调研报告——Doop

MF21330068 沈天琪

Doop 是一个针对 Java 程序的指针分析框架，在实现中使用了 Java、Datalog、Groovy 三种编程语言。这个框架提供了命令行与 Java API 两种操作接口。

在 Doop 框架中，分析的大致流程是：首先从分析的 Java 程序出发，利用 Soot 工具从字节码中抽取出待分析程序中所需要的静态分析信息；此外，根据程序所接受的不同分析选项，程序将自动化的从模块化的 Datalog 规则中构造出对应分析选项的规则；最终，分析规则与分析输入将被交给具体执行运算的 Datalog 引擎，Datalog 引擎将结合两者以具体的结果计算出来并最终保存。

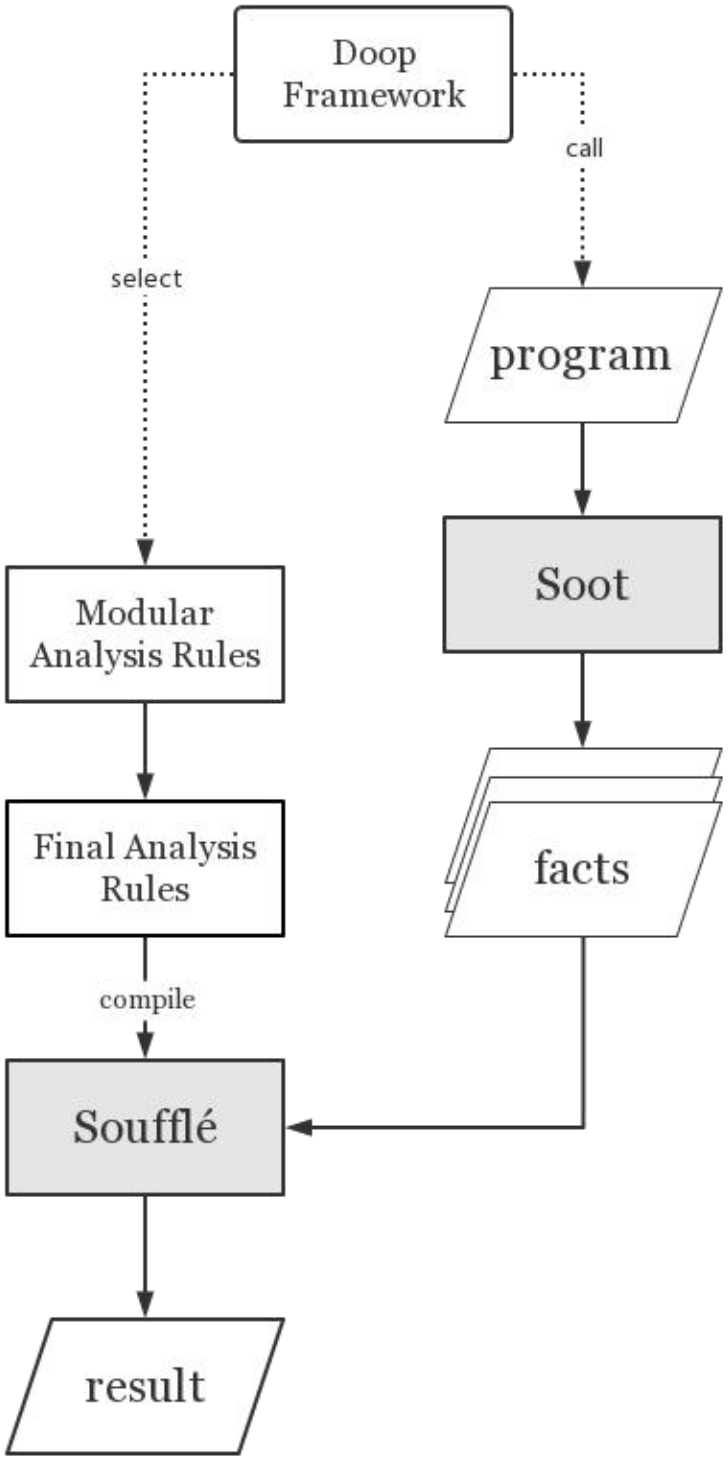
另外从细节上讲，Doop 框架实际上还适配了许多其他已有的静态分析工具的结果——譬如 Tamiflex 工具的结果也可以作为 Doop 的输入以进一步增强静态分析工具的精度。

Doop 自己的定义是一个软件框架。通常而言，软件框架是指一种抽象，就是说，这个框架可以根据第三方代码的更改来实现有选择的具体功能的更改。换句话说，它提供了一种构建和部署应用程序的标准方法，并且是一种通用的、可重用的软件环境。

一般而言，从软件工程角度而言，框架程序与普通的程序类库的区别体现在如下几点：控制反转、默认行为、可拓展性、不可修改的框架代码。

Doop 作为一个静态分析框架，他要解决的任务就是对静态分析中大量的底层实现细节做封装，为了达到这个目标，Doop 框架实际上是这样做的：它针对 Java 程序，将现有的指针分析以及信息流分析以及污点分析之类的算法通过 Datalog 语言进行实现，而将 Datalog 程序的适配的大量细节交由自己的框架代码负责。

以下我们更加详细的来说明这一点，首先从框架代码开始。在这里我们通过一个具体程序运行的流程来更加形象的展示这些内容。



正如上图所示的，灰色框表示这是 Doop 框架所引用的其他程序。在默认也是最

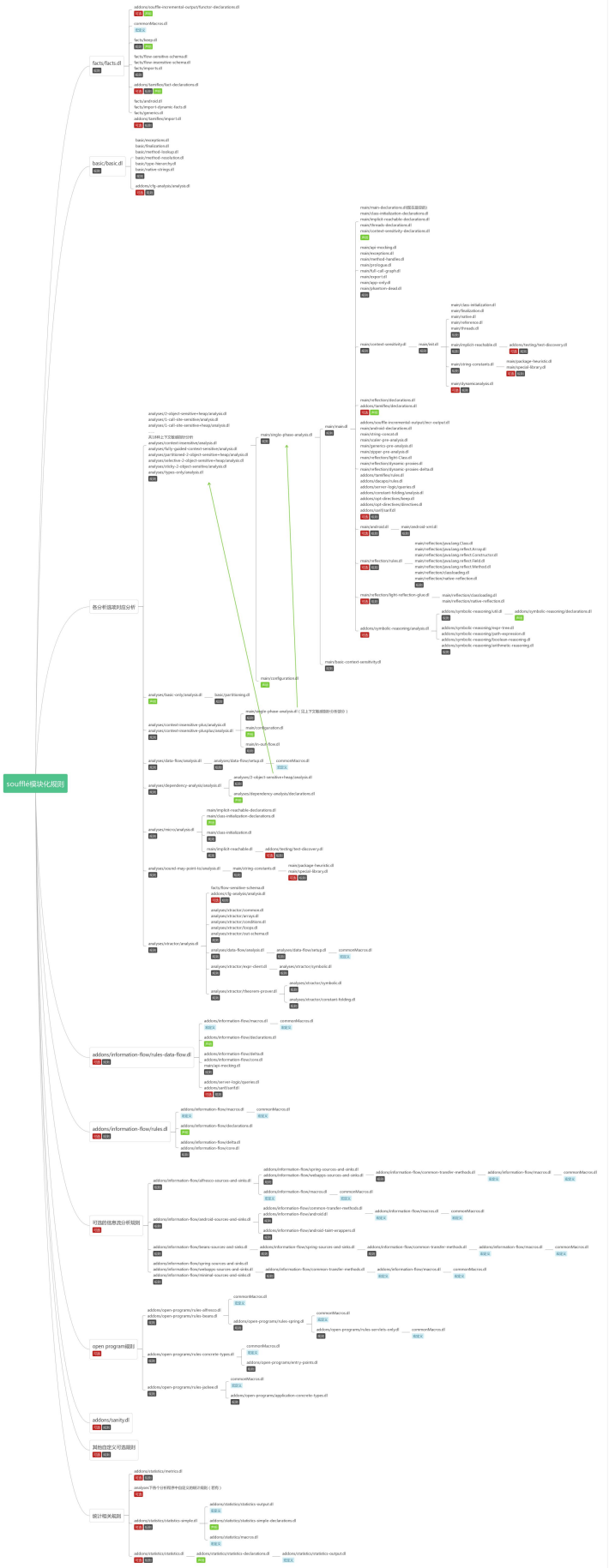
常用的的 Doop 配置下，这主要是指 Soot 和 Soufflé外部两个组件。Soot 前端组件可以通过运行配置切换到 Dalvik 可执行代码实验性 fact 生成器或 WALA 两个不同的前端上；而 Soufflé Datalog 程序求解器则可以通过配置切换为 Logicblox Datalog 程序求解器或者 DDLog Datalog 程序求解器。

在运行中，框架将控制所选定的前端模块根据预设的规则将程序中信息编码为后续 Datalog 程序中的 fact，具体来讲，以最常用的 Soot 模块为例，通过 Soot 库将字节码解析为 Soot 内部数据形式，并且以此来收集信息，例如某一个 Java 类文件中的类中有什么方法，他的父类有哪些，在方法中它调用了什么方法的签名……这些信息将被转换为 Datalog 程序所需要的格式（这个格式可能与对应的求解器相关，但是在程序运行时刻，这些信息是已经被配置的）。

Doop 将根据具体的分析选项，动态的从一组模块化的 Datalog 程序中构建对应此次分析的 Datalog 程序规则，并最终将这种程序规则形成输出，并且进一步交给下一步的 Datalog 求解器进行分析工作。

Datalog 是一种声明性逻辑编程语言，在语法上是 Prolog 的子集。Prolog 起源于一阶逻辑，一种形式逻辑，与许多其他编程语言不同，Prolog 主要用作声明性编程语言：程序逻辑用关系表示，表示为事实和规则。通过对这些关系运行查询来启动计算。与 Prolog 不同，Datalog 程序的语句可以按任何顺序进行说明。此外，有限集上的 Datalog 查询保证会终止，因此 Datalog 没有 Prolog 的 cut 运算符。这使得 Datalog 成为一种完全声明性的语言。

从 Doop 作者的角度，他认为这一点才是最关键的，Doop 是一个纯粹的使用声明式语言定义的指针分析框架，这在当时是开创性的——在此之前，所有的指针分析框架在编写指针分析程序时使用的都是命令式的语言。


souffle模块化规则
facts/facts.dl
basic/basic.dl
各分析选项对应分析
addons/information-flow/rules-data-flow.dl
addons/information-flow/rules.dl
可选的信息流分析规则
open programl规则
addons/sanity.dl
其他自定义可选规则
统计相关规则

以上这张图是当前版本的Doop程序中相关的模块化Datalog程序的一个结构的示意，明显可以从这里看到，在当前版本中，Doop 已经有了一个相当全面的声明式的分析程序组了，不同精度级别的指针分析程序也是相当完整。

此外，由于 Datalog 模块化声明的特性，我们也可以相当容易的根据我们自己的静态分析需求，编写出自己的规则——无需完全掌握复杂的 Soot 前端知识，也无需对如何编写一个高度优化的分析算法中的工程细节感到困惑，我们可以使用简单的声明式的 Datalog 语言写出我们想要的分析规则，将上面的相关细节交给 Doop 框架以及其中的 Datalog 求解程序的优化器自动完成。

总之，这个框架所采用的方式彻底将分析程序的逻辑和可能与某个语言特性相关的工程实现中的细节分离。

当然，在这里采用 Datalog 语言这种纯粹声明式的语言从静态分析建模的角度上也有一定问题：譬如由于 Datalog 的语言特性，在进行带上下文精度的指针分析时这里不支持无上界的情况，分析中上下文的深度必须是有界的。这是因为在 Datalog 语言中缺少构造函数。

但是这个问题并不会在实践中有太大的影响。因为其他精度增强，例如上下文敏感堆抽象，已被证明是比无限数量的上下文更好。

因此采用这种纯声明式的分析语言方式可以做到和之前的非声明式的语言功能上等价。

将程序分析逻辑和实现分离后还有另外一个好处：这样可以对 Datalog 程序进行方便的优化。对于 Datalog 语言的初学者，可以将此项工作基本交给 Datalog 求解器模块——例如专为复杂的指针分析规则设计的 Soufflé引擎中的自动优化组件；而对于 Datalog 专家，也有可以通过人工的跨规则优化（引入新的 Datalog

数据库索引）来取得进一步的优化。

这个特点在相当多的类似程序开发中有所体现——例如 python、java 这类语言，通过 JIT 可以自动化的优化程序、而对于那些对服务延时最关键的子程序，则可以通过专家通过更加底层的写法（cpp，优化 cache miss 等方法）可以进一步获得更好的效率。

更重要的是，这种方式也为未来 Datalog 求解器的未来可能的进一步的优化留出了升级空间——只要语法结构保持兼容，就可以完全无感的升级，从而从更先进的 Datalog 求解器的更高效率或者更新的特性中获益。

实际上这种事情已经发生过一次了，在 Doop 最开始的时候，它所使用的是获取了特殊许可的商用的 Logicblox Datalog 引擎。但是当 Soufflé引擎这个专为静态程序分析这个特殊问题的情景做出优化的引擎出现后，Doop 就将现有的规则转换到了 Soufflé引擎上，并且通过更换引擎获得了 4 倍的加速。

此外，现在的 Doop 框架也支持 DDLog 引擎，它的特点是支持增量求解，增量特性在一些静态分析的应用场景中可以大大节省时间。

这就是使用 Datalog 语言将分析逻辑与具体求解分开的具体一些的好处。

Doop 框架是一个有效的静态程序分析框架，从更细节的代码结构看，它分为了生成器模块（负责调用例如 Soot 类库生成信息）、规则模块（主要是模块化的 Datalog 规则）以及一个控制器模块（负责控制整个程序的运行流程）。由于这个框架的前端主要是解析命令行参数，所以前端模块集成在了控制器模块之中。具体适配各个组件是通过控制器模块中不同的逻辑代码实现的。

整体上看，由于 Doop 各个组件都没有考虑过在大规模分布式平台上运行，因此 Doop 框架作为一个整体也只有单机运行的能力。所以这里并没有什么微服务架

构啊、事件驱动啊这些时髦的名词。但我们也必须承认，它是一个相当模块化且易于维护与进一步拓展的程序。

如果从软件架构层面的再次反思这个程序，我从中至少学到了这一点——有时可以使用一些新的语言例如 Groovy 语言来更方便的编写代码，在 Doop 中充分运用了这一点。