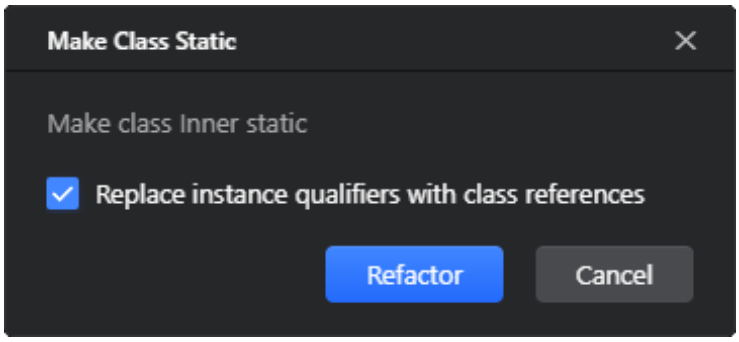
步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Make Static**。

步骤3 在打开的**Make Static**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。



- 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。



步骤4 单击**Refactor**以应用重构。

----结束

示例

作为一个例子，让我们将**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，我们可以将**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

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

  }

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

重构后

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

  }

  static class Inner {
    private Outer outer;
    private String message;

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

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

5.5.6 反转布尔值

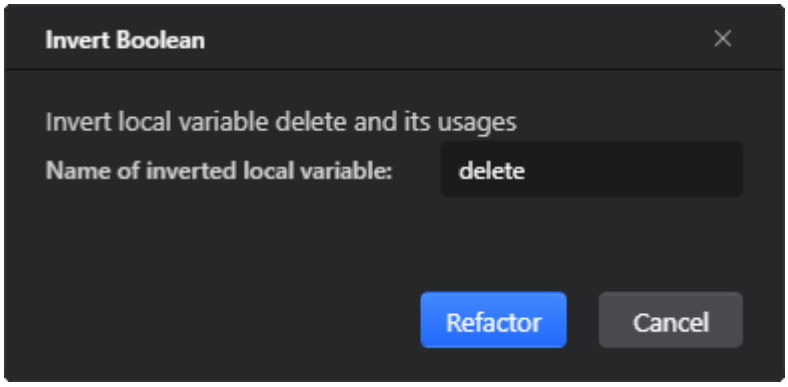
通过此重构，您可以反转布尔变量的值或方法的返回值。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Invert Boolean**。

步骤3 在打开的**Invert Boolean**对话框中，为反转变量或方法提供新名称。



步骤4 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
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

```
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;
    }
}
```

5.5.7 用委托替换继承

通过这种重构，您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。

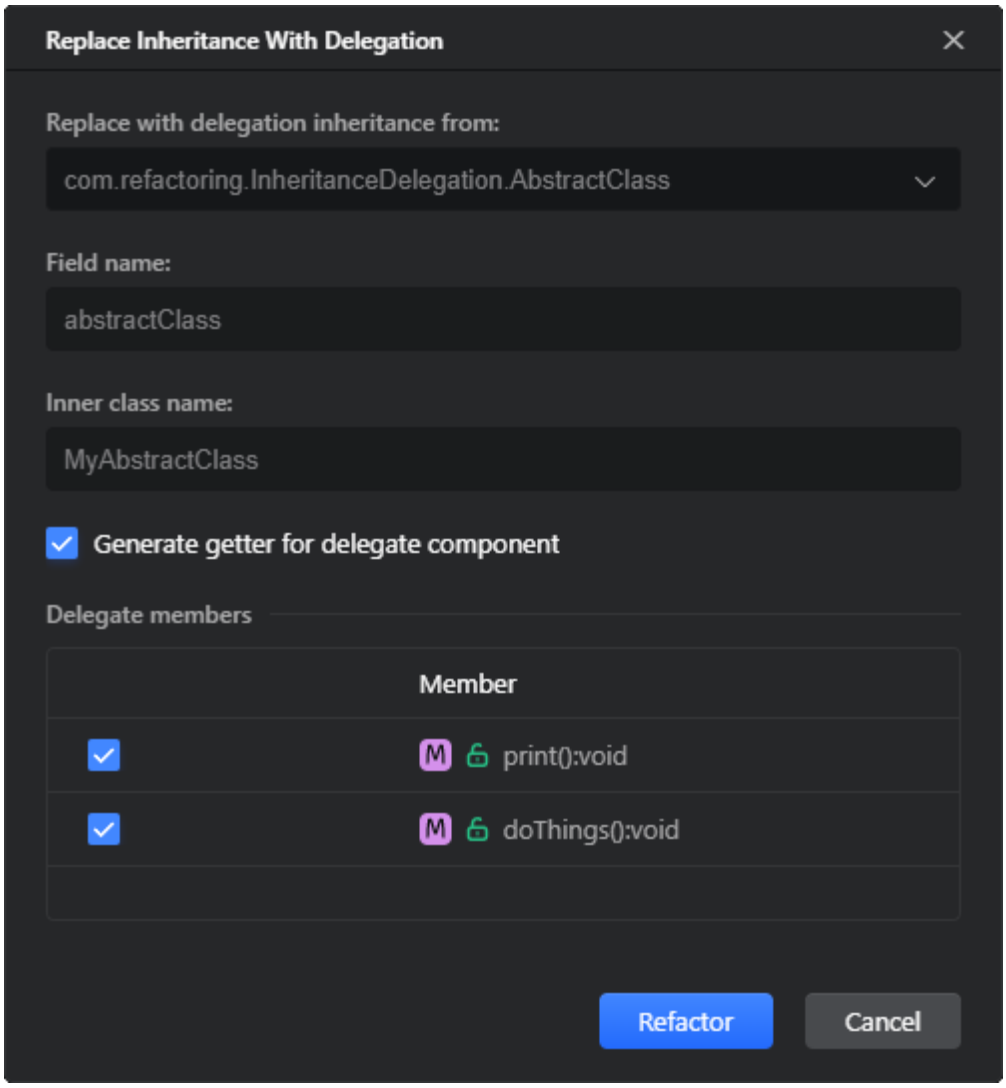
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Replace Inheritance With Delegation**。

步骤3 在打开的**Replace Inheritance With Delegation**对话框中，选择类，从中的继承将通过内部类替换为委托。提供重构选项：

- 在**Field name**字段中，指定新创建的内部类的字段名称。
- 在**Inner class name**字段中，指定新创建的内部类的名称。
- 选中**Generate getter for delegate component**复选框，为新创建的内部类创建getter方法。
- 在**Delegate members**选项中，选择要通过新创建的内部类委托的父类的成员。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们从**AbstractClass**中删除**InnerClass**的继承。因此，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

```
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

```
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();
    }
}
```

5.5.8 用工厂方法替换构造函数

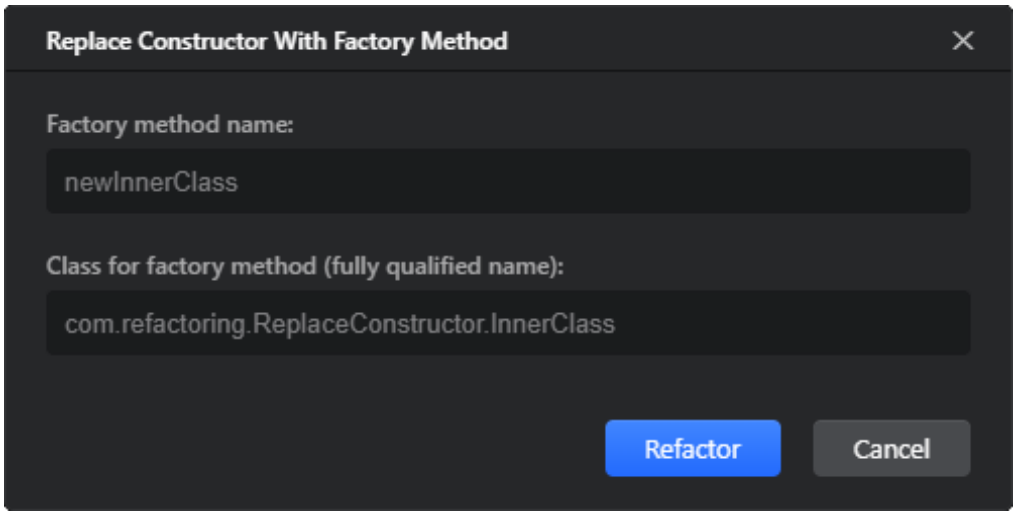
此重构允许您用返回类实例的工厂方法替换类构造函数。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Replace Constructor With Factory Method**。

步骤3 在打开的**Replace Constructor With Factory Method**对话框中，提供要创建的工厂方法的名称及其包含类。



步骤4 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
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

```
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);
    }
  }
}
```

5.5.9 用生成器替换构造函数

通过此重构，您可以将类构造函数的用法替换为对构建器类的引用。

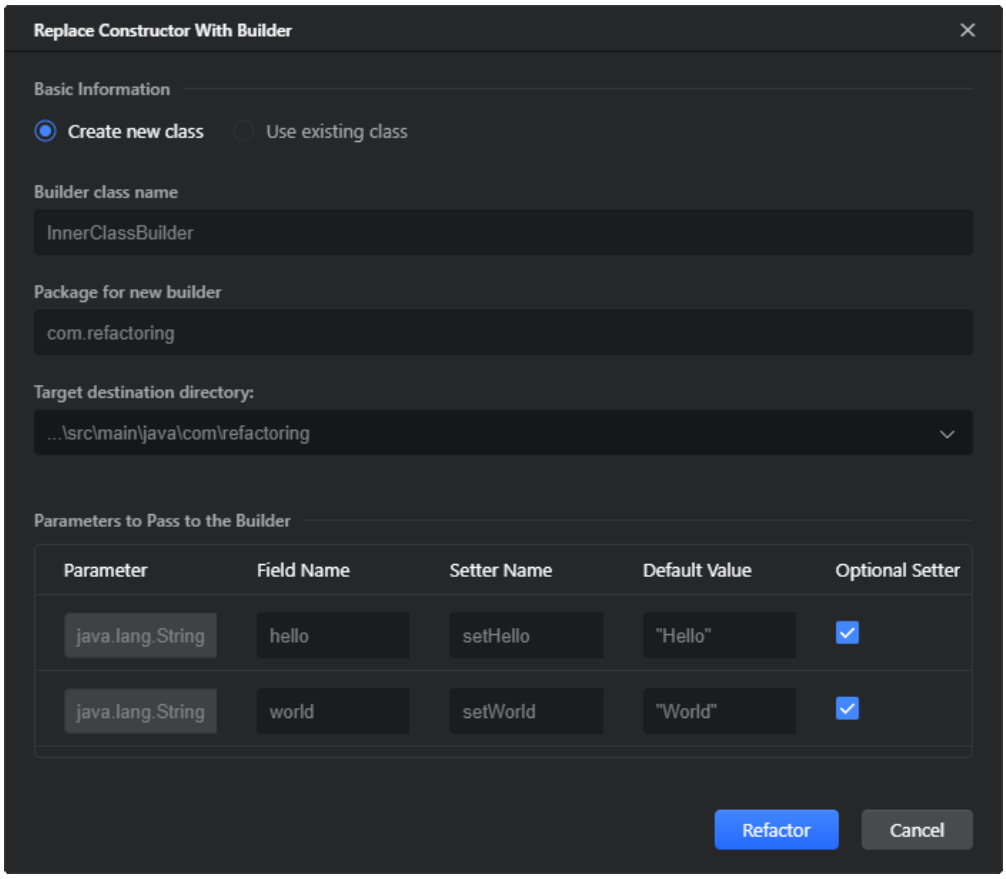
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Replace Constructor With Builder**。

步骤3 在打开的**Replace Constructor With Builder**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。
- 提供生成器类名，如果创建新类，则提供其包和目标目录。
- 在**Parameters to Pass to the Builder**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**Optional 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。



步骤4 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
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

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

  static class InnerClass {
    private String message;

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

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

public class InnerClassBuilder {
  private String hello = "Hello";
  private String world = "World";

  public InnerClassBuilder setHello(String hello) {
    this.hello = hello;
    return this;
  }

  public InnerClassBuilder setWorld(String world) {
    this.world = world;
    return this;
  }

  public ReplaceConstructor.InnerClass createInnerClass() {
    return new ReplaceConstructor.InnerClass(hello, world);
  }
}
```

5.5.10 封装字段

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

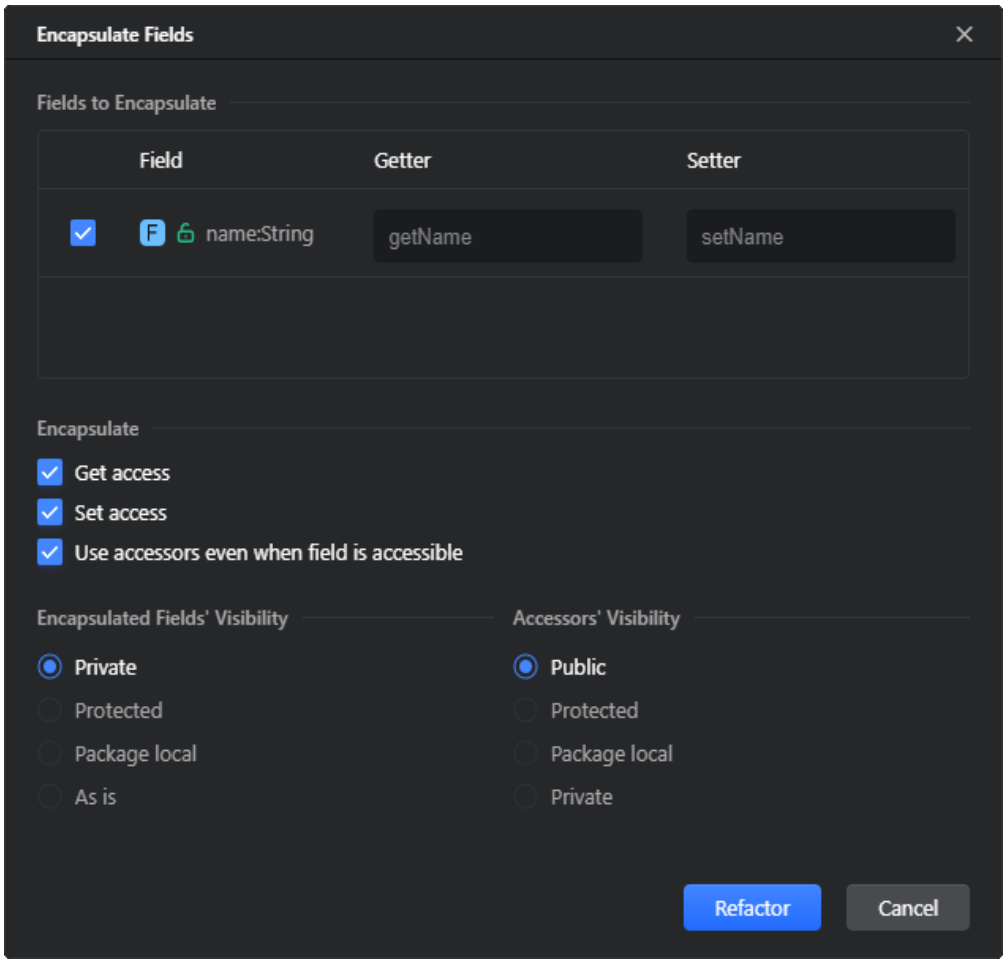
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Encapsulate Fields**。

步骤3 在打开的**Encapsulate Fields**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，选择是同时创建或单独创建getter、setter。选中**Use accessor even when field is accessible**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。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们封装**name**变量，并为其生成getter和setter方法。

重构前

```
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

```
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;
    }
}
```

5.5.11 更改方法签名

此重构允许您修改方法：重命名方法、添加异常以及添加、删除、重新排序和重命名方法的参数。

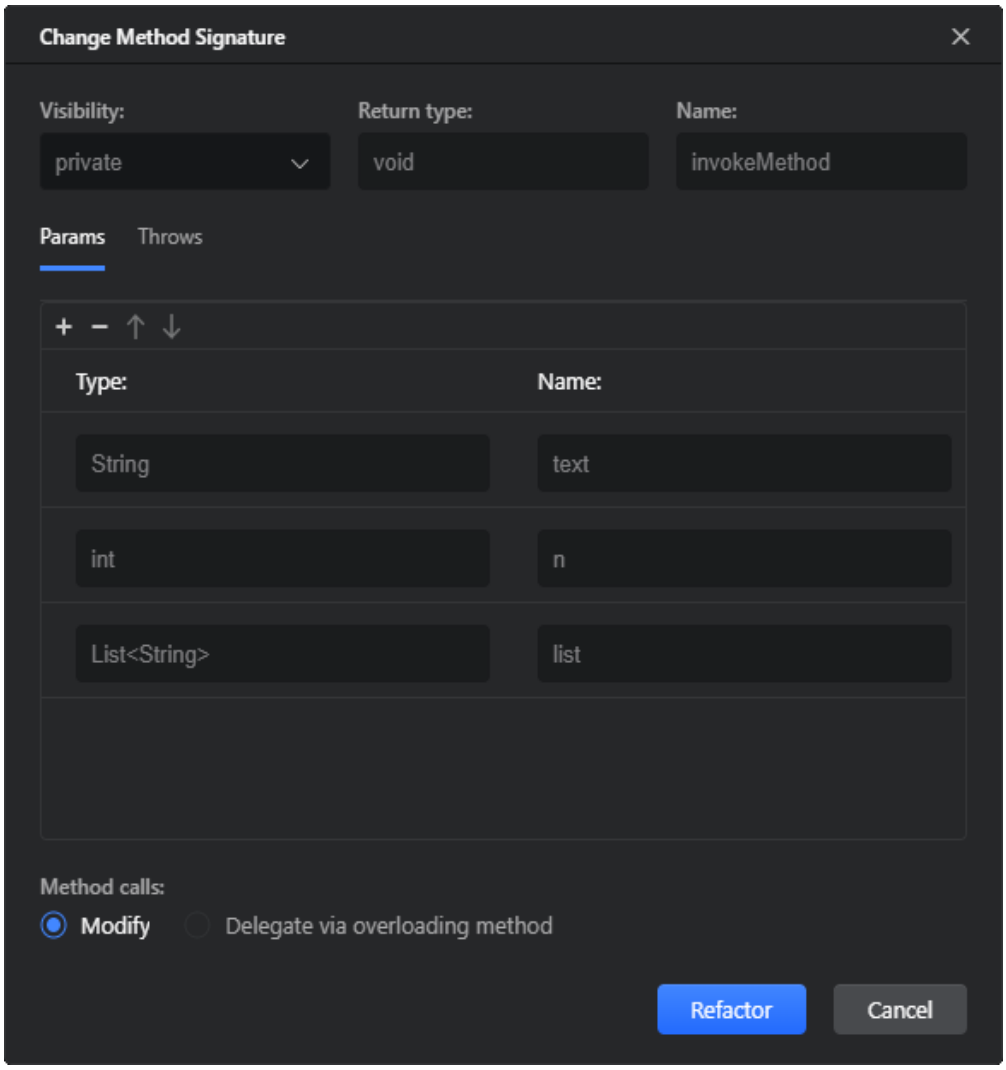
执行重构

步骤1 在代码编辑器中，将光标放置在要更改其签名的方法的声明上。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Change Method Signature**或按“Ctrl+F6”。

步骤3 在打开的**Change Method Signature**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**Params**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。目前CodeArts IDE不支持为参数提供默认值。如果添加参数，则需要手动更新方法调用。
- 在**Throws**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**Method calls**区域中，选择是修改现有方法调用还是保持原样，还是通过新创建的重载方法进行委托。



步骤4 单击**Refactor** 以应用重构。

----结束

示例

作为示例，让我们对**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

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

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

重构后

```
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);
    }
  }
}
```

5.5.12 更改类签名

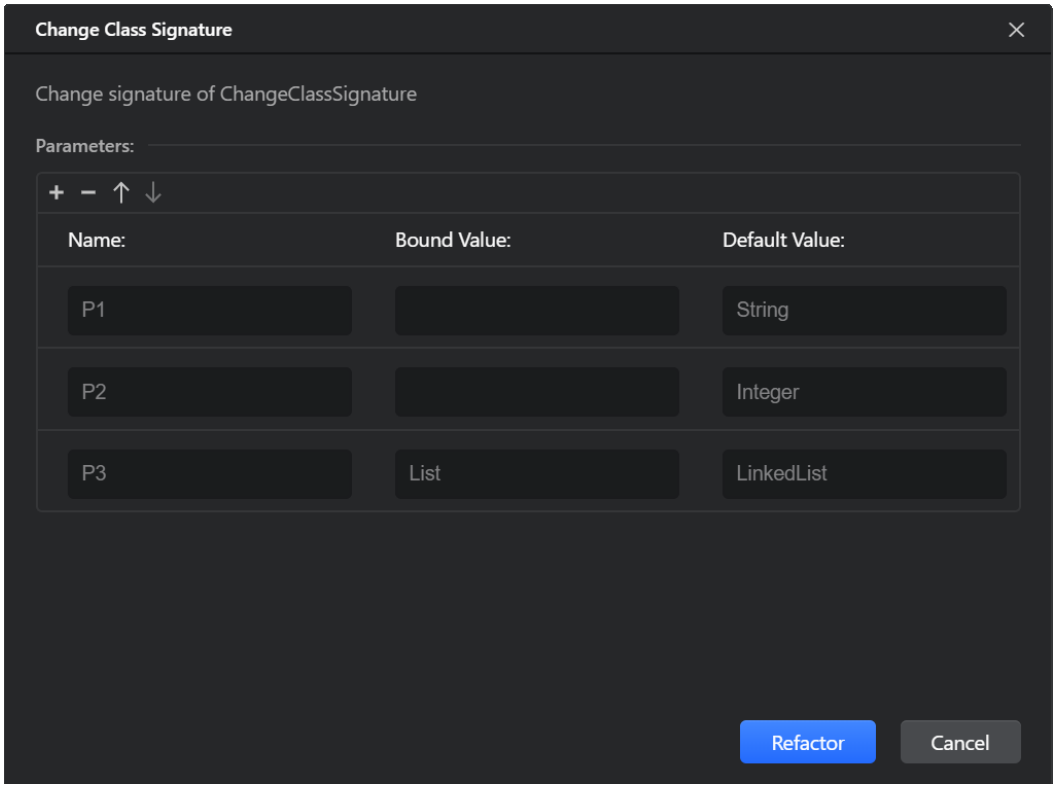
此重构允许您将类转换为泛型并操作其类型参数。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Change Class Signature**或按“Ctrl+F6”。

步骤3 在打开的 **Change Class Signature**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**Bound Value**字段中，您可以选择提供限制传递给类型参数的值的有界值。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们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

```
class ChangeClassSignature {

   public class MyOtherClass {
      ChangeClassSignature myClass;

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

重构后

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

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

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

5.5.13 将匿名类转换为内部类

此重构允许您将匿名类转换为重命名的内部类。

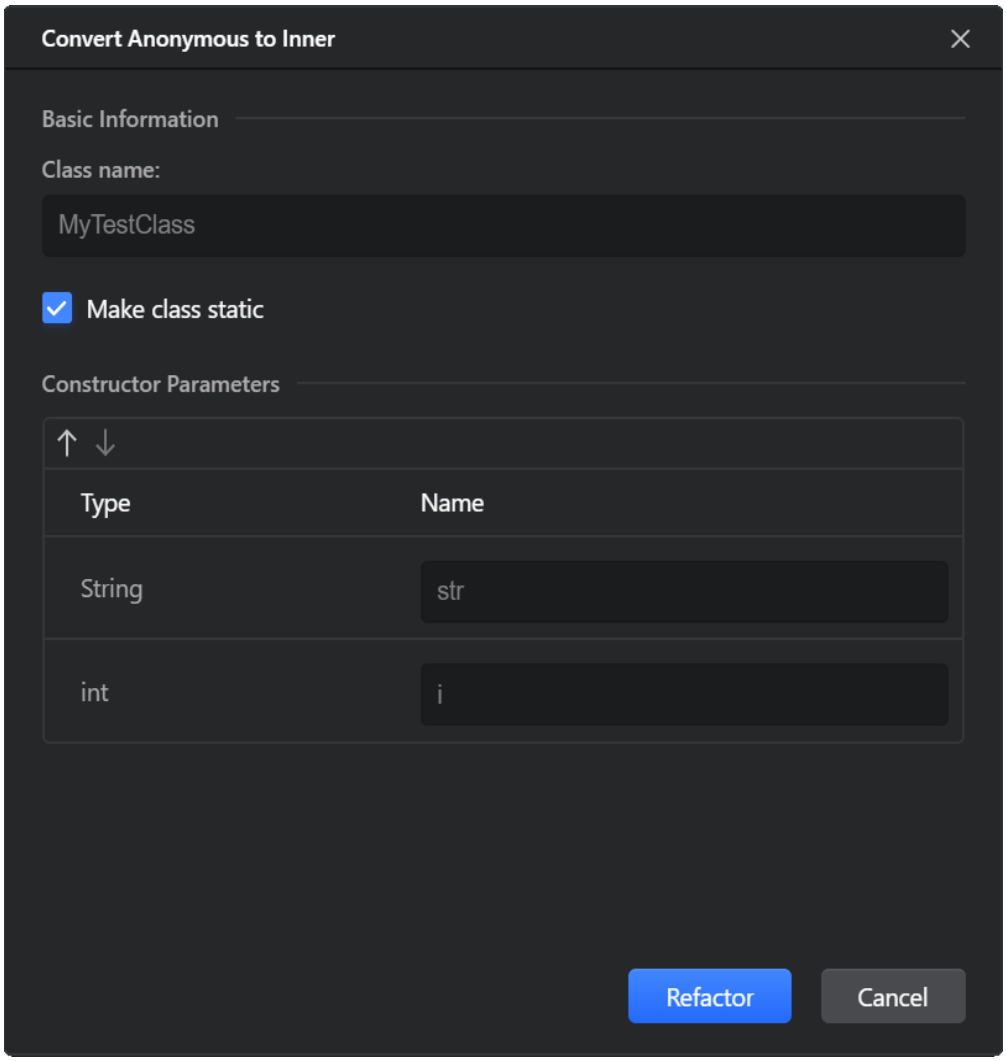
执行重构

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Convert Anonymous To Inner**。

步骤3 在打开的**Convert Anonymous To Inner**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**Constructor Parameters**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。



步骤4 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
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;
      }
    };
  }
}
```

重构后

```
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;
    }
  }
}
```

5.5.14 尽可能使用 Interface

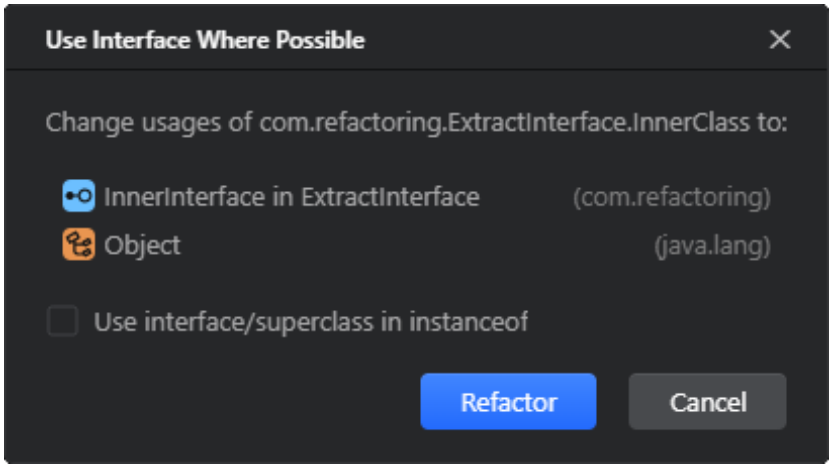
此重构允许您将从基类/接口派生的指定方法的执行委托给实现同一接口的父类或内部类的实例。

执行重构

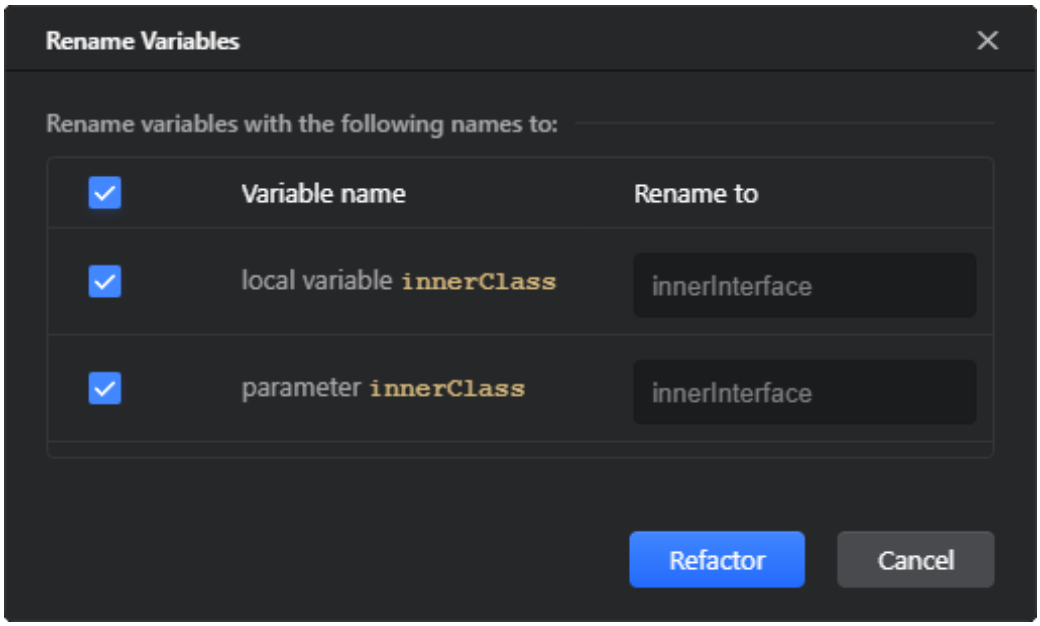
步骤1 在代码编辑器中，将光标放在应通过父类/接口委托其方法的类的声明上。

步骤2 在主菜单或编辑器上下文菜单中，选择**Refactor>Use Interface Where Possible**。

步骤3 在打开的**Use Interface Where Possible**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**instanceof**语句中**Use interface/superclass in instanceof**复选框。



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**Rename Variables**对话框中提供替代名称。



步骤4 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
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

```
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();
    }
}
```

5.5.15 类型迁移

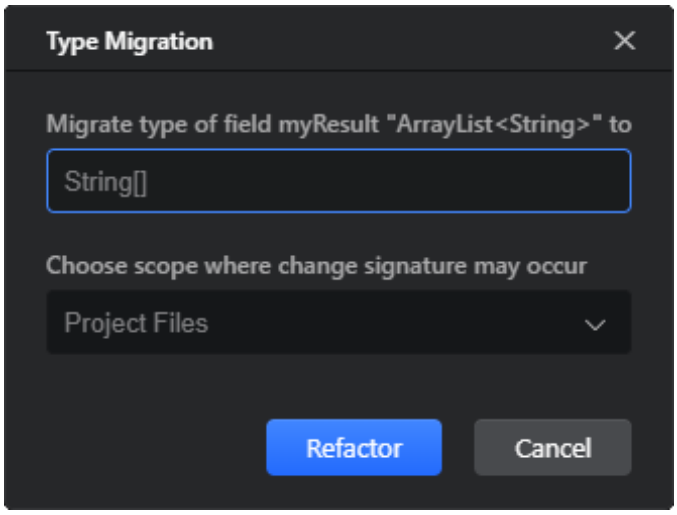
此重构允许您更改类成员、局部变量、参数、方法返回值等类型。您还可以在数组、集合之间转换变量或方法返回值的类型。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Type Migration**。

步骤3 在打开的**Type Migration**对话框中，提供要迁移到的类型。在**Choose scope**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）。



步骤4 单击**Refactor**以应用重构。

----结束

示例

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

重构前

```
class TypeMigration {

   private ArrayList<String> myResult;

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

重构后

```
class TypeMigration {

   private String[] myResult;

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

5.5.16 包装返回值

此重构允许您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。

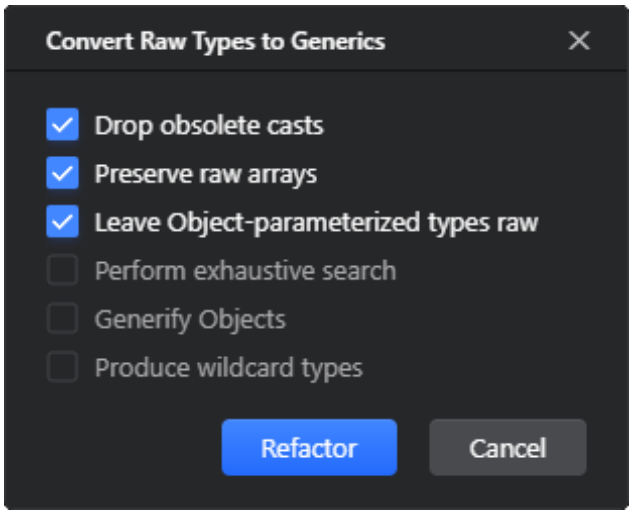
执行重构

步骤1 选择要应用重构的实体（资源管理器中的文件或文件夹、代码编辑器中的类声明或代码片段等）。

步骤2 在主菜单或上下文菜单中，选择 **Refactor>Convert Raw Types to Generics**。

步骤3 在打开的**Convert Raw Types to Generics**对话框中，提供重构选项。

- **Drop obsolete casts**：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。
- **Preserve raw arrays**：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。
- **Leave Object-parameterized types raw**： 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。
- **Perform exhaustive search**： 如果选中，则在所有节点上执行搜索。
- **Generify Objects**： 如果选中，**java.lang.Object**对象将转换为它们实际使用的类型。
- **Produce wildcard types**：如果选择此选项，则尽可能生成通配符类型（即**List<? extends String>**等表达式）。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们生成**List**和**LinkedList**类型。

重构前

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

重构后

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

5.5.17 转换为实例方法

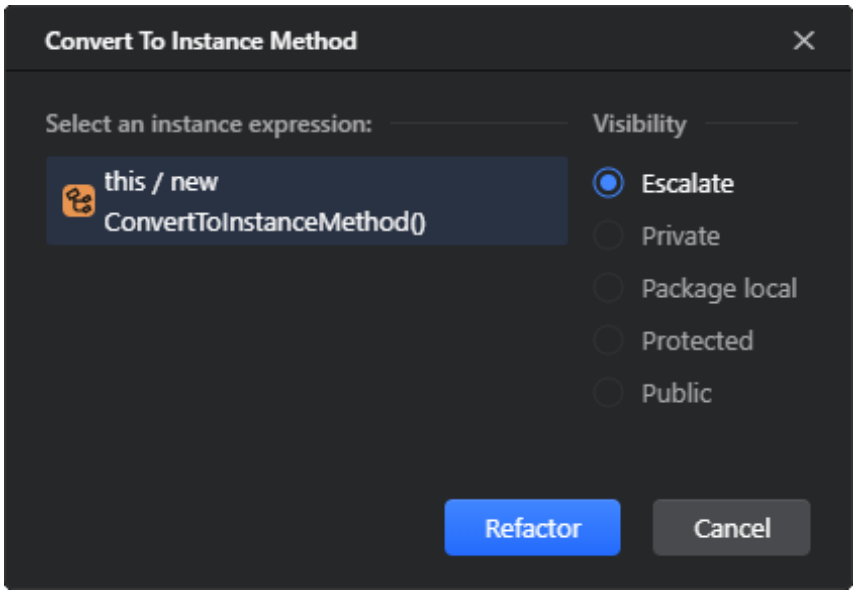
此重构允许您将类的静态方法转换为类实例的非静态方法。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Convert to Instance Method**。

步骤3 在打开的**Convert to Instance Method**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。



步骤4 单击 **Refactor**以应用重构。

----结束

示例

例如，让我们将静态方法**sayHello**转换为实例方法。

重构前

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

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

重构后

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

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

5.5.18 删除中间人

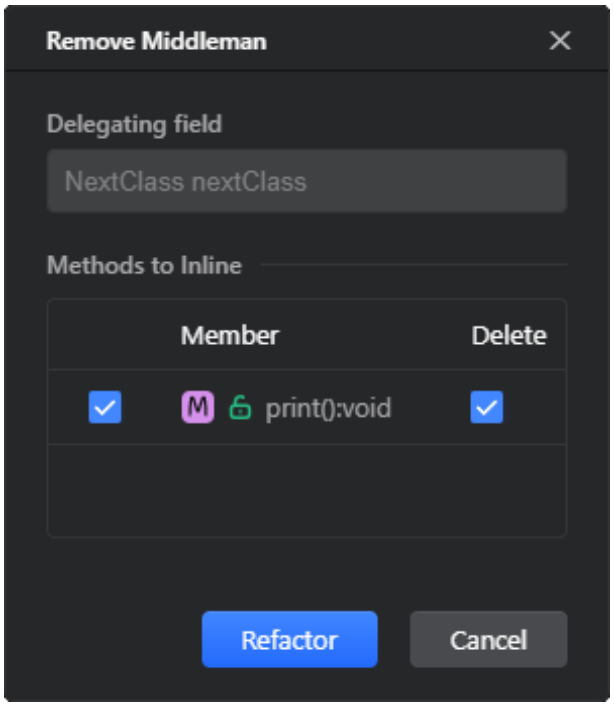
通过此重构，您可以将对类中的委托方法的调用替换为直接对委托字段的等效调用。您还可以删除委托方法，这些方法在重构后将不再使用。

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 在主菜单或编辑器上下文菜单中，选择**Refactor>Remove Middleman**。

步骤3 在打开的**Remove Middleman**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。



步骤4 单击**Refactor** 以应用重构。

----结束

示例

例如，让我们从类**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

```
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

```
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!");
        }
    }
}
```

5.5.19 安全删除

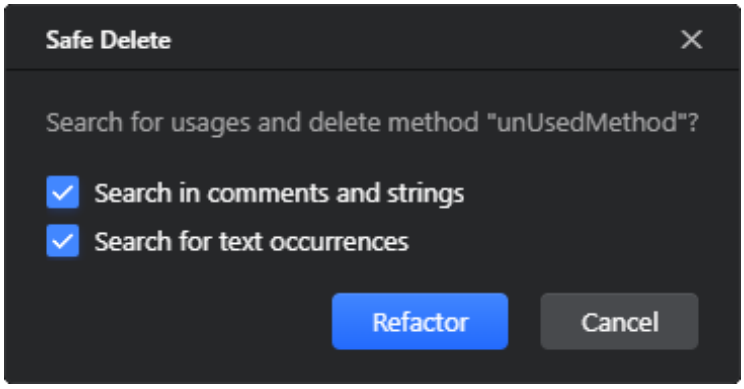
此重构允许您安全地删除文件和代码符号。CodeArts IDE将验证受影响实体的所有用法，并允许您相应地调整代码。

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 在主菜单或上下文菜单中，选择**Refactor>Safe Delete**。

步骤3 在打开的**Safe Delete**对话框中，选择CodeArts IDE是否要搜索代码中被选定符号的引用。



步骤4 单击**Refactor**以应用重构。

----结束

示例

例如，让我们在整个方法调用层次结构中删除未使用的参数i。

重构前

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

重构后

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

5.6 导航代码

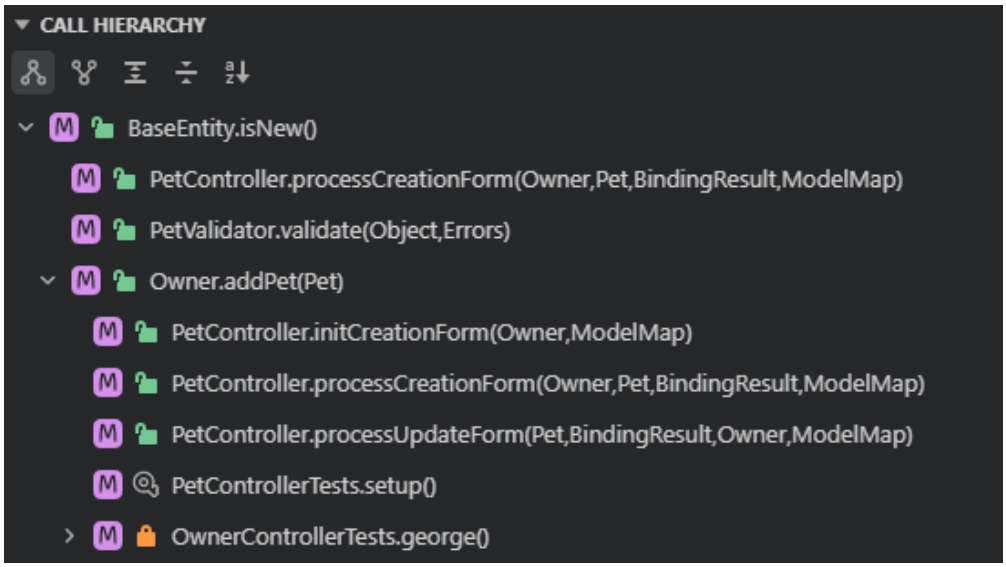
5.6.1 简介

CodeArts IDE提供了许多用于导航Java代码库的功能。除了代码导航中描述的**代码导航**技术外，您还可以使用几种特定于Java的技术。

5.6.2 Call Hierarchy

调用**Call Hierarchy**视图显示了从某个方法到其他方法的所有调用，并允许您深入到调用者和被调用者。要打开调用**Call Hierarchy**视图，在右侧的活动栏中，选择**Java**视图并展开**Call Hierarchy**节点。

- 右键单击一个方法，选择**Show Call Hierarchy**，或按“Ctrl+Alt+H”（IDEA键盘映射）。使用**Caller Methods Hierarchy**（ ）和**Callee Methods Hierarchy**（ ）工具栏按钮在调用者和被调用者列表之间切换。

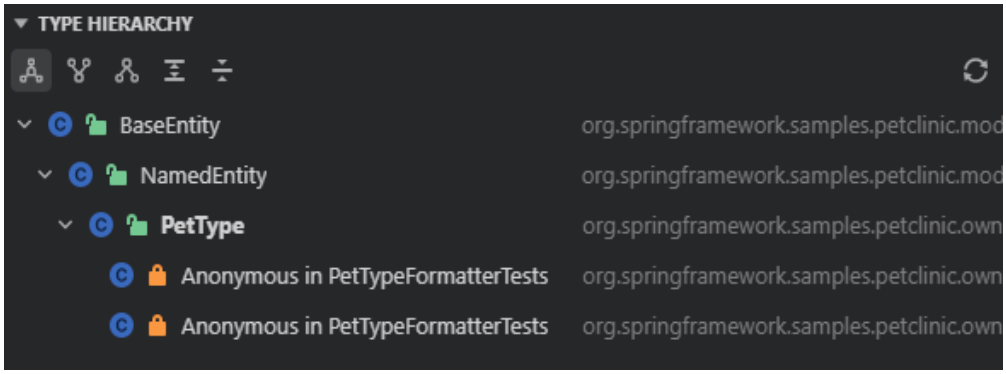


在**Call Hierarchy**视图中，您可以右键单击一个方法，从上下文菜单中选择**Base on This Type**重建层次结构，以基于所选方法重新构建层次结构。

5.6.3 Type Hierarchy

Type Hierarchy视图显示了继承关系，允许您查看所选类的父类和子类。要打开**Type Hierarchy**视图，在右侧的活动栏中，选择**Java**视图并展开**Type Hierarchy**节点。

- 右键单击一个类型，选择显示**Show Type Hierarchy**。



使用**Type Hierarchy**视图工具栏按钮在不同的查看模式之间切换。

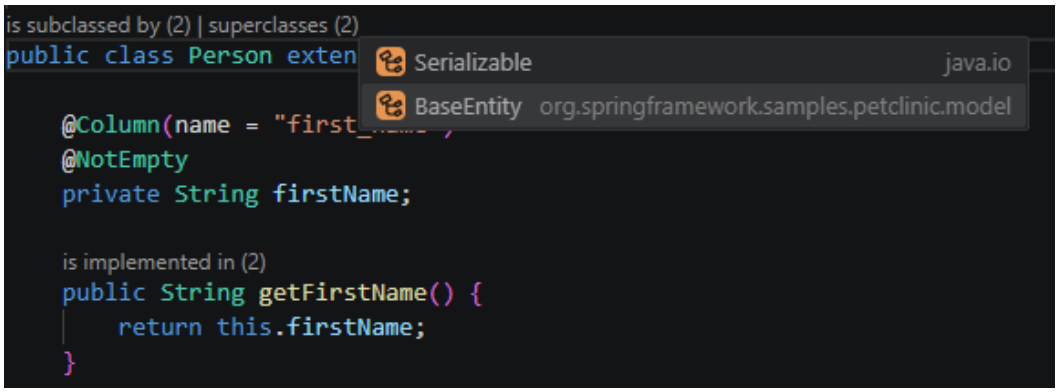
- **Class Hierarchy**：类层次结构，查看父类和子类。
- **Supertypes Hierarchy**：父类层次结构，查看父类。
- **Subtypes Hierarchy**：子类层次结构，查看子类。

在**Type Hierarchy**视图中，您可以右键单击一个类，并从上下文菜单中选择**Base on This Type**重建层次结构，以基于所选类重新构建层次结构。

5.6.4 CodeLens

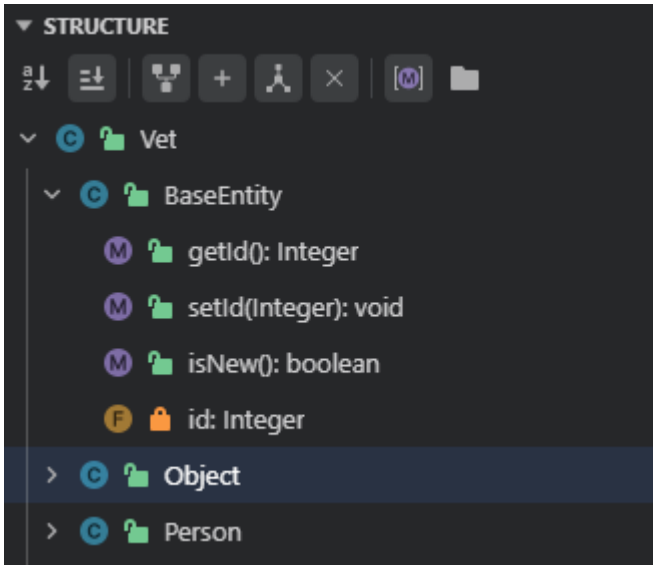
Java引用CodeLens显示当前类的超类/子类的内联计数，以及当前方法的重写。

单击CodeLens，在弹出的窗口中选择要导航到的实体。



5.6.5 Structure

Structure视图显示当前活动的**Java**文件的符号树，并提供了几种排序、分组和过滤功能。要打开**Structure**视图，在右侧活动栏中选择**Java views**并展开**Structure**节点，或按“Ctrl+Shift+F10”。

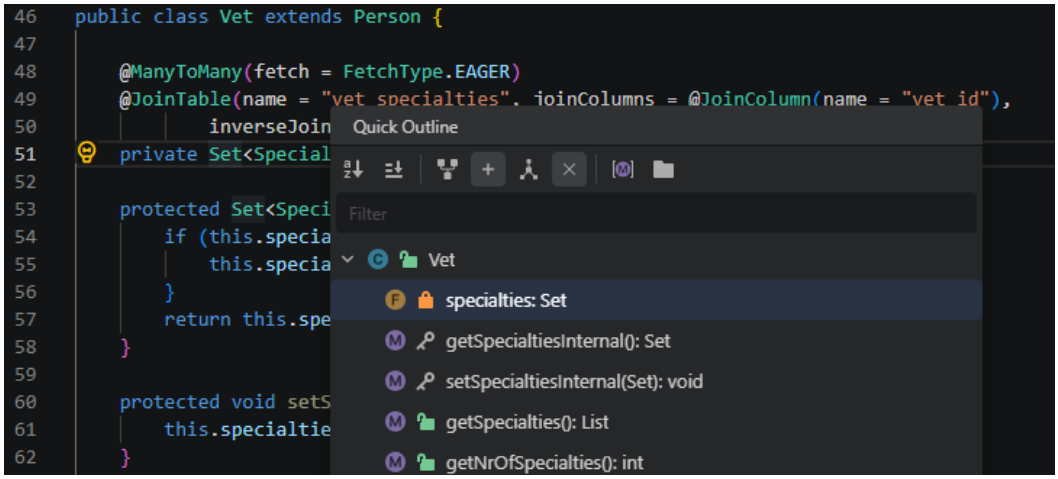


要快速导航到一个符号，在**Structure**视图列表中单击相应的项。使用**Structure**视图工具栏按钮对显示的符号进行排序、过滤和分组。

- : 单击按字母顺序对列表进行排序。
- : 单击按项的可见性修饰符对列表进行排序。
- : 单击显示继承的成员。
- : 单击显示字段。
- : 单击显示匿名类。
- : 单击显示非公共类成员。
- : 单击按它们所定义的类型对方法进行分组。
- : 单击按它们访问的属性对getter和setter方法进行分组。

说明

这个功能也可以通过Peek视图来实现，它会在当前编辑器中显示，这样您就不需要切换上下文。要在Peek视图中查看文件结构，请在主菜单中**Navigate**>**Quick Outline**，或按“**Ctrl+F10**”/“**Ctrl+F12**”。



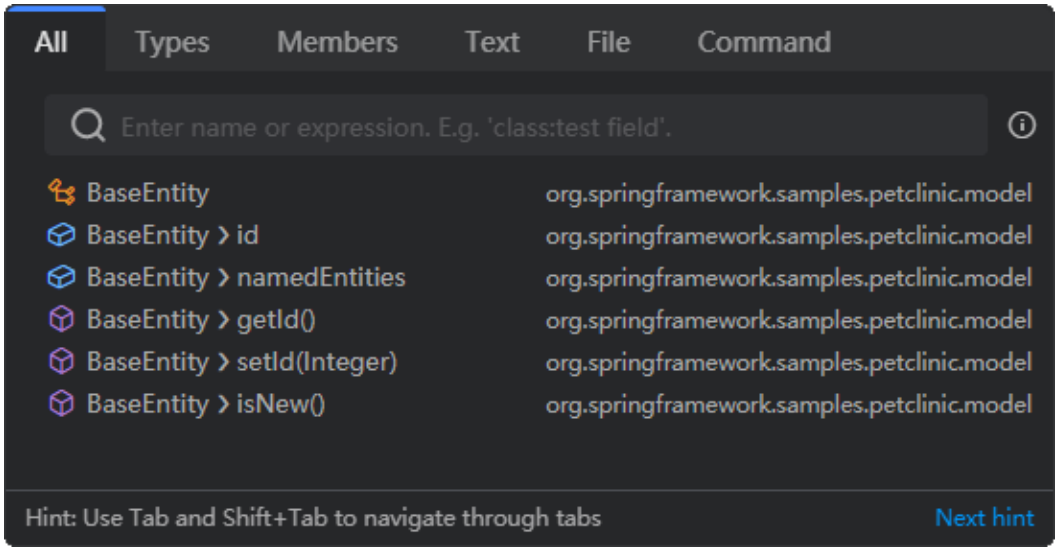
5.7 通过代码搜索

5.7.1 简介

CodeArts IDE捆绑了SmartAssist扩展，为Java项目提供高度准确的编码辅助功能。通过其智能搜索功能，您可以立即搜索和导航到任何项目位置，以及找到并执行任何CodeArts IDE命令。

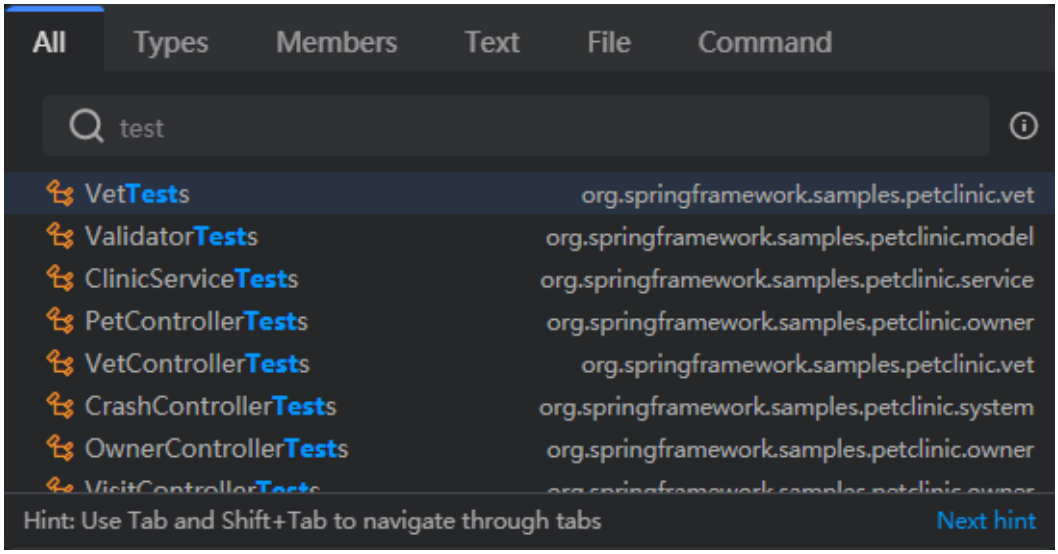
5.7.2 基本用法

步骤1 按下“Ctrl+Shift+A”/“Shift Shift”启动**SmartSearch**。

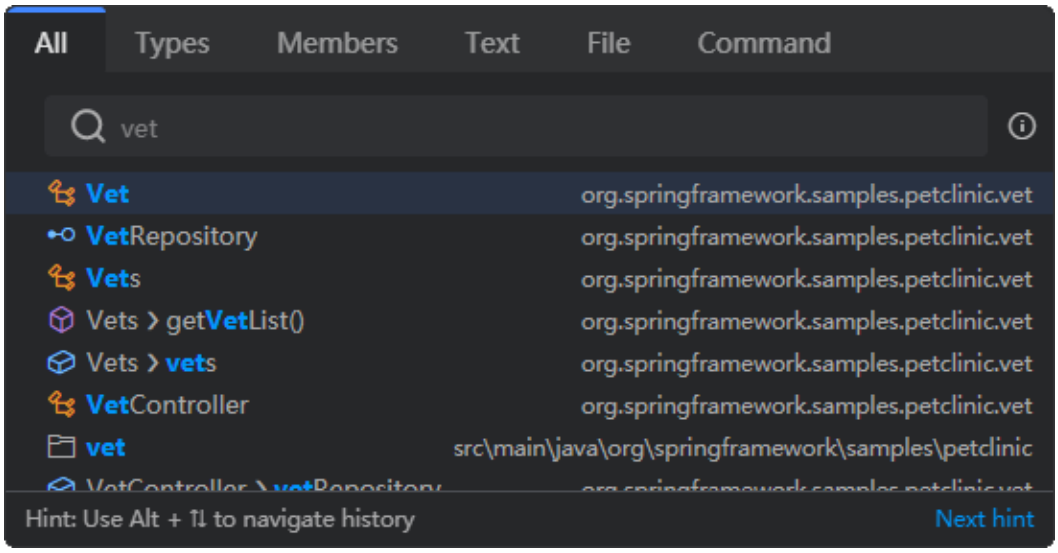


如果您当前在代码编辑器中打开了一个Java文件，**SmartSearch**窗口将自动显示其大纲，让您在代码条目（例如类成员）之间导航。

步骤2 输入搜索请求。为了缩小搜索范围，可以在**SmartSearch**窗口中切换选项卡，或使用**搜索查询语法**。



步骤3 在**SmartSearch**窗口中，当前打开文件的结果显示在**local**区域；项目中其他位置的结果显示在**general**区域。



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。或者双击所需的条目。要关闭**SmartSearch**窗口，按“Escape”键。

----结束

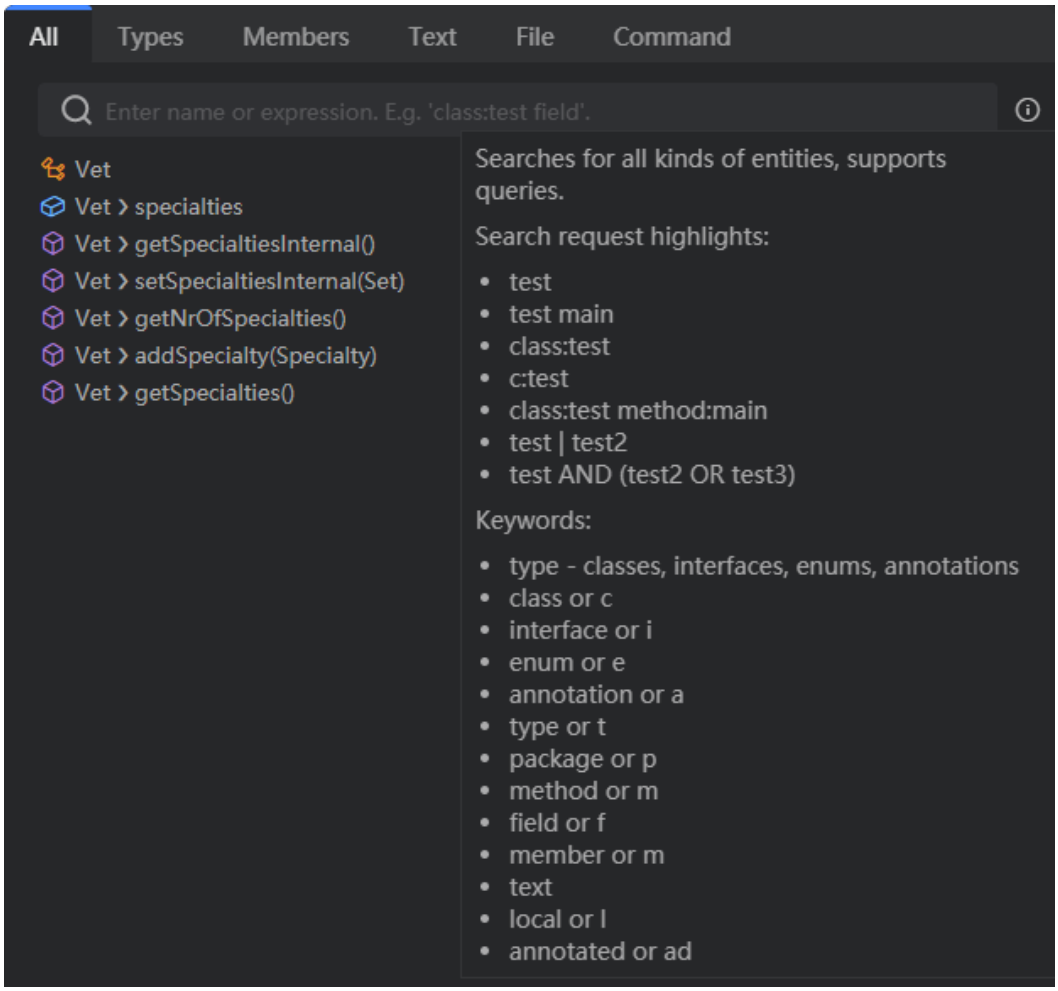
5.7.3 搜索查询语法

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。

数据源名称	数据源简码	描述
local	l	当前文件实体

数据源名称	数据源简码	描述
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。请注意，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SmartSearch查询语法，请单击SmartSearch窗口右上角的按钮。



搜索运算符

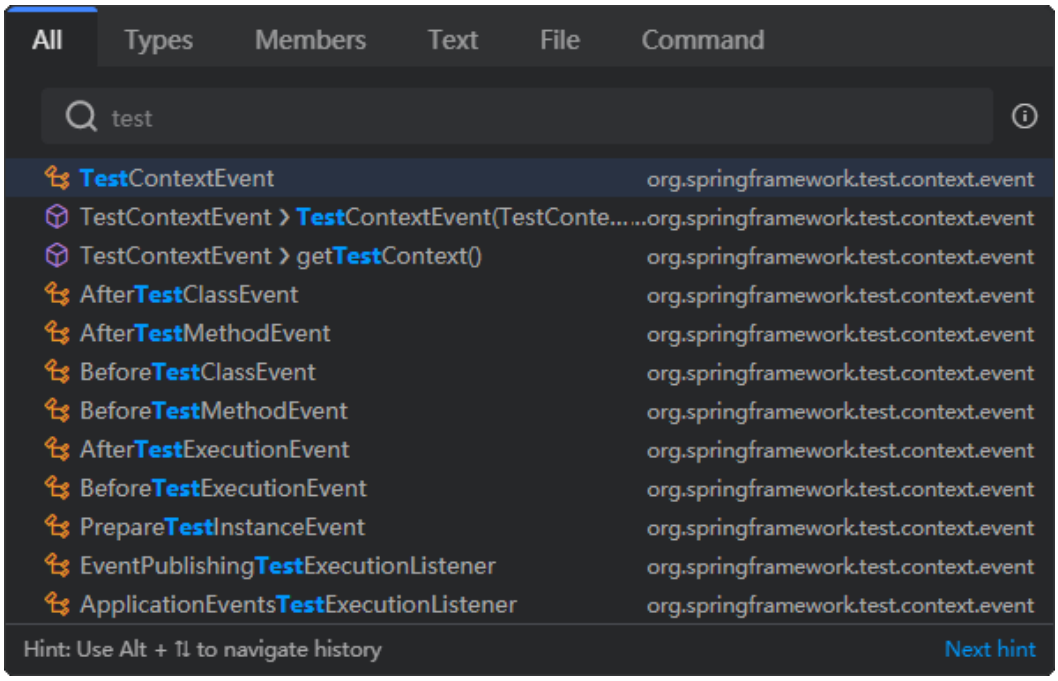
您可以通过使用**AND**和**OR**运算符，或它们的组合，来构建复杂的搜索查询，例如**class:foo AND(method:bar OR method:baz)**。

运算符	语法	描述
AND	AND，&，&&，(space character)	**SmartSearch**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，|，||	**SmartSearch**将返回与任何提供的查询匹配的所有条目。

5.7.4 示例

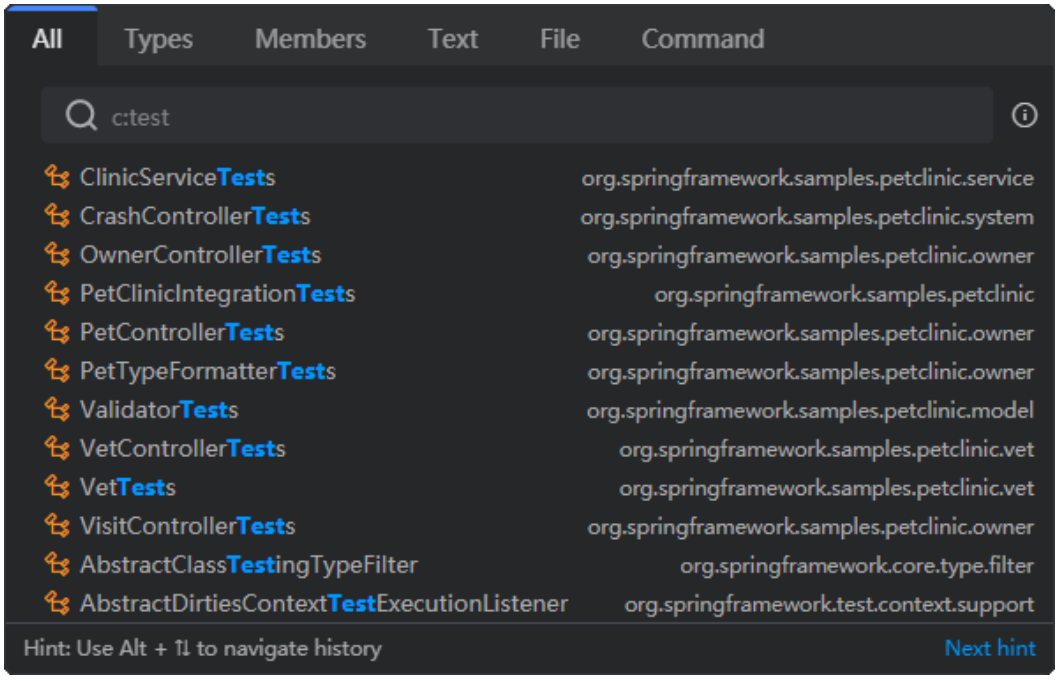
5.7.4.1 定位任意实体

一个搜索查询**test**匹配所有名称中带有**test**一词的实体。



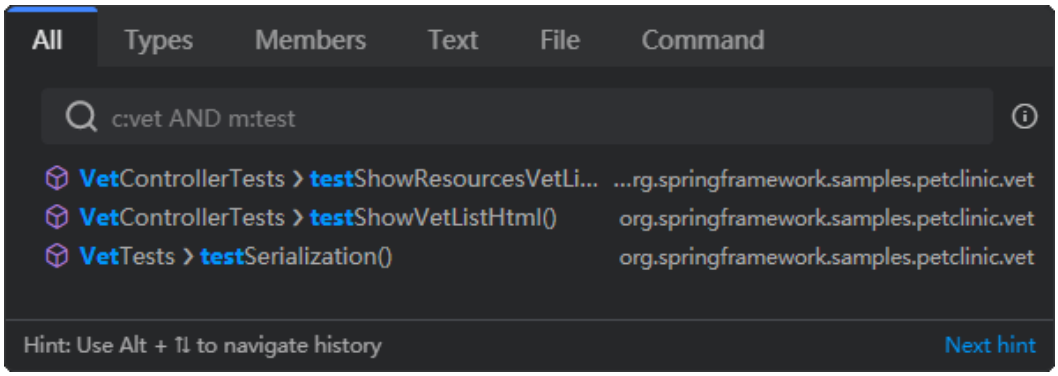
5.7.4.2 定位类

一个搜索查询**class:test**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**。

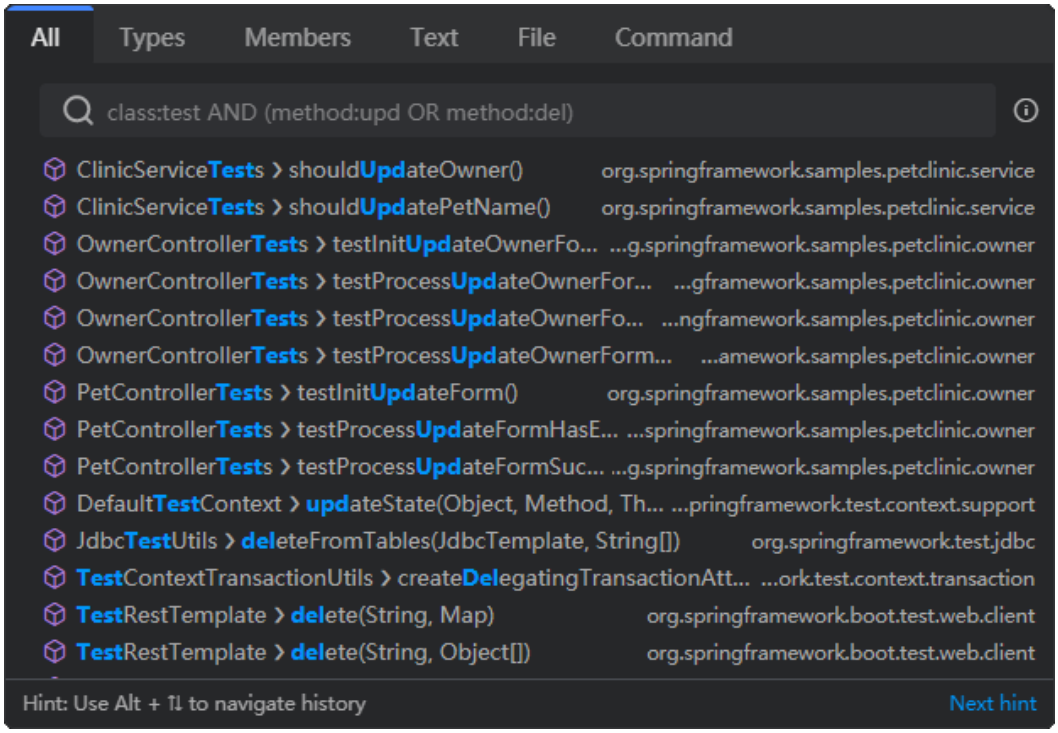


5.7.4.3 定位类中的方法

一个搜索查询**class:vet AND method:test**匹配所有名称中带有**test**的方法，并且属于名称中带有**vet**的类。



一个搜索查询**class:test AND (method:upd OR method:del)**匹配所有名称中带有**upd**或**del**的方法，并且属于名称中带有**test**的类。



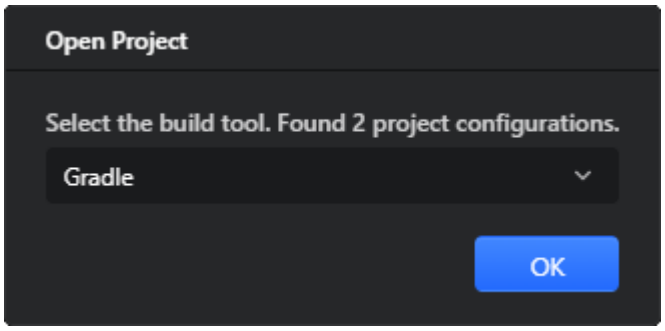
5.8 构建工具

5.8.1 简介

CodeArts IDE提供了Java构建工具的内置集成。开箱即用，支持以下构建工具：

- Gradle
- Maven.

当您首次打开包含Java源代码文件的文件夹时，CodeArts IDE会自动检测项目文件夹中的**pom.xml**或**build.gradle**文件，并加载相应的项目（Maven或Gradle）。如果同时存在这两个文件，CodeArts IDE会提示您选择构建系统。

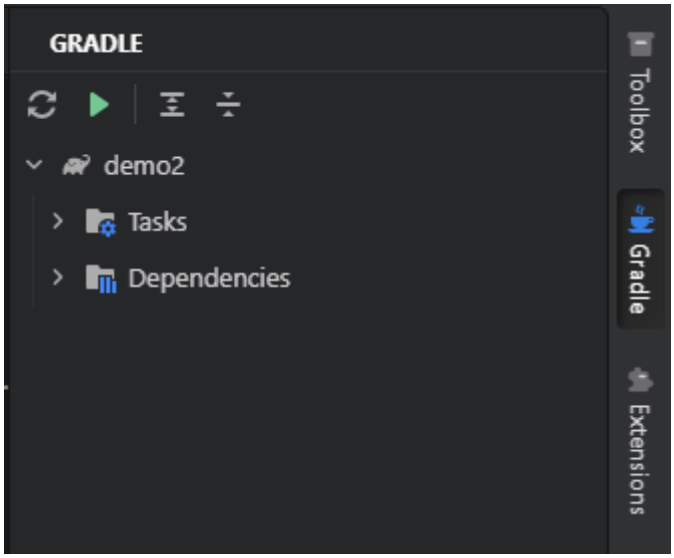


选择了构建工具后，CodeArts IDE会自动加载项目。

5.8.2 Gradle

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。

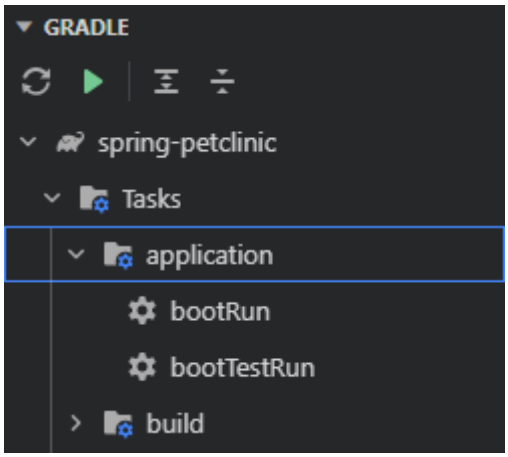
要打开**Gradle**视图，请在右侧的活动栏中选择**Gradle**视图。



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**重新加载项目**按钮（ ）以使CodeArts IDE应用您的更改。

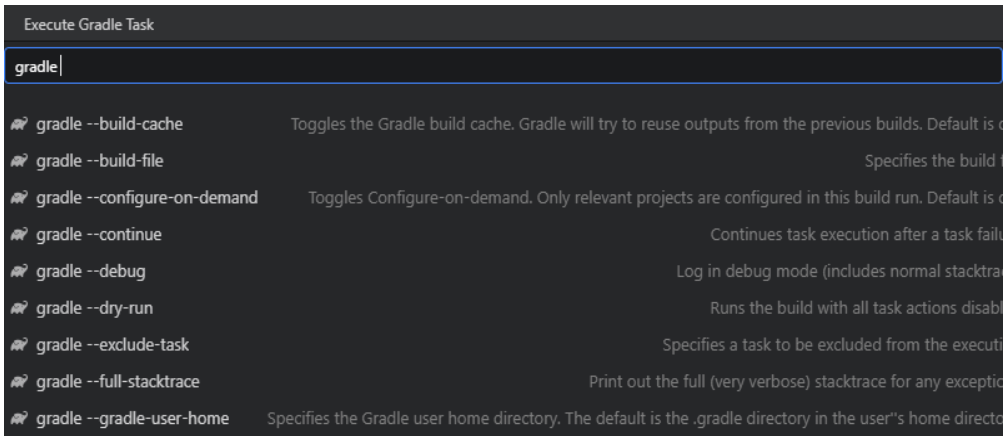
使用 Gradle 任务进行工作

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

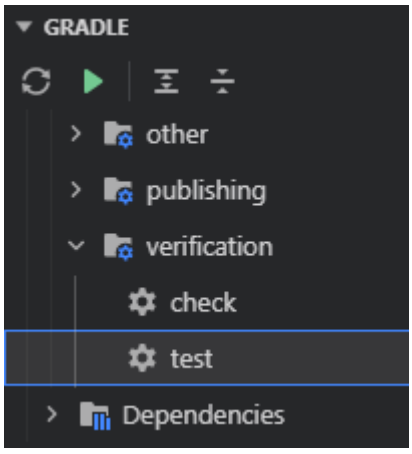


要运行任务，请执行以下任一操作：

- 双击任务列表中的任务。
- 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**弹出窗口中选择所需的任务。



以同样的方式，您可以运行在**build.gradle**的**test**任务中定义的测试。在这种情况下，CodeArts IDE将使用Gradle测试运行器。



说明

您还可以通过**专用的Gradle启动配置**来运行Gradle任务。

有关在CodeArts IDE中测试应用程序的更多信息，请参阅**测试**。

配置自定义 Gradle 集成

步骤1 通过以下任一方式打开“Java智能助手设置”：

- 依次单击左下角“管理->Java智能助手设置”。
- 在CodeArts IDE状态栏中单击**Java智能助手**。
- 在命令面板中运行**SmartAssist: Java智能助手设置**命令（命令面板通过“Ctrl+Shift+P”/“Ctrl+Ctrl”打开）。

步骤2 切换到**构建工具**页面，在**Gradle**部分定义配置选项。

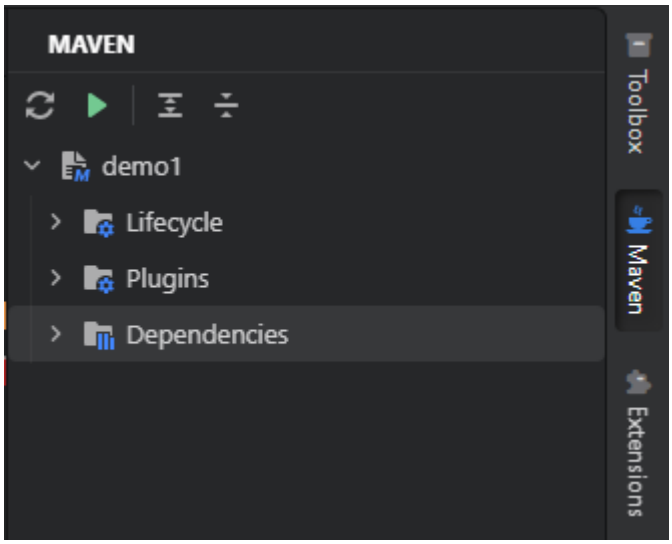
- **Gradle用户目录**：在此字段中，指定Gradle用户主目录的路径（默认为“$USER_HOME/.gradle”），该目录用于存储全局配置属性和初始化脚本、缓存和日志文件。默认值是基于**“GRADLE_USER_HOME”**环境变量的值提供的。要修改它，您可以设置环境变量或单击，并手动选择所需的Gradle用户主目录。
- **Gradle SDK**：从这个列表中选择一个与Gradle一起使用的JDK：捆绑的JDK、项目级别的JDK或从系统变量（如“JAVA_HOME”）解析的JDK。

----结束

5.8.3 Maven

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提供了集成的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。

要打开**Maven**视图，请在右侧的活动栏中选择**Maven**视图。

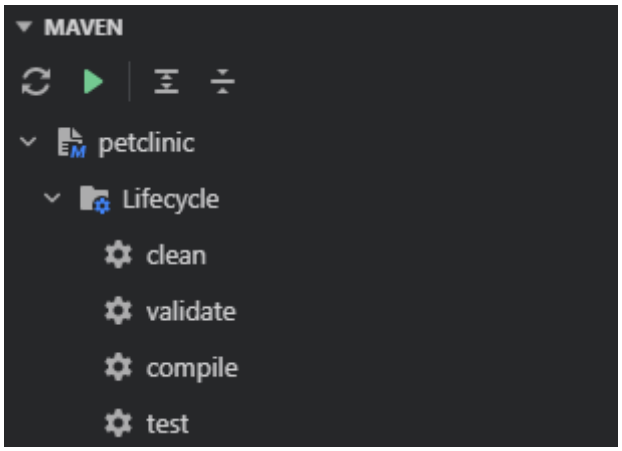


约束与限制

在修改**pom.xml**后，例如添加新的依赖项，单击**Maven**视图工具栏上的**重新加载项目按钮**（）以使CodeArts IDE应用您的更改。

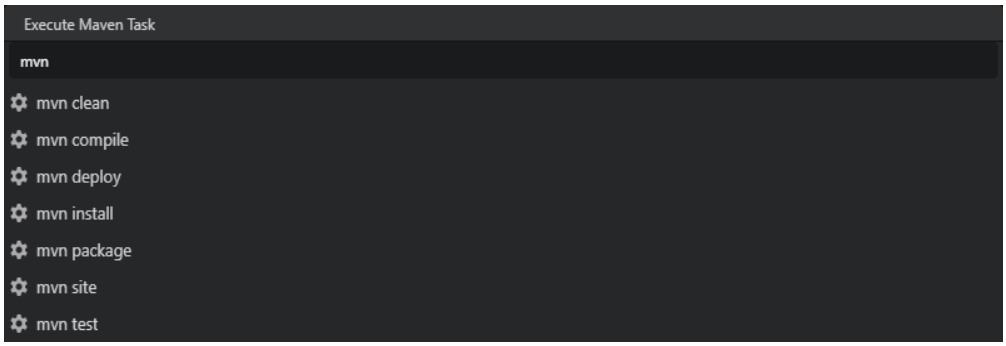
使用 Maven 任务进行工作

使用Maven任务在CodeArts IDE中打开一个Maven项目后，您可以在**Maven**视图中找到Maven任务列表。



要运行任务，请执行以下任一操作：

- 双击任务列表中的任务。
- 在**Maven**视图工具栏上，单击**Execute Maven Task**按钮（ ）然后在打开的**Execute Maven Task**弹出窗口中选择所需的任务。



说明

您还可以通过**专用的Maven启动配置**来运行Maven任务。

配置自定义 Maven 集成

步骤1 通过以下任一方式打开“Java智能助手设置”：

- 依次单击左下角“管理->Java智能助手设置”。
- 在CodeArts IDE状态栏中单击**Java智能助手**。
- 在命令面板中运行**SmartAssist: Java智能助手设置**命令（命令面板通过“Ctrl+Shift+P”/“Ctrl+Ctrl”打开）。

步骤2 切换到**构建工具**页面，在**Maven**部分定义配置选项。

- **Maven路径**：在此字段中选择捆绑的Maven版本（Maven 3），或单击选择您自己的**Maven**安装路径。
- **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。

- **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。
- **Maven SDK**：从此列表中选择要与Maven一起使用的JDK：捆绑的JDK、项目级别的JDK或从系统变量（如**JAVA_HOME**）解析的JDK。
- **离线工作**：如果选择此项，Maven将在离线模式下工作。它不连接到远程仓库，只使用本地可用的资源。此选项对应于**--offline**命令行选项。
- **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。
- **使用插件注册表**：如果选择此项，可以引用Maven的插件注册表。此选项对应于**--no-plugin-registry**命令行选项。
- **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。

----结束

5.9 调试

5.9.1 通用调试步骤

CodeArts IDE内置调试器有助于加快编辑、编译、运行和调试循环的速度。

1. 在代码中设置断点以定义程序应停止的位置。
2. 在调试模式下运行程序。
3. 当程序挂起时，在**Run and Debug**视图中检查其输出。
4. 找到错误，更正错误，然后重新运行程序。在Java上下文中，您可以使用热代码替换功能在不停止调试会话的情况下动态修改和重新加载类。

5.9.2 断点

5.9.2.1 简介

断点定义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排水沟、使用排水沟的上下文菜单或在**Run and Debug**视图的**Breakpoints**中进行切换。

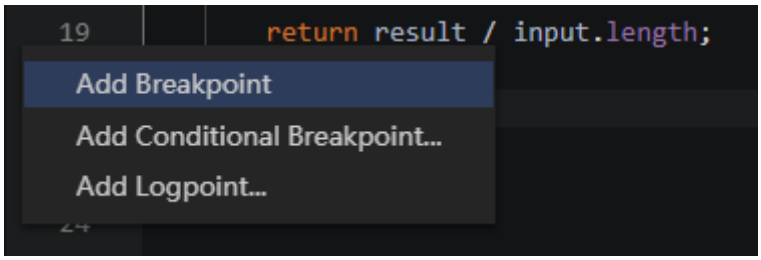
5.9.2.2 设置断点

5.9.2.2.1 行断点

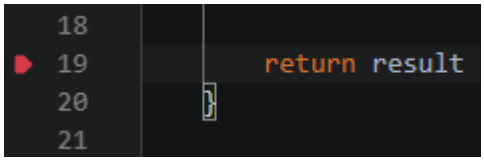
行断点是常规断点，可在设置的行上停止程序的执行。

执行以下任一操作：

- 单击编辑器排水沟中所需的行。
- 右键单击编辑器排水沟中所需的行，然后从上下文菜单中选择**Add Breakpoint**。



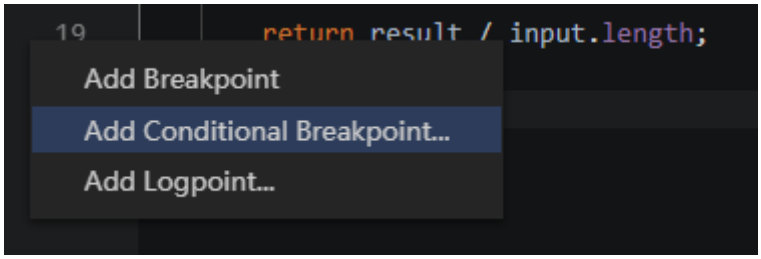
行断点由编辑器排水沟中的右箭头形图标（ ）表示：



5.9.2.2.2 条件断点

CodeArts IDE 调试器允许您根据任意表达式或命中计数设置条件断点。

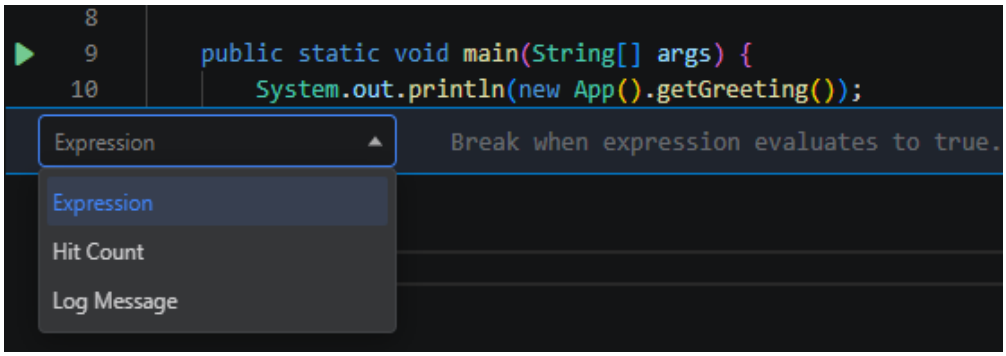
1. 在代码编辑器中，右键单击所需的行，然后从上下文菜单中选择**Add Conditional Breakpoint...** 。



或者，在主菜单中，选择**Debug**>**New Breakpoint**>**Condintional Breakpoint**。

2. 在打开的查看编辑器中，从列表中选择条件类型。

- **Expression**：表达式，只要表达式的计算结果为true ，就会命中断点。
- **Hit Count**：命中次数，断点需要命中定义的次数才能停止程序执行。



3. 输入条件并按**Enter**。

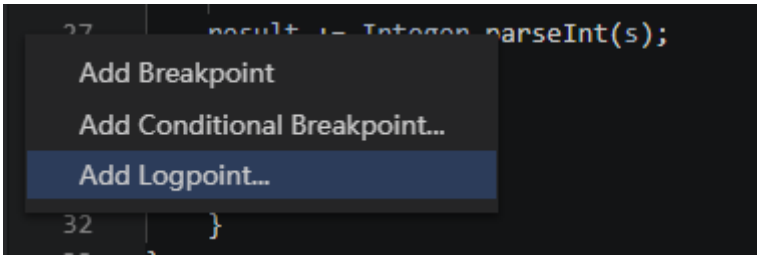
说明

您也可以向常规行断点添加条件或命中计数。右键单击编辑器装订线中的断点，然后从上下文菜单中选择所需的操作。

5.9.2.2.3 日志点

日志点是一个断点，在命中时不会停止程序执行，而是将消息记录到控制台。

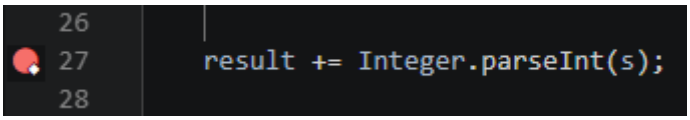
1. 在代码编辑器中，右键单击所需的行，然后从上下文菜单中选择**Add Logpoint...**。



   或者，在主菜单中，选择**Debug>New Breakpoint>Logpoint**。

2. 在打开的窥视编辑器中，键入命中日志点时应记录的消息。日志消息可以是纯文本，也可以包括要在大括号（**{}**）中计算的表达式。

   日志点由编辑器排水沟中的菱形图标（）表示：



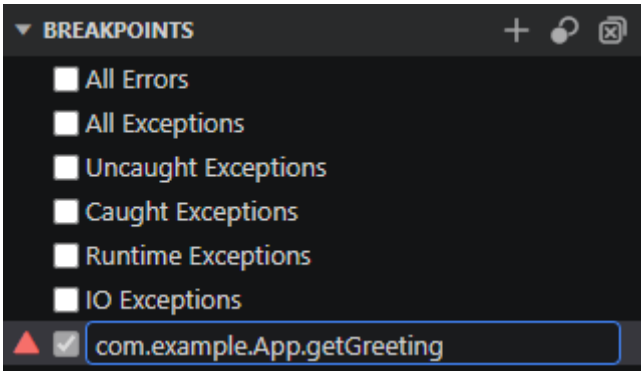
约束与限制

就像常规断点一样，日志点可以启用或禁用，也可以由条件或命中计数控制。如果设置了条件或命中计数，则仅在条件为真或达到命中计数时记录消息。

5.9.2.2.4 函数断点

除了直接在源代码中放置断点外，您还可以通过指定函数/方法名称来创建断点。程序执行在进入指定的函数时停止。

1. 单击右侧活动栏中的**Run**按钮（）或按“Ctrl+Shift+D” / “Shift+Alt+F9” / “Alt+5” / “Ctrl+Shift+F8”打开**Run and Debug**视图。
2. 在**BREAKPOINTS**部分中，单击**Add Function Breakpoint**工具栏按钮（），或者在主菜单中选择 **Debug**>**New Breakpoint**>**Function Breakpoint**.
3. 输入所需函数的完全限定名称，然后按**“Enter”**键。

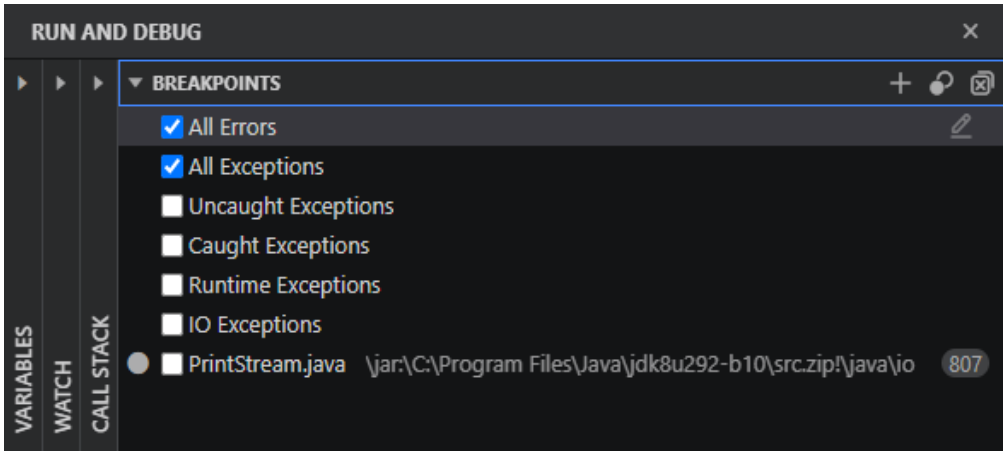


函数断点在**Run and Debug**视图的**BREAKPOINTS**部分中使用三角形图标（）表示。

5.9.2.2.5 异常断点

CodeArts IDE调试器支持异常断点，每当抛出可抛出或其子类时，断点就会挂起程序。异常断点是全局应用的，不需要特定的源代码引用。

1. 单击右侧活动栏中的**Run**按钮（）或按“Ctrl+Shift+D”/“Shift+Alt+F9”/“Alt+5”/“Ctrl+Shift+F8”打开**Run and Debug**视图。
2. 展开**BREAKPOINTS**部分，然后选中要设置的异常断点旁边的复选框。



CodeArts IDE提供了几种类型的异常断点。这些定义了在抛出时暂停程序执行的特定异常类。

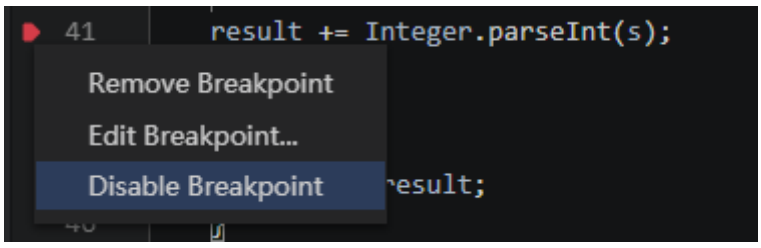
- **All Errors**： Error及其子类。
- **All Exceptions**： Exception及其子类。
- **Uncaught Exceptions**： 未捕获的Exception及其子类。
- **Caught Exceptions**： 捕获的Exception及其子类。
- **Runtime Exceptions**： RuntimeException及其子类。
- **IO Exceptions**： IOException及其子类。

5.9.2.3 启用和禁用断点

您可以同时启用或禁用单个断点，或禁用所有断点。

要禁用单个断点，请执行以下任一操作：

- 在编辑器排水沟中，右键单击断点，然后从上下文菜单中选择**Disable Breakpoint**。



- 在**Run and Debug**视图的**BREAKPOINTS**部分中，清除要禁用的断点旁边的复选框。

  要同时禁用所有断点，请执行以下操作：

  在**Run and Debug**视图的**BREAKPOINTS**部分中，单击工具栏按钮。

5.9.2.4 删除断点

您可以一次删除单个断点，也可以删除所有断点。

要删除单个断点，请单击编辑器排水沟中的断点。要同时删除所有断点，请在**Run and Debug**视图的**BREAKPOINTS**部分中，单击工具栏中的**Remove All Breakpoints**按钮（ ）。

5.9.3 在调试模式下运行程序

CodeArts IDE允许您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

如果您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。

在具有**main()**方法的类的代码编辑器中，单击**main()**方法上方的**Run**或**Debug CodeLens**项。运行应用**程序启动配置**将自动创建并运行。

```
public class MyClass {

    Run | Debug
    public static void main(String[] args) {
        System.out.println("args");
    }
}
```

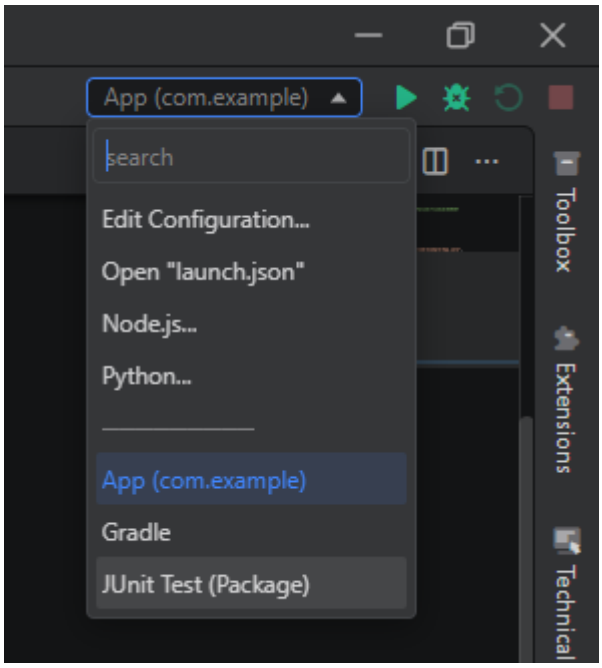
说明

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

从启动配置启动调试会话

启动配置允许您配置和保存各种调试方案的调试设置详细信息。有关使用启动配置的详细信息，请参见**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，然后按“**F5**”或者“Shift+F9”。



步骤2 执行以下任一操作：

- 在主菜单中选择**Debug**>**Start Debugging**，或按“F5 ”/ “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**Start Debugging**按钮（ ）。

----结束

调试会话启动后，将立即显示**Debug Console**面板并显示调试输出，状态栏将更改颜色（默认颜色主题为橙色）。**debug status**显示在显示活动启动配置的状态栏中。单击调试状态以更改活动启动配置并重新启动调试，而无需重新打开**Run and Debug**视图。

5.9.4 控制程序执行

启动调试会话后，可以使用调试工具栏操作控制程序的执行。

图标	动作	快捷键	描述
	Pause / Continue	F9	暂停/恢复调试会话。
	Step Over	F8	跳过当前代码行到下一行。 如果当前行中有方法调用，则将跳过它们的实现，以便您移动到调用者方法的下一行。
	Step Into	F7	进入方法以显示其实现。
	Step Out	Shift+F8	退出当前方法，跳转到调用者方法。
	Restart	Ctrl +Shift +F5	重新启动调试会话。
	Stop	Ctrl+F2	停止调试会话。
	Compile and Replace	--	在调试期间，您可以编辑程序的代码并动态重新加载更改。单击此按钮可重新编译受影响的类，并将正在运行的字节码替换为新的字节码。因此，您不需要重建整个程序和重新启动调试会话。 目前支持以下修改： • 更改任何方法的主体。 • 添加/删除私有方法。 • 更改私有方法的签名和非访问修饰符。 • 在任何方法中添加/删除/更改lambda。

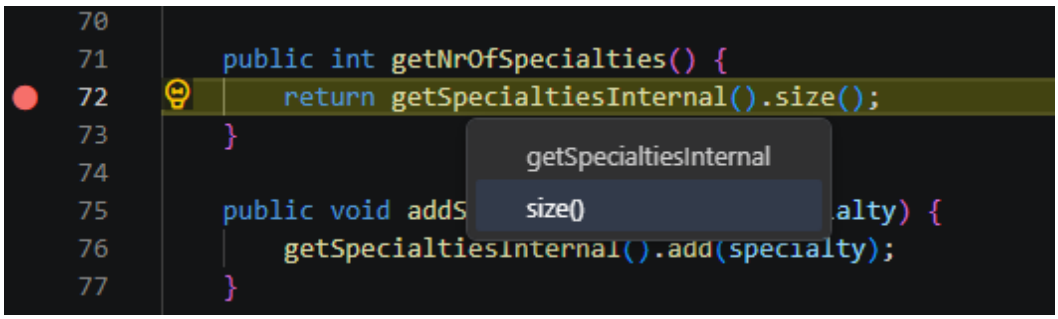
运行到光标处

当程序挂起时，您可以继续执行，直到到达光标位置。在代码编辑器中，将光标放置在所需的行，右键单击并从上下文菜单中选择**Run to Cursor**，或按“Alt+F9”。

步入目标

当一行上有多个方法调用时，**Step Into Target**功能允许您选择要逐步进入的方法调用。

在代码编辑器中，右键单击悬挂的行，然后从上下文菜单中选择**Step Into Target**。在打开的弹出菜单中，选择要进入的方法。



重新加载修改后的类

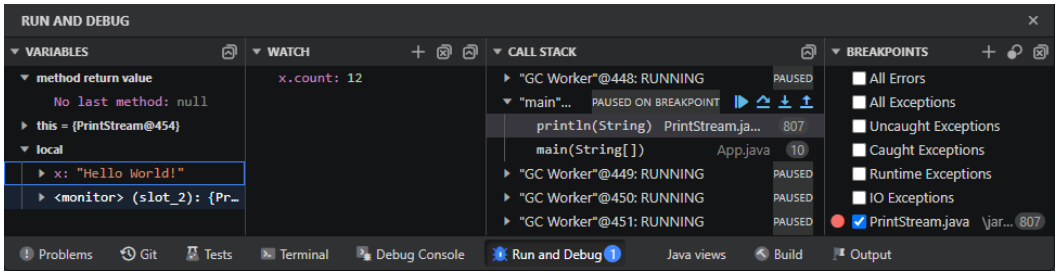
CodeArts IDE调试器提供了热代码替换功能，允许您在调试过程中编辑程序的代码，并即时重新加载您的更改。因此，您无需重新构建整个程序并重新启动调试会话。

1. 按照以**调试模式运行程序**的说明开始调试会话。
2. 当程序在断点处停止时，对代码进行必要的编辑。请注意，热代码替换支持以下修改：
   - 更改任何方法的主体。
   - 添加/删除、更改私有方法的签名和非访问修饰符。
   - 在任何方法中添加/删除/更改lambda表达式。
3. 在CodeArts IDE工具栏上，单击“**Compile and Replace**”按钮（）以重新编译受影响的类，并用新的字节码替换正在运行的字节码。

5.9.5 检查暂停的程序

5.9.5.1 简介

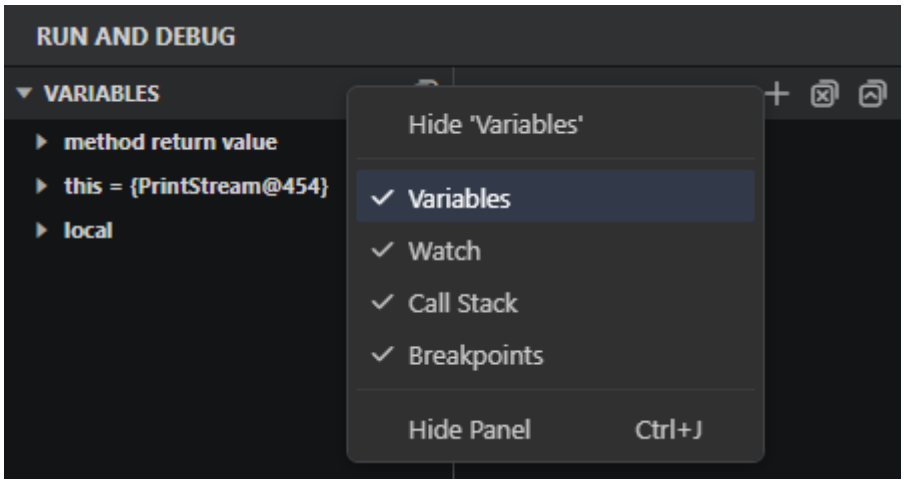
启动调试会话时，**Run and Debug**视图将打开，以显示与运行和调试相关的所有信息。手动打开 **Run and Debug**视图，单击右侧活动栏中的**Run and Debug**按钮（），或按“Ctrl+Shift+D”/“Shift+Alt+F9”/“Alt+5”/“Ctrl+Shift+F8”快捷键。



Run and Debug视图包括以下部分：

- **Variables：变量**
- **Call Stack：调用堆栈**
- **Watch：监视**
- **Breakpoints：断点**

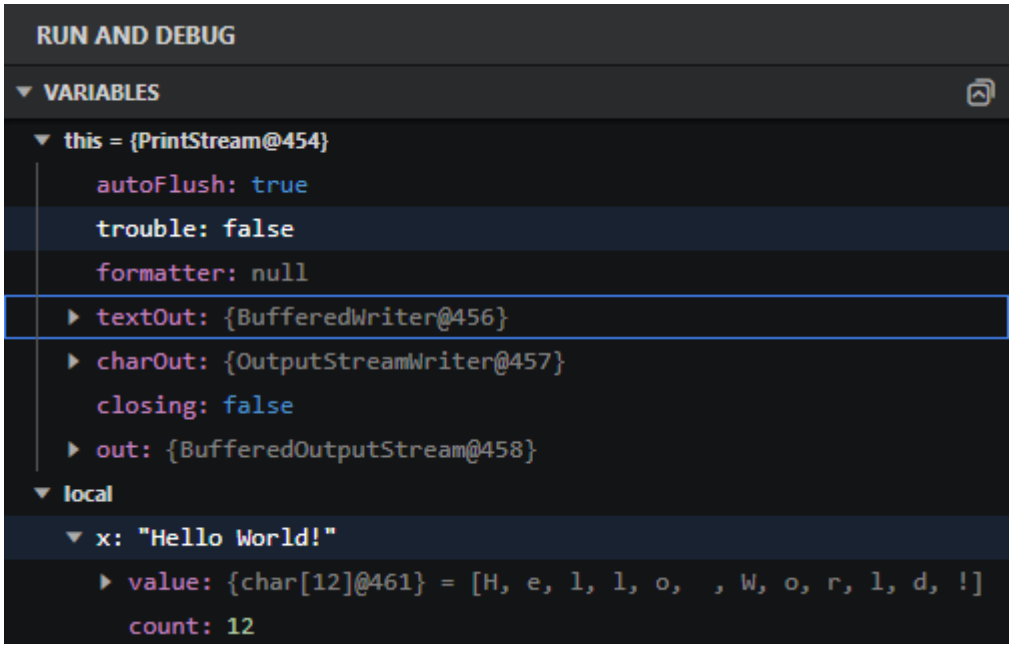
要自定义**Run and Debug**视图内容，请单击右上角的**Views and More Actions**按钮（），指向**Views**然后选中要显示的部分旁边的复选框。



5.9.5.2 检查变量

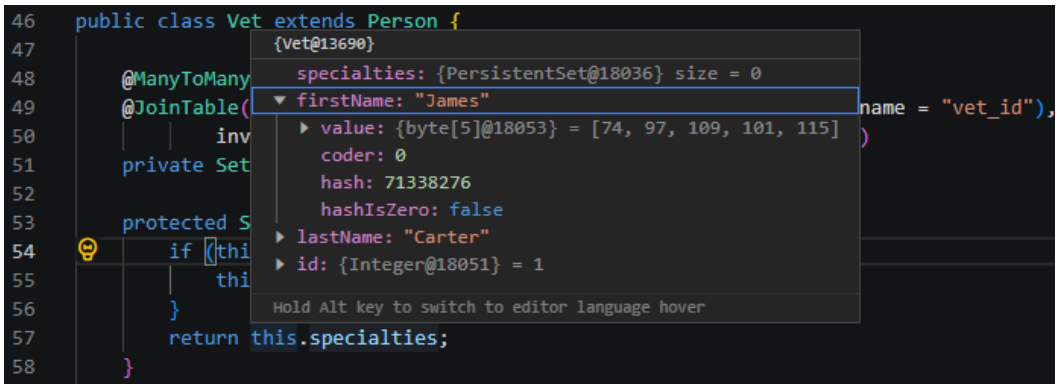
VARIABLES部分显示在当前堆栈帧（即在“调用堆栈”部分中选择的堆栈帧）中可访问的元素，并包括以下部分：

- **static**：列出静态类字段。
- **method return value**：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。这使您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。
- **local**：显示被调用方法作用域内的局部变量。



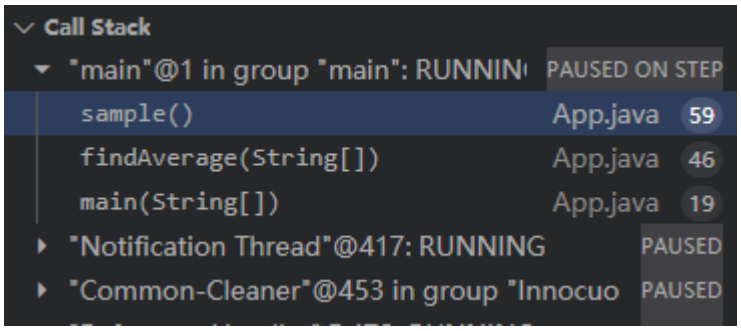
您可以使用变量上下文菜单中可用的**Set Value**操作或双击变量来修改变量的值。此外，您可以使用**Copy Value**操作复制变量的值，或使用**Copy as Expression**操作复制表达式以访问变量。

变量和表达式也可以在**Run and Debug**视图的**Watch**部分中进行计算和监视。您还可以直接在CodeArts IDE代码编辑器中计算表达式并检查值。为此，请在挂起的程序中将鼠标悬停在所需的表达式、变量或方法调用上。



5.9.5.3 检查调用堆栈

Call Stack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。

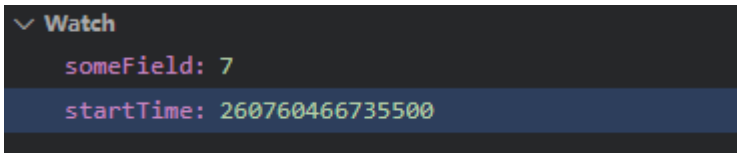


在框架内可访问的元素列在**Variables**部分中。

- 要切换到其他帧，请在**Call Stack**部分中双击其名称。
- 要在代码编辑器中快速打开方法调用，请展开**Call Stack**中的框架，然后双击所需的方法调用。

5.9.5.4 监视

Watch部分允许您跟踪程序运行时计算的变量或任意表达式。



要添加表达式，请执行以下任一操作：

- 双击**Watch**部分中的任何位置，或单击添加按钮（ ），并在显示的字段中提供所需的表达式。
- 要快速为变量添加监视，请在**Watch**部分中右键单击其名称，然后在上下文菜单中选择**Add to Watch**。

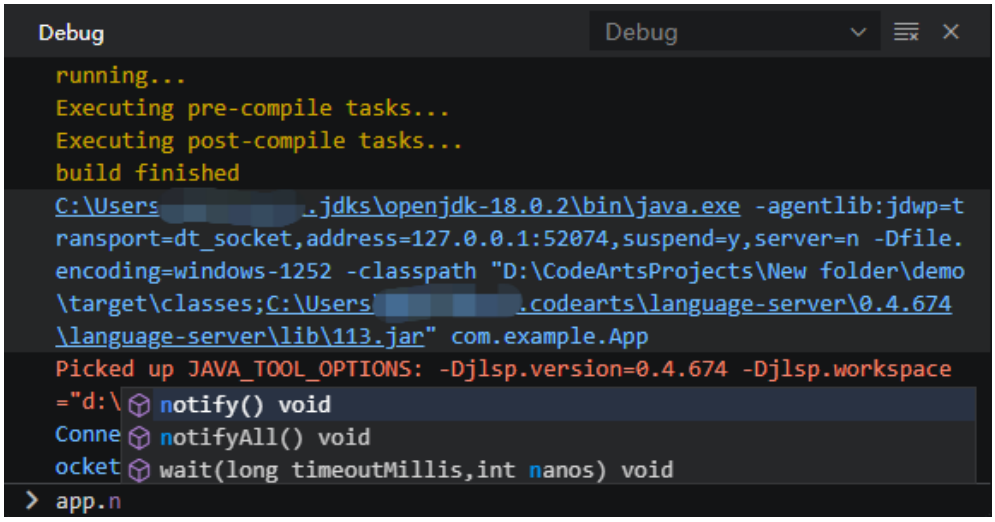
要删除表达式，请选择它，然后按**Delete**。要同时删除所有表达式，请单击**Remove all Expressions**按钮（ ）。

5.9.5.5 断点

Breakpoints部分允许您管理断点。有关详细信息，请参见**断点**。

5.9.6 调试控制台 REPL

在调试会话期间，您可以通过**Debug**控制台REPL（读取-评估-打印循环）计算表达式。**Debug**控制台输入使用活动代码编辑器的模式，这意味着它支持语法着色、缩进、代码完成、自动关闭引号和其他语言功能。



- 要打开**Debug**控制台，请单击**Debug Console**视图右上角的“切换调试控制台”按钮（ ），或按“Ctrl+Shift+Y”/“Alt+F8”。
- 要开始新行，请按“Shift+Enter”键。
- 要计算表达式，请按“Enter”键。

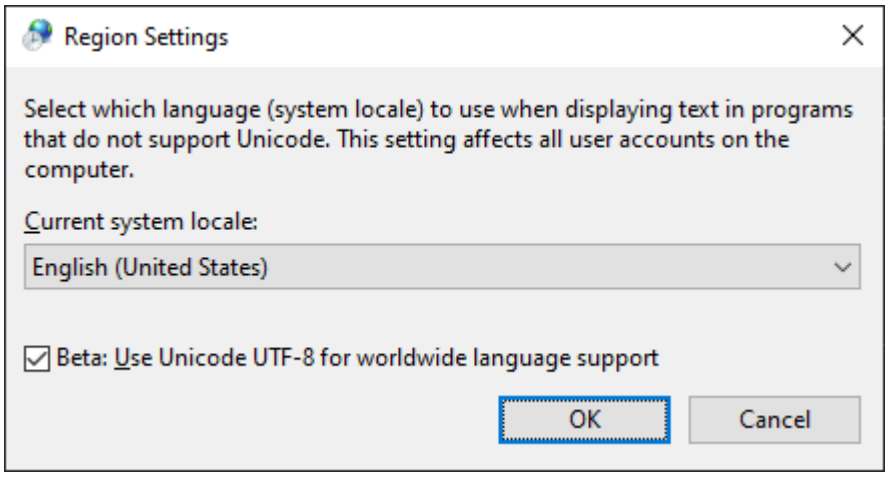
约束与限制

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：

步骤1 在 **Windows 控制面板**中，转到“**时钟和区域**”>“**区域**”。

步骤2 在“**管理**”选项卡上，单击“**更改系统区域设置**”。

步骤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**”。



步骤4 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

----结束

5.10 测试

5.10.1 将测试框架集成到项目中

CodeArts IDE提供了与**JUnit**和**TestNG**测试框架的集成，让您轻松运行和调试Java测试用例。在开始之前，请确保为项目定义了JDK，如**使用Java项目**中所述。

您可以通过在**pom.xml**（对于Maven）或**build.gradle**（对于Gradle）中声明相应的依赖来在项目中启用测试框架集成。

JUnit 3/4

对于Maven项目，请在**pom.xml**中添加以下配置。

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>YOUR_JUNIT_VERSION</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:YOUR_JUNIT_VERSION'
}
test {
    useJUnitPlatform()
}
```

JUnit 5

对于Maven项目，请在**pom.xml**中添加以下配置。

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
```

```
    <artifactId>junit-jupiter</artifactId>
    <version>RELEASE</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。

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

对于Gradle项目，请将以下行添加到**build.gradle**中：

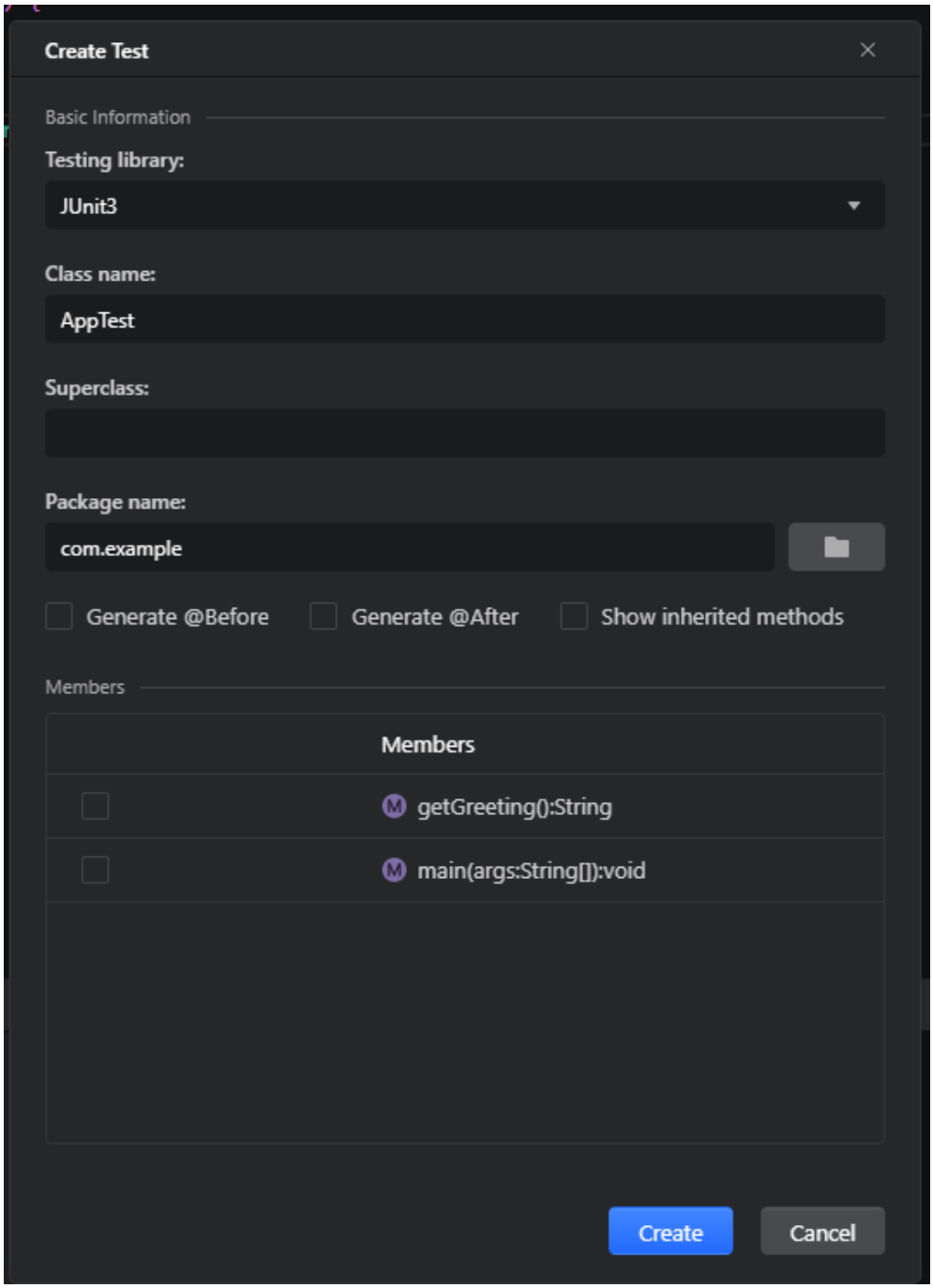
```
dependencies {
    testImplementation 'org.testng:testng:7.7.0'
}
```

5.10.2 Create tests 创建测试

CodeArts IDE提供了专门的**源操作**，帮助您搭建测试用例。

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，然后从上下文菜单中选择**Source Action**>**Test**。或者，在“命令选项板”（“Ctrl+Shift+P”/“Ctrl Ctrl”）中搜索并运行**Source Action**命令。

步骤2 在打开的**Create Test**对话框中，提供要创建的测试类的详细信息。



- 选择要使用的测试框架。
- 提供测试类的名称，或保留默认值。
- 对于JUnit3，在**Superclass**字段中提供**junit.framework.TestCase**。对于其他框架，请将该字段留空。
- 选择测试类存储在其中的包。
- 如有必要，请选择**Generate @Before/Generate @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。
- 为了能够查看并选择从超类中继承的方法，请选中**Show inherited methods**复选框。

- 在**Members**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**Create**。

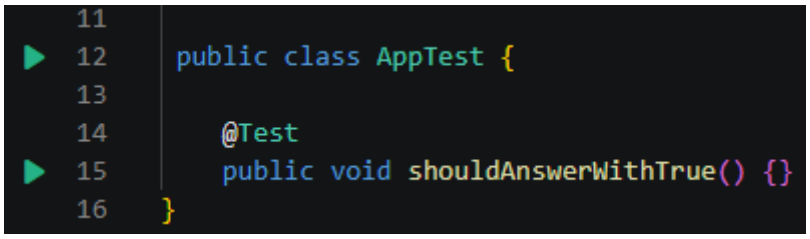
----结束

5.10.3 运行测试

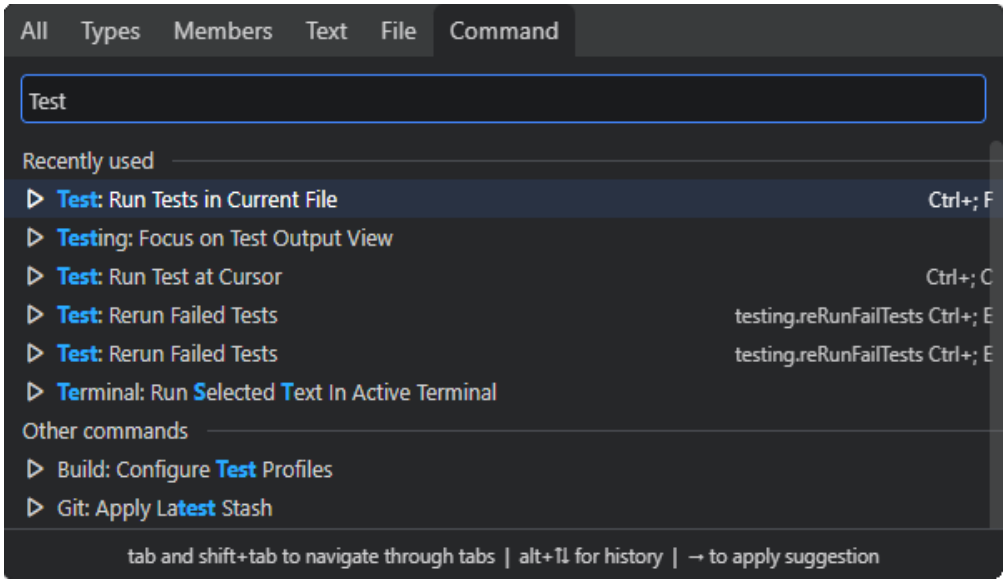
5.10.3.1 简介

CodeArts IDE提供了多个选项来运行和调试测试：

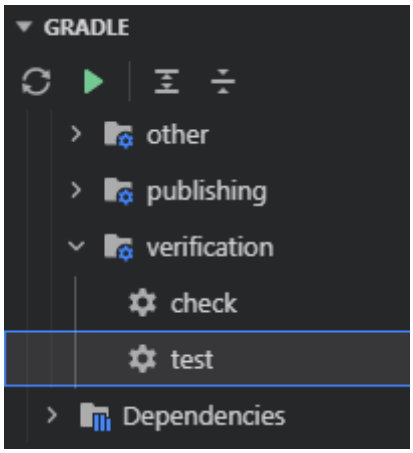
- 在测试类的代码编辑器中，单击测试类声明旁边的**Run**按钮(▶)（运行类中的所有测试）或者单个测试方法（仅运行单个测试）。如果需要调试测试，请右键单击**Run**按钮(▶)，然后从上下文菜单中选择**Debug Test** 。



- 使用**测试视图**管理和运行测试。
- 使用**测试启动配置**：**Run All Tests**（**JUnit**）和**Debug Tests**（**JUnit**）。
- 在命令面板（“Ctrl+Shift+P”/“Ctrl Ctrl”）中，搜索**Test**并使用与测试相关的命令，如**Run Tests in Current File**或**Run Test at Cursor**。



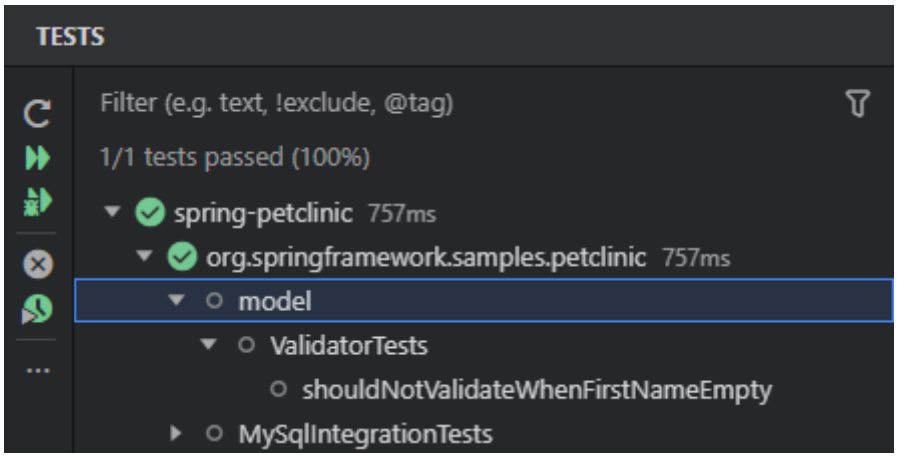
- 在Gradle项目中，在Gradle视图中，通过双击**test**来执行该任务。有关CodeArts IDE中**Gradle**集成的详细信息，请参阅**Gradle**。



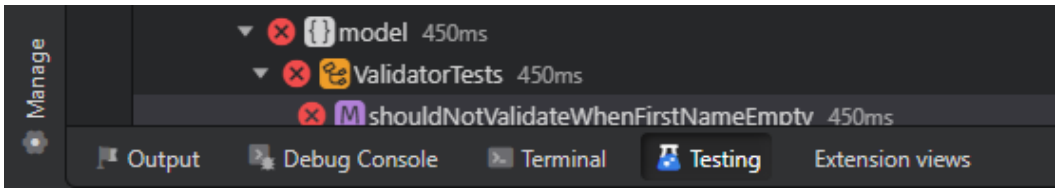
5.10.3.2 测试视图

5.10.3.2.1 简介

Tests视图列出了项目中的所有测试用例，让您可以运行它们并检查结果。



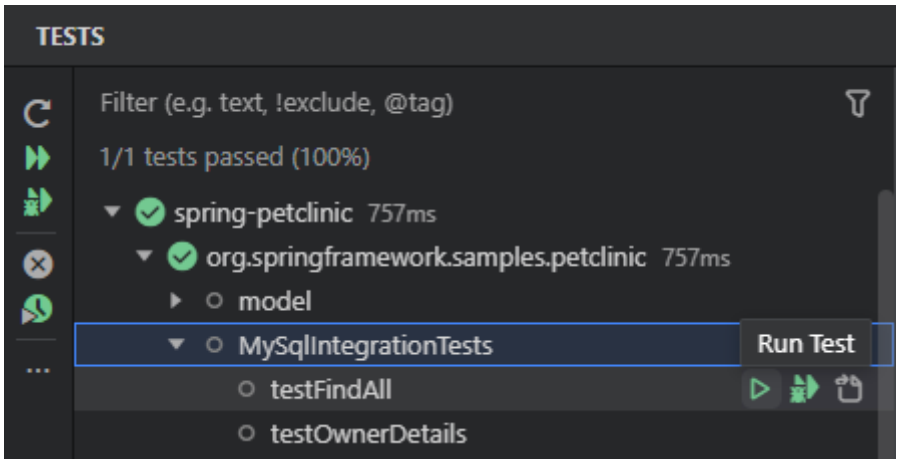
要打开**Tests**视图，请单击CodeArts IDE底部面板中的**Tests**按钮（ ）。



5.10.3.2.2 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**Run**按钮（ ）运行测试，或单击**Debug**按钮（ ）调试测试。



----结束

要运行或调试所有可用的测试，请单击**Tests**视图工具栏上的**Run Tests**（）或**Debug Tests**（）按钮。

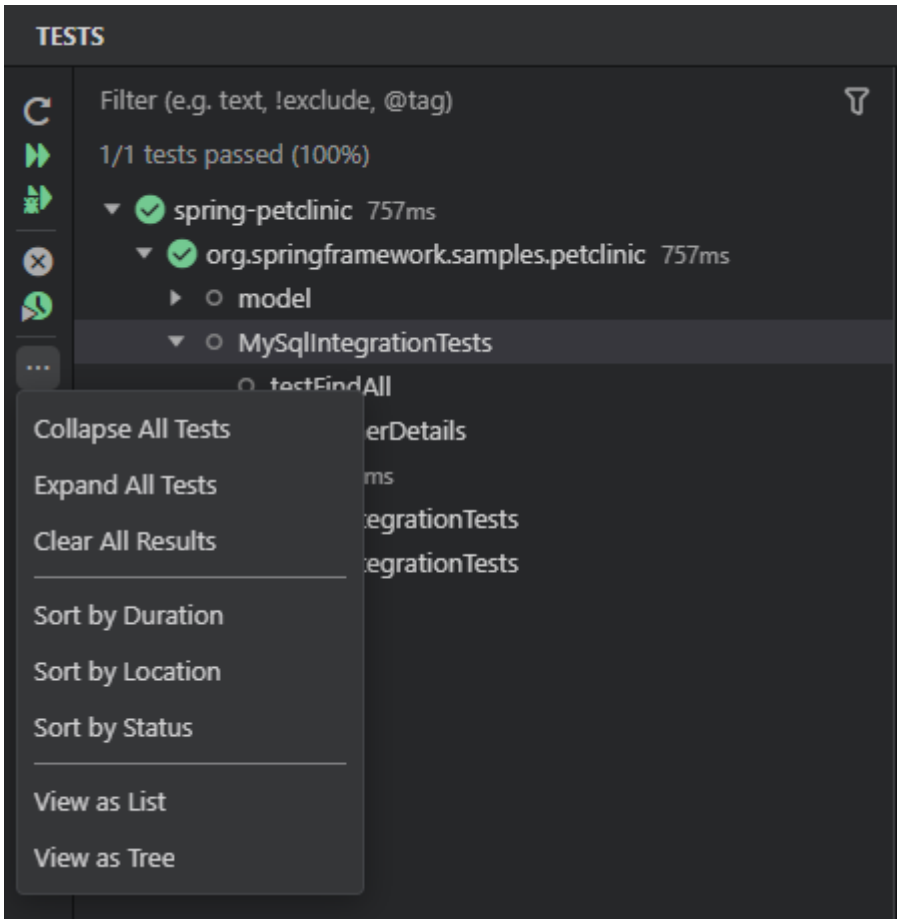
5.10.3.2.3 导航到测试类或方法

在**Tests**视图中，您可以直接从测试用例导航到相应的测试类或方法。要做到这一点，双击与类或方法对应的树节点，或将鼠标悬停在其上并单击**Go to Test**按钮（）。

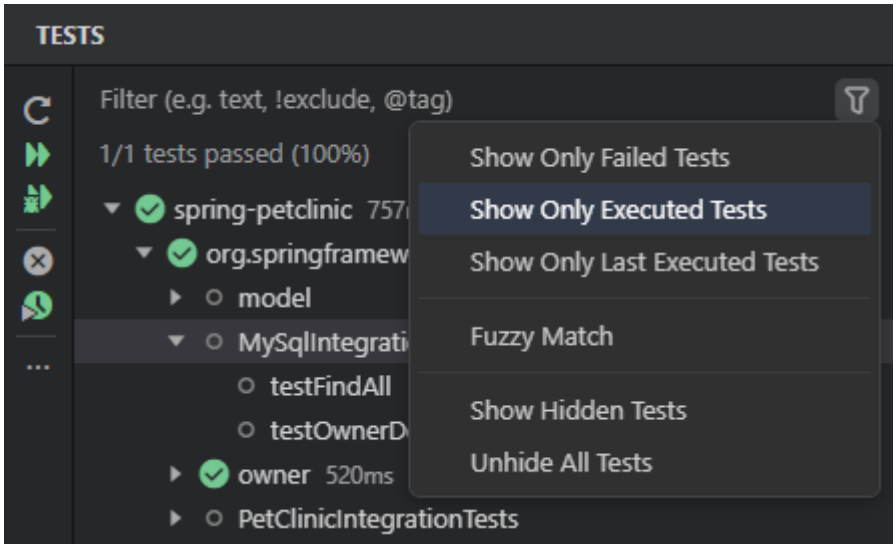
5.10.3.2.4 组织测试视图

Tests视图提供了多种功能，方便地组织测试显示。

使用左侧的**More Actions**按钮（）选择视图选项，如分组或排序。



Tests测试顶部的过滤字段允许您提供文本查询以按条件筛选测试列表。使用**More Filters**按钮（ ）应用其他过滤选项，例如仅查看失败的测试。

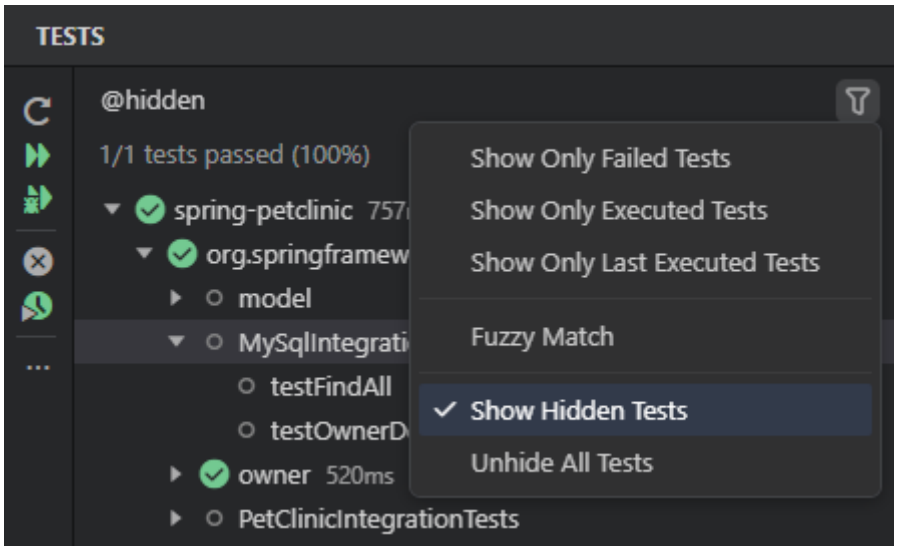


5.10.3.2.5 隐藏测试

如有必要，您可以从**Tests**视图中隐藏特定测试。要执行此操作，请右键单击与测试相对应的树节点，然后从上下文菜单中选择**Hide Test**。

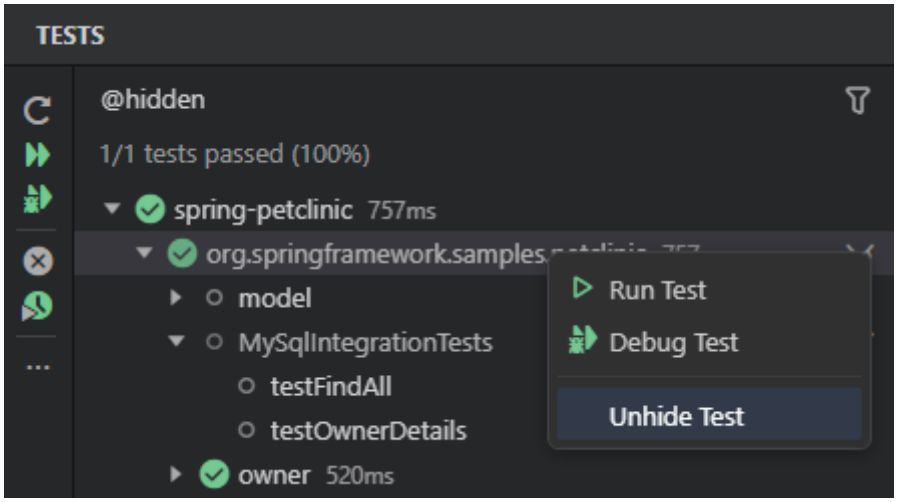
要取消隐藏测试，请执行以下操作：

步骤1 单击**Tests**视图中搜索字段的**More Filters**按钮（ ）。



步骤2 执行以下操作之一：

- 要取消隐藏所有隐藏的测试，请在弹出菜单中选择**Unhide All Tests**。
- 要取消隐藏特定测试，请在弹出菜单中选择**Show Hidden Tests**。在**Tests**视图中，显示的隐藏测试用**Hidden**图标（ ）标记。右键单击要取消隐藏的测试，然后从上下文菜单中选择**Unhide Test**。



----结束

5.10.3.3 测试启动配置

5.10.3.3.1 简介

CodeArts IDE允许您自定义运行测试用例的配置。为此，您可以将相应的**启动配置**添加到项目的**launch.json**中。

以下配置模板可用于运行和调试测试：

- **JUnit测试**

- TestNG测试

5.10.3.3.2 JUnit 测试

启动配置属性

在启动配置中，您可以选择相应的属性来运行单个测试方法、单个测试类、包中的所有测试或目录中的所有测试。

名称	描述
type	调试器的类型。对于运行和调试Java代码，应将其设置为**javadbg**。
name	启动配置的名称。
env	额外的环境变量。
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
vmOptions	JVM的额外选项。
method	完全限定的测试方法名称。
class	完全限定的测试类名称。
package	测试包名称。
directory	包含测试源代码的目录。默认情况下，此项设置为**${workspaceRoot}/src/test**。您可以使用变量来提供路径。

启动配置示例

您可以使用提供的示例作为工作启动配置示例。

运行**package.name**包中的所有测试：

```
{
    "type": "javadbg",
    "name": "JUnit Test (Package)",
    "request": "launch",
    "jUnit": {
        "package": "package.name"
    },
    "vmOptions": "-ea"
}
```

运行单个测试方法**qualified.method.name**：

```
{
    "type": "javadbg",
    "name": "JUnit Test (Method)",
    "request": "launch",
    "jUnit": {
        "method": "qualified.method.name"
```

```
    },
    "vmOptions": "-ea"
}
```

5.10.3.3.3 TestNG 测试

启动配置属性

在启动配置中，您可以选择相应的属性来运行单个测试方法、单个测试类、包中的所有测试或目录中的所有测试。

名称	描述
type	调试器的类型。对于运行和调试Java代码，应将其设置为**javadbg**。
name	启动配置的名称。
env	额外的环境变量。
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
vmOptions	JVM的额外选项。
method	完全限定的测试方法名称。
class	完全限定的测试类名称。
package	测试包名称。
directory	包含测试源代码的目录。默认情况下，此项设置为**${workspaceRoot}/src/test**。

启动配置示例

您可以使用提供的示例作为工作启动配置示例。

运行**package.name**包中的所有测试：

```
{
    "type": "javadbg",
    "name": "TestNG Test (Package)",
    "request": "launch",
    "testNG": {
        "package": "package.name"
    },
    "vmOptions": "-ea"
}
```

运行单个测试方法**qualified.method.name**：

```
{
    "type": "javadbg",
    "name": "TestNG Test (Method)",
    "request": "launch",
```



```
    "testNG": {
        "method": "qualified.method.name"
    },
    "vmOptions": "-ea"
}
```

5.11 启动配置

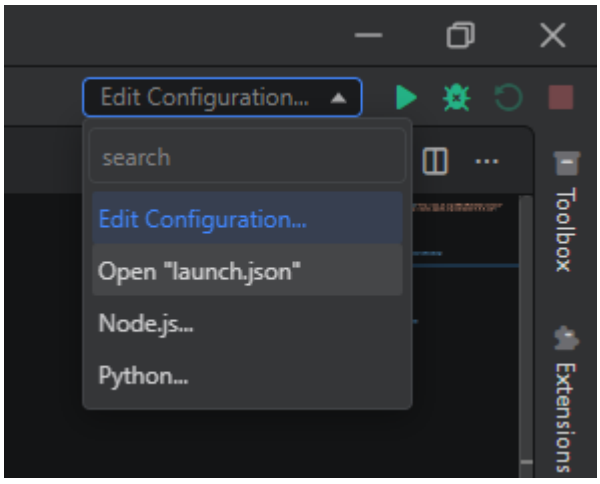
5.11.1 简介

启动配置允许您配置和保存各种场景的运行或调试设置详细信息。CodeArts IDE将配置信息保存在项目根文件夹下的**.arts**文件夹中的**launch.json**文件中。

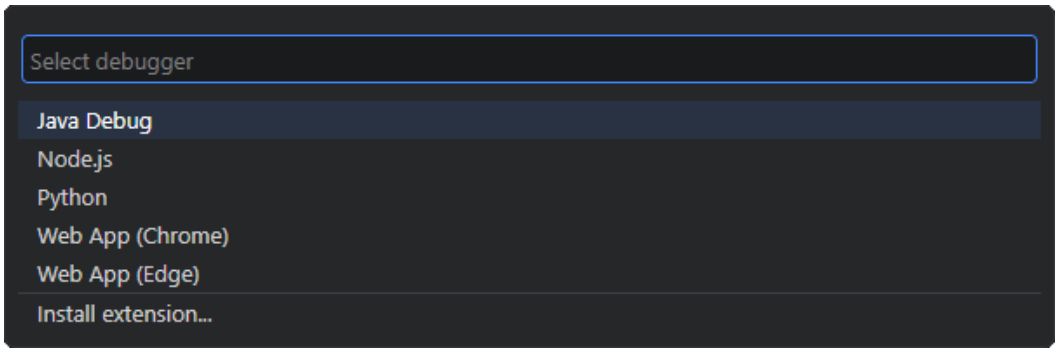
在Java环境中，可以使用以下配置模板：

- **Java类**
- **JAR应用程序**
- **Gradle任务**
- **Maven目标**
- **JUnit测试**
- **TestNG测试**
- **远程调试**

要创建**launch.json**文件，请在CodeArts IDE主工具栏上的列表中选择“**打开launch.json**”。

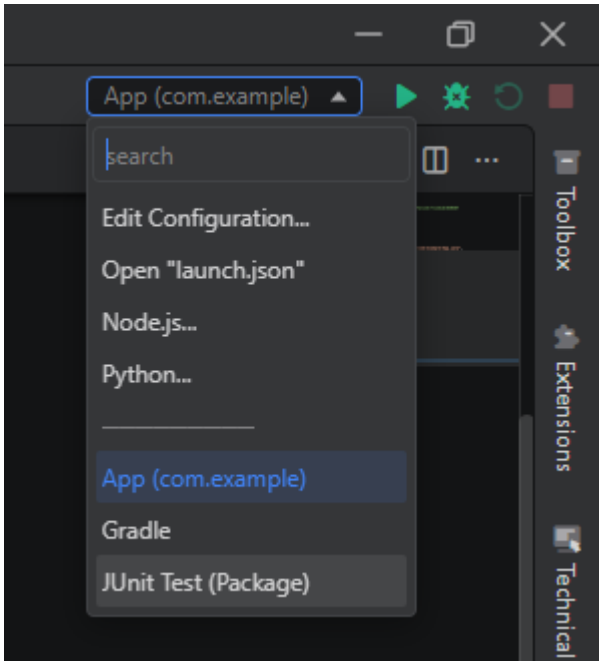


CodeArts IDE尝试自动检测您的调试环境，但如果失败，您将需要手动选择。在Java环境中，选择**Java Debug**。



CodeArts IDE创建一个**launch.json**文件，并根据检测到的环境填充默认内容。请检查所有自动生成的值，确保它们适用于您的项目和调试环境。

在**launch.json**中定义的所有启动配置都可以从CodeArts IDE主工具栏上的列表中选择。

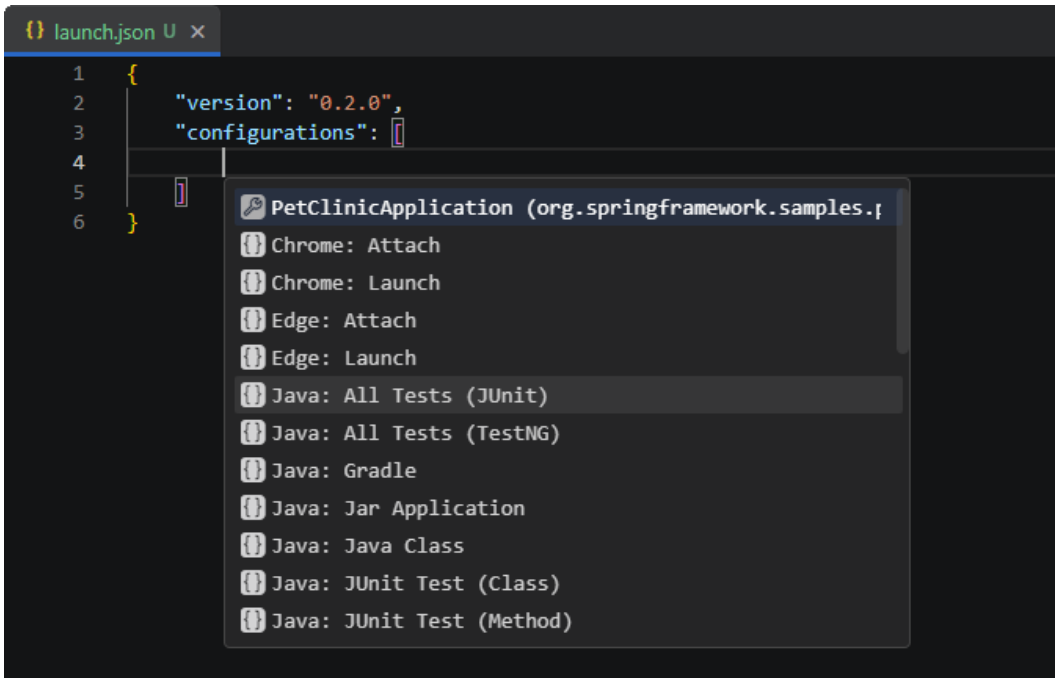


向现有的 launch.json 添加新的配置

步骤1 执行以下任一操作：

- 在**launch.json**编辑器中，单击编辑器右下角的 **Add Configuration**按钮，或将光标放置在**configurations**数组内，并使用代码完成（“Ctrl+I” / “Ctrl+Space” / “Ctrl+Shift+Space”（IDEA键盘映射））。
- 在CodeArts IDE主工具栏上的配置列表中选择**Add Configuration**。

步骤2 在弹出的建议列表中，选择要使用的启动配置模板。



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

----结束

变量替换

CodeArts IDE将常用路径和其他值作为变量提供，并支持在**launch.json**中的字符串中进行变量替换，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。

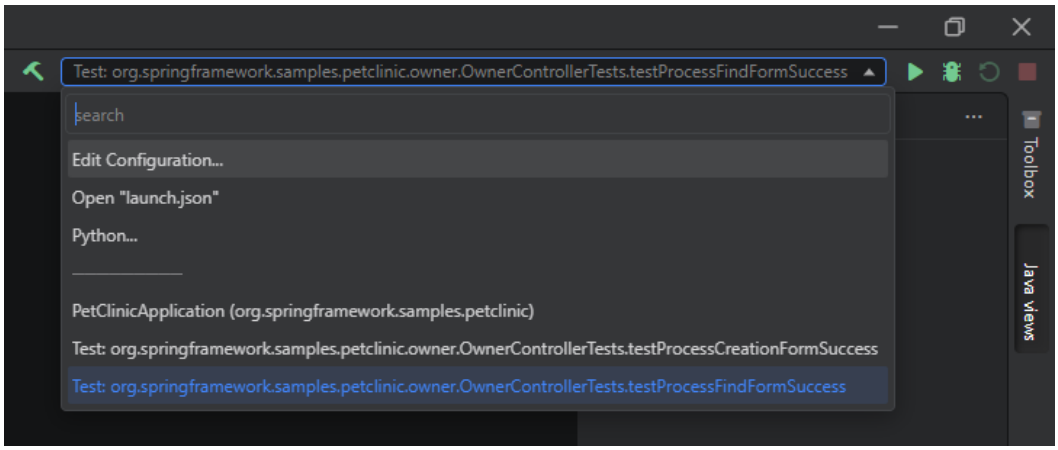
临时和永久启动配置

当您从编辑器边栏手动运行类或方法时，CodeArts IDE会自动创建相应的启动配置，并在配置列表中显示。这些配置默认为临时配置：CodeArts IDE根据指定的限制（默认为10）保留它们的数量，并在超过此限制时自动删除最少使用的配置。

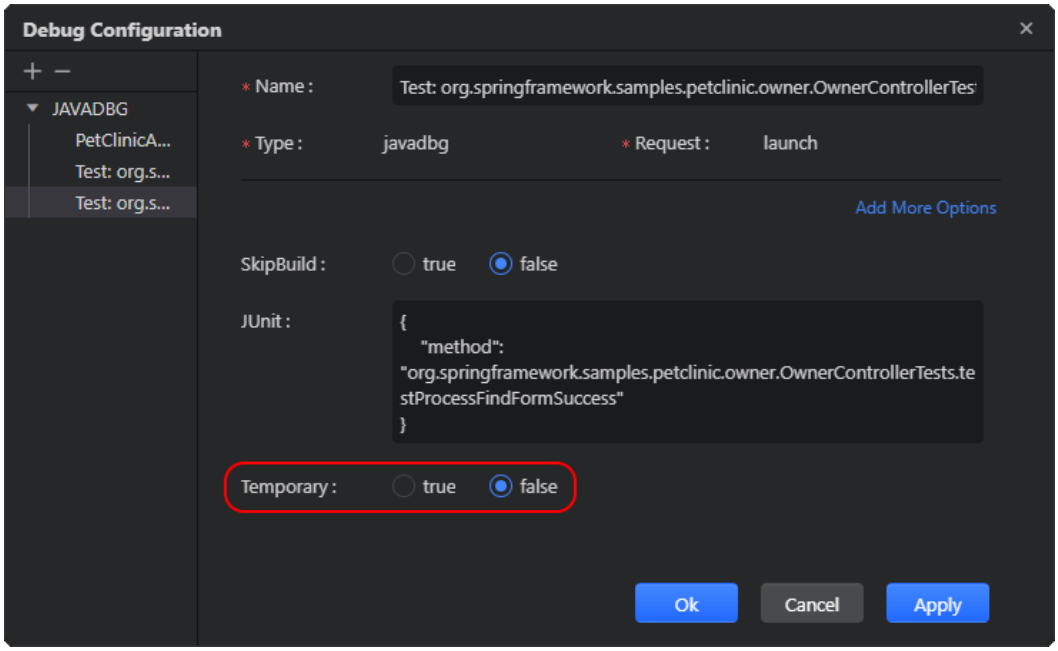
将临时启动配置保存为永久配置

您可以将临时启动配置保存为永久配置，以防止其被删除。

步骤1 在CodeArts IDE主工具栏上的配置列表中，选择**Edit Configuration**。



步骤2 在打开的**Debug Configuration**中，在左侧的配置列表中，选择要保存为永久配置的配置。然后，在配置参数中，将**Temporary**切换为**False**。



----结束

或者，您可以在**launch.json**中找到相应的启动配置记录，并为其提供**"temporary": false**顶级属性，例如：

```
{
  "type": "javadbg",
```

```
  "name": "Java Class",
  "request": "launch",
  "mainClass": {
     "name": "com.example.Main",
     "console": "integrated"
  },
  "temporary": false
}
```

调整临时启动配置的限制

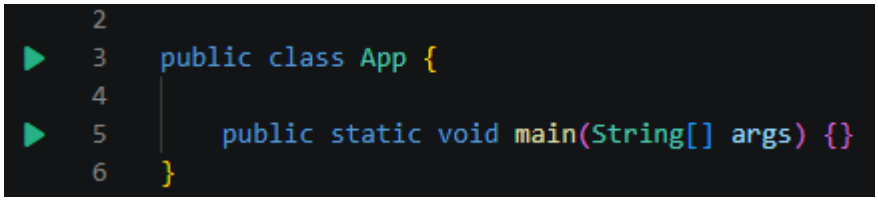
默认情况下，CodeArts IDE根据指定的限制（默认为10）保留临时启动配置的数量，并在超过限制时自动删除最少使用的配置。如果需要，您可以通过**java.executedConfigurationsLimit设置**来调整此限制。如果设置为零，CodeArts IDE不会创建任何临时启动配置，并删除任何现有的临时启动配置。

5.11.2 Java 类

使用以下启动配置来运行应用程序的Main类。

约束与限制

要快速运行一个应用程序而不必手动创建一个启动配置，只需在Main类的代码编辑器中，单击**main()**或类声明旁边的**Run**按钮（▶）即可。CodeArts IDE将自动创建相应的**temporary launch configuration**，并在配置列表中显示出来。



启动配置属性

名称	描述
type	描述 type 调试器的类型。对于运行和调试Java代码，应将其设置为**javadbg**。
name	启动配置名称。
env	额外的环境变量
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。
temporary	指示启动配置是否为临时的（设置为**true**）还是永久的（设置为**false**）。如果临时启动配置数量超过指定限制，CodeArts IDE会自动删除最不常用的配置。有关详细信息，请参阅**启动配置**。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
vmOptions	JVM的额外选项。
name	（在**mainClass**节点下指定）限定类名。

名称	描述
sourcePath	（name的替代）类源文件的路径。您可以使用**变量**来提供路径。
args	传递给**main()**方法的参数数组（**[arg1，arg2，...]**）。
console	在 **Debug Console**（**internal**）或**集成终端**（**integrated**）中显示调试输出。

启动配置示例

您可以将提供的示例作为一个可工作的启动配置示例。

```
{
   "type": "javadbg",
   "name": "Java Class",
   "request": "launch",
   "mainClass": {
      "name": "com.example.Main",
      "console": "integrated"
   }
}
```

5.11.3 JAR 应用

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

名称	描述
type	描述 type 调试器的类型。对于运行和调试Java代码，应将其设置为javadbg。
name	启动配置名称。
env	额外的环境变量
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。
temporary	指示启动配置是否为临时的（设置为**true**）还是永久的（设置为**false**）。如果临时启动配置数量超过指定限制，CodeArts IDE会自动删除最不常用的配置。有关详细信息，请参阅**启动配置**。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
vmOptions	JVM的额外选项。
path	JAR文件的路径。您可以使用变量来提供路径。
console	在**调试控制台**（**internal**）或**集成终端**（**integrated**）中显示调试输出。
args	程序传递的参数。

启动配置示例

您可以使用提供的示例作为工作的启动配置示例。

```
{
   "type": "javadbg",
   "name": "Jar Application",
   "request": "launch",
   "jar": {
      "path": "${workspaceRoot}/path/to/demo.jar",
      "console": "integrated"
   }
}
```

5.11.4 Gradle 任务

使用此启动配置来运行一个或多个Gradle任务。

启动配置属性

名称	描述
type	描述 type 调试器的类型。对于运行和调试Java代码，应将其设置为javadbg。
name	启动配置名称。
env	额外的环境变量
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。
temporary	指示启动配置是否为临时的（设置为**true**）还是永久的（设置为**false**）。如果临时启动配置数量超过指定限制，CodeArts IDE会自动删除最不常用的配置。有关详细信息，请参阅**启动配置**。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
vmOptions	JVM的额外选项。
scriptArgs	传递给Gradle的参数的数组。
tasks	要运行的Gradle任务。提供一个对象数组，每个对象都有一个**名称**（任务名称）和一个可选的**args**数组，其中包含任务的参数。

启动配置示例

您可以将提供的示例作为一个可行的启动配置示例。

```
{
   "type": "javadbg",
   "name": "Gradle",
   "request": "launch",
   "skipBuild": true,
   "gradle": {
      "scriptArgs": [
         "--info"
```

```
        ],
        "tasks": [
            "build"
        ]
    }
}
```

5.11.5 Maven 任务

使用此启动配置来运行一个或多个Maven任务。

启动配置属性

名称	描述
type	描述 type 调试器的类型。对于运行和调试Java代码，应将其设置为javadbg。
name	启动配置名称。
env	额外的环境变量
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。
temporary	指示启动配置是否为临时的（设置为**true**）还是永久的（设置为**false**）。如果临时启动配置数量超过指定限制，CodeArts IDE会自动删除最不常用的配置。有关详细信息，请参阅**启动配置**。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
vmOptions	JVM的额外选项。
goals	Maven的目标是运行。可以将目标名称作为单个字符串或字符串数组提供。

启动配置示例

您可以使用提供的示例作为工作的启动配置示例。

```
{
    "type": "javadbg",
    "name": "Maven",
    "request": "launch",
    "skipBuild": true,
    "maven": {
        "goals": [
            "install"
        ]
    }
}
```

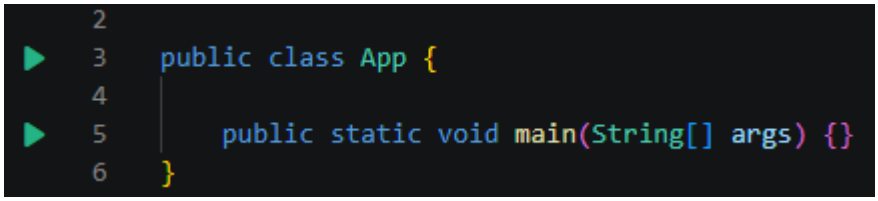
5.11.6 JUnit 测试

使用此启动配置来运行JUnit测试。

要快速运行测试而无需手动创建启动配置，请在测试类的代码编辑器中，单击测试类声明旁边的运行按钮（▶）（以运行类中的所有测试），或者单击测试方法（以仅运行

单个测试）。要快速运行一个应用程序而不必手动创建一个启动配置，只需在Main类的代码编辑器中，单击**main()**或类声明旁边的**Run**按钮即可。

CodeArts IDE将自动创建相应的**temporary launch configuration**，并在配置列表中显示出来。



启动配置属性

在启动配置中，您只能指定以下属性之一：**方法（method）**、**类（class）**、**包（package）**或**目录（directory）**，以运行单个测试方法、单个测试类、包中的所有测试或目录中的所有测试。

名称	描述
type	描述 type 调试器的类型。对于运行和调试Java代码，应将其设置为javadbg。
name	启动配置名称。
env	额外的环境变量
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。
temporary	指示启动配置是否为临时的（设置为**true**）还是永久的（设置为**false**）。如果临时启动配置数量超过指定限制，CodeArts IDE会自动删除最不常用的配置。有关详细信息，请参阅**启动配置**。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
vmOptions	JVM的额外选项。
method	完全限定的测试方法名称。
class	完全限定的测试类名称。
package	测试包名称。
directory	包含测试源代码的目录。默认情况下，此项设置为**${workspaceRoot}/src/test**。您可以使用**变量**来提供路径。

启动配置示例

您可以使用提供的示例作为工作启动配置的示例。

运行来自**package.name**包的所有测试：

```
{
   "type": "javadbg",
```

```
    "name": "JUnit Test (Package)",
    "request": "launch",
    "jUnit": {
        "package": "package.name"
    },
    "vmOptions": "-ea"
}
```

运行单个测试方法**qualified.method.name**：

```
{
    "type": "javadbg",
    "name": "JUnit Test (Method)",
    "request": "launch",
    "jUnit": {
        "method": "qualified.method.name"
    },
    "vmOptions": "-ea"
}
```

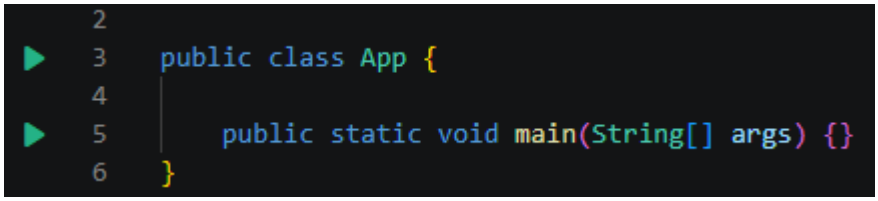
5.11.7 TestNG 测试

使用此启动配置来运行JUnit测试。

约束与限制

要快速运行测试而无需手动创建启动配置，请在测试类的代码编辑器中，单击测试类声明旁边的运行按钮（▶）（以运行类中的所有测试），或者单击测试方法（以仅运行单个测试）。要快速运行一个应用程序而不必手动创建一个启动配置，只需在Main类的代码编辑器中，单击**main()**或类声明旁边的**Run**按钮即可。

CodeArts IDE将自动创建相应的**启动配置**，并在配置列表中显示出来。



启动配置属性

在启动配置中，您只能指定以下属性之一：**方法（method）**、**类（class）**、**包（package）**或**目录（directory）**，以运行单个测试方法、单个测试类、包中的所有测试或目录中的所有测试。

名称	描述
type	描述 type 调试器的类型。对于运行和调试Java代码，应将其设置为javadbg。
name	启动配置名称。
env	额外的环境变量
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。

名称	描述
temporary	指示启动配置是否为临时的（设置为true）还是永久的（设置为**false**）。如果临时启动配置数量超过指定限制，CodeArts IDE会自动删除最不常用的配置。有关详细信息，请参阅**启动配置**。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
vmOptions	JVM的额外选项。
method	完全限定的测试方法名称。
class	完全限定的测试类名称。
package	测试包名称。
directory	包含测试源代码的目录。默认情况下，此项设置为**${workspaceRoot}/src/test**。您可以使用**变量**来提供路径。

启动配置示例

您可以使用提供的示例作为工作启动配置的示例。

运行来自**package.name**包的所有测试：

```
{
    "type": "javadbg",
    "name": "TestNG Test (Package)",
    "request": "launch",
    "testNG": {
        "package": "package.name"
    },
    "vmOptions": "-ea"
}
```

运行单个测试方法**qualified.method.name**：

```
{
    "type": "javadbg",
    "name": "TestNG Test (Method)",
    "request": "launch",
    "testNG": {
        "method": "qualified.method.name"
    },
    "vmOptions": "-ea"
}
```

5.11.8 远程调试

使用此启动配置可以通过连接到远程JVM或使调试器监听传入连接来进行远程调试。

启动配置属性

名称	描述
type	描述 type 调试器的类型。对于运行和调试Java代码，应将其设置为javadbg。
name	启动配置名称。
env	额外的环境变量
skipBuild	跳过程序的构建过程（设置为**true**）或不跳过（设置为**false**）。
temporary	指示启动配置是否为临时的（设置为**true**）还是永久的（设置为**false**）。如果临时启动配置数量超过指定限制，CodeArts IDE会自动删除最不常用的配置。有关详细信息，请参阅**启动配置**。
killPrevSession	终止具有相同名称的先前运行会话（设置为**true**），或中止启动（设置为**false**）。
debuggerMode	调试器模式，可以设置为**attach**（连接到远程JVM）或listen（监听传入连接）。默认情况下，使用attach模式。
autoRestart	仅在**debuggerMode**设置为**listen**时可用，定义调试器在远程JVM断开连接后是否自动重启。默认情况下，使用**false**。
useSocketTransport	定义是否使用套接字传输来连接进程。默认情况下，使用**true**。否则，当设置为**false**时，使用共享内存。
host	主机应用程序运行的机器的地址。默认情况下，使用**127.0.0.1**。
port	目标机器上的连接端口。默认情况下，使用**5005**。

启动配置示例

您可以使用提供的示例作为远程调试场景的示例。

主机应用程序的启动配置是一个常规的**Java类**配置。它应该包含在**vmOptions**下提供的特殊参数，以便应用程序使用调试代理启动，并且调试器能够连接到它。

```
{
    "type": "javadbg",
    "name": "Java Class",
    "request": "launch",
    "vmOptions": "-
agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=127.0.0.1:5005",
    "mainClass": {
        "name": "com.example.App",
        "console": "internal"
    }
}
```

远程调试的启动配置应该使用提供给主机应用程序启动配置的连接参数。

```
{
    "type": "javadbg",
    "name": "Remote Debug (Attach to remote JVM)",
    "request": "launch",
    "skipBuild": true,
```

```
    "remote": {
        "debuggerMode": "attach",
        "useSocketTransport": true,
        "host": "127.0.0.1",
        "port": "5005"
    }
}
```

6 Python

6.1 简介

6.1.1 安装 Python

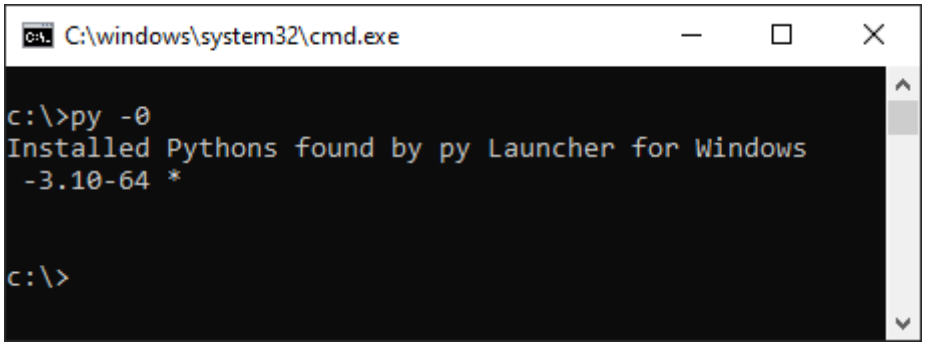
在开始在CodeArts IDE上使用Python前，请确保您已在计算机上安装了Python。

- 在Windows上，您需要手动**下载并安装Python解释器**。
- 在Linux上，您可以使用内置的Python 3安装，但为了安装其他Python包，您还需要通过**get-pip.py**安装 “pip” 包管理器。

接下来，运行以下命令验证Python安装：

- 在Windows上，请在命令提示符中运行：
  ```
  python --version
  ```
- 在Linux上，请在终端中运行：
  ```
  python3 --version
  ```

如果Python安装成功，输出将显示对应Python版本号显示。您也可以使用 “py -0” 命令来查看已安装的Python版本。默认解释器将由星号（*）标记。

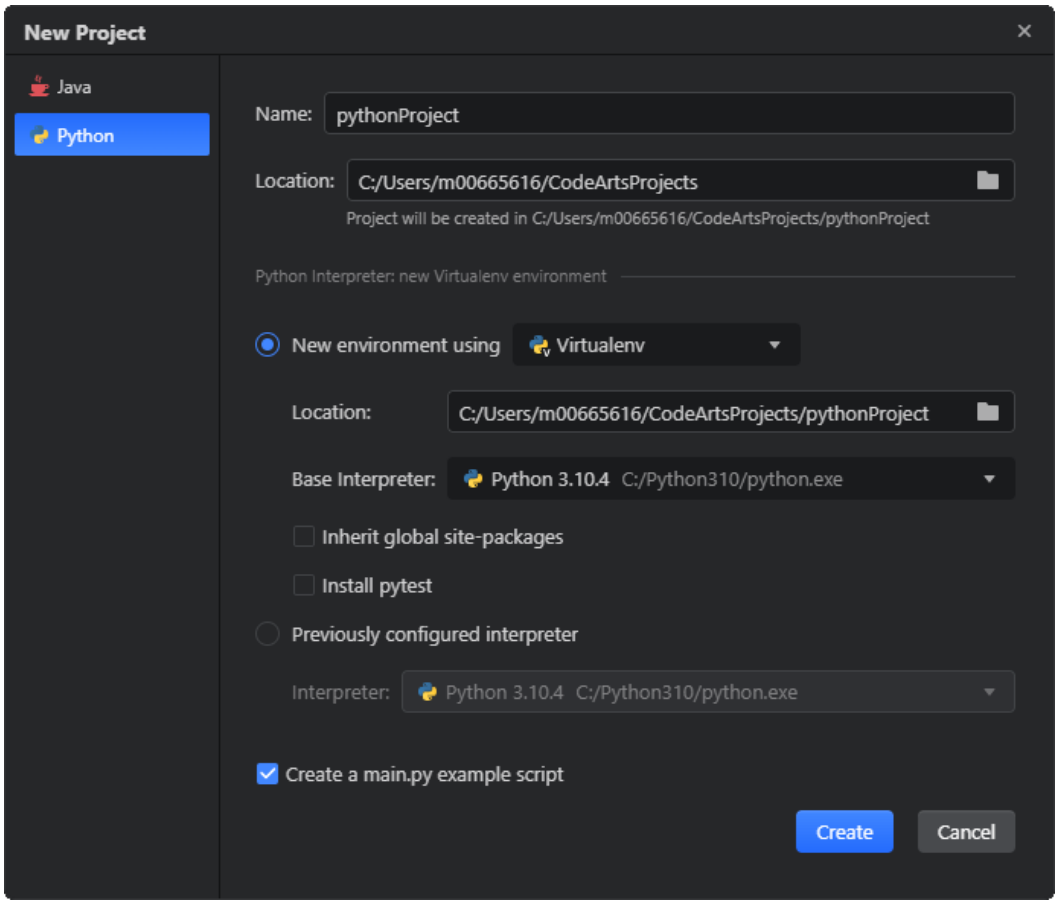


6.1.2 新建 Python 项目

CodeArts IDE提供了一个Python项目向导，帮助您更轻松地创建新项目并配置环境。

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。

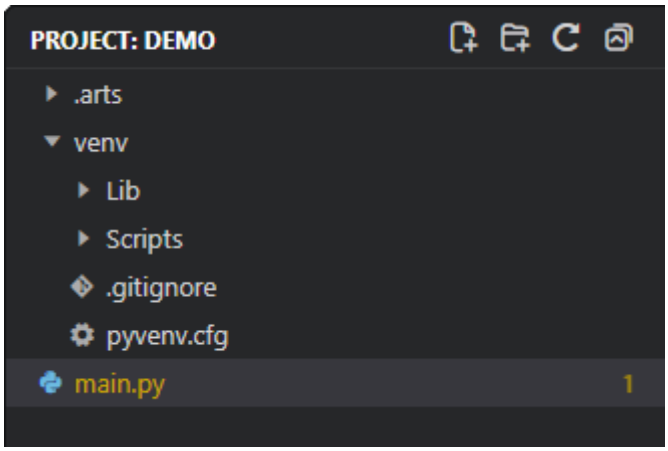


1. 设置项目名称和路径。
2. 在“**新环境使用**“的下拉框中选择使用“**Virtualenv**“选项，让CodeArts IDE 为您创建一个隔离的、特定于项目的 Virtualenv Python环境。这样就可以使您在项目级别安装包，不会污染全局Python。
   a. 保留创建环境的默认位置。
   b. 确保在基础解释器列表中已选择一个解释器。通常CodeArts IDE会自动从标准安装位置检测解释器位置并展示在此处。

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。

----结束

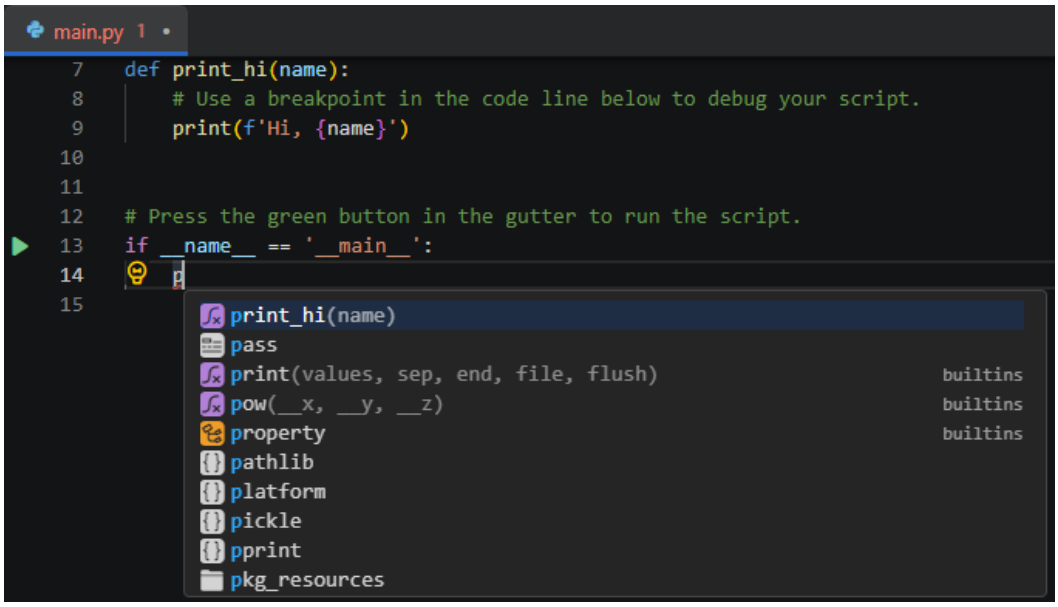


说明

有关在CodeArts IDE中管理Python项目的更多详细信息，请参阅**开始Python工程**。

6.1.3 使用代码提示

在编写代码时，CodeArts IDE会为您的项目文件、内置模块和第三方模块提供**代码补全**建议。代码补全功能会在您键入时自动显示符号和文档，您也可以通过手动按下“Ctrl+I”/“Ctrl+Space”/“Ctrl+Shift+空格键”来随时触发它。此外，您还可以将鼠标悬停在标识符上以获取更多相关信息。

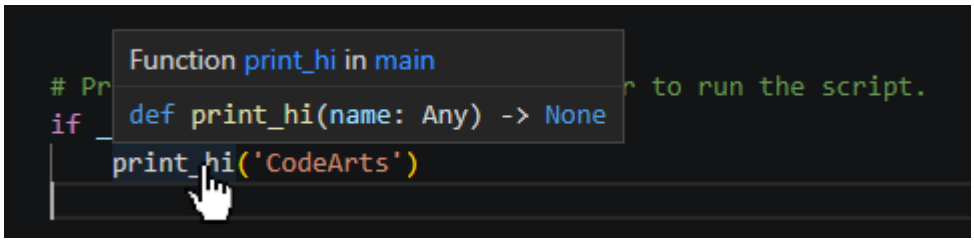


说明

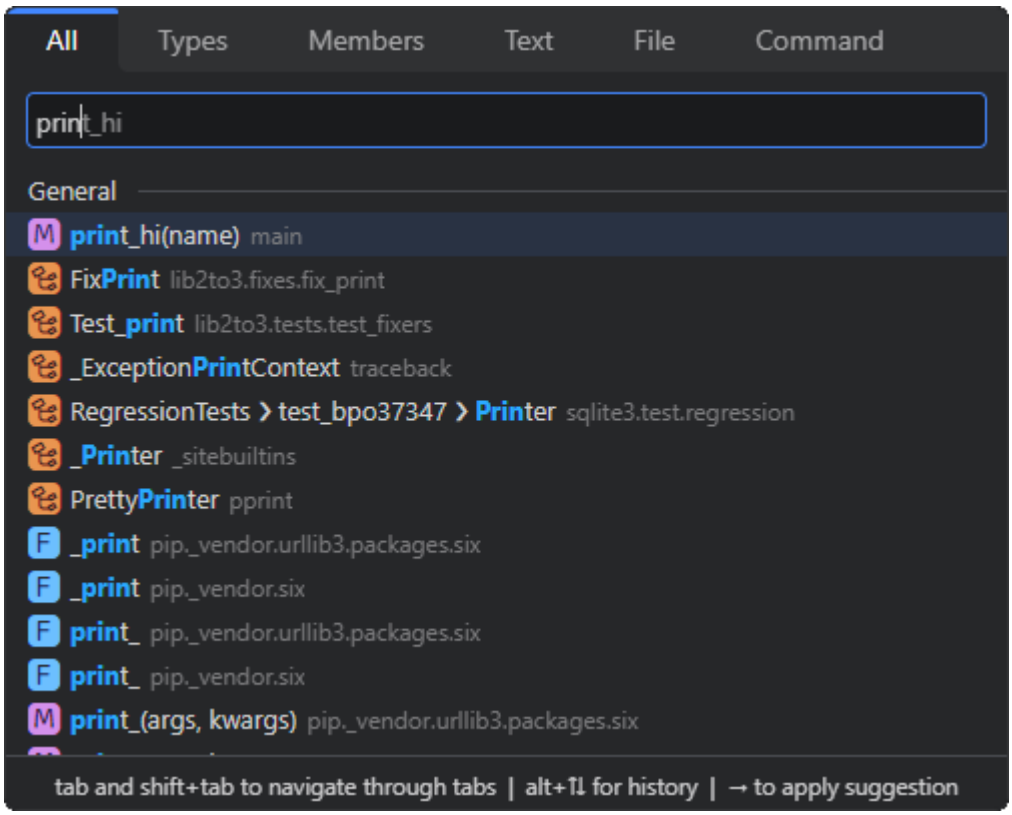
除了代码补全外，CodeArts IDE还提供了其他如导航和重构的重要编码辅助功能。要获取更多详情，请查看**编辑代码**、**浏览代码**和**搜索代码**等相关文档。

6.1.4 浏览代码

CodeArts提供了丰富的代码导航功能。例如，您可以在代码编辑器中将鼠标悬停在符号上，以查看其快速信息。通过“Ctrl+单击”符号，或按下“F3”/“Alt+F11”（IDEA快捷键）/“F4”（IDEA快捷键）/“Ctrl+Enter”（IDEA快捷键）/“Ctrl+B”（IDEA快捷键），快速导航到符号的声明位置。



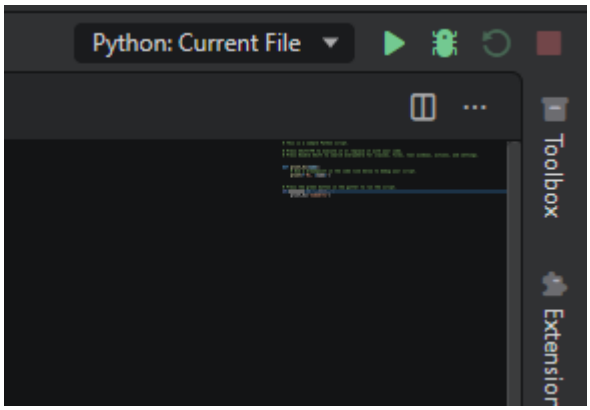
您也可以通过按下 “Shift+Shift” / “Ctrl+Shift+A” 来启动CodeArts IDE SmartSearch功能，立即搜索并导航到任何项目位置，查找和执行任何CodeArts IDE命令。



6.1.5 运行代码

您可以通过以下的任意方式来使用当前选定的解释器运行代码：

- 单击CodeArts IDE主工具栏上的“**开始执行（不调试）**”按钮（▶），以启动内置的“当前文件”启动配置。

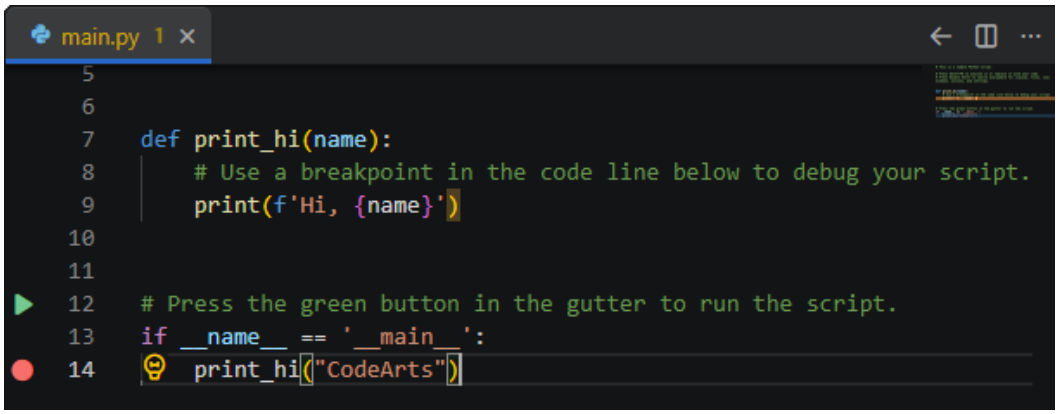


- 在代码编辑器的任意位置右键单击，并选择“**在终端中运行 Python 文件**”。如果您在选定的代码块上调用此命令，则可以仅运行该部分代码。
- 在资源管理器中右键单击Python文件，并从上下文菜单中选择“**在终端中运行 Python 文件**”。

6.1.6 调试代码

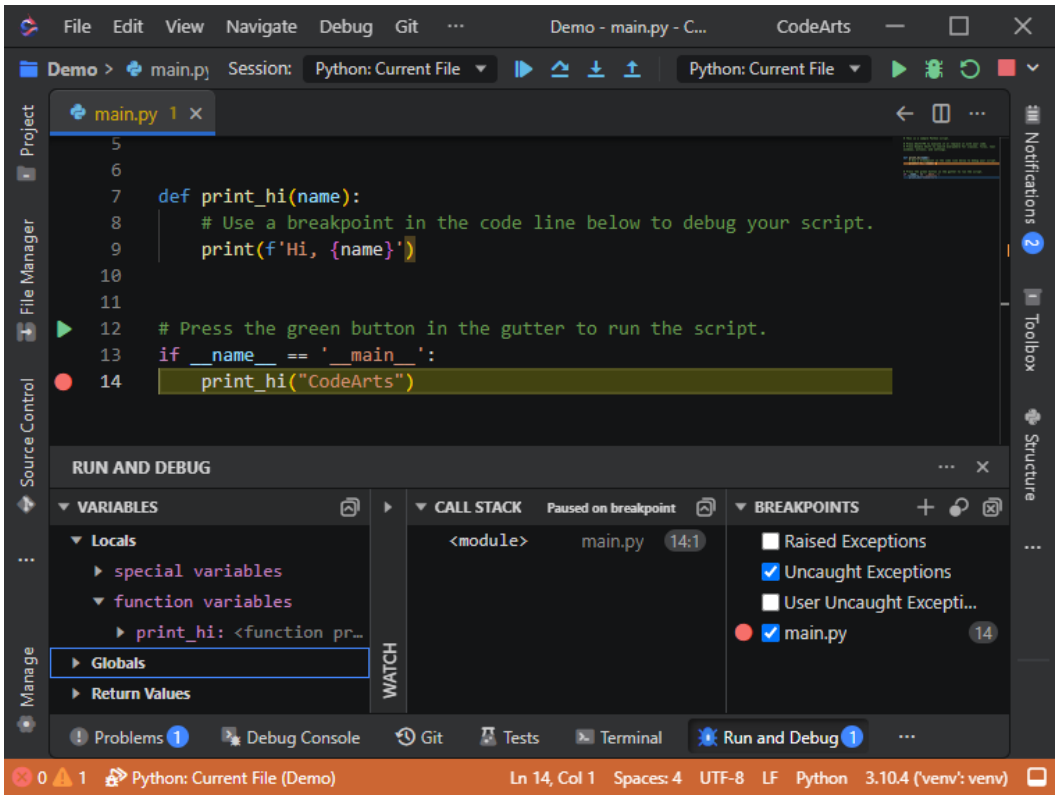
CodeArts的Python扩展提供了调试支持，让您能够设置断点、检查数据，在逐步执行程序时使用调试控制台。

对于快速开始项目，您可以在 “main.py” 文件的第 14 行设置一个断点。将光标悬置在 “print_hi” 调用上，然后按下 “F9” / “Ctrl+Shift+B” / “Ctrl+F8”（IDEA快捷键）。您也可以通过单击编辑器左侧的行号旁的空白区域来设置断点。



接下来需要初始化调试器，您可以按下 “F5” / “F11” / “Shift+F9”（IDEA快捷键），或者单击CodeArts IDE主工具栏上的“**开始调试**”按钮（ ）。

CodeArts IDE将启动内置的“当前文件”启动配置，调试器将在包含断点的那一行暂停运行。此时您可以使用调试工具栏上的命令来控制程序执行，并查看“**运行和调试**”视图中的“**变量**”来检查变量。



说明

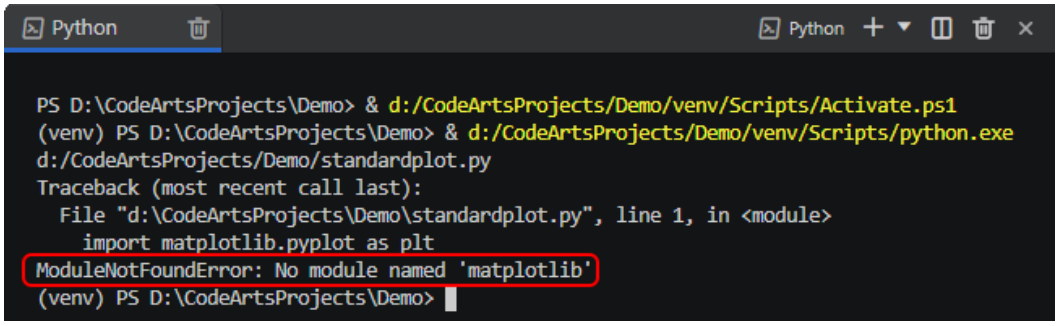
要了解在CodeArts IDE中调试Python的更多详细信息，请参阅“**Python 调试**”文档。

6.1.7 安装依赖

在本示例中，我们将引入“matplotlib”和“numpy”这两个第三方包，创建一个图标。首先，我们创建一个名为“standardplot.py”的新文件，并将以下代码粘贴进去：

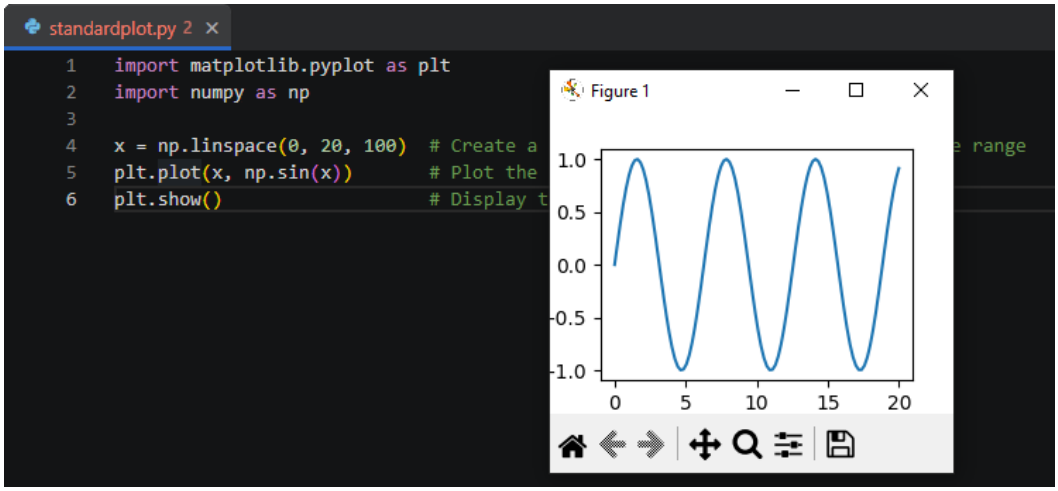
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 20, 100)  # Create a list of evenly-spaced numbers over the range
plt.plot(x, np.sin(x))       # Plot the sine of each x point
plt.show()                   # Display the plot
```

如果您在未安装依赖的情况下按照**运行代码**部分所述尝试运行此示例，CodeArts IDE将显示一个错误消息，指示所需的包不可用。



您可以使用pip将所需的包安装到环境中。打开内置终端，运行“pip install matplotlib”命令（Windows）或“pip3 install <package_name>”命令（Linux）。由于numpy包是matplotlib的依赖项，因此安装matplotlib时numpy会被同时安装。

安装完依赖包后，您可以重新运行代码示例，现在它应该能够按预期工作。



说明

有关如何使用Python环境的更多详细信息，请参阅“**构建环境**”文档。

6.1.8 测试代码

Python扩展支持使用**unittest**和**pytest**框架进行测试。CodeArts可以帮助您配置框架集成，并提供专用的“**测试**”视图，让您能够方便地识别和运行测试。

以下是一个如何创建和运行一个unittest测试的示例。

步骤1 创建一个测试对象，也就是新建一个名为 “inc_dec.py” 的文件，它包含以下内容：

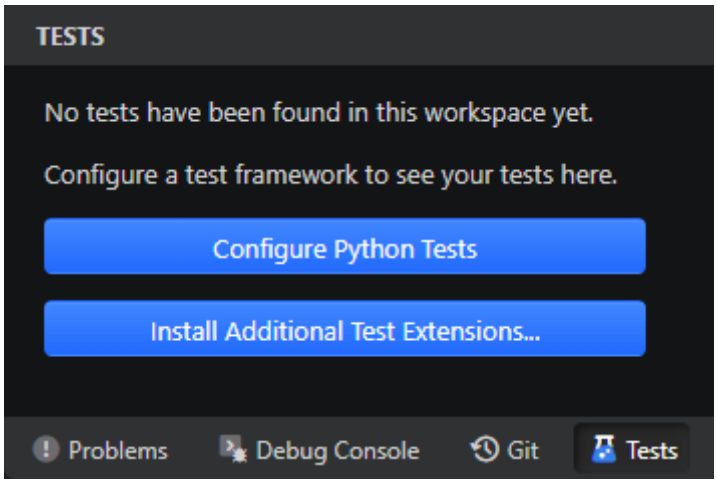
```
def increment(x):
    return x + 1

def decrement(x):
    return x - 1
```

步骤2 创建一个 “unittest” 测试来覆盖这个测试对象，即创建一个名为“inc_decunittest.py” 的文件，包含以下内容：

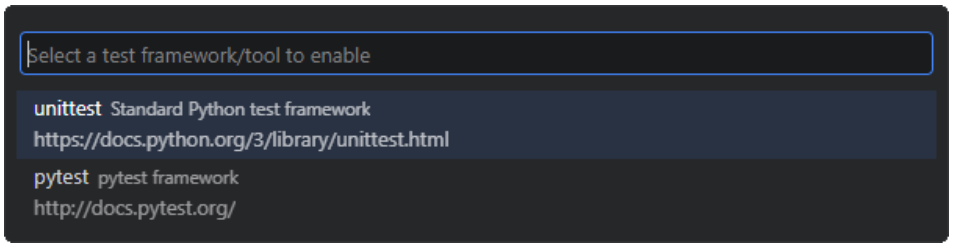
```
import inc_dec    # The code to test
import unittest   # The test framework

class Test_TestIncrementDecrement(unittest.TestCase):
    def test_increment(self):
        self.assertEqual(inc_dec.increment(3), 4)

    def test_decrement(self):
        self.assertEqual(inc_dec.decrement(3), 4)

if __name__ == '__main__':
    unittest.main()
```

步骤3 在CodeArts IDE中配置unittest框架集成。

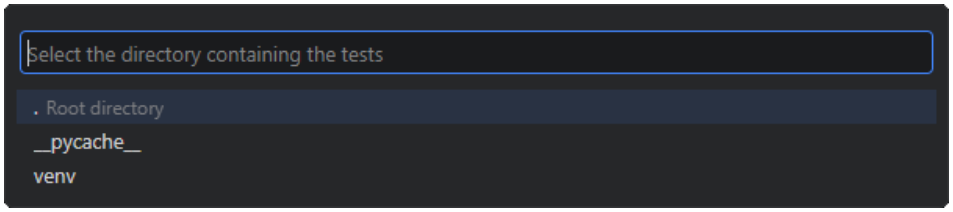
1. 单击CodeArts IDE底部面板中的“**测试**”按钮（ ）打开“**测试**”视图，单击“**Configure Python Tests**”按钮。



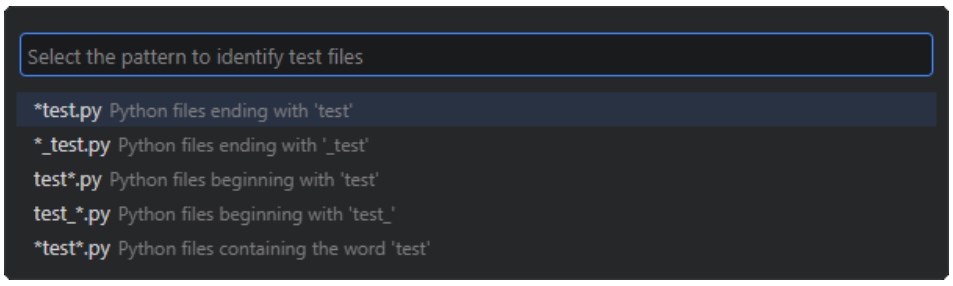
2. 在弹出的对话框中，选择您想要启用集成的测试框架，在本例中是 “unittest”。



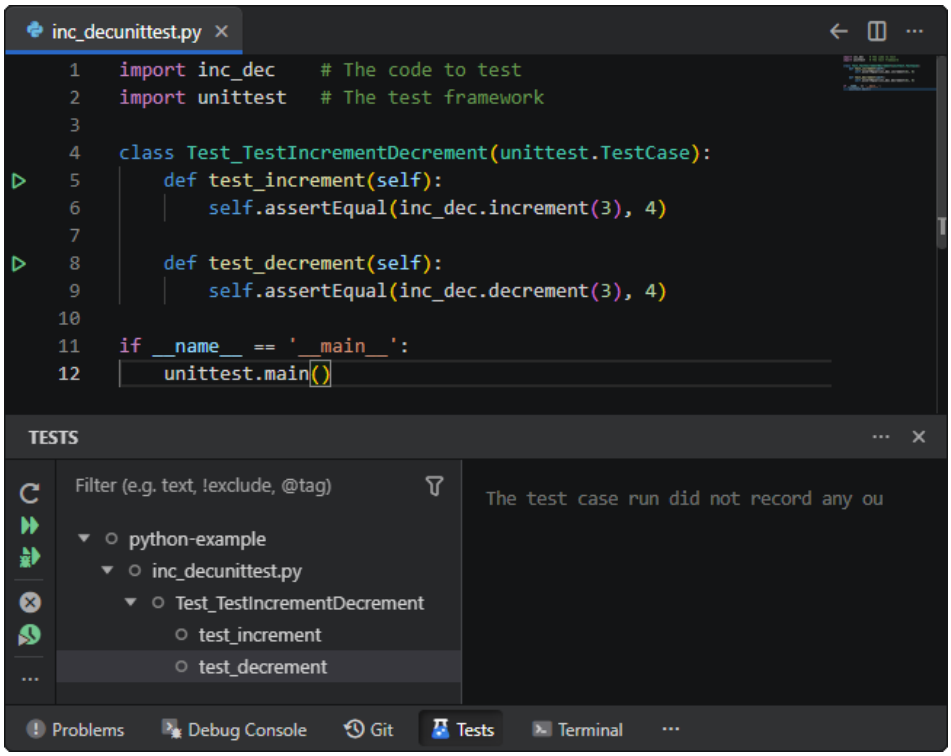
3. 在接下来的对话框中，选择包含测试源文件的项目文件夹。在我们的例子中，是项目根文件夹（ “.” ）。



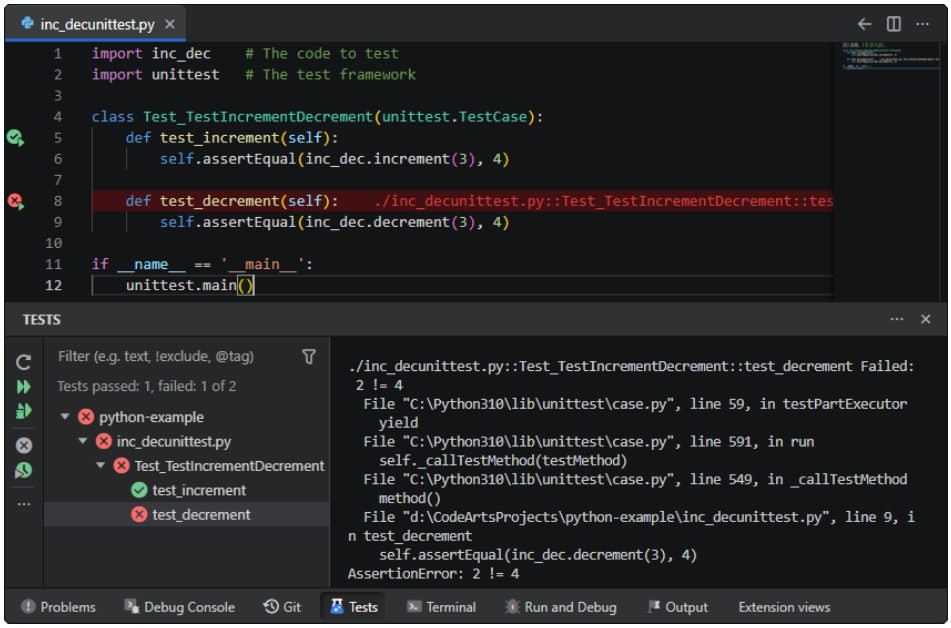
4. 下一个对话框中，选择用于标识测试文件的文件通配符模式，本例中为 “*test.py”。



步骤4 在配置完框架集成后，CodeArts IDE会自动检测测试并在“**测试**”视图中显示它们。



现在，您可以使用各种命令（在**命令面板**（“Ctrl Ctrl” / “Ctrl+Shift+P”）中，在编辑器行号区，或在“**测试**”视图中）来运行和调试测试，包括运行单个测试文件和单个方法。



----结束

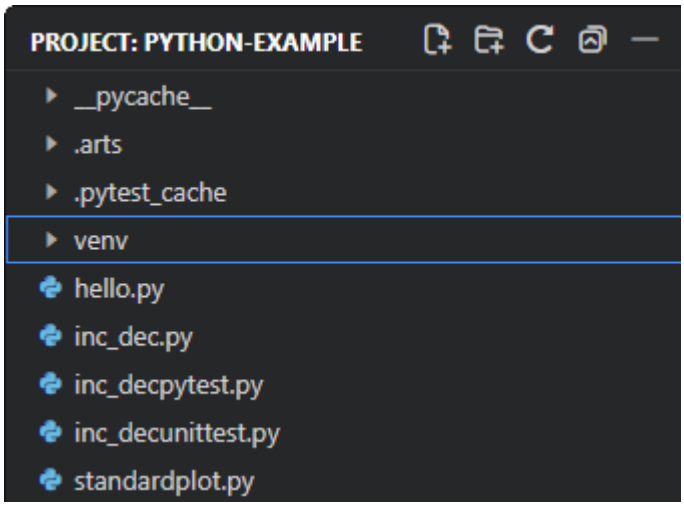
说明

有关测试Python代码的更多详细信息，请参阅**Python 测试**文档。

6.2 开始工程

在CodeArts IDE中打开带有“py”源代码的任意文件夹即可将其作为Python项目开始工程。此外，CodeArts IDE还提供了Python项目向导，帮助简化创建新项目和配置环境的过程。为了在CodeArts IDE中获得完整的编码支持和其他Python相关功能，您需要为项目指定一个解释器。

您的Python项目的内容会显示在资源管理器视图中（“Ctrl+Shift+E”/“Alt+1”（IDEA快捷键））中，该视图提供了常见的文件管理功能。



6.2.1 管理 Python 项目

6.2.1.1 打开现有项目

步骤1 在主菜单中，选择**文件** > **打开工程** 。

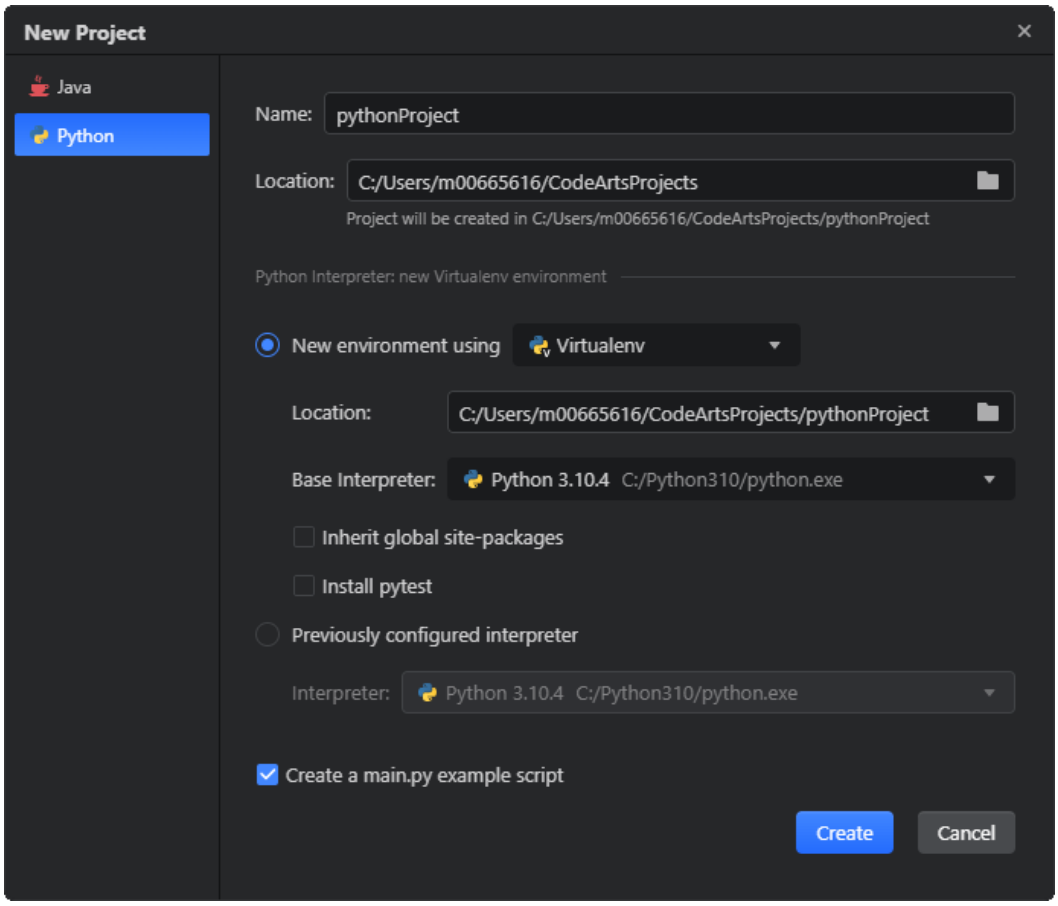
步骤2 在打开的“**打开工程**”对话框中，定位到所需的文件夹，然后单击“**选择文件夹**”。

----结束

6.2.1.2 新建项目

步骤1 在主菜单中，选择**文件** > **新建** > **工程** 。

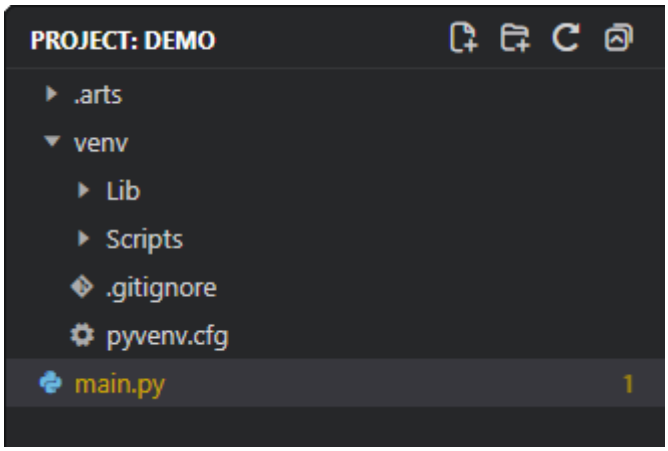
步骤2 在打开的“**新建工程**”对话框中，从左侧列表选择“**Python**“，并填入项目参数。



1. 设置项目名称和路径
2. 选择项目Python解释器
   a. 要配置新的解释器，请选择“**新环境使用**”选项，并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。勾选“**继承全局站点包**”复选框，以便CodeArts将全局Python中可用的所有包安装到创建的虚拟环境中。勾选“**安装pytest**”复选框，以便CodeArts IDE安装pytest包。
   b. 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 单击“**创建**”，CodeArts IDE将创建并打开项目，并为该项目设置解释器。

----结束



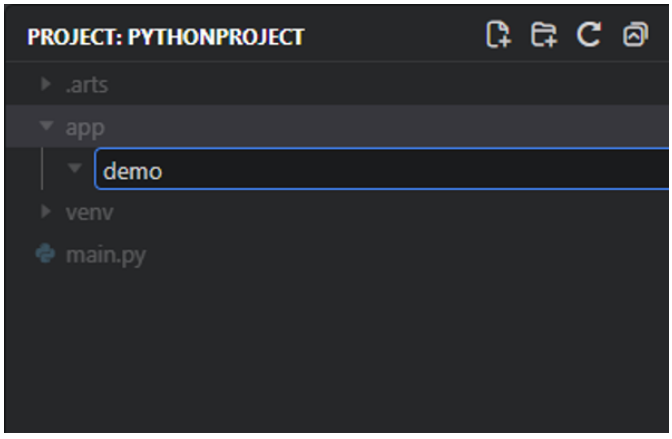
6.2.1.3 新建文件和文件夹

6.2.1.3.1 新建文件夹

步骤1 在资源管理器中，右键单击您想要在其中创建新文件夹的文件夹，在上下文菜单中选择“**新建文件夹**“。

步骤2 输入新建文件夹的名称，按下“enter”键。

----结束

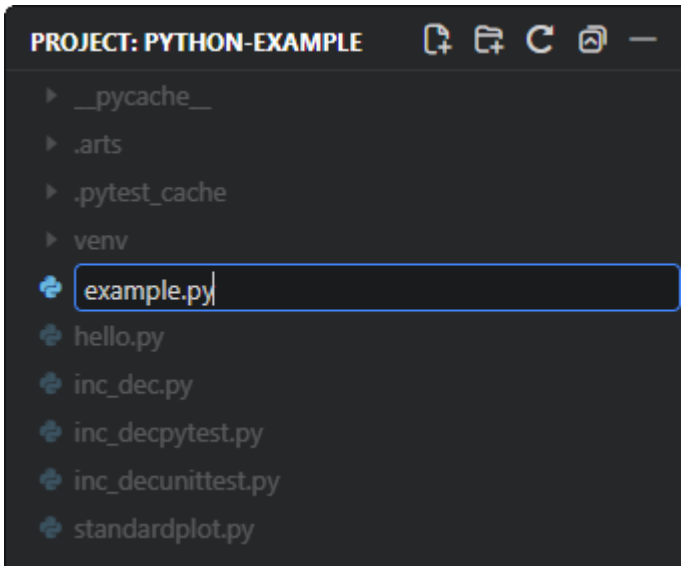


6.2.1.3.2 新建文件

步骤1 在资源管理器中，右键单击您想要在其中创建新文件的文件夹，在上下文菜单中选择**新建** > **文件** 或按下“Ctrl+Alt+N”（IDEA快捷键）。

步骤2 在输入框输入文件名，按下“enter”键。

----结束



6.3 构建环境

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**下运行，不会对特定项目有额外操作，因此任何安装或卸载的包都会影响全局环境及在其中运行的所有程序。随着时间的推移，全局环境可能会变得拥挤，难以测试应用程序。

为了避免这种混乱和不便，您可以为项目创建一个**虚拟环境**。虚拟环境是一个包含特定解释器副本的文件夹。安装到虚拟环境中的包仅安装在该文件夹中，而不会污染全局Python解释器。当您在该环境中运行程序时，它仅使用虚拟环境中这些特定的包来运行。

CodeArts IDE会自动在以下位置查找解释器：

- 标准安装位置，如“/usr/local/bin”、“/usr/sbin”、“/sbin”、“c:\python27”、“c:\python36”等。
- 工作区（项目）文件夹下直接创建的虚拟环境。
- 由“python.venvPath”设置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”的第一级子文件夹中查找虚拟环境。
- 通过pyenv和Pipenv安装的解释器。

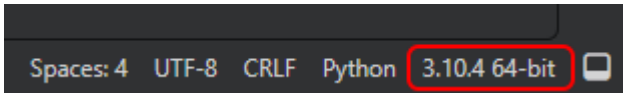
如果CodeArts IDE无法自动定位您的解释器，您可以手动指定它。此外，CodeArts IDE还会加载由“python.envFile”设置的环境变量定义文件，默认值为“${workspaceFolder}/.env”。

6.3.1 使用 Python 环境

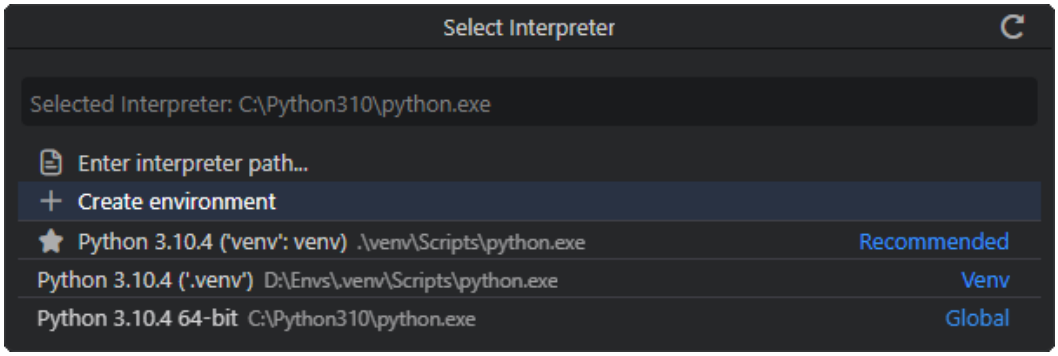
为了运行代码、获得编码帮助和其他Python相关功能，您需要为项目指定一个Python环境。

6.3.1.1 指定项目环境

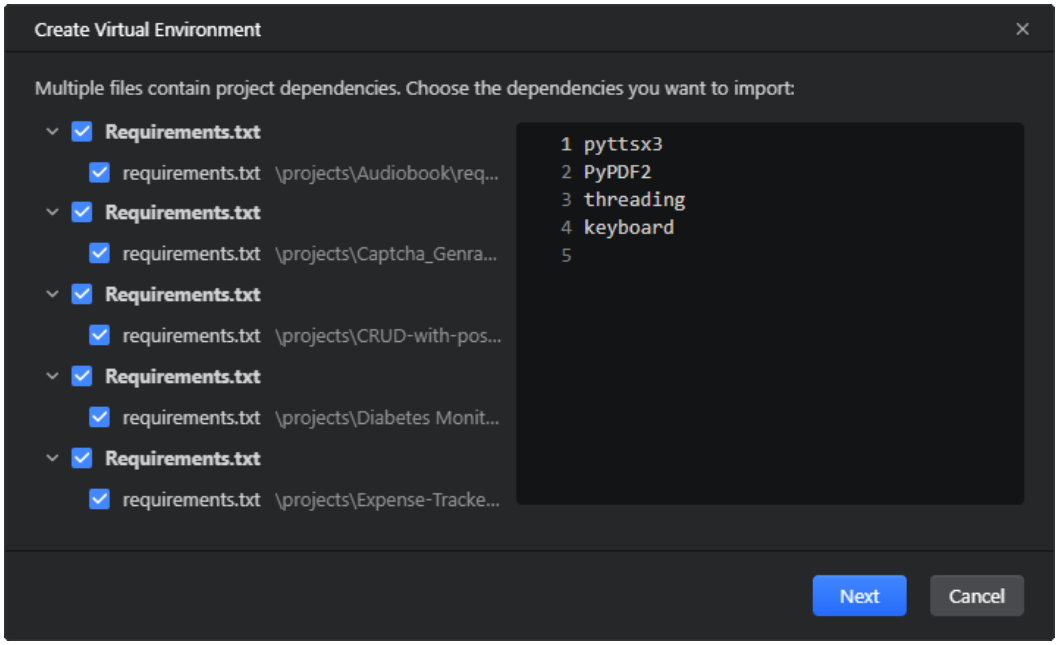
步骤1 在**命令面板**（“Ctrl+Ctrl”/“Ctrl+Shift+P”）中，搜索并运行“**Python: 选择解释器**”命令，或者单击状态栏右侧的按钮。



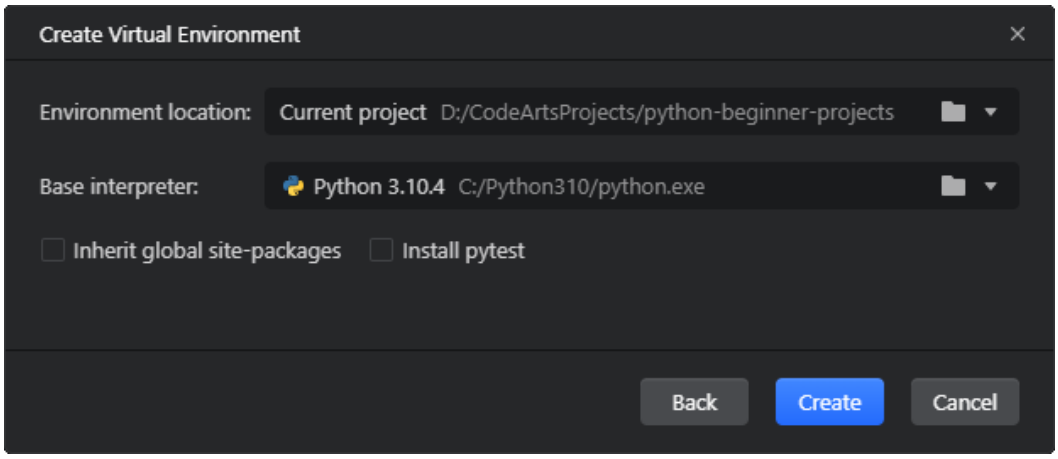
步骤2 在“**选择解释器**“对话框中选择“**创建环境**“。



步骤3 在创建环境的过程中，CodeArts IDE会自动安装列在**requirements.txt**文件中的项目依赖。如果有复数**requirements.txt**文件，那么在打开的对话框中，可以勾选想要安装的依赖旁的复选框，然后单击“**下一步**”。

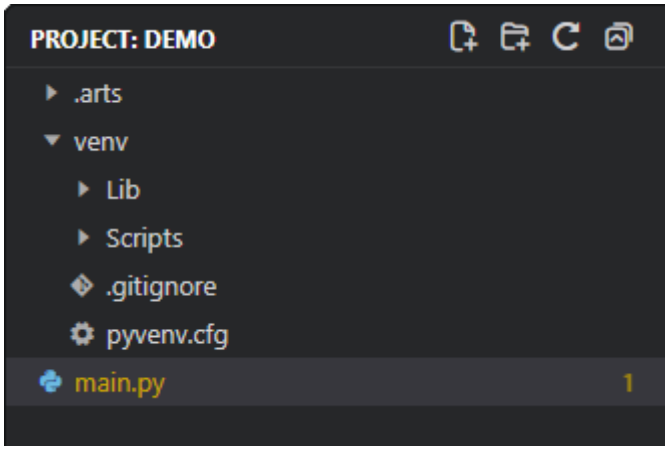


步骤4 在打开的“**创建虚拟环境**“对话框中，选择需要应用的环境创建选项。



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。有关Python代码测试的详细信息，参阅“Python测试”相关文档。

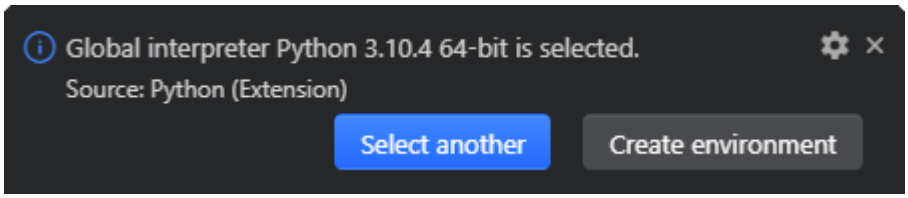
步骤5 单击“**创建**”。CodeArts IDE将在指定的文件夹中创建一个新的环境，并将其设置为项目的解释器。该环境将包含全局Python解释器的一个副本，并且特定于项目（安装到该环境中的所有包只应用于项目内部）。



----结束

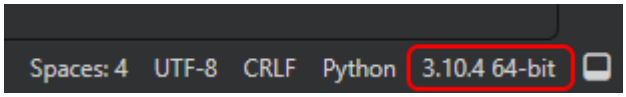
6.3.1.2 选择并激活环境

在使用项目向导创建项目时，通常会指定一个该项目的解释器。如果打开任意项目文件夹，CodeArts IDE会自动将系统路径中找到的第一个Python解释器设置为项目解释器，但您可以手动覆盖此设置。

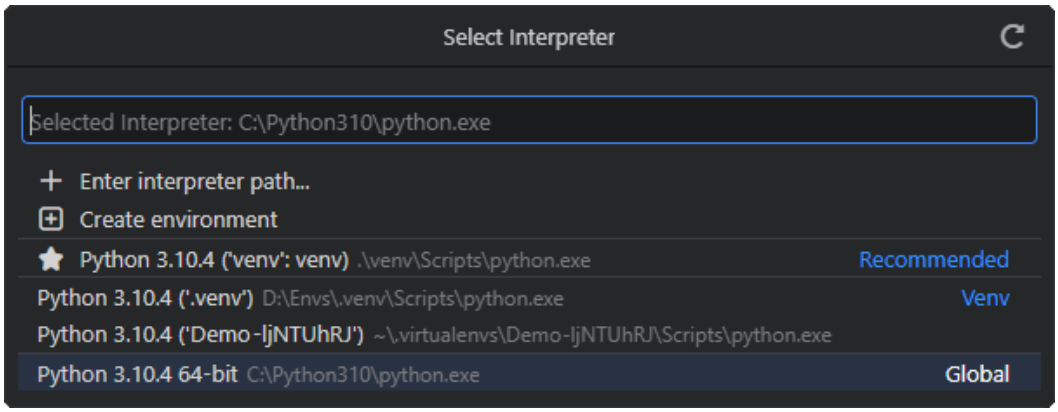


您可以在任何时候切换环境，以便根据需要使用不同的解释器或库版本来测试项目的不同部分。

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。



步骤2 从可用的全局和虚拟环境列表中选择所需的环境。



----结束

所选环境将用于运行Python代码并提供Python编码帮助（代码补全、验证、格式化等）。此外，当使用“**终端：创建新的终端**”命令打开终端时，CodeArts IDE会自动激活所选环境。若要防止自动激活所选环境，请禁用“python.terminal.activateEnvironment”设置项。

约束与限制

- 如果激活命令产生消息“Activate.ps1 is not digitally signed. You cannot run this script on the current system.”，那么您需要暂时更改 PowerShell 的执行策略以允许脚本运行（请参阅 PowerShell 文档中的“**关于执行策略**”部分）：“Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process”。
- 如果未选择解释器，状态栏也会有所反映。

- 默认情况下，CodeArts IDE 在调试代码时使用为您的工作区选定的解释器。您可以通过在调试配置的 python 属性中指定不同的路径来覆盖此行为。请参阅“**选择调试环境**”。

6.3.1.3 从命令行新建虚拟环境

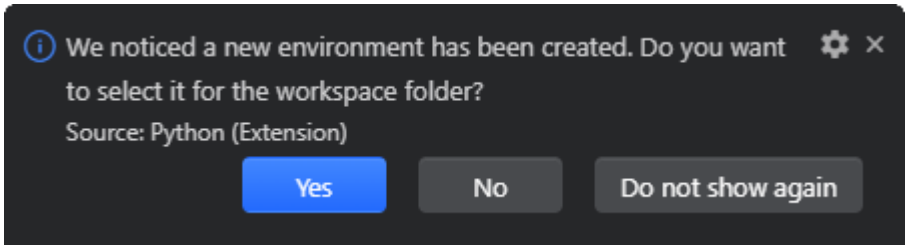
要使用“venv”手动创建虚拟环境，请使用以下命令。其中“.venv”是环境文件夹的名字。

- Windows
```
# You can also use py -3 -m venv .venv
python -m venv .venv
```
- Linux
```
# You may need to run sudo apt-get install python3-venv first
python3 -m venv .venv
```

说明

有关venv模块的更多详细信息，请参见Python.org上**创建虚拟环境**。

当您在项目文件夹内创建新的虚拟环境时，CodeArts IDE会提示您将其设置为项目解释器。



6.3.1.4 设置默认项目环境

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",
}
```

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。您还可以使用语法“${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}",
```

通过使用环境变量，确保在操作系统上设置环境变量，您可以轻松地实现在路径不同的操作系统之间移动项目。

约束与限制

变量替换仅在CodeArts IDE设置文件中支持，在“.env”**环境文件**中不会生效。

6.3.1.5 环境和终端窗口

除非将“python.terminal.activateEnvironment”设置设为“false”，否则当您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。

请注意，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，请在一个正在运行的CodeArts IDE实例的命令面板（“Ctrl+Ctrl”/“Ctrl+Shift+P”）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，你可以在拆分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”）（ ）。

6.3.1.6 选择调试环境

默认情况下，调试器将使用CodeArts IDE用户界面选择的Python解释器。但是，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**Python 启动配置**。

6.3.2 环境变量

6.3.2.1 环境变量定义文件

环境变量定义文件是一个以“environment_variable=value”为形式，键值对构成的纯文本文件，其中“#”用于注释。该文件不支持多行值，但值可以引用系统中或文件中先前已定义的其他任何环境变量。有关更多信息，请参阅**变量替换**。环境变量定义文件可用于调试和工具执行（包括linters、格式化器、代码补全和测试工具）等场景，但不应用于终端。

默认情况下，CodeArts IDE会在当前项目文件夹中查找并加载名为“.env”的文件，并应用这些定义。这是由用户设置中的默认条目“"python.envFile": "${workspaceFolder}/.env"”来决定的。你可以更改“python.envFile”设置来使用不同的定义文件。

例如，在开发Web应用程序时，你可以使用不同的定义文件来存储不同的URL和其他设置，而不是直接在代码中设置。这样您就可以轻松地在开发服务器和生产服务器之间切换，如：

dev.env file

```
# dev.env - development configuration

# API endpoint
MYPROJECT_APIENDPOINT=https://my.domain.com/api/dev/

# Variables for the database
MYPROJECT_DBURL=https://my.domain.com/db/dev
MYPROJECT_DBUSER=devadmin
MYPROJECT_DBPASSWORD=!dfka**213=
```

prod.env file

```
# prod.env - production configuration

# API endpoint
MYPROJECT_APIENDPOINT=https://my.domain.com/api/

# Variables for the database
MYPROJECT_DBURL=https://my.domain.com/db/
MYPROJECT_DBUSER=coreuser
MYPROJECT_DBPASSWORD=kKKfa98*11@
```

然后，您可以将 “python.envFile” 设置设置为 “${workspaceFolder}/prod.env”，然后将调试配置中的 “envFile” 属性设置为 “${workspaceFolder}/dev.env”。

约束与限制

当使用多种方法指定环境变量时，有以下的优先顺序。

- “launch.json”中直接定义的“env” 变量会覆盖“launch.json”中 “envFile” 设置中定义的变量，以及“python.envFile”设置指定的“.env”文件包含的环境变量。
- “launch.json”的“envFile”设置中定义的环境变量会覆盖“python.envFile” 设置指定的“.env”文件包含的环境变量。

6.3.2.2 PYTHONPATH 变量使用

PYTHONPATH环境变量指定了Python解释器应该查找模块的额外位置。在CodeArts IDE中，“PYTHONPATH”可以通过终端设置（“terminal.integrated.env.*”）、“.env”文件或者同时使用这两种方法来设置。

- 当使用终端设置时，“PYTHONPATH”会影响在终端中运行的任何工具，以及CodeArts IDE通过终端执行的如调试等的任何操作。然而，如果CodeArts IDE执行的操作不是通过终端进行的，比如使用linter或格式化器时，这个设置将不会影响模块的查找。
- 当使用“.env”文件设置“PYTHONPATH”时，它会影响CodeArts IDE代表您执行的所有操作以及调试器执行的操作，但不会影响在终端中运行的工具。

一个使用“PYTHONPATH”的例子是，如果您有一个包含源代码的“src”文件夹和一个包含测试的“tests”文件夹。在运行测试时，这些测试通常无法访问“src”中的模块，除非您硬编码相对路径。

为了解决这个问题，您可以在CodeArts IDE工作区内创建一个“.env”文件，并将“src”的路径添加到“PYTHONPATH”中。

```
PYTHONPATH=src
```

然后在 “settings.json” 文件中设置 “python.envFile” ，指向您刚刚创建的 “.env” 文件。如果 “.env” 文件位于项目根目录下，则 “settings.json” 应设置如下：

```
"python.envFile": "${workspaceFolder}/.env"
```

PYTHONPATH的值可以包含由 “os.pathsep” 分隔的多个位置：Windows上为分号（ “;” ），Linux 上为冒号（ “:” ）。无效路径将被忽略。如果您发现PYTHONPATH的值未按预期工作，请确保在操作系统的位置间使用正确的分隔符。在Windows上使用冒号分隔位置，或在Linux上使用分号分隔位置会导致 PYTHONPATH 值无效，该值将被忽略。

约束与限制

PYTHONPATH不指定Python解释器本身的路径。有关PYTHONPATH的其他信息，请阅读 **PYTHONPATH 文档**。

6.3.2.3 变量替换

在定义文件中定义环境变量时，您可以用以下通用语法来使用已存在的环境变量。

```
<VARIABLE>=...${env:EXISTING_VARIABLE}...
```

其中 “...” 表示值中使用的任何其他文本，大括号是必需的。

在这个语法中，适用以下规则：

- 变量按照它们在 “.env” 文件中出现的顺序进行处理，因此你可以使用文件中之前定义的任何变量。
- 单引号或双引号不会影响替换的值，它们会被包含在定义的值中。例如，如果 “VAR1” 的值是 “abcedfg” ，那么 “VAR2='${env:VAR1}'” 会将值 “'abcedfg'” 赋给 “VAR2” 。
- “$” 字符可以用反斜杠进行转义，如 “\$” 。
- 可以使用递归替换，例如 “PYTHONPATH=${env:PROJ_DIR};${env:PYTHONPATH}” （其中 “PROJ_DIR” 是任何其他环境变量）。
- 只能使用简单的替换；不支持如 “${_${env:VAR1}_EX}” 这样的嵌套替换。
- 具有不支持语法的条目不会被处理，而是保留原样。

6.4 代码编辑

6.4.1 简介

CodeArts IDE是一款功能强大的Python源代码编辑器，提供众多功能来帮助您高效地编写代码。

说明

有关CodeArts IDE Python编辑功能的更多详细信息，请参阅相应主题：

- **自动导入**
- **代码片段**
- **代码重构**

6.4.1.1 代码补全

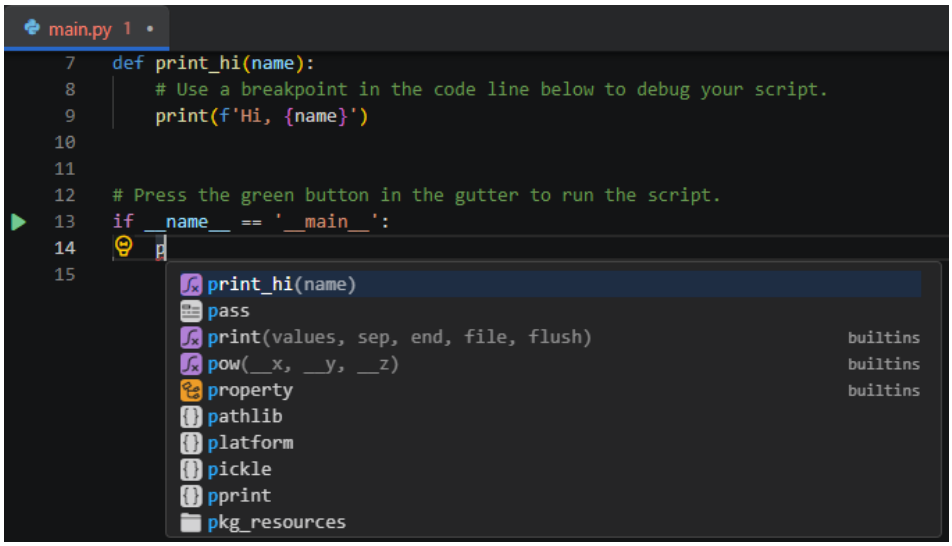
CodeArts IDE为当前项目和已安装包中的文件中的Python关键字和所有符号提供代码补全。

说明

有关CodeArts IDE代码补全功能的基本概念，请参阅**代码补全**。

6.4.1.1.1 触发代码补全

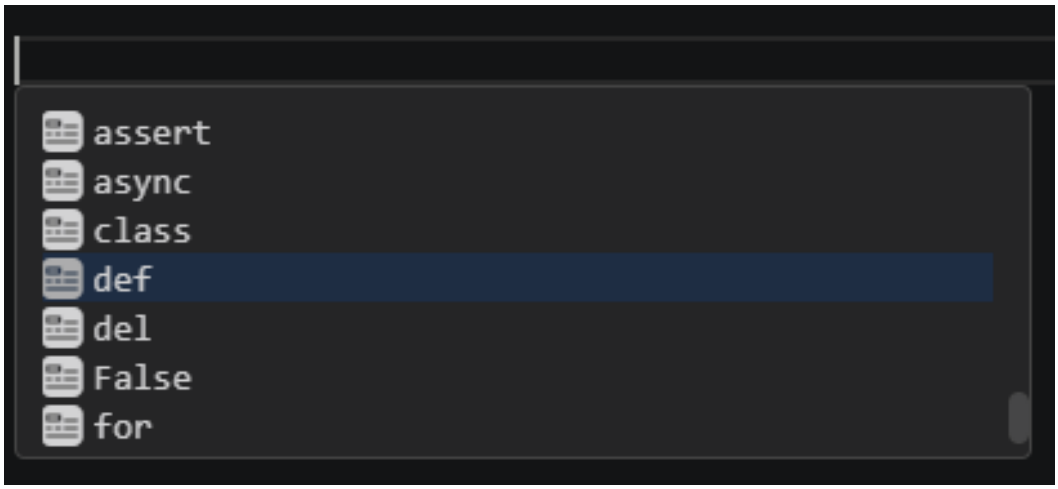
- 要手动触发代码补全，请按“Ctrl+I” / “Ctrl+Space” / “Ctrl+Shift+Space”或输入触发字符（如点字符 “.”）。



  - 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。

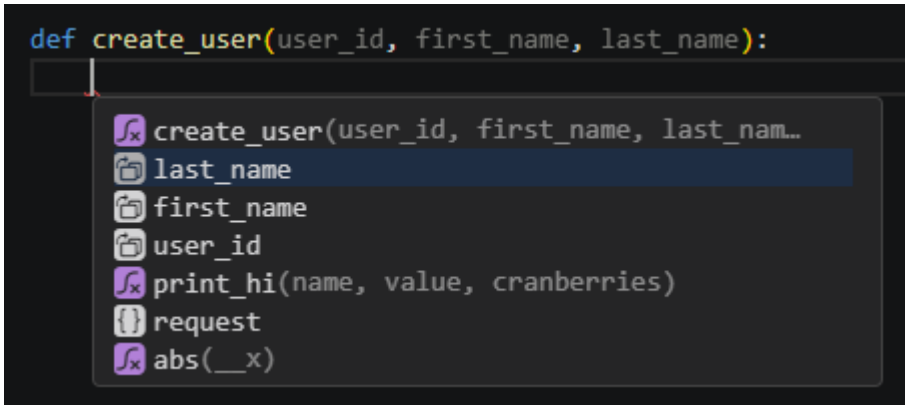
6.4.1.1.2 关键词补全

CodeArts IDE为Python保留关键字（如“assert”、“class”、“if”、“def”等）提供代码补全。



6.4.1.1.3 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。



6.4.1.2 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。

说明

有关代码折叠的更多详细信息，请参阅**代码折叠**。

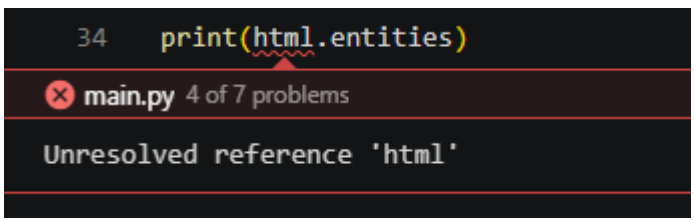
6.4.2 自动导入

如果您使用非导入符号，CodeArts IDE会帮助您添加相应的导入语句。此外，CodeArts IDE可以重新组织和验证代码中的导入。

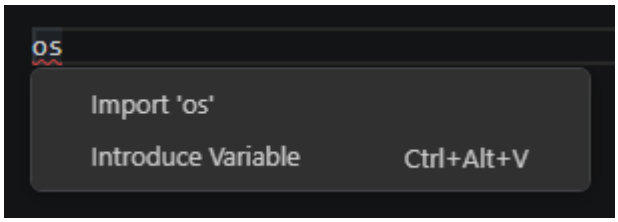
6.4.2.1 添加导入

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。

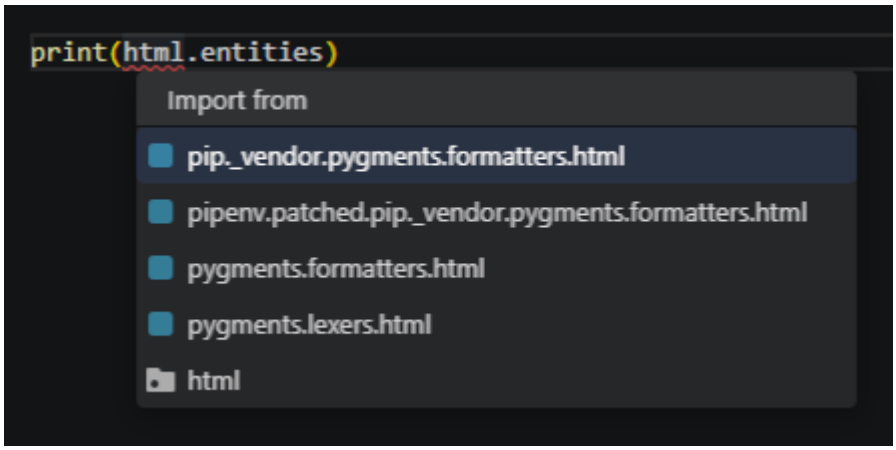
步骤1 在代码编辑器中，将光标置于强调显示的未解析符号处。



步骤2 按 “Ctrl+.” / “Ctrl+1” / “Alt+Enter”（IDEA快捷键），然后在弹出菜单中选择 “**导入<符号>**”。



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。



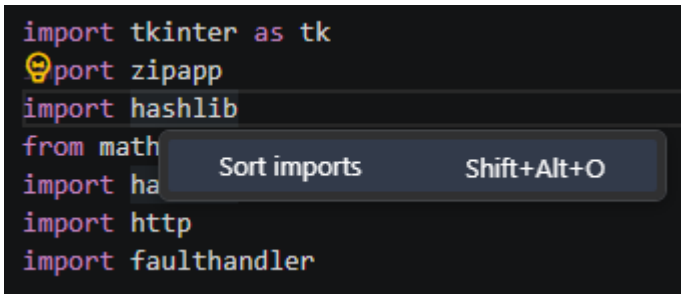
----结束

6.4.2.2 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“**Sort imports**”。



CodeArts IDE会移除不明确的导入，并按字母顺序对导入语句进行排序。

```
import faulthandler
import hashlib
import http
import tkinter as tk
import zipapp
from math import cos, pi, sin
```

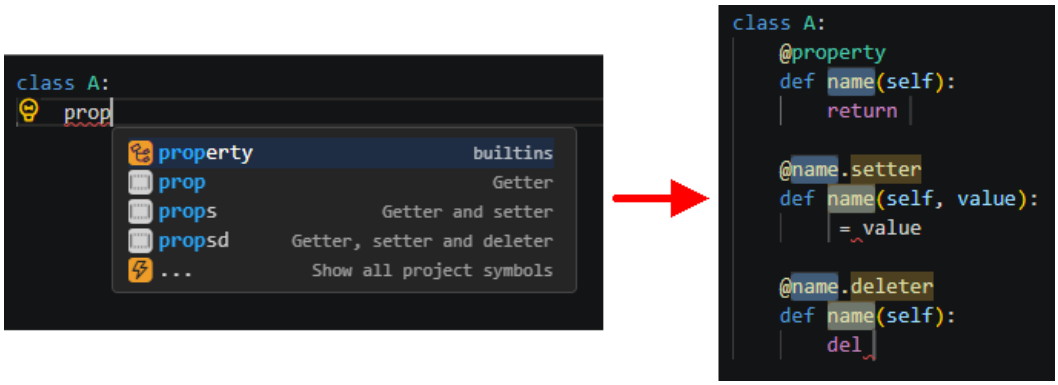
----结束

6.4.3 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”。

6.4.3.1 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、sette和deleter方法。



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。

6.4.3.1.1 条件语句

代码片段描述	缩写	扩展内容
创建一个“__name__”变量的检查守卫	“main”	if __name__ == '__main__': pass

6.4.3.1.2 循环语句

代码片段描述	缩写	扩展内容
用“for”循环迭代一个可迭代对象	“iter”	for i in <iterable>: pass

代码片段描述	缩写	扩展内容
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

6.4.3.1.3 列表解析

代码片段描述	缩写	扩展内容
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>}

6.4.3.1.4 类成员

代码片段描述	缩写	扩展内容
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

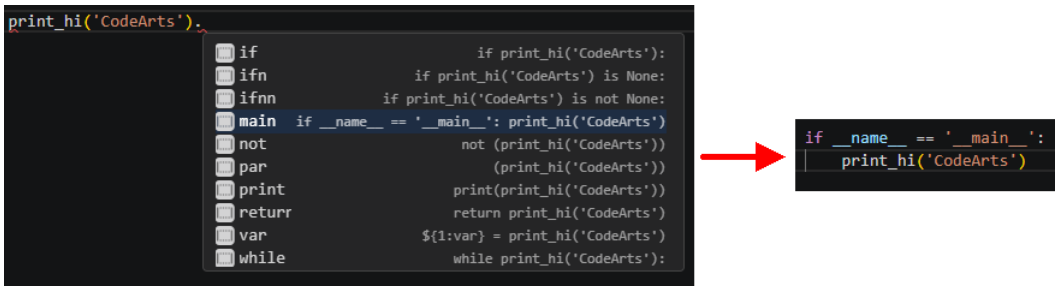
为类属性创建setter和getter方法	“props”	@property
		def name(self):
		  return
		
		@name.setter
		def name(self, value):
		  = value

代码片段描述	缩写	扩展内容

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

6.4.3.2 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。

6.4.3.2.1 一般语句

代码片段描述	缩写	扩展内容
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)

6.4.3.2.2 条件语句

代码片段描述	缩写	扩展内容
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__”检查守卫	“main”	if __name__ == '__main__': my_expression

6.4.3.2.3 循环语句

代码片段描述	缩写	扩展内容
为表达式创建“while”循环	“while”	while my_expression: <cursor>

6.4.3.2.4 程序输出

代码片段描述	缩写	扩展内容
将表达式发送到标准输出	“print”	print(my_expression)

6.4.4 代码重构

6.4.4.1 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**
  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**
  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**

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。

6.4.4.2 内联变量

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。

6.4.4.2.1 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”。

----结束

6.4.4.2.2 案例

作为示例，让我们内联变量“message”，将其替换为其初始值设定项 “Hello!”。

重构前

```
message = "Hello!"

print(message)
```

重构后

```
print("Hello!")
```

6.4.4.3 引入变量

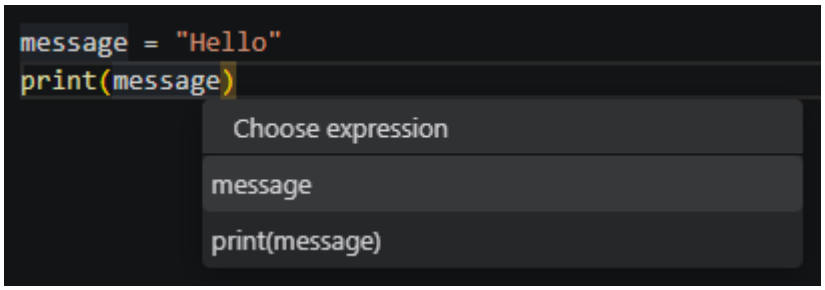
通过此重构，您可以创建一个新变量，使用所选表达式对其进行初始化，然后使用对所创建变量的引用替换原始表达式。这与内联变量重构相反。

6.4.4.3.1 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”。

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

6.4.4.3.2 案例

作为示例，让我们提取字符串“Hello!”到一个新的消息变量中。

重构前

```
print("Hello!")
```

重构后

```
message = "Hello!"

print(message)
```

6.4.4.4 变量重命名

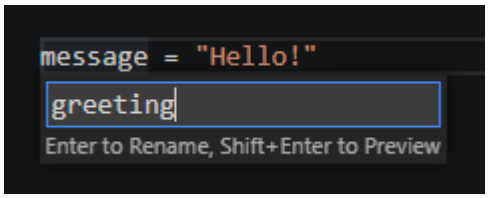
通过此重构，您可以在项目文件中重命名符号及用法。

6.4.4.4.1 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。



----结束

6.4.4.4.2 案例

作为示例，让我们将变量 “message” 重命名为“greeting”。

重构前

```
message = "Hello!"

print(message)
```

重构后

```
greeting = "Hello!"

print(greeting)
```

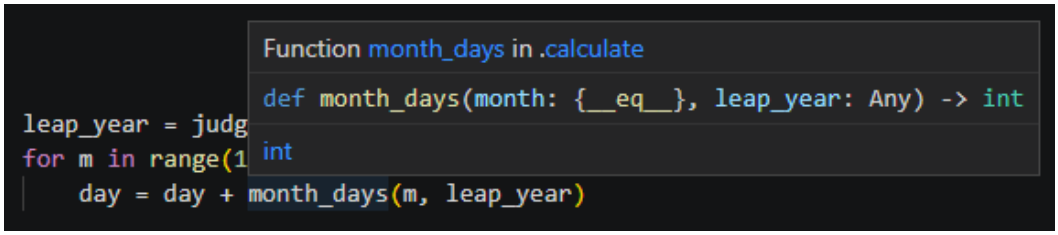
6.5 代码浏览

CodeArts IDE提供了多种浏览Python代码的功能。除了**代码导航**中描述的常规代码导航外，您还可以使用一些特定为Python语言服务的功能。

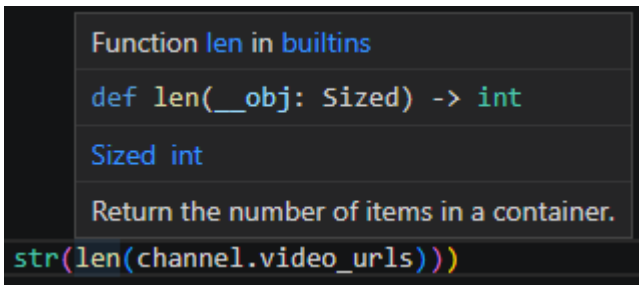
6.5.1 转到定义

您可以通过按下“F12”键或选择主菜单中的“**导航**”>“**转到定义**”来跳转到符号的定义。另外，您还可以使用 “Ctrl+单击” 符号定义在当前窗口跳转，或使用 “Ctrl+Alt+单击”在分割的编辑器中跳转。

如果将鼠标悬停在符号上，CodeArts会显示声明的预览：

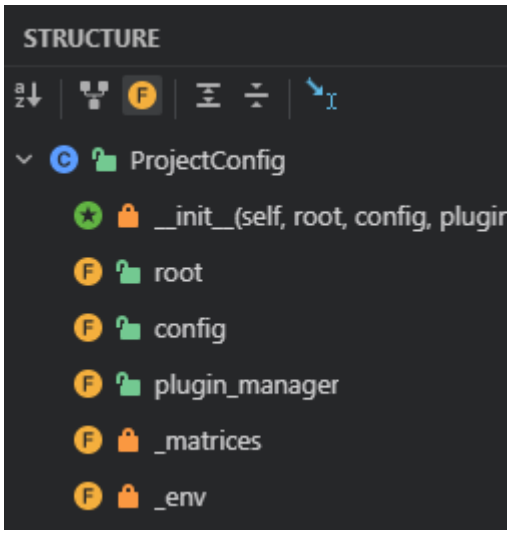


如果需要，您可以自定义定义悬停的内容。在CodeArts IDE设置中（“Ctrl+,”），转到“**扩展**”>“**Python**”，并找到“**Features: Hover**”设置。然后选择CodeArts IDE是否应显示定义的高亮代码片段和包含受影响符号的可单击链接的定义。如果选择两种都显示，可单击链接将显示在定义片段的下方。



6.5.2 结构

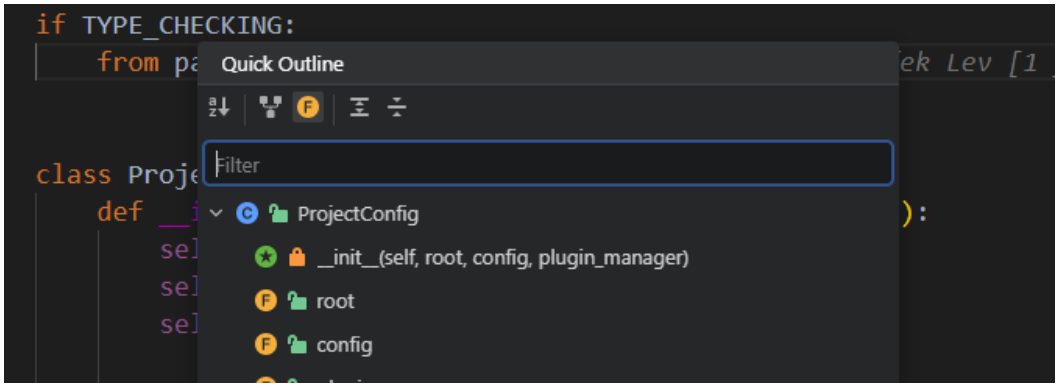
“**结构**”视图会显示当前活动的Python文件的符号树，并提供多种排序、分组和过滤功能。要打开“**结构**”视图，请在右侧的活动栏中单击“**结构**”，或按下 “Ctrl+Shift+F10”。



要快速导航到某个符号，请单击“**结构**”视图列表中的相应项。使用“**结构**”视图工具栏按钮可以对显示的符号进行排序、过滤和分组。

- : 按字母顺序对列表进行排序。
- : 显示继承的成员。
- : 显示过滤。
- /: 展开/折叠列表中的所有项。
- : 自动高亮代码编辑器中当前光标所在的元素。

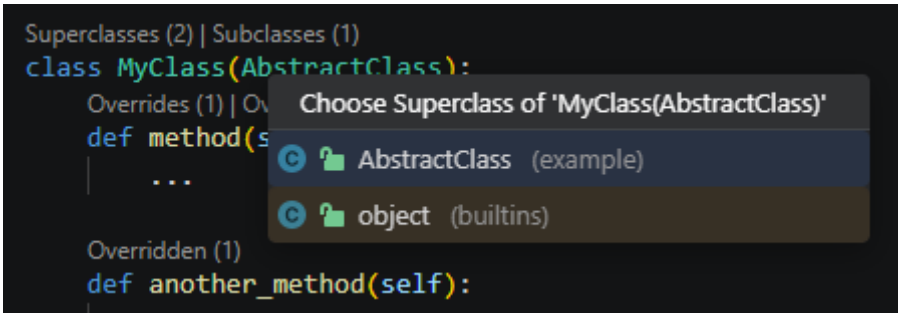
这一功能也可以通过“快速预览”视图实现，该视图在当前编辑器内显示，因此您无需切换上下文。要在“快速预览”视图中查看文件结构，请在主菜单中选择“**导航**”>“**快速大纲**”，或按下“Ctrl+F10”。在“**快速大纲**”窗口中，您可以开始键入所需符号的名称以自动过滤视图。



6.5.3 CodeLens

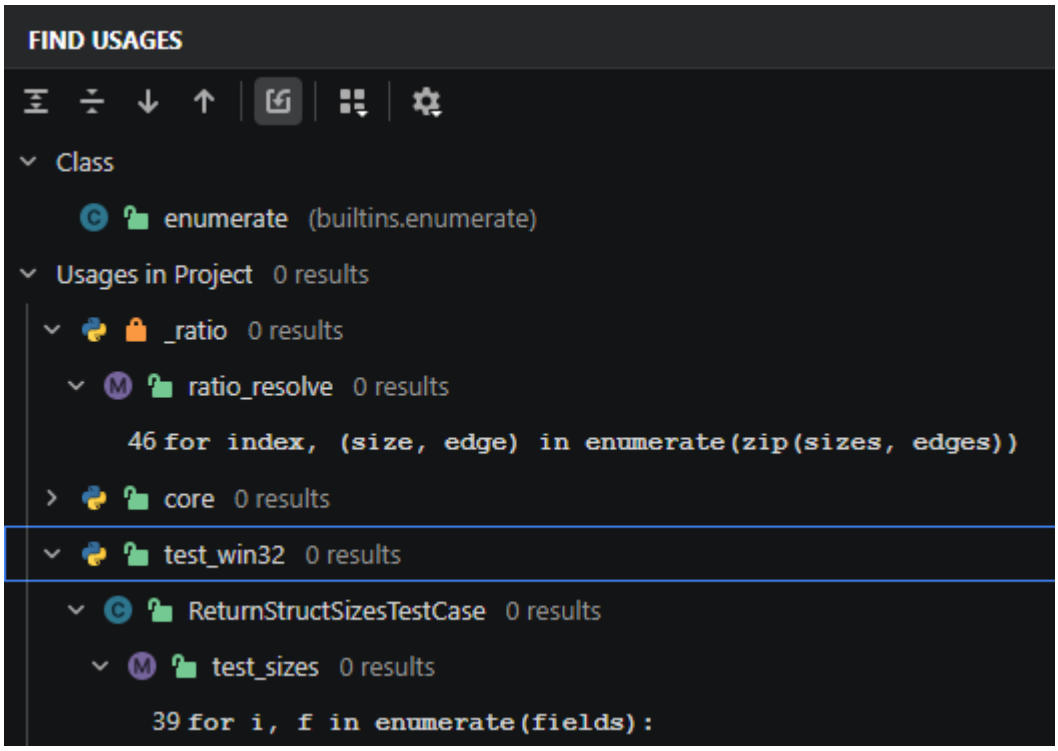
Python引用CodeLens会显示当前类的超类/子类的内联计数，以及当前方法的重写情况。

- 单击CodeLens，在打开的弹出窗口中选择要导航到的项目。



6.5.4 查找所有引用

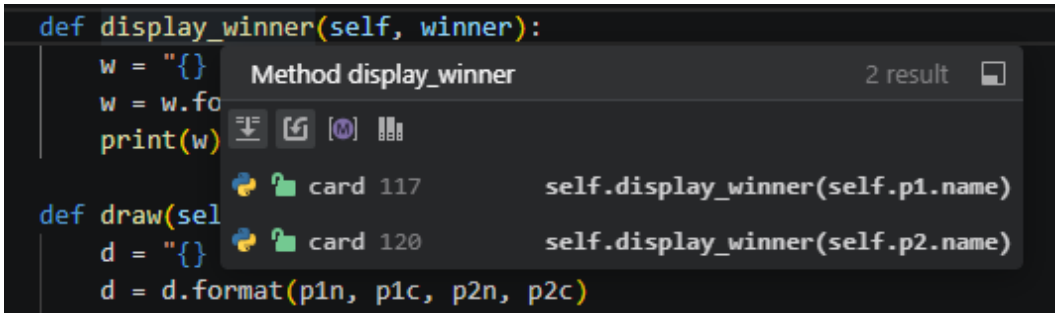
选择一个符号，然后按下“Alt+F7”（IDEA快捷键）/“Shift+Alt+F12”，在“**查找引用**”视图中打开该符号的所有引用。



“**查找引用**”视图的工具栏按钮有：

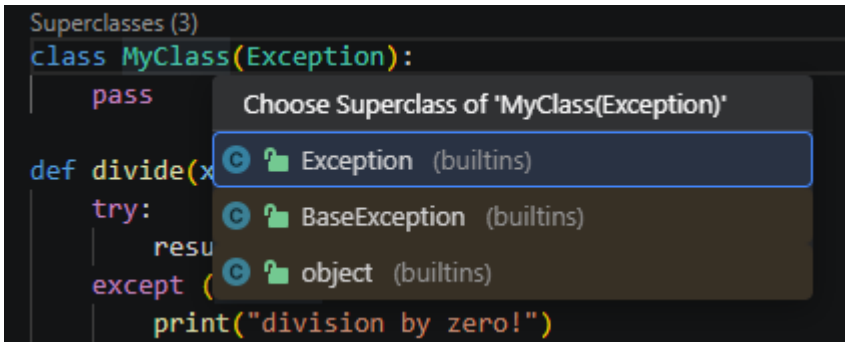
- /：展开/折叠列表中的所有项。
- /：在代码编辑器中打开符号的下一个/上一个出现位置。
- ：在导入语句中显示引用。
- ：按包含模块、类或方法进行引用分组。
- ：显示来自项目文件、依赖项或两者的引用。

这一功能也可以通过“快速预览”视图实现，该视图在当前编辑器内显示，因此您无需切换上下文。要在“快速预览”视图中查看符号的定义，请右键单击符号，在上下文菜单中选择“**快速查看**”>“**Peek Definition or Usages**”，或按下“Ctrl+B”（IDEA快捷键）/“Ctrl+Enter”（IDEA快捷键）。要在完整的“**查找引用**”视图中切换，请在“快速预览”视图中单击“**在查找引用窗口中打开**”按钮（）。



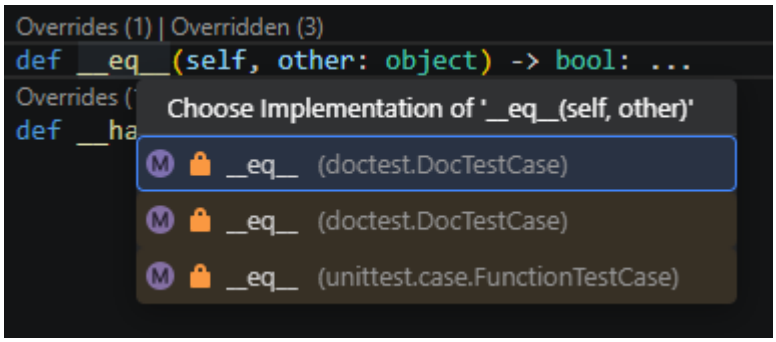
6.5.5 查找所有 Supers

您可以在代码编辑器中右键单击所选符号，然后选择上下文菜单中的“**快速查看**”>“**查看继承**”（或按下“Ctrl+U”（IDEA快捷键）），来查看所选符号的父类或父方法（Supers）。



6.5.6 查找所有实现

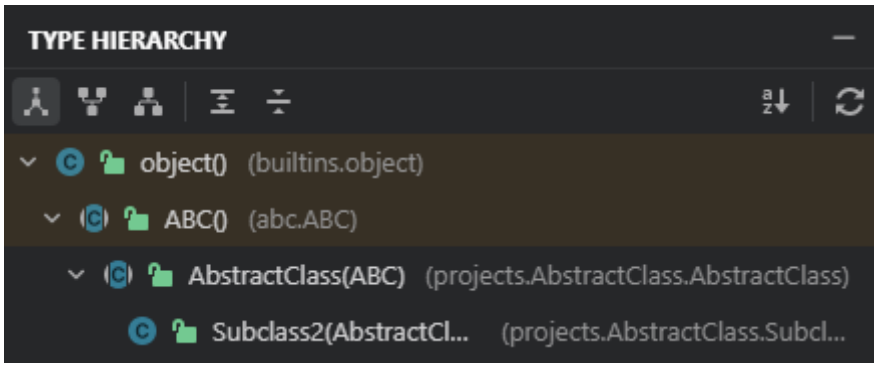
您可以在代码编辑器中右键单击所选符号，然后选择上下文菜单中的“**快速查看**”>“**查看实现**”（或按下 “Ctrl+Alt+B”（IDEA快捷键）），来查看该符号的实现。



6.5.7 类型层次结构

“**类型层次结构**”视图显示了继承关系，允许您查看选定类的父类和子类。要打开该视图，请在右侧的“活动栏”中单击“类型层次结构”。

右键单击一个类型，选择“**显示类型层次结构**”，或按下 “Ctrl+H”（IDEA快捷键）。



使用“类型层次结构”视图工具栏按钮，可以切换查看子类、父类或一起查看。

- : 查看父类和子类。
- : 仅查看父类。
- : 仅查看子类。
- /: 展开/折叠列表中的所有项。

- : 按字母顺序对列表进行排序。
- : 刷新列表内容。

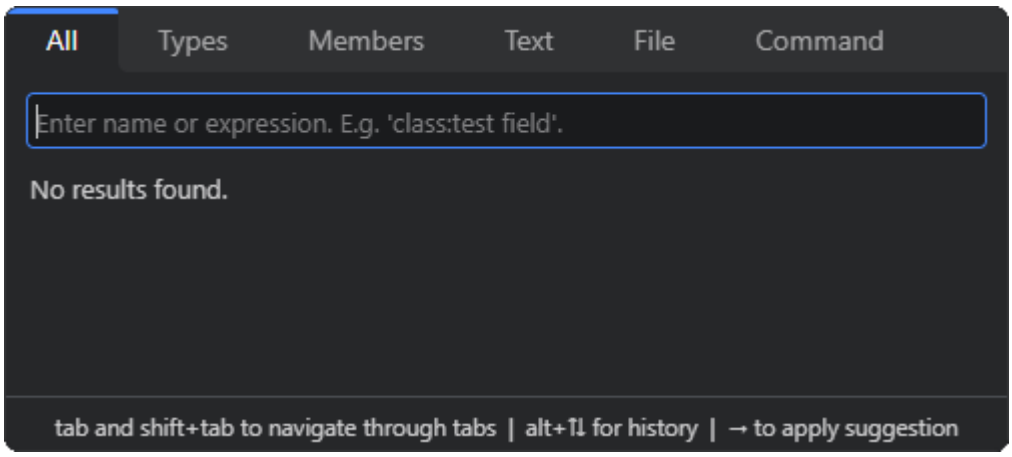
在“**类型层次结构**”视图中，您可以右键单击一个类，并从上下文菜单中选择“**基于此类型构建**”，以基于选定的类重新构建层次结构。

6.6 代码搜索

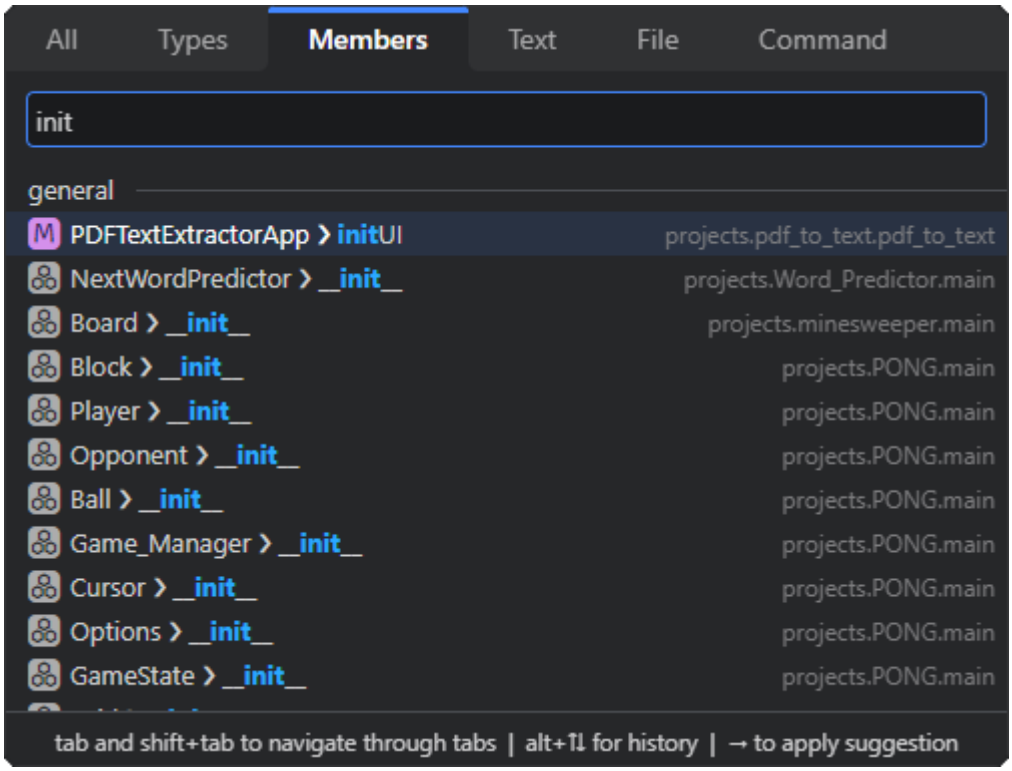
CodeArts IDE的SmartSearch功能可让您立即搜索并导航到任何项目位置，以及查找和执行任何CodeArts IDE命令。

6.6.1 基本用法

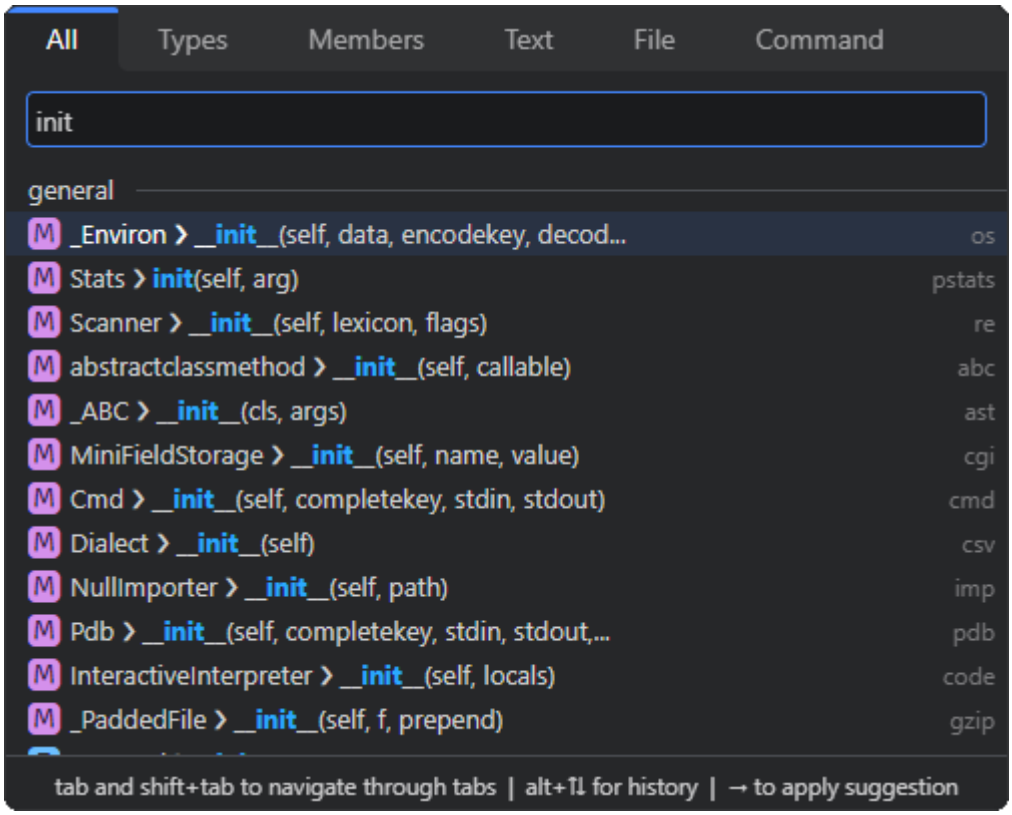
步骤1 通过按下 “Shift+Shift” / “Ctrl+Shift+A” 来启动**SmartSearch**。



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**SmartSearch**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**。



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**SmartSearch**窗口，请按下“Escape”键。



----结束

6.6.1.1 搜索查询语法

搜索查询是一个字符串，用于在**SmartSearch**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：

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

6.6.1.2 搜索运算符

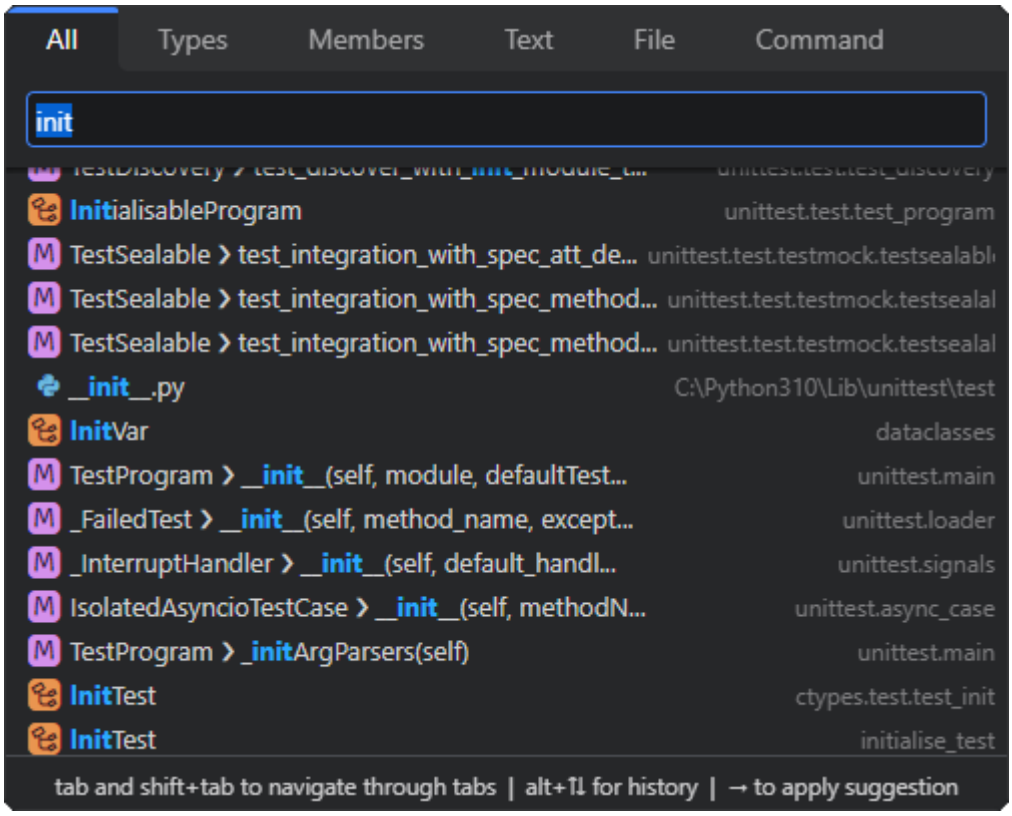
您可以使用AND和OR运算符或其组合来组成复杂的搜索查询，例如class:foo AND (method:bar OR method:baz)。

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。

6.6.2 案例

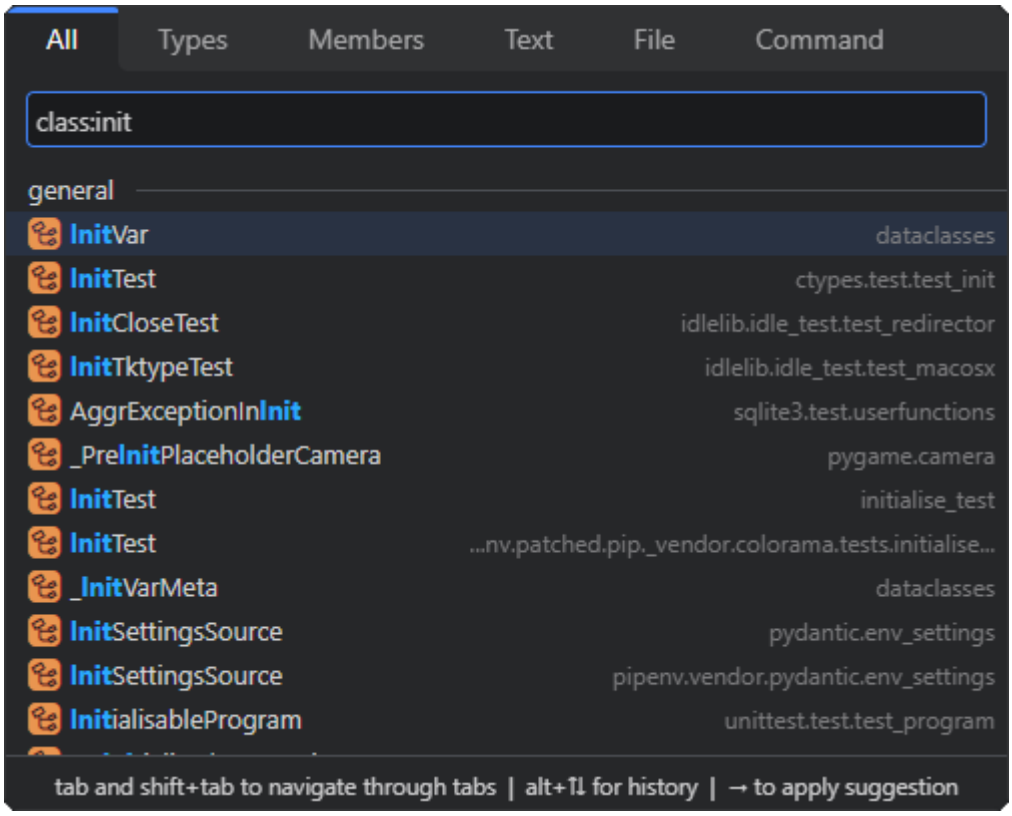
6.6.2.1 定位任意实体

搜索查询“init”将匹配名称中包含“init”的所有实体。



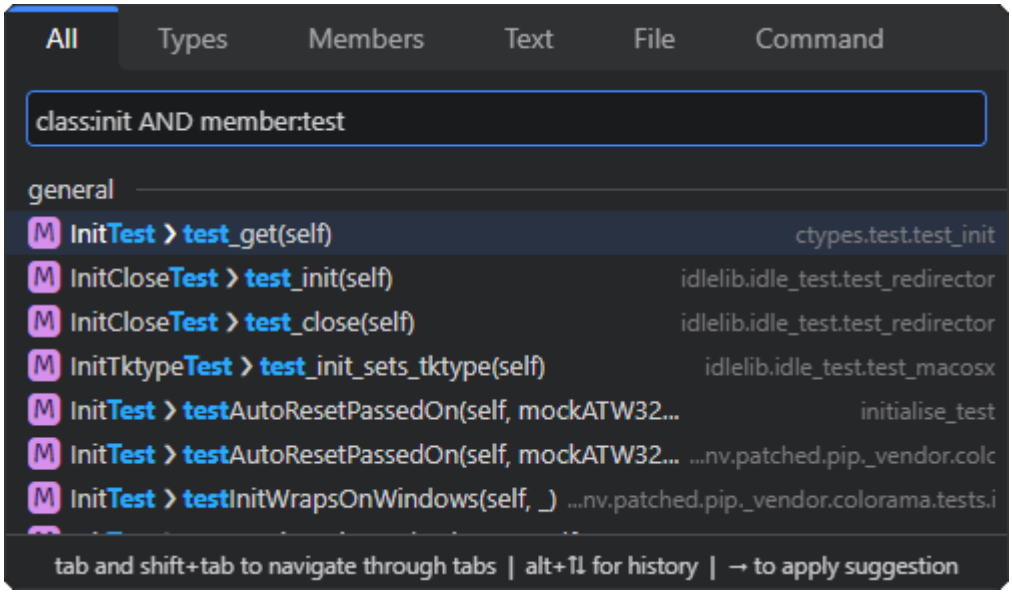
6.6.2.2 定位类

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”。

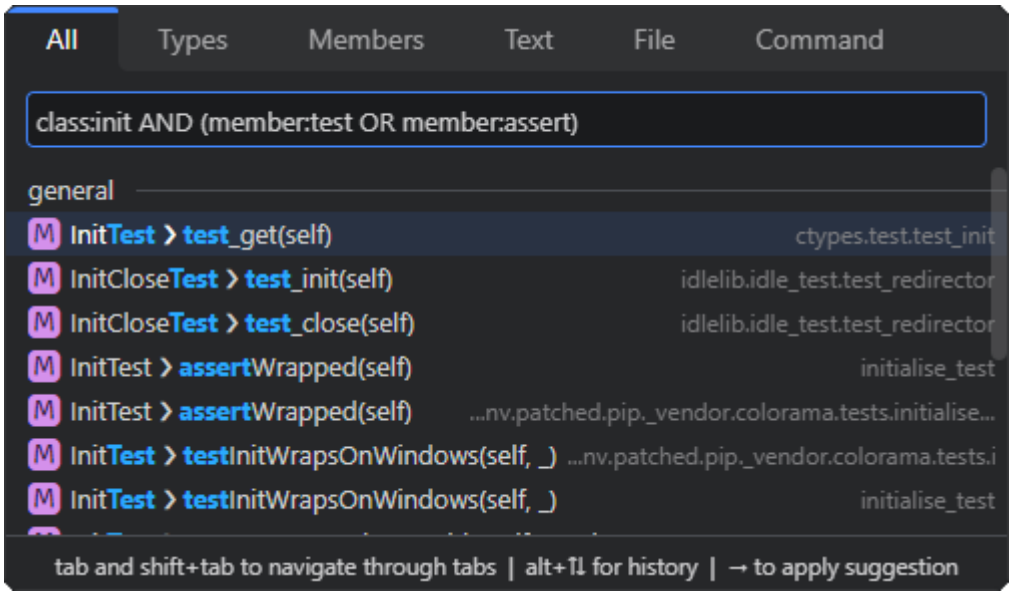


6.6.2.3 查询某个类的成员

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。

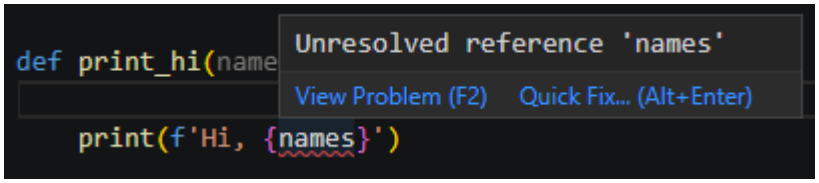


搜索查询“class:init AND (member:test OR member:assert)”将匹配名称包含“init”的类中，类成员名称中包含“test”或“assert”的所有类成员。



6.7 代码校验

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。



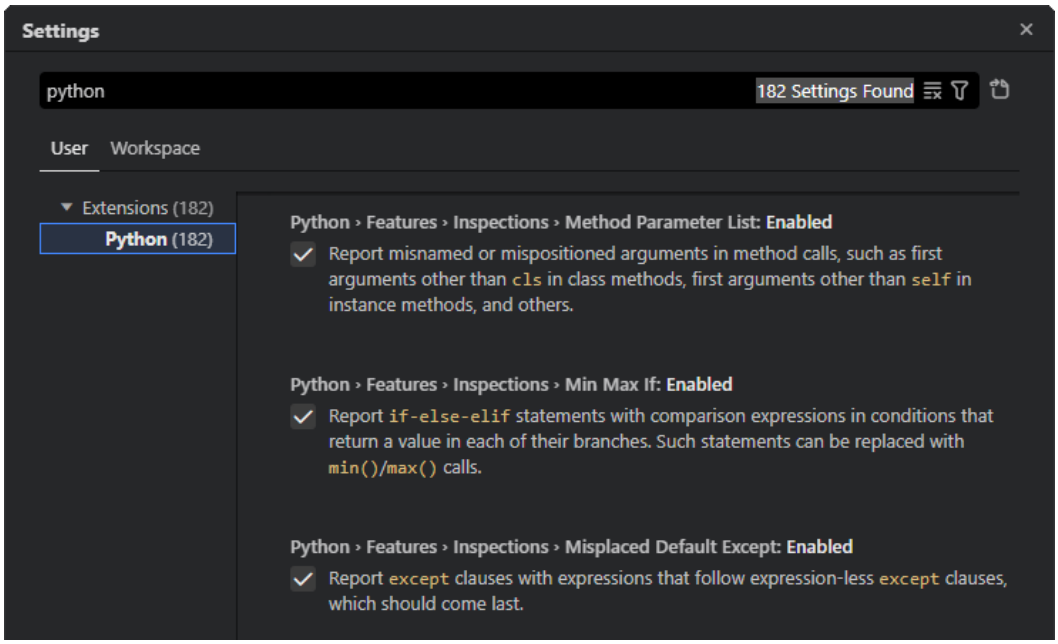
说明

有关查看警告和错误以及应用快速修复功能的详细信息，请参阅**代码校验**。

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。



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

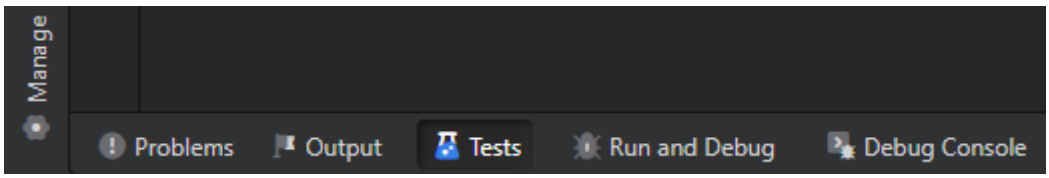
6.8 测试

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。

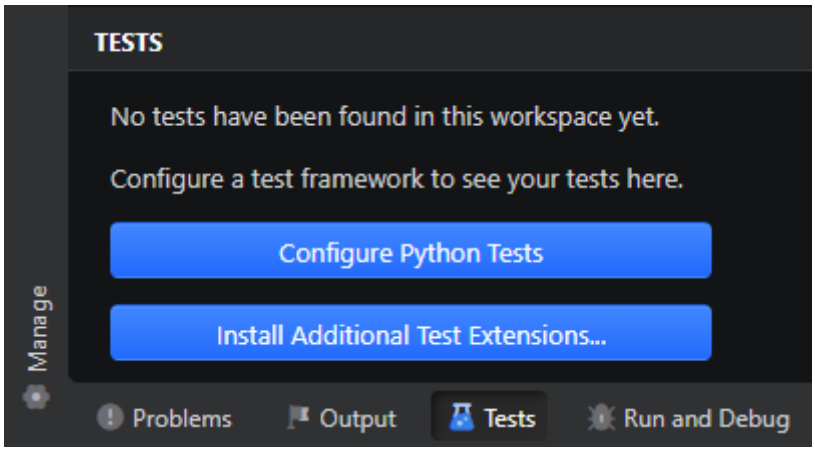
6.8.1 将测试框架集成到项目中

在您的项目中启动测试框架集成：

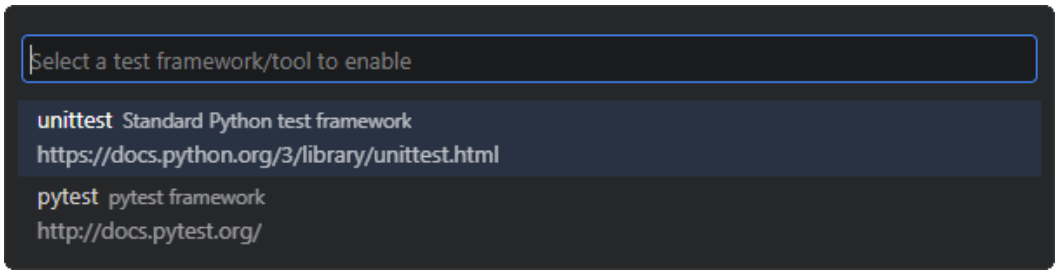
步骤1 单击CodeArts IDE底部的**“测试“**（ ）按钮来打开**测试**视图。



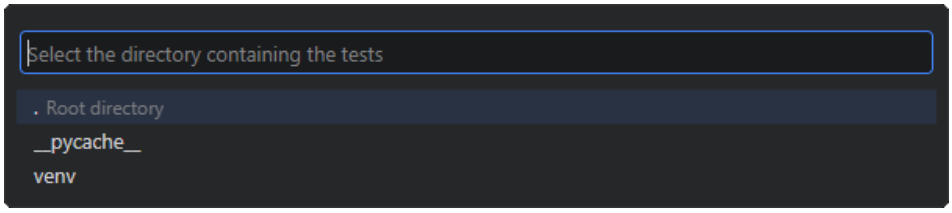
步骤2 在测试视图中，单击**“Configure Python Tests“**按钮。



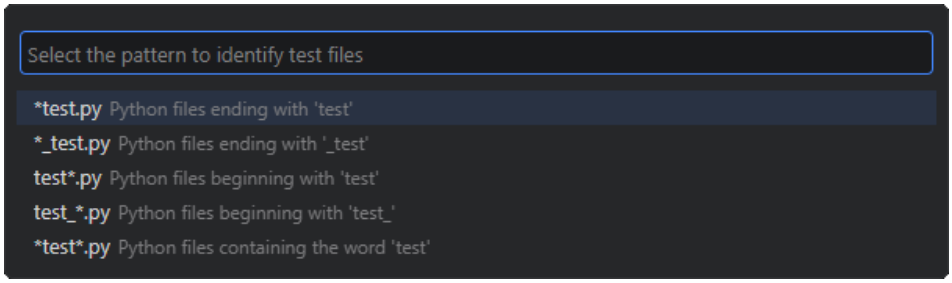
步骤3 在弹出的窗口中选择测试框架来启动对应集成。



- 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。

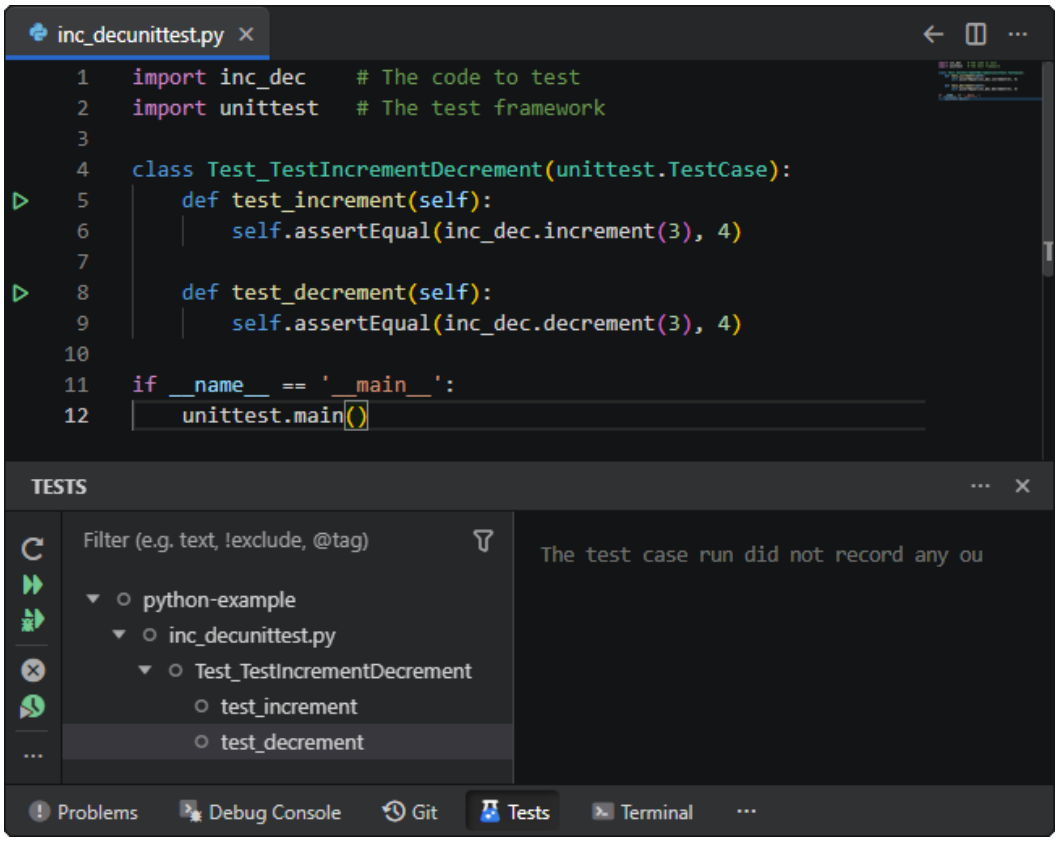
    

  - 在接着打开的对话框中，选择用于识别您的测试文件的文件通配符模式。

----结束

测试框架集成配置完成后，CodeArts IDE会在测试视图中展示发现的测试用例。



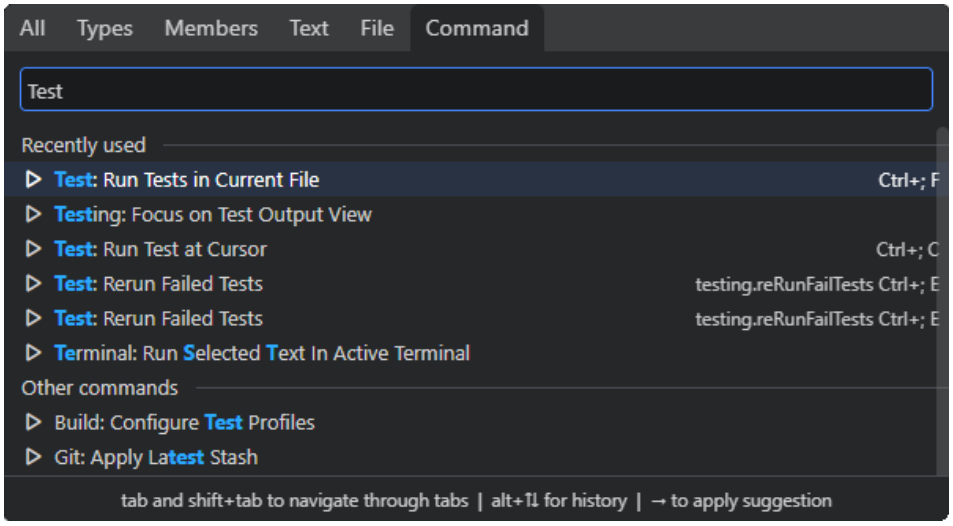
6.8.2 运行测试

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**”。

- 使用“**测试**”视图来管理和运行测试。
- 配置**pytest**和**unittest**专用的启动配置。

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**”。



6.8.3 启动配置

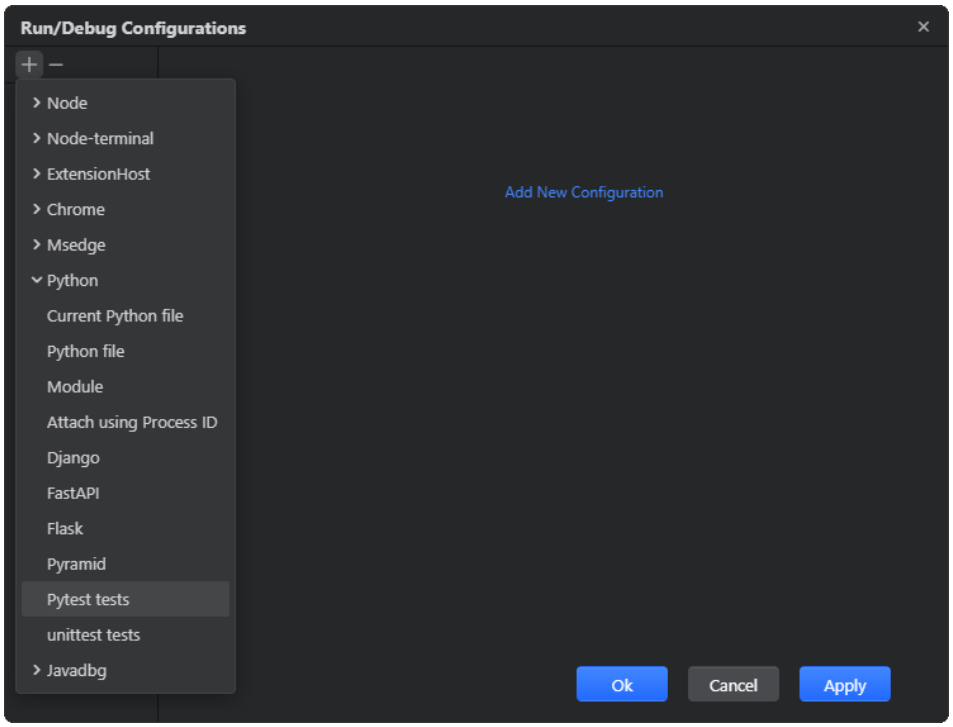
CodArts允许您自定义运行测试用例的配置。您可以在项目中添加对应的启动配置文件。

步骤1 在CodeArts IDE主工具栏中，选择列表中的**“编辑配置“**。

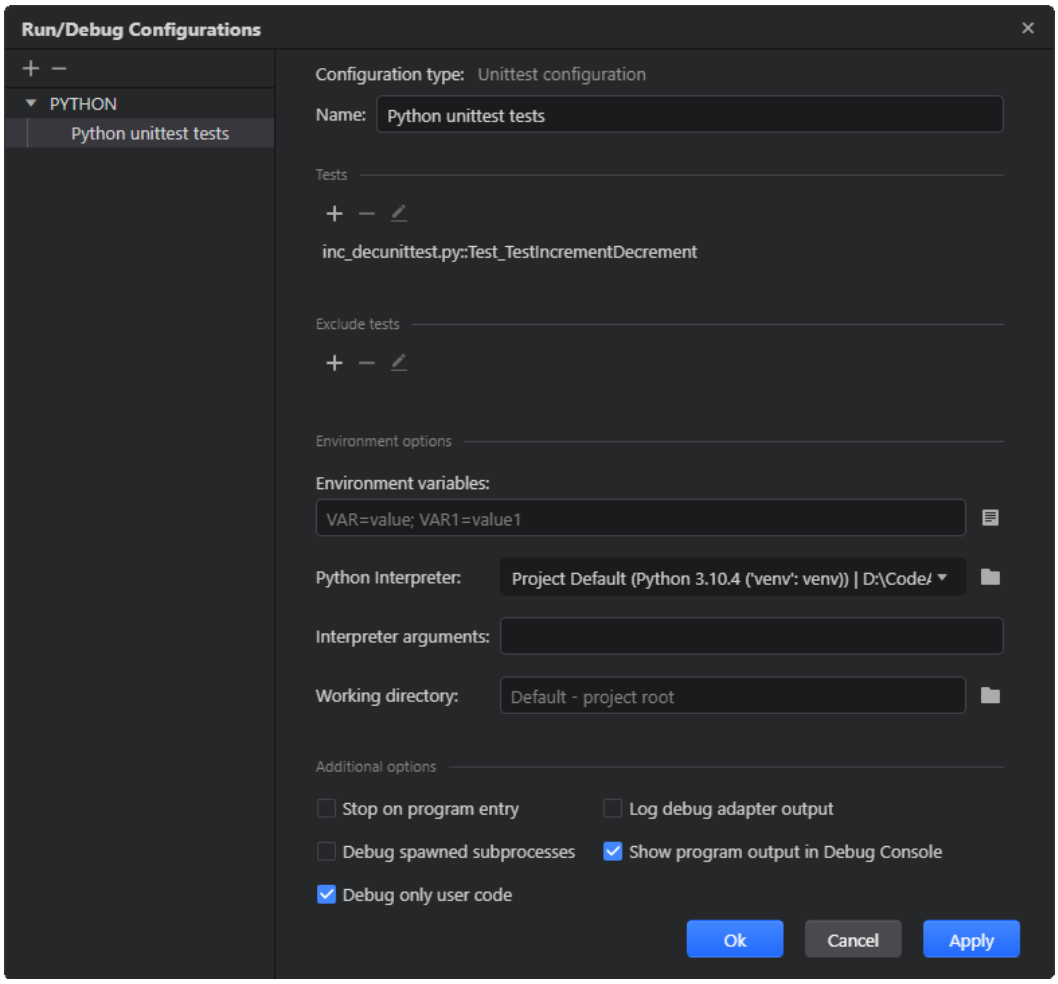
步骤2 在打开的“**运行/调试配置**”对话框中，单击工具栏上的“**新增配置项**”按钮（ ）或使用“**新增配置项**”链接。在出现的列表中，选择**Python**条目下所需的启动配置模板。

以下是用于运行和调试测试，开箱即用的配置模板：

- **pytest**
- **unittest**



步骤3 在配置参数区域里填入启动配置参数。



----结束

6.8.3.1 pytest 测试

6.8.3.1.1 启动配置属性

在启动配置中，您可以列出包含在启动配置范围内的测试ID，ID的格式如下：“test_file_name::test_class_name::test_method_name”。

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”。

名称	描述
“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“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“cwd”	调试程序工作目录的绝对路径。默认值“${workspaceFolder}”解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“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。

6.8.3.1.2 启动配置示例

以下是一个可运行的启动配置示例，该示例从“test_file_name::test_class_name::test_method_name” 运行测试。

```
{
  "excludeTestIds": [],
```

```
  "request": "test",
  "jinja": true,
  "python": "${command:python.interpreterPath}",
  "stopOnEntry": false,
  "redirectOutput": true,
  "env": {},
  "type": "python",
  "logToFile": false,
  "testIds": [
    "test_file_name::test_class_name::test_method_name"
  ],
  "cwd": "${workspaceFolder}",
  "subProcess": false,
  "justMyCode": true,
  "provider": "PYTEST",
  "pythonArgs": [],
  "name": "Python pytest tests",
  "showReturnValue": true
}
```

6.8.3.2 unittest 测试

6.8.3.2.1 启动配置属性

在启动配置中，您可以列出包含在启动配置范围内的测试ID，ID的格式如下：“test_file_name::test_class_name::test_method_name”。

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“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。

名称	描述
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“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。
“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。
“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。否则将显示和调试所有包括库调用的代码。
“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。
“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。

6.8.3.2.2 启动配置示例

以下是一个可运行的启动配置示例，该示例从“test_file_name::test_class_name::test_method_name” 运行测试。

```
{
  "excludeTestIds": [],
  "request": "test",
  "jinja": true,
  "python": "${command:python.interpreterPath}",
  "stopOnEntry": false,
  "redirectOutput": true,
  "env": {},
  "type": "python",
  "logToFile": false,
  "testIds": [
    "test_file_name::test_class_name::test_method_name"
  ],
  "cwd": "${workspaceFolder}",
  "subProcess": false,
  "justMyCode": true,
  "provider": "UNITTEST",
  "pythonArgs": [],
  "name": "Python unittest tests",
```

```
  "showReturnValue": true
}
```

6.9 调试

CodeArts IDE内置调试器有助于加快编辑、编译、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**逐步执行**、动态评估表达式等等。

6.9.1 调试步骤

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

6.9.2 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。

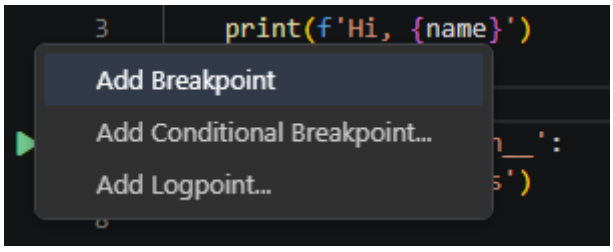
6.9.2.1 设置断点

6.9.2.1.1 行断点

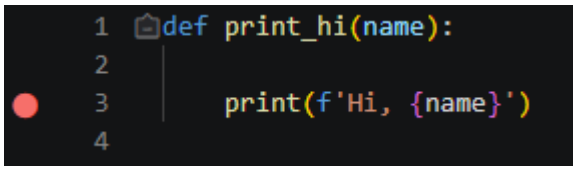
行断点是常规的断点，当程序执行到会在设置它们的行上时将暂停程序。

执行以下任一操作：

- 在编辑器行号边缘单击所需的行。
- 在编辑器所需行上的行号边缘右键单击，并从上下文菜单中选择“**添加断点**”。



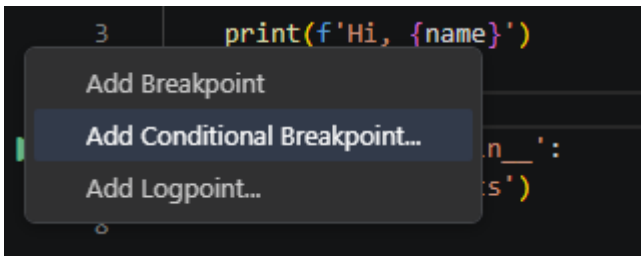
行断点在编辑器边缘以圆形图标 (🔴) 表示：



6.9.2.1.2 条件断点

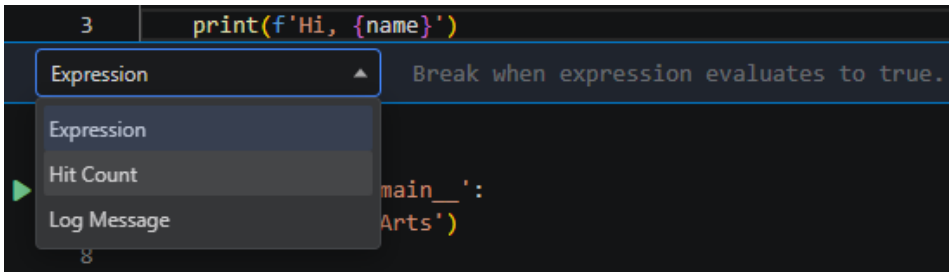
CodeArts IDE调试器允许您根据任意表达式或命中计数设置条件断点。

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**”。



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**：每次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。



步骤3 输入条件并按下**Enter**键。

----结束

您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。

6.9.2.1.3 日志点

日志点也是一种断点，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。

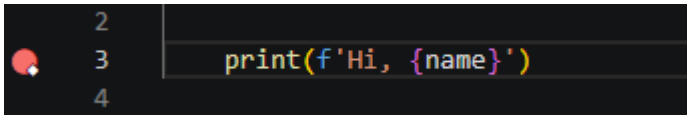
步骤1 在代码编辑器中，你可以通过右键单击想要设置日志点的行的编辑器边缘，并从上下文菜单中选择“**添加日志点**”。

另外，你也可以在主菜单中选择“**调试**”>“**新建断点**”>“**日志点**”。

步骤2 随后会打开一个预览编辑器，你可以在其中输入当日志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

在编辑器边缘，日志点由一个圆形图标（）表示。



约束与限制

与常规断点一样，日志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。

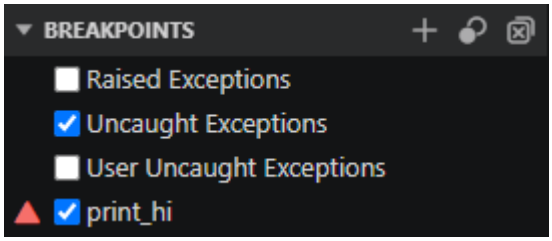
6.9.2.1.4 函数断点

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 要打开“**运行和调试**”视图，可以单击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）。

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（）。或者在主菜单中选择“**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。



----结束

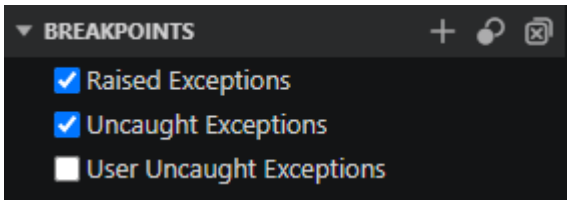
函数断点将在“**运行和调试**”视图的“**断点**”部分视图里以三角形图标（）表示。

6.9.2.1.5 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。



----结束

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。

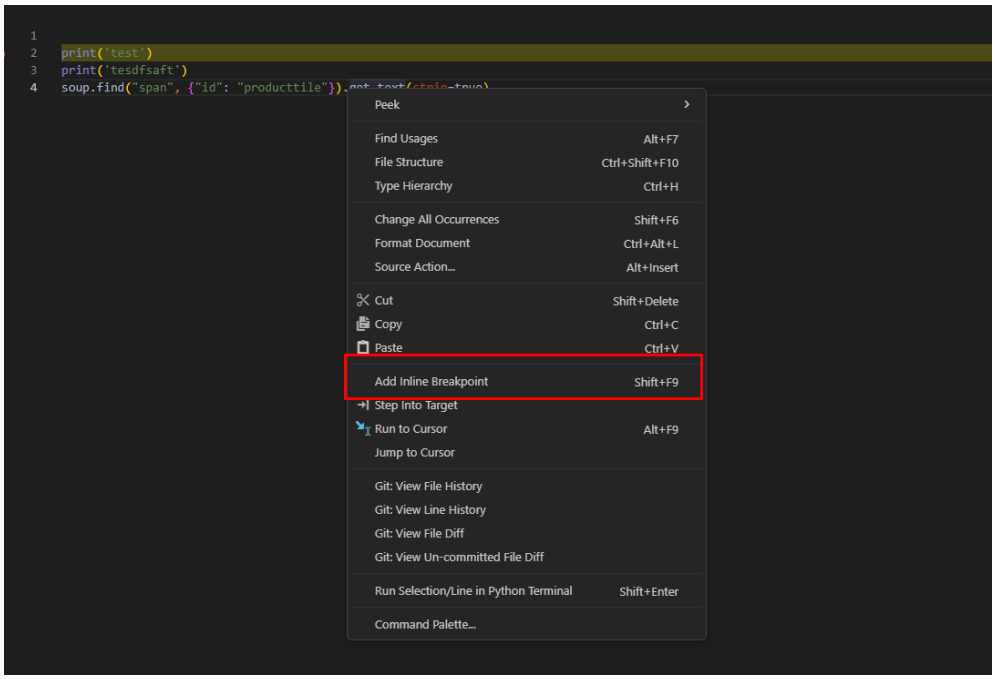
- **抛出异常**：任何抛出的异常，无论是否被捕获。
- **未捕获的异常**：任何被抛出且未被捕获的异常。

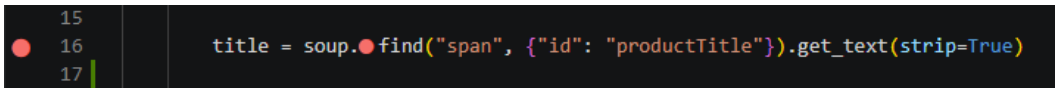
- **用户未捕获的异常**：源自用户代码（而非库）的任何未捕获异常。

6.9.2.1.6 行内断点

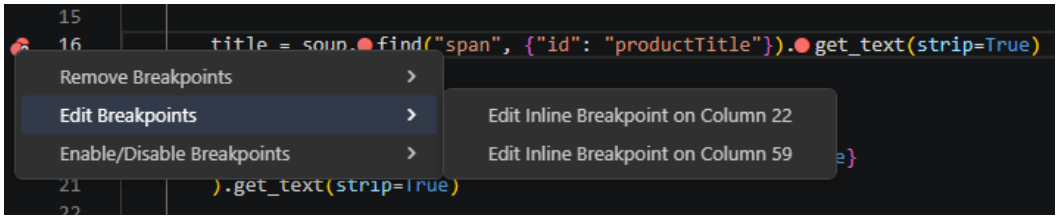
内联断点仅在执行到达与内联断点关联的列时触发。这在调试压缩代码时特别有用，因为压缩代码可能包含单行中的多个语句。

要设置内联断点，可以在主菜单中选择“**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。内联断点会直接在编辑器中内联显示。





内联断点也可以设置条件。若要编辑一行上的多个断点，请使用编辑器边缘的上下文菜单。

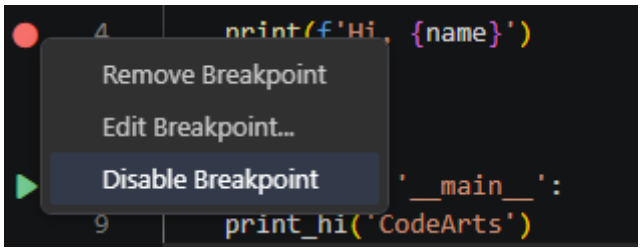


6.9.2.2 启用和禁用断点

您可以启用或禁用单个断点，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**”。



- 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。

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（ ）。

6.9.2.3 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 ( )。

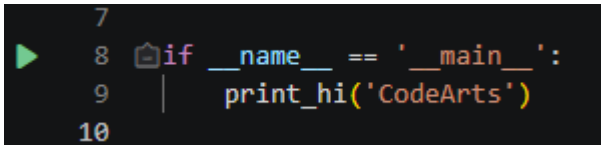
6.9.3 在调试模式下运行程序

CodeArts IDE允许您直接从代码编辑器或通过启动配置启动调试会话。

6.9.3.1 从代码编辑器启动调试会话

如果您不打算向您的程序传递任何参数，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，您可以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。



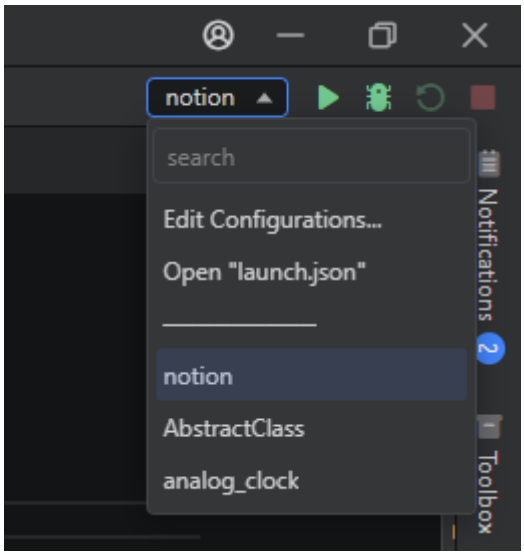
说明

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。

6.9.3.2 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“**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，并按 “F5” / “F11” / “Shift+F9”（IDEA快捷键）。

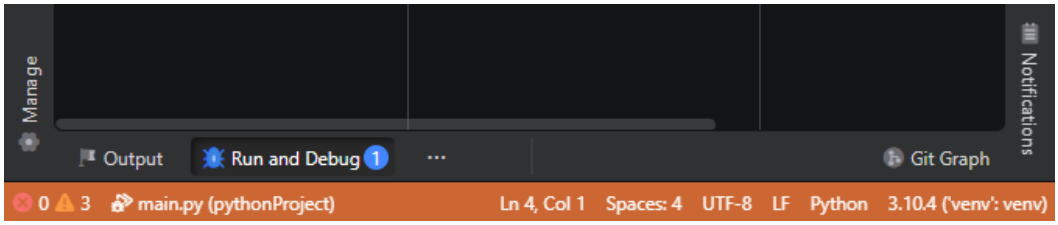


步骤2 执行以下操作之一：

- 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（）。

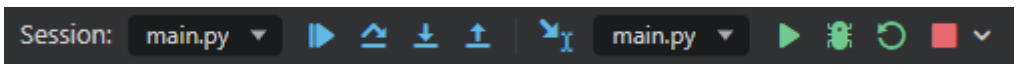
----结束

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**，显示活动的调试配置。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。



6.9.4 控制程序执行

启动调试会话后，您可以使用调试工具栏操作控制程序执行。



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。

图标	对应动作	快捷键	描述
	单步跳过	“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。

6.9.4.1 运行到光标处

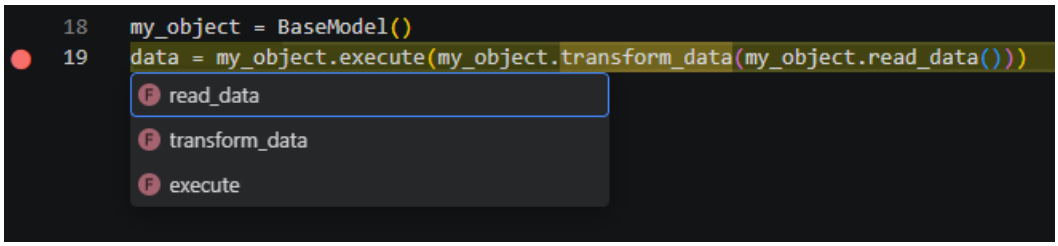
当程序暂停时，您可以继续执行到光标位置。在代码编辑器中，右键单击所需的行，然后从上下文菜单中选择“**运行到光标处**”或按 “Alt+F9” （IDEA快捷键）。

6.9.4.2 进入目标单步执行

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” 。

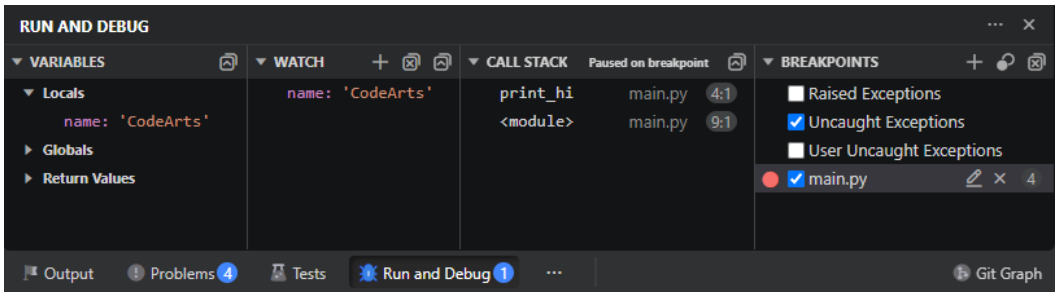
步骤2 在弹出菜单中，选择您要单步执行的方法。



----结束

6.9.5 检查暂停的程序

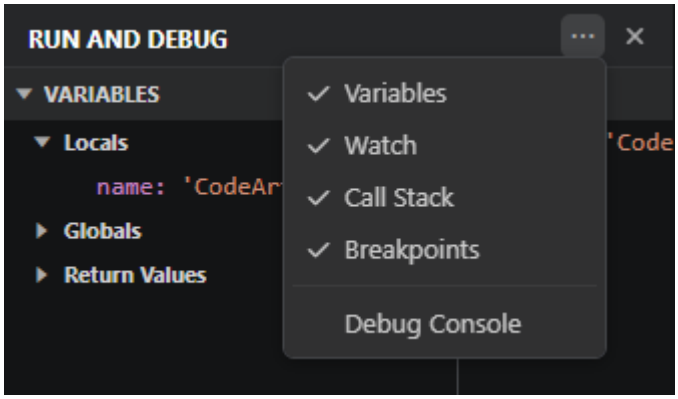
当您启动调试会话时，“**运行和调试**”视图将自动打开并显示与运行和调试相关的所有信息。要手动打开“**运行和调试**”视图，请单击CodeArts IDE底部面板中的“**运行和调试**”按钮（ ），或按 “Ctrl+Shift+D” / “Shift+Alt+F9” （IDEA快捷键）/ “Alt+5” （IDEA快捷键）/ “Ctrl +Shift+F8” （IDEA快捷键）。



“**运行和调试**”视图包含以下部分：

- **变量**
- **调用堆栈**
- **检视**
- **断点**

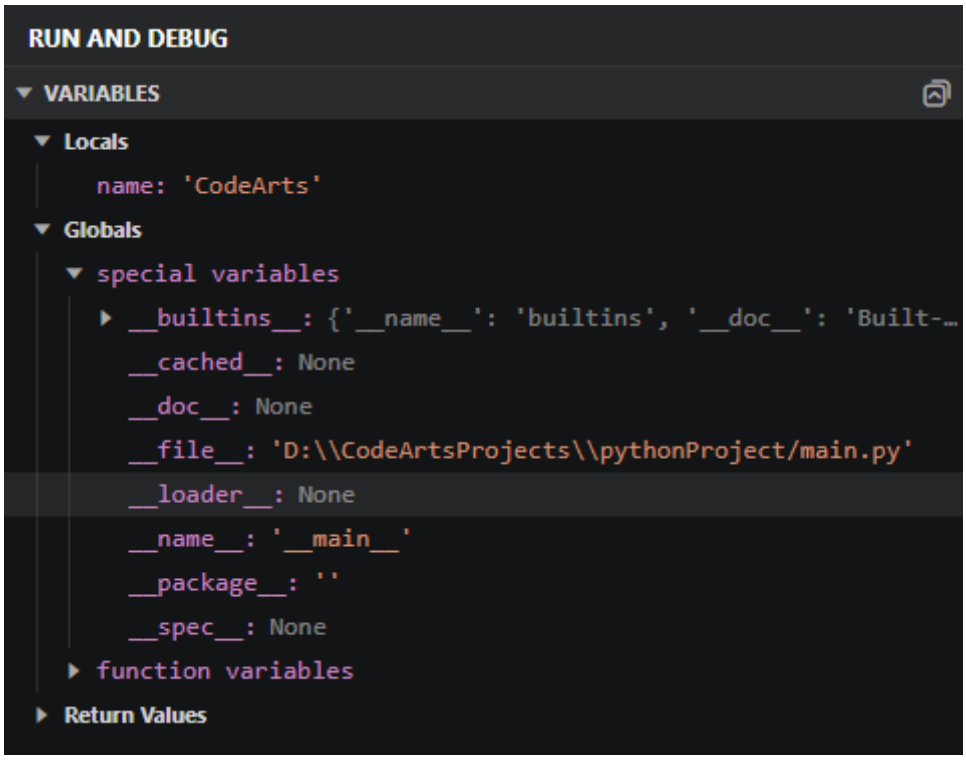
要自定义“**运行和调试**”视图内容，请单击右上角的“**视图和更多操作**”按钮（ ），然后在上下文菜单中勾选要显示的部分。或者可以右键单击“**运行和调试**”视图中任意部分的标题栏，然后在上下文菜单中选择。



6.9.5.1 检查变量

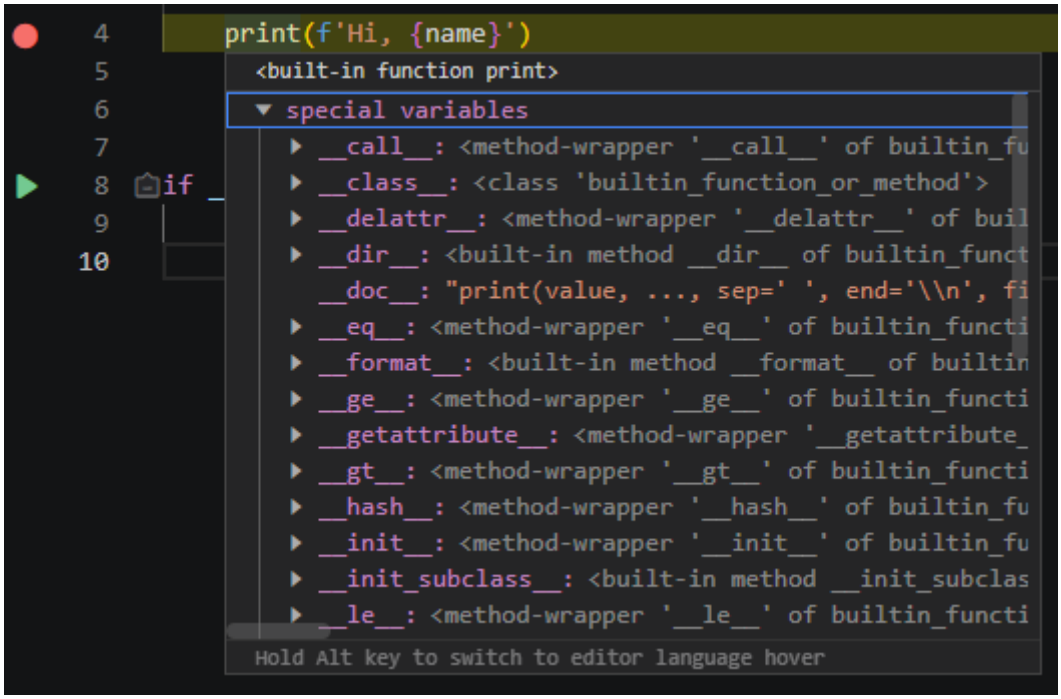
“**变量**”部分显示在当前堆栈帧（即在“**调用堆栈**”部分中选定的堆栈帧）中可访问的元素，并包含以下部分：

- **局部变量**：列出局部变量。
- **全局变量**：列出全局变量，包括特殊（带双下划线）的变量。
- **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。



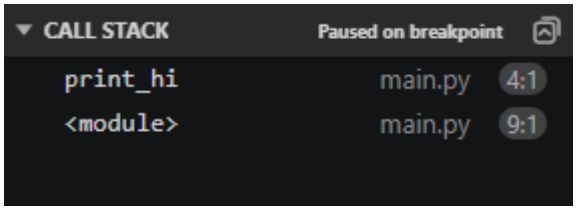
您可以通过在变量上右键单击并从上下文菜单中选择“**设置值**”来修改变量的值。此外，您还可以使用“**复制值**”操作来复制变量的值，或者使用“**复制为表达式**”操作来复制一个用于访问该变量的表达式。

您还可以在“**运行和调试**”视图的“**监视**”部分中评估和监视变量和表达式。也可以在CodeArts IDE代码编辑器中直接评估和检查表达式的值。做法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。



6.9.5.2 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。

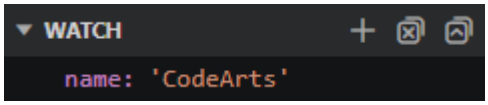


在堆栈帧内可访问的元素会在“**变量**”部分中列出。

- 要切换到不同的堆栈帧，只需在“**调用堆栈**”部分中双击该帧即可。
- 要快速在代码编辑器中打开方法调用，可以展开“**调用堆栈**”中的某个帧，双击想要查看的方法调用。

6.9.5.3 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。



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**”。

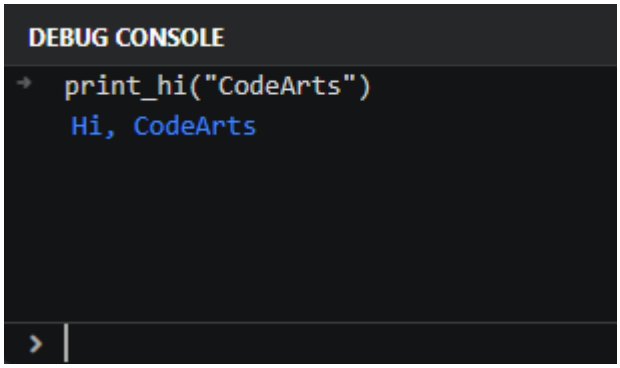
要删除一个表达式，只需选择它并按 “Delete” 键。若要一次性删除所有表达式，请单击“**删除所有表达式**”按钮（ ）。

6.9.5.4 断点

断点部分允许您管理断点，详情信息请参阅**断点**。

6.9.6 调试控制台 REPL

在调试会话期间，您可以通过**调试控制台** REPL（Read-Eval-Print Loop，读取-求值-输出循环）来评估表达式。



- 要打开**调试控制台**，请在CodeArts IDE窗口的底部单击“**调试控制台**”按钮（），或者按“Ctrl+Shift+Y”/“Shift+Escape”（IDEA快捷键）。
- 要开始新的一行，请按“Shift+Enter”。
- 要评估一个表达式，请按“Enter”。

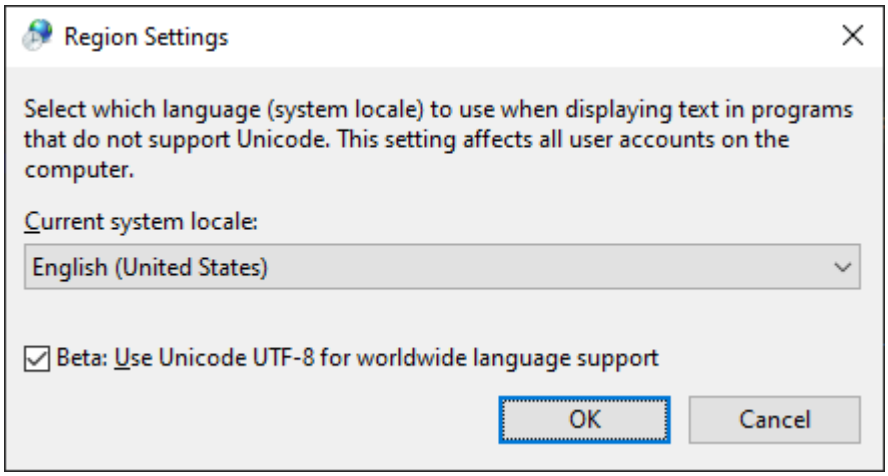
约束与限制

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：

步骤1 在 **Windows 控制面板**中，转到“**时钟和区域**”>“**区域**”。

步骤2 在“**管理**”选项卡上，单击“**更改系统区域设置**”。

步骤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**”。



步骤4 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

----结束

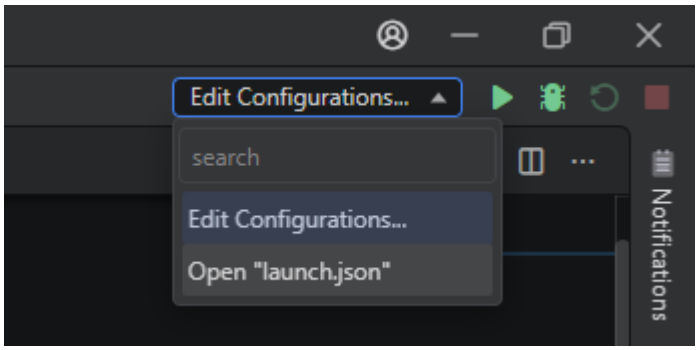
6.10 启动配置

6.10.1 简介

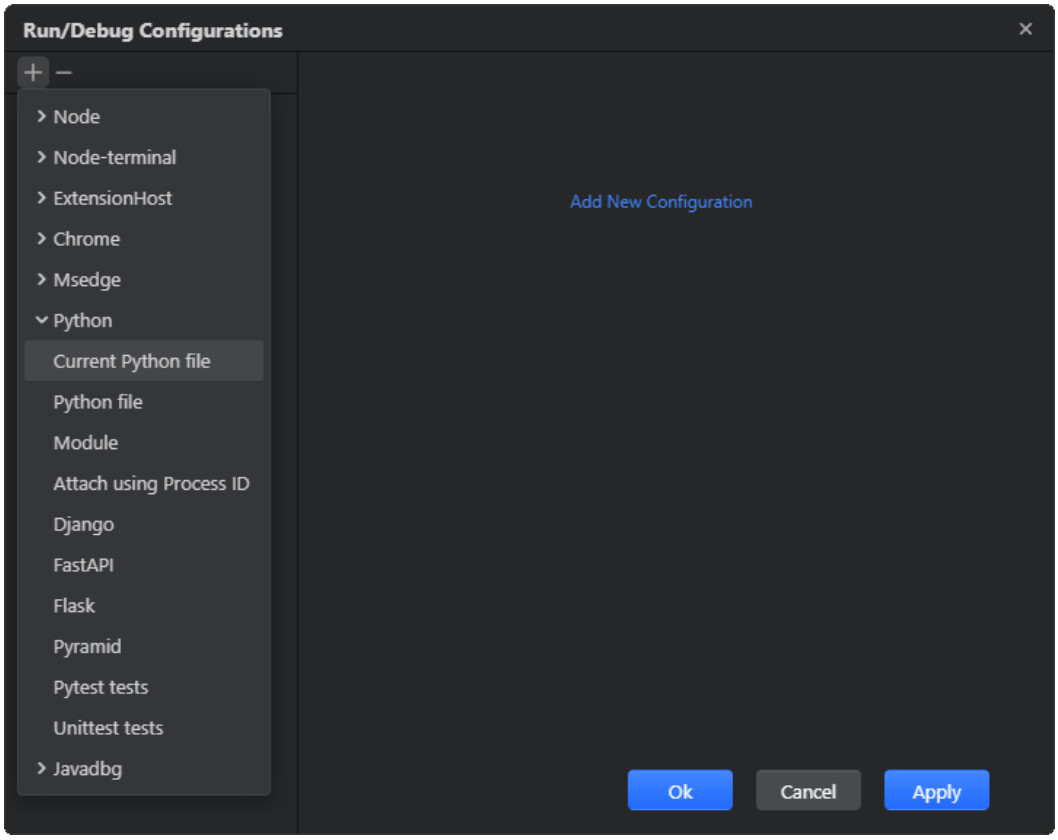
启动配置允许您配置和保存各种场景下运行或调试的设置细节。

6.10.1.1 创建 Python 启动配置

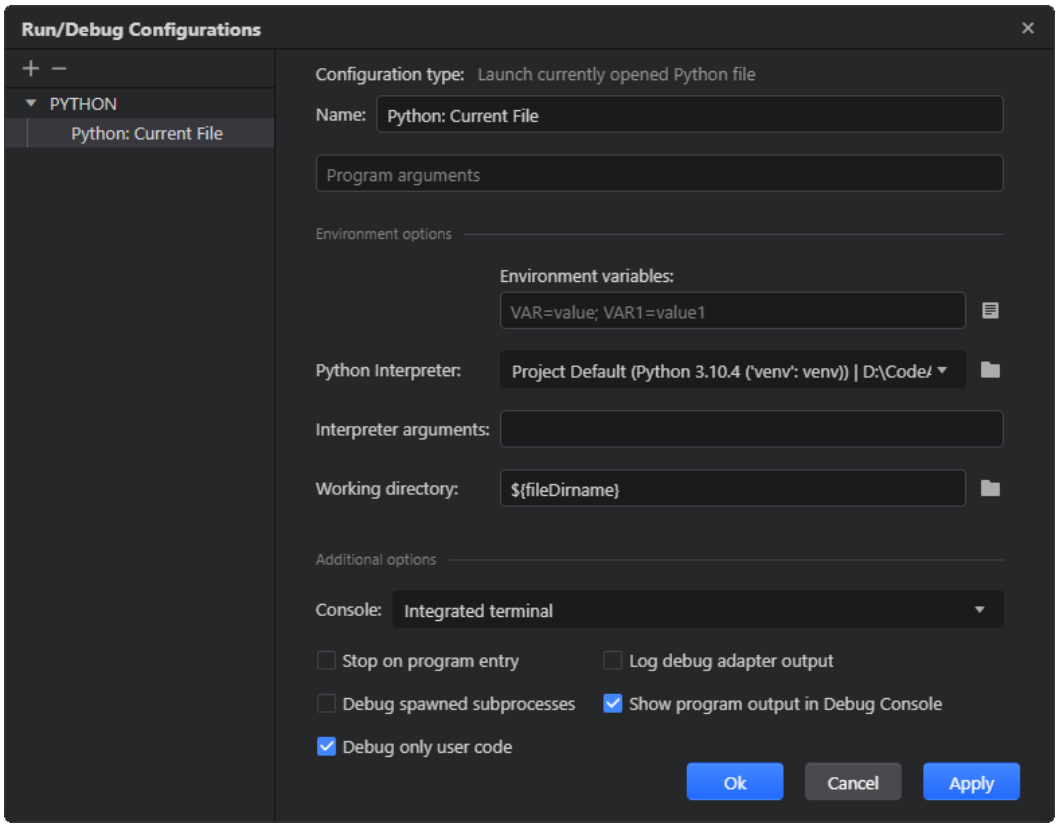
步骤1 在CodeArts IDE主工具栏上，从列表中选择“**编辑配置**”。



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。



步骤3 在参数区域中提供启动配置参数。



说明

关于启动配置参数的详细说明，请参考相应的主题：

- **当前 Python 文件**
- **Python 文件**
- **Python 模块**
- **附加到进程**
- **Django 应用程序**
- **FastAPI 应用程序**
- **Flask 应用程序**
- **Pyramid 应用程序**
- **pytest**
- **unittest**

步骤4 单击“**确定**”以应用更改并关闭对话框。

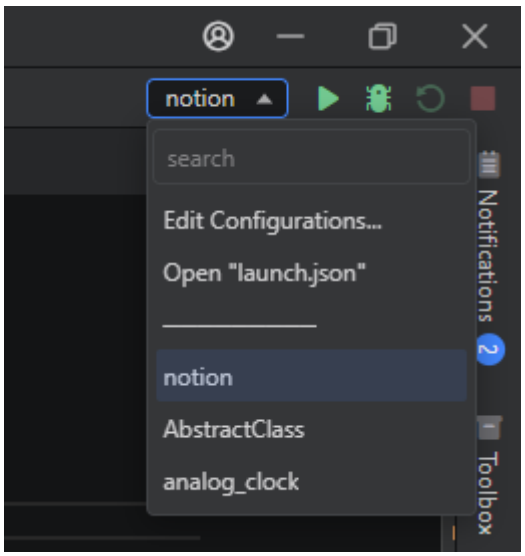
----结束

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“**删除所选配置**”按钮（ ）。

6.10.1.2 动态启动配置

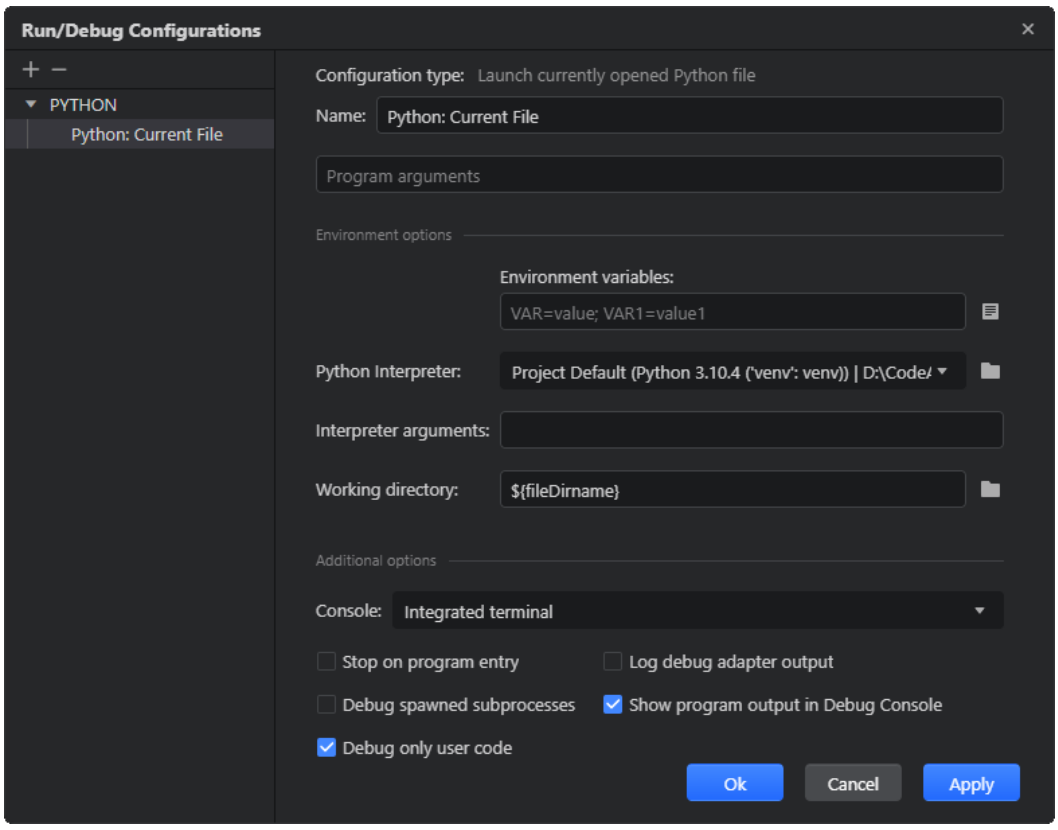
您可以通过在“资源管理器”或代码编辑器中右键单击任意Python文件，并从上下文菜单中选择“**在终端中运行 Python 文件**”来运行它。您还可以使用编辑器边栏中的按钮直接从代码编辑器中运行Python测试。在这些情况下，CodeArts IDE会根据运行的文件自动创建 **Python 文件**、**pytest** 或 **unittest** 启动配置。

然后，您可以从CodeArts IDE主工具栏中选择并运行创建的启动配置。



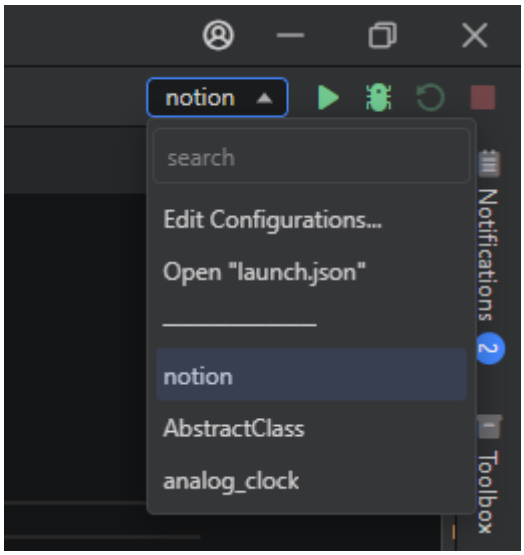
6.10.2 当前 Python 文件

使用此启动配置运行当前在代码编辑器中打开的Python文件。



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”**在终端中运行 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。



6.10.2.1 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”。
“console”	调试输出目的地可以是集成终端（“integratedTerminal”，默认）、调试控制台（“internalConsole”）或外部终端应用程序（“externalTerminal”）。
“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“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“program”	被调试文件的路径，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。

名称	描述
“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}”解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“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。

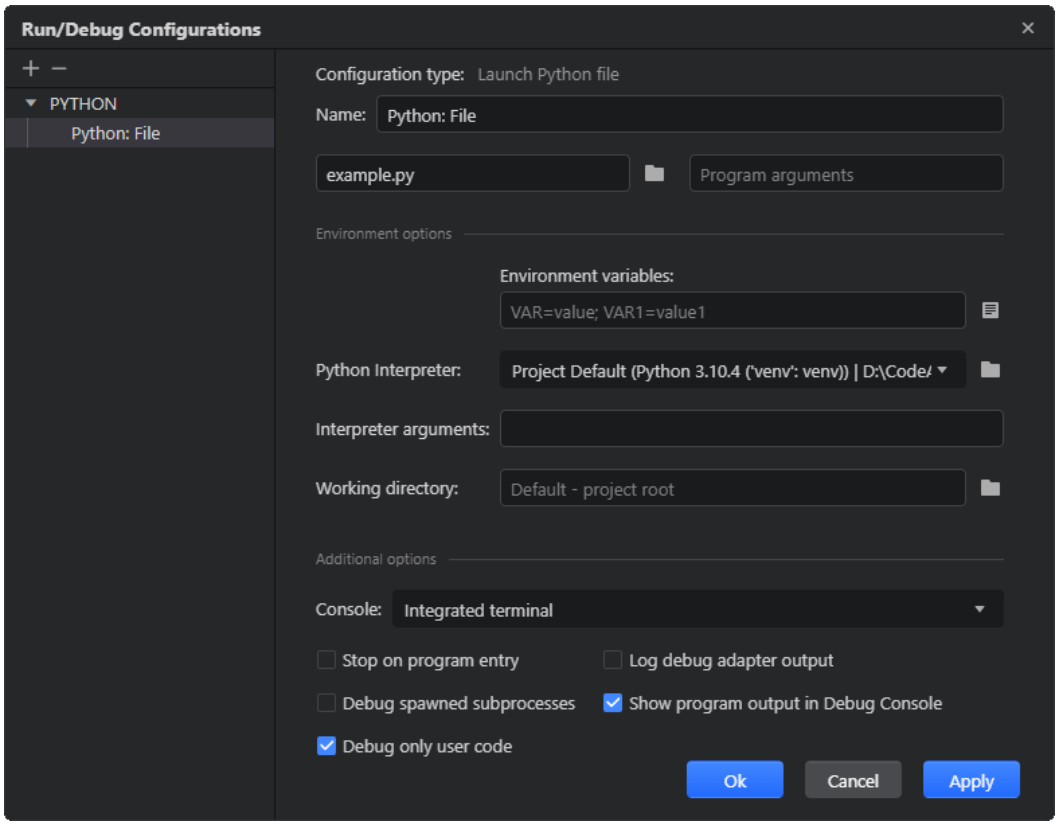
6.10.2.2 启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "request": "launch",
    "console": "integratedTerminal",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "redirectOutput": true,
    "program": "example.py",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: File",
    "showReturnValue": true
}
```

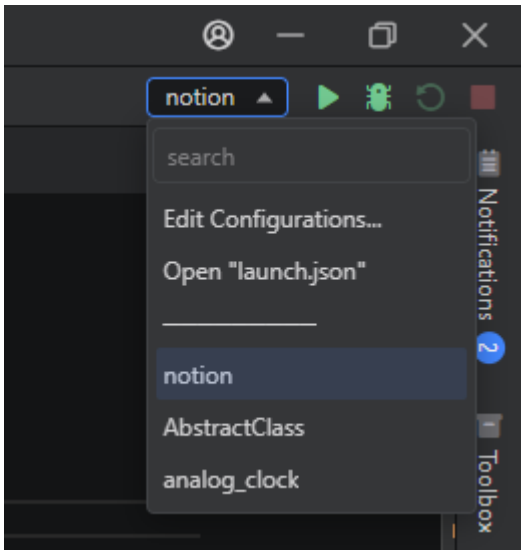
6.10.3 Python 文件

可以使用该配置来运行任意Python文件。



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“**在终端中运行 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。



6.10.3.1 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”。
“console”	调试输出目的地可以是集成终端（“integratedTerminal”，默认）、调试控制台（“internalConsole”）或外部终端应用程序（“externalTerminal”）。
“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“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“program”	被调试文件的路径，您可以使用变量来提供。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给被调试程序的命令行参数。
“cwd”	调试程序工作目录的绝对路径。默认值 “${workspaceFolder}” 解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“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”。

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。

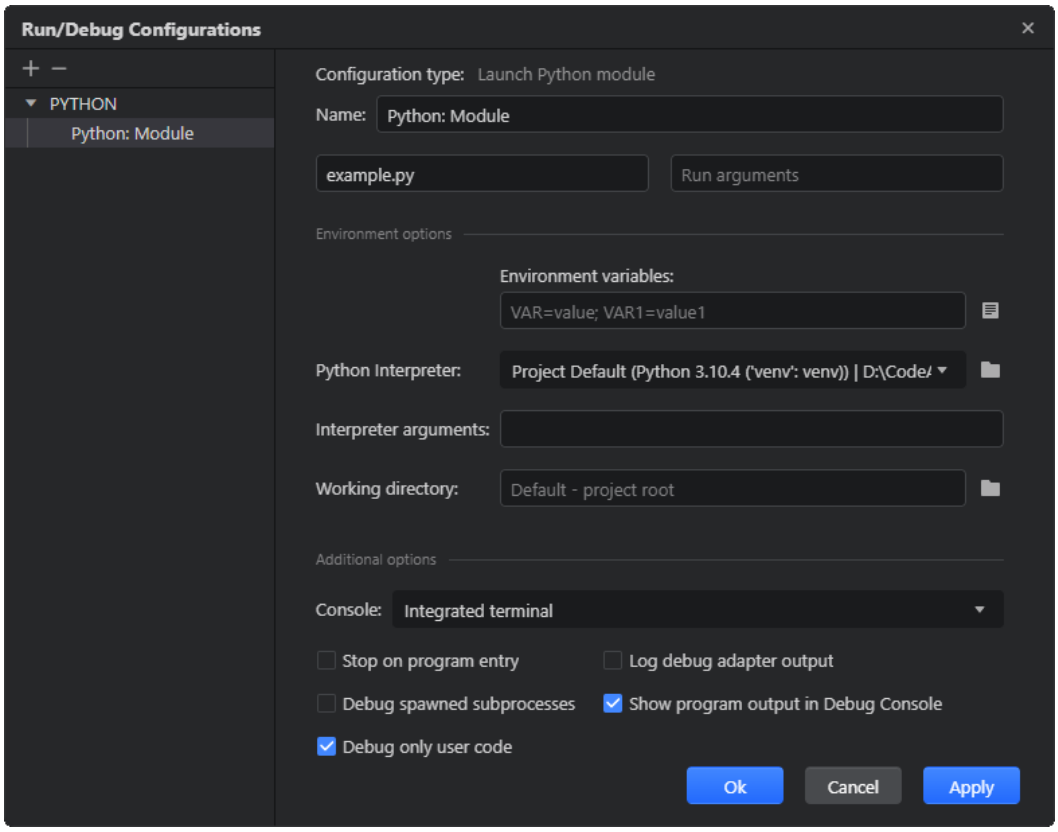
6.10.3.2 启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "request": "launch",
    "console": "integratedTerminal",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "redirectOutput": true,
    "program": "example.py",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: File",
    "showReturnValue": true
}
```

6.10.4 Python 模块

您可以使用该配置来运行任意Python模块。



6.10.4.1 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模块启动配置，此选项始终设置为“launch”。
“console”	调试输出目的地可以是集成终端（“integratedTerminal”，默认）、调试控制台（“internalConsole”）或外部终端应用程序（“externalTerminal”）。
“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“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。

名称	描述
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“module”	被调试模块的路径，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。
“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}” 解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“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。

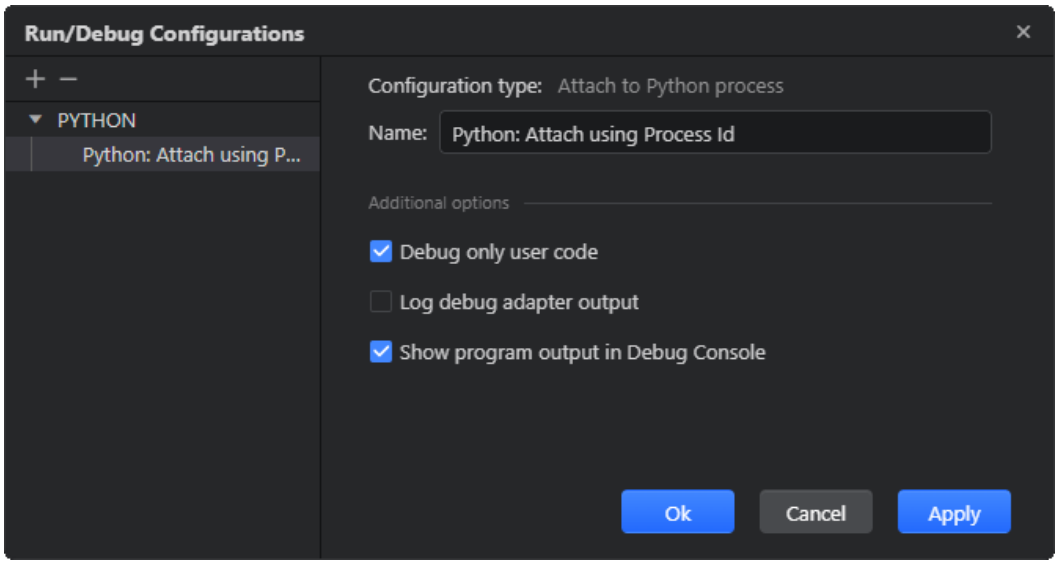
6.10.4.2 启动配置示例

以下是一个可运行的启动配置示例。

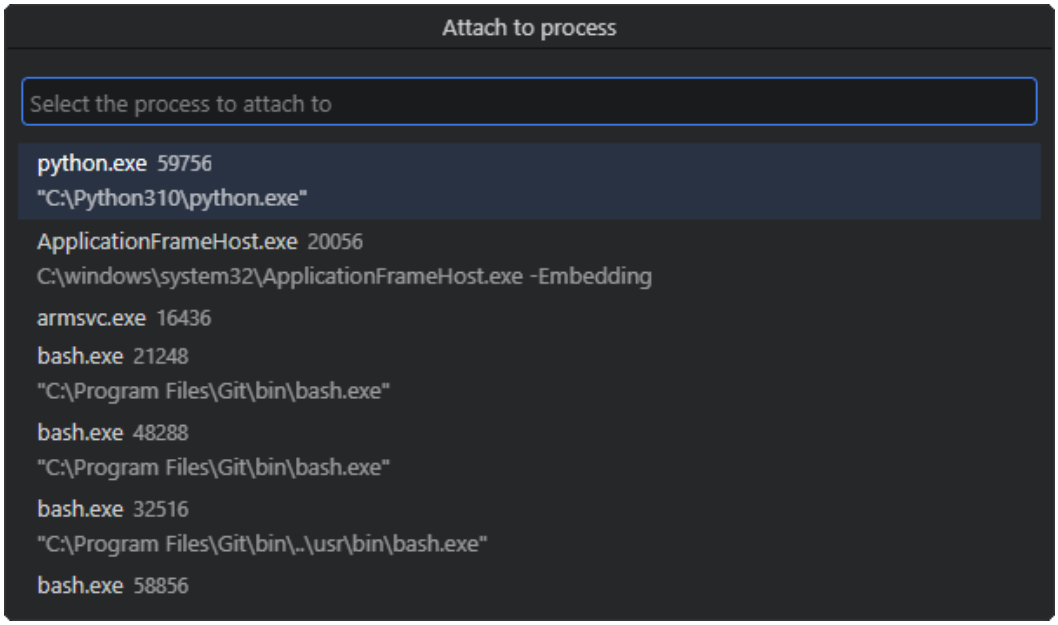
```
{
    "request": "launch",
    "console": "integratedTerminal",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "redirectOutput": true,
    "module": "example.py",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: Module",
    "showReturnValue": true
}
```

6.10.5 附加到进程

您可以使用该启动配置将调试器附加到已运行的Python程序。



当您启动该配置时，CodeArts IDE会提示您选择要附加的进程。



6.10.5.1 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。

名称	描述
“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。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“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。

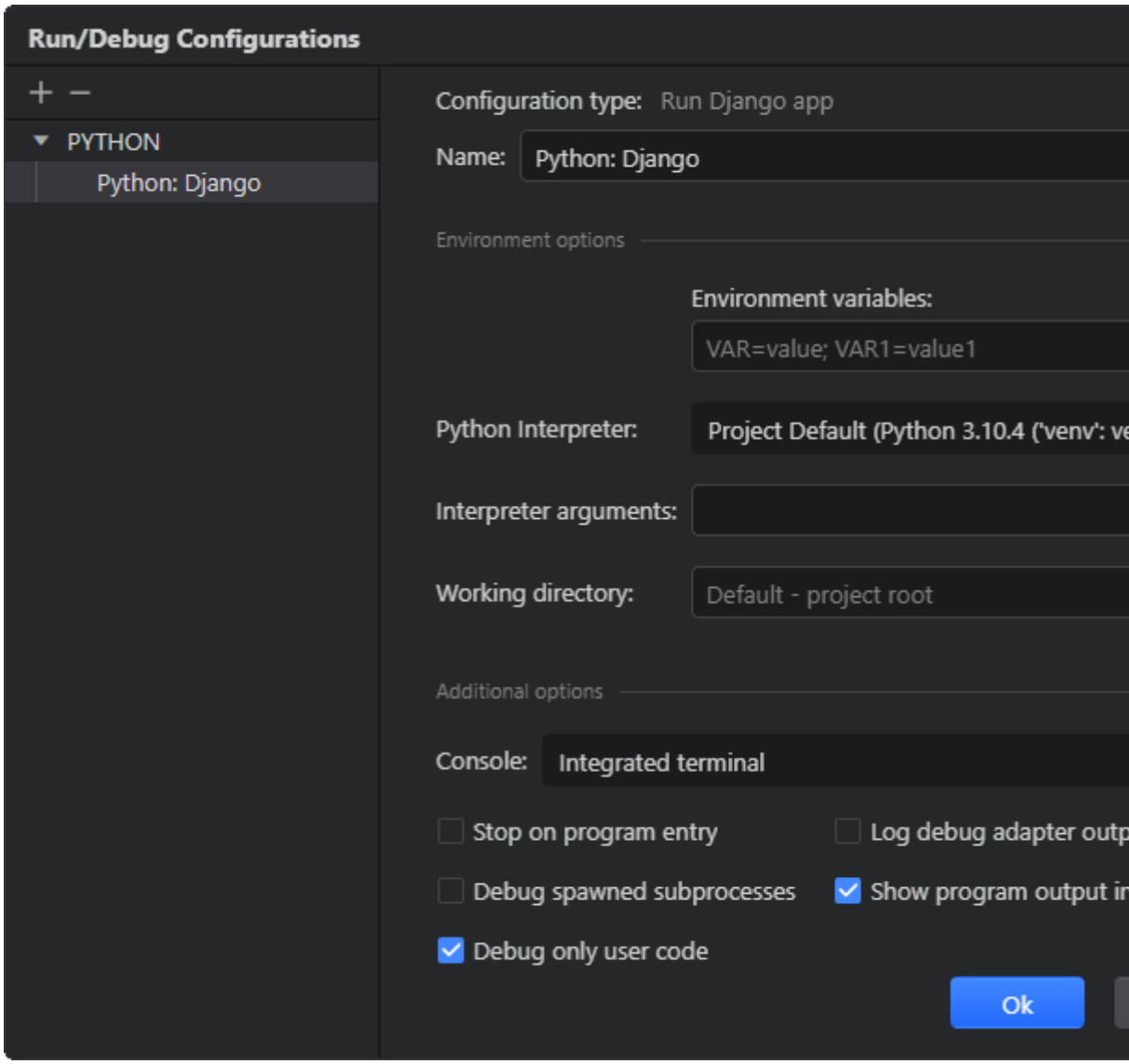
6.10.5.2 启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "request": "attach",
    "jinja": true,
    "justMyCode": true,
    "processId": "${command:pickProcess}",
    "redirectOutput": true,
    "name": "Python: Attach using Process Id",
    "type": "python",
    "logToFile": false,
    "showReturnValue": true
}
```

6.10.6 Django 应用

以下是一个可运行的启动配置示例。该配置执行 “python manage.py runserver” 命令来启动内置的 Django 开发服务器。



6.10.6.1 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Django应用启动配置，此选项始终设置为“launch”。
“console”	调试输出目的地可以是集成终端（“integratedTerminal”，默认）、调试控制台（“internalConsole”）或外部终端应用程序（“externalTerminal”）。

名称	描述
“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“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“program”	“manage.py” 的路径，它是 Django 框架中的一个命令行管理工具。默认将设置为 “${workspaceFolder}\\manage.py” ，即项目根目录的“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。
“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}” 解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“django”	如果设置为 “true”（默认），CodeArts IDE将为Django页面模板启用对应调试功能。
“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。

6.10.6.2 启动配置示例

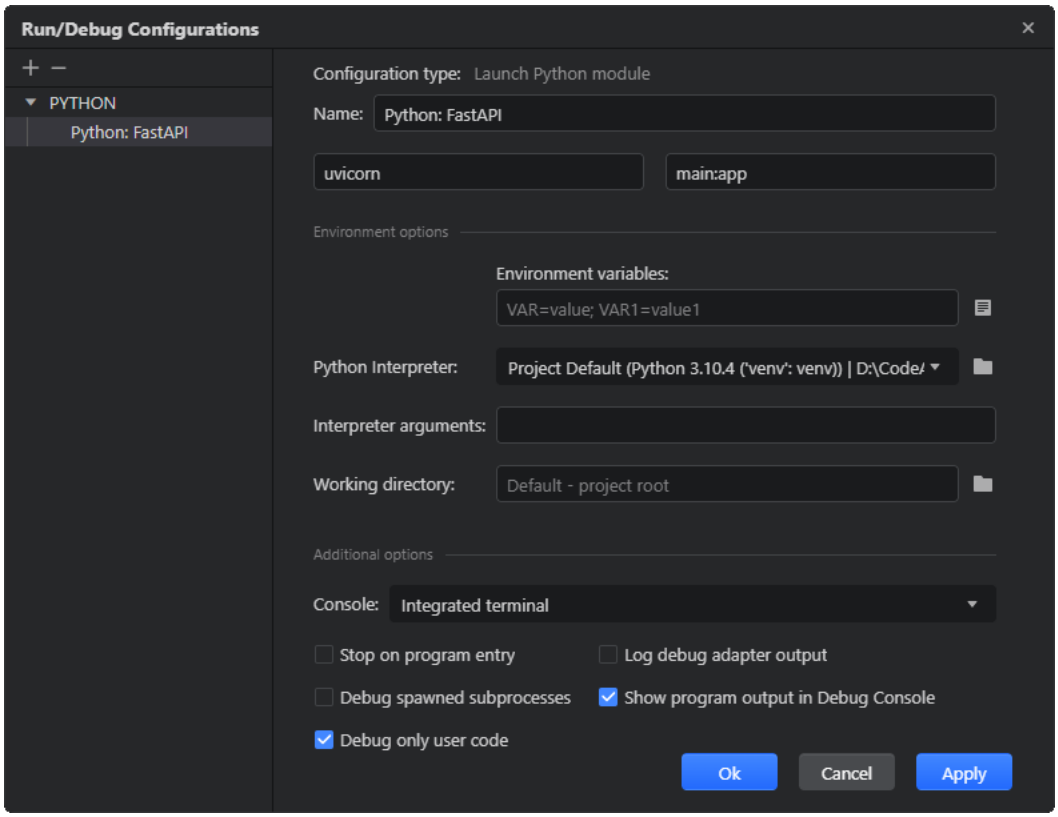
以下是一个可运行的启动配置示例。

```
{
  "request": "launch",
```

```
    "console": "integratedTerminal",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "redirectOutput": true,
    "program": "${workspaceFolder}\\manage.py",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [
        "runserver"
    ],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "django": true,
    "pythonArgs": [],
    "name": "Python: Django",
    "showReturnValue": true
}
```

6.10.7 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，启动一个运行您应用程序的**uvicorn**服务器。



6.10.7.1 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”。
“console”	调试输出目的地可以是集成终端（“integratedTerminal”，默认）、调试控制台（“internalConsole”）或外部终端应用程序（“externalTerminal”）。
“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“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“module”	用于运行应用程序服务器的 “uvicorn” 模块的路径，默认情况下设置为 “uvicorn”。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给被调试程序的命令行参数。
“cwd”	调试程序工作目录的绝对路径。默认值 “${workspaceFolder}” 解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。

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。

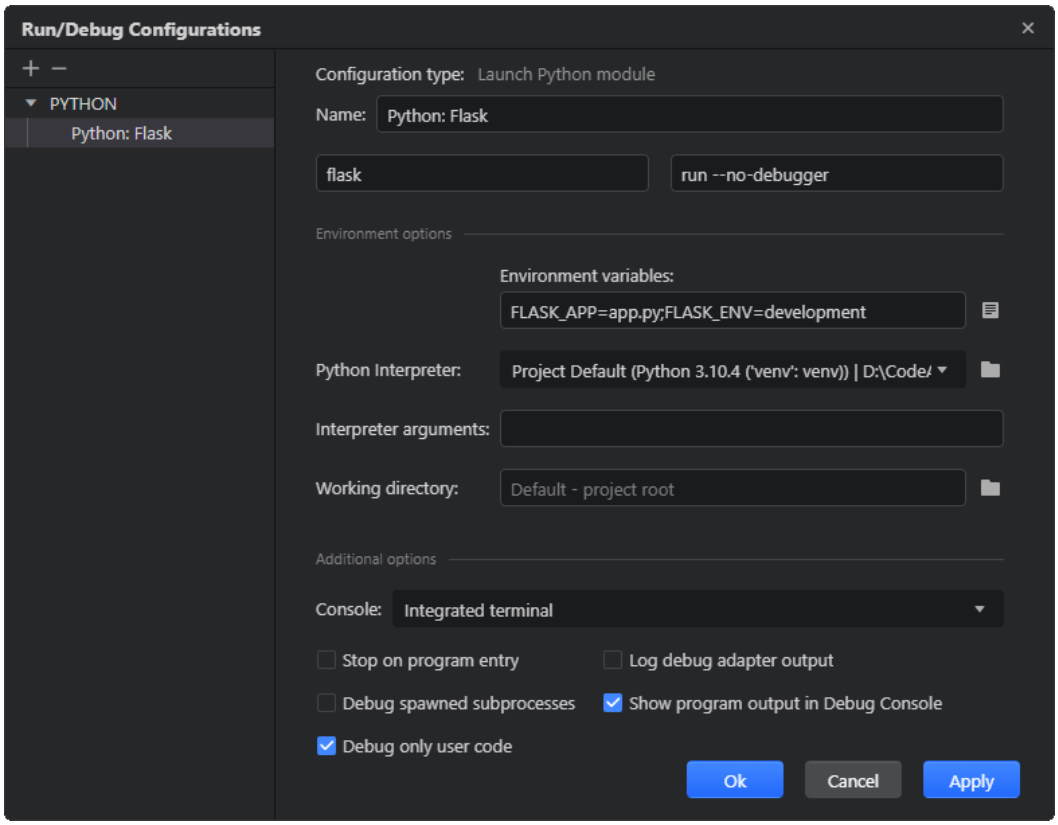
6.10.7.2 启动配置示例

以下是一个可运行的启动配置示例。

```
{
    "request": "launch",
    "console": "integratedTerminal",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "redirectOutput": true,
    "module": "uvicorn",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [
        "main:app"
    ],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: FastAPI",
    "showReturnValue": true
}
```

6.10.8 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。



6.10.8.1 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”。
“console”	调试输出目的地可以是集成终端（“integratedTerminal”，默认）、调试控制台（“internalConsole”）或外部终端应用程序（“externalTerminal”）。
“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“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。

名称	描述
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“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。
“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"}”。
“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}” 解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“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。

6.10.8.2 启动配置示例

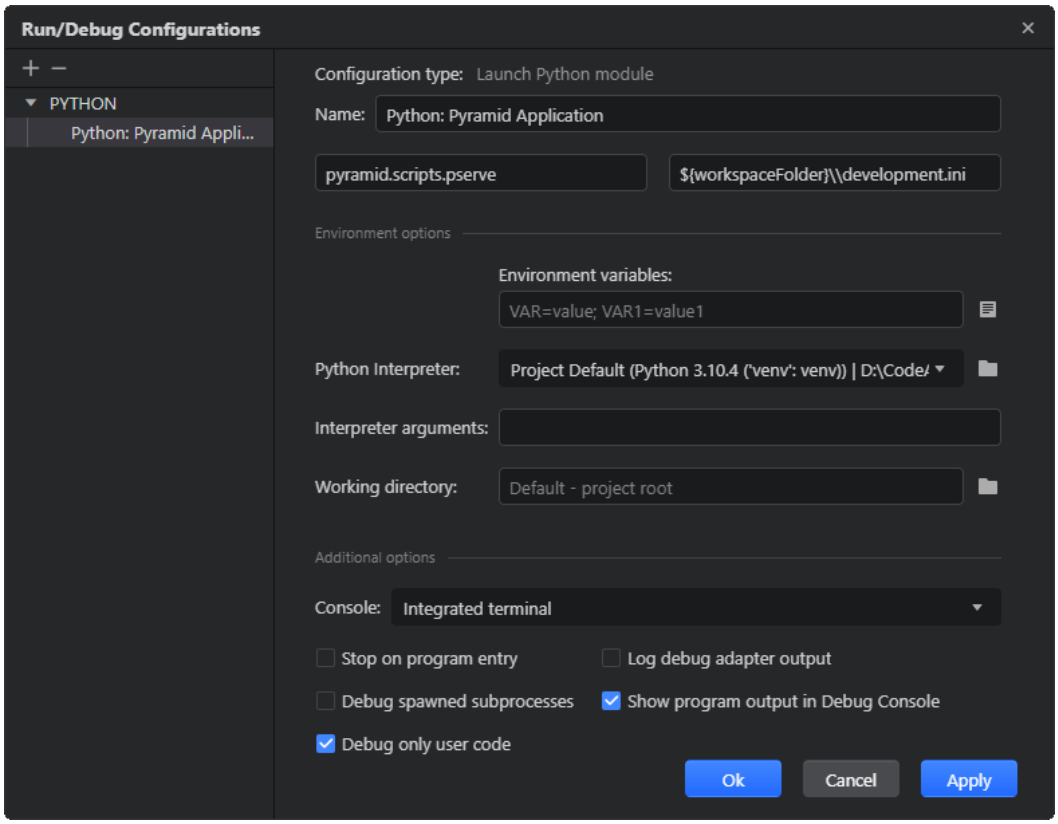
以下是一个可运行的启动配置示例。

```
{
    "request": "launch",
    "console": "integratedTerminal",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "redirectOutput": true,
    "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
}
```

6.10.9 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。



6.10.9.1 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”。

名称	描述
“console”	调试输出目的地可以是集成终端（“integratedTerminal”，默认）、调试控制台（“internalConsole”）或外部终端应用程序（“externalTerminal”）。
“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“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“module”	用于运行 Pyramid 应用程序服务器的模块的名称。默认设置为“pyramid.scripts.pserve”。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。默认设置为 “{"FLASK_APP": "app.py","FLASK_ENV": "development"}”。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“args”	传递给被调试程序的命令行参数。
“cwd”	调试程序工作目录的绝对路径。默认值 “${workspaceFolder}” 解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“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。

6.10.9.2 启动配置示例

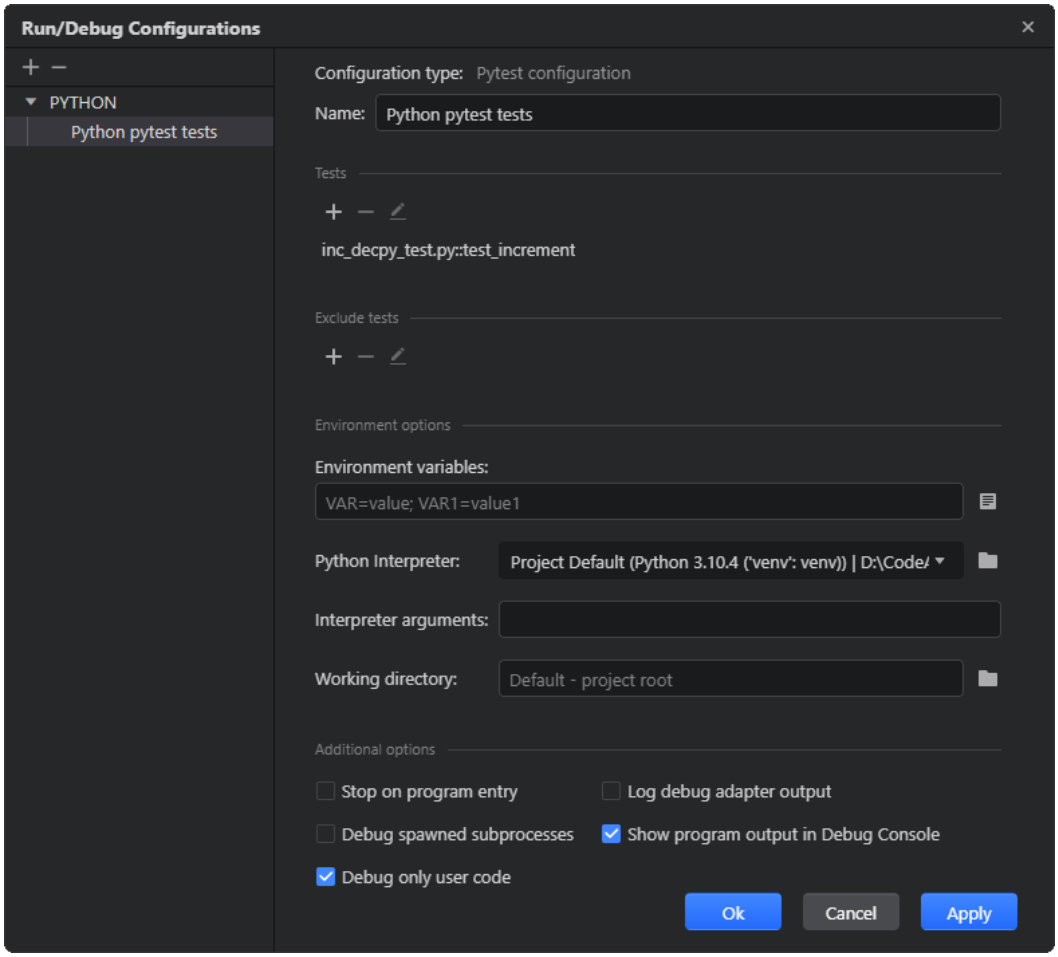
以下是一个可运行的启动配置示例。

```
{
   "request": "launch",
```

```
    "console": "integratedTerminal",
    "jinja": true,
    "python": "${command:python.interpreterPath}",
    "stopOnEntry": false,
    "redirectOutput": true,
    "module": "pyramid.scripts.pserve",
    "env": {},
    "type": "python",
    "logToFile": false,
    "args": [
        "${workspaceFolder}\\development.ini"
    ],
    "cwd": "${workspaceFolder}",
    "subProcess": false,
    "justMyCode": true,
    "pythonArgs": [],
    "name": "Python: Pyramid Application",
    "showReturnValue": true
}
```

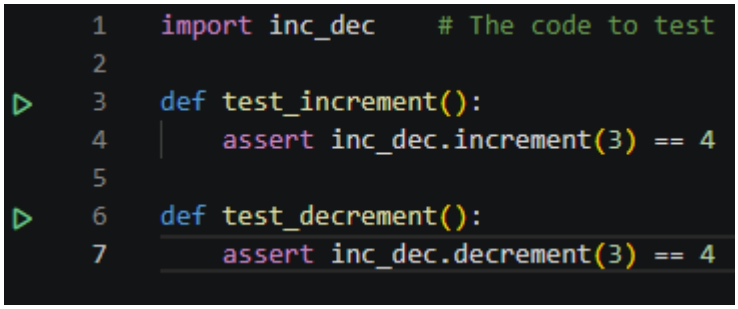
6.10.10 pytest

使用该启动配置来运行pytest测试。



约束与限制

要在没有手动创建启动配置的时候快速运行 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）。



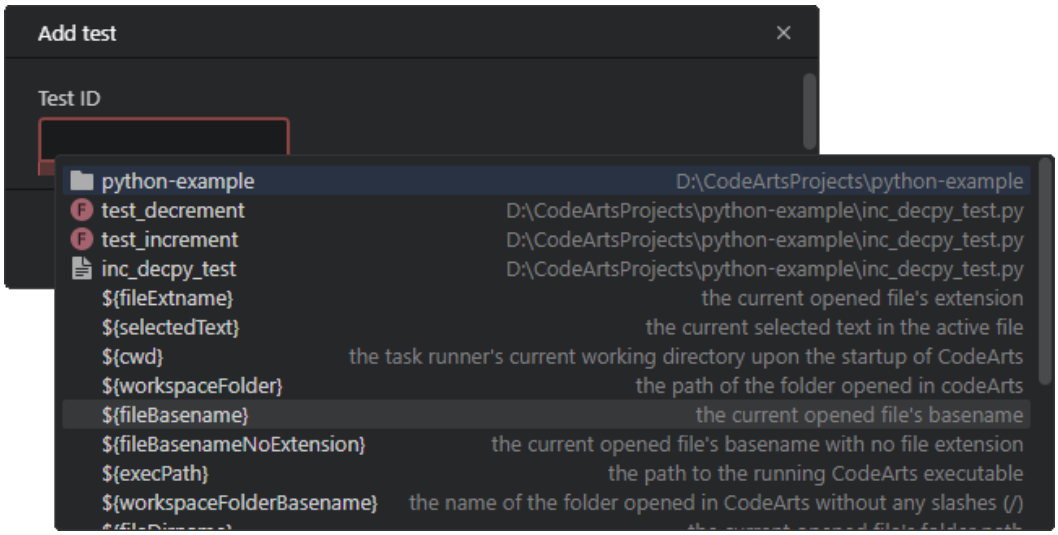
CodeArts IDE会自动创建相应的pytest启动配置并将其显示在配置列表中。

6.10.10.1 从启动配置中包含/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**代码补全**功能（“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“**Save**”以将所选测试添加到列表中。

----结束

要从列表中删除测试，请选择它并单击“**Remove Selected**”按钮（）。

6.10.10.2 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“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“cwd”	调试程序工作目录的绝对路径。默认值 “${workspaceFolder}” 解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“provider”	测试框架。对于 pytest 启动配置，此选项始终设置为“PYTEST”。

名称	描述
“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。

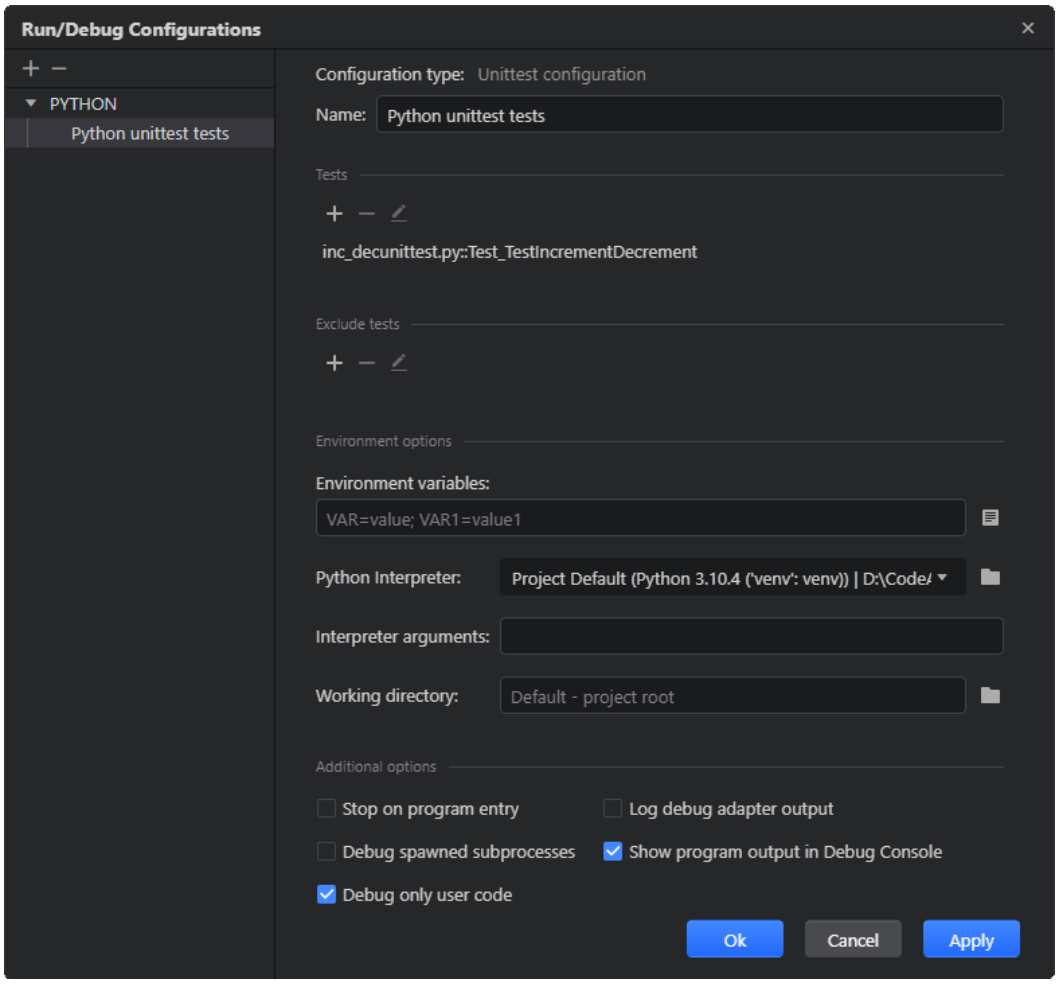
6.10.10.3 启动配置示例

以下是一个可运行的启动配置示例。

```
{
  "excludeTestIds": [],
  "request": "test",
  "jinja": true,
  "python": "${command:python.interpreterPath}",
  "stopOnEntry": false,
  "redirectOutput": true,
  "env": {},
  "type": "python",
  "logToFile": false,
  "testIds": [
      "test_file_name::test_class_name::test_method_name"
  ],
  "cwd": "${workspaceFolder}",
  "subProcess": false,
  "justMyCode": true,
  "provider": "PYTEST",
  "pythonArgs": [],
  "name": "Python pytest tests",
  "showReturnValue": true
}
```

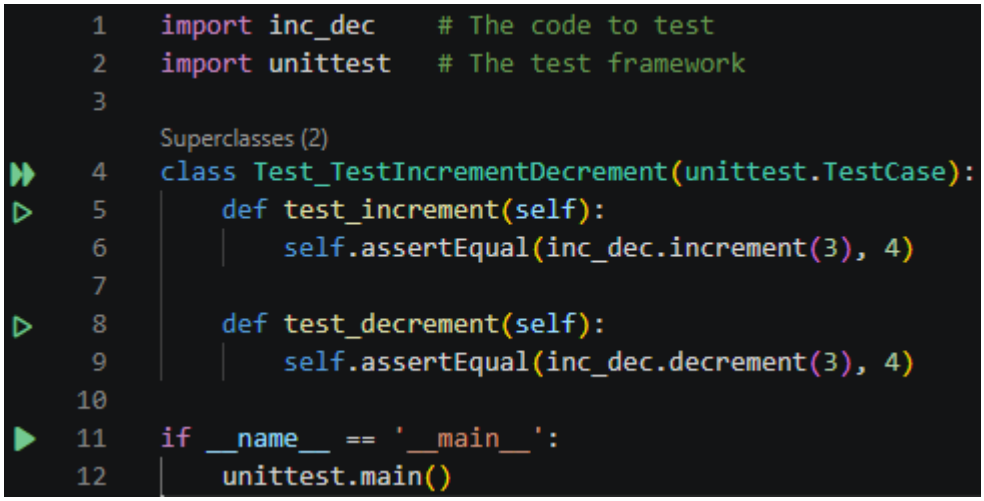
6.10.11 unittest

使用该启动配置来运行**unittest**测试。



约束与限制

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）。



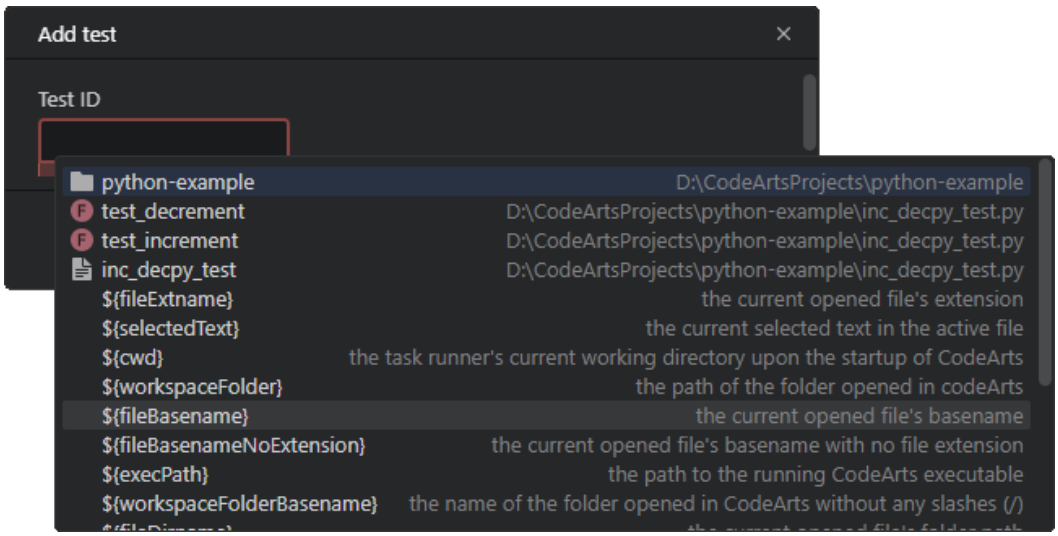
CodeArts IDE会自动创建相应的unittest启动配置并将其显示在配置列表中。

6.10.11.1 从启动配置中包含/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。使用代码补全功能（“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“**Save**”以将所选测试添加到列表中。

----结束

要从列表中删除测试，请选择它并单击“**Remove Selected**”按钮（ ）。

6.10.11.2 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”。

名称	描述
“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“${command:python.interpreterPath}”解析为当前选定的项目解释器。有关在项目中使用Python解释器的详细信息，请参阅“**构建环境**”。
“stopOnEntry”	当设置为“true”时，程序将在启动时自动挂起。
“redirectOutput”	当设置为“true”（默认）时，程序的输出将重定向到作为“console”值设置的控制台。此设置仅适用于将“console”设置为“internalConsole”或“integratedTerminal”时。
“env”	一组定义为键值对的环境变量。属性键为环境变量，属性值为环境变量的值。
“logToFile”	当设置为“true”时，调试器事件将记录到文件中。默认情况下，此选项设置为“false”。默认的日志目录是“%userprofile%\.codearts\extensions\codearts.python-<version>\javaFiles\<project-name>\dap_<date>”。
“cwd”	调试程序工作目录的绝对路径。默认值 “${workspaceFolder}” 解析为项目根文件夹。
“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。否则将显示和调试所有包括库调用的代码。
“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。
“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。

6.10.11.3 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,
  "redirectOutput": true,
  "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
}
```

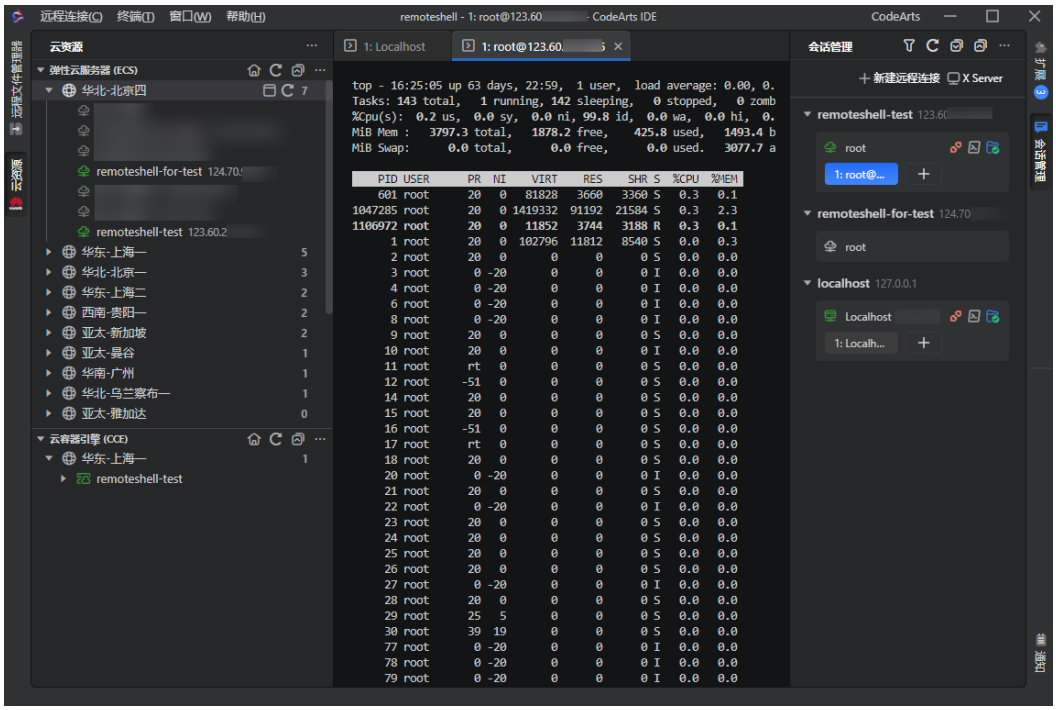
7 RemoteShell

7.1 简介

RemoteShell是CodeArtsIDE的一种产品形态，提供了基于SSH 协议访问具有EIP的华为云ECS主机的文件系统和终端的能力，以及基于 kubectl 访问华为云现网容器集群的能力，让华为云用户轻松访问和使用云资源。

RemoteShell目前可用于Windows。

7.2 用户界面概述



RemoteShell的用户界面由以下主要部分组成：

- **“云资源”**区域，列出与您的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，该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。
- **“通知”**区域，列出了RemoteShell中最近发生的通知和事件。

7.3 管理主机

通过RemoteShell，您可以连接华为云主机或任意主机。对于每个已配置的主机，您可以创建和维护多个用户连接。

添加主机连接

步骤1 执行以下操作之一：

- 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。

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。如果您必须通过代理连接，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 7-1 主机连接参数

参数项	描述
主机地址	选定主机的IP地址
用户名	登录主机的用户
端口	连接的端口
认证方式	通过密码、密钥或双因子认证
用户凭证	密码、私钥等

步骤3 在**“高级”**标签页，可以选择RemoteShell是否应该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。如果您要连接的主机提供 X11转发以通过 SSH 启动图形应用程序，您可以通过选中**“X11转发”**复选框在客户端侧启用它。

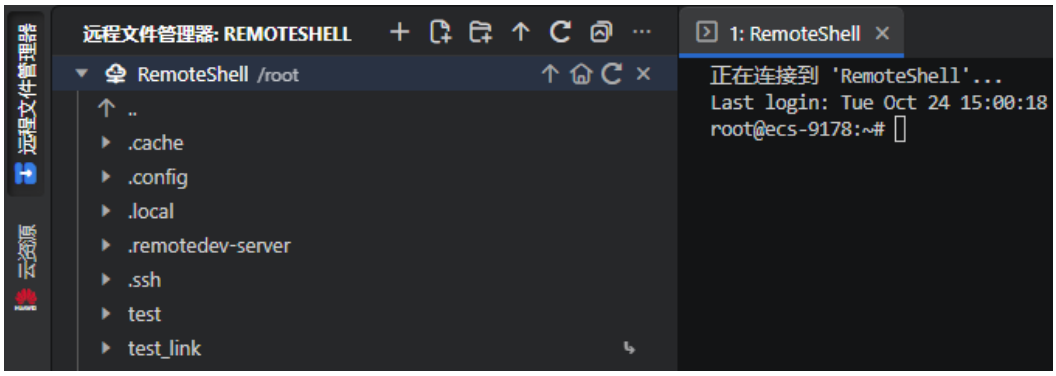
单击**“配置X11转发”**可以指定X11转发的目的地址。如果您选择将X11连接转发到外部的X Server，请在**“X Display的路径或地址”**中设置Display的值。

步骤4 单击**“连接”**。

----结束

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。



编辑主机

步骤1 在“**会话管理**”区域中，单击要编辑的主机右边的···或者单击鼠标右键。

步骤2 在弹出菜单中，选择“**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，单击要编辑的主机右边的···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**”。

----结束

7.4 管理连接

您能够通过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，指定主机连接参数。参考**添加主机连接**。

- 在“**通用**”标签页，指定连接名称、端口、用户名、用户凭据和认证方式。
- 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。

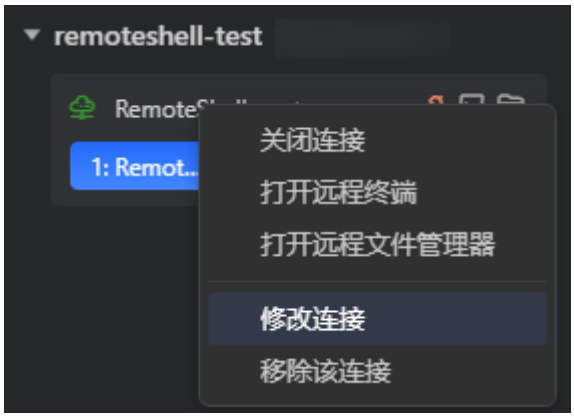
步骤3 单击“**保存**”。

----结束

连接成功后，新的连接记录将添加到所选主机下面。

修改连接

步骤1 在“**会话管理**”区域中，使用鼠标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**”。



步骤2 在打开的“**编辑连接**”窗口中，根据需要修改连接参数。

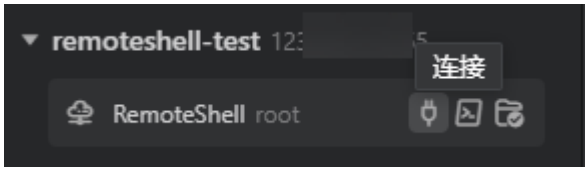
- 在“**通用**”标签页，可以修改连接名称、端口、用户名、认证方式和用户凭据。
- 在“**高级**”标签页，通过勾选“**X11转发**”复选框选择是否启用X11转发。

步骤3 单击“**保存**”以应用更改。

----结束

建立连接

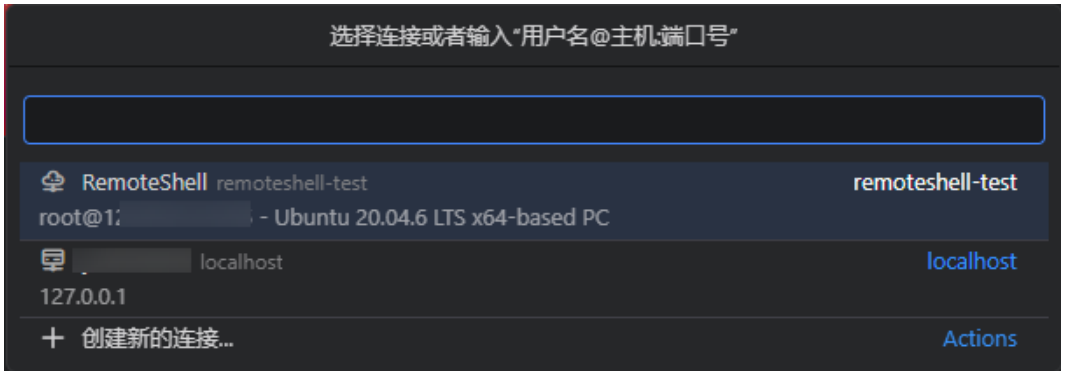
步骤1 在“**会话管理**”区域，单击所需用户连接右边的。



----结束

您还可以通过“选择连接”弹出窗口快速创建和建立连接。

步骤1 在主菜单栏选择“RemoteShell >打开远程连接”（全模式）或“远程连接 >打开远程连接”（精简模式），或者按下**Ctrl+Alt+O**，“选择连接”弹出窗口将被打开



步骤2 执行以下操作之一：

- 选择现有主机的连接之一，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。

----结束

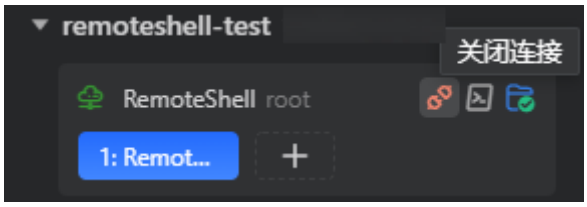
须知

有关添加主机的详细信息，请参阅**添加主机连接**。

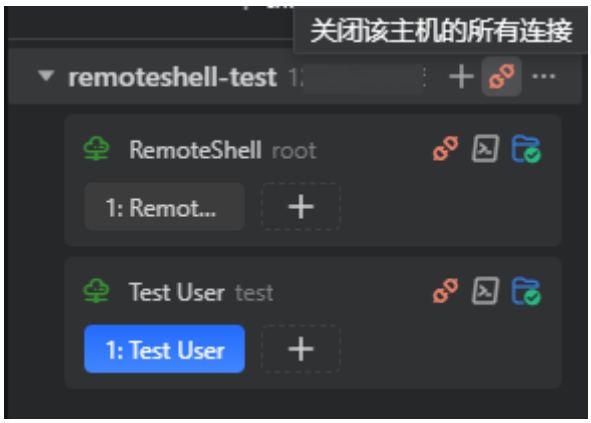
关闭连接

执行以下操作之一：

- 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（）。



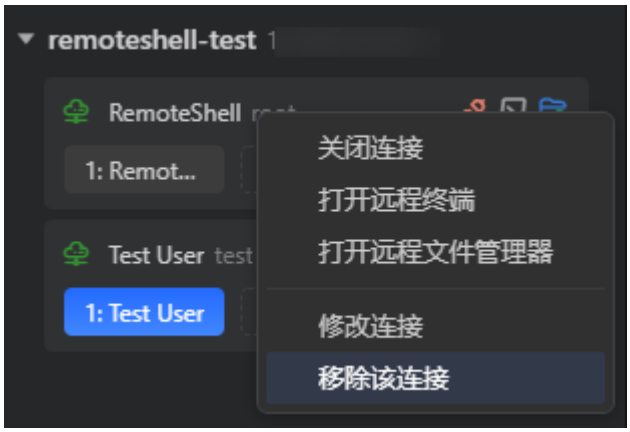
- 要终止与主机的所有连接，请在“**会话管理**”区域中，单击要断开的主机旁边的“**关闭该主机的所有连接**”按钮（）。



移除连接

步骤1 在“**会话管理**”区域中，在要删除的连接记录行单击鼠标右键。

步骤2 在弹出菜单中，选择“**移除该连接**”。



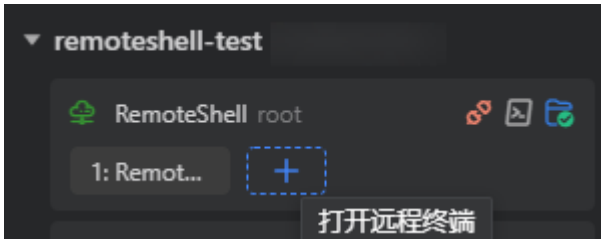
----结束

7.5 管理终端会话

当您与主机建立连接时，RemoteShell会自动启动一个终端会话。如有必要，您可以为每个已建立的连接打开多个单独的终端会话。

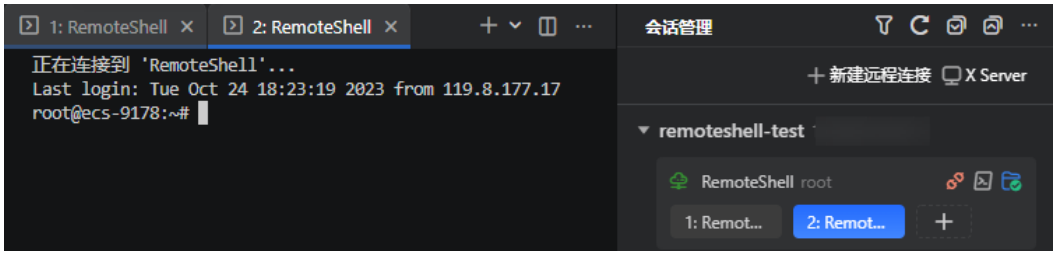
启动终端会话

步骤1 在**“会话管理”**区域中，单击要打开远程终端的连接记录旁边的**“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**。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“RemoteShell>复制远程终端”（全模式）或“终端>复制远程终端”（精简模式），或按**“Ctrl+Alt+D”**。



----结束

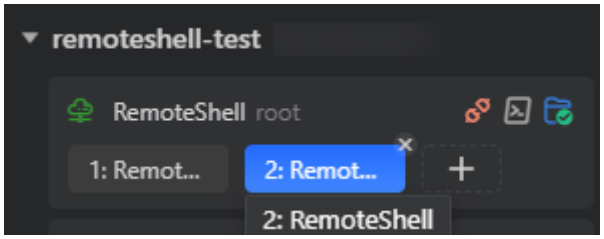
新会话在**“编辑器”**区域的新选项卡中打开，会话记录将添加到**“会话管理”**区域的相应主机下。



关闭终端会话

执行以下任一操作：

- 在“**编辑器**”区域中，单击要关闭的选项卡右侧的“**关闭**”按钮（×），或按**Ctrl+W / Ctrl+F4**。
- 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（×）。



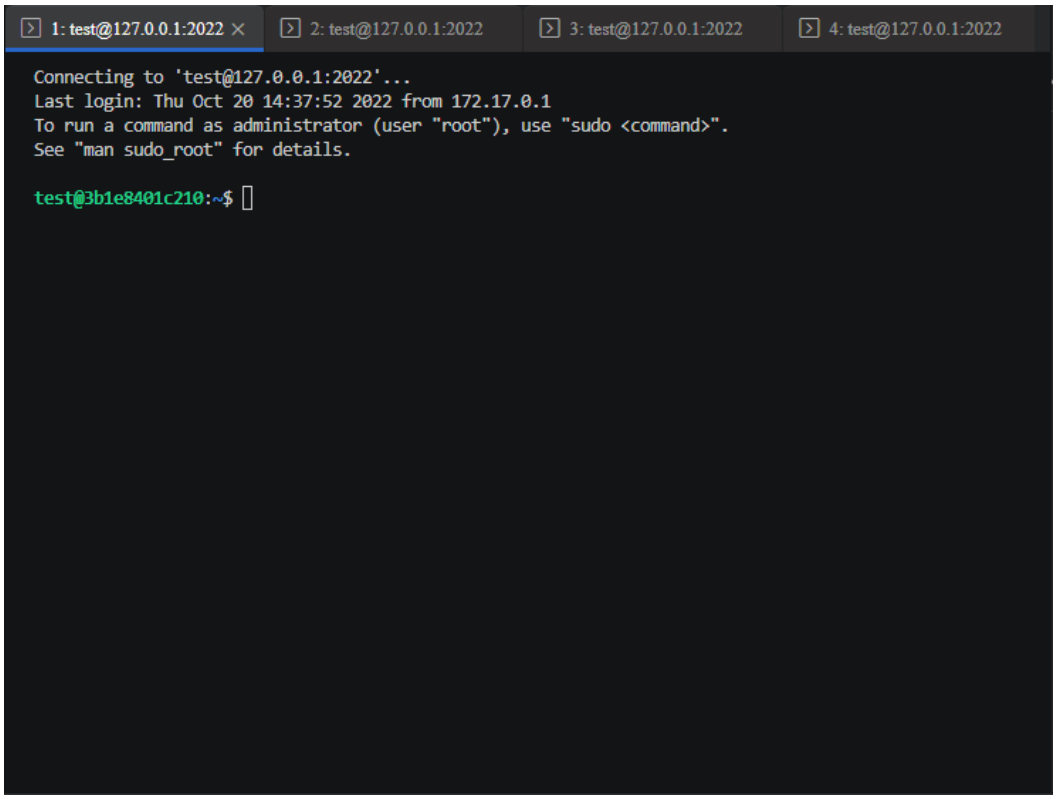
组织编辑器区域布局

为了便于使用几个打开的选项卡，您可以重新组织“**编辑器**”区域布局，以并排查看它们。

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。



----结束

7.6 管理文件

7.6.1 简介

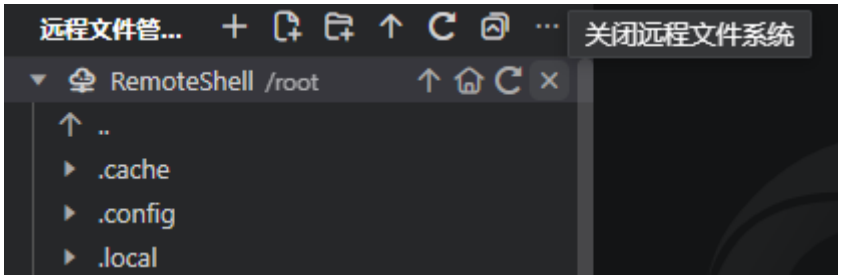
RemoteShell提供了对远程主机文件系统的访问，并支持各种常规文件操作：您可以上传和下载文件，在主机之间传输文件，打开文件进行编辑，重命名或删除文件等。

7.6.2 打开远程主机的文件系统

当您与主机建立连接时，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，单击所需连接的记录旁边的**“打开远程文件管理器”**按钮（ ）。

- 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（ ）。



7.6.3 设置根文件夹

为了方便浏览远程主机文件系统，您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。

步骤1 在**“远程文件管理器”**区域中，右键单击要设置为根的文件夹。

步骤2 在上下文菜单中，选择**“设置为根文件夹”**。

----结束

7.6.4 上传文件和文件夹

RemoteShell允许您将文件和文件夹从本地计算机上传到远程主机。

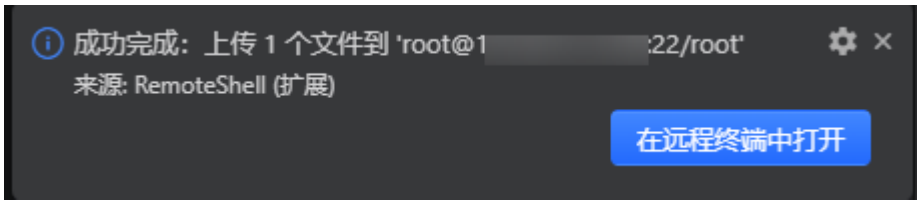
步骤1 在“**远程文件管理器**”区域中，右键单击要上传的文件或文件夹的目标文件夹。

步骤2 在上下文菜单中，选择“**上传文件**”或“**上传文件夹**”。

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。

----结束

上传完成后，RemoteShell在**“通知”**区域中显示相应的通知。



7.6.5 下载文件和文件夹

RemoteShell允许您将文件和文件夹从远程主机下载到本地计算机。

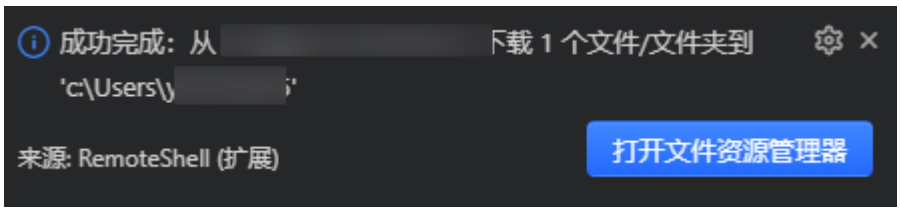
步骤1 在**“远程文件管理器”**区域中，右键单击要下载的文件或文件夹。

步骤2 在上下文菜单中，选择**“下载”**。

步骤3 在打开的文件选择器对话框中，选择下载位置。

----结束

下载完成后，RemoteShell在**“通知”**区域中显示相应的通知。



7.6.6 新建和编辑文件和文件夹

RemoteShell允许您对文件和文件夹执行所有常规操作。

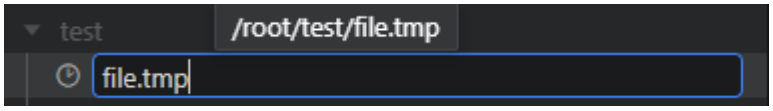
创建文件/文件夹

步骤1 在**“远程文件管理器”**区域中，选择要新建的文件/文件夹的目标文件夹。

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**。

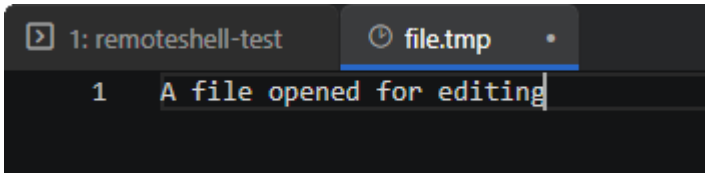
步骤3 在出现的输入框中，提供要新建的文件/文件夹的名称，按**Enter**确认。



----结束

编辑文件

步骤1 在**“远程文件管理器”**区域中，双击要编辑的文件。或者，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。该文件将在**“编辑器”**区域的单独选项卡中打开。



----结束

要保存文件，请在主菜单中选择“文件>保存”（全模式）或“”（精简模式）或按**Ctrl+S**。如果您编辑了多个文件，您可以通过选择“文件>全部保存”（全模式）或**“远程连接>全部保存**（精简模式）或按**Ctrl+K S**来保存它们。

7.6.7 复制和移动文件和文件夹

您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

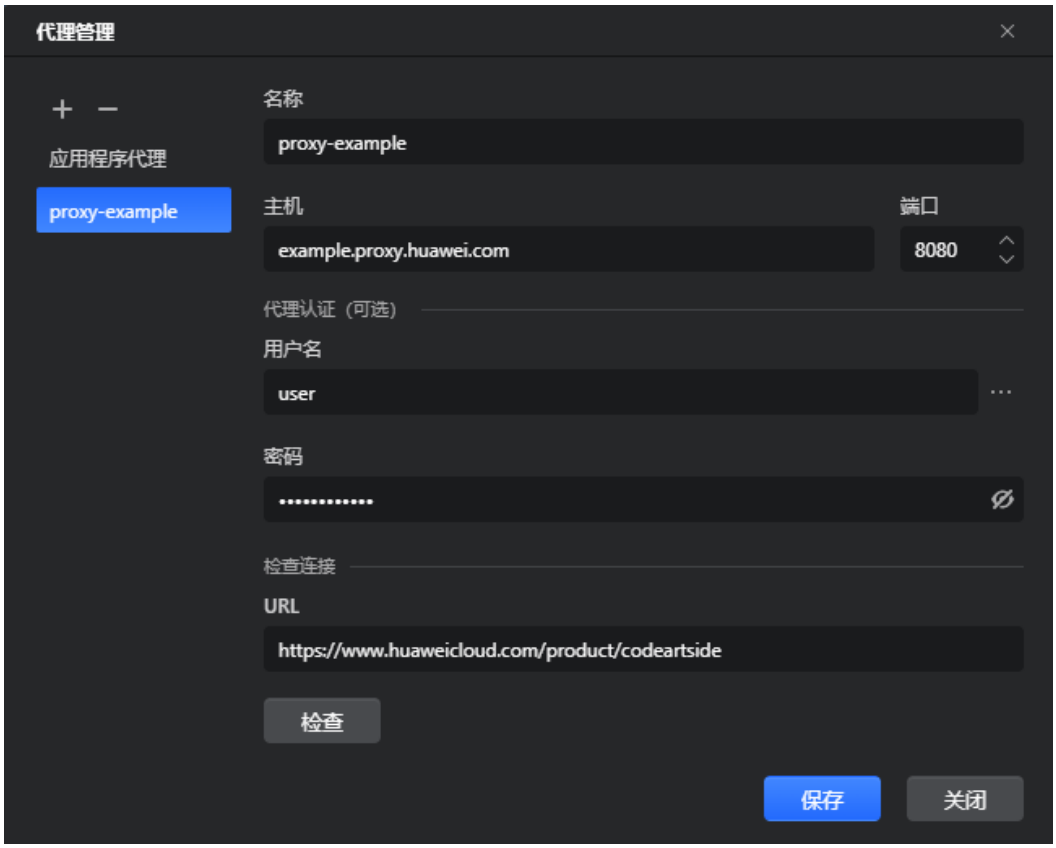
7.7 配置代理

RemoteShell支持您通过代理连接到主机。您可以配置多个代理，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”支持您更加便捷地管理代理配置。

步骤1 请在**“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**“打开代理管理”**。或者在主菜单中，选择“RemoteShell>代理管理”（全模式）或“ > 远程连接 > >代理管理”（精简模式）。

步骤2 在打开的“代理管理”窗口中，在左侧列表中选择现有代理，或单击**“新增代理”**按钮（ ＋ ）以添加新代理。要修改CodeArts IDE应用程序代理的连接参数，请选择**“应用程序代理”**。

步骤3 提供代理连接地址和身份验证参数。



要验证提供的参数并确保代理可访问，在“**检查连接**”区域的“**URL**”字段中提供任意Web地址，然后单击“**检查**”。

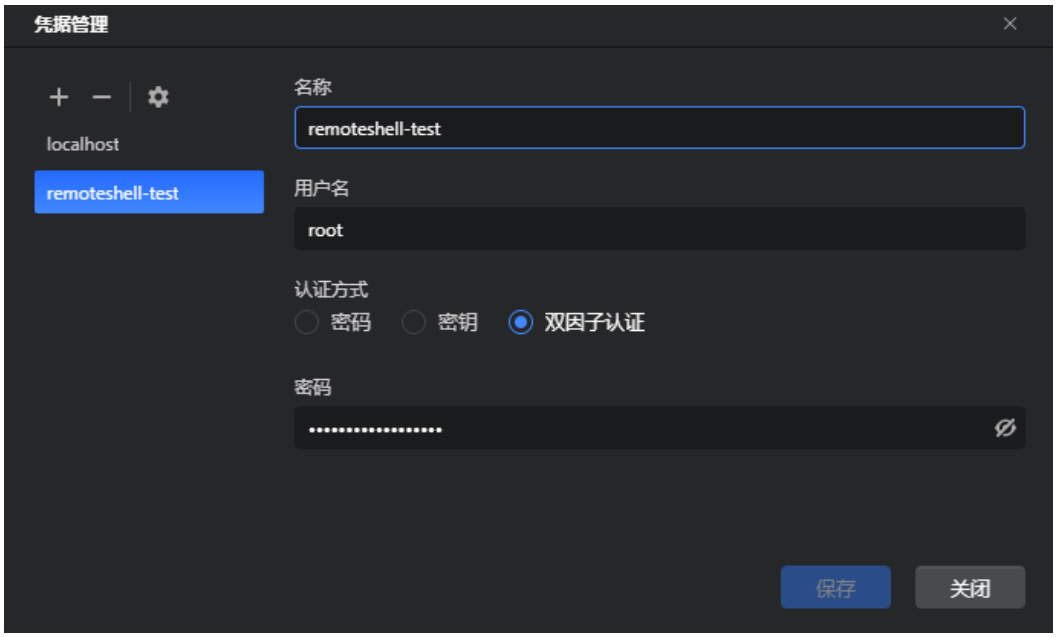
步骤4 单击“**保存**”以应用更改。

----结束

7.8 管理凭据

RemoteShell提供“凭据管理”，可让您轻松管理所有提供的凭据。要打开“凭据管理”，请在“**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**”。

“凭据管理”窗口展示了所有存在的凭据记录。



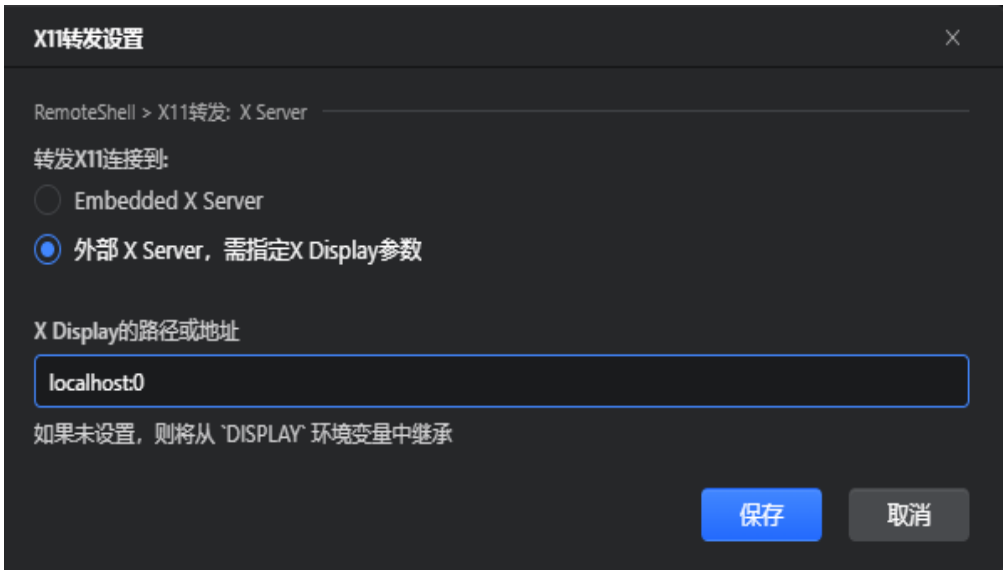
- 要添加新的凭据记录，请在“凭据管理”的工具栏上单击**“新增凭据”**按钮（ ）。
- 要删除凭据记录，请在“凭据管理”的工具栏上单击**“移除凭据”**按钮（ ）。
- 要配置凭据管理，请在“凭据管理”的工具栏上单击设置按钮（ ）。在打开的弹出窗口中，选择控制是否在“凭据管理”中自动保存成功使用后的新的连接和代理服务器的凭据。

7.9 与 X Server 一起使用

RemoteShell提供X11转发功能，让您可以通过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。

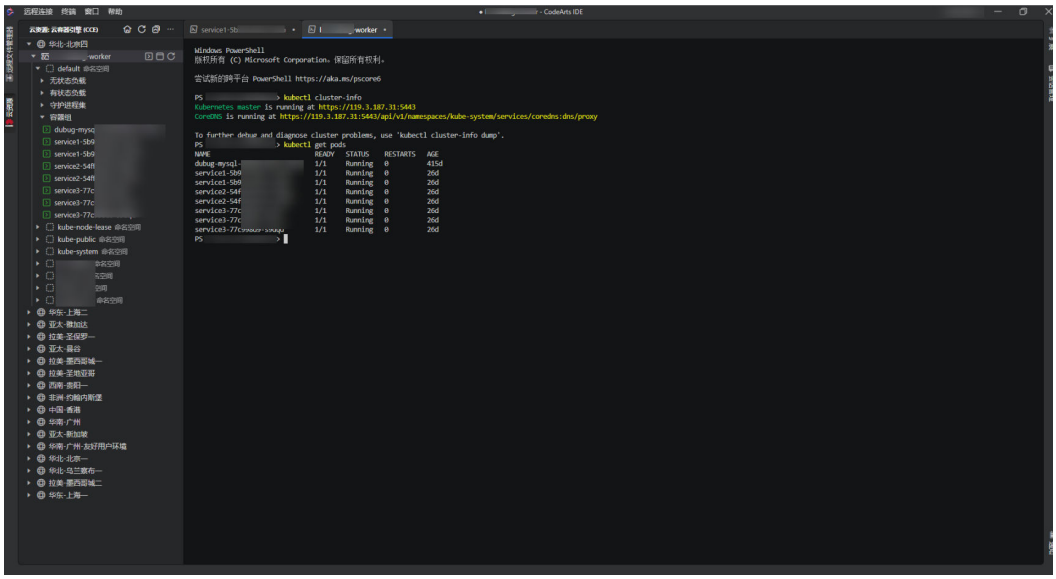
在**“会话管理”**区域中，单击**“X Server”**按钮（ ）并从弹出列表中选择所需的选项。

- 使用**“启动X Server”**/**“停止X Server”**”命令来启动/停止内置X Server。如果您想要使用第三方X Server来使用X11转发，其与内置X Server会有冲突，您可能需要停止内置X Server。
- 使用**“配置 X11”**命令可以配置X11转发。如果您选择将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。



7.10 访问 Kubernetes 集群

“云资源”区域的**“云容器引擎(CCE)”**部分列出了与您的华为云账号关联的Kubernetes集群。



约束与限制

仅支持v1.15，v1.17，v1.19，v1.21，v1.23，v1.25版本的CCE集群和CCE Turbo集群。

请在连接前检查与集群之间的网络连通。可以使用弹性公网IP(EIP)或者云专线(DC)打通网络。

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。

步骤2 单击“**打开Kubectl命令行**”按钮（ ）。

----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

8 插件市场

8.1 简介

CodeArts IDE具备内置插件市场，提供了丰富的扩展能力，您可以通过内置插件市场安装、使用并管理您的插件。

CodeArts IDE还为插件开发者提供了插件创建、开发调试以及打包发布的完整能力。

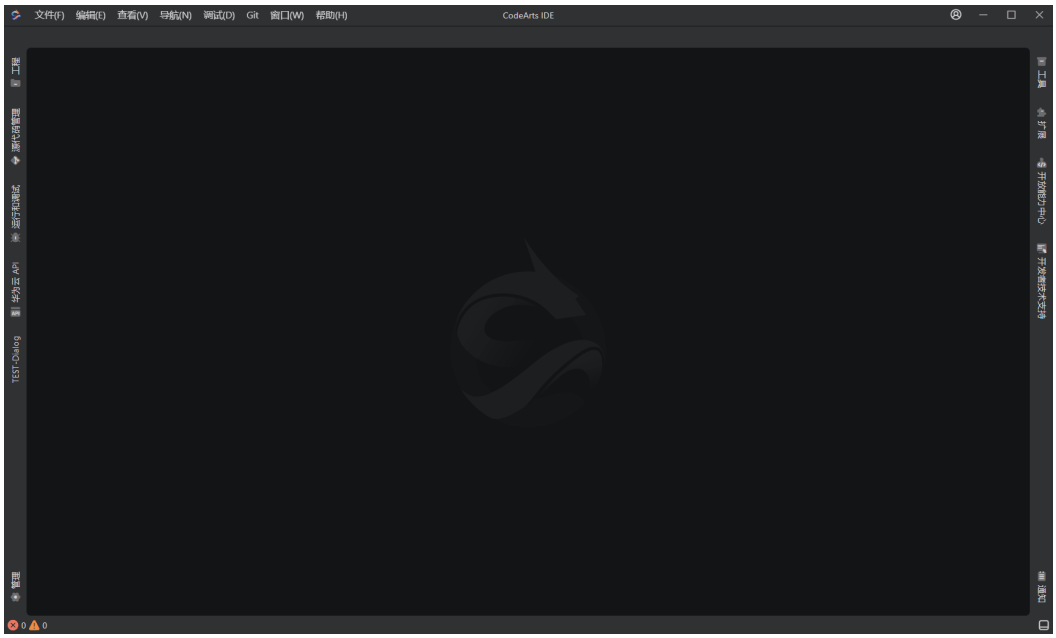
8.2 插件开发

安装环境

在开始插件开发之前，请检查是否已安装 Node.js 16.10.0 或以上版本 (**https://nodejs.org/en/**)。若已安装，可在本地CMD或CodeArts IDE终端使用命令行 node -v 以及 npm -v 查看相应的安装版本。

插件创建

步骤1 打开 CodeArts IDE，单击菜单“文件 ->新建 ->工程...”，选择“扩展”。



- 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”以及“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：发布商必须为插件市场中已创建的发布商，否则将无法在插件市场上发布插件，也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”，等待插件工程创建完成，选择是否在当前窗口打开新建的插件工程。

----结束

插件调试与运行

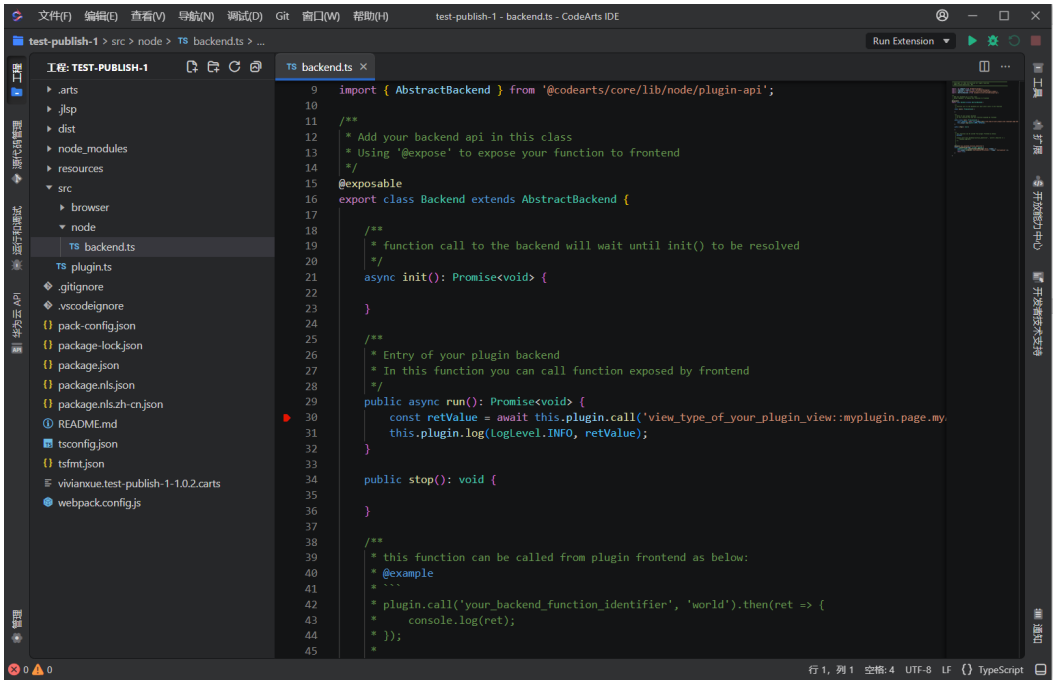
后端调试

在插件的 src/node/ 目录下存放的是插件的后端代码，后端代码运行在 nodejs 环境中，插件工程在创建的时候已经默认生成了一个后端文件 backend.ts，对于轻量级的插件，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。

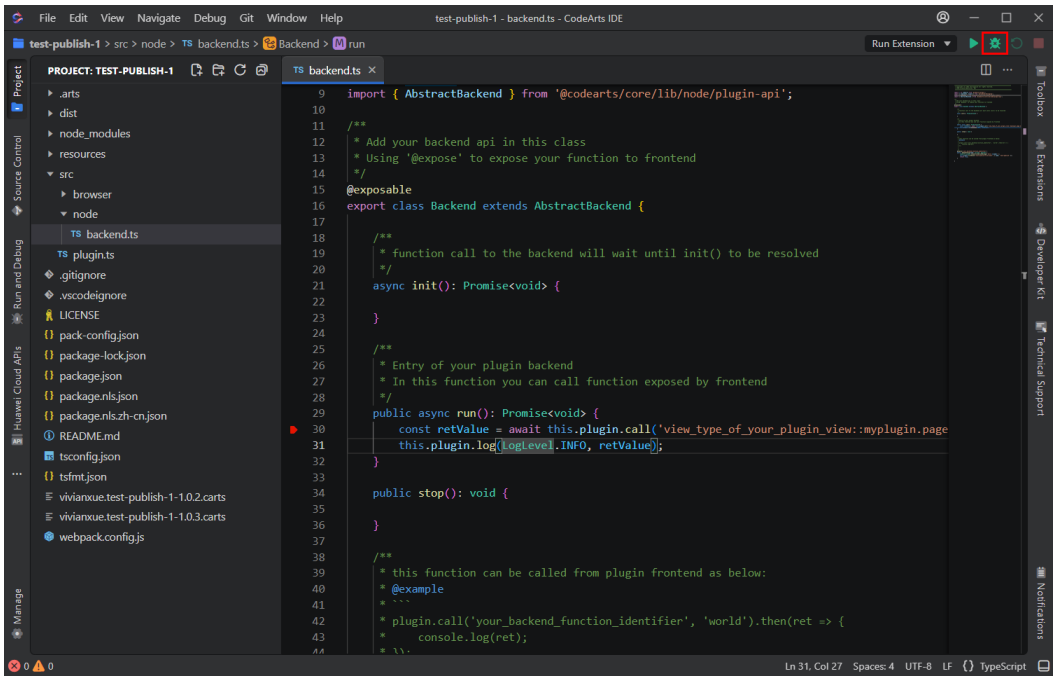
这里我们简单介绍下init，run和stop方法：

- init 函数：作为该后端实例的初始化方法，可以在插件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：将会在插件被停止前被调用，如有需要可以进行一些资源清理的操作。

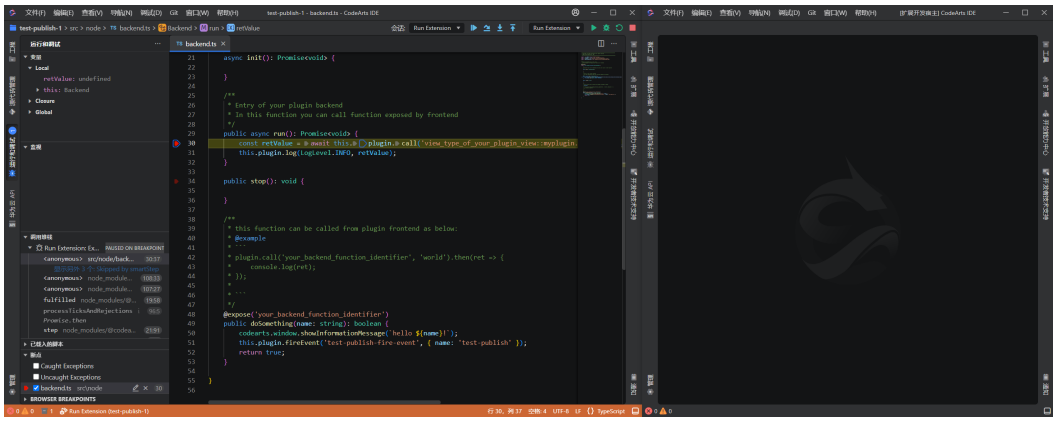
步骤1 添加断点：在backend.ts 的 run() 函数中添加一个断点。



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。



步骤3 进入断点，进行调试。



----结束

前端调试

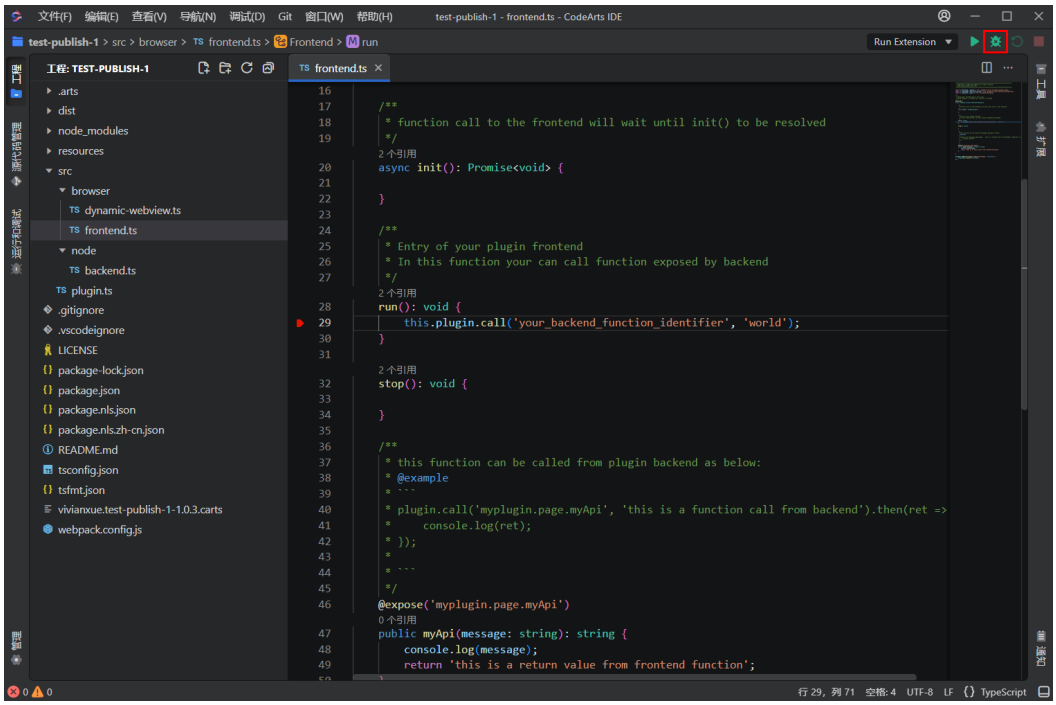
与插件的后端不同，前端的代码最终将被编译并运行于浏览器环境中，前端的代码存放于 src/browser 目录中，插件工程在创建的时候会默认生成两个前端源码文件 frontend.ts 和 dynamic-webview.ts。这两个文件的内容与后端 backend.ts 的结构非常相似，只不过运行的环境不同而已，这里就不再重复对这两个文件中 init()、run()、stop() 方法进行介绍。由于前端运行在浏览器环境中，代码调试将借助于浏览器自带的调试功能。如果需要自动重新编译前端代码，可以在终端中执行命令 npm run watch-browser，然后再运行调试。在启动调试后如果修改了代码，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。

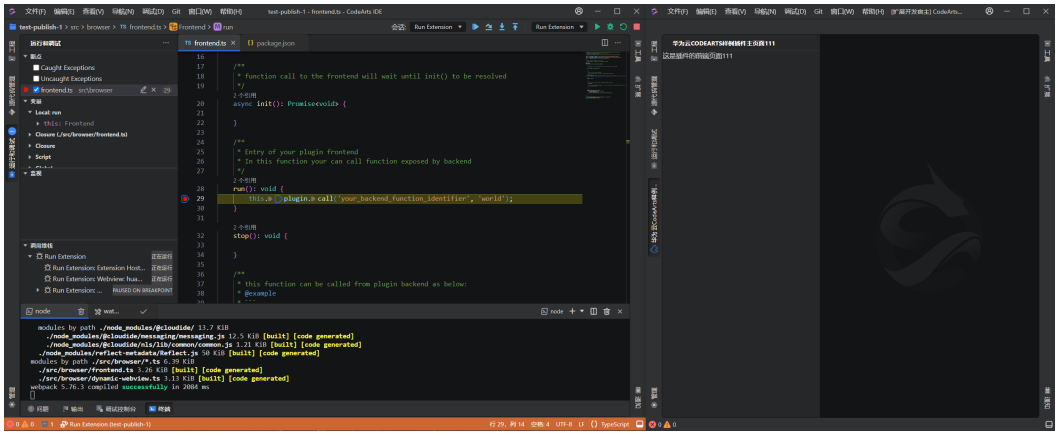

```
webpack.config.js
1   const path = require('path');
2
3   module.exports = {
4       entry: {
5           'page/dist/index': './src/browser/frontend.ts',
6           'page/dist/dynamic-webview-index': './src/browser/dynamic-webview.ts'
7       },
8       mode: 'development',
9       module: {
10          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。



步骤4 打开插件注册的视图，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。



----结束

前后端方法相互调用

后端调用前端

1. **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，我们自定义了一个 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';
   }
   ```

2. **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 AbstractBacend，需要实现 init()， run()， stop() 这三个方法，我们可以在 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 retValue = await
   this.plugin.call('view_type_of_your_plugin_view::myplugin.page.myApi', 'this is a function call from backend');
       this.plugin.log(LogLevel.INFO, retValue);
   }
   ```

前端调用后端

类似的，我们可以在后端定义自己的方法并将方法暴露给前端调用。

1. **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;
   }
   ```


2. **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

1. **在插件后端监听事件**

   打开 src/node 目录下的 backend.ts 文件，在 Backend 类的 run() 方法中我们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。

2. **在插件前端监听事件**

   打开 src/browser 下的 fronted.ts 文件，我们通过在 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，我们需要使用 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

插件创建完后，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插件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，如果插件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}!"
}
```

1. **内置成员plugin的localize方法**

   我们还在前后端内置的 plugin 成员变量中实现了 localize 方法。Frontend类 (src/browser/fronted.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');
```

2. **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');
   ```

3. **页面文件中的国际化方法**

   通用插件可以使用 ejs 和 pug 引擎来渲染界面，无论是 ejs 还是 pug 引擎插件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'] %>
   ```

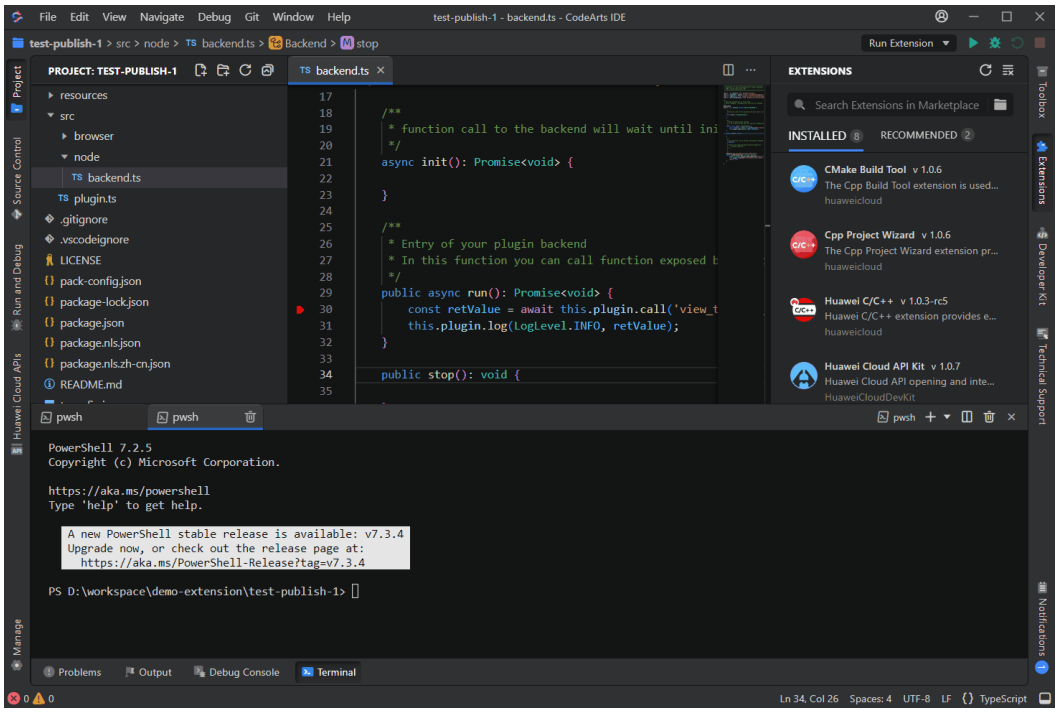
   对于使用 pug 引擎的开发者可以使用如下方式：

   ```
   #{l10n['plugin.hello']}
   ```

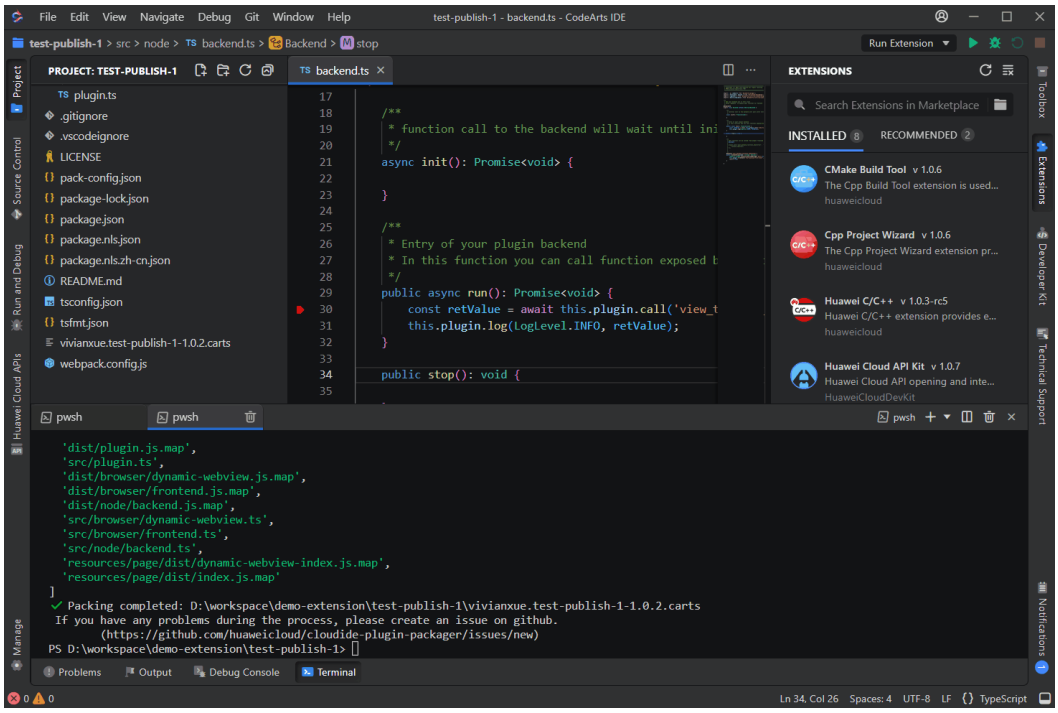
8.3 插件发布

插件打包安装

在终端执行命令“npm run package”可以对插件进行打包，打包文件后缀为“.carts”。



在 IDE 安装打包的插件后即可使用。



插件发布

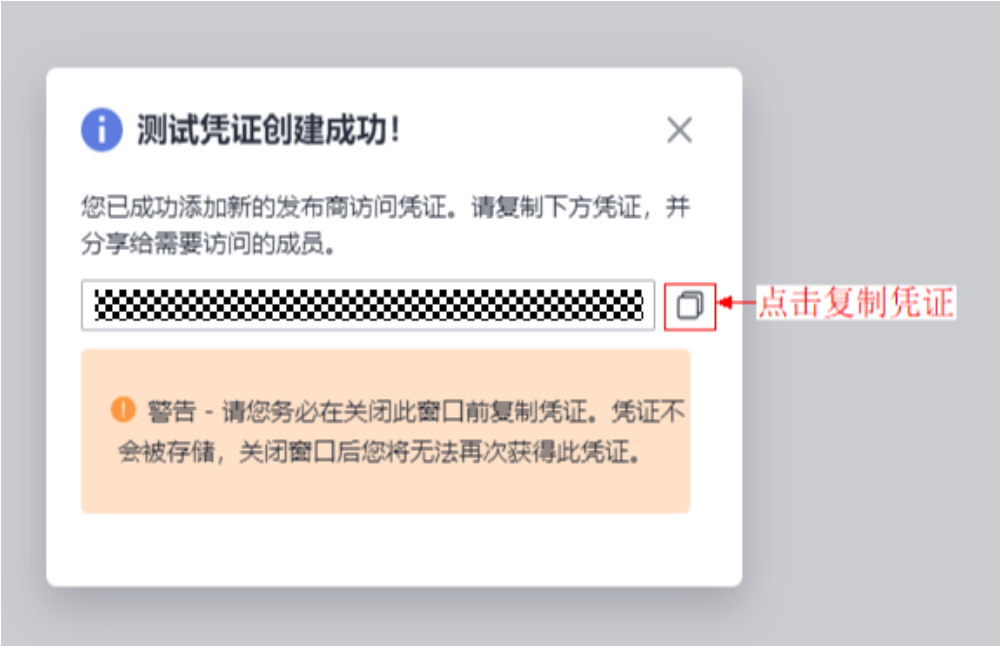
通过 IDE 直接发布到插件市场

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》**，前往插件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。

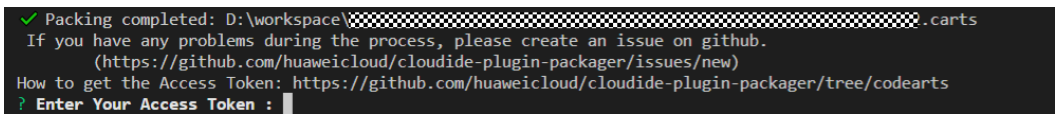
2. 输入凭证名称，并设置过期时间。

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。

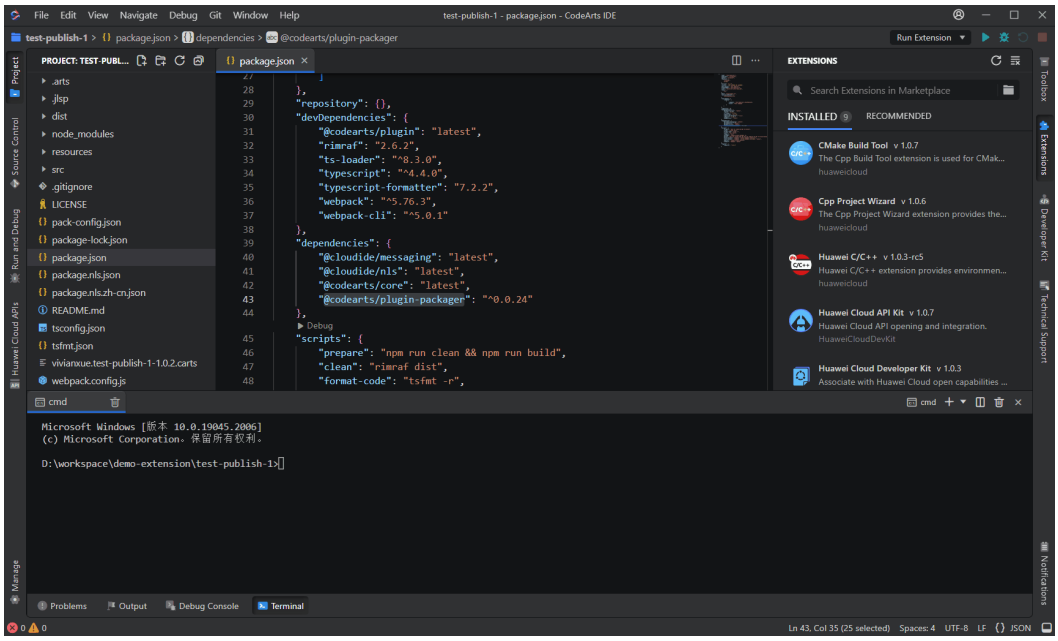


步骤2 在发布前，请确认 package.json 中的 publisher 与发布商凭证对应的发布商的唯一标识相符。

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。



步骤3 等待插件上传并发布。



步骤4 发布成功后，可以在插件市场的插件管理中看到已上传的插件。

----结束

通过插件市场进行发布

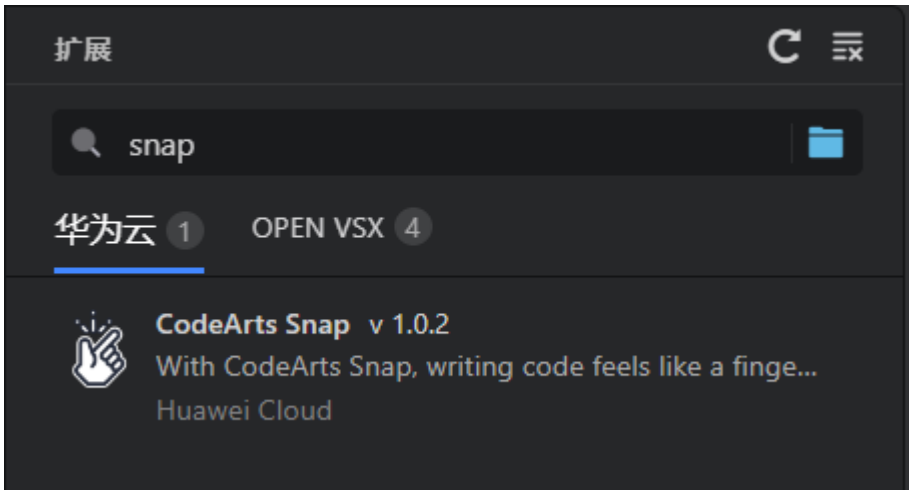
请参考《CodeArts IDE插件市场帮助文档》，按照步骤在插件市场上传和管理自己的插件。

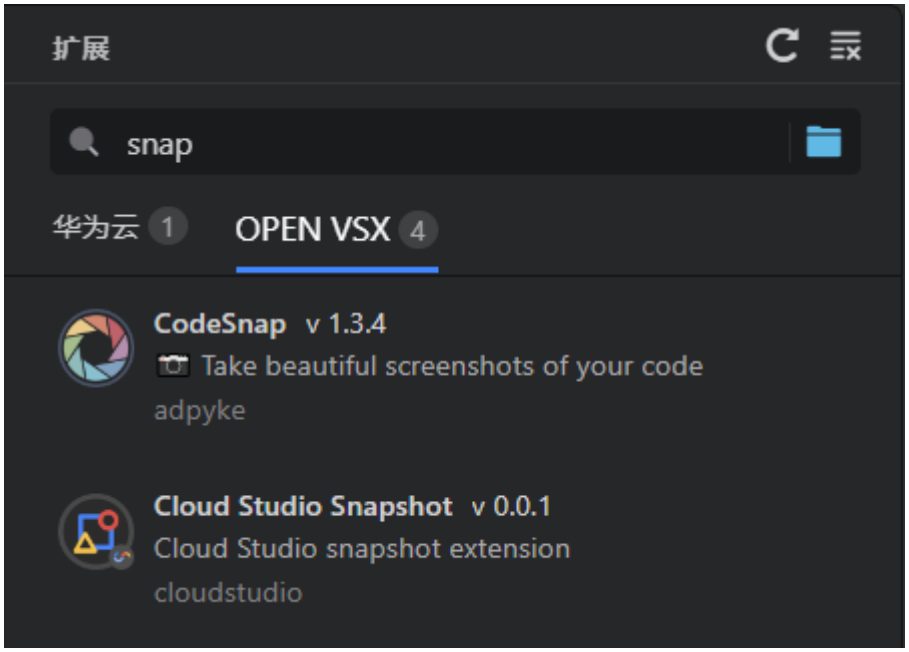
8.4 插件搜索、安装及使用

插件搜索

插件搜索

您可以通过 IDE 内置的插件市场搜索华为云插件市场及 OpenVSX 插件市场上的插件，单击搜索框下方的标签页可切换搜索平台。



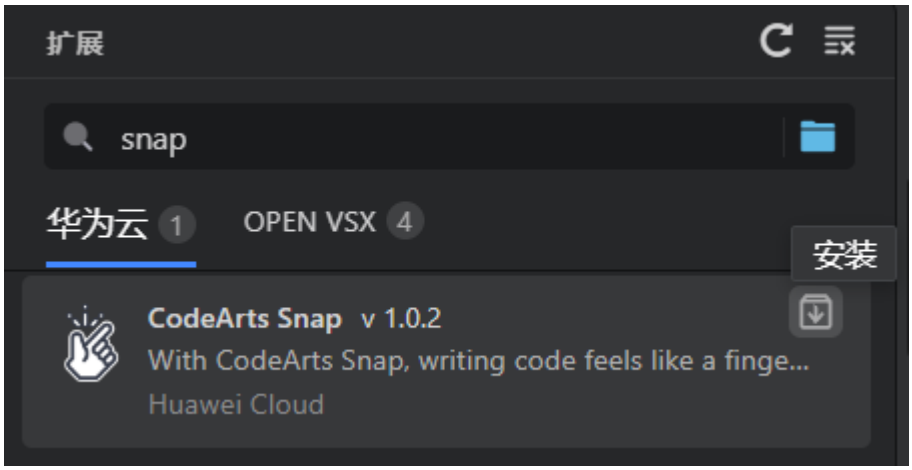


单击右上角的按钮可刷新页面，单击按钮可一键清空搜索框。

插件安装

通过插件市场安装

单击按钮可通过插件市场进行安装。



下载安装华为云平台的插件时，请先登录华为云。

本地插件安装

如果需要安装本地的插件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。

插件管理

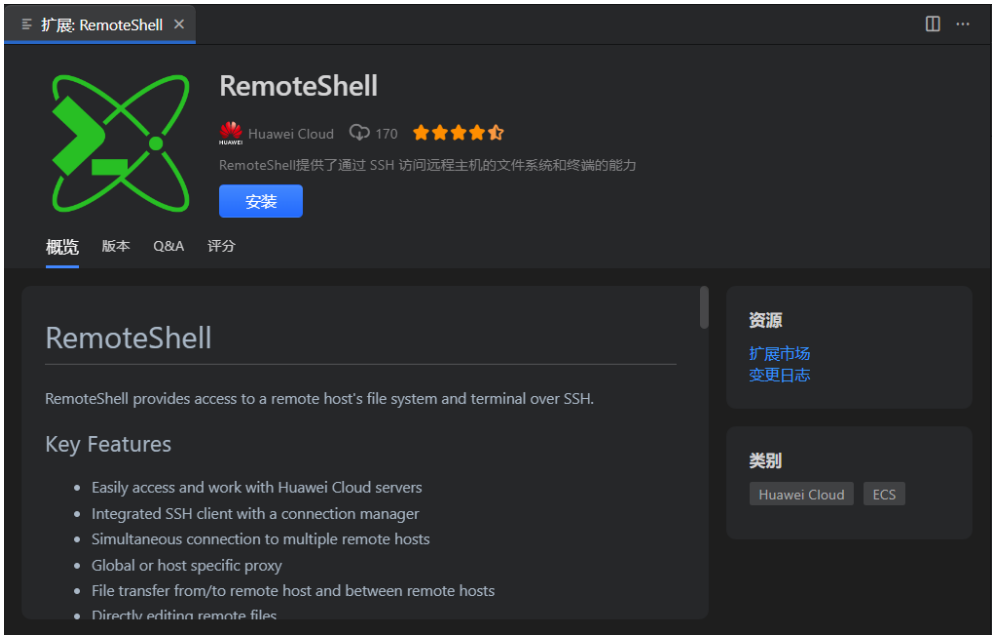
插件启用、禁用及卸载

已安装列表显示了所有当前已安装的插件，单击  按钮可以对插件进行启用、禁用或卸载操作。IDE 的内置插件无法启用、禁用或卸载。

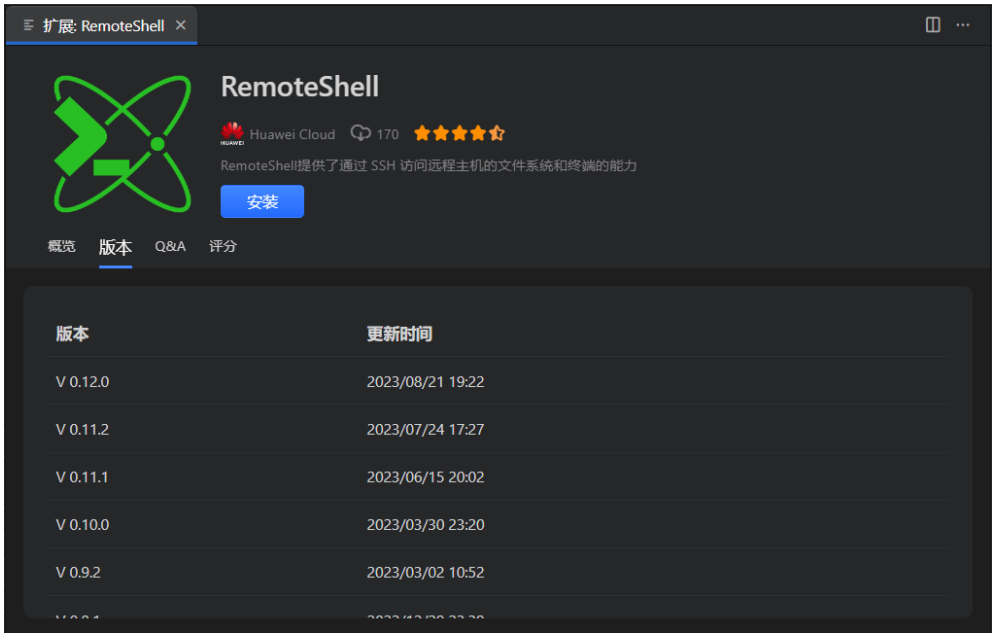
插件详情

单击插件可打开插件详情页，详情页中包含插件的概览、版本，对于已发布至插件市场上的插件，则还会包含Q&A、评分。

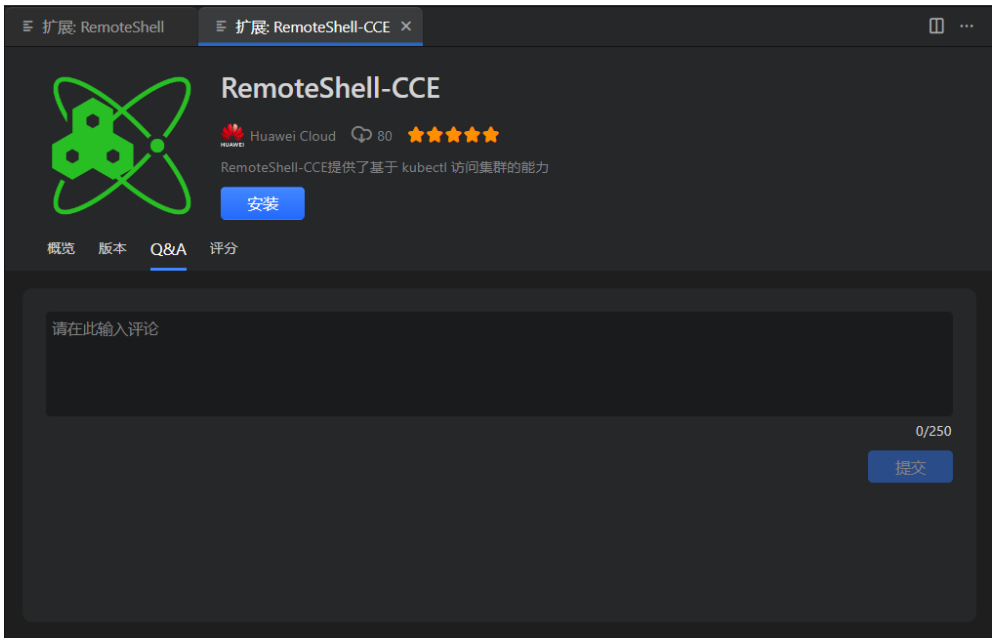
- 概览：包含插件的介绍，相关链接，类别和标签信息



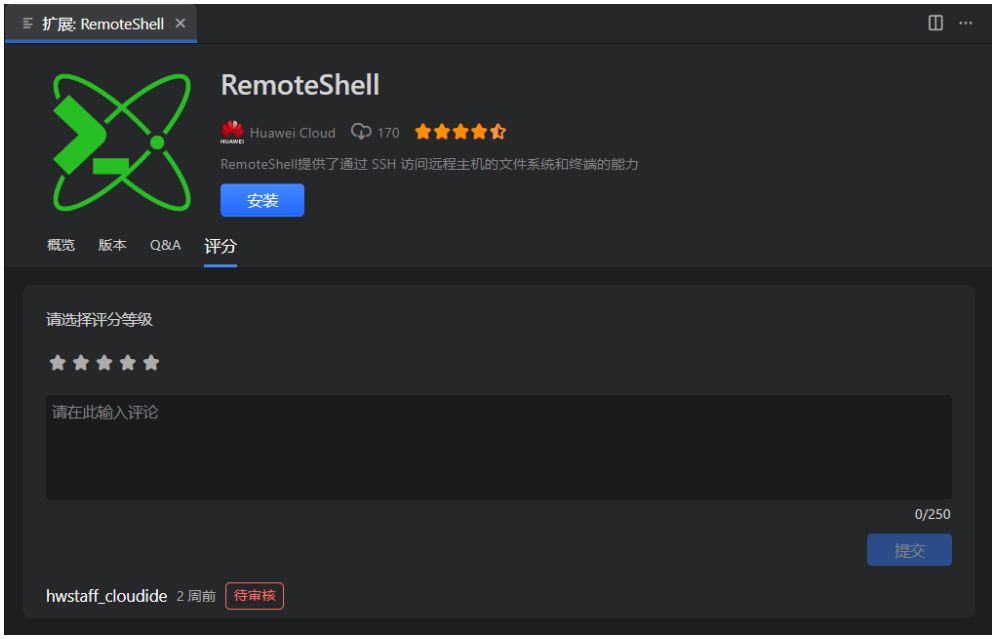
- 版本：插件发布的历史版本



- Q&A：华为云插件市场上此插件的Q&A，您在登录后，可以直接在 IDE 中发表评论或回复评论。OpenVSX插件暂不支持功能。



- 评分：华为云插件市场上此插件的用户评分，您在登录后，可以直接在 IDE 中发表评分及评价。OpenVSX插件暂不支持发表评分功能。



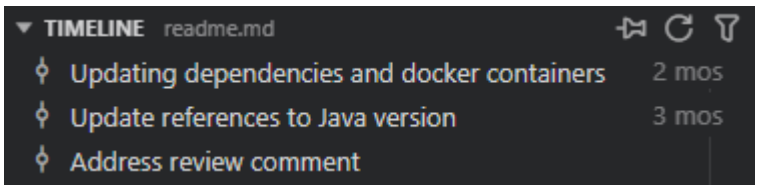
9 版本控制

9.1 简介

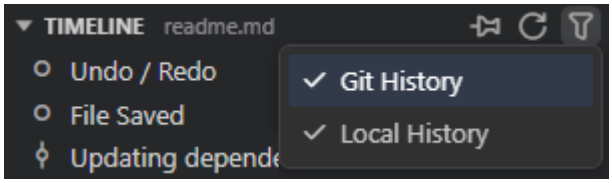
CodeArts IDE支持同时处理多个源代码管理（SCM）提供商，其中**Git**支持已经内置。CodeArts IDE还维护了**本地历史记录**，即使您的项目没有关联的SCM提供商，也可以跟踪文件的更改。

时间线视图

Timeline视图可以在默认情况下通过右侧活动栏中的工具箱（ ）访问，它提供了一个文件的时间序列事件的统一视图。在事件列表中，本地文件事件标记为 ，与版本控制系统相关的事件标记为 。



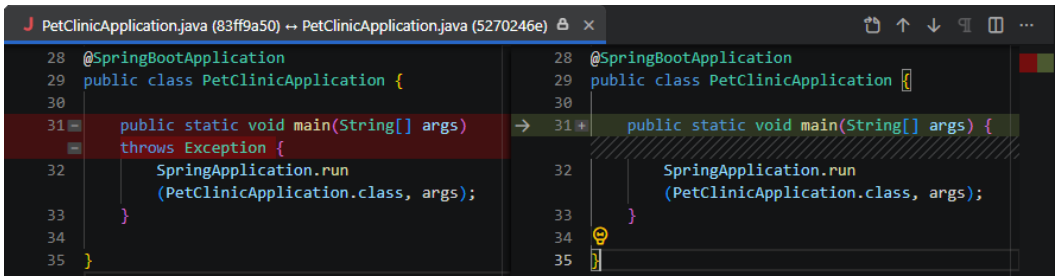
您可以过滤**Timeline**事件列表，只查看来自版本控制系统或本地历史记录的事件。在**Timeline**视图工具栏中，单击**Filter Timeline**按钮（ ）并选择所需的视图选项。



Timeline视图显示当前在代码编辑器中打开的文件的事件，并在您在编辑器选项卡之间切换时自动切换文件。要禁用自动切换并始终显示某个文件的事件，请切换到其编辑器选项卡，然后单击**Timeline**视图工具栏中的**Pin Timeline**按钮（ ）。

差异查看器

CodeArts IDE提供了一个内置的差异查看器。

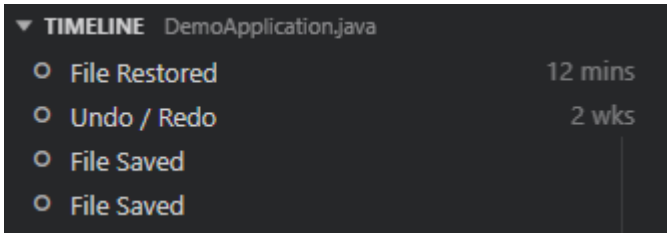


您可以通过首先在资源管理器或打开的编辑器列表中右键单击文件，然后选择**Select for Compare**，然后再右键单击要与之比较的第二个文件，并选择**Compare with**‘**file_name_you_chose**’”来比较任意两个文件。或者，按下“Ctrl+Shift+P”/“Ctrl Ctrl”，然后选择**File: Compare Active File With**，您将看到最近文件的列表。差异的默认视图是并排视图。通过单击右上角的更多操作（ ）按钮，然后选择**Toggle Inline View**来切换到内联视图。如果您喜欢内联视图，可以将“diffEditor.renderSideBySide”设置为false。

9.2 本地历史

使用本地历史记录，CodeArts IDE可以记录文件上执行的事件历史。您可以查看和恢复文件在任何事件发生时的内容。

要访问文件的本地历史记录，请在代码编辑器中打开文件，然后单击右侧活动栏中的**工具箱**（ ）并展开**Timeline**视图。每次保存文件时，都会向**Timeline**事件列表中添加一个新记录。



查看事件之间的更改

您可以使用CodeArts IDE差异查看器来检查每个事件引入的更改。在**Timeline**视图中，右键单击一个事件，然后从上下文菜单中选择所需的操作。

- **Compare with File**（与文件比较）：将文件内容与文件的当前状态进行比较。
- **Compare with Previous**（与上一个比较）：比较此事件和上一个事件之间的文件内容。
- **Select for Compare**（选择比较）：选择要进行比较的事件。选择了一个事件后，右键单击要将文件内容与之比较的另一个事件，然后从上下文菜单中选择**Compare with Selected**。

查看和恢复文件内容

您可以查看特定事件时的文件内容，并将其重新应用于文件的当前状态。在**Timeline**视图中，右键单击一个事件，然后从上下文菜单中选择所需的操作。

- **Show contents**（显示内容）：在单独的只读编辑器选项卡中查看事件时的文件内容。

- **Restore contents**（恢复内容）：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。

9.3 GIT 支持

9.3.1 简介

约束与限制

虽然这个主题主要关注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**。

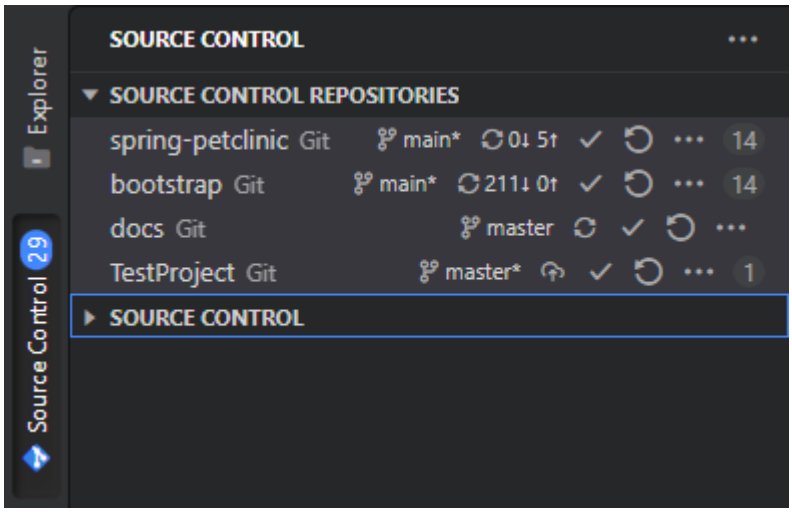
在使用源代码管理（SCM）功能之前，CodeArts IDE将利用您机器上的Git安装，因此您需要安装Git。请确保安装的是2.0.0版本或更高版本。

9.3.2 访问源代码控制功能

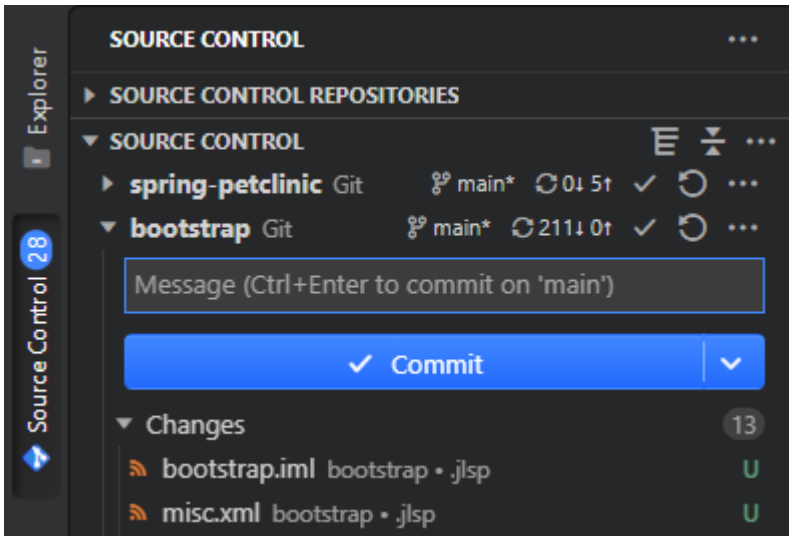
大多数与源代码管理相关的操作可以在**Source Control**视图中执行。要打开它，请执行以下任一操作：

- 单击左侧活动栏中的**Source Control**按钮（ ）。
- 在主菜单中，选择**View**>**Source Control**。
- 按“Ctrl+Shift+G”/“Alt+9”（IDEA键盘映射）。

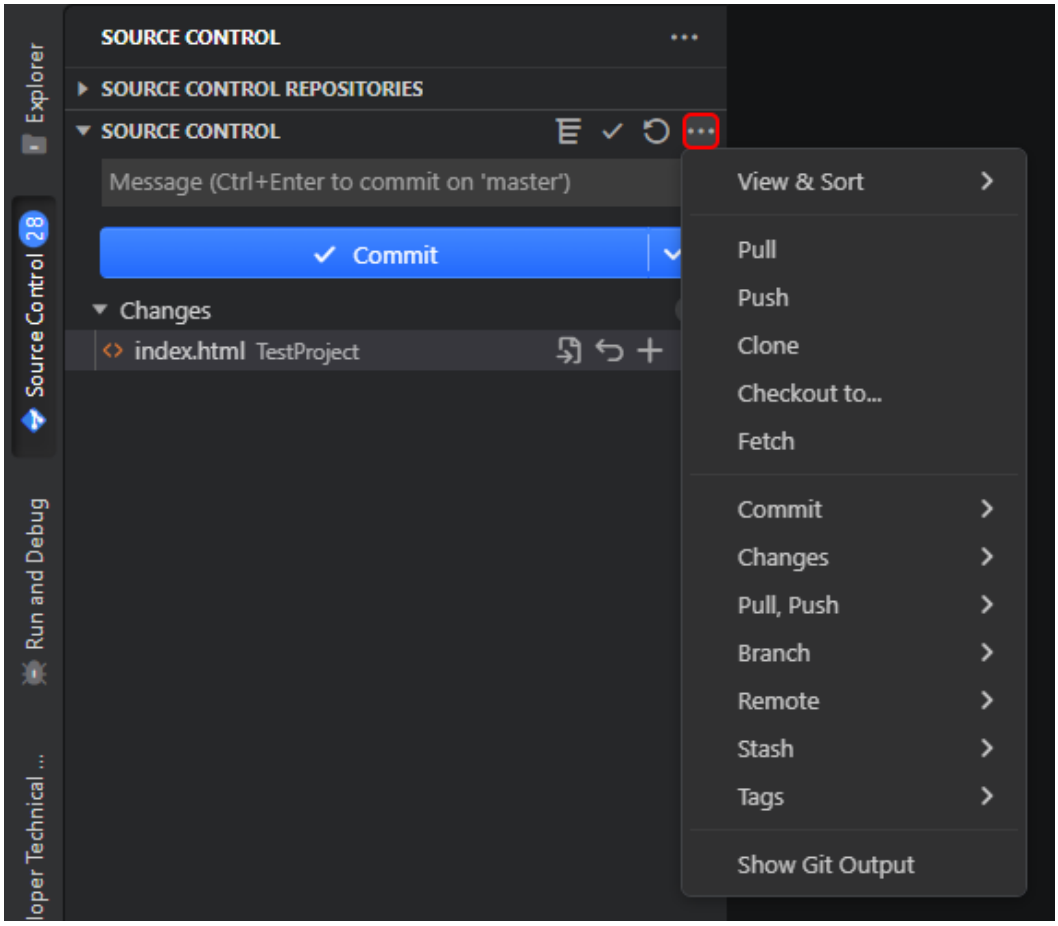
Source Control Repositories部分列出了当前工作区中检测到的存储库。



Source Control部分显示了打开的存储库中的更改。您可以通过选择特定的存储库来限定更改的显示。

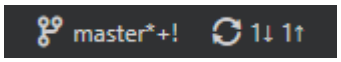


在源代码控制部分的**More Actions**（ ）菜单中，您可以快速访问大多数与**Source Control**相关的操作。



CodeArts IDE Git 状态栏操作

CodeArts IDE状态栏允许您运行**Synchronize Changes**命令，将远程更改拉取到本地仓库，然后将本地提交推送到上游分支。您还可以通过状态栏创建新分支并在分支之间切换。



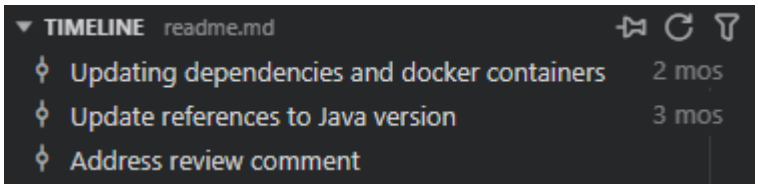
装订线指示器

当您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装订线指示。

时间线视图

Timeline视图可以在默认情况下通过**右侧活动栏**中的工具箱（ ）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。

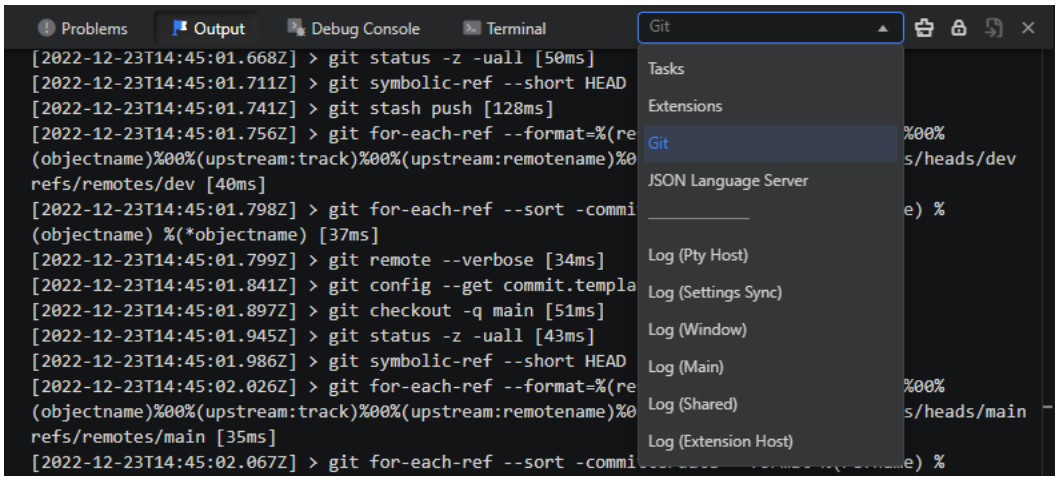


- 双击提交以打开引入该提交的更改的差异视图。
- 右键单击提交以获取复制提交ID和复制提交消息的选项。

查看 Git 输出

您始终可以查看当前执行的Git命令的详细信息。

要打开Git输出窗口，请运行**View**>**Output**或使用**Toggle Output**命令（“Ctrl+Shift+U”）。然后从列表中选择**Git**。

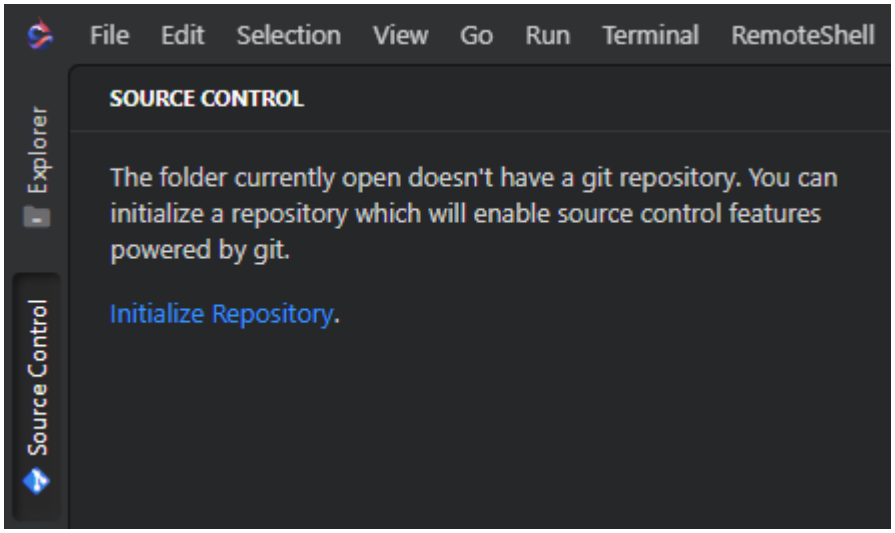


9.3.3 管理存储库

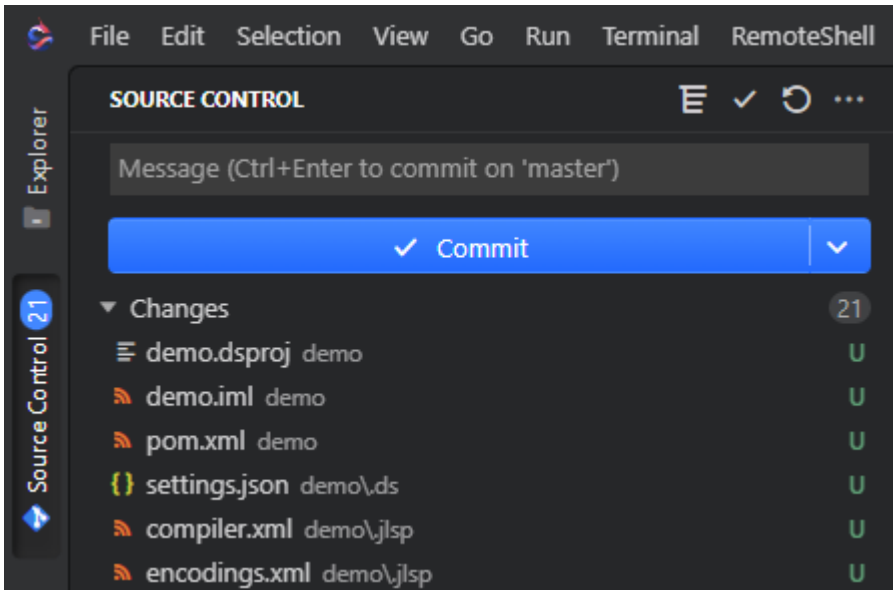
9.3.3.1 初始化存储库

当您打开一个本地文件夹时，可以通过在其中初始化一个Git存储库来启用Git源代码控制。

- 打开**Source Control**视图（“Ctrl+Shift+G”/“Alt+9”（IDEA键盘映射））。
- 单击**Initialize Repository**。

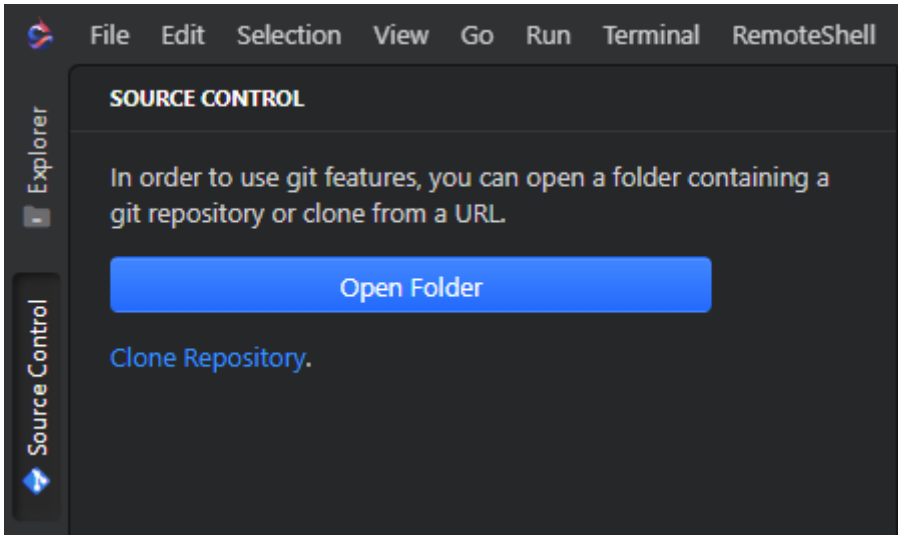


CodeArts IDE将创建必要的Git存储库元数据文件，并将工作区文件显示为未跟踪的更改，准备进行暂存。



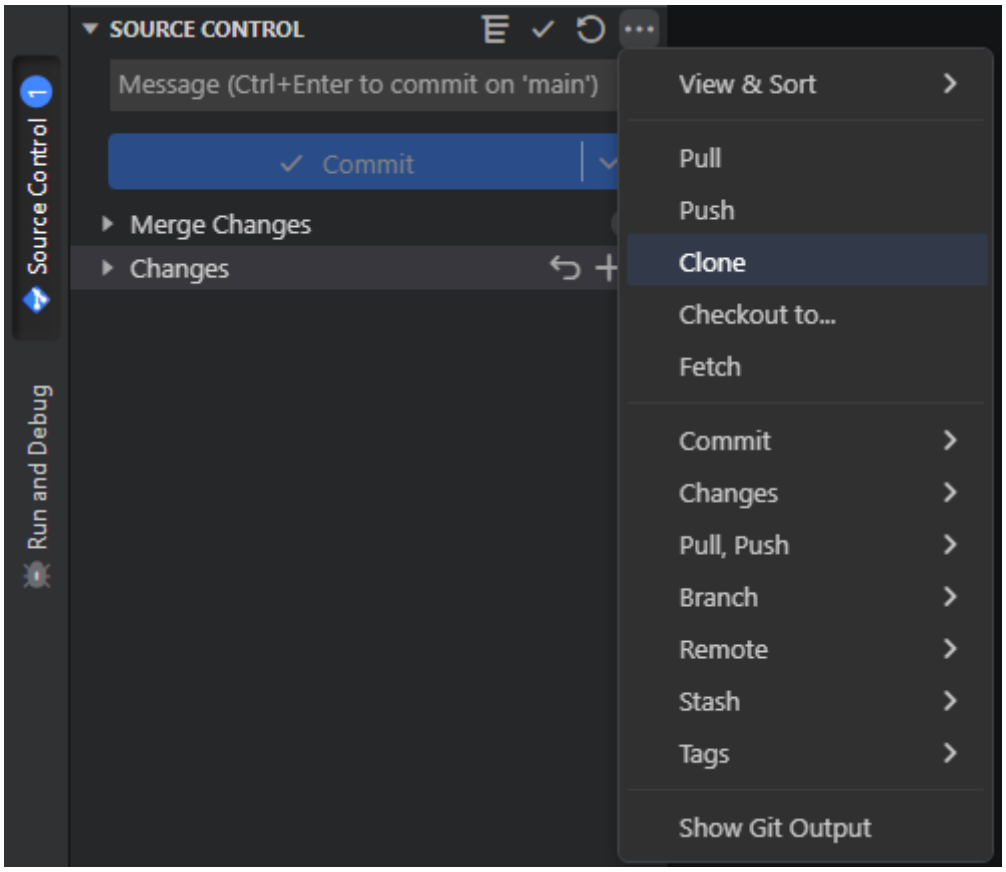
9.3.3.2 克隆现有存储库

如果尚未打开任何文件夹，则**Source Control**视图可让您打开本地文件夹或克隆存储库。

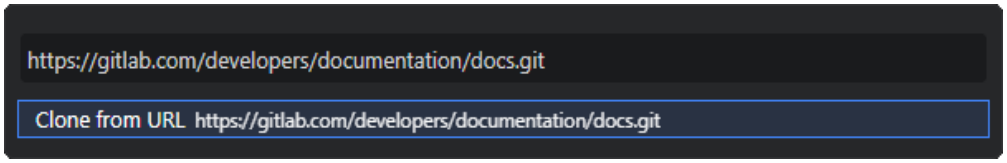


如果已经打开了某个文件夹，请按以下步骤克隆存储库。

1. 在**Source Control**视图中，展开Source Control Repositories部分。
2. 单击**More Actions**按钮（ ··· ）并选择**Clone**。

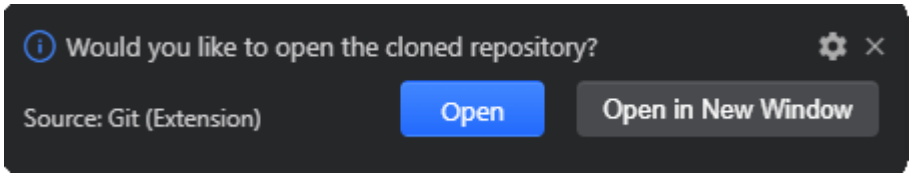


3. 在打开的弹出窗口中，提供远程存储库的URL并按Enter。



4. 在打开的对话框中，选择一个本地文件夹来存储克隆的存储库，然后单击选择**Select Repository Location**。

存储库克隆完成后，CodeArts IDE会提示您打开它。



9.3.3.3 管理远程仓库

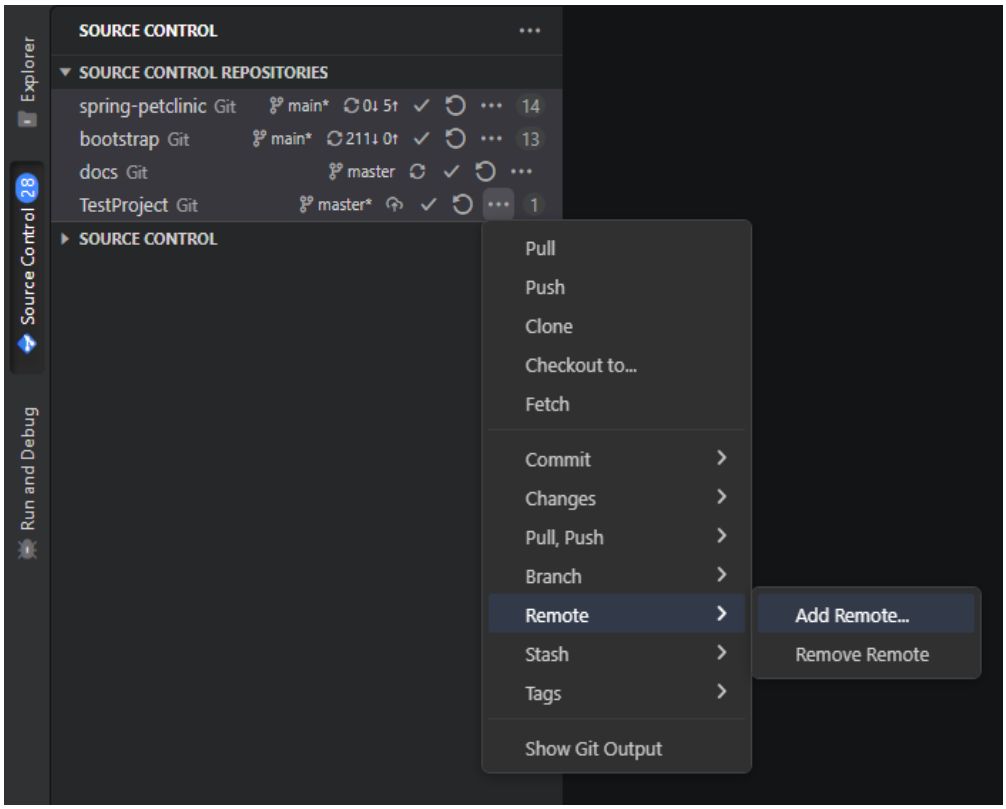
在创建本地Git仓库之后，您可以添加一个远程仓库，以便能够在Git项目上进行协作。如果您已经克隆了一个远程Git仓库，那么与原始克隆位置链接的远程仓库将自动配置，但您可以添加任意多个远程仓库。

约束与限制

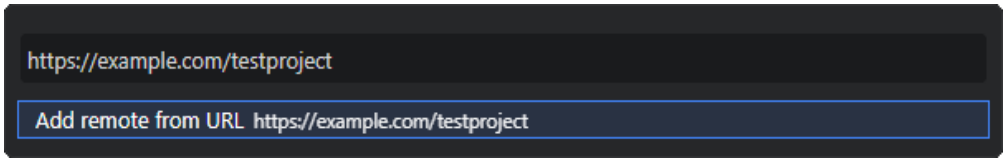
您应该设置一个**凭据助手**，以避免每次CodeArts IDE与Git远程仓库通信时都被要求输入凭据。否则，考虑通过git.autofetch设置禁用自动获取以减少身份验证提示的次数。

添加远程仓库

1. 添加远程仓库在您选择的SCM托管平台上创建一个空仓库。
2. 在**Source Control**视图中，展开**Source Control Repositories**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**Remote**>**Add Remote**。



4. 在打开的弹出窗口中，提供远程仓库的URL并按Enter键。



5. 在打开的弹出窗口中，提供远程仓库的名称并按Enter键。

移除远程仓库

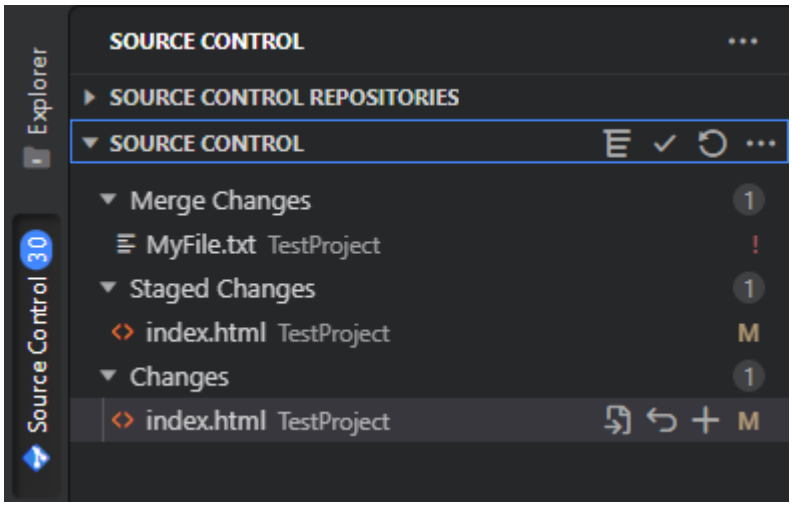
1. 在**Source Control**视图中，展开**Source Control Repositories**部分。
2. 单击要移除远程仓库的仓库旁边的**More Actions**按钮（···），然后选择**Remote**>**Remove Remote**。
3. 在打开的弹出窗口中，选择要移除的远程仓库并按Enter键。

9.3.4 管理版本控制下的文件

9.3.4.1 简介

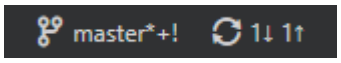
当您的项目与源代码管理（SCM）系统关联时，CodeArts IDE会跟踪项目文件中发生的所有更改。左侧**活动栏**中的**Source Control**按钮（ ）显示您当前在存储库中拥有的更改数量。

Source Control视图显示当前存储库更改的详细信息，分为**Changes**、**Staged Changes**和**Merge Changes**更改三个组。



单击每个项目将详细显示每个文件中的文本更改。请注意，对于未暂存的更改，右侧的编辑器仍然允许您编辑文件。

CodeArts IDE提供了存储库状态的指示：当前分支、脏状态指示以及当前分支中传入和传出提交的数量。您可以在**Source Control Repositories**部分的存储库记录旁边或CodeArts IDE状态栏中查看它们。

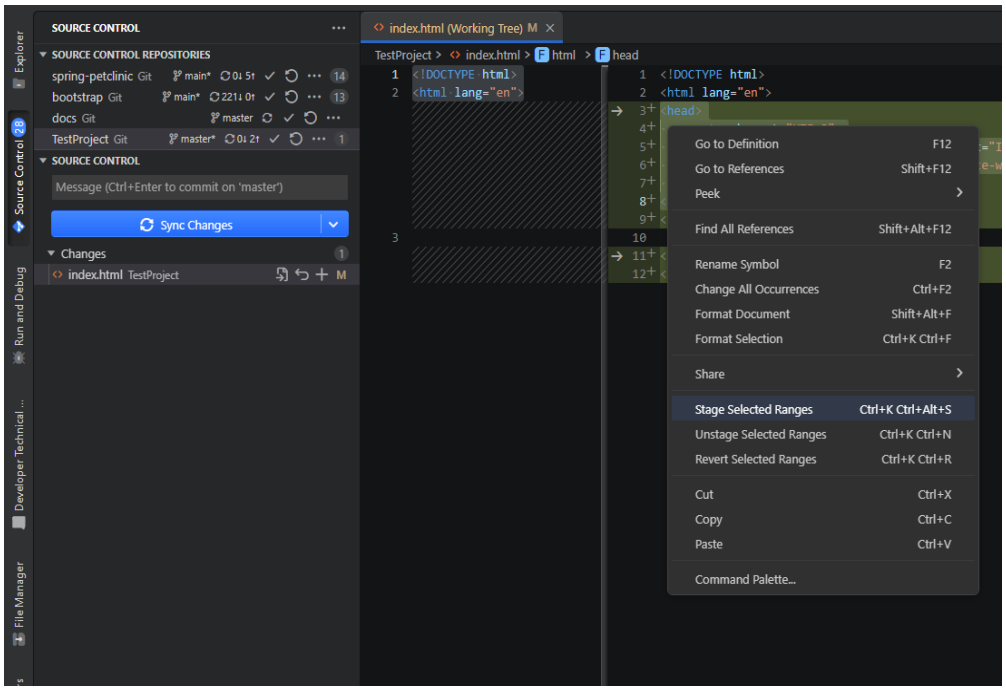


9.3.4.2 提交

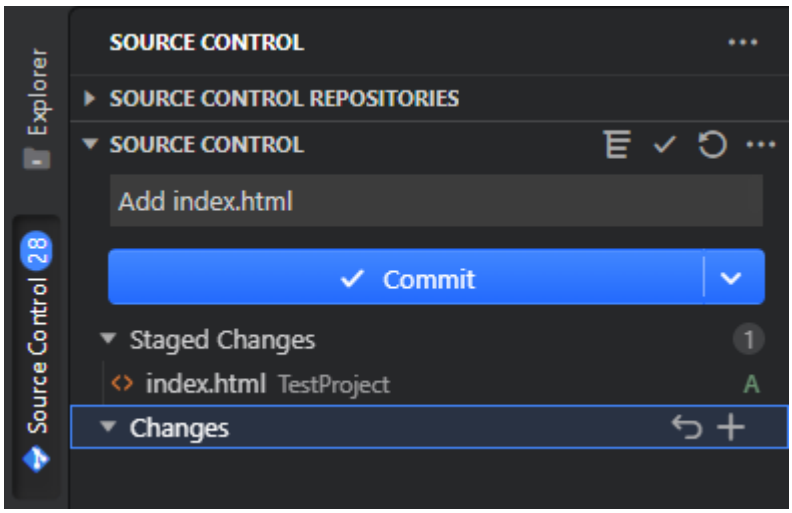
当您对代码进行一些更改时，您需要将它们提交到本地项目存储库，然后将它们**推送**到远程存储库，以便团队成员可以使用。

步骤1 通过将更改添加到暂存区来准备提交。要执行此操作，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，请单击**Stage Changes**按钮（ ），或右键单击文件并选择**Stage Changes**。
- 要暂存文件的一部分，请双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**Stage Selected Ranges**，或按“Ctrl+K”/“Ctrl+Alt+S”。



步骤2 在**Source Control**视图中，在字段中输入提交消息，然后单击**Commit**按钮或按**Ctrl+Enter**。



----结束

要撤消提交，请单击**More Actions**按钮（ ··· ）并选择**Undo Last Commit**。更改将重新添加到**Staged Changes**部分。

约束与限制

在提交之前，请确保您的Git配置中设置了用户名和/或电子邮件。否则，Git将使用本地计算机上的信息。您可以在**Git提交信息**中找到详细信息。

9.3.4.3 获取、拉取和推送更改

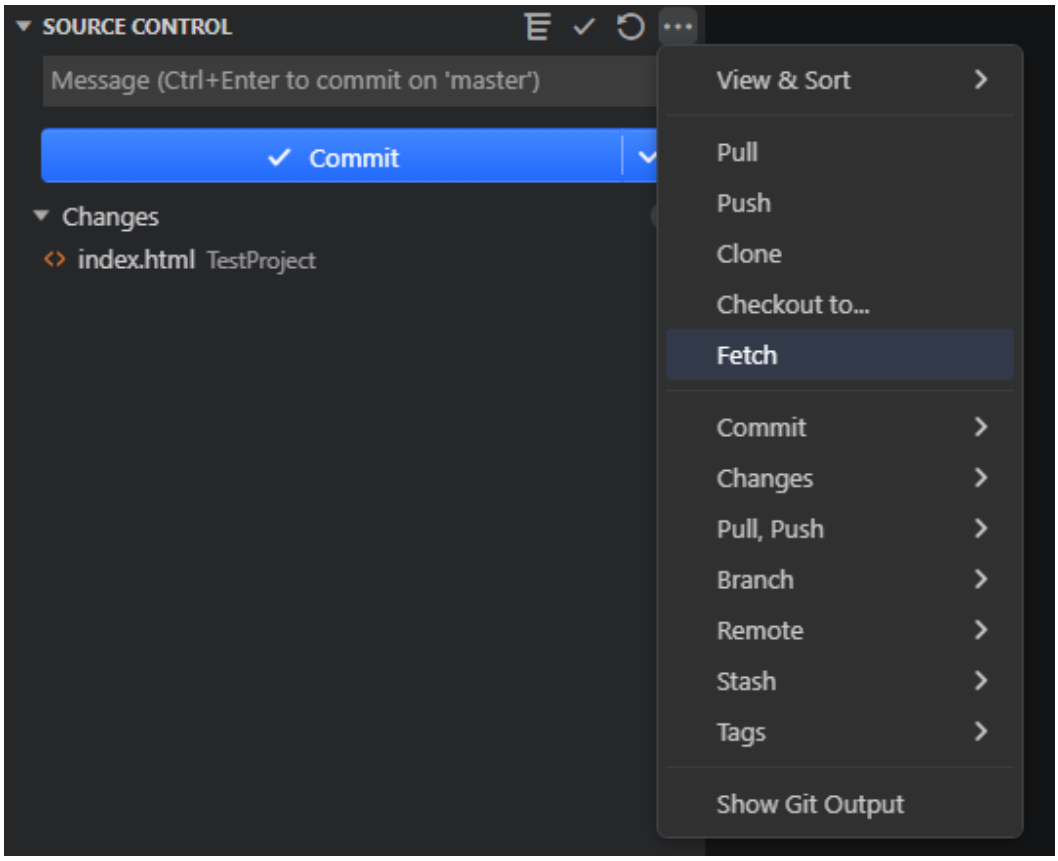
当您的存储库连接到远程，并且您的检出分支与远程的分支有**上游链接**时，CodeArts IDE允许您推送、拉取和同步（拉取后紧接着推送）该分支。

获取

从远程获取更改可以查看本地存储库相对于远程的超前或落后情况。这些更改本身不会合并到本地工作树中。CodeArts IDE可以执行自动定期获取。此功能默认禁用，但您可以通过git.autofetch设置启用它。

手动获取远程更改的方法如下：

- 在**Source Control**视图中，展开**Source Control Repositories**部分。
- 单击要获取更改的存储库旁边的**More Actions**按钮（···），然后选择**Fetch**。



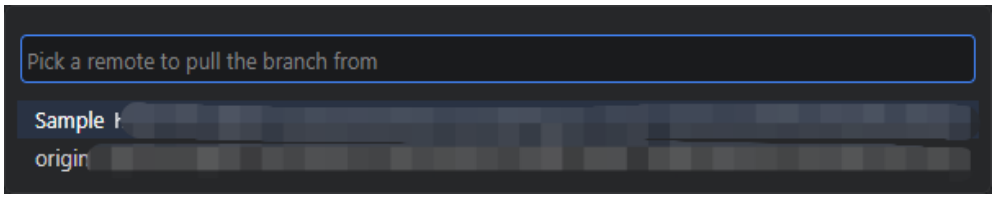
如果您配置了多个远程，可以通过选择**Pull**、**Push**>**Fetch All Remotes**.来从所有远程获取更改。如果您在源代码控制托管上删除了一个远程分支，它仍然会在CodeArts IDE中可见。要清理这样的孤立分支，请选择**Pull**、**Push**>**Fetch(Prune)**。

拉取

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**Source Control**视图中，展开**Source Control Repositories**部分。
2. 单击要将更改拉取到的存储库旁边的**More Actions**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**Pull**，或按“Ctrl+T”（IDEA键盘映射）。
   - 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**Push**，**Pull>Pull(Rebase)**。

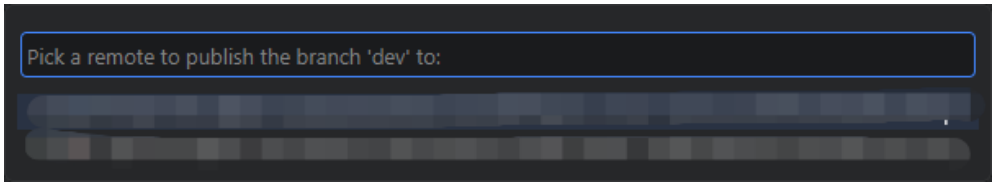
- 要从不同配置的远程存储库拉取更改，请选择**Push，Pull>Pull from**。然后在打开的弹出窗口中选择所需的远程存储库。



推送

在本地**提交**更改后，您需要运行推送命令将其上传到远程存储库。

1. 在**Source Control**视图中，展开**Source Control Repositories**部分。
2. 单击要推送更改的存储库旁边的**More Actions**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**Push**。
   - 要将更改推送到不同配置的远程存储库，请选择**Push，Pull>Push to**。然后在打开的弹出窗口中选择所需的远程存储库。



9.3.4.4 Stash 存储

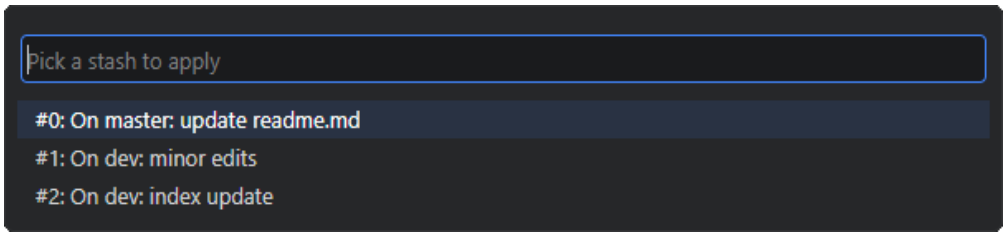
使用**stash**存储，您可以将当前更改移动到临时位置，而无需将其提交，从而将工作副本恢复到“干净”（即HEAD提交）状态。

要存储更改，请执行以下操作：

1. 在**Source Control**视图中，展开**Source Control Repositories**部分。
2. 单击要存储更改的存储库旁边的**More Actions**按钮（···），指向**Stash**，然后执行以下操作之一：
   - 要存储已暂存的更改，请选择**Stash**。
   - 要存储所有更改，包括未暂存和未版本化的文件，请选择**Stash（Include Untracked）**。

移动更改后，您可以随时重新应用它们到您的工作副本中。

1. 在**Source Control**视图中，展开**Source Control Repositories**部分。
2. 单击要重新应用更改的存储库旁边的**More Actions**按钮（···），指向**Stash**，然后执行以下操作之一：
   - 要应用最近的存储，请选择**Apply Latest Stash**。如果您还想从存储堆栈中删除已应用的存储，请选择**Apply Latest Stash**。
   - 要应用任意存储，请选择**Apply Stash**，然后在打开的弹出窗口中选择所需的存储。如果您还想从存储堆栈中删除已应用的存储，请选择**Pop Stash**。



您可以清理存储堆栈以删除不再需要的存储。

1. 在**Source Control**视图中，展开**Source Control Repositories**部分。
2. 单击要重新应用更改的存储库旁边的**More Actions**按钮（···），指向**Stash**，然后执行以下操作之一：
   - 要删除任意存储，请选择**Drop Stash**，并在打开的弹出窗口中选择所需的存储。
   - 要删除所有存储，请选择**Drop All Stashes**。

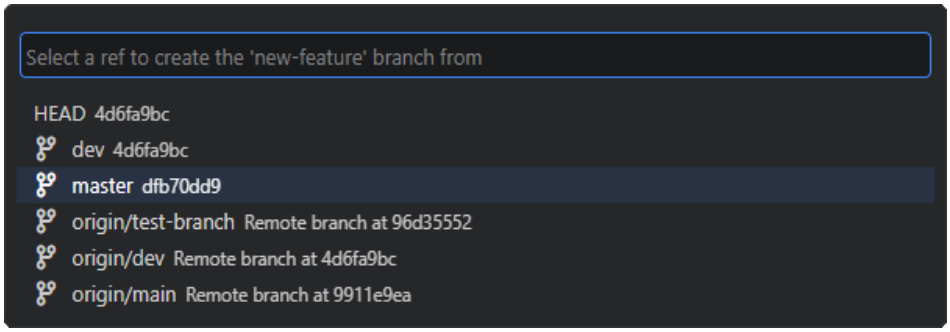
9.3.5 管理分支

9.3.5.1 创建/切换分支

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。

创建分支

1. 在**Source Control**视图中，展开**Source Control Repositories**部分。
2. 单击要在其中创建新分支的存储库旁边的**More Actions**按钮（···），指向**Branch**，然后执行以下操作之一：
   - 要从当前正在工作的分支创建新分支，请选择**Create Branch**，并在打开的弹出窗口中提供新分支的名称，然后按“Enter”键。
   - 要从存储库中的其他分支创建新分支，请选择**Create branch from**，并在打开的弹出窗口中选择源分支。

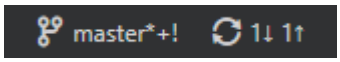
     

     然后在打开的弹出窗口中提供新分支的名称，然后按“Enter”键。

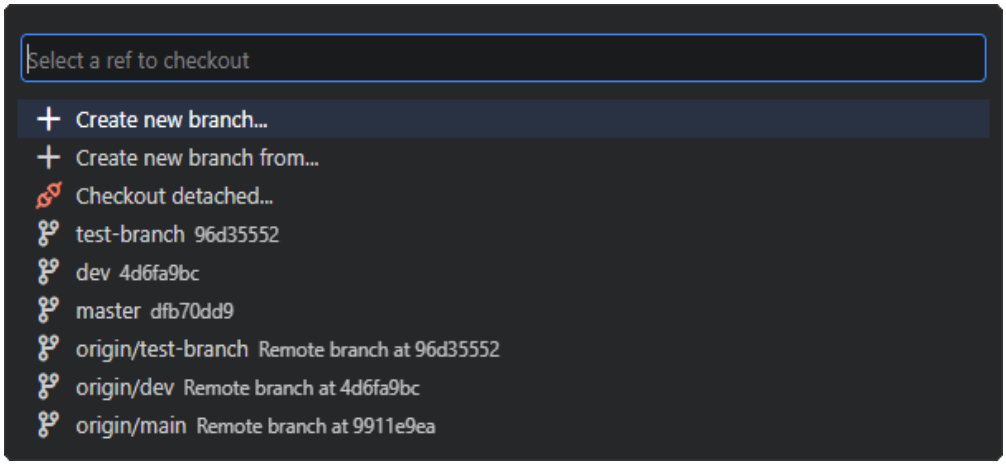
CodeArts IDE会自动创建一个新分支并切换到该分支。如果Git存储库已设置远程，可以在**Source Control Repositories**部分或CodeArts IDE状态栏中单击**Publish**按钮（）将当前分支发布到远程。

切换分支

1. 执行以下操作之一：
   - 在**Source Control**视图中，展开**Source Control Repositories**部分，单击要切换到另一个分支的存储库旁边的**More Actions**按钮（···），然后选择**Checkout to**。
   - 在CodeArts IDE状态栏中单击分支名称。

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。您还可以通过**Checkout to**弹出窗口创建新的本地分支。

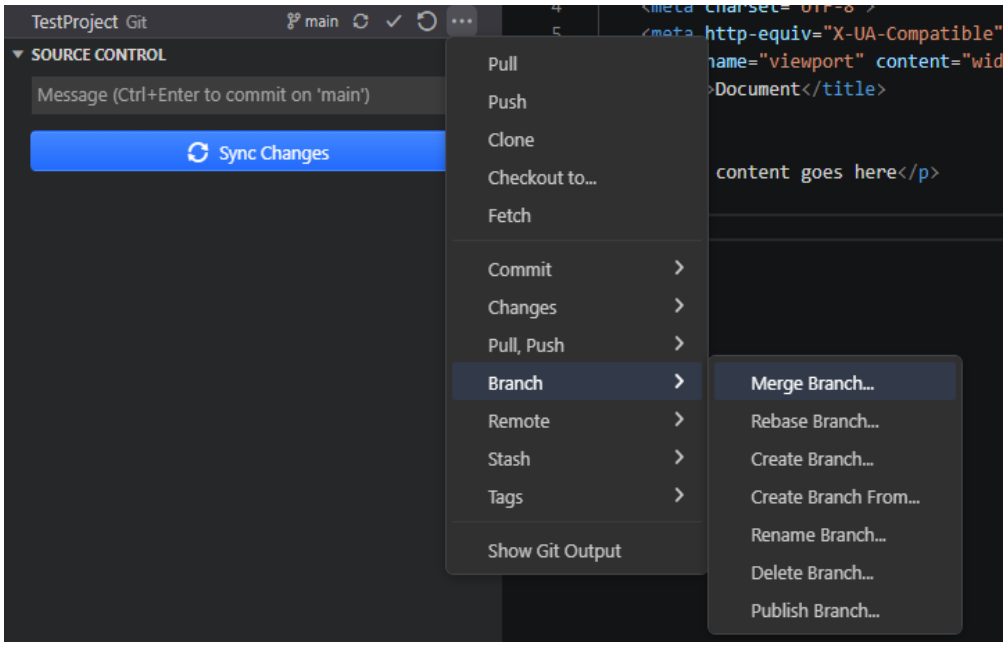
9.3.5.2 应用分支之间的更改

CodeArts IDE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。

合并

Merge命令允许您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，即您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**Source Control**视图中，展开**Source Control Repositories**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**More Actions**按钮（···），指向**Branch**，然后选择**Merge Branch**。

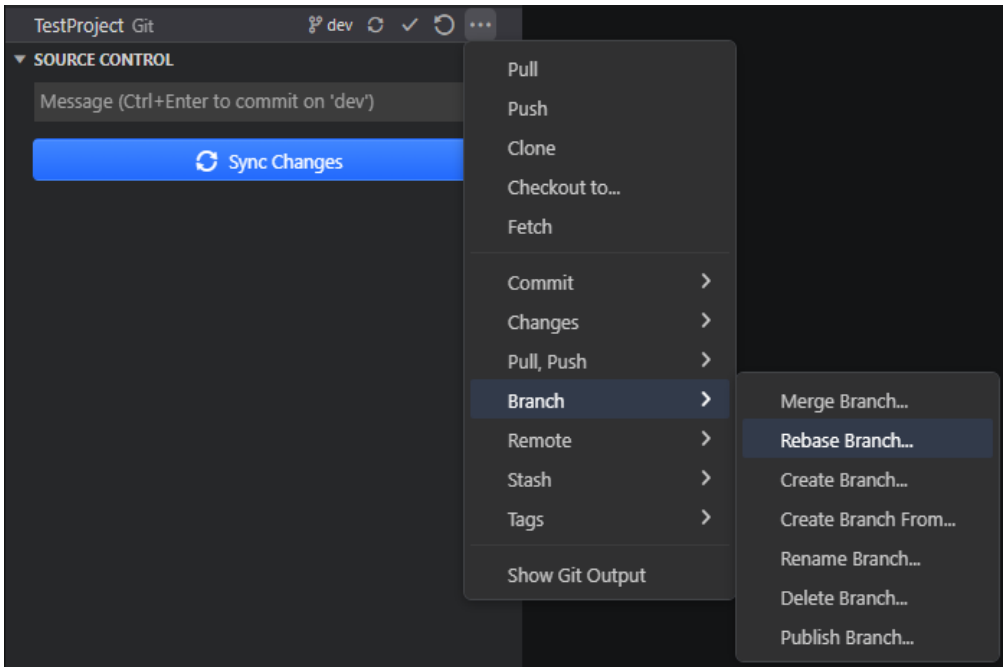


在打开的弹出窗口中，选择要从中合并更改的分支。如果发生合并冲突，请按照**解决合并冲突**中描述的方法解决它。

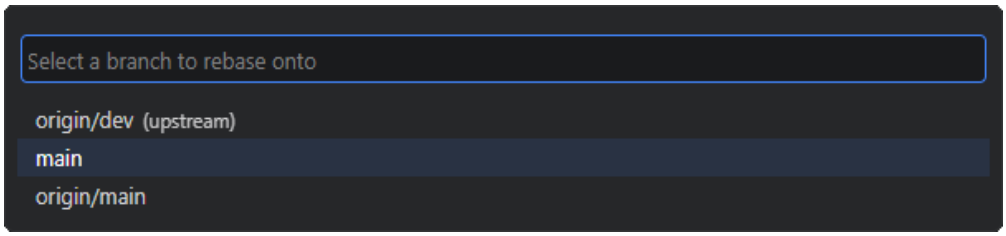
变基

Rebase命令允许您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，即您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**Source Control**视图中，展开**Source Control Repositories**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**More Actions**按钮（ ），指向**Branch**，然后选择**Rebase Branch**。

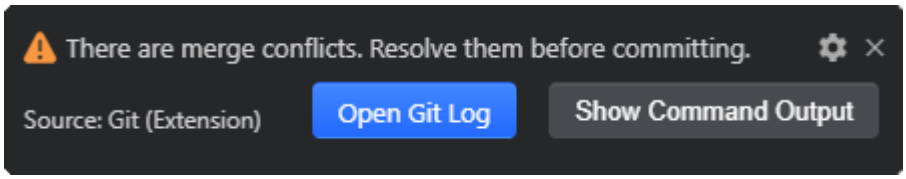


4. 在打开的弹出窗口中，选择您要将更改应用到的目标分支。



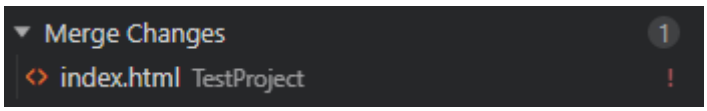
解决合并的冲突

在某些情况下，您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。

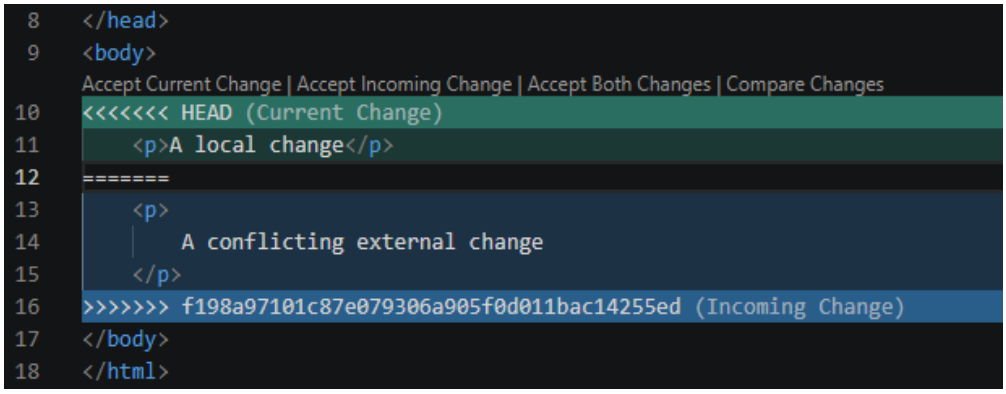


解决合并冲突的步骤如下：

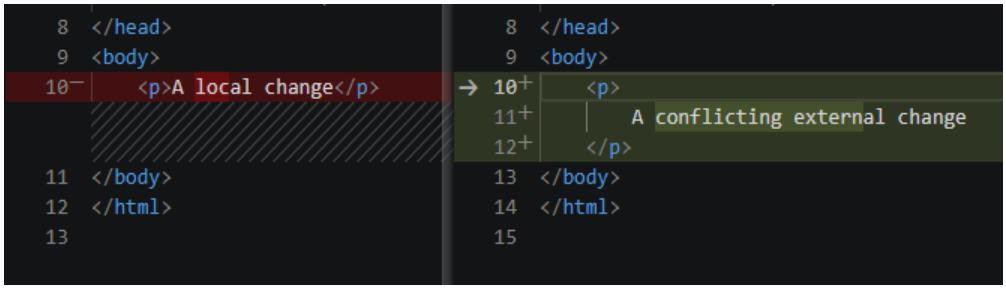
1. 在源代码控制视图的合并更改部分，找到包含冲突更改的文件。



2. 双击该文件，在代码编辑器中打开它，进入专门的冲突视图。使用内联的CodeLens来处理合并冲突：您可以接受当前更改、传入的更改，或者两个更改都接受。



要通过**差异查看器**详细查看更改，请单击**Compare Changes**。



一旦冲突解决完毕，您可以将冲突的文件暂存并**提交**更改。

9.3.6 CodeArts IDE 作为 Git 编辑器

当您从**命令行启动CodeArts时**，您可以传递--wait参数，使启动命令等待直到您关闭新的CodeArts实例。这在将CodeArts配置为Git的外部编辑器时非常有用，这样Git就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**Add to PATH**。
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：

```
[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~3 -i：使用CodeArts进行交互式编辑。

git commit：将CodeArts用作提交消息的编辑器。

git add -p：接着输入e进行交互式添加。

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

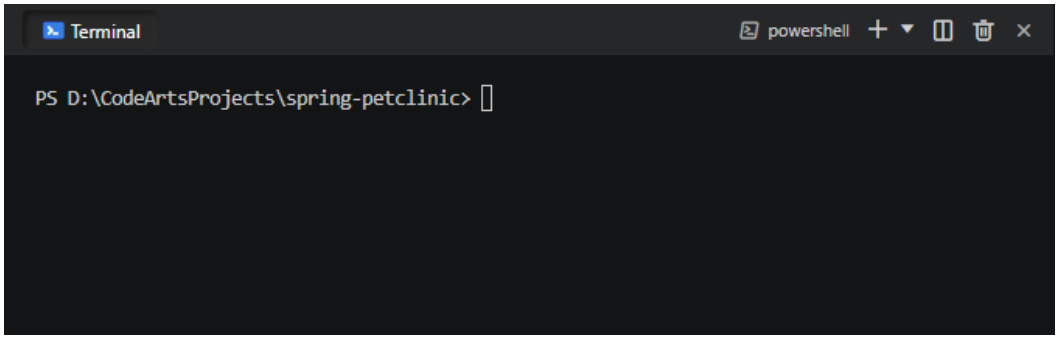
10 集成终端

10.1 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。它提供的编辑器集成能支持**链接**等功能。

要打开终端，请执行以下任一操作：

- 按“Ctrl+`”/“Shift+Escape”/“Alt+F12”。
- 在主菜单中，选择**View**>**Terminal**。
- 从**命令选项板**（“Ctrl+Shift+P”）中，运行**View: Toggle Terminal**命令。

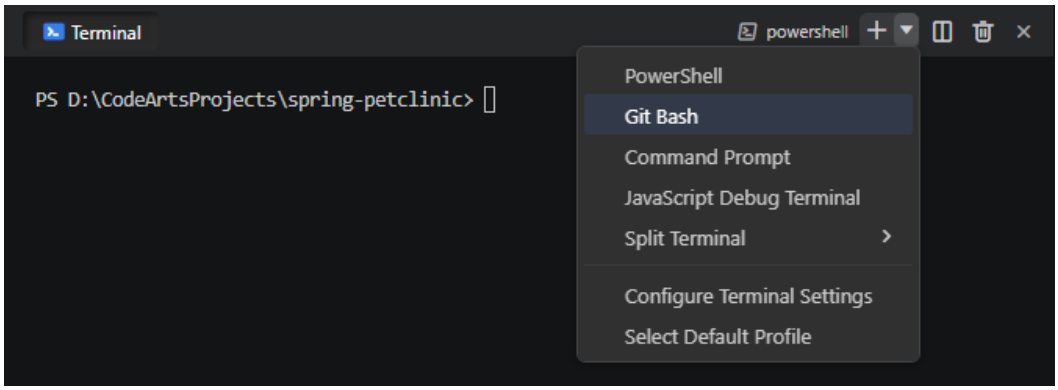


、

集成终端shell正在以CodeArts IDE的权限运行。如果您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。如果您想要在CodeArts IDE的外部打开终端，可以通过按“Ctrl+Shift+C”。

10.2 终端

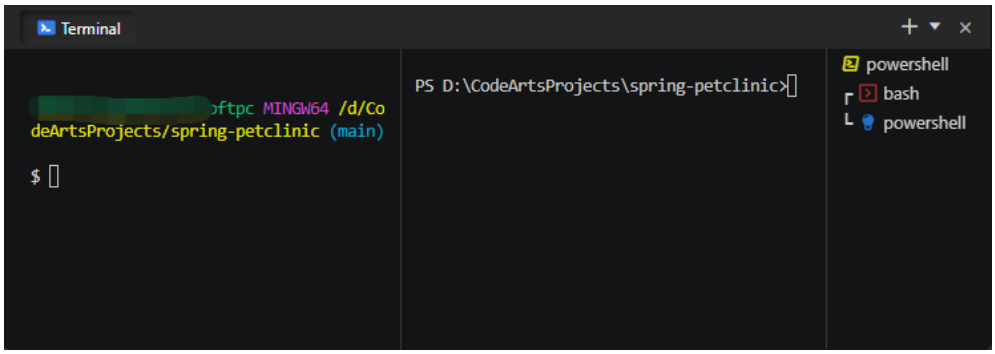
集成终端可以使用安装在计算机上的各种终端，默认为PowerShell。您可以从**Launch Profile**列表（ ▾ ）中选择其他可用的终端类型（如Command Prompt或Git Bash）。



10.3 终端管理

10.3.1 简介

终端实例以选项卡显示，这些选项卡展示在**Terminal**视图的右侧。每个实例都有一个条目，其中包含其名称、图标、颜色和组合装饰（对于分组实例）。您可以通过终端设置项terminal.integrated.tabs.location更改选项卡位置。



10.3.2 添加终端实例

执行以下任一操作：

- 单击**Terminal**视图右上角的**Add**按钮（+），或按“Ctrl+Shift+`”，打开具有默认配置文件的实例。
- 单击**Launch Profile**按钮（▾），然后从列表中选择所需的配置文件。

10.3.3 删除终端实例

在选项卡列表中，悬停一个选项卡，然后单击**Delete**按钮（🗑），或选择一个选项卡并按“Delete”。

10.3.4 终端实例分组

Terminal提供了多种功能，让您可以自定义其布局。

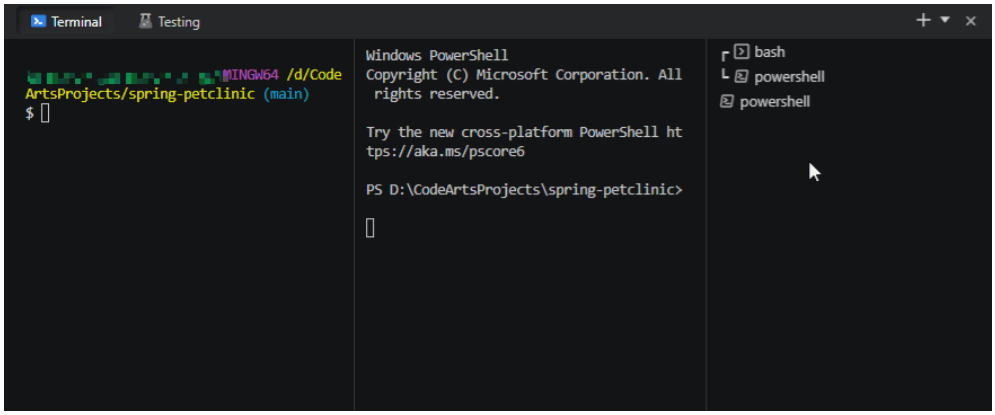
要将当前终端实例拆分为两个，从而创建组，请执行以下任一操作：

- 在选项卡列表中，悬停选项卡，然后单击**Split**按钮（⫿）。

- “Alt +Click”选项卡或单击**Add**按钮（ ）。
- 按“Ctrl+Shift+5”。

要将终端添加到组，请将选项卡拖入主**Terminal**区域。要重新排列组中的选项卡，请将它们拖到列表中。

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**Unsplit Terminal**。



要在终端组内导航，请使用以下键盘快捷键。

- 按“Ctrl+Pagedown”/“Alt+Right”键聚焦下一组。
- 按“Ctrl+Pageup”/“Alt+Left”键聚焦上一组。

在组中，通过使用“Alt+Up”/“Alt+Left”聚焦上一个窗格，并使用“Alt+Down”/“Alt+Right”聚焦下一个窗格，在终端之间导航。

10.3.5 自定义选项卡

要更改终端的名称、图标或选项卡颜色，请右键单击其选项卡，然后从上下文菜单中选择相应的操作，或通过“命令选项板”（“Ctrl+Shift+P”/“Ctrl Ctrl”）触发命令：

命令	命令ID
Terminal: Rename	**workbench.action.terminal.rename**
Terminal: Change Icon	**workbench.action.terminal.changeIcon**
Terminal: Change Color	**workbench.action.terminal.changeColor**

10.4 终端简介

10.4.1 简介

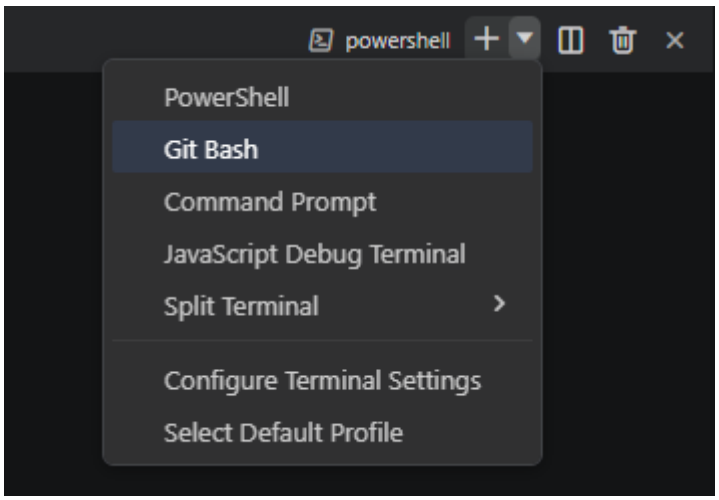
终端配置文件是特定于平台的终端配置，由可执行路径、参数和其他自定义项组成。

配置文件示例：

```
{
  "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"
}
```

您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）。

通过运行**Terminal: Select Default Profile**命令配置默认集成终端，该命令也可通过**Terminal**视图中的**Launch Profile**列表访问。



10.4.2 配置模板

要创建新的配置文件，请运行**Terminal: Select Default Profile**命令，然后单击shell右侧的**Configure Terminal Profile**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**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，您可以使用直接指向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"]
    }
  }
}
```

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"
}
```

10.4.3 删除内置配置文件

要从**Launch Profile**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，请使用以下设置。

```
{
  "terminal.integrated.profiles.windows": {
    "Git Bash": null
  }
}
```

10.5 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。请注意，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"
}
```

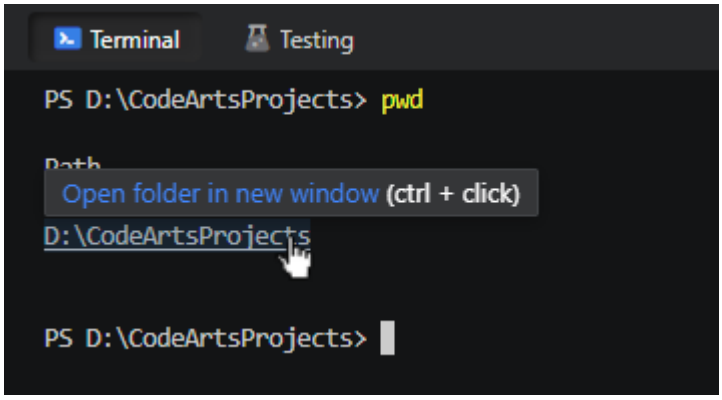
10.6 终端进程重连

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。

10.7 链接

终端检测链接，当您悬停文件或URL时，显示下划线。您可以“Ctrl+Click”链接以转到其目标。



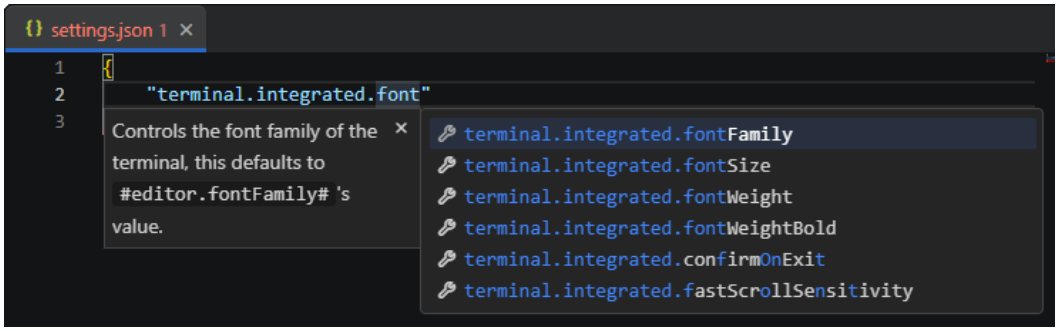
根据链路类型，激活它将执行以下操作之一。

- 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。
- 使用包含所有匹配项的快速拣货搜索工作区。

10.8 终端外观

您可以使用**terminal.integrated.***组中的以下设置自定义终端的外观。

- Font：字体系列、字号和字体粗细，
- Spacing：行高和字母间距。
- Cursor：样式、宽度和闪烁。



10.9 使用鼠标

10.9.1 右键单击行为

在终端内部单击鼠标右键时，将复制选定的文本（如果有），并释放选中内容。如果未选择文本，则执行粘贴。这可以使用terminal.integrated.rightClickBehavior单击行为设置进行配置。

10.10 键绑定和 shell

当**Terminal**视图聚焦时，许多CodeArts IDE键绑定将不起作用，因为击键将传递到终端本身并由终端本身消耗。有一个预定义的命令列表，这些命令跳过shell处理，而是发送到CodeArts IDE密钥绑定系统。您可以使用**terminal.integrated.commandsToSkipShell**设置自定义此列表。可以通过将命令名称添加到列表中来将命令添加到此列表中，并通过将命令名称添加到以破折号-开头的列表中来删除命令。请参阅设置详细信息以查看默认命令的完整列表。

```
{
  "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**并将键绑定发送到shell而不是工作台，请设置**terminal.integrated.sendKeybindingsToShell**。

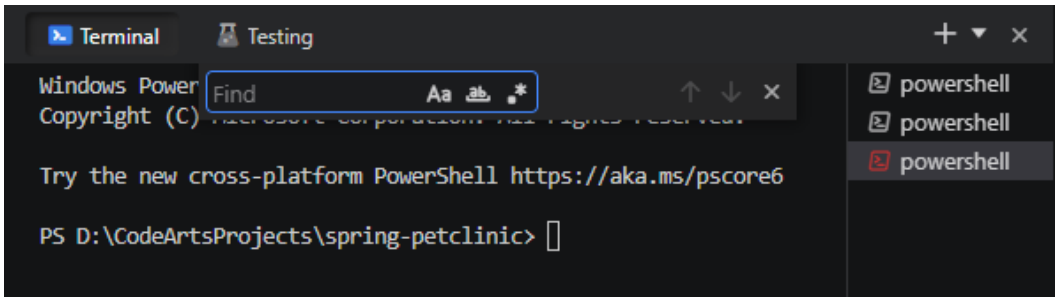
终端中的同时按下键绑定

默认情况下，当同时按下键绑定是最高优先级时，它将始终跳过终端shell（绕过**terminal.integrated.commandsToSkipShell**）转到CodeArts IDE。这通常是想要的行为，除非您希望shell使用“Ctrl+K”（对于bash，这将剪切光标之后的行）。这可以通过**terminal.integrated.allowChords**设置禁用：

```
{
  "terminal.integrated.allowChords": false
}
```

10.11 在终端中查找文本

集成终端具有查找功能，可使用**Ctrl+F**触发。



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：

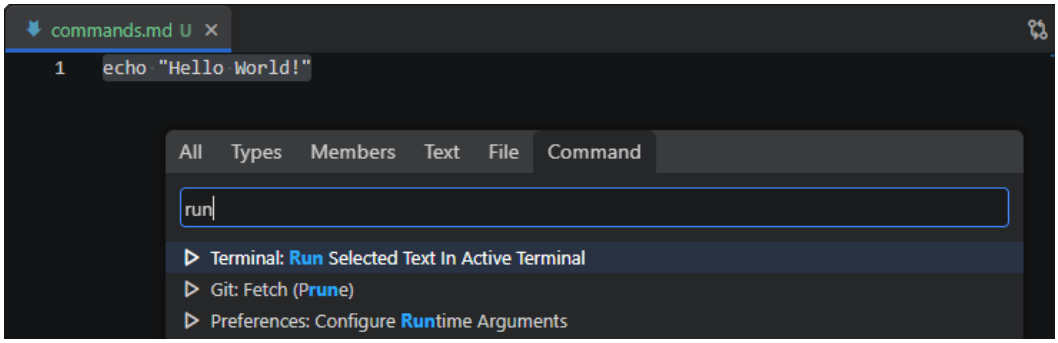
```
{
  "terminal.integrated.commandsToSkipShell": [
    "-workbench.action.terminal.focusFind"
  ],
}
```

说明

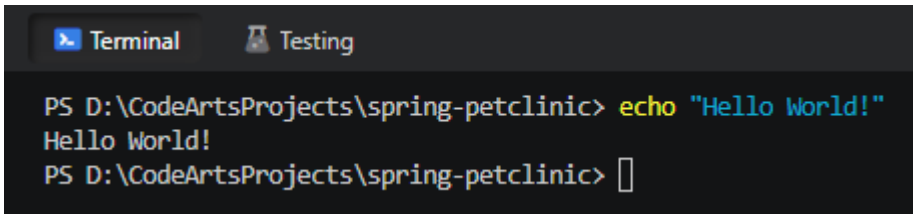
有关在CodeArts IDE中搜索文本的详细信息，请参见通过**代码搜索**。

10.12 运行选定的文本

要通过终端执行某些文本，如脚本的一部分，请在编辑器中选择它，然后通过“命令面板”（“Ctrl+Shift+P”/“Ctrl Ctrl”）运行命令**Terminal: Run Selected Text in Active Terminal**：



终端尝试运行选定的文本。



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。

11 命令行界面

11.1 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。

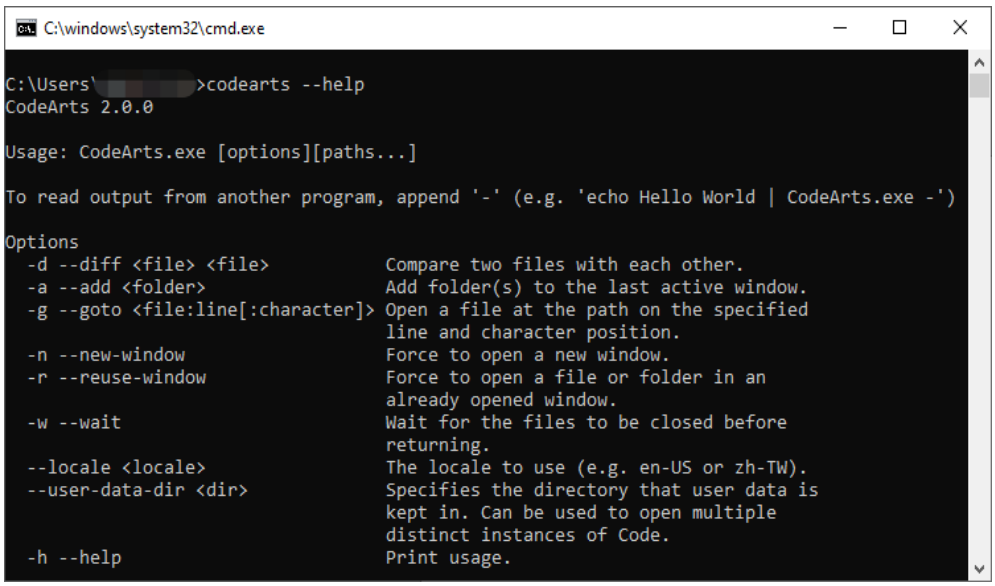
要使CodeArts IDE命令行实用程序正常工作，必须将CodeArts IDE二进制位置（默认情况下是C:\Program Files\CodeArts\bin）添加到系统路径中。通常，这在安装期间自动执行。否则，您可以手动将位置添加到Path环境变量中。

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**集成终端**。

11.2 命令行帮助

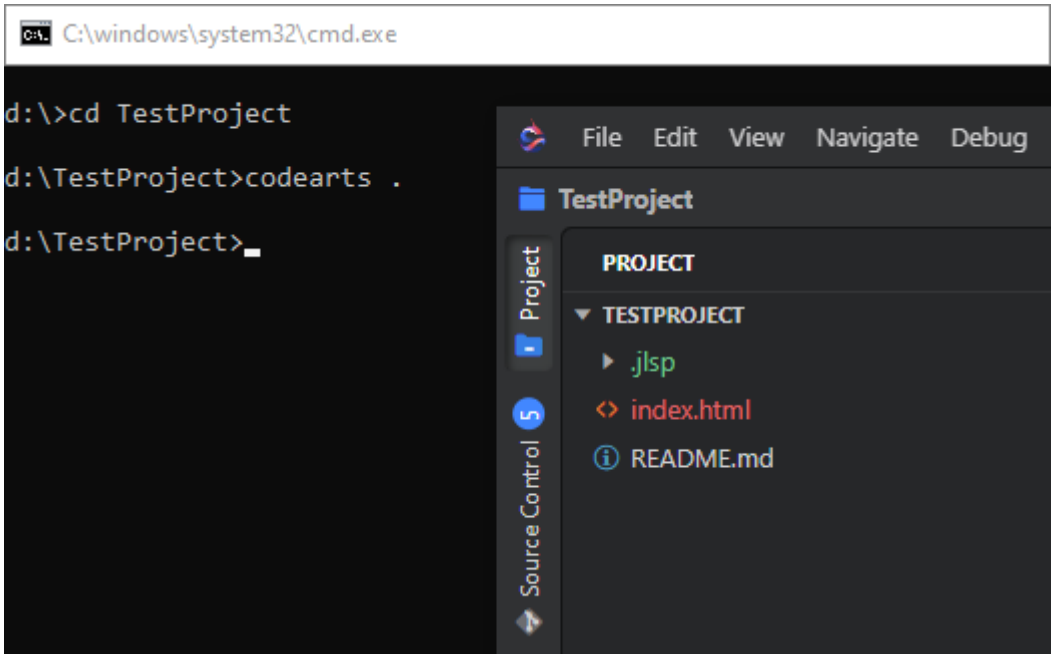
要了解CodeArts IDE命令行界面的概述，请打开终端或命令提示符，然后键入**codearts --help**。



11.3 从命令行启动

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。



11.4 核心 CLI 选项

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。

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**文件差异编辑器**。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**）。

11.5 打开文件和文件夹

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：

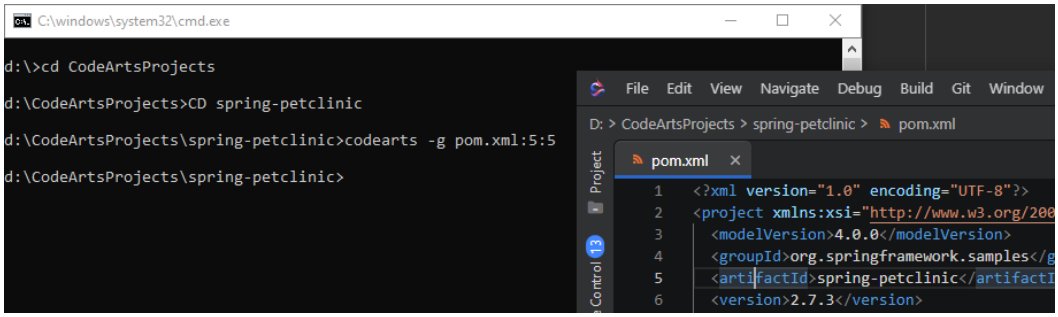
```
codearts index.html style.css documentation\readme.md
```

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。

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。



11.6 使用扩展

您可以从命令行安装和管理CodeArts IDE扩展。

参数	描述
--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。

11.7 CLI 高级选项

有几个CLI选项有助于重现错误和高级设置。

参数	描述
-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。

11.8 通过 URLs 打开 CodeArts IDE

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。

12 CodeArts IDE 设置

12.1 简介

您可以通过CodeArts IDE设置配置CodeArts IDE编辑器、用户界面和功能行为的几乎每一部分。

CodeArts IDE提供了几个设置范围：

- **User Settings**全局应用于您打开的任何CodeArts IDE实例。
- **Workspace Settings**存储在工作区中，仅在打开工作区时应用。

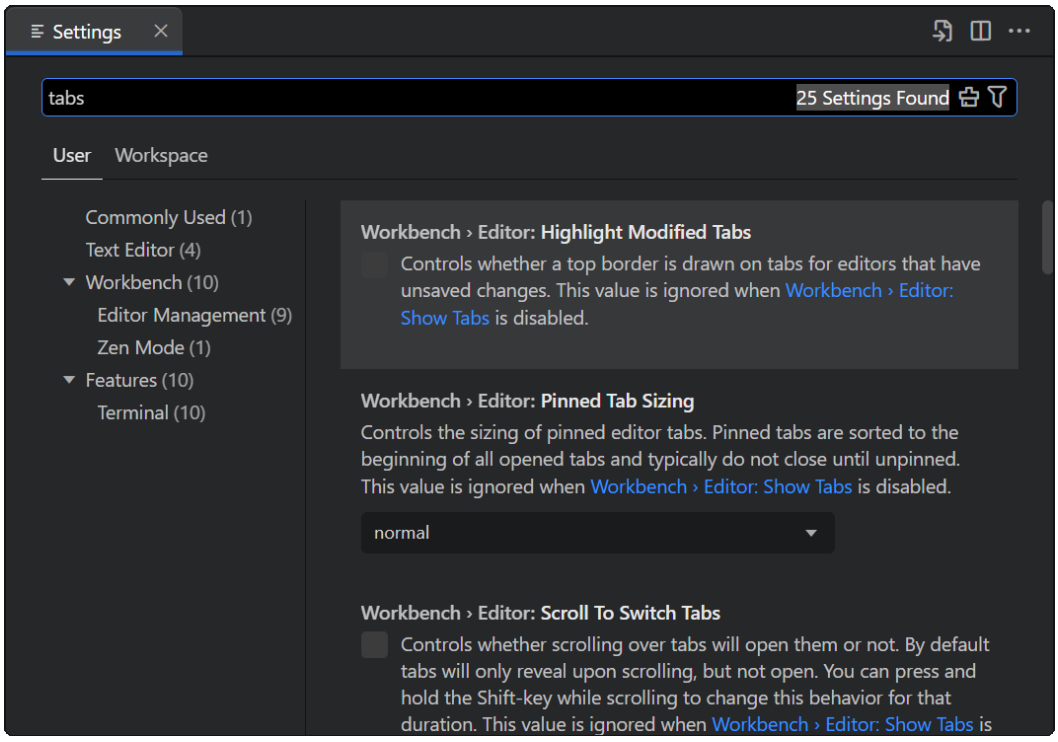
12.2 设置编辑器

12.2.1 简介

要修改用户设置，请使用**Settings**编辑器，您可以通过执行以下任何操作打开该编辑器。

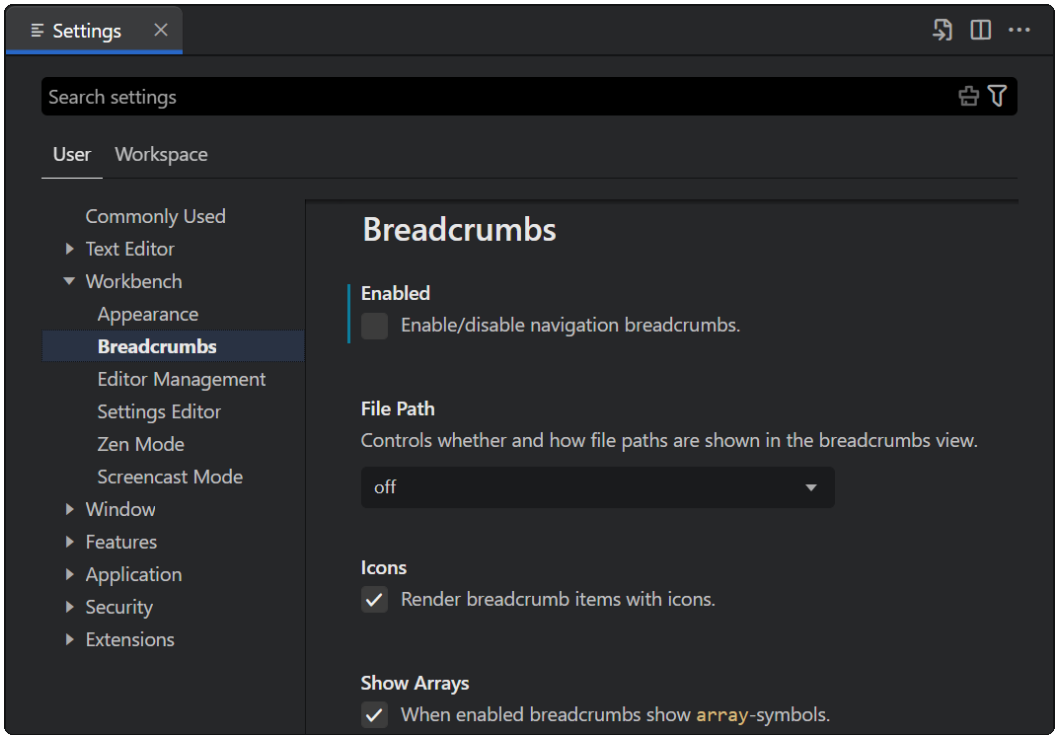
- 按“Ctrl+，”。
- 在左侧活动栏中选择**Manage**>**Settings**。
- 在命令选项板（“Ctrl+Shift+P”/“Ctrl Ctrl”）中，搜索并运行**Preferences: Open Settings**命令。

在**Settings**编辑器中，使用**Search**字段搜索所需的设置。这将显示并突出显示与搜索条件匹配的设置，并过滤掉不匹配的设置。

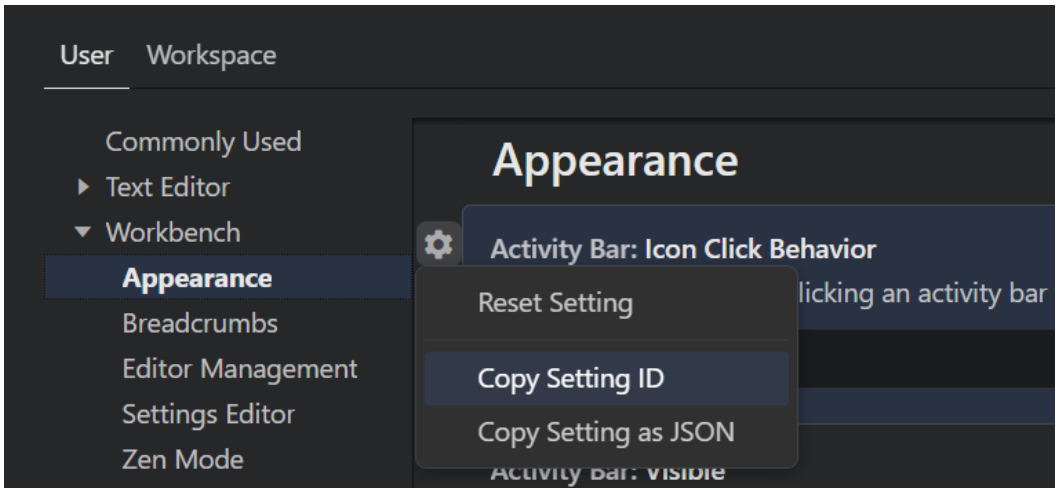


CodeArts IDE会自动动态应用对设置的更改。修改后的设置用一条类似于编辑器中修改后的行的蓝线表示。

在以下示例中，**面包屑导航**已禁用。

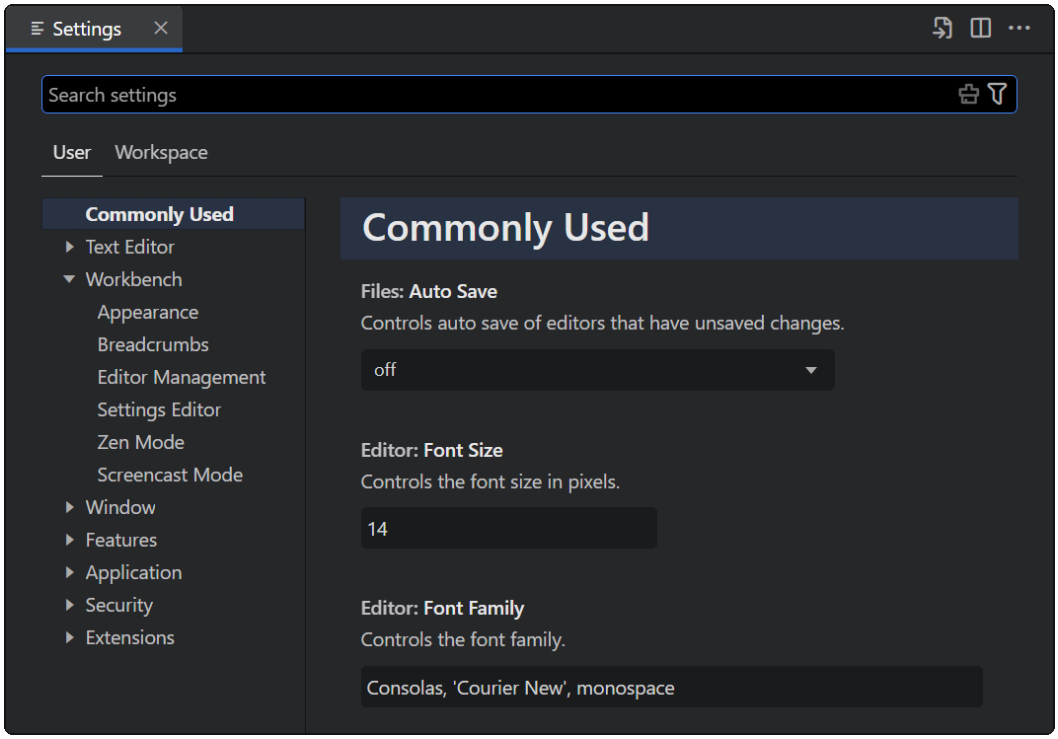


选择一个设置，然后单击**More Actions**（ ）按钮，或按“Shift+F9”打开上下文菜单，允许您将设置重置为其默认值，并复制其ID或JSON名称-值对。



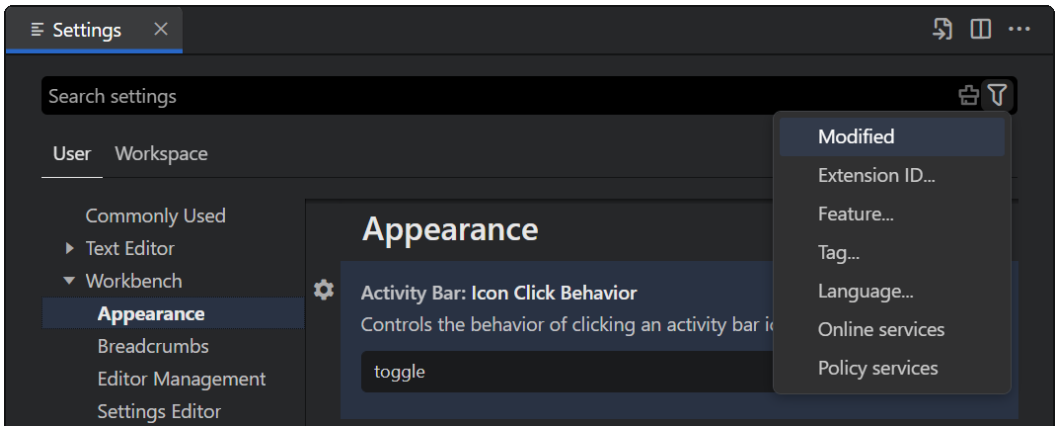
12.2.2 设置组

设置以组表示，以便您可以轻松导航它们。顶部的**Commonly Used**组列出了常用的自定义设置。CodeArts IDE扩展还可以添加自己的自定义设置，这些设置收集在**Extensions**部分下。

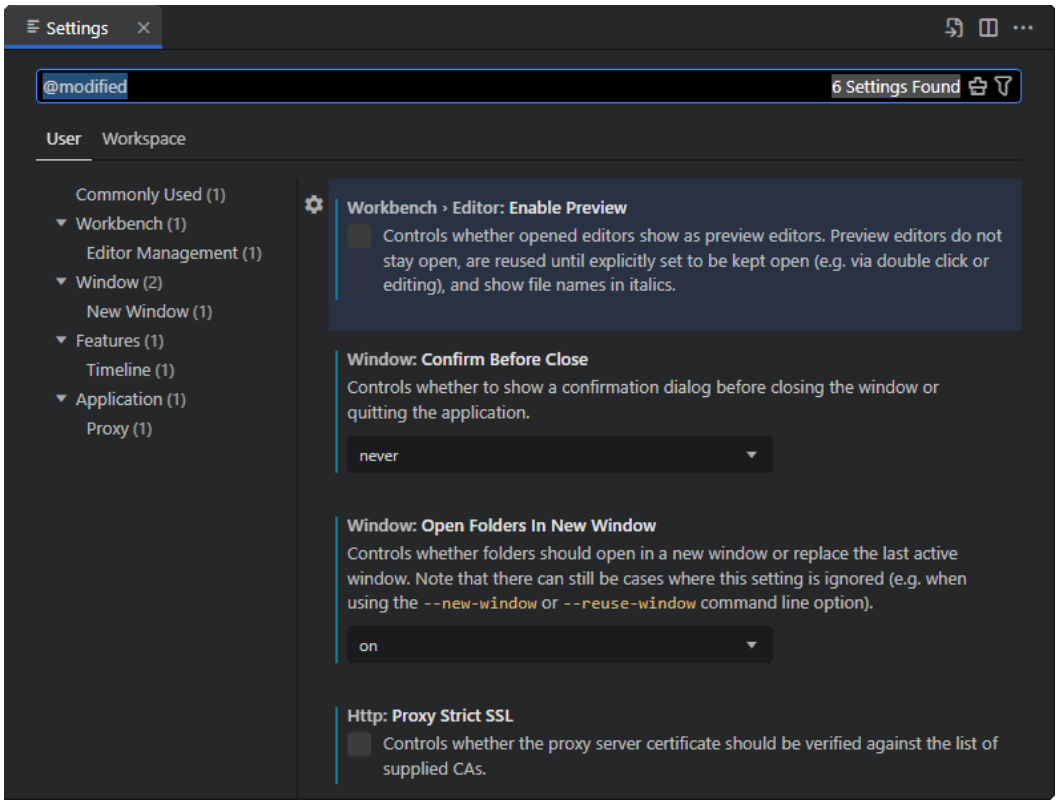


12.2.3 设置编辑器筛选器

Settings编辑器**Search**栏提供了几个过滤器，使您更容易管理设置。使用**Search**栏中的**Filter**按钮（ ）轻松添加过滤器。



要检查您配置的设置，请使用**@modified**的过滤器。如果设置的值与默认值不同，或者如果其值在相应的设置JSON文件中显式设置，则会显示在此过滤器下。



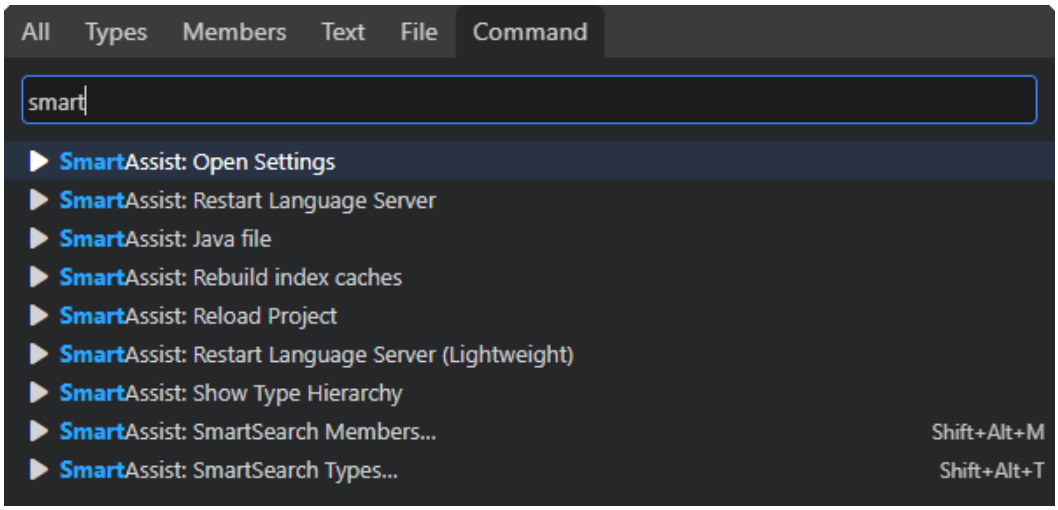
还有几个其他方便的过滤器可以帮助搜索设置：

- **@ext**：特定于分机的设置。您提供扩展ID，如**@ext:markdown-language-features**。
- **@feature**：特定于**Features**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应用语言过滤器。例如，**@lang:typescript**。
- **@tag**：特定于CodeArts IDE子系统的设置。

Search栏记住您的设置搜索查询，并支持撤消/重做（“Ctrl+Z”/“Ctrl+Shift+Z”/“Ctrl+Y”）。您可以使用**Search**栏右侧的**Clear Settings Search Input**按钮（ ）快速清除搜索项或筛选器。

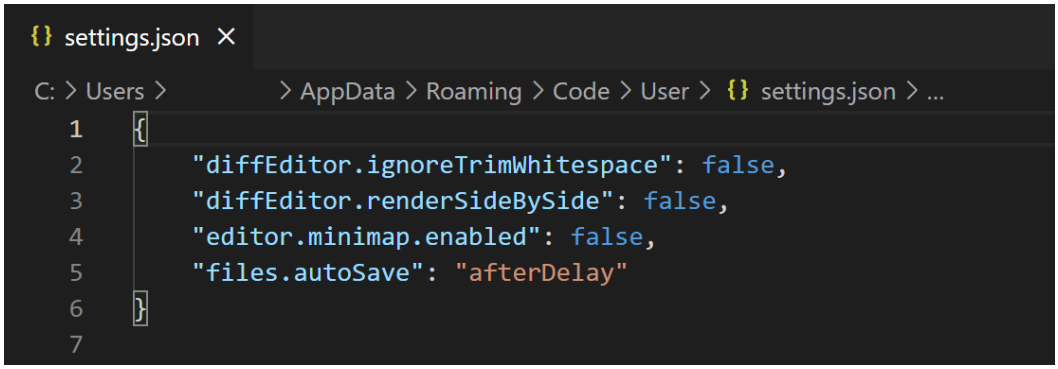
12.2.4 分机设置

安装的CodeArts IDE扩展也可以贡献自己的设置，您可以在设置编辑器的**Extensions**部分查看这些设置。要打开SmartAssist for Java设置，请在“命令选项板”中运行**SmartAssist: Open Settings**命令（“Ctrl+Shift+P”/“Ctrl Ctrl”）。

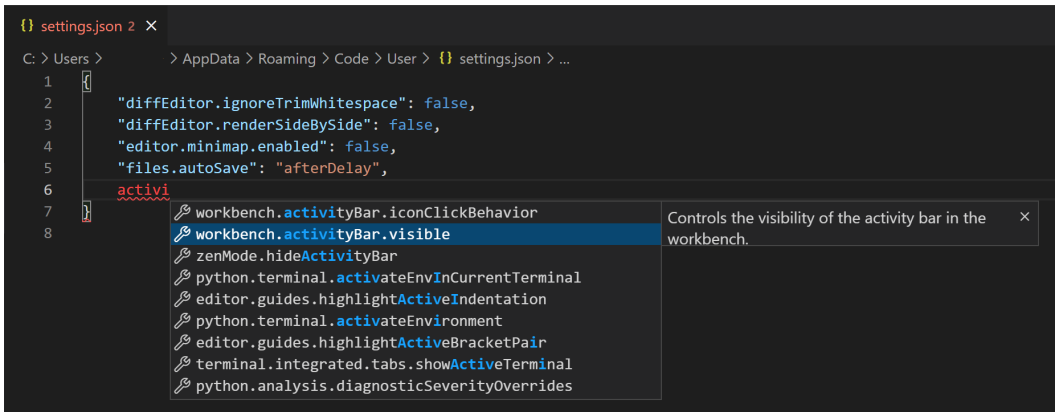


12.3 settings.json

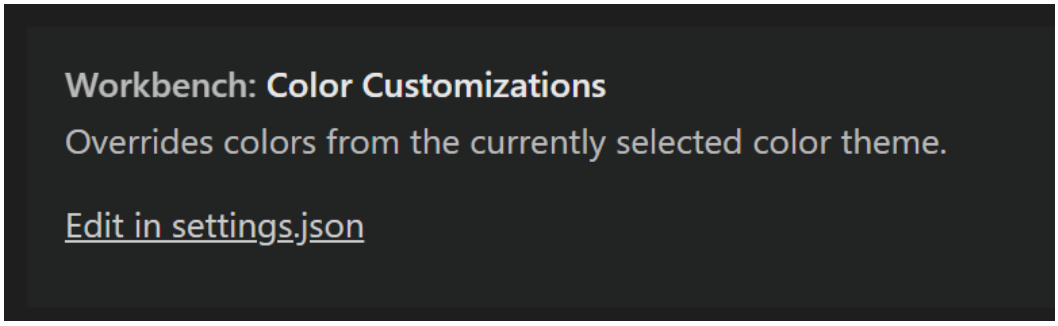
设置编辑器是允许您查看和修改存储在**settings.json**文件中的设置值的UI。您可以通过使用**Preferences: Open Settings (JSON)**命令在编辑器中打开此文件，直接查看和编辑此文件。通过指定设置ID和值，设置将写入JSON。



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。



某些设置，如**Workbench: Color Customizations**只能在**settings.json**中编辑，并在设置编辑器中显示**Edit in settings.json**链接。



如果您喜欢始终直接使用**settings.json**，您可以设置**workbench.settings.editor: json**，以便将**Manage**>**Settings**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

用户设置文件位于%APPDATA%\CodeArts\User\Settings.json下。

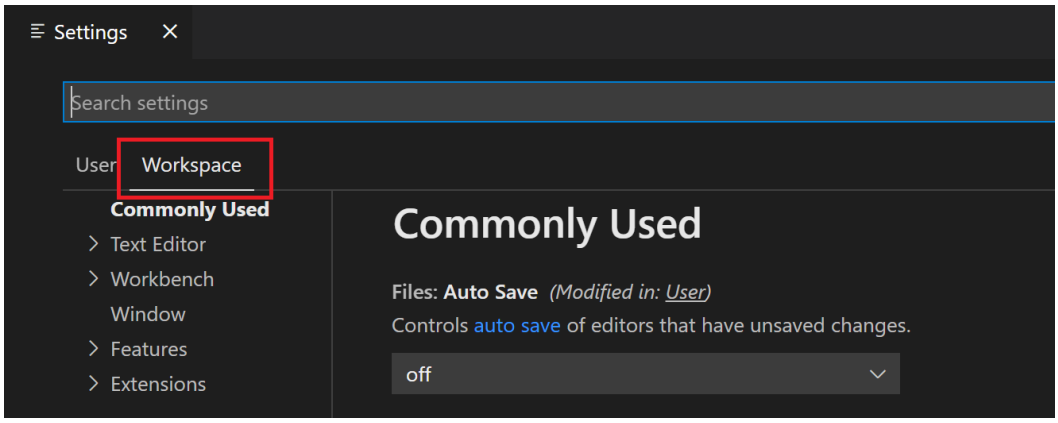
重置所有设置

虽然您可以通过设置编辑器中的**Reset Setting**命令单独重置设置，但您也可以通过打开**settings.json**并删除大括号**{}**之间的条目来重置所有更改的设置。当您通过清除**settings.json**重置设置时，无法恢复其以前的值。

12.4 工作区设置

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。

您可以在设置编辑器的**Workspace**选项卡上编辑工作区设置。要快速打开此选项卡，请使用“命令选项板”中的**Preferences: Open Workspace Setting**（“Ctrl+Shift+P”/“Ctrl Ctrl”）命令。



设置编辑器的所有功能，如设置组、搜索和筛选，对于工作区设置的行为相同。并非所有用户设置都可作为工作区设置使用。例如，与更新和安全性相关的应用程序范围设置不能被Workspace设置覆盖。

工作区 settings.json 位置

与用户设置类似，工作区设置也存储在一个**settings.json**文件中，您可以通过**Preferences: Open Workspace Settings (JSON)**命令直接编辑该文件。工作区设置文件位于项目根文件夹的**.arts**文件夹下。当您将工作区设置**settings.json**文件添加到**源代码管理**时，该项目的设置将由该项目的所有用户共享。

12.5 设置优先级

配置可以在多个级别上被不同的设置范围覆盖。在以下列表中，较晚的作用域覆盖较早的作用域：

- Default settings：此范围表示默认未配置的设置值。
- User settings：全局应用于所有CodeArts IDE实例。
- Workspace settings：应用于打开的文件夹或工作区。
- Language-specific default settings：这些是特定于语言的默认值，可由扩展提供。
- Language-specific user settings：特定于语言的用户设置：与用户设置相同，但特定于语言。
- Language-specific workspace settings：与工作区设置相同，但特定于语言。

设置值可以是多种类型：

- String："files.autoSave"："afterDelay"
- Boolean："editor.minimap.enabled"：true
- Number："files.autoSaveDelay"：1000
- Array："editor.rulers"：[]
- Object："search.exclude"：{ "**/node_modules"： true, "**/bower_components"： true }

具有基元类型和数组类型的值将被覆盖，这意味着使用作用域中优先于另一个作用域的配置值，而不是另一个作用域中的值。但是，具有对象类型的值将合并。

例如，**workbench.colorCustomizations**采用一个对象，该对象指定一组UI元素及其所需颜色。如果您的用户设置将编辑器背景设置为蓝色和绿色。

```
"workbench.colorCustomizations": {
  "editor.background": "#000088",
  "editor.selectionBackground": "#008800"
}
```

打开的工作区设置将编辑器前景设置为红色：

```
"workbench.colorCustomizations": {
  "editor.foreground": "#880000",
  "editor.selectionBackground": "#00FF00"
}
```

当该工作区打开时，结果是这两种颜色自定义的组合，就像您指定了：

```
"workbench.colorCustomizations": {
  "editor.background": "#000088",
  "editor.selectionBackground": "#00FF00",
  "editor.foreground": "#880000"
}
```

如果存在冲突的值，如上面示例中的**editor.selectionBackground**，则会发生通常的覆盖行为，工作区值优先于用户值，语言特定的值优先于非语言特定的值。

12.6 设置和安全性

某些设置允许您指定CodeArts IDE将运行以执行某些操作的可执行文件。例如，您可以选择集成终端应使用的shell。为了增强安全性，此类设置只能在用户设置中定义，而不能在工作区范围中定义。

以下是工作区设置中不支持的设置列表：

- **git.path**
- **terminal.external.windowsExec**
- **terminal.external.osxExec**
- **terminal.external.linuxExec**

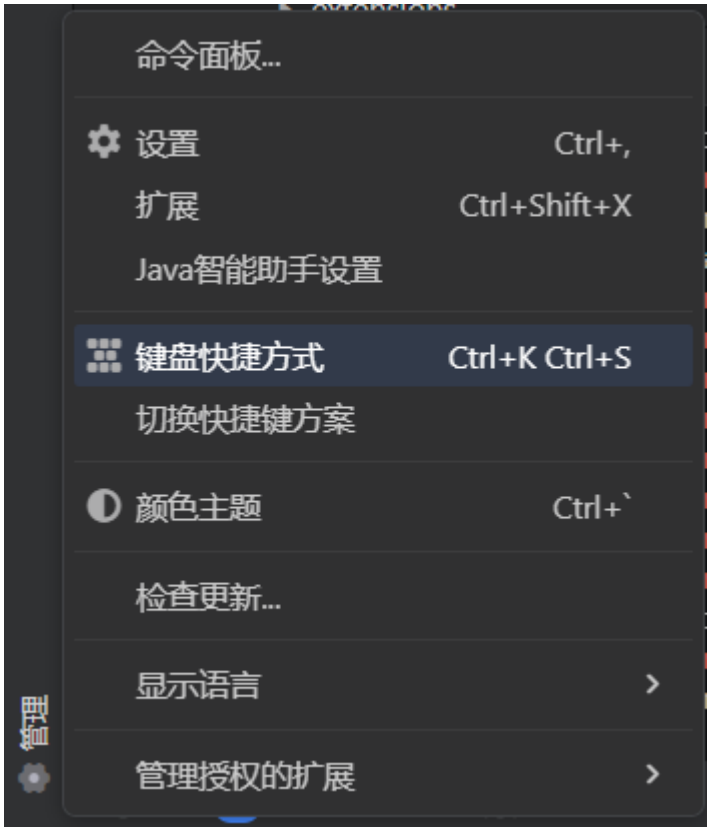
第一次打开定义这些设置中任何一个的工作区时，CodeArts IDE将警告您，然后始终忽略之后的值。

13 默认键绑定

13.1 简介

CodeArts IDE中提供了丰富的快捷键功能，您可以根据您的需求随时查看当前快捷键方案或者修改快捷键方案。

如果您需要查看当前CodeArts IDE的快捷键方案，请单击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。



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**Keyboard Shortcuts**编辑器中应用@source:default过滤器（**Source**为

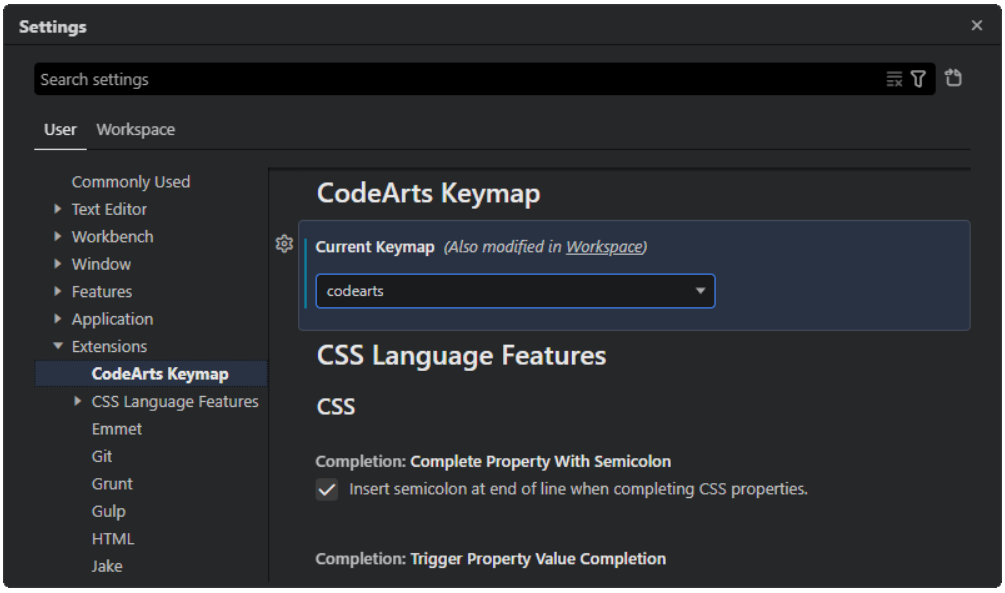
“Default”）。

您还可以通过使用**Preferences: Open Default Keyboard Shortcuts (JSON)**命令，将默认键盘快捷方式视为JSON文件进行查看。

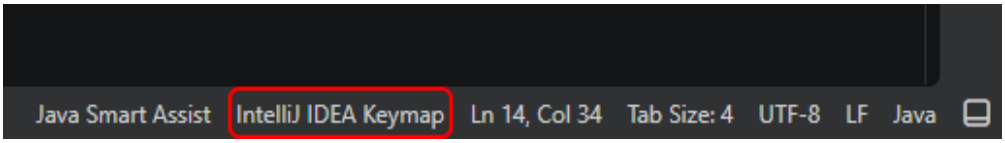
CodeArts IDE默认提供了两个预定义的键位映射：本机CodeArts IDE和IDEA，后者使用IntelliJ IDEA中对应的映射覆盖了一些最常用的快捷方式。如果您习惯在IntelliJ IDEA中工作，您可以选择IDEA键位映射，从而继续使用熟悉的快捷方式。

要在键位映射之间切换，请执行以下任一操作：

- 在**Settings**编辑器（“Ctrl+,”）中，转到**Extensions**>**CodeArts IDE Keymap**。然后从**Current Keymap**列表中选择所需的键位映射。



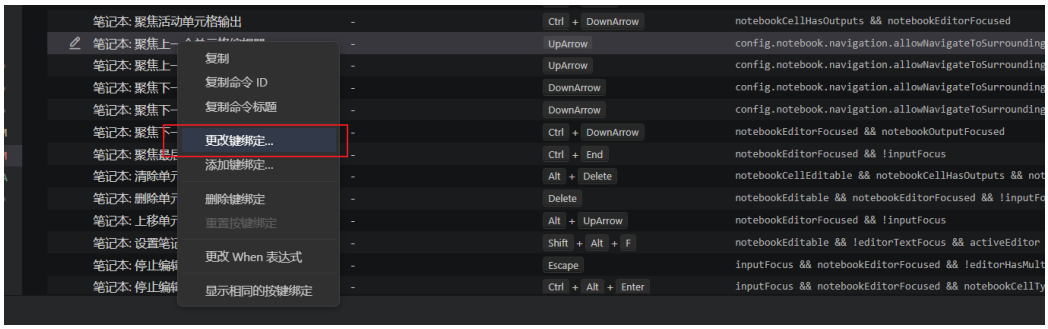
- 在CodeArts IDE状态栏中，单击键位映射指示器，然后在弹出的菜单中选择所需的键位映射。



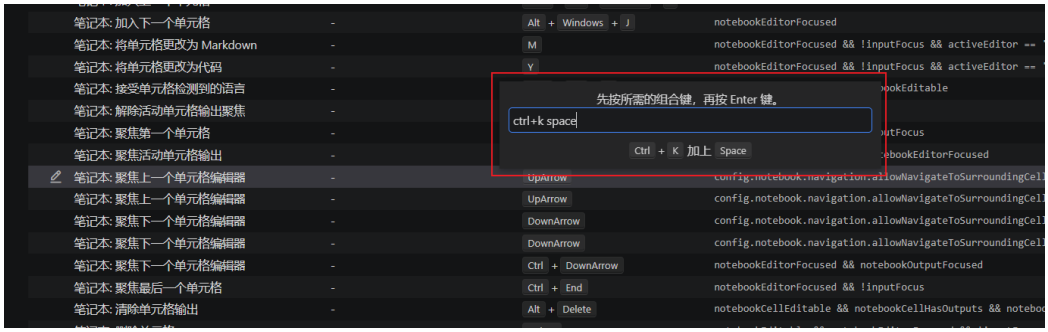
13.2 修改快捷键方案

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。

方式1：右键单击您想要修改的快捷键，选择**更改键绑定菜单**。

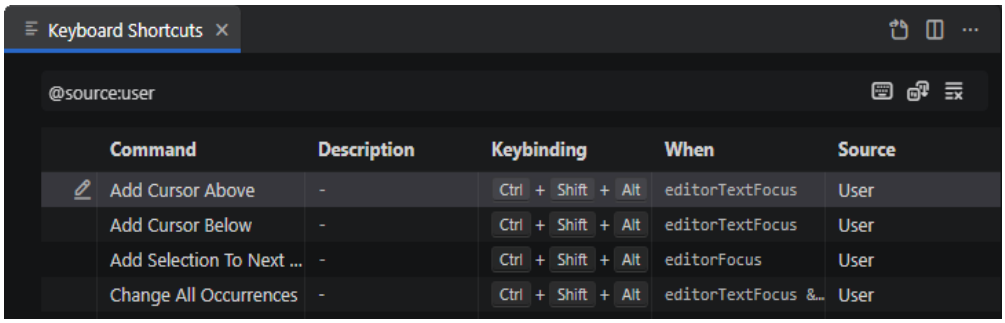


方式2：双击该快捷键方案所在的行，在唤起的弹窗中输入想要绑定的新快捷键：



13.3 查看和重置已修改的键绑定

要查看CodeArts IDE中任何用户修改的键盘快捷键，请单击**More Actions**按钮（ ）并选择弹出菜单中的**Show User Keybindings**。这将在**Keyboard Shortcuts**编辑器中应用**@source:user**过滤器（**source**为“用户”）。



执行以下操作之一：

- 要重置单个按键绑定，请右键单击列表中相应的条目，然后选择**Reset Keybinding**。
- 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\User\keybindings.json**。

13.4 快捷键绑定参考

13.4.1 基本编辑

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
行剪切	Shift+Delete Ctrl+X	Ctrl+X Shift+Delete	editor.action.clipboardCutAction
行复制	Ctrl+Insert Ctrl+C	Ctrl+Insert Ctrl+C	editor.action.clipboardCopyAction
粘贴	Shift+Insert Ctrl+V	Shift+Insert Ctrl+V	editor.action.clipboardPasteAction
行删除	Ctrl+Shift+K	Ctrl+Shift+K	editor.action.deleteLines
在下面插入行	Ctrl+Enter	Shift+Enter Ctrl+Shift+Enter	editor.action.insertLineAfter
在上面插入行	Ctrl+Shift+Enter	Ctrl+Alt+Enter	editor.action.insertLineBefore
下移行	Alt+Down	Shift+Alt+Down Ctrl+Shift+Down	editor.action.moveLinesDownAction
上移行	Alt+Up	Shift+Alt+Up Ctrl+Shift+Up	editor.action.moveLinesUpAction
向下复制行	Shift+Alt+Down	Ctrl+D	editor.action.copyLinesDownAction
向上复制行	Shift+Alt+Up	Shift+Alt+Up	editor.action.copyLinesUpAction
撤销	Ctrl+Z	Ctrl+Z	undo
重做	Ctrl+Shift+Z Ctrl+Y	Ctrl+Shift+Z Ctrl+Y	redo
将选择添加到下一个查找匹配项	Ctrl+D	Alt+J	editor.action.addSelectionToNextFindMatch
将上一个选择移动到下一个查找匹配项	Ctrl+K Ctrl+D	Ctrl+K Ctrl+D	editor.action.moveSelectionToNextFindMatch

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
撤销上一次光标操作	Ctrl+U	Shift+Alt+J	cursorUndo
在选定的每行末尾插入光标	Shift+Alt+I	Shift+Alt+I	editor.action.insertCursorAtEndOfEachLineSelected
选择当前选择的所有出现位置	Ctrl+Shift+L	Ctrl+Shift+Alt+J	editor.action.selectHighlights
选择当前单词的所有出现位置	Ctrl+F2 F2	Shift+F6	editor.action.changeAll
选择当前行	Ctrl+L	Ctrl+L	expandLineSelection
将光标插入下方	Ctrl+Alt+Down	Ctrl+Alt+Down	editor.action.insertCursorBelow
将光标插入上方	Ctrl+Alt+Up	Ctrl+Alt+Up	editor.action.insertCursorAbove
跳转到匹配的括号	Ctrl+Shift+\	Ctrl+Shift+\	editor.action.jumpToBracket
缩进线	Ctrl+]	Ctrl+]	editor.action.indentLines
突出线	Ctrl+[	Ctrl+[	editor.action.outdentLines
转到行首	Home	Home	cursorHome
转到行尾	End	End	cursorEnd
转到文件末尾	Ctrl+End	Ctrl+End	cursorBottom
转到文件开头	Ctrl+Home	Ctrl+Home	cursorTop
向下滚动行	Ctrl+Down	Ctrl+Down	scrollLineDown
向上滚动	Ctrl+Up	Ctrl+Up	scrollLineUp
向下滚动页	Alt+Pagedown	Alt+Pagedown	scrollPageDown
向上滚动页	Alt+Pageup	Alt+Pageup	scrollPageUp
折叠（折叠）区域	Ctrl+Shift+[	Ctrl+- Ctrl+Numpad-	editor.fold
展开（展开）区域	Ctrl+Shift+]	Ctrl+= Ctrl+Numpad+	editor.unfold

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
折叠（折叠）所有子区域	Ctrl+K Ctrl+[	Ctrl+Alt+- Ctrl+Alt +Numpad-	editor.foldRecursively
展开（展开）所有子区域	Ctrl+K Ctrl+]	Ctrl+Alt+= Ctrl+Alt +Numpad+	editor.unfoldRecursively
折叠（折叠）所有区域	Ctrl+K Ctrl+0	Ctrl+Shift+- Ctrl+Shift +Numpad-	editor.foldAll
展开（展开）所有区域	Ctrl+K Ctrl+J	Ctrl+Shift+= Ctrl+Shift +Numpad+	editor.unfoldAll
添加行注释	Ctrl+K Ctrl+C	Ctrl+K Ctrl +C	editor.action.addCommentLine
删除行注释	Ctrl+K Ctrl+U	Ctrl+K Ctrl +U	editor.action.removeCommentLine
切换行注释	Ctrl+/	Ctrl+/ Ctrl +Numpad/	editor.action.commentLine
切换块评论	Shift+Alt+A	Ctrl+Shift+/ Ctrl+Shift +Numpad/	editor.action.blockComment
寻找	Ctrl+F	Ctrl+F	actions.find
代替	Ctrl+H	Ctrl+R	editor.action.startFindReplaceAction
找下一个	F3 Enter	F3	editor.action.nextMatchFindAction
查找上一个	Shift+F3 Shift+Enter	Shift+F3	editor.action.previousMatchFindAction
选择所有出现的查找匹配项	Alt+Enter	Alt+Enter	editor.action.selectAllMatches
Toggle查找区分大小写	Alt+C	Alt+C	toggleFindCaseSensitive
Toggle查找正则表达式	Alt+R	Alt+R	toggleFindRegex

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
Toggle查找整个单词	Alt+W	Alt+W	toggleFindWholeWord
切换自动换行	Alt+Z	Alt+Z	editor.action.toggleWordWrap

13.4.2 编码辅助

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	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+Q	editor.action.showHover
向侧面开放定义	Ctrl+K F12	Ctrl+K F12	editor.action.revealDefinitionAside
快速解决	Ctrl+.	Alt+Enter	editor.action.quickFix
扩大选择	Shift+Alt+Right	Ctrl+W	editor.action.smartSelect.expand
缩小选择	Shift+Alt+Left	Ctrl+Shift+W	editor.action.smartSelect.shrink

13.4.3 搜索

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
显示搜索	Ctrl+Shift+F	Ctrl+Shift+F	omnisearch.open.file

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
再次搜索	Ctrl+Shift+R	Ctrl+Shift+R	rerunSearchEditorSearch
删除文件结果	Ctrl+Shift +Backspace	Ctrl+Shift +Backspace	search.searchEditor.action.deleteFileResults

13.4.4 导航

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+N	workbench.action.showAllSymbols

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
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+Alt+N Ctrl+F12	workbench.action.gotoSymbol
转到定义	F12	F12	editor.action.revealDefinition
前往声明	--	Alt+F11 F4 Ctrl+Enter Ctrl+B	editor.action.goToDeclaration
前往实施	Ctrl+F12	Ctrl+Alt+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	Ctrl+Shift+Alt+F12	Ctrl+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	F8	F2	editor.action.marker.next
转到上一个错误或警告	Shift+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

13.4.5 重构

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

13.4.6 调试

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

13.4.7 版本控制

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
阶段选定的范围	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.stageSelectedRanges
恢复选定范围	Ctrl+K Ctrl+R	Ctrl+Alt+Z	git.stageSelectedRanges

13.4.8 编辑器/窗口管理

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

13.4.9 文件管理

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
新文件	Ctrl+N	Alt+Insert	workbench.action.files.newUntitledFile
打开文件	Ctrl+O	Ctrl+O	workbench.action.files.openFile
节省	Ctrl+S	Ctrl+S	workbench.action.files.save
保存全部	Ctrl+K S	Ctrl+K S	saveAll
另存为	Ctrl+Shift+S	Ctrl+Shift+S	workbench.action.files.saveAs
关闭	Ctrl+W Ctrl+F4	Ctrl+F4	workbench.action.closeActiveEditor
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
保持开放	Ctrl+K Enter	Ctrl+K Enter	workbench.action.keepEditor
复制活动文件的路径	Shift+Alt+C	Ctrl+Shift+C	workbench.action.files.copyFilePath
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
比较打开的文件与	--	Ctrl+D	workbench.files.action.compareFileWith

13.4.10 显示

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
重置缩放	Ctrl+Numpad0	Ctrl+Numpad0	workbench.action.zoomReset
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

13.4.11 首选项

命令	键（CodeArts IDE键盘映射）	键（IDEA键盘映射）	命令ID
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