YASA使用笔记（一）Java程序分析

什么是YASA

引用官方的一段话：

> YASA（Yet another static analyzer）是一款基于UAST（Unified AST）上的程序静态分析工具，是蚂蚁集团基础安全部研发的一款静态程序分析引擎，其将程序源码转换成统一语法树中间表示（Unified Abstract Syntax Tree ），并在此之上做程序建模分析，最终产出相关代码报告。

基于YASA，我们能对多种程序语言进行统一的静态分析任务，从而提升我们安全检测的效率。本文即围绕着Java语言的分析与检测，向大家介绍YASA的使用。

YASA环境搭建

YASA的官方手册在：https://www.yuque.com/u22090306/bebf6g，这里也记录下我的搭建以及使用过程

首先我用了一个简单的docker环境，进行源码的使用（我这里的是0.2.25版本的，如果遇到版本迭代问题可以修改对应的dockerfile，更改源码获取）：

代码块

```
FROM ubuntu:22.04

ENV DEBIAN_FRONTEND=noninteractive

COPY sources.list /etc/apt/sources.list

RUN apt-get update && apt-get upgrade -y && apt-get install -y wget gnupg unzip cmake python3 python3-pip unzip git curl

RUN ln -s /usr/bin/python3 /usr/bin/python

RUN curl -fsSL https://deb.nodesource.com/setup_20.x | bash -

RUN apt-get install -y nodejs

WORKDIR /YASA

RUN git clone https://github.com/antgroup/YASA-Engine.git # change to wanted version

RUN git clone https://github.com/alipay/ant-application-security-testing-benchmark.git

RUN pip3 config set global.index-url https://mirrors.tuna.tsinghua.edu.cn/pypi/web/simple
```

如果大家也是VSCode的环境，也可以在YASA-Engine目录下创建.vscode目录，并新建launch.json，内容如下，方便后续的源码debug

代码块

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Debug Source Code",
      "program": "${workspaceFolder}/src/main.ts",
      "runtimeArgs": ["--inspect-brk"],
      "args": [
        "--sourcePath",
        "./target/test.java",
        "--language",
        "java",
        "--dumpAllCG"
      ],
      "console": "integratedTerminal"
    }
  ]
}
```

安装所需依赖，即可使用

代码块

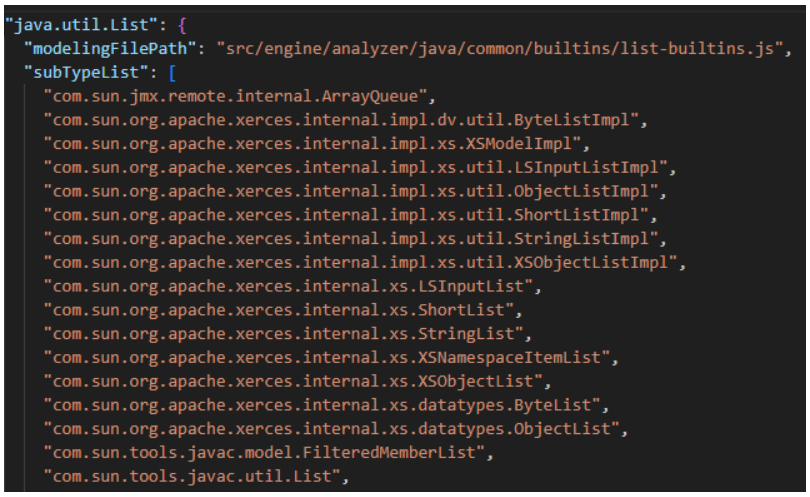
```
npm install
npm install --save-dev ts-node typescript @types/node
```

YASA-Java分析过程初探

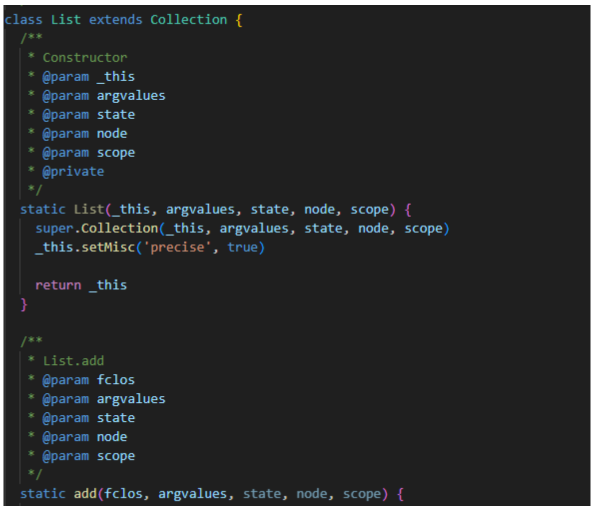
java-analyzer的位置在src/engine/analyzer/java/common/java-analyzer.js中，其分析可以总结为如下的几个步骤（其实应该不仅限于Java，作为一款统一多语言程序分析工具，其它语言的分析流程应该也都大差不差）：

preProcess

在这一步首先会加载对Java全局基本类的一个建模，例如class-hierarchy-and-modeling.json这个静态映射文件，分别记录了Java基本类对应的建模文件（模拟了这个类的行为，如集合的读取等行为）和其拥有的子类，是可以进行扩展的一个地方

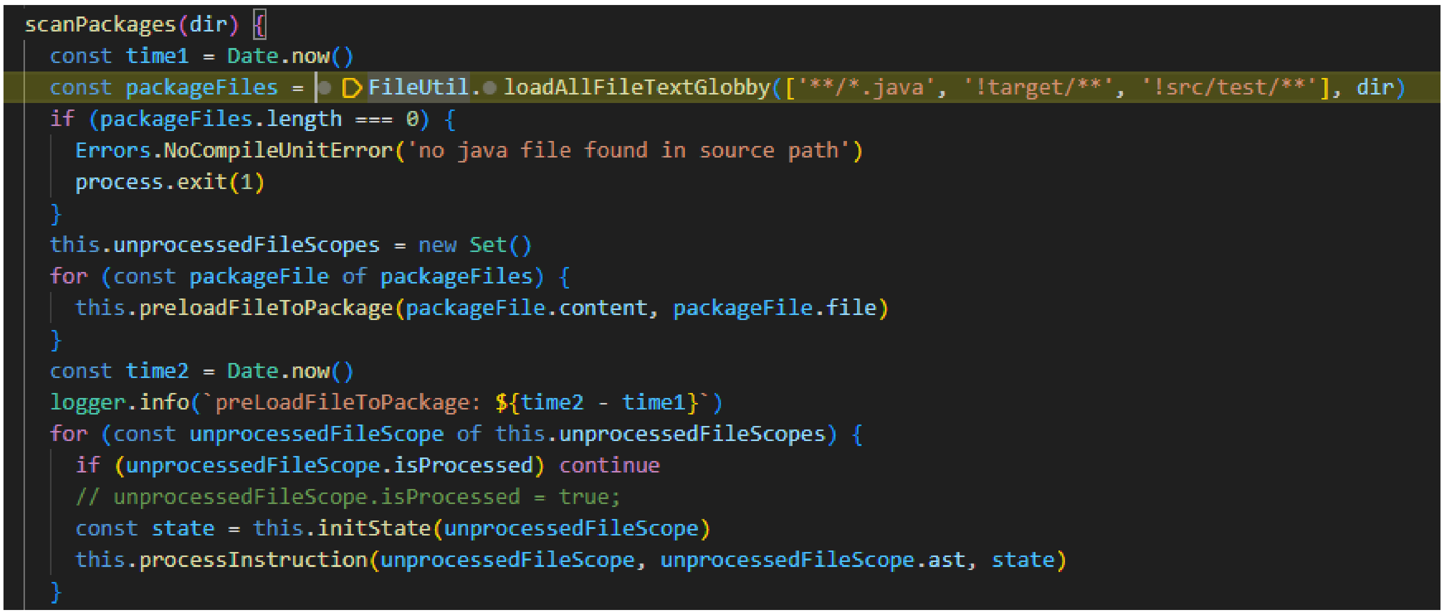


```
"java.util.List": {
  "modelingFilePath": "src/engine/analyzer/java/common/builtins/list-builtins.js",
  "subTypeList": [
    "com.sun.jmx.remote.internal.ArrayQueue",
    "com.sun.org.apache.xerces.internal.impl.dv.util.ByteListImpl",
    "com.sun.org.apache.xerces.internal.impl.xs.XSModelImpl",
    "com.sun.org.apache.xerces.internal.impl.xs.util.LSInputListImpl",
    "com.sun.org.apache.xerces.internal.impl.xs.util.ObjectListImpl",
    "com.sun.org.apache.xerces.internal.impl.xs.util.ShortListImpl",
    "com.sun.org.apache.xerces.internal.impl.xs.util.StringListImpl",
    "com.sun.org.apache.xerces.internal.impl.xs.util.XSObjectListImpl",
    "com.sun.org.apache.xerces.internal.xs.LSInputList",
    "com.sun.org.apache.xerces.internal.xs.ShortList",
    "com.sun.org.apache.xerces.internal.xs.StringList",
    "com.sun.org.apache.xerces.internal.xs.XSNamespaceItemList",
    "com.sun.org.apache.xerces.internal.xs.XSObjectList",
    "com.sun.org.apache.xerces.internal.xs.datatypes.ByteList",
    "com.sun.org.apache.xerces.internal.xs.datatypes.ObjectList",
    "com.sun.tools.javac.model.FilteredMemberList",
    "com.sun.tools.javac.util.List",
```



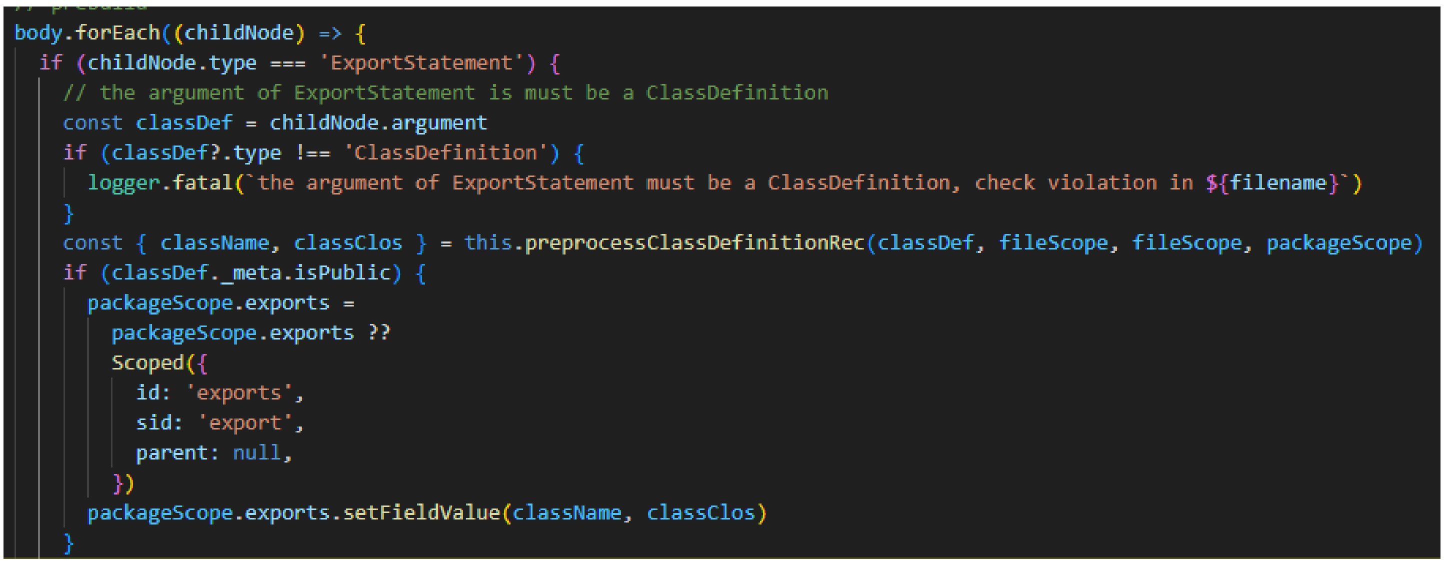
```
class List extends Collection {
  /**
   * Constructor
   * @param _this
   * @param argvalues
   * @param state
   * @param node
   * @param scope
   * @private
   */
  static List(_this, argvalues, state, node, scope) {
    super.Collection(_this, argvalues, state, node, scope)
    _this.setMisc('precise', true)

    return _this
  }

  /**
   * List.add
   * @param fclos
   * @param argvalues
   * @param state
   * @param node
   * @param scope
   */
  static add(fclos, argvalues, state, node, scope) {
```

随后在scanPackages函数中，首先会扫描解析Java文件，在这一步基于正则匹配所有的Java源码文件



```
scanPackages(dir) {
  const time1 = Date.now()
  const packageFiles = FileUtil.loadAllFileTextGlobby(['**/*.java', '!target/**', '!src/test/**'], dir)
  if (packageFiles.length === 0) {
    Errors.NoCompileUnitError('no java file found in source path')
    process.exit(1)
  }
  this.unprocessedFileScopes = new Set()
  for (const packageFile of packageFiles) {
    this.preloadFileToPackage(packageFile.content, packageFile.file)
  }
  const time2 = Date.now()
  logger.info(`preLoadFileToPackage: ${time2 - time1}`)
  for (const unprocessedFileScope of this.unprocessedFileScopes) {
    if (unprocessedFileScope.isProcessed) continue
    // unprocessedFileScope.isProcessed = true;
    const state = this.initState(unprocessedFileScope)
    this.processInstruction(unprocessedFileScope, unprocessedFileScope.ast, state)
  }
```

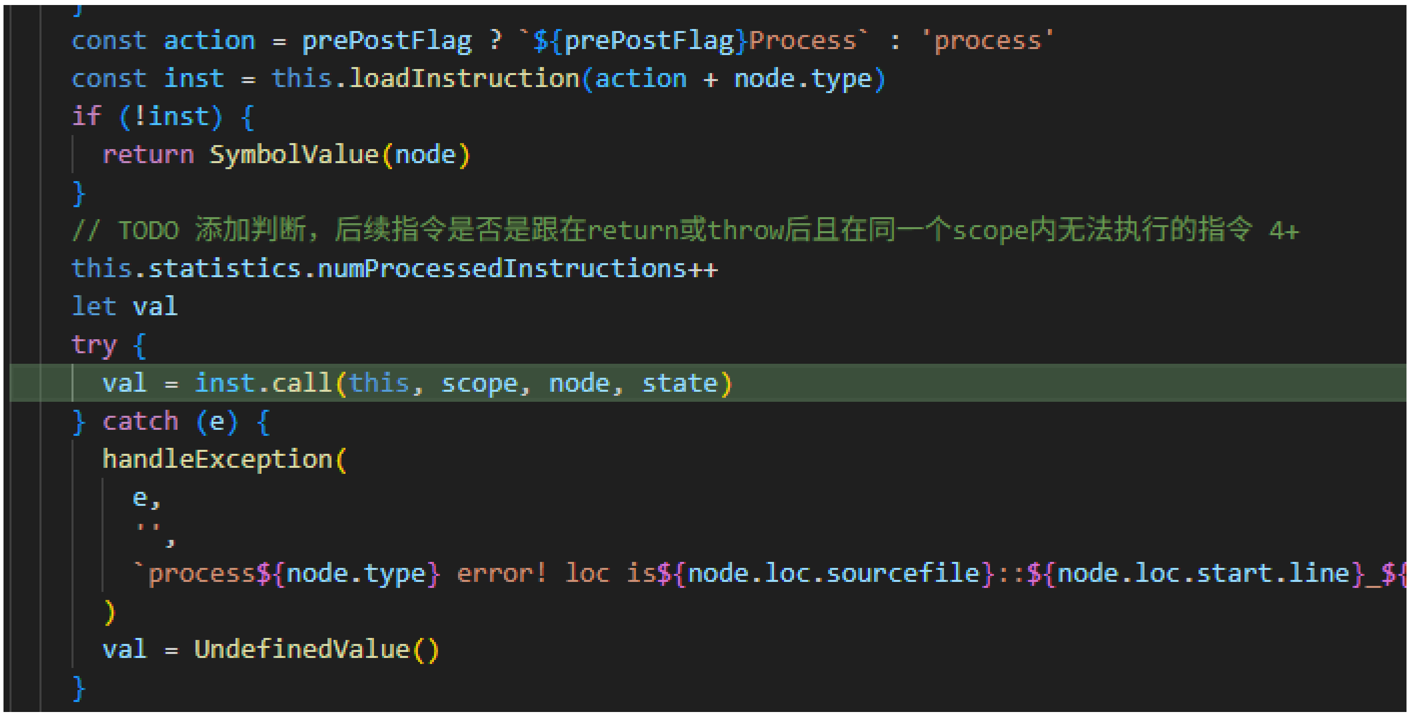
进入preloadFileToPackage，获得file对应的UAST，并首先处理ExportStatement（利用js中exports模拟Java中的public）



```
body.forEach((childNode) => {
  if (childNode.type === 'ExportStatement') {
    // the argument of ExportStatement is must be a ClassDefinition
    const classDef = childNode.argument
    if (classDef?.type !== 'ClassDefinition') {
      logger.fatal(`the argument of ExportStatement must be a ClassDefinition, check violation in ${filename}`)
    }
    const { className, classClos } = this.preprocessClassDefinitionRec(classDef, fileScope, fileScope, packageScope)
    if (classDef._meta.isPublic) {
      packageScope.exports =
        packageScope.exports ??
        Scoped({
          id: 'exports',
          sid: 'export',
          parent: null,
        })
      packageScope.exports.setFieldValue(className, classClos)
    }
```

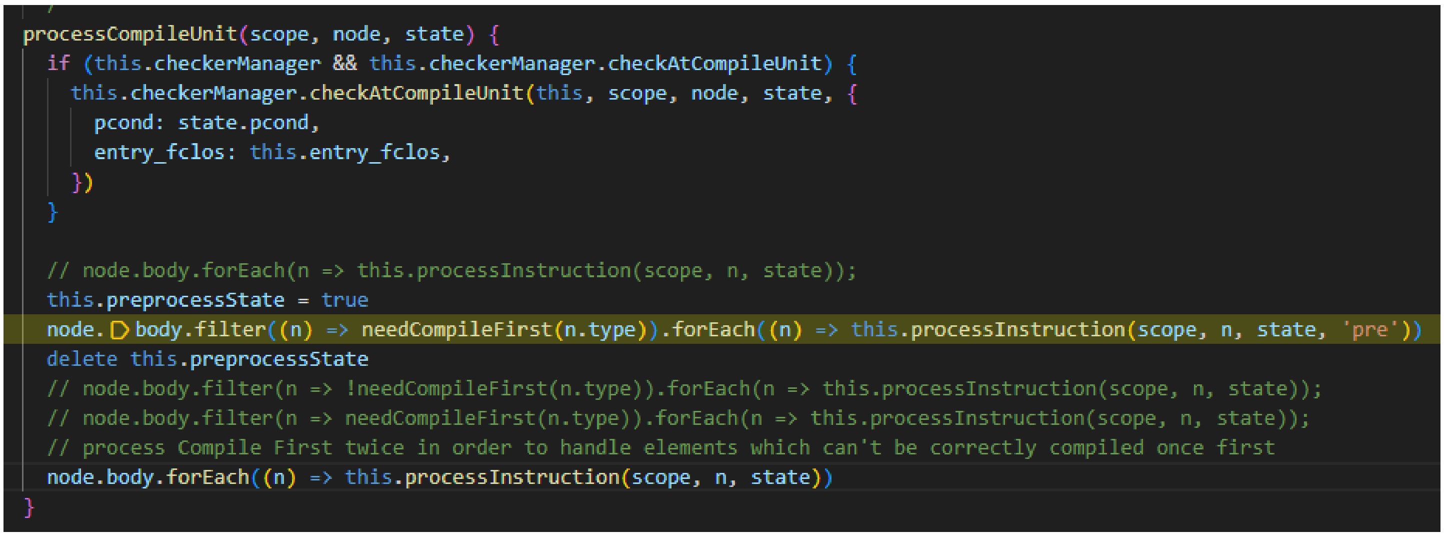
这一步最后会调用checkAtEndOfCompileUnit，触发相应checker的检测

随后进入processInstruction中处理指令，会根据不同的node的type调用不同的处理函数

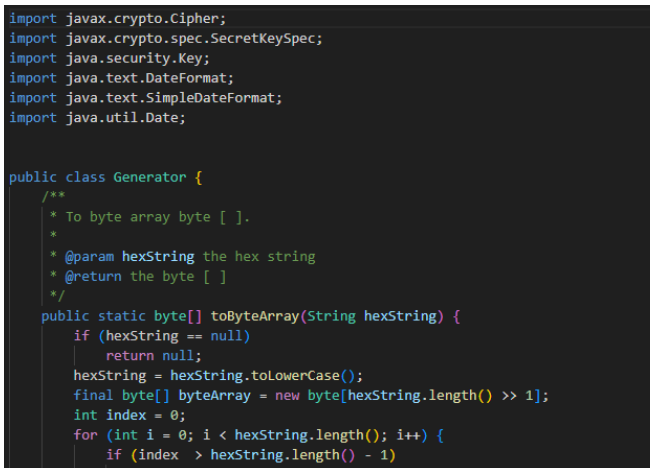


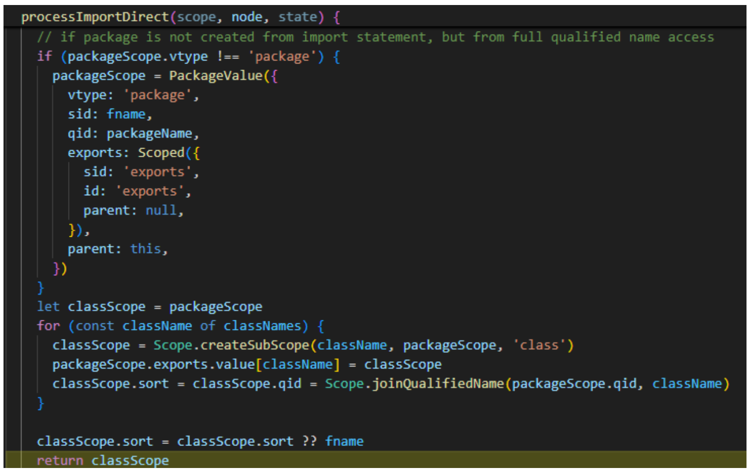
```
const action = prePostFlag ? `${prePostFlag}Process` : 'process'
const inst = this.loadInstruction(action + node.type)
if (!inst) {
  return SymbolValue(node)
}
// TODO 添加判断，后续指令是否是跟在return或throw后且在同一个scope内无法执行的指令 4+
this.statistics.numProcessedInstructions++
let val
try {
  val = inst.call(this, scope, node, state)
} catch (e) {
  handleException(
    e,
    '',
    `process${node.type} error! loc is${node.loc.sourcefile}::${node.loc.start.line}_${
  )
  val = UndefinedValue()
}
```

node的最顶层是一个CompileUnit，因此会先调用processCompileUnit，这里也有注册checker的调用，并会对每个node都再次调用processInstruction（AST的模拟执行思想）


```
processCompileUnit(scope, node, state) {
  if (this.checkerManager && this.checkerManager.checkAtCompileUnit) {
    this.checkerManager.checkAtCompileUnit(this, scope, node, state, {
      pcond: state.pcond,
      entry_fclos: this.entry_fclos,
    })
  }

  // node.body.forEach(n => this.processInstruction(scope, n, state));
  this.preprocessState = true
  node.body.filter((n) => needCompileFirst(n.type)).forEach((n) => this.processInstruction(scope, n, state, 'pre'))
  delete this.preprocessState
  // node.body.filter(n => !needCompileFirst(n.type)).forEach(n => this.processInstruction(scope, n, state));
  // node.body.filter(n => needCompileFirst(n.type)).forEach(n => this.processInstruction(scope, n, state));
  // process Compile First twice in order to handle elements which can't be correctly compiled once first
  node.body.forEach((n) => this.processInstruction(scope, n, state))
}
```

例如在我们的分析中第一个要处理的就是ImportExpression，对应源码的第一行，代码中也有具体的处理，随后就是类似这样的流程process不同的node

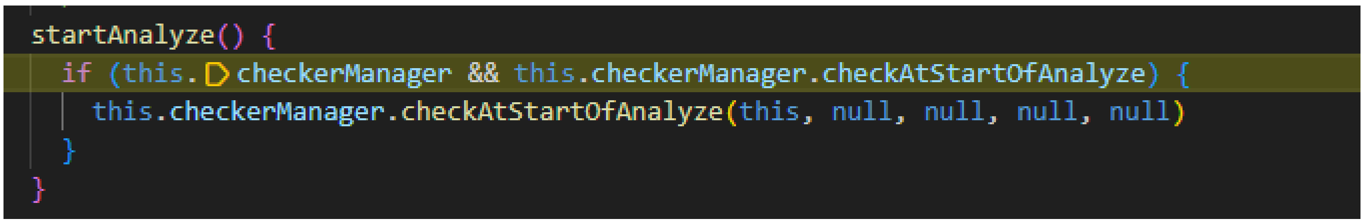

```
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.security.Key;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;


public class Generator {
    /**
     * To byte array byte [ ].
     *
     * @param hexString the hex string
     * @return the byte [ ]
     */
    public static byte[] toByteArray(String hexString) {
        if (hexString == null)
            return null;
        hexString = hexString.toLowerCase();
        final byte[] byteArray = new byte[hexString.length() >> 1];
        int index = 0;
        for (int i = 0; i < hexString.length(); i++) {
            if (index  > hexString.length() - 1)
```


```
processImportDirect(scope, node, state) {
  // if package is not created from import statement, but from full qualified name access
  if (packageScope.vtype !== 'package') {
    packageScope = PackageValue({
      vtype: 'package',
      sid: fname,
      qid: packageName,
      exports: Scoped({
        sid: 'exports',
        id: 'exports',
        parent: null,
      }),
      parent: this,
    })
  }
  let classScope = packageScope
  for (const className of classNames) {
    classScope = Scope.createSubScope(className, packageScope, 'class')
    packageScope.exports.value[className] = classScope
    classScope.sort = classScope.qid = Scope.joinQualifiedName(packageScope.qid, className)
  }

  classScope.sort = classScope.sort ?? fname
  return classScope
```

最后会对package的scope进行init

StartAnalyze

这里到相对简单了，调用注册的checker即可，具体的实现逻辑都在对应的checker里


```
startAnalyze() {
  if (this.checkerManager && this.checkerManager.checkAtStartOfAnalyze) {
    this.checkerManager.checkAtStartOfAnalyze(this, null, null, null, null)
  }
}
```

symbolInterpret

对语言以及框架的建模，例如自定义source入口方式，并根据入口自主加载source，未来再对这一部分进行更深入的分析

EndAnalyze

同StartAnalyze，不再赘述

YASA能力测试

Java调用图生成

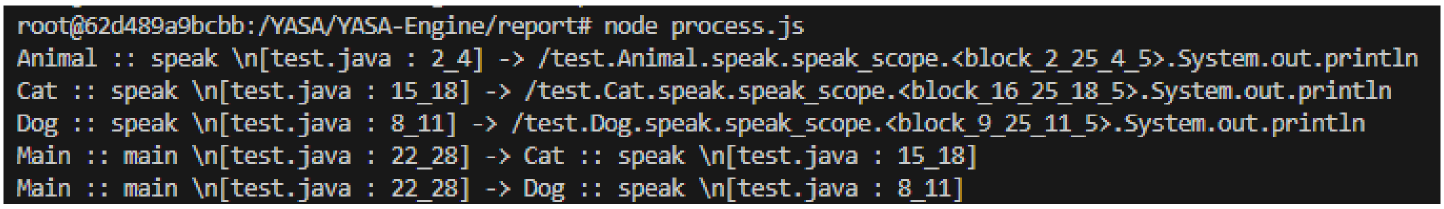
多态

我们首先测试下对于Java多态的支持如何，下面是一个例子

代码块

```java
class Animal {
    public void speak() {
        System.out.println("animal");
    }
}

class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("dog");
    }
}

class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("ca't");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Dog(); // poly
        Animal a2 = new Cat();

        a1.speak(); // dog
        a2.speak(); // cat
    }
}
```

通过添加--dumpAllCG即可输出调用图，但是这个结果包括了很多信息，不太好看，因此我们处理一下，仅输出调用关系



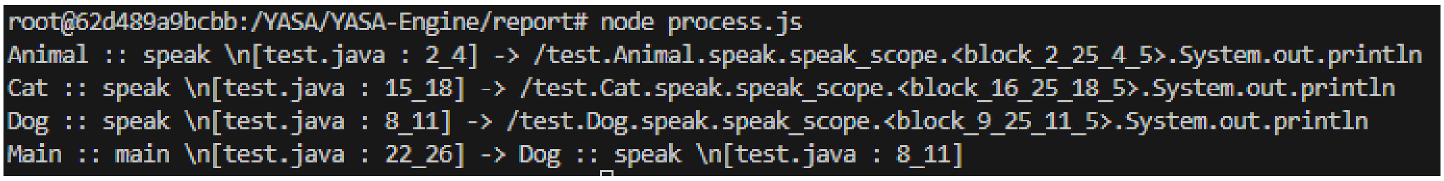
```
root@62d489a9bcbb:/YASA/YASA-Engine/report# node process.js
Animal :: speak \n[test.java : 2_4] -> /test.Animal.speak.speak_scope.<block_2_25_4_5>.System.out.println
Cat :: speak \n[test.java : 15_18] -> /test.Cat.speak.speak_scope.<block_16_25_18_5>.System.out.println
Dog :: speak \n[test.java : 8_11] -> /test.Dog.speak.speak_scope.<block_9_25_11_5>.System.out.println
Main :: main \n[test.java : 22_28] -> Cat :: speak \n[test.java : 15_18]
Main :: main \n[test.java : 22_28] -> Dog :: speak \n[test.java : 8_11]
```

这里看出对于main函数的多态调用YASA成功进行了识别，是比较准确的，我们还可以做些其他的测试，例如将main函数改成

代码块

```java
public class Main {
    public static void main(String[] args) {
        Animal a1 = new Dog(); // poly

        a1.speak(); // dog
    }
}
```

此时结果如下，这里是因为YASA没有仅从main方法入口进行分析，而是从每个function进行的分析（即dumpAllCG），否则应该不会再分析出cat的调用了，这个也是可以后续进行支持的



```
root@62d489a9bcbb:/YASA/YASA-Engine/report# node process.js
Animal :: speak \n[test.java : 2_4] -> /test.Animal.speak.speak_scope.<block_2_25_4_5>.System.out.println
Cat :: speak \n[test.java : 15_18] -> /test.Cat.speak.speak_scope.<block_16_25_18_5>.System.out.println
Dog :: speak \n[test.java : 8_11] -> /test.Dog.speak.speak_scope.<block_9_25_11_5>.System.out.println
Main :: main \n[test.java : 22_26] -> Dog :: speak \n[test.java : 8_11]
```

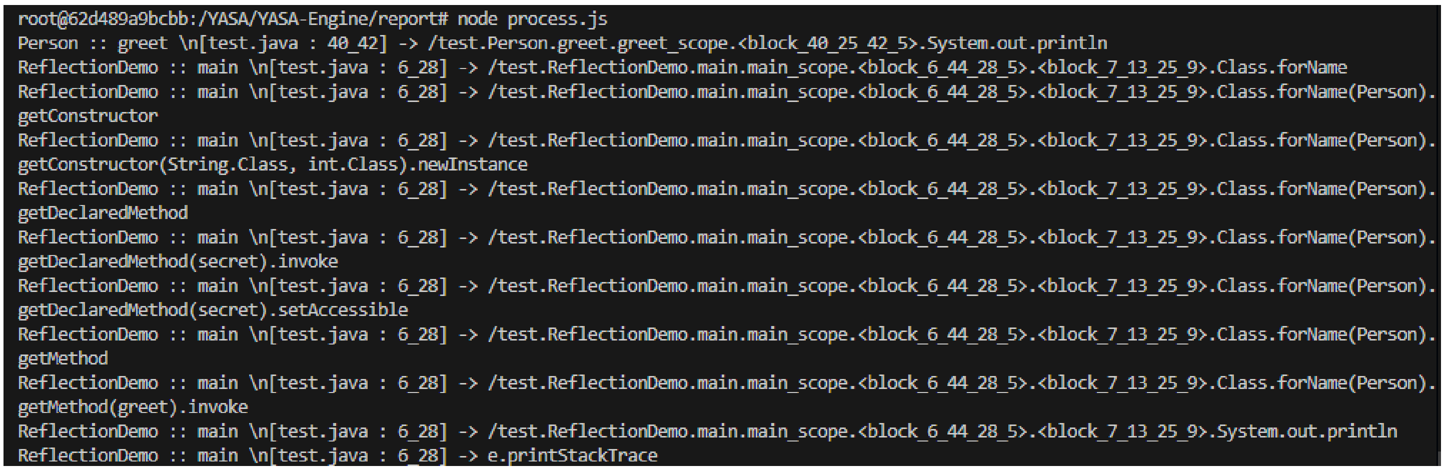
反射

我们再做一个反射的测试

代码块

```java
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ReflectionDemo {
    public static void main(String[] args) {
        try {
            Class<?> cls = Class.forName("Person");

            Constructor<?> ctor = cls.getConstructor(String.class, int.class);
            Object person = ctor.newInstance("Alice", 30);

            Method greet = cls.getMethod("greet");
            greet.invoke(person); // 输出: Hello, I'm Alice, 30

            Method secret = cls.getDeclaredMethod("secret");
            secret.setAccessible(true);
            Object secretResult = secret.invoke(person);
            System.out.println("secret returned: " + secretResult);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void greet() {
        System.out.println("Hello, I'm " + name + ", " + age);
    }

    private String secret() {
        return "my secret: " + (age * 2);
    }
}

```

结果如下，可以看出这部分暂时没有实际连起来，可以在之后进行支持



```
root@62d489a9bcbb:/YASA/YASA-Engine/report# node process.js
Person :: greet \n[test.java : 40_42] -> /test.Person.greet.greet_scope.<block_40_25_42_5>.System.out.println
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.Class.forName
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.Class.forName(Person).
getConstructor
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.Class.forName(Person).
getConstructor(String.Class, int.Class).newInstance
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.Class.forName(Person).
getDeclaredMethod
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.Class.forName(Person).
getDeclaredMethod(secret).invoke
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.Class.forName(Person).
getDeclaredMethod(secret).setAccessible
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.Class.forName(Person).
getMethod
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.Class.forName(Person).
getMethod(greet).invoke
ReflectionDemo :: main \n[test.java : 6_28] -> /test.ReflectionDemo.main.main_scope.<block_6_44_28_5>.<block_7_13_25_9>.System.out.println
ReflectionDemo :: main \n[test.java : 6_28] -> e.printStackTrace
```

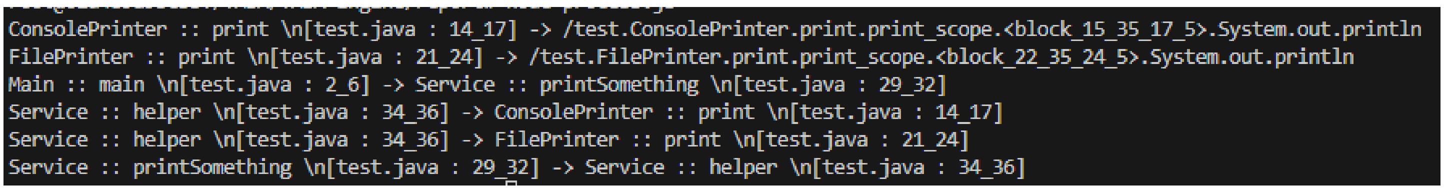
上下文敏感

对于下述的例子

代码块

```java
public class Main {
    public static void main(String[] args) {
        Service s = new Service();
        s.printSomething(false);
        s.printSomething(true);
    }
}

interface Printer {
    void print(String msg);
}

class ConsolePrinter implements Printer {
    @Override
    public void print(String msg) {
        System.out.println("Console: " + msg);
    }
}

class FilePrinter implements Printer {
    @Override
    public void print(String msg) {
        System.out.println("File: " + msg);
    }
}

class Service {

    public void printSomething(boolean toFile) {
        Printer printer = toFile ? new FilePrinter() : new ConsolePrinter();
        helper(printer);
    }

    private void helper(Printer printer) {
        printer.print("Hello World!");
    }
}
```

结果如图所示，可以看出仅记录了main对printSomething的一次调用，而对Service的helper函数的callee是FilePrinter和ConsolePrinter对应方法，貌似对上下文敏感性支持还有改进的地方



```
ConsolePrinter :: print \n[test.java : 14_17] -> /test.ConsolePrinter.print.print_scope.<block_15_35_17_5>.System.out.println
FilePrinter :: print \n[test.java : 21_24] -> /test.FilePrinter.print.print_scope.<block_22_35_24_5>.System.out.println
Main :: main \n[test.java : 2_6] -> Service :: printSomething \n[test.java : 29_32]
Service :: helper \n[test.java : 34_36] -> ConsolePrinter :: print \n[test.java : 14_17]
Service :: helper \n[test.java : 34_36] -> FilePrinter :: print \n[test.java : 21_24]
Service :: printSomething \n[test.java : 29_32] -> Service :: helper \n[test.java : 34_36]
```

Java漏洞数据集测试

xast中Java的数据集已经比较丰富了，涵盖了多种场景，这里我们额外用java-sec-code来测试下，例如对于如下的命令执行

代码块

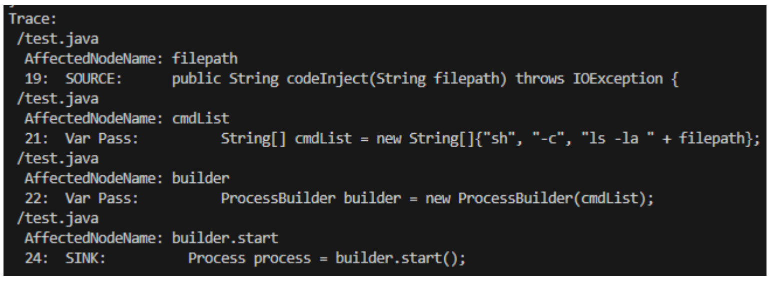
```java
package org.joychou.controller;

import org.joychou.security.SecurityUtil;
import org.joychou.util.WebUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

@RestController
public class CommandInject {

    protected final Logger logger = LoggerFactory.getLogger(this.getClass());

    @GetMapping("/codeinject")
    public String codeInject(String filepath) throws IOException {

        String[] cmdList = new String[]{"sh", "-c", "ls -la " + filepath};
        ProcessBuilder builder = new ProcessBuilder(cmdList);
        builder.redirectErrorStream(true);
        Process process = builder.start();
        return WebUtils.convertStreamToString(process.getInputStream());
    }

    @GetMapping("/codeinject/host")
    public String codeInjectHost(HttpServletRequest request) throws IOException {

        String host = request.getHeader("host");
        logger.info(host);
        String[] cmdList = new String[]{"sh", "-c", "curl " + host};
        ProcessBuilder builder = new ProcessBuilder(cmdList);
        builder.redirectErrorStream(true);
        Process process = builder.start();
        return WebUtils.convertStreamToString(process.getInputStream());
    }

    @GetMapping("/codeinject/sec")
    public String codeInjectSec(String filepath) throws IOException {
        String filterFilePath = SecurityUtil.cmdFilter(filepath);
        if (null == filterFilePath) {
            return "Bad boy. I got u.";
        }
        String[] cmdList = new String[]{"sh", "-c", "ls -la " +
filterFilePath};
        ProcessBuilder builder = new ProcessBuilder(cmdList);
        builder.redirectErrorStream(true);
        Process process = builder.start();
        return WebUtils.convertStreamToString(process.getInputStream());
    }
}
```

我们添加sink、sanitizer到配置文件中，在运行时也要添加sanitizer到checkerIDs中

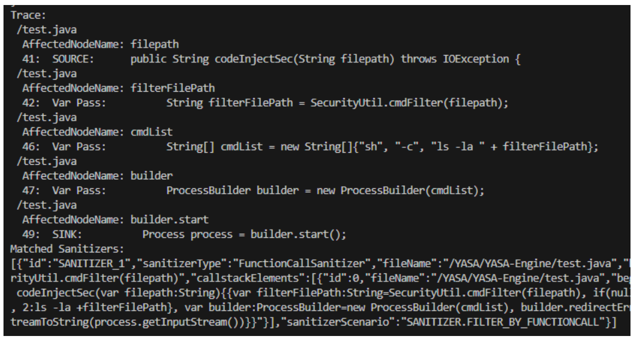
代码块

```json
[
  {
    "checkerIds": [
      "taint_flow_java_input",
      "taint_flow_spring_input"
    ],
    "sanitizers": [
      {
        "id": "SANITIZER_1",
        "sanitizerType": "FunctionCallSanitizer",
        "sanitizerScenario": "SANITIZER.FILTER_BY_FUNCTIONCALL",
        "calleeType": "org.joychou.security.SecurityUtil",
        "fsig": "cmdFilter",
        "args": [
          "0"
        ]
      }
    ],
    "sources": {},
    "sinks": {
      "FuncCallTaintSink": [
        {
          "args": [
            "0"
          ],
          "attribute": "JavaCommandExec",
          "calleeType": "",
          "fsig": "Runtime.getRuntime().exec"
        }
      ],
      "ObjectTaintFuncCallSink": [ // 通过这种方式指定receiver变量被污染，后续使用-1
的方式
        {
          "attribute": "JavaCommandExec",
          "calleeType": "ProcessBuilder",
          "fsig": "start",
          "sanitizerIds": [
              "SANITIZER_1"
          ]
        }
      ]
    }
  }
]
```

最终是可以成功检测的：

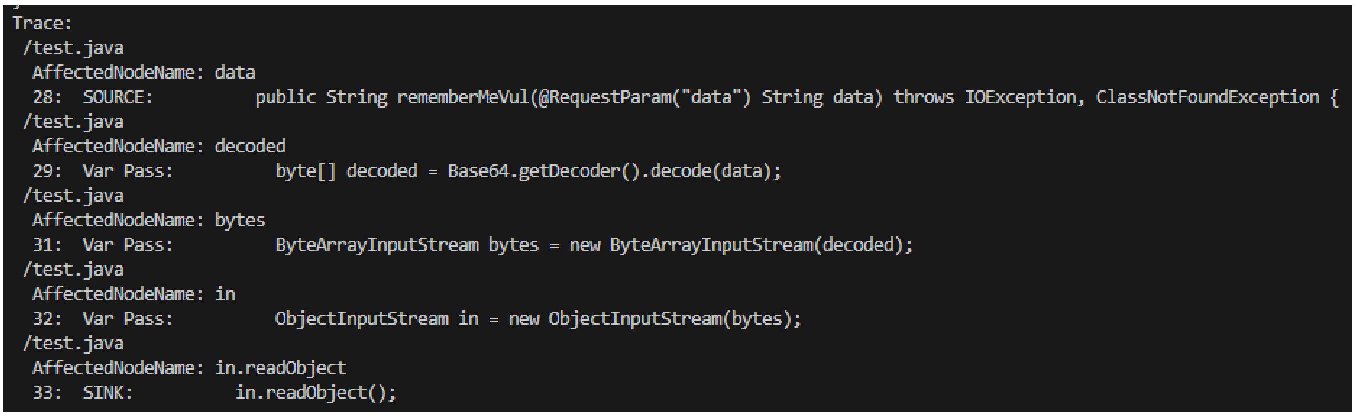


```
Trace:
 /test.java
  AffectedNodeName: filepath
  19:  SOURCE:      public String codeInject(String filepath) throws IOException {
 /test.java
  AffectedNodeName: cmdList
  21:  Var Pass:          String[] cmdList = new String[]{"sh", "-c", "ls -la " + filepath};
 /test.java
  AffectedNodeName: builder
  22:  Var Pass:          ProcessBuilder builder = new ProcessBuilder(cmdList);
 /test.java
  AffectedNodeName: builder.start
  24:  SINK:          Process process = builder.start();
```



```
Trace:
 /test.java
  AffectedNodeName: filepath
  41:  SOURCE:      public String codeInjectSec(String filepath) throws IOException {
 /test.java
  AffectedNodeName: filterFilePath
  42:  Var Pass:          String filterFilePath = SecurityUtil.cmdFilter(filepath);
 /test.java
  AffectedNodeName: cmdList
  46:  Var Pass:          String[] cmdList = new String[]{"sh", "-c", "ls -la " + filterFilePath};
 /test.java
  AffectedNodeName: builder
  47:  Var Pass:          ProcessBuilder builder = new ProcessBuilder(cmdList);
 /test.java
  AffectedNodeName: builder.start
  49:  SINK:          Process process = builder.start();
Matched Sanitizers:
[{"id":"SANITIZER_1","sanitizerType":"FunctionCallSanitizer","fileName":"/YASA/YASA-Engine/test.java","
rityUtil.cmdFilter(filepath)","callstackElements":[{"id":0,"fileName":"/YASA/YASA-Engine/test.java","be
 codeInjectSec(var filepath:String){{var filterFilePath:String=SecurityUtil.cmdFilter(filepath), if(nul
, 2:ls -la +filterFilePath}, var builder:ProcessBuilder=new ProcessBuilder(cmdList), builder.redirectEr
treamToString(process.getInputStream())}}"}],"sanitizerScenario":"SANITIZER.FILTER_BY_FUNCTIONCALL"}]
```

同时也可以实现Java反序列化漏洞的检测



```
Trace:
 /test.java
  AffectedNodeName: data
  28:  SOURCE:          public String rememberMeVul(@RequestParam("data") String data) throws IOException, ClassNotFoundException {
 /test.java
  AffectedNodeName: decoded
  29:  Var Pass:          byte[] decoded = Base64.getDecoder().decode(data);
 /test.java
  AffectedNodeName: bytes
  31:  Var Pass:          ByteArrayInputStream bytes = new ByteArrayInputStream(decoded);
 /test.java
  AffectedNodeName: in
  32:  Var Pass:          ObjectInputStream in = new ObjectInputStream(bytes);
 /test.java
  AffectedNodeName: in.readObject
  33:  SINK:          in.readObject();
```

值得注意的是，对于SQL注入漏洞的检测可能需要修改sink中calleeType，例如将java.sql.Statement改为Statement（如果代码中显示有import java.sql.Statement即不用修改，若是通过import java.sql.*则需要，因为对这种import的建模还尚未完全支持）

总结

总的来说，YASA是一款十分优秀的静态分析工具，我们可以基于YASA开发多种多样的功能，后续也将进一步测试YASA在真实项目上的能力，以及开发更多好用的checker