ArkAnalyzer 快速入门指南

SMAT-Lab@BUAA

https://smat-lab.github.io

1. ArkAnalyzer 简介

ArkAnalyzer 是针对基于 ArkTS 语言开发的鸿蒙原生应用的静态代码分析框架。图 1 展示了其基本工作原理。目前，ArkAnalyzer 的输入为 ArkTS 文件（即后缀为 ets 的文件）。ArkAnalyzer 会先为 ArkTS 代码生成一个抽象语法树（AST），接着遍历这颗语法树并生成一个 Scene 数据结构。这个 Scene 数据结构对代码结构进行了抽象，用户可通过该数据结构快速获取 ArkTS 项目中某个具体的类、函数或者属性。接下来，ArkAnalyzer 为每一个函数生成一个控制流程图（CFG），用户可基于此图进行控制流相关的分析。基于 CFG，ArkAnalyzer 进一步实现方法调用图的生成（CG），并基于此支持用户实现数据流分析。

图 1：ArkAnalyzer 基本工作原理。

1.1 ArkAnalyzer-IR（三地址码）简介

临时变量命名规则

临时变量命名采用“$temp”+number 的形式，其中 number 为临时变量为所在的 CFG 中的零食变量序号

代码简化（循环消减、去语法糖化）

源代码中的循环，包括 while，for，for of，for in 在 CFG 中全部改为代码块配合 if 语句的形式。

语法糖指编程语言中提供的一种简化或更加直观的语法结构，这种语法结构能够使得代码更加易读、易写，但并不增加新的语言功能。TS 中的语法糖包括匿名函数和对集合元素处理时用到的 for-each 函数等。在下图的例子中，myArray 是一个数组，使用 forEach 方法遍历数组，对于数组中的每个元素，都执行一个匿名函数。在下半可以看到方舟分析器将匿名函数显

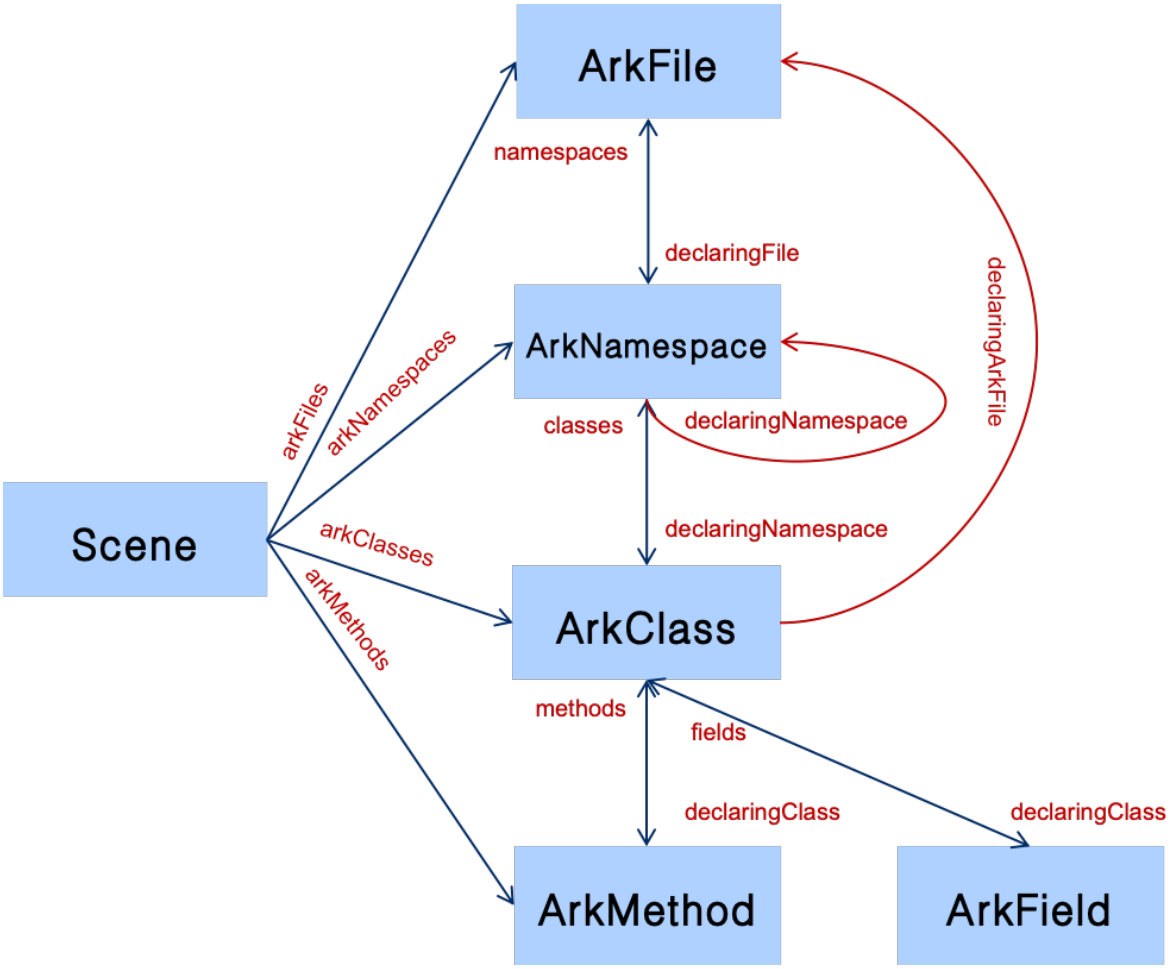
示定义出来，命名为 AnonymousFunc$desuagring$0，for-each 使用时直接调用。

```
function desuagring(){
    const myArray: number[] = [1, 2, 3, 4, 5];

    myArray.forEach((value, index) => {
        console.log(index,value);
    });
}
```

```
*** arkMethod:  AnonymousFunc$desuagring$0
value = parameter0: unknown
index = parameter1: unknown
this = this: @TypeInferenceTest/test: _DEFAULT_ARK_CLASS
instanceinvoke console.<@_UnkownProjectName/_UnkownFileName: .log()>(index, value)
return
*** arkMethod:  desuagring
this = this: @TypeInferenceTest/test: _DEFAULT_ARK_CLASS
$temp1 = newarray (number)[5]
$temp1[0] = 1
$temp1[1] = 2
$temp1[2] = 3
$temp1[3] = 4
$temp1[4] = 5
myArray = $temp1
instanceinvoke myArray.<@_UnkownProjectName/_UnkownFileName: .forEach()>(AnonymousFunc$desuagring$0)
return
```

1.2 Scene 数据结构

Scene 类为 ArkAnalyzer 的核心类，用户可以通过该类访问所分析代码（项目）的所有信息，包括文件列表、类列表、方法列表、属性列表等。Scene 类具体数据结构如下图所示。



2. ArkAnalyzer 环境配置

2.1 安装 arkanalyzer

2.1.1 Nodejs 安装

arkanalyzer 需要运行在 nodejs18 及以上版本，可以使用 node 命令检查环境中的 nodejs 版本，版本不满足时需要安装 nodejs。

```
node -v
v18.20.4
```

Windows：https://nodejs.org/zh-cn 下载安装最新的 LTS 版本

Linux：使用 nvm 来管理 nodejs 版本

```
curl -o- https://https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash
source ~/.bashrc
nvm install v18.20.4
```

2.1.2 创建 arkanalyzer 分析项目

新文件夹，创建新的项目，安装 arkanalyzer

```
mkdir arkanalyzerProject
cd arkanalyzerProject
npm init
npm install arkanalyzer
npm install ts-node -D
```

配置 tsconfig.json

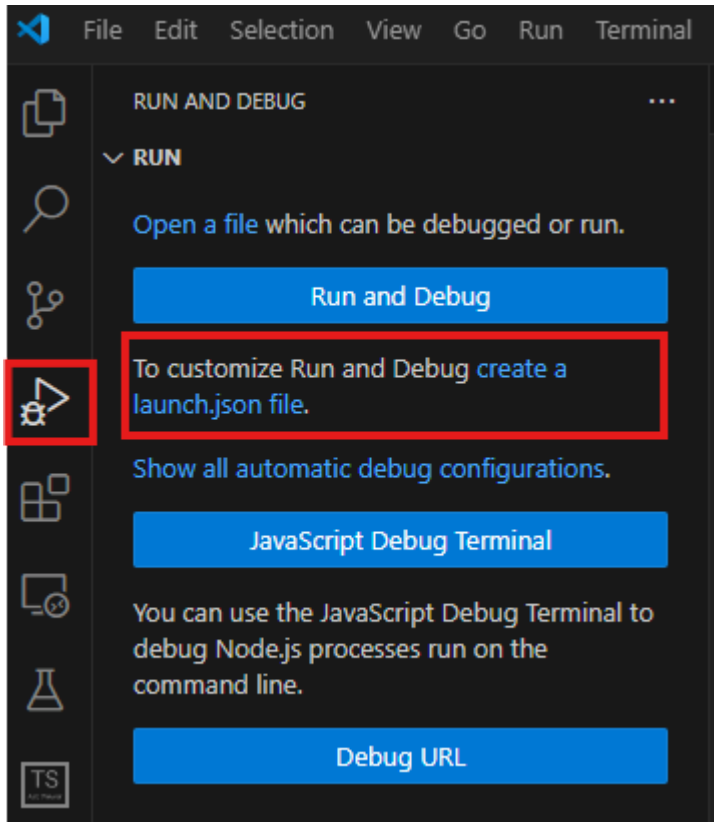
```
{
    "compilerOptions": {
        "module": "CommonJS"
    }
}
```

环境配置好后的目录结构

```
arkanalyzerProject
|---.vscode
|      |---launch.json
|---node_modules
|      |---arkanalyzer
|      |---ts-node
|---src
|---package.json
|---tsconfig.json
```

2.1.3 vscode 调试 ts

方式 1：配置 launch.json

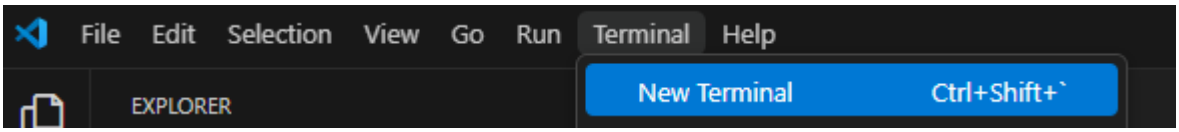



```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            // 作用：调试时加载 ts-node 包（在调试时直接运行 ts 代码）
            "runtimeArgs": ["-r", "ts-node/register"],
            // 作用：调试当前选择的文件
            "args": ["${file}"]
        }
    ]
}
```

launch.json 配置好后，在 vscode 打开需要调试的 ts 文件，F5 启动调试

方式 2：JavaScript Debug Terminal

打开 vscode 终端窗口



在终端窗口新建 JavaScript Debug Terminal

在终端命令输入 node 命令启动调试

```
node -r ts-node/register src/ex/ex01/basicUsage.ts
```

2.2 分析 ts 项目

使用 SceneConfig 类指定项目根目录分析 ts 项目，样例如下：

```
import { SceneConfig, Scene} from 'arkanalyzer';
const projectRoot = 'tests';
let config: SceneConfig = new SceneConfig();
config.buildFromProjectDir(projectRoot);
let scene: Scene = new Scene();
scene.buildSceneFromProjectDir(config);
```

2.3 分析 Hap 应用项目

使用 SceneConfig 类指定 json 文件分析 Hap 应用项目，样例如下：

```
import { SceneConfig, Scene} from 'arkanalyzer';
const projectConfigJson = 'tests/project_config.json';
let config: SceneConfig = new SceneConfig();
config.buildFromJson(projectConfigJson);
let scene: Scene = new Scene();
scene.buildBasicInfo(config);
```

分享 Hap 应用项目 json 配置文件，包含了待待分析项目路径、OpenHarmony SDK 路径、其他需要分析 SDK 路径，其中"targetProjectDirectory"指定的目录下必须存在 build-profile.json5，否则构建失败。样例如下（注：真实的 json 文件中不能含有注释）

```
{
    "targetProjectName": "Example",//项目名称
    "targetProjectDirectory": "tests\\resources\\example",//HAP 项目所在位置路径
    "logPath": "out/log.txt",//日志输出位置
    "ohosSdkPath": "",//ohos sdk api 路径，项目不包含 ohos api 时为空
    "kitSdkPath": "",//ohos 套件 api 路径，项目不包含 ohos api 时为空
    "systemSdkPath": "",//系统 sdk 路径，项目不包含 ohos api 时为空
    "otherSdks": []//其他 sdk，一般为空
}
```

3. ArkAnalyzer 使用样例

3.1 基本功能（ex01)

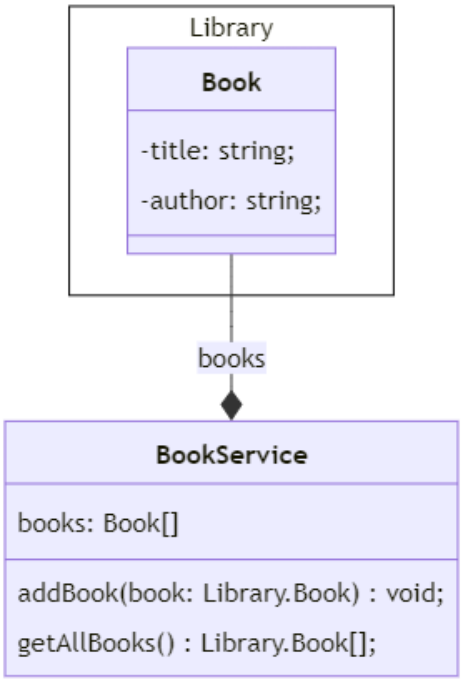
下面将以以下项目为例，说明 ArkAnalyzer 基本功能的使用方法。

项目示例

项目结构：

```
demo_project/src
|---models
|      |---books.ts
|---services
|      |---bookService.ts
|---index.ts
```

类图：



示例文件内容：

books.ts

```
export namespace Library {
  export class Book {
    public title: string;
    public author: string;
    constructor(title: string,author: string) {
      this.title = title;
      this.author = author;
    }
  }
}
```

bookService.ts

```ts
import { Library } from '../models/book';

export class BookService {
  private books: Library.Book[] = [];

  public addBook(book: Library.Book): void {
    this.books.push(book);
  }

  public getAllBooks(): Library.Book[] {
    return this.books;
  }

  public getBooksByAuthor(author: string): Library.Book[] {
    let ans: Library.Book[] = [];
    for (let i = 0; i < this.books.length; i++) {
      let book = this.books[i];
      if (book.author === author) {
        ans.push(book);
      }
    }
    return ans;
  }
}
```

index.ts

```
import { Library } from "./models/book";
import { BookService } from "./services/bookService";

const bookService = new BookService();
bookService.addBook(new Library.Book("The Hobbit", "J.R.R. Tolkien"));
const books = bookService.getAllBooks();
console.log(books);
```

ex01.1：获取所有文件

为获取项目中所有的文件，我们使用 Scene 类的 getFiles() 方法。以下是示例代码和输出。

basicUsage.ts

```
let files: ArkFile[] = scene.getFiles();
let fileNames: string[] = files.map((file) => file.getName());
console.log(fileNames);
```

输出

```
[
  'src\\index.ts',
  'src\\models\\book.ts',
  'src\\services\\bookService.ts'
]
```

ex01.2：获取命名空间

为获取项目中定义的所有命名空间，我们使用 Scene 类的 getNamespaces() 方法。以下是示例代码和输出。

basicUsage.ts

```
let namespaces: ArkNamespace[] = scene.getNamespaces();
let namespaceNames: string[] = namespaces.map((ns) => ns.getName());
console.log(namespaceNames);
```

输出

```
[ 'Library' ]
```

此外，也可以从 ArkFile 中获取该文件下的所有命名空间，我们使用 ArkFile 类的 getNamespaces() 方法。以下是示例代码和输出。

basicUsage.ts

```
let namespaces2: ArkNamespace[] = files[1].getNamespaces();
let namespaceNames2: string[] = namespaces2.map(ns => ns.getName());
console.log(namespaceNames2);
```

输出

```
[ 'Library' ]
```

ex01.3：获取所有类

要获取项目中定义的所有类，我们使用 Scene 类的 getClasses() 方法。以下是示例代码和输出。

basicUsage.ts

```
let classes: ArkClass[] = scene.getClasses();
let classNames: string[] = classes.map(cls => cls.getName());
console.log(classNames);
```

输出

```
[
  '_DEFAULT_ARK_CLASS',
  '_DEFAULT_ARK_CLASS',
  '_DEFAULT_ARK_CLASS',
  'Book',
  '_DEFAULT_ARK_CLASS',
  'BookService'
]
```

说明：ArkAnalyzer 会为每一个文件和命名空间分别创建一个默认类，记为

_DEFAULT_ARK_CLASS。

同样，也可以从 ArkFile 中获取该文件下的所有类，我们使用 ArkFile 类的 getClasses() 方法。以下是示例代码和输出。

basicUsage.ts

```
let classes2: ArkClass[] = files[2].getClasses();
let classNames2: string[] = classes2.map((cls) => cls.getName());
console.log(classNames2);
```

输出

```
[ '_DEFAULT_ARK_CLASS', 'BookService' ]
```

当然，我们也可以从 ArkNamespace 中获取该命名空间下的所有类，我们使用 ArkNamespace 类的 getClasses()方法。以下是示例代码和输出。

basicUsage.ts

```
let classes3: ArkClass[] = namespaces[0].getClasses();
let classNames3: string[] = classes3.map((cls) => cls.getName());
console.log(classNames3);
```

输出

```
[ '_DEFAULT_ARK_CLASS', 'Book' ]
```

ex01.4：获取所有属性

为获取特定类中定义的所有属性，我们首先需要获取该类的 ArkClass，然后调用其 getFields()方法。以下是示例代码和输出。

basicUsage.ts

```
let bookClass: ArkClass = classes.filter((value) => value.getName() == 'Book')[0];
let fields: ArkField[] = bookClass.getFields();
let fieldNames: string[] = fields.map((fld) => fld.getName());
console.log(fieldNames);
```

输出

```
[ 'title', 'author' ]
```

ex01.5：获取所有方法

为获取特定类中定义的所有方法，我们首先需要获取该类的 ArkClass，然后调用其 getMethods() 方法。以下是示例代码和输出。

basicUsage.ts

```
let serviceClass: ArkClass = classes.filter((value) => value.getName() === 'BookService')[0];
let methods: ArkMethod[] = serviceClass.getMethods();
let methodNames: string[] = methods.map((mthd) => mthd.getName());
console.log(methodNames);
```

输出

```
[ 'addBook', 'getAllBooks', 'getBooksByAuthor' ]
```

同样的，也可以从 Scene 中获取其内部的全部方法，示例如下。

basicUsage.ts

```
let methods1: ArkMethod[] = scene.getMethods();
let methodNames1: string[] = methods1.map((mthd) => mthd.getName());
console.log(methodNames1);
```

输出

```
[
  '_DEFAULT_ARK_METHOD',
  '_DEFAULT_ARK_METHOD',
  '_DEFAULT_ARK_METHOD',
  'constructor',
  '_DEFAULT_ARK_METHOD',
  'addBook',
  'getAllBooks',
  'getBooksByAuthor'
]
```

说明：与 ArkClass 类似，ArkAnalyzer 会为每一个文件和命名空间分别创建一个默认方法，记为_DEFAULT_ARK_METHOD。

ex01.6：获取方法 CFG

通过 ArkMethod 的 getBody()方法获取方法体，再通过 getCfg()方法可以获取方法的 CFG，示例如下所示。

```
let method = serviceClass.getMethodWithName('getBooksByAuthor');
let cfg: Cfg | undefined = method?.getBody()?.getCfg();
```

下面我们简要介绍 ArkAnalyzer 中 CFG 的数据结构设计。

属性：

blocks：一个 Set<BasicBlock> 集合，存储了图中所有的基本块。

stmtToBlock：一个 Map<Stmt, BasicBlock> 映射，关联每个语句（Stmt）到它所属的基本块。

startingStmt：表示图中的起始语句。这个起始语句用于标识控制流图的入口点。

defUseChains：存储定义-使用链的数组。

declaringMethod：存储声明了这个 CFG 的方法。

declaringClass：存储声明了这个 CFG 的类。

方法：

getStmts：遍历所有基本块，收集并返回图中所有的语句。

getBlocks、getStartingBlock、getStartingStmt 提供了获取基本块集合、起始块和起始语句的方法。

getDefUseChains：返回定义-使用链的集合。

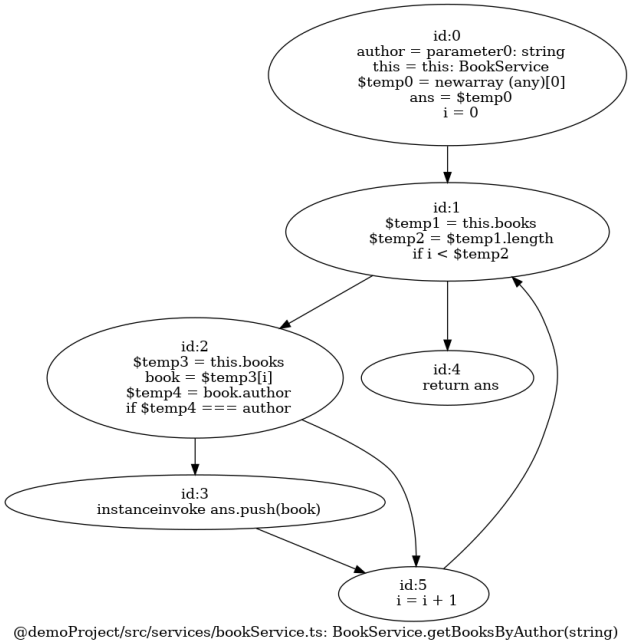
此外，为了便于更直观理解 CFG 结构，ArkAnalyzer 提供 DotMethodPrinter 类，能够把 CFG 以 Dot 图形式保存，可视化的展示 CFG 图。获取 getBooksByAuthor 方法 CFG 示例如下所示。

```
let method = serviceClass.getMethodWithName('getBooksByAuthor');
let cfg: Cfg | undefined = method?.getBody()?.getCfg();
let dotMethodPrinter = new DotMethodPrinter(method!);
PrinterBuilder.dump(dotMethodPrinter, 'out/getBooksByAuthor_cfg.dot');
```

dump()方法输出的.dot 文件，可以通过 Graphviz Online 平台将.dot 文件识别为 Dot 图，或者使用 dot 命令行生成图 dot -tsvg -oxx.svg xx.dot。例如 getBooksByAuthor 方法

```
public getBooksByAuthor(author: string): Library.Book[] {
  let ans: Library.Book[] = [];
  for (let i = 0; i < this.books.length; i++) {
    let book = this.books[i];
    if (book.author === author) {
      ans.push(book);
    }
  }
  return ans;
}
```

转化为 Dot 图如下：



@demoProject/src/services/bookService.ts: BookService.getBooksByAuthor(string)

3.2 调用图分析（ex02）

ArkAnalyzer 生成的方法调用图是局部代码的方法调用图，构建时需要指定分析的起始点。方法调用图会从该起始点开始进行 bfs，打印出所有相关的调用关系。

以下面的代码文件作为测试用例，分别使用三种算法进行方法调用图生成

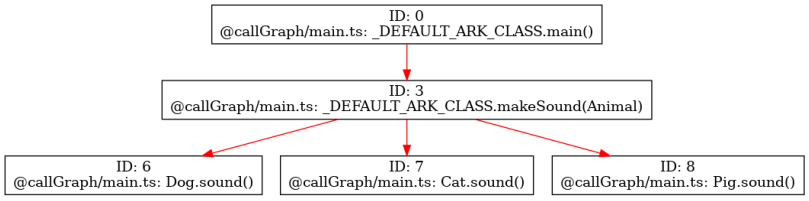
```
abstract class Animal {
    abstract sound(): void;
}
class Dog extends Animal {
    sound(): void {}
}
class Cat extends Animal {
    sound(): void {}
}
class Pig extends Animal {
    sound(): void {}
}
function makeSound(animal: Animal) {
    animal.sound();
}

function main() {
    let cat = new Cat();
    makeSound(new Dog());
}
```

通过下面的方式指定分析入口点函数，并对项目所有代码进行类型推导

```
// 加入需要进行分析的方法签名,method 是文件中默认类下 main 方法
let methodSignature = method.getSignature()
let entryPoints: MethodSignature[] = [methodSignature]
// 进行类型推导
scene.inferTypes()
```

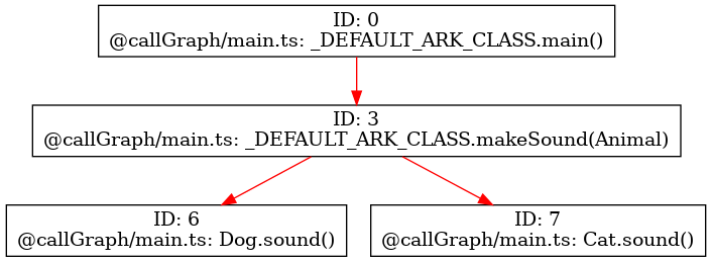
ex02.1：打印函数调用图（CHA）

```
// 调用 CHA 算法
let callGraph = scene.makeCallGraphCHA(entryPoints) ;
// 保存 dot 图
callGraph.dump('./out/CHA.dot');
// 获取输出
let methods = new Set<Method>();
for (const entry of callGraph.getEntries()) {
    methods.add(callGraph.getMethodByFuncID(entry)!);
}
let calls = callGraph.getDynEdges()
```



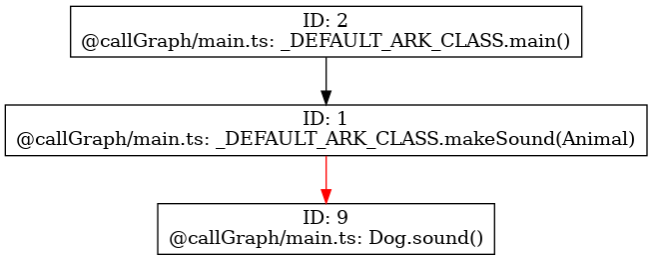
ex02.2：打印函数调用图（RTA）

```
// 调用 RTA 算法
let callGraph = scene.makeCallGraphRTA(entryPoints)
// 保存 dot 图
callGraph.dump('./out/RTA.dot');
let methods = new Set<Method>();
for (const entry of callGraph.getEntries()) {
    methods.add(callGraph.getMethodByFuncID(entry)!);
}
let calls = callGraph.getDynEdges()
```



ex02.3：打印函数调用图（PTA）

```
// 调用 PTA 算法
let callGraph = new CallGraph(scene);
let cgBuilder = new CallGraphBuilder(callGraph, scene);
cgBuilder.buildDirectCallGraphForScene();
let pag = new Pag();
let entry = callGraph.getEntries();
let ptaConfig = new PointerAnalysisConfig(2, './out', true, true);
let pta = new PointerAnalysis(pag, callGraph, scene, ptaConfig);
pta.setEntries(entry);
pta.start();
// 保存 dot 图
callGraph.dump('./out/PTA.dot');
```



ex02.4：找到所有 Log 点

在这个实验中我们用一个简单的自定义的 log 类模拟日志打印

```
//log.ts
export class Logger {
    public static info(message: string) {
        console.log(message)
    }
    public static warn(message: string) {
        console.log(message)
    }
    public static error(message: string) {
        console.log(message)
    }
}
```

```
// basicUsage.ts
import { Logger } from "./log"
abstract class Animal { sound(): void { } }
class Dog extends Animal { sound(): void { Logger.warn("dog sound") } }
class Cat extends Animal { sound(): void { Logger.error("dog sound") } }
function main() {
    makeSound(new Dog())
    Logger.info("create new dog")
}
function makeSound(animal: Animal) {
    animal.sound()
}
```

通过上述方法生成方法调用图（这里以 CHA 为例），得到下面的结果

```
Calls:
        <@project/main2: _DEFAULT_ARK_CLASS.main()  ->
                <@project/main2: _DEFAULT_ARK_CLASS.makeSound(Animal)
                <@project/log: Logger.info(string)

        <@project/main2: _DEFAULT_ARK_CLASS.makeSound(Animal)  ->
                <@project/main2: Dog.sound()
                <@project/main2: Cat.sound()

        <@project/main2: Dog.sound()  ->
                <@project/log: Logger.warn(string)

        <@project/main2: Cat.sound()  ->
                <@project/log: Logger.error(string)
```

3.3 函数内数据流分析（ex03）

ex03.1：SSA 观察

获取 scene 后，可对 scene 下的所有方法进行 SSA 处理，处理前需要使用 inferType（ ）函数进行类型推导，SSA 处理代码示例如下

```
const scene: Scene;
scene.inferTypes();
...
const arkMethod: ArkMethod;
const body = arkMethod.getBody();
let ssaFormer = new StaticSingleAssignmentFormer();
ssaFormer.transformBody(body);
```

ex03.2：理解 Def-Use Chain

获取指定的 ArkMethod 后，可通过如下方式获取 Def-Use Chain。每个 chain 有三个属

性，分别是 value（变量），def（定义语句），use（使用语句）。需要注意的是，Def-Use Chain 只适用于函数内的局部变量分析，不能分析对象属性和数组。

```
const arkMethod: ArkMethod;
const cfg = arkMethod.getBody()!.getCfg();
cfg.buildDefUseChain();
const chains = cfg.getDefUseChains()
variable: y, def: y = 0, use: y = y + 1
variable: x, def: x = 1, use: x = x * 3
variable: y, def: y = y + 1, use: z = y + 4
```

ex03.3：规则检测示例

在 TS 中，exec 函数可以直接执行命令，因此需要检查 exec 函数的参数是否有例如“rm -rf /bin”之类的会对计算机带来不可逆损害的命令。下面的示例代码为：获取指定 ArkMethod 后，逐句检查是否有 exec 函数调用，如果有则输出参数。

```
const arkMethod: ArkMethod;
const cfg = arkMethod.getCfg()!;
for (const stmt of cfg.getStmts()) {
    if (stmt.getExprs().length > 0) {
        const expr = stmt.getExprs()[0];
        if (expr instanceof ArkStaticInvokeExpr &&
expr.getMethodSignature().getMethodSubSignature().getMethodName() ==
"exec") {
            const args = expr.getArgs();
             console.log(args);
        }
    }
}
```

3.4 ArkUI ViewTree 分析(ex04)

ex04.1: 理解 ArkUI 中的 ViewTree

ArkUI 采用的是声明式的开发语言，UI 和数据的绑定关系采用@装饰器修饰的状态变量。而声明式语法规则和复杂的状态变量数据绑定传递关系给程序分析带来了挑战。为了屏蔽声明式 UI 和状态变量的变化逻辑，ArkAnalyzer 构建了基础的 UI 程序分析数据结构 ViewTree。ViewTree 作为描述 UI 组件关系的树，每个结点包含基础的 UI 组件和组件使用的状态变量、属性函数、事件函数、状态变量跨组件的传递。基于 ViewTree 可以分析 UI 中 Page 的组成，Page->Page，组件->组件的迁移，支撑高层 UI 的进一步分析。

CountDown(src\ex\resources\CountDown)是一个倒计时 Hap 应用，ex05.1 利用 ViewTree 查找父组件 CountDown 与子组件 Clock 传递的数据流。

```
import { ClassSignature, Decorator, PrinterBuilder, Scene, SceneConfig, ViewTreePrinter } from 'arkanalyzer';
import { join } from 'path';

function decorators2str(decorators: Decorator[]): string {
    return decorators.map((value) => `@${value.getContent()}`).join(', ');
}

let config: SceneConfig = new SceneConfig();
config.buildFromJson(join(__dirname, '../resources/CountDown/arkanalyzer_config.json'));
let scene: Scene = new Scene();
scene.buildBasicInfo(config);
scene.buildScene4HarmonyProject();
scene.collectProjectImportInfos();
scene.inferTypes();

let arkFile = scene.getFiles().find((file) => file.getName().endsWith('CountDown.ets'));
let arkClass = arkFile?.getClassWithName('CountDown');
let vt = arkClass?.getViewTree();
let root = vt?.getRoot();
root?.walk((item) => {
    if (item.isCustomComponent() && item.signature instanceof ClassSignature && item.signature?.getClassName() === 'Clock') {
        let values: string[] = [];
        for (const [key, value] of item.stateValuesTransfer!) {
            values.push(`${decorators2str(value.getStateDecorators())} ${value.getName()} -> ${decorators2str(key.getStateDecorators())} ${key.getName()}`);
        }

        if (values.length > 0) {
```

```
// 输出打印结果
CountDown->Clock transfer values
 @State rotates -> @Link rotates,
 @State progressVal -> @Link progressVal
```

3.5 函数间数据流分析（ex05）

ex05.1: 空指针检测

空指针分析可以检查代码是否存在程序试图使用空指针（即指向不存在对象的指针）时，导致程序崩溃或出现未定义的行为。例如程序用到了 a.b，如果 a 为 undefined 或者 null 会导致程序崩溃。下面的代码演示了如何对程序进行空指针分析，入口函数为第一个文件的“U2”，检测从这个函数开始到结束是否有空指针的属性调用。

```
const projectRoot = 'src/ex/resources/UndefinedVariable';
let config: SceneConfig = new SceneConfig();
config.buildFromProjectDir(projectRoot);
let scene = new Scene();
scene.buildSceneFromProjectDir(config);

const method = scene.getMethods().filter((v) => v.getName() === 'U2')[0];
const problem = new UndefinedVariableChecker(
    [...method.getCfg()!.getBlocks()][0].getStmts()[method.getParameters().length],
    method
);
const solver = new UndefinedVariableSolver(problem, scene);
solver.solve();

for (const outcome of problem.getOutcomes()) {
    let position = outcome.stmt.getOriginPositionInfo();
    console.log('undefined error in line ' + position.getLineNo());
}
```