□□□□□ □□□□ को विश्लेषित करके निर्भरता ग्राफ बनाएं

आपके प्रश्न का जवाब देने के लिए, मैं एक समाधान प्रदान करूँगा जो □□□ प्रारूप में एक निर्भरता ग्राफ उत्पन्न करने के लिए एक निर्देशिका में सभी `pom.xml` फ़ाइलें पुनरावृत्ती से खोजने और उनके निर्भरताओं का विश्लेषण करने का तरीका बताता है, बिना किसी निश्चित □□□□□ बहु-मॉड्यूल प्रोजेक्ट संरचना जैसे उदाहरण में (`project/module1/pom.xml`, आदि) का अनुमान लगाया जाए।

समाधान सारांश

यहाँ स्क्रिप्ट क्या करेगा:

1. **पुनरावृत्ती से `pom.xml` फ़ाइलें एकत्र करें**: निर्दिष्ट मूल निर्देशिका को ट्रैवर्स करें ताकि सभी `pom.xml` फ़ाइलें, उनकी स्थिति या संरचना के बिना, खोजें।
2. **मॉड्यूल जानकारी को पर्स करें**: प्रत्येक `pom.xml` से `groupId`, `artifactId`, और निर्भरताओं को निकालें।
3. `groupId` **विरासत को संभालें**: यदि एक `pom.xml` में `groupId` स्पष्ट नहीं है, तो एक `<parent>` घोषणा से विरासत ग्रहण करें, यदि मौजूद है।
4. **इंट्रा-प्रोजेक्ट निर्भरताओं को पहचानें**: `groupId` और `artifactId` के आधार पर प्रोजेक्ट के भीतर मॉड्यूलों के निर्भरताओं को मैच करें।
5. □□□ **आउटपुट उत्पन्न करें**: मॉड्यूलों के बीच निर्भरताओं का एक दिशा-निर्दिष्ट ग्राफ बनाएं।

यह दृष्टिकोण किसी पूर्व-निर्धारित निर्देशिका हाइरार्की पर निर्भर नहीं करता और यह मानता नहीं है कि माता-पिता `pom.xml` फ़ाइलें माता-पिता निर्देशिकाओं में होंगी।

पाइथन स्क्रिप्ट

नीचे आपके आवश्यकताओं के अनुसार तैयार पाइथन स्क्रिप्ट है:

```
import os
import sys
import xml.etree.ElementTree as ET

# Maven POM
NS = "{http://maven.apache.org/POM/4.0.0}"

# groupId
group_id_cache = {}

def get_group_id(pom_path, pom_map):
    """
```

```
  pom.xml      groupId      ,    -

Args:
    pom_path (str): pom.xml        .
    pom_map (dict): pom.xml                        .

Returns:
    str:       groupId.
"""
if pom_path in group_id_cache:
    return group_id_cache[pom_path]

tree = ET.parse(pom_path)
root = tree.getroot()
group_id_elem = root.find(NS + 'groupId')

if group_id_elem is not None:
    group_id = group_id_elem.text.strip()
else:
    #   -
    parent = root.find(NS + 'parent')
    if parent is not None:
        parent_group_id = parent.find(NS + 'groupId').text.strip()
        parent_artifact_id = parent.find(NS + 'artifactId').text.strip()
        parent_relative_path = parent.find(NS + 'relativePath')
        if parent_relative_path is not None and parent_relative_path.text:
            parent_pom_path = os.path.normpath(
                os.path.join(os.path.dirname(pom_path), parent_relative_path.text)
            )
        else:
            # relativePath           -
            parent_pom_path = os.path.join(os.path.dirname(pom_path), '..', 'pom.xml')
            parent_pom_path = os.path.normpath(parent_pom_path)

        if parent_pom_path in pom_map:
            group_id = get_group_id(parent_pom_path, pom_map)
        else:
            raise ValueError(f"{pom_path}        -   POM      : {parent_pom_path}")
    else:
        raise ValueError(f"{pom_path}     groupId     -          ")
```

```
    group_id_cache[pom_path] = group_id
    return group_id

def get_artifact_id(pom_path):
    """
      pom.xml      artifactId

    Args:
        pom_path (str): pom.xml        .

    Returns:
        str:       artifactId.
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    artifact_id_elem = root.find(NS + 'artifactId')

    if artifact_id_elem is None:
        raise ValueError(f"pom.xml   artifactId          : {pom_path}")

    return artifact_id_elem.text.strip()

def get_dependencies(pom_path):
    """
      pom.xml

    Args:
        pom_path (str): pom.xml        .

    Returns:
        list:              (groupId, artifactId)        .
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    dependencies = []

    for dep in root.findall(NS + 'dependencies/' + NS + 'dependency'):
        dep_group_id_elem = dep.find(NS + 'groupId')
        dep_artifact_id_elem = dep.find(NS + 'artifactId')
```

```
        if dep_group_id_elem is not None and dep_artifact_id_elem is not None:
            dep_group_id = dep_group_id_elem.text.strip()
            dep_artifact_id = dep_artifact_id_elem.text.strip()
            dependencies.append((dep_group_id, dep_artifact_id))

    return dependencies

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

    root_dir = sys.argv[1]
    if not os.path.isdir(root_dir):
        print(f"  : {root_dir}           ")
        sys.exit(1)

    #    1:          pom.xml
    pom_files = [
        os.path.join(root, file)
        for root, _, files in os.walk(root_dir)
        for file in files if file == 'pom.xml'
    ]

    if not pom_files:
        print(f"{root_dir}     pom.xml         ")
        sys.exit(1)

    #    2:    POMs               -
    pom_map = {pom_file: None for pom_file in pom_files}

    #    3:
    modules = {}  # (groupId, artifactId) -> pom_path
    for pom_file in pom_files:
        try:
            group_id = get_group_id(pom_file, pom_map)
            artifact_id = get_artifact_id(pom_file)
            modules[(group_id, artifact_id)] = pom_file
        except ValueError as e:
```

```
        print(f"    :          {pom_file}           : {e}")
        continue

    #    4:
    dependencies = set()
    for pom_file in pom_files:
        try:
            importer_group_id = get_group_id(pom_file, pom_map)
            importer_artifact_id = get_artifact_id(pom_file)
            importer_key = (importer_group_id, importer_artifact_id)
            deps = get_dependencies(pom_file)
            for dep_group_id, dep_artifact_id in deps:
                dep_key = (dep_group_id, dep_artifact_id)
                if dep_key in modules and dep_key != importer_key:
                    #        (importer, imported)           , artifactId
                    dependencies.add((importer_artifact_id, dep_artifact_id))
        except ValueError as e:
            print(f"    : {pom_file}                    : {e}")
            continue

    #    5: DOT
    print('digraph G {')
    for from_module, to_module in sorted(dependencies):
        print(f'  "{from_module}" -> "{to_module}";')
    print('}')
```

यह कैसे काम करता है

1. कमांड-लाइन इनपुट

- एकल तर्क लेता है: `<root_directory>`, पुनरावृत्ती खोज के लिए शुरुआती बिंदु।
- यह एक निर्देशिका है कि यह सत्यापित करता है।

2. `pom.xml` फ़ाइलें खोजें

- `os.walk` का उपयोग करके निर्देशिका वृक्ष को पुनरावृत्ती से ट्रैवर्स करें और सभी `pom.xml` फ़ाइलें एक सूची में एकत्र करें।

3. मॉड्यूल जानकारी को पर्स करें

- **ग्रुप आईडी (`groupId`)**:
  - प्रत्येक `pom.xml` से निकाला जाता है।
  - यदि मौजूद नहीं है, तो एक `<parent>` खंड की जांच करें और संदर्भित माता-पिता से `groupId` को सुलझाएं, `relativePath` (या यदि छोड़ दिया गया है तो माता-पिता निर्देशिका पर डिफ़ॉल्ट करें) का उपयोग करके।
  - परिणामों को कैश करें ताकि पुनरावृत्ती से बचा जाए।
- **आर्टिफैक्ट आईडी (`artifactId`)**: प्रत्येक `pom.xml` में मौजूद होना चाहिए।
- **निर्भरताएं**: `<dependencies>` खंड से `(groupId, artifactId)` जोड़े निकालें।

4. निर्भरता विश्लेषण

- सभी मॉड्यूलों के लिए एक नक्शा बनाएं `(groupId, artifactId)` को `pom_path` के लिए।
- प्रत्येक `pom.xml` के लिए, इसे मॉड्यूल नक्शे के खिलाफ अपने निर्भरताओं की जांच करें ताकि प्रोजेक्ट के भीतर निर्भरताओं को पहचानें।
- स्वयं निर्भरताओं (जहां एक मॉड्यूल स्वयं पर निर्भर है) को छोड़ें।
- निर्भरताओं को `(importer_artifactId, imported_artifactId)` जोड़ों के रूप में रिकॉर्ड करें।

5. आउटपुट

- एक दिशा-निर्दिष्ट ग्राफ को प्रारूप में आउटपुट करें, सरलता के लिए `artifactId` को नोड लेबल के रूप में उपयोग करें।

उदाहरण उपयोग

अगर आपका निर्देशिका संरचना असामान्य है:

```
myproject/
  app/pom.xml (groupId="com.myapp", artifactId="app", "core"       )
  libs/core/pom.xml (groupId="com.myapp", artifactId="core")
  tools/util/pom.xml (groupId="com.myapp", artifactId="util", "core"       )
```

स्क्रिप्ट चलाएं:

```
python script.py myproject
```

आउटपुट:

```
digraph G {
  "app" -> "core";
  "util" -> "core";
}
```

यह दिखाता है कि दोनों app और util core पर निर्भर हैं।

टिप्पणियाँ और सीमाएँ

- **निर्देशिका द्वारा समूह नहीं**: मूल स्क्रिप्ट में वर्णित विचार ट्रेस के विपरीत, यह संस्करण निर्देशिका संरचना द्वारा मॉड्यूलों को समूहित नहीं करता क्योंकि आपने इसे नहीं मांगा। यह artifactId को ग्राफ में सीधे उपयोग करता है।
- **माता-पिता सुलझाना**: relativePath पर निर्भर करता है <parent> खंड में या माता-पिता निर्देशिका पर डिफ़ॉल्ट करता है यदि छोड़ दिया गया है। यदि माता-पिता □□□ एकत्रित सेट में नहीं है, तो यह एक त्रुटि उत्पन्न करता है।
- **सरल मैचिंग**: केवल groupId और artifactId के आधार पर निर्भरताओं को मैच करता है, संस्करणों को नजरअंदाज करता है।
- **गुणस्तर प्रतिस्थापन नहीं**: □□□□□ गुणों जैसे ${project.groupId} को संभालता नहीं है।
- **त्रुटि संचालन**: त्रुटियों वाली pom.xml फ़ाइलें एक चेतावनी के साथ छोड़ दी जाती हैं, पूरी तरह से विफल होने के बजाय।

यह स्क्रिप्ट आपके अनुरोध को पूरा करता है "पुनरावृत्ती से इटरेट करें, पहले सभी □□□□ को प्राप्त करें, और फिर विश्लेषण करें"बिना किसी निश्चित फ़ाइल संरचना का अनुमान लगाया जाए। मुझे बताएं अगर आपको किसी संशोधन की आवश्यकता है!