使用Python构建终端应用程序探究

摘要

本文探讨了使用Python语言开发一个终端版本的Code Runner。该应用程序旨在解析指定代码文件的信息，并基于文件类型编译或打开解释器以实现运行。通过整合 `argparse`、`json`、 `subprocess` 和 `pathlib` 等标准库模块，以及利用 `rich` 库提供丰富的控制台输出体验，本应用展示了如何在终端环境中实现复杂功能的同时保持用户友好性。

引言

终端应用程序在自动化任务、数据处理和系统管理中扮演着重要角色。Python因其简洁的语法和强大的库支持，成为开发此类工具的理想选择。本文描述的应用程序结合了文件系统操作、命令行参数解析、JSON配置文件读取以及子进程执行，以实现对不同文件类型的智能处理。

应用程序结构与实现

准备工作

通过导入标准库 `argparse` 以实现命令行参数解析、 `json` 以读取配置文件、 `subprocess` 以实现shell命令执行、 `pathlib` 以实现文件解析

并且安装、导入 `rich` 以提供丰富的命令行输出体验

```
import argparse  # 用于解析命令行参数
import json      # 用于处理JSON配置文件
import subprocess  # 用于执行外部命令
import sys         # 提供访问和使用解释器的变量和功能的接口
from pathlib import Path  # 提供了面向对象的文件和目录路径接口

from rich.console import Console  # 用于控制台输出的库
from rich.markdown import Markdown  # 将Markdown文本渲染到控制台
from rich.table import Table  # 用于创建表格输出
```

参数解析与配置读取

应用程序首先使用 `argparse` 模块解析命令行参数，允许用户指定要处理的文件及是否启用调试模式。调试模式下，程序会详细输出文件解析和命令执行过程中的内部状态，便于开发和故障排查。

```
# 创建控制台对象，用于输出
console = Console(color_system="256", style=None)

# 解析命令行参数
parser = argparse.ArgumentParser()
parser.add_argument("file", type=str)
parser.add_argument("-d", "--debug", action="store_true")  # 启用调试模式的选项
args = parser.parse_args()
```

此外，通过 `json` 模块读取 `filetype.json` 配置文件，其中定义了不同文件类型的处理规则，包括文件扩展名与执行命令的映射。这使得应用程序能够灵活地扩展支持的文件类型和相关操作。

```
# 从配置文件加载文件类型及其关联的命令
with open("filetype.json") as configure_file:
    config = json.load(configure_file)

# 如果启用了调试模式，更新配置
if args.debug:
    config["debug"] = True
```

配置文件如下：

```
{
    "debug": false,
    "filetype": {
        "python": {
            "extension": ["py","pyw"],
            "command": [
                "python -u \".\\$f\""
            ]
        },
        "c++": {
            "extension": ["cpp","cxx"
            ],
            "command": [
                "clang++ \".\\$f\" -o \".\\$n.exe\"",
                ".\\\"$n.exe\""
            ]
        },
        "javascript": {
            "extension": ["js"],
            "command": ["node \".\\$f\""]
        }
```

```
    }
}
```

文件解析与信息展示

`RunnerApp`类的核心功能之一是解析文件信息。它使用`pathlib.Path`对象来获取文件的元数据，如文件名、扩展名和路径，并将这些信息以表格形式展示给用户，这得益于`rich.Table`提供的格式化能力。在调试模式下，这些信息对于验证输入和理解程序行为至关重要。

通过`pathlib.Path`对象来获取文件的元数据：

```
self.file_source = str(file.resolve())  # 文件的绝对路径
self.file = file.name  # 文件名
self.file_name = file.stem  # 文件名（不包含扩展名）
self.file_type = file.suffix[1:]  # 文件扩展名
self.file_dir = str(file.parent)  # 文件所在目录的路径
self.exec_cmd = []  # 存储要执行的命令列表
```

命令执行与反馈

`ExecCode`方法负责根据文件类型确定并执行适当的命令。它遍历配置文件中的规则，查找匹配的文件类型，并替换命令模板中的占位符（如文件名或路径）。通过`ReplaceSymbol`函数，应用程序实现了更安全的字符串替换机制，防止了潜在的语法错误。

替换机制：

> 配合传入的字典`replace`，逐个读取字符，如果遇到`$`则检测下一个字符，如果仍为`$`则将这两个字符替换为一个`$`，如果下一个字符在字典中则替换为对应内容，否则报错并停止程序
> 例如：
>
> ```
> ReplaceSymbol("$$$$$f$$"{"f":"a.py"})
> ```
>
> 返回值：`$$a.py$`

```
# 定义一个替换特殊符号的函数，用于命令字符串的动态生成
def ReplaceSymbol(text: str, replace: dict) -> str:
    returnTxt = ""
    i = 0
    while i < len(text):
        # 检查是否有双$符号，这表示一个实际的美元符号而非占位符
        if text[i] == "$":
            if i + 1 < len(text) and text[i + 1] == "$":
```

```
                returnTxt += "$"
                i += 2
            # 如果下一个字符是一个预定义的键，则进行替换
            elif i + 1 < len(text) and text[i + 1] in replace:
                returnTxt += replace[text[i + 1]]
                i += 2
            else:
                # 错误处理：未知的替换符号
                console.print(text[: i - 1], end="")
                console.print(text[i : i + 2], style="red", end="")
                console.print(text[i + 2 :])
                console.print(
                    "Syntax ERROR",
                    style="white on red",
                )
                sys.exit()
        else:
            returnTxt += text[i]
            i += 1
    return returnTxt
```

根据设置找到该文件类型对应命令模板：

```
for i in fileConfig:
    for j in fileConfig[i]["extension"]:
        if j == self.file_type:
            self.exec_cmd = fileConfig[i]["command"]
            break
```

逐个替换模板内容：

```
# 如果找到匹配的命令，添加到表格中
    self.commandTable.add_column("")
    self.commandTable.add_column("[bold]Command")
    if self.exec_cmd:
        for i in range(len(self.exec_cmd)):
            # 替换命令中的占位符为实际值
            cmd = ReplaceSymbol(
                cmd,
                {
                    "f": self.file,
                    "n": self.file_name,
                    "t": self.file_type,
                    "p": self.file_dir,
                    "d": self.file_dir,
                },
            )
            self.commandTable.add_row(str(i + 1), cmd)
```

```
            self.exec_cmd[i] = cmd
        if debugMode:
            console.print(self.commandTable)
```

命令执行过程中，`rich.Console`用于在控制台上以美观的方式呈现每个命令，同时`subprocess.run`确保命令在正确的目录下运行，提供了与操作系统交互的能力。

```
# 实际执行命令
for i in self.exec_cmd:
    console.rule()
    console.print(">", i, style="cyan")
    subprocess.run(i, cwd=self.file_dir, shell=True)
```

对于未找到命令：

```
console.print(
              "No execution command found for file type: .",
              self.file_type,
              style="white on red",
            )
```

总而言之

项目地址

https://github.com/I-AM-A-NOOB/Code-Runner/

完整实现

```
import argparse  # 用于解析命令行参数
import json  # 用于处理JSON配置文件
import subprocess  # 用于执行外部命令
import sys  # 提供访问和使用解释器的变量和功能的接口
from pathlib import Path  # 提供了面向对象的文件和目录路径接口

from rich.console import Console  # 用于控制台输出的库
from rich.markdown import Markdown  # 将Markdown文本渲染到控制台
from rich.table import Table  # 用于创建表格输出


# 定义一个替换特殊符号的函数，用于命令字符串的动态生成
def ReplaceSymbol(text: str, replace: dict) -> str:
    returnTxt = ""
    i = 0
```

```
    while i < len(text):
        # 检查是否有双$符号，这表示一个实际的美元符号而非占位符
        if text[i] == "$":
            if i + 1 < len(text) and text[i + 1] == "$":
                returnTxt += "$"
                i += 2
            # 如果下一个字符是一个预定义的键，则进行替换
            elif i + 1 < len(text) and text[i + 1] in replace:
                returnTxt += replace[text[i + 1]]
                i += 2
            else:
                # 错误处理：未知的替换符号
                console.print(text[: i - 1], end="")
                console.print(text[i : i + 2], style="red", end="")
                console.print(text[i + 2 :])
                console.print(
                    "Syntax ERROR",
                    style="white on red",
                )
                sys.exit()
        else:
            returnTxt += text[i]
            i += 1
    return returnTxt


# 主应用程序类
class RunnerApp:
    def __init__(self) -> None:
        self.fileTable = Table()  # 表格用于显示文件信息
        self.commandTable = Table()  # 表格用于显示执行的命令

    # 解析文件信息并添加到表格中
    def ParseFile(self, debugMode: bool, file: Path) -> None:
        self.file_source = str(file.resolve())  # 文件的绝对路径
        self.file = file.name  # 文件名
        self.file_name = file.stem  # 文件名（不包含扩展名）
        self.file_type = file.suffix[1:]  # 文件扩展名
        self.file_dir = str(file.parent)  # 文件所在目录的路径
        self.exec_cmd = []  # 存储要执行的命令列表

        # 如果处于调试模式，添加文件信息到表格
        if debugMode:
            self.fileTable.add_column("[bold]Item")
            self.fileTable.add_column("[red]Value")
            self.fileTable.add_row("File source", self.file_source)
            self.fileTable.add_row("File", self.file)
            self.fileTable.add_row("File name", self.file_name)
            self.fileTable.add_row("File type", self.file_type)
            self.fileTable.add_row("File dir", self.file_dir)
```

```
            console.print(self.fileTable)

    # 根据文件类型执行相应的命令
    def ExecCode(self, debugMode: bool, fileConfig: dict) -> None:
        for i in fileConfig:
            for j in fileConfig[i]["extension"]:
                if j == self.file_type:
                    self.exec_cmd = fileConfig[i]["command"]
                    break

        # 如果找到匹配的命令，添加到表格中
        self.commandTable.add_column("")
        self.commandTable.add_column("[bold]Command")
        if self.exec_cmd:
            for i in range(len(self.exec_cmd)):
                # 替换命令中的占位符为实际值
                cmd = ReplaceSymbol(
                    cmd,
                    {
                        "f": self.file,
                        "n": self.file_name,
                        "t": self.file_type,
                        "p": self.file_dir,
                        "d": self.file_dir,
                    },
                )
                self.commandTable.add_row(str(i + 1), cmd)
                self.exec_cmd[i] = cmd
            if debugMode:
                console.print(self.commandTable)

            # 实际执行命令
            for i in self.exec_cmd:
                console.rule()
                console.print(">", i, style="cyan")
                subprocess.run(i, cwd=self.file_dir, shell=True)
        else:
            # 如果没有找到匹配的命令，输出错误信息
            console.print(
                "No execution command found for file type: .",
                self.file_type,
                style="white on red",
            )


# 创建控制台对象，用于输出
console = Console(color_system="256", style=None)

# 解析命令行参数
parser = argparse.ArgumentParser()
```

```
parser.add_argument("file", type=str)
parser.add_argument("-d", "--debug", action="store_true")  # 启用调试模式的选项
args = parser.parse_args()

# 从配置文件加载文件类型及其关联的命令
with open("filetype.json") as configure_file:
    config = json.load(configure_file)

# 如果启用了调试模式，更新配置
if args.debug:
    config["debug"] = True

# 初始化应用程序实例
App = RunnerApp()
file = Path(args.file)
if file.exists():
    # 如果文件存在，开始解析和执行
    App.ParseFile(config["debug"], file)
    App.ExecCode(config["debug"], config["filetype"])
else:
    # 如果文件不存在，输出错误信息
    console.print('"%s" is NOT a valid file path.' % args.file, style="white on red")
```

执行效果

```
> python main.py --debug .\a.py
┏━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ Item        ┃ Value                                 ┃
┡━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│ File source │ D:\Projects\Project\code runner\a.py │
│ File        │ a.py                                  │
│ File name   │ a                                     │
│ File type   │ py                                    │
│ File dir    │ .                                     │
└─────────────┴───────────────────────────────────────┘
┏━━━┳━━━━━━━━━━━━━━━━━━━━━━┓
┃   ┃ Command              ┃
┡━━━╇━━━━━━━━━━━━━━━━━━━━━━┩
│ 1 │ python -u ".\a.py"   │
└───┴──────────────────────┘
────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────
> python -u ".\a.py"
Hello World
```

结论

本文介绍的终端应用程序是一个综合案例，展示了Python如何有效处理文件、执行系统命令并提供用户友好的界面。通过使用标准库和第三方库的组合，该应用程序不仅实现了预期的功能，还确保了代码的可维护性和可扩展性。这一实践对于那些希望深入了解Python终端应用开发的开发者而言，提供了一个实用的参考。

参考文献

- Rich
  https://rich.readthedocs.io/en/stable/
- Pathlib
  https://docs.python.org/3/library/pathlib.html
- Argparse
  https://docs.python.org/3/library/argparse.html
- Subprocess
  https://docs.python.org/3/library/subprocess.html
- Json
  https://docs.python.org/3/library/json.html