code$_{a}nalysis Documentation$

Release 0.1.5

Stefan Braun

Apr 10, 2021

CONTENTS:

CHAPTER

ONE

CODE_ANALYSIS

Analyze source code dependencies and call trees in Neo4j.

- Free software: MIT license
- Documentation: https://code-analysis.readthedocs.io.

1.1 Features

This project provides generators for Cypher code for import into Neo4J from call tree and package dependencies. Some of these generators rely on external tools to provide their ingoing data.

Static calltrees of Java code can be created with java-callgraph, which can be found on GitHub: https://github.com/gousiosg/java-callgraph.

Dependencies of Java packages can be determined using JDepend, which can also be found on GitHub: https://github.com/clarkware/jdepend.

Python dependencies are determined using the compiler and AST (Abstract Syntax Tree) with a tool provided in this project.

1.1.1 Generating Cypher for a Java call tree

Create the call tree using java-callgraph and save it into a file, e.g., *java_call_tree_input.txt*. Run:

```
java_call_tree java_call_tree_input.txt > calltree-cypher.txt
```

calltree-cypher.txt contains two Cypher statements, one to insert all classes into a Neo4j database, and another to insert the call relations on method level. You can just copy each statement and paste it into the Neo4j browser.

The database schema looks like this::

```
(:Class)-[:uses]->(:Class)
(:Method)-[:calls]->(:Method)
```

1.1.2 Generating Cypher code for Java dependencies

Create dependencies using JDepend and save it in a file, e.g., *java_depend.txt*.

Run following command::

```
java_dependencies java_depend.txt > java_depend.cypher
```

Now you can copy the Cypher statements stored in *java_depend.cypher* and paste it into the Neo4j browser.

The database schema looks like this::

```
(:Package)-[:depends_on]->(:Package)
```

To check for cycles you may run the query::

```
MATCH (p:Package)-[r1:depends_on]->(q:Package)-[r2:depends_on]->(p:Package)
RETURN p, q, r1, r2
```

It helps to switch off the default setting, which shows all relations, in the browser settings.

1.1.3 Generating Cypher code for Python dependencies

Determination of dependencies and generation of Cypher code are done in one step in this case::

```
python_dependencies <path to your package> > python-deps.cypher
```

The tool compiles the code and walks the AST looking for import statements. Then it generates Cypher code modelling the relationships between the packages.

The database schema looks like this::

```
(:Package)-[:contains]->(:Module)
(:Module)-[:uses]->(:Module)
```

1.2 Credits

This package was created with Cookiecutter and the stbraun/cookiecutter-pypackage project template based on *audreyr/cookiecutter-pypackage*.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install code_analysis, run this command in your terminal:

```
$ pip install code_analysis
```

This is the preferred method to install code_analysis, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for code_analysis can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/stbraun/code_analysis
```

Or download the tarball:

```
$ curl  -OL https://github.com/stbraun/code_analysis/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

USAGE

3.1 java_call_tree

To generate Cypher code first generate a call tree with java-callgraph. Xou can download it from GitHub: https://github.com/gousiosg/java-callgraph. Then feed the output of *java-callgraph* into *java_call_tree*::

```
java -jar javacg-0.1-SNAPSHOT-static.jar <your jar> <optional jars> > output.txt
java_call_tree output.txt > calltree.cypher
```

Now you can paste the content of *calltree.cypher* into the Neo4j browser of your database.

3.2 java_dependencies

First generate a dependency file with JDepend. You can download it from GitHub: https://github.com/clarkware/jdepend. Then feed the output into *java_dependencies*::

```
java jdepend.xmlui.JDepend -file jdepend_output.txt <path to Java project>
java_dependencies jdepend_output.txt > dependency.cypher
```

Now paste the content of *dependency.cypher* into the Neo4j browser to import your dependencies. You can add as many Java projects as you need.

3.3 python_dependencies

Just point *pyython_dependencies* to the package you want to analyze::

```
python_dependencies <some package> > dependencies.cypher
```

Now paste the content of *dependency.cypher* into the Neo4j browser to import your dependencies.

CHAPTER

FOUR

TECHNICAL SPECIFICATION

4.1 Java call trees

The tool for the generation of the raw call tree data, *java-callgraph*, writes its output as formatted text. The structure follows this schema::

```
C:<qualified class name> <qualified class name>
M:<qualified class name>:<method>(<params>) (x)<qualified class name>:<method>(
↪<params>)
```

java_call_tree uses the first character to distinguish classes and methods. The second qualifier, denoted as *x*, is not used. Parameters are also cut off.

An example of the input format can be found in the *resources* folder of this project.

4.2 Java dependencies

jdepend.xmlui.JDepend writes XML. *java_dependencies* uses only the subset related to dependencies. It looks like follows::

```
<JDepend>
    <Packages>
        <Package name = "some name" >
            <DependsUpon>
                <Package>qualified.package.name</Package>
                ...
                <Package>another.package</Package>
            </DependsUpon>
        </Package>
        ...
    </Packages>
</JDepend>
```

An example of the input format can be found in the *resources* folder of this project.

4.3 Python dependencies

Dependencies of Python packages are determined using the *ast* module of the Python standard library. Modules of the analyzed package are parsed into an AST (abstract syntax tree). Then the imports are extracted by walking the tree. In this case no intermediate file format is required, but the results of the tree walk are directly processed.

CHAPTER

FIVE

CODE_ANALYSIS

5.1 code_analysis package

5.1.1 Submodules

5.1.2 code_analysis.java_call_tree module

5.1.3 code_analysis.java_dependencies module

5.1.4 code_analysis.python_dependencies module

5.1.5 Module contents

Top-level package for code_analysis.

CHAPTER

SIX

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/stbraun/code_analysis/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

code_analysis could always use more documentation, whether as part of the official code_analysis docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/stbraun/code_analysis/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *code_analysis* for local development.

1. Fork the *code_analysis* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/code_analysis.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ cd code_analysis/
$ pipenv shell
$ pipenv install --dev
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with nox:

```
$ flake8 code_analysis tests
$ make test or pytest
$ nox
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/stbraun/code_analysis/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ py.test tests.test_code_analysis
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

SEVEN

HISTORY

7.1 0.1.0 (2021-04-04)

- First release on PyPI.

7.2 0.1.3 (2021-04-05)

- Documentation enhanced.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

C

M