数据说明

本文件目录下一共有六个项目，供同学们参考和调试程序。现给出以下几条说明：

- 生成数据使用的测试选择代码采用0-CFA算法进行调用图构建（`Util.makeZeroCFABuilder`），同学们也可以使用CHA算法构建调用图（`new CHACallGraph`）；
- 每个项目文件夹下都有一个`data/`目录，里面存放了样例数据。除`CMD`外，其他5个项目给出的数据包括：
    1. `change_info.txt`，变更信息文件。每行一条记录，格式为`<类的内部表示> <方法签名>`，模拟发生变更的生产代码，作为测试选择的输入之一。本文档将在**相关知识**部分详细介绍类的内部表示形式和方法签名；
    2. `selection-class.txt`和`selection-method`，两种不同粒度的测试选择结果。每行一条记录，格式为`<类的内部表示> <方法签名>`，表示一个被选中的测试方法，表示程序的输出。同学们可以参考这两个文件调试程序。**注意**：由于程序的内部实现细节不同可能会导致输出有所差别，同学们应该保证测试选择的结果和**代码依赖图**能够对应上；
    3. **提示**：每一条用于表示某个方法`<类的内部表示> <方法签名>`的记录都是通过WALA得出，同学们可以参考以下步骤以获取到类似的信息：

```
/* 省略构建分析域（AnalysisScope）对象scope的过程 */

// 1.生成类层次关系对象
ClassHierarchy cha = ClassHierarchyFactory.makeWithRoot(scope);

// 2.生成进入点
Iterable<Entrypoint> eps = new AllApplicationEntrypoints(scope, cha);
// 3.利用CHA算法构建调用图
CallGraph cg = new CHACallGraph(cha);
cg.init(eps);
// 4.遍历cg中所有的节点
for(CGNode node: cg) {
    // node中包含了很多信息，包括类加载器、方法信息等，这里只筛选出需要的信息
    if(node.getMethod() instanceof ShrikeBTMethod) {
        // node.getMethod()返回一个比较泛化的IMethod实例，不能获取到我们想要的信
息
        // 一般地，本项目中所有和业务逻辑相关的方法都是ShrikeBTMethod对象
        ShrikeBTMethod method = (ShrikeBTMethod) node.getMethod();
        // 使用Primordial类加载器加载的类都属于Java原生类，我们一般不关心。

 if("Application".equals(method.getDeclaringClass().getClassLoader().toS
tring())) {
            // 获取声明该方法的类的内部表示
            String classInnerName =
method.getDeclaringClass().getName().toString;
            // 获取方法签名
            String signature = method.getSignature();
            System.out.println(classInnerName + " " + signature);
        }
    } else {
        System.out.println(String.format("'%s'不是一个ShrikeBTMethod：%s",
                           node.getMethod(),
node.getMethod().getClass());
    }
```

```
}
```

- 除常规的输入输出文件外，`0-CMD/data/`还包含四个依赖图文件，即`class-CMD-cfa.dot`、`class-CMD-cfa.pdf`、`method-CMD-cfa.dot`和`method-CMD-cfa.pdf`，分别对应两种粒度。同学们可以参考给出`.dot`文件（当做文本文件打开）的语法生成自己的`.dot`文件，并在配置好`graphviz`环境变量后参考下面命令行生成PDF格式的依赖图：

```
dot -T pdf -o <文件名>.pdf <文件名>.dot
```

- 所有项目均具有Maven项目结构，这里建议同学们使用字节码（`.class`）作为程序分析的原料。每个项目编译出的字节码文件在`target/`目录下，其中：生产代码在`target/classes/`下，测试代码在`target/test-classes`下。字节码文件缺失的同学在`pom.xml`所在目录下，使用以下命令重新生成`target`目录：

```
mvn clean test
```

相关知识

- 本小节将介绍一些本文档涉及的相关知识。相关内容均来自ASM框架的（https://asm.ow2.io/）的使用文档。ASM是一个字节码操作框架，提供了大量符合JVM规格的API。
- 本文档旨在帮助同学们了解一些Java相关的表示形式，很多内容为个人理解，如有问题请发邮件指正，联系方式见文档底部。
- 更多知识请参阅JVM规格说明：https://docs.oracle.com/javase/specs/jvms/se8/html/

内部表示

java程序需要先编译成字节码文件（`.java` → `.class`），然后才能在Java虚拟机上运行。`.class`和`.java`表示一个类型的方法有所不同，一般使用**类型描述符**（Type Descriptor）表示一个类。在这里我们将`.class`文件中类型的表示方式称作**内部表示**。

各种Java类型的类型描述符

Java Type	Type Descriptor
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object;
int[]	[I
Object	[[Ljava/lang/Object;

规律:

1. 基本类型的类型描述符都是单个字符;
2. 类类型的类型描述符是**以L开头、";"结尾**的字符串

签名

由于允许递归嵌套（例如`List<List<E>>`），泛型的语法十分复杂，文档中仅展示了部分语法。本项目使用到的方法签名均由WALA生成。由于分析的代码比较简单，大多不涉及泛型。方法签名的基本表示形式可以归纳为：`<声明类>.<方法签名>`

对泛型的详细规定可以在JVM规格说明中找到。

语法1：类型签名

> *TypeSignature*: **Z** | **C** | **B** | **S** | **I** | **F** | **J** | **D** | *FieldTypeSignature*
>
> *FieldTypeSignature*: *ClassTypeSignature* | **[** *TypeSignature* | *TypeVar*
>
> *ClassTypeSignature*: **L** *Id* (/ *Id*)* *TypeArgs*? (**.** *Id* *TypeArgs*?)* **;**
>
> *TypeArgs*: **<** *TypeArg+* **>**
>
> *TypeArg*: ***** | (**+** | **-**)? *FieldTypeSignature*
>
> *TypeVar*: **T** *Id* **;**

- **签名举例**:

类型（Type）	类型签名（Type Signature）
`List<E>`	`Ljava/util/List<TE;>;`
`List<?>`	`Ljava/util/List<*>;`
`List<? extends Number>`	`Ljava/util/List<+Ljava/lang/Number;>;`
`List<? super Integer>`	`Ljava/util/List<-Ljava/lang/Integer;>;`
`List<List<String>[]>`	`Ljava/util/List<[Ljava/util/List<Ljava/lang/String;>;>;`
`HashMap<K, V>.HashIterator<K>`	`Ljava/util/HashMap<TK;TV;>.HashIterator<TK;>;`

- **解释**:
  - Type Signature: 类型签名。类型签名对后续的几条方法进行了汇总，将**基本类型**的签名也包括在内。其中，基本类型的签名就是该基本类型的描述符。
  - Field Type Signature: 域类型签名。对应泛型尖括号中`extends`和`super`后面跟随的类型。可以是类的类型签名、数组的类型签名以及类型变量中的其中一个。对应第3行的`Number`、第4行的`Integer`。

    ⚠根据*TypeSignature*语法的描述，表中举出的所有类型签名都是**域类型签名**。
  - Class Type Signature: 类类型签名。用于表示一个类的签名。对应类类型签名的语法规则给出了签名的基本组成形式，直接对应表格中第6行。

    ⚠根据*FieldTypeSignature*语法的描述，表中所有的类型签名又都是**类类型签名**。因为表中没有给出基本类型和数组类型的类型签名。
  - Type Argument: 类型参数。用于处理泛型中出现的`?`、`extends`和`super`，对应表格中的2~4行。
  - Type Variable: 类型变量。指的是类似于`<K, V>`这种类型参数，对应表格中的第6行。

语法2：方法签名

> *MethodTypeSignature*: *TypeParams*? (*TypeSignature**) (*TypeSignature* | **V**) *Exception**
>
> *Exception*: **^** *ClassTypeSignature* | **^** *TypeVar*
>
> *TypeParams*: **<** *TypeParam*+ **>**
>
> *TypeParam*: *Id* **:** *FieldTypeSignature*? (**:** *FieldTypeSignature*)*

- **签名举例**:

方法声明（Method Declaration）	方法签名（Method Signature）
`static <T> Class<? extends T> m (int n)`	`<T:Ljava/lang/Object;>(I)Ljava/lang/Class<+TT;>`

- **解释**:
  - Method Type Signature: 方法类型签名。由三部分组成:
    - 类型参数：对应`<T>`
    - 参数列表：`()`包裹0或多个类型签名
    - 返回值：类型签名或者`V`（void）
    - 异常列表：通过^连接的、类类型签名或者类型变量的0或多次重复。
  - Exception: 异常。描述方法可能抛出的异常，异常可能也带有泛型（*TypeVar*）。通过^符号连接方法参数列表和异常数组。

- Type Parameter: 类型参数（⚠注意与Type Argument区分开）。对应一个有类型参数的方法，一般出现在方法声明的access码后面。对应举例中的 `<T>` ⇒ `<T:Ljava/lang/Object;>`

语法3：类签名

> *ClassSignature*: *TypeParams*? *ClassTypeSignature ClassTypeSignature**

⚠需要和类类型签名区分开来：**类签名**用于在类的**声明**；而**类类型签名**用于类的**使用**（构造、传参等）。

- **签名举例**:

类（Class）	类签名（Class Signature）
`C<E> extends List<E>`	`<E:Ljava/lang/Object;>Ljava/util/List<TE;>;`

- **解释**:
  - Class Signature: 类签名由三个部分组成：
    - 类型参数：0~1个*TypeParams*，声明时带类型参数的类才有这一部分。
    - 类层级关系：表示该类继承了哪些类，至少有一个*ClassTypeSignature*

      ❓为什么语法里面不是*ClassTypeSignature+*

联系方式

关于此文档有任何问题，欢迎发邮件询问或指正：qrx@smail.nju.edu.cn