

# 数据库构建

## 数据库构建

- 一、创建数据库
- 二、数据库目录结构
- 三、数据库生成原理
  - 1、分析数据库构建日志
  - 2、自动构建脚本
  - 3、Python tracer
  - 4、提取器
  - 5、trap文件
  - 6、rel文件

文档以Python举例，其他语言的思路应该是一致的，可能方法上略有不同：

主要流程大概是：数据库初始化 --> 提取器执行（通过AST生成trap文件） -- trap文件导入原始数据集 -- 生成最终database数据库

## 一、创建数据库

创建命令：

```
codeql database create codeql_security_test2 --language=python --source-root=D:/qy/codeql/codeql_learn
```

## 二、数据库目录结构

### CodeQL database

A database (or CodeQL database) is a directory containing:

- queryable data, extracted from the code.
  - a source reference, for displaying query results directly in the code.
  - query results.
  - log files generated during database creation, query execution, and other operations.
- 
- db-python：可查询数据集
  - log：创建和查询日志
  - results：查询结果的bqrs文件，用来转换为其他结果格式
  - src.zip：源代码

## 三、数据库生成原理

### 1、分析数据库构建日志

实际创建一个数据库，查看日志，大致分为一下三个阶段：

- 数据库初始化：  
创建空的codeql数据库

```

[2023-01-16 17:01:51] log file is started late.
[2023-01-16 17:01:51] [PROGRESS] database create> Initializing database at D:\qy\codeql\codeql_security_test4.
[2023-01-16 17:01:51] Running plumbing command: codeql database init --language-python --source-root=D:\qy\codeql\codeql_learn --allow-missing-source-root=false --allow-already-existing -- D:\qy\codeql\codeql_security_test4
[2023-01-16 17:01:51] calling plumbing command: codeql resolve-languages --format=json
[2023-01-16 17:01:51] [DETAILS] resolve languages> Scanning for [codeql-extractor.yml] from D:\SAST\tools\codeql\codeqlmanifest.json
[2023-01-16 17:01:51] [DETAILS] resolve languages> Parsing D:\SAST\tools\codeql\cpp\codeql-extractor.yml.
[2023-01-16 17:01:51] [DETAILS] resolve languages> Parsing D:\SAST\tools\codeql\csharp\codeql-extractor.yml.
[2023-01-16 17:01:51] [DETAILS] resolve languages> Parsing D:\SAST\tools\codeql\csharp\codeql-extractor.yml.

```

- 提取器执行：

```

118 [2023-01-16 17:01:52] Plumbing command codeql database init completed.
119 [2023-01-16 17:01:52] [PROGRESS] database create> Running build command: []
120 [2023-01-16 17:01:52] Running plumbing command: codeql database trace-command --working-dir=D:\qy\codeql\codeql_learn --index-traceless-dbs --no-db-cluster -- D:\qy\codeql\codeql_security_test4
121 [2023-01-16 17:01:52] Using autobuild script D:\SAST\tools\codeql\python\tools\autobuild.cmd.
122 [2023-01-16 17:01:52] [PROGRESS] database trace-command> Running command in D:\qy\codeql\codeql_learn: [D:\SAST\tools\codeql\python\tools\autobuild.cmd]
123 [2023-01-16 17:01:53] [build-stdout] python -3.10.9
124 [2023-01-16 17:01:53] [build-stderr] Installed Pythons found by py Launcher for Windows *
125 [2023-01-16 17:01:53] [build-stderr] Requested Python version (2) not installed, use -b for available pythons

```

从上图可以看到提取器执行trace-command命令，以及自动构建脚本autobuild.cmd，主要为提取源码，基于AST生成trap文件，下面几小节分析。

- 数据集导入：

```

765 [2023-01-16 17:02:57] [PROGRESS] database finalizes running now: import for Codeql database at D:\qy\codeql\codeql_security_test4...
766 [2023-01-16 17:02:57] Running plumbing command: codeql dataset import --dbscheme=D:\SAST\tools\codeql\python\semmlecode\python.dbscheme -- D:\qy\codeql\codeql_security_test4
767 [2023-01-16 17:02:57] Clearing disk cache since the version file D:\qy\codeql\codeql_security_test4\db-python\default\cache\version does not exist
768 [2023-01-16 17:02:57] Tuple pool not found. Clearing relations with cached strings
769 [2023-01-16 17:02:57] Trimming disk cache at D:\qy\codeql\codeql_security_test4\db-python\default\cache in mode brutal.
770 [2023-01-16 17:02:57] Sequence stamp origin is -6364712992313465275
771 [2023-01-16 17:02:57] Pausing evaluation to hard-clear memory at sequence stamp 0x0

```

将tarp文件导入最终数据集，也就是 db-python 目录

## 2、自动构建脚本

```

1 @echo off
2
3 rem Legacy environment variables for the autobuild infrastructure.
4 set LGTM_SRC=%CD%
5 set LGTM_WORKSPACE=%CODEQL_EXTRACTOR_PYTHON_SCRATCH_DIR%
6
7 type NUL && python "%CODEQL_EXTRACTOR_PYTHON_ROOT%\tools\index.py"
8 exit /b %ERRORLEVEL%
9

```

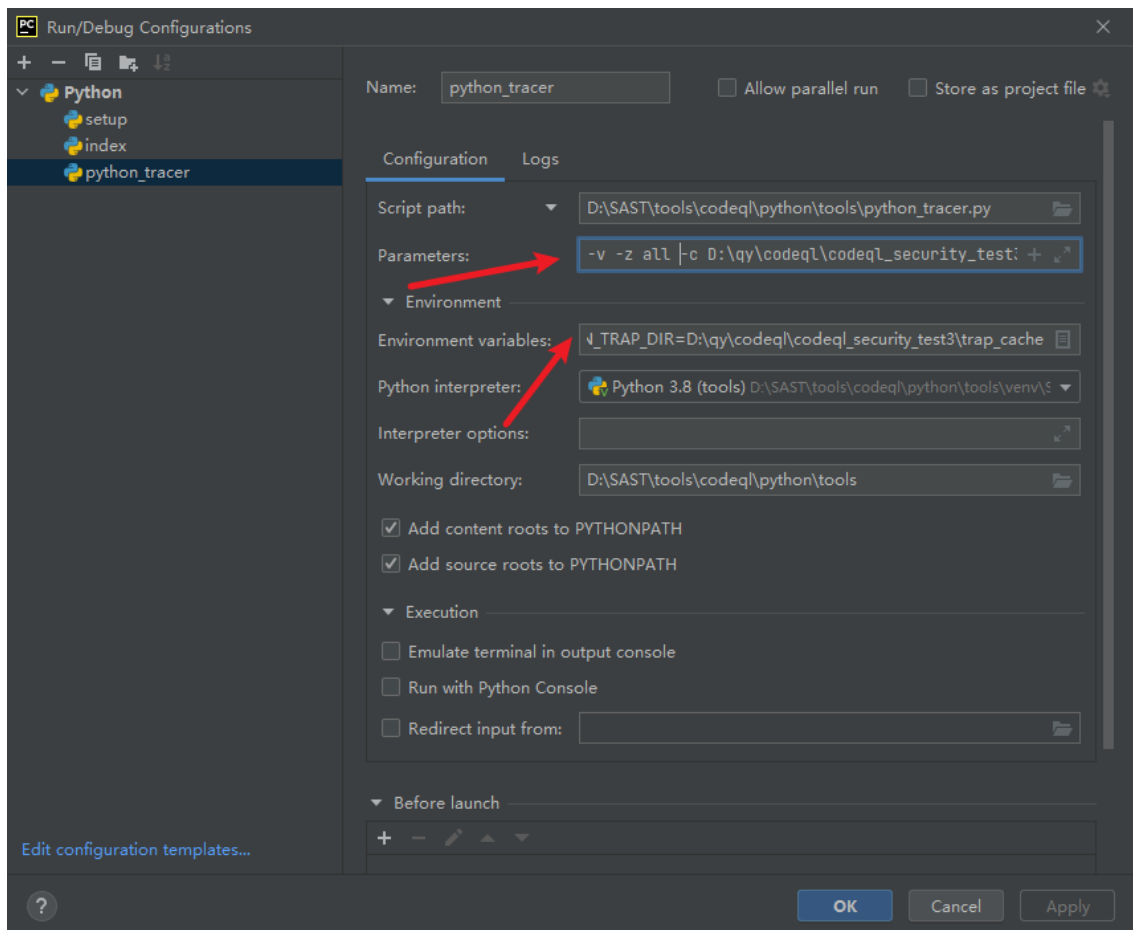
- 执行当前目录的index脚本
- index脚本调用

```

import buildtools.index
buildtools.index.main()

```

- 解压python3src.zip 得到包： buildtools
- 接下来可自行构建相关Pycharm调试环境，这一步不赘述
- 有一些环境变量需自行添加
- 进行调试会发现主要入口调用： `python_tracer.py`，相关参数自行调试获取
-



### 3、Python tracer

- 主要流程是：创建进程池+任务队列

```

70 class ExtractorPool(object):
71
72     def __init__(self, outdir, archive, proc_count, options, logger):
73         if (proc_count < 1):
74             raise ValueError('Number of processes must be at least one.')
75         self.verbose = options.verbose
76         self.outdir = outdir
77         self.max_import_depth = options.max_import_depth
78         method = ('spawn' if (sys.platform == 'darwin') else None)
79         try:
80             ctx = mp.get_context(method)
81         except AttributeError:
82             ctx = mp
83         self.module_queue = ctx.Queue((proc_count * 2))
84         self.reply_queue = ctx.Queue((proc_count * 20))
85         self.archive = archive
86         self.local_queue = deque()
87         self.enqueued = set()
88         self.done = set()
89         self.requirements = {}
90         self.import_graph = ModuleImportGraph(options.max_import_depth)
91         logger.debug('Source archive: %s', archive)
92         self.logger = logger
93         args = (self.module_queue, outdir, archive, options, self.reply_queue, logger)
94         self.procs = [ctx.Process(target=extract_loop, args=((n + 1),) + args + ((n == 0),)) for n in range(proc_count)]
95         for p in self.procs:
96             p.start()
97         self.start_time = time.time()

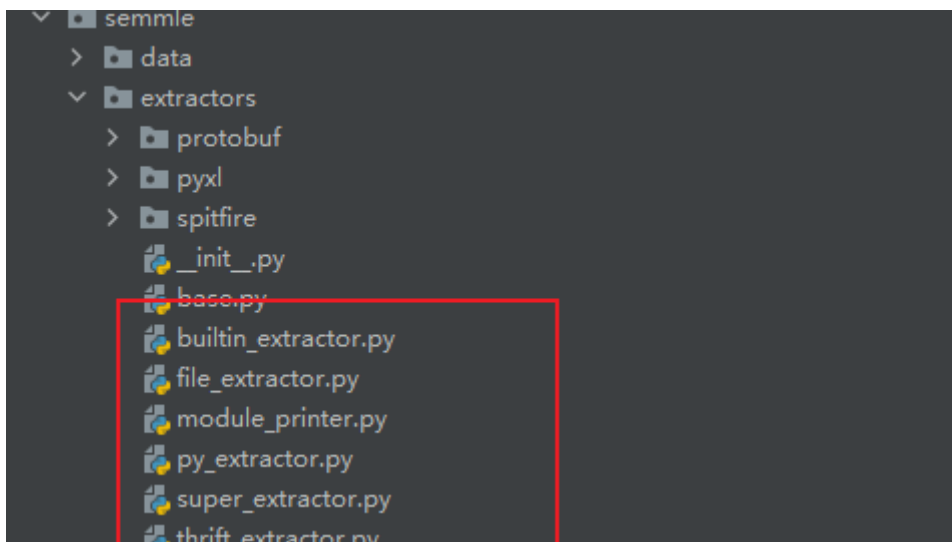
```

- 进程池调用提取器函数处理文件
- 遍历所有文件，逐个文件发给上述队列处理

### 4、提取器

做扫描引擎开发的可能感兴趣这一块

- 分为内建提取器、第三方包提取器、代码提取器、包提取器、文件提取器



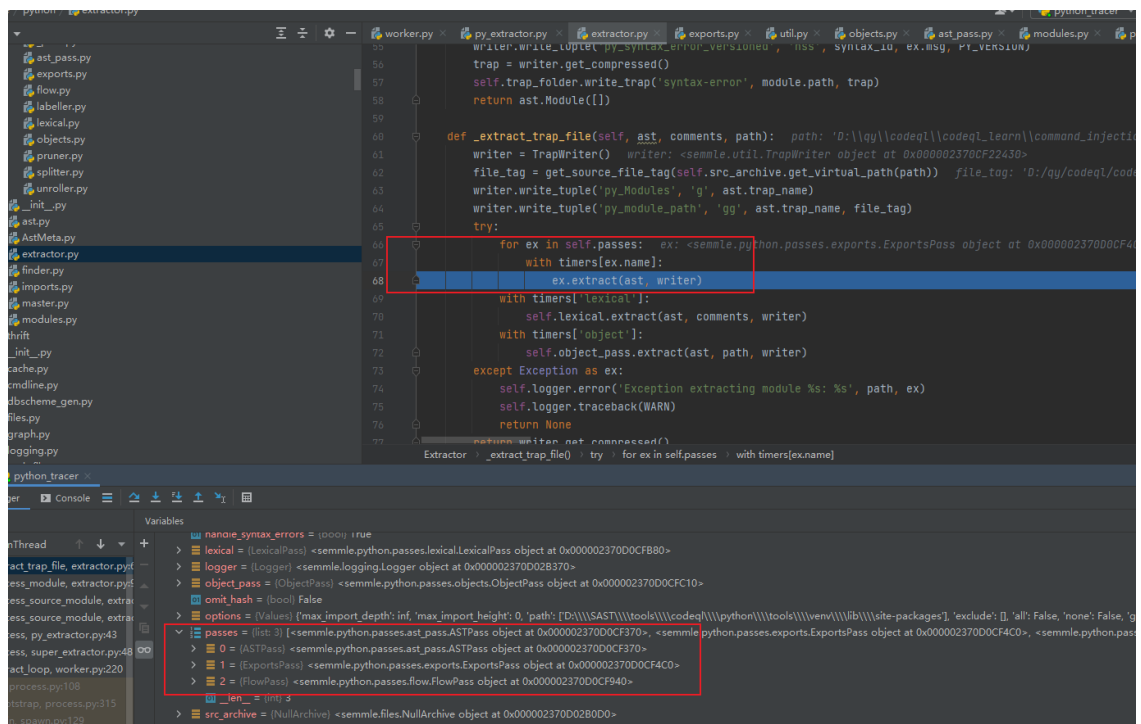
- 遍历、递归处理每一个文件，以及文件import的包，递归将每一个包以及依赖的系统包和第三方包全部处理，都会从系统安全的Python环境里查找

```
219 | start = time.time() start: 1674985939.2660053
220 | imports = extractor.process(unit) imports:
221 | end_time = time.time()
222 | extraction_time += (end_time - start)
223 | if (imports is SkippedBuiltin):
224 |     logger.info('Skipped built-in %s', unit)
225 | else:
226 |     for imp in imports:
227 |         reply_queue.put(('IMPORT', unit, imp))
228 | send_time += (time.time() - end_time)
229 | logger.info('Extracted %s in %0.0fms', unit, ((end_time - start) * 1000))
```

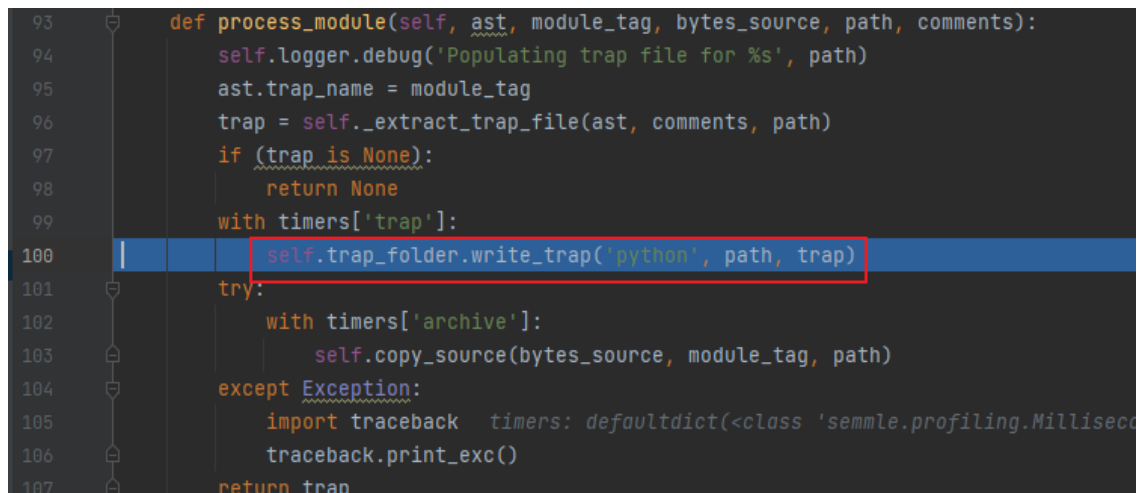
- 源码处理模块：主要处理入口 process\_source\_module

```
211 | def process_source_module(self, module):
212 |     if (self.worker is None):
213 |         raise Exception('worker is not set')
214 |     key = self.get_cache_key(module)
215 |     trap = self.cache.get(key)
216 |     if (trap is None):
217 |         trap = self.worker.process_source_module(module)
218 |         if (trap is not None):
219 |             self.cache.set(key, trap)
220 |     else:
221 |         self.logger.debug('Found cached trap file for %s', module.path)
222 |         self.worker.trap_folder.write_trap('python', module.path, trap)
223 |         try:
224 |             self.worker.copy_source(module.bytes_source, module.trap_name, module.path)
225 |         except Exception:
226 |             self.logger.traceback(WARN)
227 |     return trap
```

- 三阶段pass处理AST，有点类似于LLVM pass阶段了
  - 处理AST的基本结构
  - 处理导出表
  - 处理函数调用，生成调用图



- 处理完成写入trap文件：压缩包



## 5、trap文件

- 解压trap文件查看

```
codeql > codeql security_tests > trap_cache > 命令 injection.py.p683zpzOPkVgeZ4qCPH-aNRq8w=.trap
1  #10000 = @"PythonSourceModule:D:\qy\codeql\codeql_learn\command_injection.py:Bv1aqzFr7kuu6bhqf8yn6Y4TkLw="
2  py_modules(#10000)
3  #10001 = @"D:/qy/codeql/codeql_learn/command_injection.py;sourcefile"
4  py_module_path(#10000, #10001)
5  #10002 = *
6  #10003 = @"PythonSourceModule:D:\qy\codeql\codeql_learn\command_injection.py:Bv1aqzFr7kuu6bhqf8yn6Y4TkLw="
7  variable(#10002, #10003, "__name__")
8  #10004 = *
9  variable(#10004, #10003, "__package__")
10 #10005 = *
11 variable(#10005, #10003, "$")
12 py_extracted_version(#10000, "3")
13 py_modules(#10003)
14 #10006 = *
15 py_stmt_lists(#10006, #10003, 2)
16 #10007 = *
17 py_stmts(#10007, 12, #10006, 0)
18 py_scopes(#10007, #10003)
19 #10008 = *
20 py_alias_lists(#10008, #10007)
21 #10009 = *
22 py_aliases(#10009, #10008, 0)
23 #10010 = *
24 py_exprs(#10010, 13, #10009, 0)
25 py_scopes(#10010, #10003)
26 py_ints(0, #10010)
27 py_strs("subprocess", #10010, 3)
28 py_bools(#10010, 4)
29 #10011 = *
30 py_exprs(#10011, 19, #10009, 1)
31 py_scopes(#10011, #10003)
32 #10012 = *
33 variable(#10012, #10003, "subprocess")
34 py_variables(#10012, #10011)
35 #10013 = *
36 py_expr_contexts(#10013, 5, #10011)
37 #10014 = *
38 py_stmts(#10014, 12, #10006, 1)
39 py_scopes(#10014, #10003)
40 #10015 = *
41 py_alias_lists(#10015, #10014)
42 #10016 = *
43 py_aliases(#10016, #10015, 0)
44 #10017 = *
45 py_exprs(#10017, 13, #10016, 0)
46 py_scopes(#10017, #10003)
47 py_ints(0, #10017)
```

- tarp文件格式：datalog数据查询语言

Datalog是一种数据查询语言，专门设计与大型关系数据库交互[1]，语法与Prolog相似。正如SQL只是一个规范，Transact-SQL、PL-SQL是其具体实现一样；Datalog也是一个规范，bddbdb[2]、DES[3]、OverLog[4]、Deals[5]等都按照Datalog的语法实现了自己的语言，所以Datalog没有特定的执行环境（如Java之于Java虚拟机，Prolog之于SWI-Prolog）。

## Datalog [编辑]

维基百科，自由的百科全书

**Datalog**是一种数据查询语言，专门设计与大型关系数据库交互<sup>[1]</sup>，语法与Prolog相似。正如SQL只是一个规范，Transact-SQL、PL-SQL是其具体实现一样；Datalog也是一个规范，bddbdb<sup>[2]</sup>、DES<sup>[3]</sup>、OverLog<sup>[4]</sup>、Deals<sup>[5]</sup>等都按照Datalog的语法实现了自己的语言，所以Datalog没有特定的执行环境（如Java之于Java虚拟机，Prolog之于SWI-Prolog）。

### 起源 [编辑]

二十世纪九十年代，为了解决更多的问题，带有人工智能的系统通常要携带一个自行开发的数据库。这样的数据库非常简陋，不能数据共享与恢复，也不能在其他人工智能系统间通用。为了降低智能系统与数据库之间的耦合（智能系统可以使用现有的成熟的数据库，并方便地从一种数据库切换到另一种数据库），需要一种在数据库与智能系统间交互的语言，于是Datalog应运而生。<sup>[1]</sup>

David Maier发明了Datalog这个名称<sup>[6]</sup>。

### 与Prolog的异同 [编辑]

Datalog的语法是Prolog的子集；但是Datalog的语义与Prolog不同。

Prolog程序里的事实和规则的出现顺序决定了执行结果，很可能两条规则的出现顺序对换，程序就陷入死循环。Datalog程序对事实和规则的出现顺序不做要求，两条规则的出现顺序对换，执行结果仍然是一样的。

## 6、rel文件

- 上述trap文件生成后，codeql命令行执行 数据集导入，生成最终rel二进制文件，用来最终进行查询
- rel文件是自动生成，可以不用关注
- 生成rel文件后，数据库最终生成

