□□□□ □□□□□□□□□□□□□ का विश्लेषण

नीचे एक □□□□□□ स्क्रिप्ट है जो एक प्रोजेक्ट में सभी `.java` फाइलों को स्कैन करता है, उनके डिपेंडेंसेज़ को इम्पोर्ट स्टेटमेंट्स के आधार पर विश्लेषण करता है, और एक "डिपेंडेंसी टेक्स्ट"के रूप में एक डिपेंडेंसी ग्राफ़ को □□□ फॉर्मेट में आउटपुट करता है। इस स्क्रिप्ट का ध्यान प्रोजेक्ट के भीतर क्लासों के बीच के आंतरिक डिपेंडेंसेज़ पर केंद्रित है, जो ग्राफविज़ जैसी टूल्स के साथ दृश्यमान हो सकता है।

स्क्रिप्ट समीक्षा

स्क्रिप्ट: 1. **प्रोजेक्ट डायरेक्टरी** को रिकर्सिव रूप से स्कैन करता है ताकि सभी `.java` फाइलों को ढूंढ सके। 2. **डिपेंडेंसेज़** का विश्लेषण करता है, प्रत्येक `.java` फाइल से पैकेज घोषणाओं और इम्पोर्ट स्टेटमेंट्स को निकालकर। 3. **एक डिपेंडेंसी टेक्स्ट** को □□□ फॉर्मेट में आउटपुट करता है, जहां एक क्लास दूसरे क्लास को प्रोजेक्ट के भीतर इम्पोर्ट करता है, दिशा-प्रवणित एजेज़ दिखाता है।

यहाँ पूरी स्क्रिप्ट है:

```python
import os
import sys
import re
from collections import defaultdict

def get_package(file_path):
    """
      .java

    Args:
        file_path (str): .java

    Returns:
        str:      ,              None.
    """
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                match = re.search(r'^\s*package\s+([\w.]+);', line)
                if match:
                    return match.group(1)
    except Exception as e:
        print(f"    : {file_path}          : {e}")
    return None

def get_specific_imports(file_path):
    """
```

```python
    .java                    ,

    Args:
        file_path (str): .java

    Returns:
        list:
    """
    imports = []
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                match = re.search(r'^\s*import\s+([\w.]+);', line)
                if match:
                    imp = match.group(1)
                    #           ( , import java.util.*;)
                    if not imp.endswith('.*'):
                        imports.append(imp)
    except Exception as e:
        print(f"   : {file_path}          : {e}")
    return imports

if __name__ == '__main__':
    #   -
    if len(sys.argv) != 2:
        print("   : python script.py <root_directory>")
        sys.exit(1)

    root_dir = sys.argv[1]
    all_classes = set()

    #      :
    for root, dirs, files in os.walk(root_dir):
        for file in files:
            if file.endswith('.java'):
                file_path = os.path.join(root, file)
                package = get_package(file_path)
                if package:
                    class_name = file.replace('.java', '')
                    full_class_name = f"{package}.{class_name}"
```

```
                all_classes.add(full_class_name)

    #            :     ->
    dependencies = defaultdict(set)

    #      :
    for root, dirs, files in os.walk(root_dir):
        for file in files:
            if file.endswith('.java'):
                file_path = os.path.join(root, file)
                package = get_package(file_path)
                if package:
                    class_name = file.replace('.java', '')
                    full_class_name = f"{package}.{class_name}"
                    imports = get_specific_imports(file_path)
                    for imp in imports:
                        #
                        #   -
                        if imp in all_classes and imp != full_class_name:
                            dependencies[full_class_name].add(imp)

    #           DOT
    print('digraph G {')
    for class_name in sorted(dependencies):
        for dep in sorted(dependencies[class_name]):
            print(f'  "{class_name}" -> "{dep}";')
    print('}')
```

यह कैसे काम करता है

1. कमांड-लाइन इनपुट

- स्क्रिप्ट एकल अर्ग्यूमेंट की अपेक्षा करता है: जावा प्रोजेक्ट का रूट डायरेक्टरी।
- उदाहरण उपयोग: `python script.py /path/to/project`
- अगर कोई अर्ग्यूमेंट प्रदान नहीं किया गया है, तो यह उपयोग निर्देशों को प्रिंट करता है और बाहर निकलता है।

2. `.java` फाइलों को ढूंढना

- `os.walk()` का उपयोग करके निर्दिष्ट डायरेक्टरी को रिकर्सिव रूप से ट्रैवर्स करता है और सभी `.java` से समाप्त होने वाले फाइलों को पहचानता है।

3. क्लास जानकारी निकालना

- **पैकेज निकालना**: `get_package` फंक्शन प्रत्येक `.java` फाइल को पढ़ता है और एक रेगुलर एक्सप्रेशन (`^\s*package\s+([\w.]+);`) का उपयोग करके पैकेज घोषणा (जैसे, `package com.mycompany.myproject;`) को ढूंढता है।
  - अगर कोई पैकेज नहीं मिला या फाइल पढ़ी नहीं जा सकी, तो `None` लौटाता है।
- **क्लास नाम**: मानता है कि क्लास नाम फाइल नाम के साथ मिलता है (जैसे, `MyClass.java` `MyClass` को परिभाषित करता है).
- **पूर्ण रूप से क्वालीफाइड नाम**: पैकेज और क्लास नाम को मिलाकर (जैसे, `com.mycompany.myproject.MyClass`).

4. सभी क्लासों को इकट्ठा करना

- पहले पास में, प्रोजेक्ट में सभी पूर्ण रूप से क्वालीफाइड क्लास नामों की एक सेट बनाता है ताकि बाद में जल्दी से लुकअप किया जा सके।

5. डिपेंडेंसेज़ का विश्लेषण

- **इम्पोर्ट निकालना**: `get_specific_imports` फंक्शन एक रेगुलर एक्सप्रेशन (`^\s*import\s+([\w.]+);`) का उपयोग करके इम्पोर्ट स्टेटमेंट्स निकालता है, वाइल्डकार्ड इम्पोर्ट (जैसे, `import java.util.*;`) को छोड़कर।
  - उदाहरण: `import com.mycompany.myproject.utils.Helper;` से, यह `com.mycompany.myproject.utils.Helper` निकालता है।
- **डिपेंडेंसी मैपिंग**: प्रत्येक `.java` फाइल के लिए:
  - इसका पूर्ण रूप से क्वालीफाइड क्लास नाम प्राप्त करता है।
  - इसके विशेष इम्पोर्ट्स की जांच करता है।
  - अगर एक इम्पोर्ट किया गया क्लास प्रोजेक्ट के क्लास सेट में है और क्लास स्वयं नहीं है, तो एक डिपेंडेंसी जोड़ता है।

6. डिपेंडेंसी टेक्स्ट आउटपुट करना

- एक दिशा-प्रवणित ग्राफ़ को DOT फॉर्मेट में आउटपुट करता है:
  - `digraph G {` से शुरू होता है।
  - प्रत्येक क्लास के साथ डिपेंडेंसेज़ के लिए एजेज़ जैसे `"ClassA" -> "ClassB";` प्रिंट करता है।
  - `}` से समाप्त होता है।
- क्लास और डिपेंडेंसेज़ को सॉर्ट किया जाता है ताकि आउटपुट सुसंगत हो सके।
- उदाहरण आउटपुट:

```
digraph G {
  "com.mycompany.myproject.ClassA" -> "com.mycompany.myproject.utils.Helper";
  "com.mycompany.myproject.ClassB" -> "com.mycompany.myproject.ClassA";
}
```

उपयोग उदाहरण

1. स्क्रिप्ट को `analyze_deps.py` के रूप में सेभ करें।
2. इसे चलाएं:

   ```
   python analyze_deps.py /path/to/java/project
   ```

3. आउटपुट को एक फाइल में रेडाइरेक्ट करें:

   ```
   python analyze_deps.py /path/to/java/project > dependencies.dot
   ```

4. ग्राफविज़ के साथ दृश्यमान करें:

   ```
   dot -Tpng dependencies.dot -o dependencies.png
   ```

   यह एक □□□ छवि बनाता है जो डिपेंडेंसी ग्राफ़ दिखाता है।

अनुमान और सीमाएं

- **एक पब्लिक क्लास प्रति फाइल**: मानता है कि प्रत्येक `.java` फाइल में एक पब्लिक क्लास है जिसका नाम फाइल के नाम के साथ मिलता है, जैसा कि जावा परंपरा है।
- **इम्पोर्ट-आधारित डिपेंडेंसेज़**: केवल विशेष इम्पोर्ट स्टेटमेंट्स द्वारा इंगित डिपेंडेंसेज़ को ही माना जाता है (जैसे, `import com.example.SomeClass;`), वाइल्डकार्ड इम्पोर्ट (`import com.example.*;`) और उसी पैकेज के संदर्भों को छोड़कर (जिन्हें इम्पोर्ट की आवश्यकता नहीं होती है).
- **आंतरिक ध्यान**: केवल प्रोजेक्ट के भीतर क्लासों के बीच डिपेंडेंसेज़ को ट्रैक करता है, बाहरी लाइब्रेरियों को छोड़कर।
- **बुनियादी पार्सिंग**: सरलता के लिए रेगुलर एक्सप्रेशन का उपयोग करता है, जो किन्हीं किन्हीं एज केस (जैसे, टिप्पणियों के भीतर इम्पोर्ट) को छोड़ सकता है। एक पूर्ण जावा पार्सर अधिक सक्षम होगा लेकिन जटिल होगा।

कस्टमाइजेशन

- **आउटपुट फॉर्मेट**: प्रिंट स्टेटमेंट्स को बदलकर डिपेंडेंसी टेक्स्ट फॉर्मेट को बदल सकते हैं (जैसे, एक साधारण सूची के बजाय □□□).
- **एरर हैंडलिंग**: अनपढ़ फाइलों को नजरअंदाज करने के लिए □□□-□□□□□□ ब्लॉक्स के साथ बेहतर बनाया गया है।
- **स्कोप**: बाहरी डिपेंडेंसेज़ या उसी पैकेज के डिपेंडेंसेज़ को शामिल करने के लिए गहन कोड विश्लेषण के साथ बढ़ाया जा सकता है।

इस स्क्रिप्ट प्रोजेक्ट के आंतरिक क्लास संबंधों के माध्यम से जावा प्रोजेक्ट के संरचना को समझने के लिए डिपेंडेंसी विश्लेषण के लिए एक व्यावहारिक शुरुआत प्रदान करता है।