

JAnalyzer基础构件开发文档

Developing Documents of JAnalyzer Fundamental
Components

周晓聪(isszxc@mail.sysu.edu.cn)
中山大学计算机科学系, 广州 510275

2017年12月3日

版权所有，翻印必究

目录

目录	i
第一章 引言	1
1.1 JAnalyzer平台背景	1
1.2 JAnalyzer基础构件功能概述	1
1.3 本文档的组织与结构	2
1.4 其他说明	3
第二章 抽象语法树生成构件	5
2.1 抽象语法树生成的需求分析	5
2.1.1 需求概述	5
2.1.2 用例模型	6
2.1.3 领域模型	8
2.2 抽象语法树生成的设计	9
2.2.1 类的职责与属性	9
2.2.2 用例实现	11
2.3 抽象语法树生成构件的实现	14
2.3.1 源代码文件类SourceCodeFile	15
2.3.2 源代码文件集类SourceCodeFileSet	16
2.3.3 源代码文件集迭代器类SourceCodeFileSetIterator	18
2.3.4 源代码位置类SourceCodeLocation	18
2.4 抽象语法树生成构件的使用	19
第三章 名字表生成构件的需求分析	23
3.1 需求概述	23
3.2 领域模型	23
3.2.1 名字定义	24
3.2.2 名字作用域	26
3.2.3 名字定义与名字作用域	27
3.2.4 源代码位置	30
3.2.5 名字定义的属性	30

3.2.6 名字作用域的属性	34
3.2.7 程序中的名字定义、作用域和名字引用	36
3.2.8 名字引用及其属性	40
3.2.9 名字引用组	43
3.2.10 名字引用的解析	47
3.2.11 名字表生成的领域模型总结	51
3.3 用例模型	52
3.3.1 名字定义和作用域的生成	53
3.3.2 名字定义的查找、遍历与访问	54
3.3.3 名字作用域的查找、遍历与访问	54
3.3.4 名字引用的生成与使用	55
3.3.5 名字表与抽象语法树	56
3.3.6 名字表生成构件功能总结	56
3.3.7 名字生成构件领域模型补充	57
第四章 名字表生成构件的设计	59
4.1 名字定义与名字作用域	59
4.1.1 枚举-名字作用域类别和名字定义类别	59
4.1.2 接口-名字作用域	60
4.1.3 抽象类-名字定义	60
4.1.4 类-程序包定义	62
4.1.5 类-编译单元作用域	62
4.1.6 抽象类-类型定义	64
4.1.7 类-详细类型定义	65
4.1.8 类-导入类型定义和导入静态成员定义	67
4.1.9 类-枚举类型定义和枚举常量定义	68
4.1.10 类-类型参数定义	68
4.1.11 类-方法定义	70
4.1.12 类-字段定义和变量定义	72
4.1.13 类-系统作用域	73
4.1.14 类-局部作用域	74
4.1.15 名字定义和名字作用域相关类的设计小结	76
4.2 名字引用	76
4.2.1 枚举-名字引用类别	76
4.2.2 类-名字引用	77
4.2.3 类-程序包引用和文字引用	78
4.2.4 类-类型引用及其子类	79
4.2.5 类-方法引用	81
4.2.6 类-值引用	82

目录	iii
4.2.7 抽象类-名字引用组及其子类	83
4.2.8 名字引用解析过程	87
4.2.9 名字引用相关类设计小结	89
4.3 名字表使用	89
4.3.1 类-名字表访问器及其子类	90
4.3.2 类-名字定义过滤器和名字作用域过滤器	91
4.3.3 类-名字表管理器	92
4.3.4 类-名字表抽象语法树桥接器	93
4.4 名字表生成	94
4.4.1 Java程序语法结构	94
4.4.2 类-名字表生成器及相关类	98
4.4.3 类-名字定义生成器及相关类	101
4.4.4 类-名字引用生成器及相关类	101
4.5 用例实现	104
第五章 名字表生成构件的实现	109
5.1 名字作用域	109
5.1.1 接口NameScope	110
5.1.2 类SystemScope	111
5.1.3 类CompilationUnitScope	113
5.1.4 类LocalScope	114
5.2 名字定义	116
5.2.1 类NameDefinition	118
5.2.2 类TypeDefinition	120
5.2.3 类DetailedTypeDefinition与类AnonymousClassDefinition	122
5.2.4 类EnumTypeDefinition与类EnumConstantDefinition	125
5.2.5 类ImportedTypeDefinition与类ImportedStaticMemberDefinition	126
5.2.6 类TypeParameterDefinition	129
5.2.7 类MethodDefinition和类AutoGeneratedConstructor	133
5.2.8 类FieldDefinition	136
5.2.9 类VariableDefinition	137
5.2.10 类PackageDefinition	138
5.3 名字引用	140
5.3.1 类NameReference	140
5.3.2 类TypeReference	144
5.3.3 类ParameterizedTypeReference	146
5.3.4 类NamedTypeReference和类QualifiedTypeReference	146
5.3.5 类IntersectionTypeReference和类UnionTypeReference	148
5.3.6 类WildcardTypeReference	150

5.3.7 类MethodReference	151
5.3.8 类ValueReference	151
5.3.9 类LiteralReference	153
5.3.10 类PackageReference	154
5.3.11 类NameReferenceGroup及其子类	154
5.4 名字表使用	157
5.4.1 名字表访问器类	157
5.4.2 名字表过滤器类	162
5.4.3 类NameTableManager	165
5.4.4 类NameTableASTBridge	167
5.5 名字表生成类	169
5.5.1 类CompilationUnitFile	171
5.5.2 类TypeASTVisitor	171
5.5.3 类BlockASTVisitor	172
5.5.4 类BlockDefinitionASTVisitor	174
5.5.5 类BlockReferenceASTVisitor	175
5.5.6 类ExpressionASTVisitor	177
5.5.7 类ExpressionDefinitionASTVisitor	178
5.5.8 类ExpressionReferenceASTVisitor	179
5.5.9 类NameTableCreator	180
5.5.10 类NameDefinitionCreator	182
5.5.11 类NameReferenceCreator	182
5.5.12 类ImportedTypeManager	184
第六章 名字表生成构件的使用	187
6.1 打印所有名字表实体	187
6.2 打印所有名字定义	188
6.3 打印所有名字作用域	190
6.4 打印所有名字引用	191
6.5 打印引用给定名字定义的所有名字引用	192
6.6 通过控制流图打印名字定义和名字引用	193
6.7 更为复杂的名字表使用简介	195
6.8 名字表构件使用总结	198
第七章 其他程序包简介	199
7.1 概述	199
7.2 控制流图的生成与使用	200
7.2.1 程序视图基础类	200
7.2.2 控制流图的存储	201
7.2.3 控制流图的生成	205

目录

v

7.2.4 基于控制流图的数据流分析	209
参考文献	219

第一章 引言

1.1 JAnalyzer平台背景

为开展面向对象软件理解与分析的研究，我们需要开发一个平台实现面向对象软件理解与分析的基本功能，并探索、试验与评价面向对象软件理解与分析的方法与技术。

我们选择Java语言实现平台，以Eclipse为开发环境，而且而以Java语言实现的面向对象软件为理解和分析对象，因为Java语言和Eclipse具有很好的开放性。我们将整个平台称为**Java语言源代码理解与分析平台**(Platform for Understanding and Analyzing Java Source Codes)，简称**JAnalyzer**。

JAnalyzer可完成对Java软件源代码的编译、名字表的生成和软件结构的提取，并在此基础上支持对Java源代码的各种静态分析技术，例如，程序控制流图和软件依赖图的生成与可视化、软件结构度量与质量评价、软件版本变化检测与软件演化分析、软件克隆检测和代码坏味检测等等。

JAnalyzer的目标是实现一个轻量级、可扩展、可重用、开放源代码的面向对象软件分析与理解平台。目前存在一些软件分析与理解工具，如Understand¹、JHawk²、JArchitect³等。但功能比较完善，界面设计良好的工具往往是商业软件，无法免费用于学习和研究，而一些开源的，或者由研究人员实现的工具则往往只有特定的部分功能，而且不容易重用，也很难满足我们对软件理解与分析技术的探索与评价。因此，我们自己实现一个这样的平台，以满足我们对软件理解与分析技术研究的需要。

1.2 JAnalyzer基础构件功能概述

JAnalyzer的基础构件指完成抽象语法树生成、名字表生成和软件结构提取这三个功能的程序包及其类。这三个功能是对源代码做静态分析的基础，任何对软件源代码的理解和分析都需要在生成抽象语法树和名字表的基础上，基于软件的结构完成。

具体来说：

(1) **抽象语法树生成**是指将源代码文本进行语法分析，得到表示源代码的抽象语法树(Abstract syntax tree, AST)。计算机程序不同于普通的文本，对计算机程序的理解通常不能基于程序文本，而要基于程序的语法。当然为了生成源代码的抽象语法树，我们必须读取源代码文本内容，并进行管理，因为有时对程序的分析（例如计算源点啊行数）也需要基于源代码文本内容。一个软件通常

¹<https://scitools.com/>

²<http://www.virtualmachinery.com/jhawkprod.htm>

³<http://www.jarchitect.com/>

含有多个源代码文件，而且我们可能还需要同时管理一个软件项目的多个版本。

(2) **名字表的生成**是指对源代码中由源代码编写人员自定义的名字进行管理，这些名字也就是指程序编写人员用来命名程序的各种元素，如程序包、类型、字段（类型数据成员）、方法（类型方法成员）、变量（对象）等的标识符。程序员通过命名各种程序元素，并引用这些名字而实现程序所需要完成的功能。我们使用名字表收集这些名字的定义，从而可确定在程序中引用这些名字时具体是引用程序的哪个元素。一个标识符到底是对程序元素的名字定义还是对程序元素的名字引用取决于它出现在程序的哪种语法结构中，因此名字表的构件必须基于抽象语法树，而非纯程序文本。

(3) **软件结构的生成**是指在确定程序中名字定义和引用的基础上，确定程序中各种元素之间的关系。这种关系通常包括元素之间的组成关系，即一个程序元素是由哪些程序元素构成，例如一个程序包有哪些类和接口，一个类或接口有哪些成员等等。另一种常见的关系是使用（依赖）关系，即一个程序元素要使用另外一个程序元素，例如一个类的方法要调用另外一个类的方法等等。在面向对象软件中，特别是在Java软件中，还有类之间的继承关系，接口之间的继承关系以及类对接口的实现关系。所有这些关系构成了软件的结构。

实际上，无论是抽象语法树、名字表还是软件结构，生成的目的都是为了给JAnalyzer的其他构件使用，因此JAnalyzer基础构件除了完成它们的生成外，还需要提供类及其方法供其他构件使用抽象语法树、名字表和软件结构。因此，实际上基础构件要完成的功能包括语法树、名字表和软件结构的生成、存储和访问。

JAnalyzer的基础构件主要是提供API给其他构件使用，因此比较少涉及界面。在下面分析和建模过程中，我们将被分析的基础构件分别称为“**抽象语法树生成构件(AST Creating Component)**”、“**名字表生成构件(Name Table Creating Component)**”和“**软件结构提取构件(Software Structure Extracting Component)**”。我们将使用（基础）构件的JAnalyzer构件称为“**构件使用者(Component Client)**”。

这里的“构件”的含义实际上就是JAnalyzer的子系统(sub-system)，不过由于JAnalyzer本身分析的对象也是软件系统，因此我们将“系统”、“子系统”这样的词汇用来指JAnalyzer的分析对象，而对于JAnalyzer本身我们用“平台”、“构件”来命名它以及它的子系统。

1.3 本文档的组织与结构

本文档是JAnalyzer基础构件的开发文档。JAnalyzer的基础构件实现对Java软件源代码的编译（抽象语法树的生成）、名字表的生成。这些都是要实现任何源代码静态分析技术所必须完成的功能。本文档说明这些构件的功能分析、构件设计、实现与使用。通过阅读本文档，JAnalyzer的开发人员可了解这些功能的设计和实现原理，并能在使用这些功能的基础上实现对Java软件源代码更复杂的理解与分析功能。

本文档按照面向对象软件开发规范，描述这些构件的需求分析、设计、实现与使用。在需求分析阶段，我们对各基础构件所要实现的功能做进一步细分，提取关键用例(use case)，并对关键用例的事件流进行说明，然后在功能需求和用例模型的基础上，提取实体及其主要属性，建立构件的领域模型。

在设计阶段，基于构件的领域模型和用例模型，将其中的实体转化为软件中的类的设计，使用类图描述类与类之间的关系，以及交互图、顺序图或/和活动图等描述类（的对象）之间如何进行交

互以支持关键用例的实现，然后对关键的类或复杂的类展开类的详细设计，包括关键数据结构及其存储方法、关键方法的算法的设计等。由于基础构件主要是提供一些API(application programming interface，应用程序编程接口)供其他构件使用，因此基础构件没有界面设计。

在实现阶段，我们根据构件的实际实现情况，给出实现构件功能的程序包、程序包的每个类及其成员，并对关键数据结构和算法的实现要点进行说明。

对于基础构件的使用，目前的实现中都提供了一些演示(或说测试)程序(类)，我们以这些程序为基础，概要说明基础构件所提供的一些关键API的使用方式。我们也将基于前面的用例模型，补充一些测试程序(类)，从而演示用例模型中所给出的关键用例如何实现。

由于名字表生成和软件结构提取这两部分涉及的概念、实体和类都比较多，为了快速找到这些构件中的类的设计与实现，我们将抽象语法树生成的需求、设计、实现和使用在一章介绍，而名字表生成和软件结构提取这两部分的需求、设计、实现和使用都单列一章进行说明。因此：

第二章 抽象语法树生成构件：2.1节给出抽象语法树生成构件的需求，包括功能概述、用例模型和领域模型；2.2、2.3和2.4节分别给出它的设计、实现与使用。

第三章 名字表生成构件的需求分析：3.1节给出名字表构件的需求概述；由于名字表生成部分设计的概念和实体比较多，而作为基础构件与构件使用者的交互比较少，因此3.2节先建立名字表构件的领域模型，围绕名字表的实体—名字定义、名字作用域和名字引用进行领域建模，3.3节再描述构件的用例模型，描述名字表的生成和使用。

第四章 名字表生成构件的设计：在名字表领域模型的基础，详细描述名字定义、名字作用域、名字引用、名字表使用以及名字表生成相关的类的设计，包括类的主要属性和职责，类与类之间的主要关系，然后使用顺序图描述这些类之间的交互，说明主要用例的实现。

第五章 名字表生成构件的实现：基于实际的实现，描述名字表生成构件的程序包组织、类的具体实现，给出类的属性(字段)和方法说明，并就其中的一些实现要点进行说明。

第六章 名字表生成构件的使用：这一章结合用例模型和名字表生成构件的演示程序说明构件使用者如何使用名字表生成构件。

第七章 其他程序包简介：这一章简单介绍了目前JAnalyzer平台所提供的其他程序包，主要是控制流图的生成和使用，以及基于控制流图的数据流分析。

使用基础构件进行开发的程序员可通过阅读每部分构件的需求分析了解基础构件所提供的功能及其涉及的一些重要概念，看是否可以满足自己的需求，然后可阅读每部分构件的使用了解每个功能的使用方式从而完成自己的开发。如果需要对基础构件的功能进行扩展，则需要进一步阅读每部分构件的设计和实现，以了解它们的具体设计与实现原理。

1.4 其他说明

本文档在描述构件的用例模型、领域模型、设计模型的时候参考了李运华所编著的《面向对象葵花宝典》一书中关于面向对象软件开发模型的构建方法。

本文档的UML模型视图使用楚凡科技公司⁴出品的免费软件“Trufun Plato 免费版V6[UML工具]”进行绘制。文档中所展示的模型全部使用中文名字，但实际上我们绝大多数模型都相应地给出

⁴<http://trufun.net/>

了英文名字，这些英文名字大多数都对应软件中的实际实现。虽然使用的中文名字，但我们在描述设计模型中动态模型（顺序图、交互图、活动图）的交互操作都严格使用相应类提供的职责。

本文档所描述的实现可能是在我们撰写本文档过程中描述完构件的需求和设计模型之后对已有实现进行重构之后的实现。我们保证本文档所描述的构件实现和使用方式与重构之后的构件最终版本一致。

最后需要注意的是，JAnalyzer目前放在OSChina的码云平台上：

访问地址: <https://gitee.com/zhouxiaocong/JAnalyzer>
用户名: isszxc@mail.sysu.edu.cn
访问密码: zxc2sysu.myja

第二章 抽象语法树生成构件

2.1 抽象语法树生成的需求分析

2.1.1 需求概述

基于软件理解与分析的需要，构件使用者对抽象语法树生成构件的功能需求主要包括：

1. 给定一个Java软件系统的一组源程序代码文件，创建所有文件的抽象语法树，并能获得所有抽象语法树的根节点；
2. 给定一个Java软件系统的一组源程序代码文件，创建某个指定文件的抽象语法树，并获得其根节点；

由于抽象语法树的生成必须读取源程序代码文件文本内容，抽象语法树生成构件必须管理源代码文件文本内容：

3. 给定一个Java软件系统，获得所有源代码文件文本内容；
4. 给定一个Java软件系统，获得指定源代码文件文本内容。

所以，构件使用者只要给定一个Java软件系统，就可获得该软件系统下所有或指定的源代码文件文本内容，为所有或指定源代码文件创建抽象语法树，获得所有或指定源代码文件抽象语法树的根节点。

注意，在Java语言中，每个源代码文件在语法上被称为一个编译单元(Compilation Unit)，抽象语法树的生成只能以编译单元为单位，而非以整个软件系统为单位。每个编译单元生成的抽象语法树都有一个根，基于根节点就可遍历抽象语法树的每个节点。

我们基于Eclipse JDT来完成Java源代码文件的编译，编译的结果就是该源代码文件的抽象语法树根节点。Eclipse JDT也提供了对抽象语法树的遍历。因此JAnalyzer的抽象语法树生成构件并不需要实现编译和生成抽象语法树的细节，其真正的功能在于管理一个Java软件系统的多个源代码文件的内容和抽象语法树，提供给构件使用者查找指定源代码文件的内容和抽象语法树根节点，或遍历所有源代码文件内容和抽象语法树根节点。

图2.1给出了抽象语法树生成构件的框架图，它以Java源代码文件集为输入，基于Eclipse JDT实现源代码的编译并生成抽象语法树(AST)根节点集。

由于软件源代码文件都是由操作系统的文件管理系统管理，因此所谓给定一个Java软件系统，实际上只要指定这个java软件系统源代码文件所在的起始路径，也即我们将这个起始路径下（包括其子目录）的所有Java源代码文件看做一个Java软件系统。或者说，JAnalyzer分析的是一个Java源代码文件集合(Source Code Set)，其所有文件都在操作系统文件管理系统的某个目录之下，这个目录称为这个源代码文件集的起始路径(Start Path)。

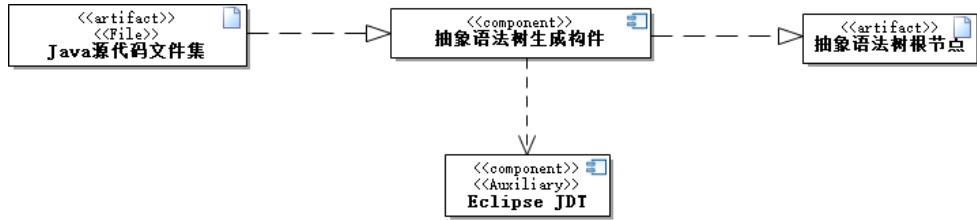


图 2.1 抽象语法树生成构件框架

2.1.2 用例模型

根据上面的分析，我们得到四个用例：(1) 获取指定源代码文件内容；(2) 获取指定源代码文件抽象语法树根节点；(3) 获取所有源代码文件内容；(4) 获取所有源代码文件抽象语法树根节点。图2.2给出了抽象语法树生成构件的用例图。

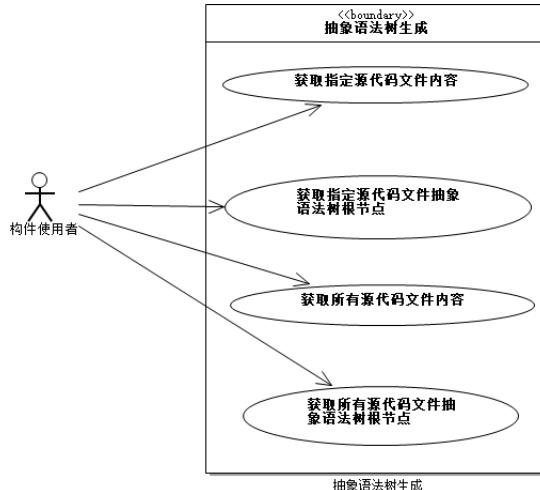


图 2.2 抽象语法树生成构件用例模型

表 2.1 用例1. 获取指定源代码文件内容

【用例名称】	获取指定源代码文件内容
【用例场景】	
参与者：	构件使用者
【用例价值】	构件使用者获得存有指定源代码文件内容的String对象
【用例描述】	<ol style="list-style-type: none"> 1. 构件使用者指定源代码文件集起始路径 <ol style="list-style-type: none"> 1.1 如果指定的起始路径不存在则返回空的(NULL)对象 2. 构件使用者指定相对于起始路径的源代码文件名 3. 返回存有指定源代码文件内容的String对象给构件使用者 <ol style="list-style-type: none"> 3.1 如果指定的源代码文件不存在则返回空的(NULL)对象 3.2 如果指定的源代码文件内容太大没有足够内存空间则返回空对象

表2.1、2.2、2.3 和2.4给出了这四个用例的描述。注意，我们在获得所有源代码文件内容，或所

表 2.2 用例2. 获取指定源代码文件抽象语法树根节点

【用例名称】 获取指定源代码文件抽象语法树根节点
【用例场景】
参与者：构件使用者
【用例价值】
构件使用者获得存有指定源代码文件抽象语法树根节点的对象
【用例描述】
<ol style="list-style-type: none"> 1. 构件使用者指定源代码文件集起始路径 <ol style="list-style-type: none"> 1.1 如果指定的起始路径不存在则返回空的(NULL)对象 2. 构件使用者指定相对于起始路径的源代码文件名 3. 构件使用者查询生成抽象语法树是否成功 <ol style="list-style-type: none"> 3.1 如果生成成功，则可获得指定源代码文件的抽象语法树根节点对象 3.2 如果生成不成功，则可获得指示该源代码文件编译错误的String对象

表 2.3 用例3. 获取所有源代码文件内容

【用例名称】 获取所有源代码文件内容
【用例场景】
参与者：构件使用者
【用例价值】
构件使用者可获得每个存有源代码文件内容的String对象
【用例约束】
不应该由于获取所有源代码文件内容而占有太多内存空间
【用例描述】
<ol style="list-style-type: none"> 1. 构件使用者指定源代码文件集起始路径 <ol style="list-style-type: none"> 1.1 如果指定的起始路径不存在则返回 2. 构件使用者遍历源代码文件集中的源代码文件 <ol style="list-style-type: none"> 2.1 构件使用者在遍历过程中可使用每个存有源代码文件内容的String对象

表 2.4 用例4. 获取所有源代码文件抽象语法树根节点

【用例名称】	获取所有源代码文件抽象语法树根节点
【用例场景】	
参与者:	构件使用者
【用例价值】	构件使用者获得存有指定源代码文件抽象语法树根节点的对象
【用例约束】	不应该由于获取所有源代码抽象语法树根节点而占有太多内存空间
【用例描述】	<p>1. 构件使用者指定源代码文件集起始路径 1.1 如果指定的起始路径不存在则返回</p> <p>2. 构件使用者遍历源代码文件集中的源代码文件 2.1 对每个源代码文件查询生成抽象语法树是否成功 2.1.1 如果生成成功，则可使用该源代码文件的抽象语法树根节点对象 2.1.2 如果生成不成功，则可获得指示该源代码文件编译错误的String对象</p>

有源代码文件抽象语法树根节点的时候，由于内存限制，构件使用者只能对源代码文件集中的文件进行遍历，逐一使用文件内容或抽象语法树根节点。因为这种遍历，在源代码文件集中我们有所谓的“**当前代码文件**”的概念。根据上面的用例描述，我们可得到，抽象语法树生成构件至少应提供的功能，如表2.5。

表 2.5 抽象语法树生成构件功能列表

编号	功能描述	涉及用例
01	设定源代码文件集起始路径	用例1、用例2、用例3、用例4
02	在源代码文件集中查找指定文件	用例1、用例2
03	遍历源代码文件集中所有文件	用例3、用例4
04	返回当前源代码文件的文本内容	用例1、用例3
05	查询当前源代码文件抽象语法树生成是否成功	用例2、用例4
06	返回当前源代码文件的抽象语法树根节点	用例2、用例4
07	返回当前源代码文件的编译错误信息	用例2、用例4

2.1.3 领域模型

基于上述功能概述和用例模型，我们可得到这一构件所涉及到的实体及其主要属性：

- 源代码文件集(SourceCodeFileSet)，主要属性有：(1) 起始路径(startPath)；(2) 当前源代码文件(currentSourceCodeFile)；
- 源代码文件(SourceCodeFile)，一个源代码文件集有多个源代码文件构件。源代码文件的主要属性有：(1) 相对源代码文件集起始路径的源代码文件名(fileName)；(2) 源代码文件内容(fileContent)；(3) 源代码文件抽象语法树根节点(rootASTNode)；(3) 源代码文件编译错误信息(parsingErrorMessage)。

图2.3给出了抽象语法树生成构件的领域模型，其中还使用到了一个辅助实体：[抽象语法树节点](#)，它由Eclipse JDT 提供，这里暂时不考虑它的属性。

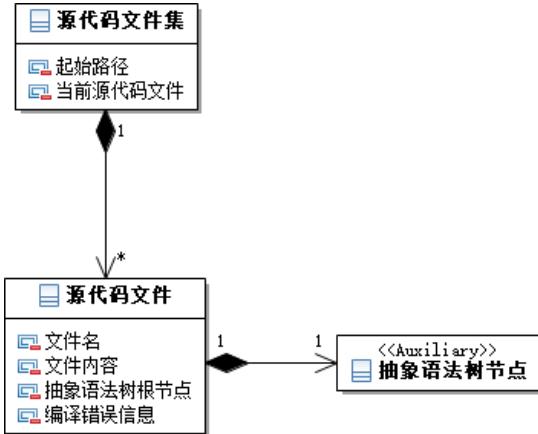


图 2.3 抽象语法树生成构件领域模型

2.2 抽象语法树生成的设计

2.2.1 类的职责与属性

基于抽象语法树构件的功能、用例模型和领域模型，可以看到，抽象语法树生成构件至少有两个类：类源代码文件 (`SourceCodeFile`)和类源代码文件集(`SourceCodeFileSet`)。先考虑源代码文件 (`SourceCodeFile`)这个类。它的基本属性已经在领域分析中给出，我们考虑它应该具备的职责。显然，表??中列出的抽象语法树生成构件中的功能编号04到07都应该属于这个类的职责(responsibility, RP)：

- (RP1). 返回文本内容(`getContent()`)。
- (RP2). 查询抽象语法树生成是否成功(`hasCreatedAST()`)。
- (RP3). 返回抽象语法树根节点(`getASTRoot()`)。
- (RP4). 返回编译错误信息(`getParsingErrorMessage()`)。

为了完成这些职责，类源代码文件(`SourceCodeFile`)还应该有如下职责：

- (RP5). 装载文本内容(`loadContent()`)：从文件管理系统中读入该文件的内容到内存中。
- (RP6). 生成抽象语法树(`createAST()`)：调用Eclipse JDT提供的API生成源代码文件的抽象语法树。

读入文件内容和生成抽象语法树可能花费很长时间，但一个文件的内容和抽象语法树又可能被多次访问，因此最好是缓冲读入的内容和生成的抽象语法树，因此这个类的属性(attribute, AT)有：

- (AT1). 文件名(`fileName`)：注意是相对于源代码文件集起始路径的文件全名。
- (AT2). 文件内容(`fileContent`)：用于缓冲源代码文件内容。
- (AT3). 抽象语法树根节点(`rootASTNode`)：用于存放已经生成的抽象语法树根节点。
- (AT4). 编译错误信息(`parsingErrorMessage`)：用于存放生成抽象语法树时得到的编译错误信息。

但是当有许多源代码文件类缓冲文件内容和抽象语法树时，可能会造成系统内存不足，因此需要在适当时候释放文件内容和抽象语法树所占用的内容，因此类源代码文件(`SourceCodeFile`)还应该提供以下职责：

- (RP7). 释放文本内容(`releaseContent()`)：释放文本内容所占用的缓存空间；
- (RP8). 释放抽象语法树(`releaseAST()`)：释放抽象语法树所占用的缓存空间。

但这些存储空间何时释放则由构件使用者决定。

表??中列出的抽象语法树生成构件中的功能编号01到03应该属于类源代码文件集(`SourceCodeFileSet`)的职责：

- (RP1). 设定起始路径(`setStartPath()`)。
- (RP2). 查找指定源代码文件(`findSourceCodeFile()`)。

对于遍历源代码文件集中所有文件这个功能，不是单一职责可以完成。我们考虑使用[迭代器](#)设计模式，将遍历所有源代码文件这项功能从这个类独立出来，因此需要有一个迭代器(`Iterator`)类，这样类源代码文件集(`SourceCodeFileSet`)只需要提供以下职责即可：

- (RP3). 返回源代码文件集迭代器(对象)(`iterator()`)。

为快速查找指定源代码文件，需要将起始路径下的所有源代码文件名缓存起来。我们考虑使用映射(`map`)来存放源代码文件集中源代码文件名与源代码文件(类的对象)之间的映射，这样类源代码文件集(`SourceCodeFileSet`)的属性有：

- (AT1). 起始路径(`startPath`)。
- (AT2). 源代码文件名与源代码文件映射(`fileMap`)。

若文件的遍历由迭代器完成，则我们无需当前源代码文件(`currentSourceCodeFile`)这个属性。为了构建源代码文件名和源代码文件之间的映射，类源代码文件集(`SourceCodeFileSet`)需要提供以下职责：

- (RP4). 装载所有源代码文件名(`loadAllFiles()`)：装入起始路径下所有源代码文件名(.java文件名)，建立与源代码文件之间的映射。

源代码文件集的迭代器(`Iterator`)类与所有其他迭代器具有相同的职责：

- (RP1). 判断是否有下一个元素(`hasNext()`)。
- (RP2). 返回下一个元素(`next()`)：这个返回的类源代码文件(`SourceCodeFile`)的对象就是源代码文件集的当前源代码文件。

为了实现这些职责，迭代器持有源代码文件集中的源代码文件名与源代码文件映射即可：

- (AT1). 源代码文件名与源代码文件映射(`fileMap`)：要遍历的源代码文件集中的映射的副本。

最后，当源代码文件集中使用源代码文件名与源代码文件(对象)的映射后，类源代码文件(`SourceCodeFile`)无需再保持源代码文件名这个属性(因为它们之间的映射是一一对应的，无需重复存放)。而为了方便装入文件内容，我们在这个类中保持一个操作系统的文件句柄(即Java语言中的类`File`的对象)：

- (AT1'). 操作系统文件句柄(`fileHandle`)。

注意，实际上，通过这个句柄我们也可获得相应文件的包括文件名在内的各种信息。基于文件句柄，类源代码文件(`SourceCodeFile`)还可提供如下职责：

- (RP9). 返回文件单元名(`getUnitName()`)：由于文件有全名、相对名和简单名(不含任何路径信息的文件名)，而相对于源代码文件集起始路径的相对名对我们最为重要，因此我们将这个相对

名特别地称为**单元名**(unit name), 这个称呼来源于Java语言的编译单元(compilation unit), 一个源代码文件对应一个编译单元。

(RP10). 返回文件全名(getFullName()): 返回文件全名(包括所有路径信息在内的文件名)。

(RP11). 返回文件简单名(getSimpleName()): 返回文件简单名(不包括任何路径信息的文件名)。

根据上述设计, 我们得到抽象语法树生成构件的类图, 如图2.4。

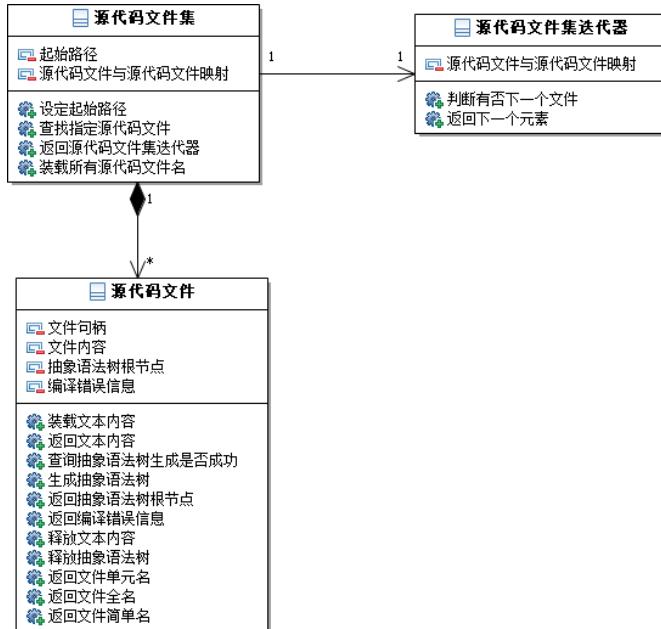


图 2.4 抽象语法树生成构件的类图

2.2.2 用例实现

基于前面类的设计, 我们检查是否已有的类及其提供的职责能实现前面给出的用例。为此使用顺序图(sequence diagram)描述类的对象之间如何交互以实现用例。

图2.5给出了实现用例1 获取指定源代码文件内容的顺序图。可以看到, 该用例可通过类的对象如下交互完成:

1. 构件使用者在设定起始路径时创建源代码文件集对象, 当然也可在创建源代码文件集对象之后再设定起始路径;
2. 源代码文件集对象装载所有源代码文件名, 这时也为每个源代码文件名创建了源代码文件对象, 并建立它们之间的映射;
3. 构件使用者给出指定的源代码文件单元名, 查找指定源代码文件(对象);
4. 构件使用者获得源代码文件对象之后, 使其返回文本内容, 这时源代码文件对象可能装载(如果原来没有缓存, 或者已经释放)文本内容;
5. 在使用文本内容之后, 构件使用者可在适当时候释放文本内容, 接触文本内容对内存的占用。

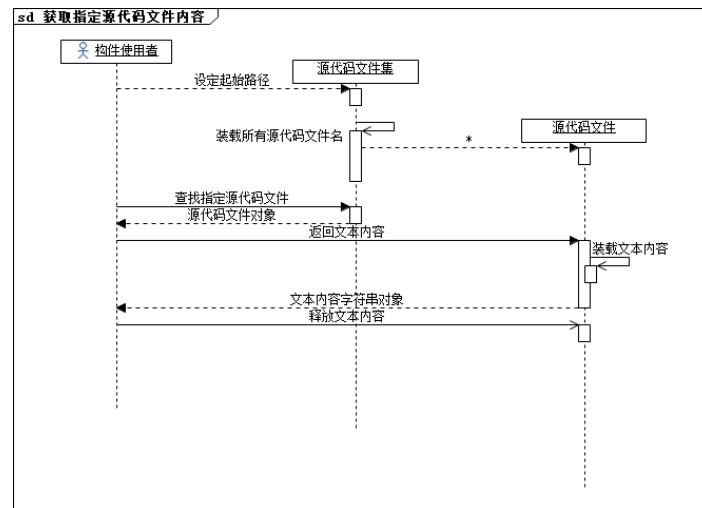


图 2.5 实现用例1 获取指定源代码文件内容的顺序图

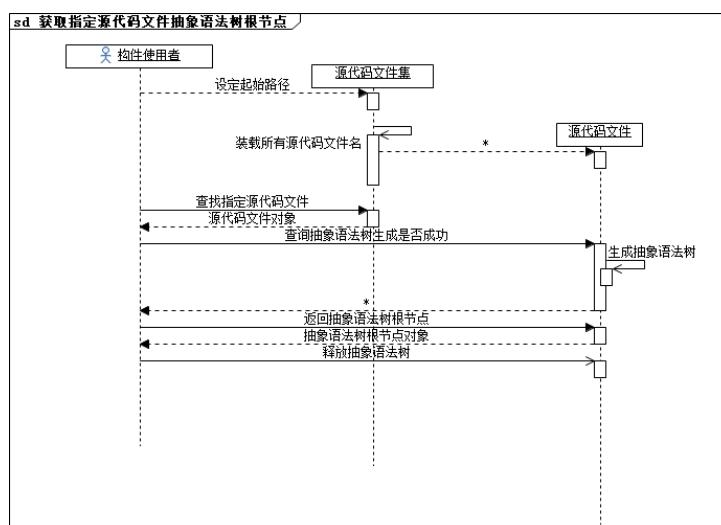


图 2.6 实现用例2 获取指定源代码抽象语法树根节点的顺序图

图2.6给出了实现用例2 获取指定源代码抽象语法树根节点的顺序图。可以看到，该用例的实现与用例1的实现类似：

1. 构件使用者在设定起始路径时创建源代码文件集对象，当然也可在创建源代码文件集对象之后再设定起始路径；
2. 源代码文件集对象装载所有源代码文件名，这时也为每个源代码文件名创建了源代码文件对象，并建立它们之间的映射；
3. 构件使用者给出指定的源代码文件单元名，查找指定源代码文件（对象）；
4. 构件使用者获得指定的源代码文件对象之后，查询员代码文件对象生成抽象语法树是否成功。该查询可能引起源代码文件对象生成抽象语法树（如果原来没有生成，或者已经释放抽象语法树）
5. 如果生成成功，则构件使用者可获得抽象语法树根节点对象；
6. 构件使用者在使用抽象语法树后，可在适当时候释放抽象语法树，以解除抽象语法树对内存的占用。

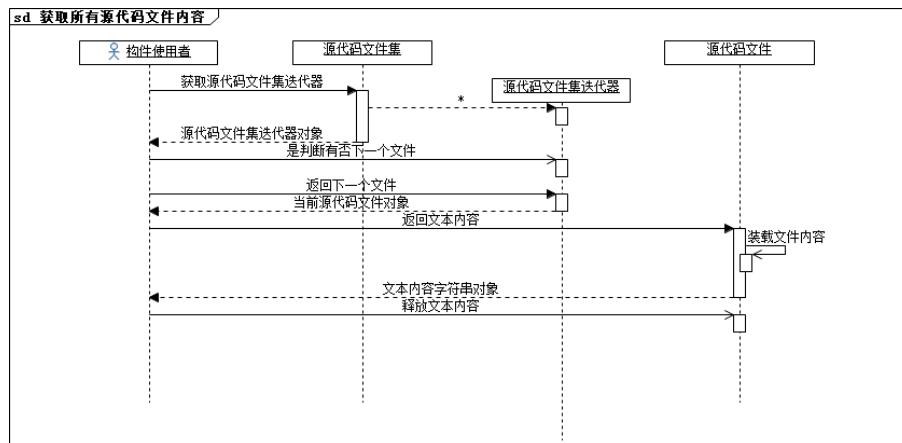


图 2.7 实现用例3 获取所有源代码文件文法内容的顺序图

图2.7给出了实现用例3 获取所有源代码文本内容的顺序图。由于设定起始路径并创建映射的交互方式同前两个用例（或者说，可以预先设定起始路径并创建映射，这样构件使用者可得到已经创建源代码文件名与源代码文件之间映射的源代码文件集对象），这里省略最开始的这两步，而从获得构件使用者获得源代码文件集对象之后开始描述：

1. 构件使用者获取源代码文件集迭代器；
2. 判断有否下一个文件，如果有则得到下一个文件（当前源代码文件对象）；
3. 构件使用者获得当前源代码文件对象后，使其返回文本内容，这时源代码文件对象可能装载（如果没有缓存，或者已经释放）文本内容；
4. 在使用文本内容后，构件使用者可在适当时候释放文本内容，接触文本内容对内存的占用。

可以看到，在获得当前源代码文件对象之后，这里第3、4步与用例1的第4、5步相同。图2.8给出了实现用例4 获取所有源代码抽象语法树根节点的顺序图。同样我们省略了获得源代码文件集对象之前的交互：

1. 构件使用者获取源代码文件集迭代器；
2. 判断有否下一个文件，如果有则得到下一个文件（当前源代码文件对象）；

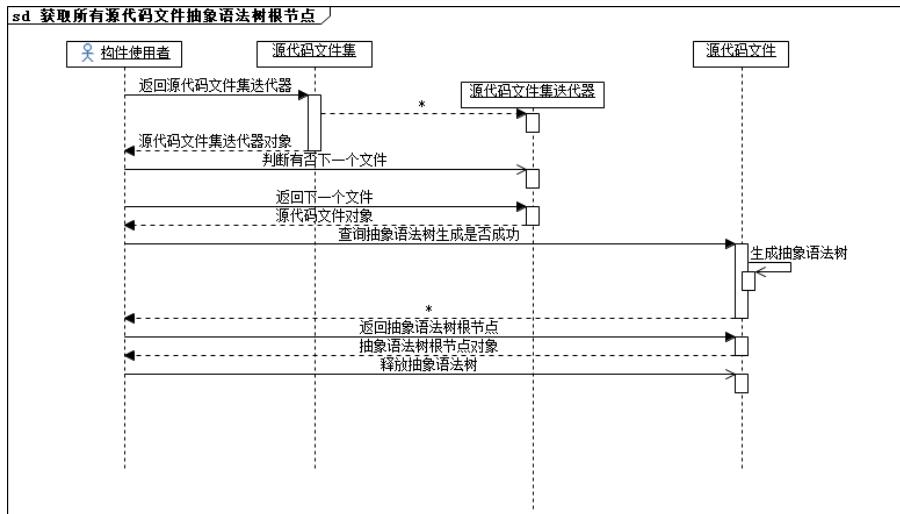


图 2.8 实现用例4 获取所有源代码抽象语法树根节点的顺序图

3. 构件使用者获得当前源代码文件对象后，查询源代码文件对象生成抽象语法树是否成功。该查询可能引起源代码文件对象生成抽象语法树（如果原来没有生成，或者已经释放抽象语法树）
4. 如果生成成功，则构件使用者可获得抽象语法树根节点对象；
5. 构件使用者在使用抽象语法树后，可在适当时候释放抽象语法树，以解除抽象语法树对内存的占用。

可以看到，这个顺序图的第1、2步与图2.7相同，而后面第3到第4步则与图2.6相同。

2.3 抽象语法树生成构件的实现

在编写此文档前，抽象语法树生成构件的功能主要由在程序包util中的类SourceCodeParser实现，但根据上面的设计我们对抽象语法树生成这一部分的构件进行了重构，新的类都在程序包sourceCodeAST中。这个程序包有四个Java 源代码文件（编译单元），如图2.9所示。

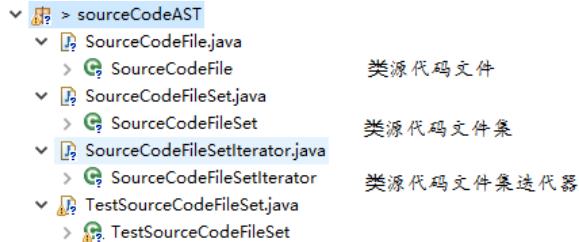


图 2.9 实现抽象语法树生成构件的程序包sourceCodeAST

编译单元SourceCodeFile.java包含源代码文件类SourceCodeFile, SourceCodeFileSet.java包含源代码文件集类SourceCodeFileSet, SourceCodeFileSetIterator.java包含源代码文件迭代器类SourceCodeFileIterator, 最后编译单元TestSourceCodeFileSet.java则简单演示了这些类的使用。

2.3.1 源代码文件类SourceCodeFile

成员	描述
fileContent	文件内容
fileHandle	文件句柄
hasParsingError	是否有编译错误
parsingErrorMessage	编译错误信息
rootASTNode	抽象语法树根节点
totalLines	文件总代码行数
totalSpaces	文件占用的总空间
createAST()	生成抽象语法树
getASTRoot()	返回抽象语法树根节点
getFileContent()	返回文件内容
getFileFullName()	返回文件全名
getFileHandle()	返回文件句柄
getParsingErrorMessage()	返回编译错误信息
getTotalLines()	返回文件总行数
getTotalSpaces()	返回文件总空间(以字节数计)
hasCreatedAST()	判断是否成功生成抽象语法树
loadContent()	装载文件内容
releaseAST()	释放抽象语法树(占用的空间)
releaseFileContent()	释放文件内容(占用的空间)

图 2.10 源代码文件类SourceCodeFile的实现

存放源代码文件内容及抽象语法树(根节点)信息的类源代码文件(SourceCodeFile)在编译单元SourceCodeFile.java中,如图2.10所示。可以看到,类SourceCodeFile的实现基本上遵循前面的设计,除了下面的小修改:

1. 没有实现返回文件单元名和返回文件简单名的方法,因为单元名只能在一个源代码文件集中才有意义,然后源代码文件类没有源代码文件集的起始路径信息,因此无法生成单元名。没有返回简单名的方法,是因为到目前为止还没有这方面的需求;
2. 增加了返回文件总行数、总空间(以字节计)的方法,这两个方法帮助构件使用者统计员代码行数和源代码字节数。

类SourceCodeFile的方法的实现都比较简单,值得一说的是方法loadContent()的实现,其关键代码如图2.11所示。这里使用类java.io.LineNumberReader读取源代码文件内容,以计算源代码文件的文本行数。最初我们曾经使用更通用的类java.util.Scanner逐行读入源代码文件内容,但是经过测试发现,这个类不能正确读入带有多国字符的源代码文件(其表现则为该文件在EditPlus这样的文本编辑器无法正常显示这些字符)。

类SourceCodeFile的方法createAST()为源代码文件生成抽象语法树,其关键代码如图2.12所示。可以看到,目前我们基于Java 1.8规范生成抽象语法树,实际的生成是基于Eclipse JDT的类org.eclipse.jdt.core.dom.ASTParser提供的方法。

类SourceCodeFile的方法loadContent()和createAST()都是私有方法,这意味着源代码文件内容的装载和抽象语法树的生成都是由类SourceCodeFile自己在需要的时候装载。简单地说,类SourceCodeFile的方法getFileContent()在需要的时候调用loadContent(),而方法hasCreatedAST()在需要的时候调用createAST()。

```

LineNumberReader reader = new LineNumberReader(new FileReader(fileHandle));
String line = reader.readLine();
StringBuffer buffer = new StringBuffer();
while (line != null) {
    buffer.append(line + "\n");
    totalLines = totalLines + 1;
    totalSpaces += line.length();
    line = reader.readLine();
}
reader.close();
fileContent = buffer.toString();

```

图 2.11 实现方法loadContent()的关键代码

```

ASTParser parser = ASTParser.newParser(AST.JLS8);
parser.setKind(ASTParser.K_COMPILATION_UNIT);

// For parsing the source code in Java 1.8, the compile options must be set!
Map options = JavaCore.getOptions();
JavaCore.setComplianceOptions(JavaCore.VERSION_1_8, options);
parser.setCompilerOptions(options);

parser.setSource(fileContent.toCharArray());
parsingErrorMessage = null;
hasParsingError = false;
rootASTNode = (CompilationUnit) parser.createAST(null);

```

图 2.12 实现方法createAST()的关键代码

由于Java程序的内存是由JVM自动管理，因此方法releaseFileContent()和releaseAST()并没有真正（马上）释放文本内容和抽象语法树占用的内存，这两个方法仅仅是类SourceCodeFile的成员fileContent和rootASTNode设成空指针(null)而已，易变JVM的垃圾回收系统能回收文本内容和抽象语法树占用的内存。

2.3.2 源代码文件集类SourceCodeFileSet

存放源代码文件集起始路径以及保存源代码文件单元名与源代码文件对象映射的类源代码文件集 (SourceCodeFileSet)在编译单元SourceCodeFileSet.java中，如图2.13所示，表2.6给出这个类成员的描述。



图 2.13 源代码文件集类SourceCodeFileSet的实现

类SourceCodeFileSet的大部分方法的实现也非常简单。值得说的是它的构造方法以一个字符串为参数，这个字符串可以是起始路径，也可以是一个完整的Java文件名，如果是一个Java文件名，则这个文件所在的目录为起始路径，但是整个源代码文件集将只有这一个文件，如果是一个起始路径，则整个源代码文件集包含该起始路径下（包含子目录下）的所有.java文件。

表 2.6 类SourceCodeFileSet 的成员描述

成员	描述
<code>fileMap</code>	文件单元名与源代码文件对象映射
<code>pathSeparator</code>	文件路径分隔符, 即“\”
<code>rootFile</code>	根文件
<code>startPath</code>	起始路径
<code>createFileMap()</code>	装载起始目录下所有源代码文件, 并建立映射
<code>findSourceCodeFileASTRootByFileUnitName()</code>	使用给定单元名查找抽象语法树根节点
<code>findSourceCodeFileByFileUnitName()</code>	使用给定单元名查找源代码文件对象
<code>findSourceCodeFileContentByFileUnitName()</code>	使用给定单元名查找源代码文件内容
<code>getAllJavaSourceFiles()</code>	读取给定目录下所有文件(目录)列表
<code>getFileNumber()</code>	返回源代码文件集中文件个数
<code>getFileUnitName(SourceCodeFile)</code>	给定源代码文件对象, 返回其单元名
<code>getFileUnitName(String)</code>	将一个源代码文件全名转换为单元名
<code>getSourceCodeFileMap()</code>	返回单元名与源代码文件对象映射
<code>getStartPath()</code>	返回源代码文件集起始路径
<code>getTotalLineNumbersOfAllFiles()</code>	返回所有文件的总行数
<code>getTotalSpacesOfAllFiles()</code>	返回所有文件总空间(以字节数计)
<code>iterator()</code>	返回一个源代码文件集迭代器对象
<code>releaseAllASTs()</code>	释放所有源代码文件对象的抽象语法树
<code>releaseAllFileContents()</code>	释放所有源代码文件的内容
<code>releaseAST()</code>	释放给定单元名的源代码文件的抽象语法树
<code>releaseFileContent()</code>	释放给定单元名的源代码文件的内容

类SourceCodeFileSet的构造调用私有方法`createFileMap()`装载起始目录下所有源代码文件(或装载由参数`rootFile`指定的单个.java文件), 并为每个文件确定单元名(即将文件全名中的起始路径部分替换为空串), 分配类SourceCodeFile对象, 并`put()`到映射`fileMap`。`fileMap`是一个TreeMap, 因此源代码文件集中存放的文件将按照单元名的字典顺序进行排序(这个序与文件在文件管理系统中的排序是一致的)。

私有方法`createFileMap()`调用`getAllJavaSourceFiles()`读取起始路径及其子目录下的所有.java文件, 为读取子目录下的文件, 方法`getAllJavaSourceFiles()`将针对子目录递归调用自己。这个方法使用内部类`JavaSourceFileFilter`对目录下的文件进行过滤, 也选取所有的.java文件及子目录。

类SourceCodeFileSet的方法`iterator()`返回一个源代码文件集迭代器, 以便构件使用者遍历源代码文件集中的源代码文件对象。类SourceCodeFileSet实现接口`java.lang.Iterable`, 因此可像遍历其他Java容器类一样使用`for-each`循环遍历SourceCodeFileSet中的源代码文件集对象。

类SourceCodeFileSet的公有方法`getFileUnitName(SourceCodeFile)`返回一个给定的源代码文件对象的单元名, 由于源代码文件对象无法访问源代码文件集的起始路径, 因此确定单元名的方法在类SourceCodeFileSet中。构件使用者在遍历所有源代码文件或查找指定源代码文件之后可调用这个方法获得该源代码文件对象的单元名。在调用这个方法时, 构件使用者需要保证这个源代码文件对象确实是该源代码文件集起始目录下的文件。因为, 这个公有方法实际上调用类SourceCodeFileSet的私有方法`getFileUnitName(String)`将基于源代码文件对象返回的文件全名中的起始路径替换为空串之后的串作为单元名。

2.3.3 源代码文件集迭代器类SourceCodeFileSetIterator

图2.14给出了源代码文件集迭代器类SourceCodeFileSetIterator的实现。这个类实现接口Iterator<SourceCodeFile>。



成员	描述
<code>currentFileUnitName</code>	当前源代码文件对象的单元名
<code>fileMap</code>	文件单元名与源代码文件对象映射
<code>fileUnitNameSet</code>	源代码文件集的文件单元名集合
<code>setIterator</code>	文件单元名集合的迭代器
<code>getCurrentFileUnitName()</code>	返回当前源代码文件对象的单元名
<code>hasNext()</code>	判断是否还有下一个源代码文件对象
<code>next()</code>	返回下一个源代码文件对象作为当前对象

图 2.14 源代码文件集迭代器SourceCodeFileSetIterator的实现

类SourceCodeFileSetIterator的构造方法以一个源代码文件集对象，即类SourceCodeFileSet的对象为参数，但实际上我们只在类SourceCodeFileSetIterator中存放了该源代码文件集中单元名与源代码文件对象映射，即其成员fileMap的一个副本（保存在类SourceCodeFileSetIterator自己的成员fileMap中），并得到该映射的键(key)集合，即源代码文件单元名集合，保存在成员fileUnitNameSet中，并得到该单元名集合的迭代器保存在成员setIterator中。

构件使用者使用源代码文件集迭代器的方法hasNext()判断是否存在下一个源代码文件对象，这个方法的内部实现实际上是调用setIterator的hasNext()方法。构件使用者使用源代码文件集迭代器的方法next()得到下一个源代码文件对象，这个方法的内部实际上是调用setIterator的next()方法得到下一个源代码文件对象的单元名，然后再从映射(fileMap)中得到对应的源代码文件对象作为返回值。

构件使用者在调用方法next()之后，可通过源代码文件集迭代器的方法getCurrentFileUnitName()获得由方法next()返回的（当前）源代码文件（对象）的单元名。但当构件者使用for-each循环遍历源代码文件集时，迭代器是隐含使用的，这时无法调用方法getCurrentFileUnitName()，但可使用类SourceCodeFileSet的方法getFileUnitName()获得源代码文件对象的单元名。

2.3.4 源代码位置类SourceCodeLocation

类SourceCodeLocation的实例代表一个源代码位置。虽然这个类在抽象语法树生成时并没有用到，但所有的源代码位置都是基于抽象语法树中的信息计算（而不是直接从源代码文本文件计算），而且被广泛用于名字表生成和控制流图生成，因此这个类放在程序包sourceCodeAST中。

根据我们的设计，源代码位置由编译单元名（不是全名，也不是简单名）、行号和列号构成，编译单元名在一个源代码文件集中是唯一的，因此再加上行号基本能确定一个名字表实体（名字定义、作用域或名字引用）的位置，或者控制流图中一个可执行点（通常是一条语句或条件表达式）的位置，即使在同一源代码行有相同的名字引用，通过（名字引用的起始）列号也能唯一区分。不过，源代码位置的列号通常并不完全精确，因为制表符所代码的字符数的原因，给出的列号可能与不同的源代码文件编辑器给出的列号可能不尽相同。源代码位置的行号通常不会有偏差，但有些文本文件编辑器在解释回车换行("\r\n")时会解释成两行（有些则解释成一行），所以也可能会有偏差，但



成员	描述
FILE_NAME_BEGINNER	分隔行列号与单元名的字符
LINE_COLUMN_SPLITTER	分隔行号与列号的字符
getEndLocation()	计算AST节点结束点源代码位置
getFileUnitNameFromId()	从唯一标识中提取单元名
getLocation()	将唯一标识串转换为对象实例
getStartLocation()	计算AST节点开始点源代码位置
column	列号
fileUnitName	编译单元名
lineNumber	行号
SourceCodeLocation(int, int, String)	
compareTo(SourceCodeLocation) : int	两个源代码位置进行有序比较
equals(Object) : boolean	两个源代码位置进行相等比较
getColumn() : int	返回列号
getFileUnitName() : String	返回编译单元名
getLineNumber() : int	返回行号
getUniqueId() : String	返回源代码位置的唯一标识
hashCode() : int	计算散列值
isBetween(SourceCodeLocation, SourceCodeLocation) : boolean	判断是否在给定两位置之间
toString() : String	返回仅含行号、列号的字符串

图 2.15 源代码位置类SourceCodeLocation的实现

这种情况下会产生明显的不必要的空行，源代码文件阅读者可以很容易区别。

简单地说，源代码位置主要依赖编译单元名和行号区别，列号则为了进一步避免位置的重复，而且它们的相对大小可很容易地确定名字表实体或控制流图的可执行点的在源代码中的位置。

类SourceCodeLocation的方法getUniqueId()返回一个源代码位置的唯一标识，这个标识由行号、列号和单元名构成，并且在行号与列号之间使用常量字符LINE_COLUMN_SPLITTER（目前的值是':'分隔），而列号与单元名之间使用常量字符FILE_NAME_BEGINNER（目前的值是'@'分隔）。静态方法getLocation()可以将符合上述格式的字符串转换为类SourceCodeLocation的实例。静态方法getFileUnitNameFromId()则从符合上述格式的字符串中提取编译单元名。

类SourceCodeLocation实现接口Comparable<SourceCodeLocation>，以便比较两个源代码位置的顺序，它先按照编译单元名的字典顺序进行比较，然后按照行的大小比较，然后再按照列的大小进行比较。方法equals()和hashCode()也基于编译单元名、行号、列号做相等比较和计算散列值。

类SourceCodeLocation的方法toString()返回一个只含有行号列号的字符串，这可用于编译单元已知的情况下比较简洁地输出一个源代码位置。

类SourceCodeLocatoin提供静态方法getStartLocation()和getEndLocation()计算一个抽象语法树节点的起始点和结束点的源代码位置，这种计算都需要编译单元名及该编译单元的抽象语法树根节点信息，因为只有根节点才能将抽象语法树节点的起始点和结束点（以字符数计的位置）转换为行号、列号信息。

2.4 抽象语法树生成构件的使用

抽象语法树生成构件实际上完成对源代码文件集的管理，可遍历源代码文件集的源代码文件，

获取其文本内容和抽象语法树根节点。

```

21 String startPath = "C:\\MyProgram\\";
22 SourceCodeFileSet fileSet = new SourceCodeFileSet(startPath);
23 SourceCodeFileSetIterator iterator = fileSet.iterator();
24
25 // Use iterator explicitly
26 while (iterator.hasNext()) {
27     SourceCodeFile codeFile = iterator.next();
28     // Get unit name of current source code file by using iterator
29     String unitName = iterator.getCurrentFileUnitName();
30     // Get unit name of current source code file by using SourceCodeFileSet
31     unitName = fileSet.getFileUnitName(codeFile);
32     // Get the content of current source code file
33     String content = codeFile.getFileContent();
34     // Process the content of the current source code file
35     // ...
36     // Get the root AST node of current source code file
37     if (codeFile.hasCreatedAST()) {
38         CompilationUnit root = codeFile.getASTRoot();
39         // Process the AST
40         // ...
41     } else {
42         String errorMessage = codeFile.getParsingErrorMessage();
43         // Process the parsing error message
44         // ...
45     }
46
47     // If have called getFileContent(), then it's better to release it
48     // after processing the content
49     codeFile.releaseFileContent();
50     // If have called hasCreatedAST(), then it's better to release it
51     // after process the AST
52     codeFile.releaseAST();
53 }
54
55 // Use for-each loop
56 for (SourceCodeFile codeFile : fileSet) {
57     String unitName = fileSet.getFileUnitName(codeFile);
58     if (codeFile.hasCreatedAST()) {
59         CompilationUnit root = codeFile.getASTRoot();
60     } else {
61         String errorMessage = codeFile.getParsingErrorMessage();
62     }
63 }
64 fileSet.releaseAllASTs();

```

图 2.16 遍历源代码文件集的核心代码片段

抽象语法树生成构件包含三个类：源代码文件类SourceCodeFile、源代码文件集类SourceCodeFileSet和源代码文件集迭代器类SourceCodeFileSetIterator。这三个类都在程序包sourceCodeAST中。这个程序包的编译单元TestSourceCodeFileSet.java给出了抽象语法树生成构件的基本使用示例。

编译单元TestSourceCodeFileSet.java有四个方法：

1. 方法testGetAllASTs()展示如何显式使用源代码文件集迭代器对象遍历源代码文件集获得所有源代码文件的抽象语法树根节点；
2. 方法testGetAllASTsByForEach()展示如何使用for-each循环遍历源代码文件集获得所有源代码文件的抽象语法树根节点；
3. 方法testGetAllContents()展示如何显式使用源代码文件集迭代器对象遍历源代码文件集获得所有源代码文件的文本内容；
4. 方法testGetAllContents()展示如何显式使用源代码文件集迭代器对象，及如何for-each循环遍历源代码文件集获得所有源代码文件的单元名。

读者可参考这四个方法的源代码以了解源代码文件集的遍历方法。图2.16给出了如何遍历源

代码文件集的核心代码片段。代码片段的第22行（由于是截图，该代码片段的行号不是从1开始计数）使用一个起始路径创建一个源代码文件集对象，第23行得到一个源代码文件集迭代器对象，第26行到第53行的循环显式使用迭代器对象遍历源代码文件集，而第56行到第63行的循环则使用`for-each`循环遍历源代码文件集。

总的来说，构件使用者在使用这三个类时要注意以下几点：

1. 源代码文件的**单元名**是指将文件全名中的源代码文件集起始路径串替换为空串之后得到的串。源代码文件的单元名将用于在名字表中确定名字定义和名字引用在源代码中的位置。由于原代码文件集的所有源代码文件都在同一源起始路径下，因此根据单元名可区别所有的源代码文件；
2. 使用迭代器对象的`getCurrentFileUnitName()`获取源代码文件单元名必须在调用`next()`方法之后；
3. 源代码文件对象`getASTRoot()`方法要获得正确的抽象语法树根节点必须在调用`hasCreatedAST()`方法之后；
4. 为了让JVM的垃圾回收系统回收源代码文件文本内容和源代码文件抽象语法树根节点所占用的内存，在遍历源代码文件集时，处理完当前源代码对象的本文内容/抽象语法树之后最好调用类`SourceCodeFile`的方法`releaseFileContent()` / `releaseAST()`将这个类的对象内部指向文本内容/抽象语法树的指针（对象）置为null。如果觉得有足够的内存，也可在遍历完源代码文件集之后，使用类`SourceCodeFileSet`的方法`releaseAllFileContents()` / `releaseAllASTs()`释放所有源代码文件对象的文本内容/抽象语法树。

第三章 名字表生成构件的需求分析

3.1 需求概述

根据Java语言规范[1], **名字**(name)用于引用声明在程序中的实体, 也就是说, 基于程序语法声明一个实体就定义了一个名字供其他源代码文本使用。**JAnalyzer**的名字表生成构件就是要找出一个软件的源代码文件中所定义的所有名字以及对这些名字的引用, 从而帮助构件使用者分析和理解程序中实体的结构和功能。从构件使用者的角度看, 名字表生成构件至少应提供以下功能:

1. **名字表的生成**, 也就是要找出所给定的软件源代码文件集中所有的**名字定义**(name definition)和**名字引用**(name reference)。
2. **遍历与查找满足某种条件的一个或多个名字定义或名字引用**。构件使用者希望能快速找到满足某种条件的名字定义或名字引用, 有时需要找到满足条件的一个(或说第一个), 有时则需要处理所有满足条件的名字定义或名字引用。
3. **访问名字定义或名字引用的属性**。构件使用者在找到名字定义或名字引用后, 需要访问相关的信息从而完成对软件进行理解和分析的其他功能。

因此, 概括地说, 名字表生成构件需要提供的功能很简单, 就是名字表的生成、查找、遍历与访问。考虑到名字表实际上包含名字定义和名字引用两个方面, 因此更准确的概括是**名字定义的生成、查找、遍历与访问**, 以及**名字引用的生成、查找、遍历与访问**。这里我们将“查找”理解为要求名字表生成构件返回满足指定条件的第一个名字定义或名字引用, 而“遍历”理解为要求名字表生成构件返回满足指定条件的所有名字定义或名字引用。

但为了满足构件使用者的这些需求, 名字表生成构件需要实现的职责实际上却很复杂, 因为程序中有很多不同的实体, 因此有很多名字定义, 同样对这些实体也有各种复杂的使用方式, 因此也有很多名字引用, 而且同样的名字引用在不同的上下文下可能是使用不同的程序实体。

所以, 与前面抽象语法树生成构件的需求分析不同, 这里我们先建立名字生成构件的领域模型, 弄清楚到底有哪些名字定义、名字引用, 它们的主要属性是什么, 然后再弄清楚构件使用者可以应该怎样确定查找与遍历名字定义与引用的条件, 最后再从构件使用者角度给出细化的用例模型。

3.2 领域模型

所谓的领域模型就是要弄清楚名字表生成构件到底有什么? 或者说更简单地说, 名字表到底有什么? 弄清楚这个之后, 我们才能更好地说明构件使用者到底能怎样使用名字表构件。实际上, 抽象语法树生成构件本身最复杂的地方也是在抽象语法树到底是什么这个问题, 但是由于我们是利用Eclipse JDT的API完成抽象语法树的真正生成, 因此2.1节没有探索抽象语法树到底有什么这个

问题。但是，名字表的生成要依赖于程序的语法，即抽象语法树，而不能只依赖于源代码文本，因此这一节我们可能反而要涉及一些抽象语法树的细节以更好地理解名字表的生成与使用。

3.2.1 名字定义

根据我们的理解，**名字定义**是程序在基于语法声明一个程序实体时引入的，那么根据Java语言的语法，程序可声明哪些实体，也即到底有哪些情况是属于名字定义呢？按照Java语言规范[1]，声明的实体包括：

1. **程序包**(package)，在关键字package后面的名字就是一个程序包声明，或者按照我们的说法，在关键字package后面就给出了一个程序包的名字定义。
2. **导入的类型**(imported type)，在关键字import后面的单一类型导入(single-type-import)或按需类型导入(type-import-on-demand)声明；
3. **导入的静态成员**(imported static member)，在关键字import后面的单一静态导入(single-static-import)或按需静态导入(static-import-on-demand)声明；
4. **类**(class)，在关键字class后面的类类型声明中声明；
5. **接口**(interface)，在关键字interface后面的接口类型声明中声明；
6. **类型参数**(type parameter)，作为类属(generic)类、类属接口、类属方法或类属构造方法的一部分声明；
7. 引用类型的成员，这又包括：
 - 7.1 **成员类**(member class); 7.2 **成员接口**(member interface); 7.3 **枚举常量**(enum constant);
 - 7.4 **字段**（或说域）(field)，这又可能是：
 - 7.4.1 类类型或枚举类型的字段；
 - 7.4.2 接口类型或注解(annotation)类型的字段；
 - 7.4.3 任意数组类型的任何字段length。
 - 7.5 **方法**，这又可能是：
 - 7.5.1 类类型或枚举类型的抽象或非抽象方法；
 - 7.5.2 接口类型或注解类型的方法（总是抽象方法）；
8. **参数**(parameter)，这包括：
 - 8.1 类类型或枚举类型的方法、构造方法的形式参数，或兰姆达(lambda)表达式的形式参数；
 - 8.2 接口类型或注解类型的抽象方法的形式参数；
 - 8.3 在try语句的catch子句中声明的**异常参数**(exception parameter)。
9. **局部变量**(local variable)，这可能是：
 - 9.1 声明在语句块中的局部变量；
 - 9.2 声明在for语句中的局部变量。

基于Java程序中的这些声明，我们认为名字表生成构件应该有以下实体：

1. **名字定义**(nameDefinition)，所有其他对应程序中声明的实体都是它的泛化；
2. **程序包定义**(packageDefinition)，对应程序包声明时定义的名字；
3. **类型定义**(TypeDefintion)，对应类型声明时定义的名字。在Java程序中的类型名字定义可用于声明字段类型、方法返回类型、方法参数类型、局部变量类型等等。它又包含三种具体的类型定义：

3.1 详细类型定义(DetailedTypeDefinition)，我们将那些从源代码中能获得类型成员信息的类型定义称为详细类型定义。类、接口、枚举和注解的声明都定义一个详细类型名字定义，但在Java语言中，枚举是特殊的类，而注解是特殊的接口，而类与接口除了一些语法上的不同，从程序理解和分析的角度看并无不同，因此我们在详细类型定义下只派生出两个更具体的实体：

3.1.1 枚举类型定义(EnumTypeDefinition)，对应枚举类型的声明；

3.1.2 注解类型定义(AnnotationTypeDefinition)，对应注解类型的声明。

3.2 导入类型定义(ImportedTypeDefinition)，仅仅是在编译单元中通过导入声明定义的类型名字，无法获得该类型的成员信息（通常是因为是第三方库定义的类，缺乏源代码信息）；

3.3 类型参数定义(TypeParameterDefinition)，类型参数也可作为类型的名字定义变量类型、方法返回类型、方法参数类型、局部变量类型等，但它仅仅是一个名字而已，也没有成员信息；

4. 字段定义(FieldDefinition)，对应字段的声明；

5. 方法定义(MethodDefinition)，对应方法的声明；

6. 变量定义(VariableDefinition)，对应变量的声明，从程序理解和分析的角度，参数的声明与变量的声明没有本质差别，因此我们将参数声明也看做是变量定义。枚举常量实际是枚举这个类的对象实例，因此我们也将它看做是变量声明：

6.1 枚举常量定义(EnumConstantDefinition)，对应枚举常量的声明。

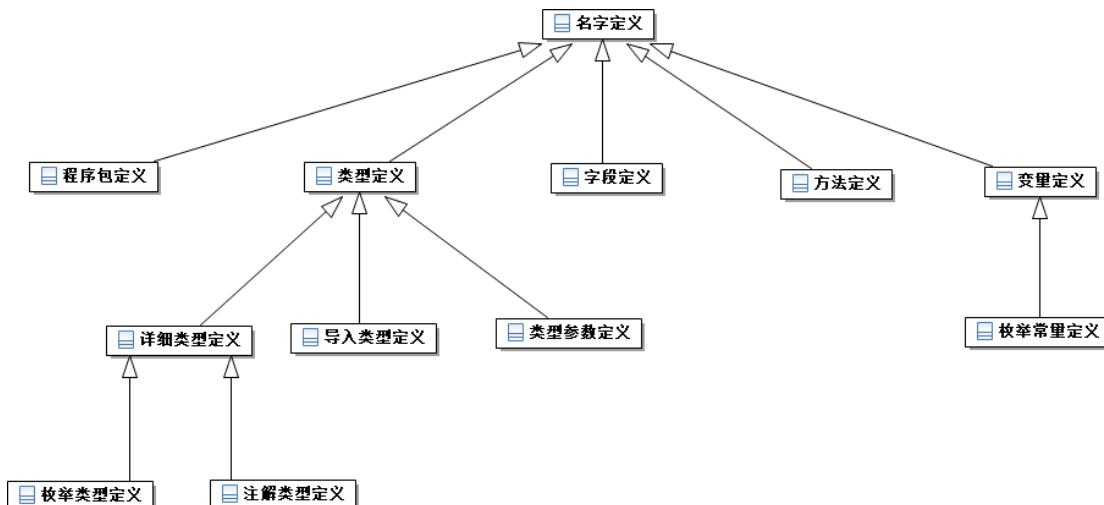


图 3.1 对应名字定义的名字表生成实体

图3.1给出了对应名字定义的名字表生成实体及其之间的泛化关系，再次强调，我们所说的名字定义(name definition)是指程序中的实体在声明时所引入的名字，也就是说一个名字定义对应一个程序实体。注意，在这一节的领域建模中，“实体”有时指JAnalyzer要理解和分析的Java程序中的实体，但有时又指JAnalyzer的领域模型的实体，读者需要根据上下文加以区分，通常这一节说“程序中的实体”、“Java程序中的实体”或“程序实体”时都是指要理解和分析的Java程序中的实体，而无任何修饰的“实体”则通常是指领域模型中的实体。

根据Java规范，Java程序中的实体有简单名(simple name)和限定名(qualified name)之分，一个实体的简单名只有一个标识符，而它的限定名则是由句点(".")分隔的多个标识符。实体限定名的最后一个标识符就是实体的简单名，此前的有句点分隔的多个标识符是另外一个实体的名字。实体的限定名实际上说明了实体之间的从属关系。

3.2.2 名字作用域

根据Java规范，每个程序实体的声明都有一个作用域，也就是说，每个名字定义都定义在某个作用域中。名字定义的**作用域**(scope)是源代码文本的一个区域，则这个区域中，名字定义所对应的实体可使用简单名引用，除非该实体是不可见的。

从概念上，由一个Java源代码文件集（即一个Java软件项目的所有源代码文件）可划分为多个可能嵌套的作用域，每个作用域对应一片源代码文本区域。Java程序的作用域可分为以下几种：

1. **系统作用域**(system scope): 指该Java源代码文件集所有源代码文本构成的区域；
2. **程序包作用域**(package scope): 指同属于一个程序包的Java源代码文件构成的区域；
3. **编译单元作用域**(compilation unit scope): 每个Java源代码文件构成一个编译单元作用域；
4. **类型体作用域**(type body scope): 每一个具有成员的类型（即我们前面所说的详细类型定义）的所有成员声明所组成的代码文本区域形成了该类型的类型体作用域；
5. **方法作用域**(method scope): 方法的参数声明以及实现方法的方法体的文本区域构成了该方法的方法作用域；
6. **局部作用域**(local scope): Java源代码文本的某个特定区域（通常是由左右花括号括起来多个语句构成的一片区域）构成局部作用域。左右花括号括起来的多个语句被称为一个**块语句**(block statement)。

我们可以认为一个程序包定义就确定了一个程序包作用域，一个详细类型定义就确定了一个类型体作用域，而一个方法就确定了一个方法作用域。

这些作用域可能是嵌套的，或者说，从组成关系上以系统作用域为根构成一个树状结构，：

1. 一个Java源代码文件集对应一个系统作用域，一个系统作用域至少包含一个程序包作用域，也可能包含多个程序包作用域；
2. 一个程序包作用域至少包含一个编译单元作用域，也可能包含多个编译单元作用域；
3. 一个编译单元作用域至少包含一个类型体作用域，也可能有多个类型体作用域；
4. 一个类型体作用域可能包含零个或多个局部作用域（对应其静态或非静态初始化块）、零个或多个类型体作用域（对应内部类）、以及零个或多个方法体作用域（对应其方法成员）；
5. 一个方法作用域可能包含零个或一个局部作用域（对应实现方法的方法体块语句）；
6. 一个局部作用域可能包含零个或多个类型体作用域（对应局部类或匿名类），以及零个或多个局部作用域（对应块语句）。

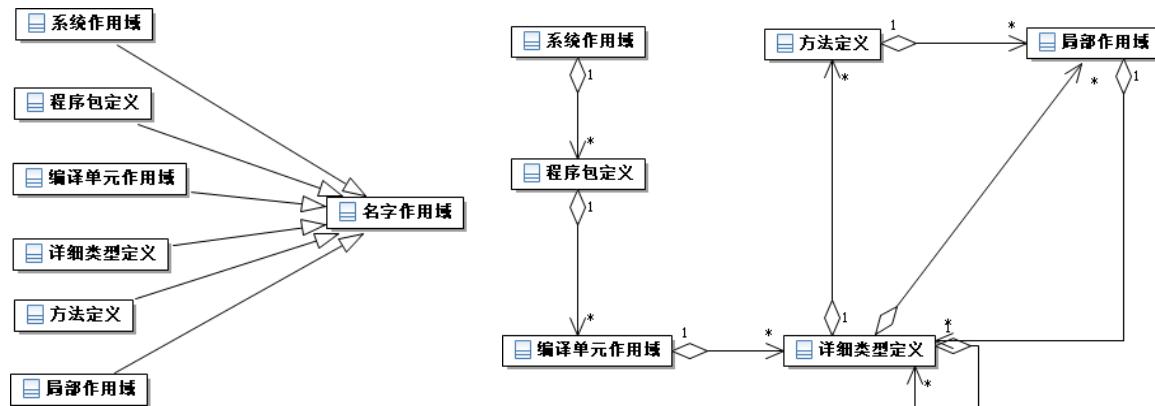


图 3.2 对应名字作用域的名字表生成实体

图3.2给出了名字作用域所涉及的实体，左边给出系统作用域(SystemScope)、程序包定义(PackageDefinition)、编译单元作用域(CompilationUnitScope)、详细类型定义(DetailedTypeDefinition)、方法定义(MethodDefinition)和局部作用域(LocalScope)都是名字作用域，这里我们直接使用程序包定义、详细类型定义和方法定义表示程序包作用域、类型体作用域和方法作用域。图3.2的右边给出了这些作用域之间的上述聚合关系。如果作用域A包含作用域B，我们称作用域B是作用域A的子作用域(sub-scope)。

3.2.3 名字定义与名字作用域

根据Java规范，每个实体都在某个作用域中定义，也即每个名字都在某个作用域中定义，而程序包定义、详细类型定义和方法定义也同时确定了一个作用域。

1. 程序包定义与名字作用域

程序包的声明在编译单元中使用关键字package后跟程序包名字完成，必须是编译单元的第一条语句。多个编译单元可以声明同样的程序包名字，这时这多个编译单元属于相同的程序包，或者说这个程序包名字定义所命名的程序包由这些编译单元构成，也即这个程序包名字定义所确定的作用域就是这些编译单元的所有源代码文本区域。

程序包的命名结构是层次化，可以由多个句点(".")分隔的标识符构成，例如程序包a.b.c，每个标识符实际上都定义了一个程序包，例如命名程序包为a.b.c实际上意味着还有程序包a和程序包a.b，而且程序包a.b是程序包a的子程序包，程序包a.b.c又是程序包a.b的子程序包。这时程序包a.b.c的简单名是c，而a.b.c是它的完全限定名，b.c也是它的（部分）限定名。程序包a.b的简单名是b，a.b是它的完全限定名。

一个程序包的成员包括它的子程序包，以及声明在它的编译单元中的顶层类类型和顶层接口类型，这些成员都必须具有不同的完全限定名。但一个程序包定义所确定的作用域并不包含它的子程序包，而仅仅由它所包含的编译单元构成。

从概念上看，程序包名字本身定义在系统作用域，即使它是另外一个程序包的子程序包。在Java语言中，程序包和子程序包只是用于层次化命名类型，本身并不能被独立引用，而必须与类型名一起被引用，用于更好地确定所引用的类型名到底是引用哪个类型。从作用域角度看，子程序包并不是程序包的子作用域，因此我们认为所有的程序包名字都定义在系统作用域。

Java程序允许一个编译单元没有程序包声明语句，这时这个编译单元属于不具名程序包(unnamed package)。一个Java软件（源代码文件集）至多有一个不具名程序包，而且它没有任何子程序包。这个不具备程序包所确定的作用域由所有没有程序包声明语句的编译单元构成。

2. 详细类型定义与名字作用域

注意，我们这里所说的详细类型定义(detailed type definition)是那些在源代码文件中声明的类类型或接口类型的名字定义，而非导入的没有源代码的外部库中的类类型或接口类型，也不包含用于类属声明中的类属参数。

非匿名类类型或接口类型声明的标志是关键字class或interface，紧跟在这些关键字之后的是类型名字，我们说这就确定了一个详细类型名字定义。详细类型都有使用花括号括起来的语句块给出它的成员声明（即使没有成员也必须使用花括号括起来的空语句进行声明），这个语句块称为

该详细类型的**声明体**(declaration body)

匿名类出现在对象实例创建语句(以关键字new为标志)中,是某个已有类的子类或实现某个已有接口,在已有类或接口的构造方法之后使用花括号括起来的语句块给出匿名类自己的成员,这个语句块是匿名类的声明体。从程序分析角度看,我们不妨认为匿名类有一个自动生成的名字(例如Java编译器根据它是编译单元的第几个匿名类而自动命名),从而匿名类和非匿名类可做相同的处理,都认为它们是详细类型定义。

详细类型名字定义所确定的类型体作用域就是它的声明体,从声明体的左花括号处开始到对应的右花括号处结束。

详细类型名字定义本身所在的作用域则视它的声明所在的位置而定。如果它的声明没有处在任何花括号之中,则称为**顶层类型**(top-level type)。**顶层类型所在的作用域是声明它的编译单元所属的整个程序包作用域**,即属于同一程序包的任意编译单元(包括其他编译单元)都可使用简单名引用该顶层类型,除非该简单名被遮蔽(shadowed)或遮掩(obscured)。

详细类型也会声明在另外一个类型的声明体中,这时该详细类型称为内部类(inner class)或内部接口(inner interface)。如果这时包含该详细类型声明的直接外围花括号块就是另外一个类型的声明体,则**该详细类型定义所在的作用域就是包含它的类型的声明体全部**(不管该详细类型的声明位置在哪里),即在包含它的类型的声明体任何位置都可使用该详细类型定义的简单名进行引用,除非该简单名被遮蔽或遮掩。

如果包含该详细类型声明的直接外围花括号块是不是类型的声明体,而是该声明体中的方法体、初始化块,或方法体、初始化块中的语句块,则该详细类型声明是局部类或匿名类,它所在的作用域则是**包含它的语句块对应的局部作用域**,而且起点是该详细类型声明体及其之后到该局部作用域的右花括号处结束,也即在该局部作用域在该详细类型声明之前的位置也不能引用该详细类型。

注意,Java中的枚举类型是特殊的类类型,只是它使用关键字enum而非class进行声明,而注解类型是特殊的接口类型,只是它使用@interface的形式声明,而非interface进行声明。

3. 导入类型定义的作用域

导入类型定义本身不能确定作用域,它所在的作用域(也就是它的作用域)是导入类型声明所在的编译单元(严格地说是编译单元在该导入类型声明之后的剩余部分)。

4. 类型参数的作用域

类型参数的声明可能出现在类型(类与接口)声明以及方法(包括构造方法)的声明处。在声明类型时,类型名字后使用尖括号("<>")括起来的部分中出现的名字定义了类型参数,也就是给出了一个类型参数名字定义,称为**类或接口的类型参数**。

类或接口的类型参数名字定义本身也不能确定作用域,它所在的作用域是该名字之后的类型声明部分,包括声明类型参数的尖括号的剩余部分,以及超类(使用extends声明)、超接口(使用implements声明)的声明部分,以及该类或接口的声明体。

在声明方法时,出现在方法的返回类型之前(对于构造方法则构造方法名,即类名之前)使用尖括号("<>")括起来的部分中出现的名字定义了**方法的类型参数**。

方法类型参数名字定义本身也不能确定作用域,它所在的作用域是该名字之后的方法声明部分,包括方法的返回类型、参数以及方法方法体。

5. 字段定义的作用域

字段声明只能出现在详细类型定义之中，每个字段声明给出一个字段名字定义。字段名字定义本身不能确定作用域，它所在的作用则是它所在的详细类型的声明体全部，无论该字段声明的位置在哪里。

6. 方法定义与名字作用域

方法声明（无论是构造方法还是非构造方法，是静态方法还是非静态方法）都只能出现详细类型定义之中，每个方法声明给出一个方法名字定义。

方法名字定义所确定的作用域是这个方法的方法作用域，包括方法参数与方法体。

方法名字定义所在的作用域是它所在的详细类型的声明体全部，无论该方法声明的位置在哪里。

7. 变量定义的作用域

变量声明只能出现在局部作用域中，我们说每个变量声明给出一个变量名字定义。变量名字定义当然不能确定作用域，它所在的作用域（也即它的作用域）就是该局部作用域在此变量声明之后的剩余部分。

特别地，枚举常量定义所在的作用域是该枚举常量所在的枚举类型的整个声明体。

基本for语句的初始化块中声明的变量，其作用域是该变量声明之后的点到这个for语句的循环体结束，如果循环体只有一条语句，则到这个语句结束，如果循环体是使用花括号括起来的语句块，则到这个语句块的右花括号处。

增强for语句（或说for-each语句）中声明的变量其作用域也是到这个for-each语句的循环体结束，如果循环体只有一条语句，则到这个语句结束，如果循环体是使用花括号括起来的语句块，则到这个语句块的右花括号处。

异常处理try语句的catch子句中声明的变量的作用域是该catch子句所关联的整个语句块。

表 3.1 名字定义与名字作用域

名字定义	确定的作用域	所在的作用域
程序包定义	程序包作用域	系统作用域
详细类型定义	类型体作用域	程序包作用域、类型体作用域或局部作用域
导入类型定义	无	编译单元作用域
类或接口的类型参数	无	类型体作用域
方法的类型参数	无	方法作用域
字段定义	无	类型体作用域
方法定义	方法作用域	类型体作用域
变量定义	无	局部作用域

表3.1给出了名字定义所确定的名字作用域，以及名字定义所在的名字作用域。这里所谓的名字定义所确定的名字作用域是指程序包、详细类型和方法本身都可确定一个作用域，而名字定义所在的名字作用域也可看做是名字定义的作用域，指在这个文本区域中可使用简单名引用该名字定义，除非该简单名被遮蔽或遮掩。

一个名字定义在它所在的作用域内可能被另外一个具有同样简单名的名字定义所遮蔽(shadowed)，这时被遮蔽的名字定义在它的作用域内部分区域就不能通过简单名进行引用。

常见的遮蔽情况例如方法的参数遮蔽了具有同样名字的字段。实际上，从名字引用的角度看，原则就是简单名的引用优先匹配最内层作用域中的名字定义，在同一层作用域优先匹配离名字引用最近的名字定义。

一个简单名的引用可能根据上下文而被解释为对某个变量定义的引用、对某个类型的引用，或者对某个程序包的引用，因而有时会发生一个名字定义不能在它的作用域内使用简单名引用，这种情况称为该名字定义被遮掩(obscured)。

名字定义的遮蔽和遮掩都会影响一个名字引用的解析(resolve)。名字引用的解析是指确定该名字引用到底是引用哪个名字定义的过程。我们在名字引用一节将概括名字引用的解析规则。

3.2.4 源代码位置

名字定义除了简单名、(完全)限定名以及所在的名字作用域之外，为了更好的理解和分析源代码，我们需要建立名字定义、名字引用与源代码文件之间的关联，因此需要了解名字定义(以及名字引用)在源代码的位置信息。

给定代表一个软件项目的源代码文件集，其中每个源代码文件(也即每个编译单元)的单元名(相对于源代码文件集起始路径的相对文件名)可用于区分不同的源代码文件。而在每个编译单元中，程序标识符所在的行基本上可确定不同的名字定义，因为在同一行源代码不可能声明具有相同简单名的不同实体。但对于名字引用，则同一行源代码可能会有对同一名字的不同引用，因此我们使用“**行号:列数@单元名**”的形式作为源代码位置，也就是说，从概念上看，每个源代码位置由编译单元的单元名+行号+列数确定。

通过源代码位置我们可对照源代码找到每个名字定义和每个名字引用的位置，虽然由于制表符在不同编辑器所代表的空白符个数不同会导致名字出现的列数会有差异，但通过比较列数容易区别同一行同名的名字引用。

3.2.5 名字定义的属性

1. (一般的)名字定义的属性

根据上面的讨论，(一般的)名字定义(NameDefinition)的主要属性包括：

- (1). 名字定义的简单名(simpleName)与全限定名(fullQualifiedName);
- (2). 名字定义(所在的)作用域(scope);
- (3). 名字定义的源代码位置(location)，通常我们将声明这个名字定义的起始位置作为名字定义的源代码位置。

从名字定义的角度看，程序包定义(PackageDefinition)只有简单名、全限定名即可，它所在的作用域总是系统作用域。程序包定义实际上是在各个编译单元，因此它没有源代码位置(或者说其原代码位置为null)。特别地，对于不具名程序包，我们给它赋一个特殊的简单名和全限定名，例如"<UnnamedPackage>"，并且保证在整个系统作用域中只有一个这样的程序包。

2. 类型定义与详细类型定义的属性

对于类型定义(TypeDefinition)，由于从程序理解和分析的角度，Java语言的类类型和接口类型没有本质差别，因此我们将其统一看做是类型定义，而只是简单地在类型定义中设置一个属性是

否接口(isInterface)来区分类类型和接口类型，以避免引入过多的继承层次。

详细类型定义(DetailedTypeDefinition)的简单名就是在声明中关键字class/interface/enum(对于枚举类型)@interface(对于注解类型)后给出的标识符(identifier)。全限定名是它所在的程序包再加上它的简单名。详细类型定义所在的作用域可能是程序包作用域(即顶层类型)、类型体作用域(即内部类)或局部作用域(即局部类、匿名类)。详细类型定义的源代码位置是声明这个详细类型的源代码起始位置。

除继承自名字定义和类型定义的属性外，详细类型定义(DetailedTypeDefinition)的主要属性还有它的成员：

- (1). 详细类型定义的字段定义列表(fieldList)，其元素类型是字段定义(fieldDefinition)；
- (2). 详细类型定义的方法定义列表(methodList)，其元素类型是方法定义(methodDefinition)；
- (3). 详细类型定义的类型定义列表(typeList)，其元素类型是详细类型定义(DetailedTypeDefinition)，该列表记录该详细类型的类体作用域内声明的内部类，这些内部类也是详细类型定义。

详细类型定义除成员外，还可能扩展其他类或接口，或实现其他接口：

(4). 详细类型定义的超类型引用列表(superList)，其元素类型是类型名字引用(TypeReference)。在声明类扩展(extends)的其他类或实现(implements)的接口，以及接口扩展其他接口时，是对其他类、接口的引用，因此详细类型定义的超类型引用列表的元素是类型名字引用。

详细类型定义还可能有静态和非静态初始化块，从程序理解和分析的角度看，静态和非静态初始化块没有本质差别，而且与一个方法的方法体也相同，都是给出了一个局部作用域，在这个作用域中有语句。对于名字表生成而言，我们关系其中的名字定义和名字引用。因此：

- (5). 详细类型定义的初始化块列表(initializerList)，其元素类型是局部作用域(LocalScope)，每个局部作用域存放该初始化列表中可能的名字定义。

详细类型可以声明为零个或多个类型参数从而成为类属类型，因此详细类型定义还有属性：

- (6). 详细类型定义的类型参数列表(typeParameterList)，元素是类型参数定义(TypeParameterDefinition)。

3. 枚举类型定义的属性

Java语言的枚举类型(EnumTypeDefinition)是一种特殊的类类型，在枚举类型中也能声明类能声明的字段、方法和内部类，而且也能实现其他接口(但不能再扩展其他类，因为所有枚举类型都扩展类java.lang.Enum)。因此，我们可将枚举类型扩展详细类型定义，但枚举类型的特殊性在于它可声明枚举常量(实际上枚举常量是该枚举类型的对象实例)：

- (1). 枚举类型的枚举常量列表(constantList)，其元素类型是枚举常量定义(EnumConstantDefinition)；

4. 注解类型定义的属性

Java语言的注解类型(AnnotationTypeDefinition)是一种特殊的接口，其中定义的成员像接口的(抽象)方法成员：

- (1). 注解类型的注解类型成员列表(memberList)，元素是注解类型成员定义(AnnotationTypeMemberDefinition)。

我们需要在前面给出的名字定义的基础上增加一个注解类型成员定义(AnnotationTypeMemberDefinition)，它也扩展名字定义，它的简单名是在生命注解类型成员时给出的标识符，而全限定名是注解类型的全限定名加上它的简单名，所在的作用域就是注解类型的类型体作用域，而它的源代码位置是生命注解类型成员的起始位置。除这些属性外，它具有如下主要属性：

- (1). 注解类型成员的类型(type)，是一个类型名字引用(TypeReference)；

(2). 注解类型成员的缺省值(`defaultValue`)，是一个常量。在后面我们看到，为了推断某些表达式的类型，在解析名字引用时需要记录一些常量，因此需要一个实体来对应常量，我们将这样的实体看做特殊的引用，称为常量引用(`LiteralReference`)。因此注解类型成员的缺省值就是一个常量引用。

5. 导入类型定义的属性

导入类型定义是在编译单元的`import`语句中声明的，由于我们缺乏外部库的源代码，因此按需导入声明（即最后使用“*”表示导入某个程序包的所有可能需要的类）外部库的程序包（的所有可能需要的类型时），我们无法得到具体的类型名字，因此在该编译单元使用简单名引用按需导入的程序包中的类型时，实际上我们无法解析该简单名。但对于单个导入声明所导入的类型，则我们能得到其简单名和全限定名，但是没有任何成员信息，

因此我们所说的导入类型定义(`ImportedTypeDefinition`)是指单个导入声明所导入的类型。导入类型声明中给出的名字是该导入类型定义的全限定名，而简单名则是全限定名去掉程序包名之后的名字。导入类型声明的作用域是所在的编译单元作用域，而源代码位置是该导入声明的起始位置。

由于单个静态导入声明导入的是某个类型的静态成员，而不是导入类型，因此我们需要增加实体导入静态成员定义 (`ImportedStaticMemberDefinition`)，它直接扩展名字定义，单个静态导入声明中给出的名字是它的全限定名，而简单名则是去掉程序包名和类型名之后的名字。导入静态成员定义的作用域也是它所在的编译单元作用域，源代码位置是该导入声明的起始位置。

导入类型定义和导入静态成员定义除继承自名字定义的属性之外，没有其他重要属性。

6. 类型参数定义的属性

类型参数是在声明类型（包括类与接口类型）或方法（包括构造方法）时声明。类或接口的类型参数是在给出类或接口的名字之后使用尖括号括起来，而方法的类型参数是在给出方法返回类型之前使用尖括号括起来。在尖括号中可以声明多个类型参数，声明的类型参数除类型参数名字外，还可以声明它的界。类型参数名字只是一个简单名，而它的界只能是一个交类型(intersection type)。交类型实际上是一个类型引用表达式，由多各类型引用的交构成。

因此，类型参数定义(`TypeParameterDefinition`)的全限定名和简单名都是在定义类型参数时给出的名字（因为类型参数只能用在它所在的作用域，而不能被其他编译单元导入，所以实际上不需要全限定名），其作用域是声明它的类型体作用域，或者方法作用域，源代码位置是声明类型参数的源代码起始位置。其主要属性是：

(1). 类型参数定义的类型界列表(`typeBoundList`)：其元素是类型引用(`TypeReference`)，实际上，类型引用就是一个类型表达式，它的结果就是某个类型定义（也即它将绑定到某个类型定义），可能是详细类型定义、导入类型定义甚至也可以是类型参数定义。

7. 字段定义的属性

字段定义(`FieldDefinition`)在某个详细类型定义中声明，它的简单名是声明字段时给出的标识符，全限定名是它所在的详细类型定义全限定名加该简单名。所在的作用域则是该详细类型的类型体作用域，而源代码位置是该字段声明的源代码起始位置。除此之外，字段定义的主要属性是：

(1). 字段定义的类型(`type`)，是一个类型引用(`TypeReference`)。

8. 方法定义的属性

方法定义(MethodDefinition)在某个详细类型定义中声明，它的简单名是声明方法时给出的标识符(方法名)，全限定名是它所在的详细类型定义全限定名加该简单名。所在的作用域则是该详细类型的类型体作用域，而源代码位置是该方法声明的源代码起始位置。除此之外，方法定义的主要属性是：

- (1). 方法定义的返回类型(returnType)，是一个类型引用(TypeReference)；
- (2). 方法定义的参数列表(parameterList)，其元素是变量定义(VariableDefinition)。实际上，方法形式参数的声明与变量声明没有本质差别，因此无需设置一个参数定义的实体，直接使用变量定义即可；
- (3). 方法定义的抛出异常列表(throwTypeList)，其元素是类型引用(TypeReference)，因为每个异常实际上是一个类类型，在方法中的声明则是对这些类类型的引用；
- (4). 方法定义的类型参数列表(typeParameterList)，元素是类型参数定义(TypeParameterDefinition)；
- (5). 方法定义的方法体(body)，是一个局部作用域(LocalScope)，因为方法体是一个块语句，给出了一个局部作用域，在生成名字表时关系其中的名字定义和名字引用；
- (6). 是否构造方法(isConstructor)。我们在名字表中不设置分别对应构造方法和非构造方法的实体，而只是用这样一个布尔类型的标记来区分构造方法和非构造方法。构造方法的名字与类型的名字相同，返回类型为null(与返回类型void不同)。

8. 变量定义的属性

变量定义(VariableDefinition)在某个局部作用域中声明，它的简单名与全限定名相同，都是声明该变量时给出的标识符。所在的作用域则是声明它的局部作用域，源代码位置是声明它的源代码的起始位置。除此之外，它还有属性：

- (1). 变量定义的类型(type)，是一个类型引用。

注意，即使在程序中声明为final的变量，对我们来说也是变量定义。枚举常量从概念上来说也是一个变量，它是枚举类型的对象实例，它的简单名是声明时给出的标识符，全限定名是枚举类型的全限定名加上该简单名，所在的作用域为该枚举类型的类型体作用域，源代码位置是声明它的源代码起始位置。除此之外，枚举常量声明中可使用圆括号给出一些参数，这些参数是调用该枚举类型的构造函数时所需的实际参数。因此枚举常量定义(EnumConstantDefinition)除继承自名字定义的属性之外，还有：

- (1). 枚举常量定义的实际参数列表(argumentList)：其元素是名字引用(NameReference)。实际参数是一个在运行时可计算值的表达式，因而实际上是对一些变量、常量的引用。后面我们将看到，Java程序中的表达式对应JAnalyzer平台中的名字引用组(NameReferenceGroup)，即将一些相关的名字引用组织在一起，构成一个特定的上下文(即表达式)，从而便于推断每个名字引用到底是引用哪个程序实体。

基于对各种名字定义的主要属性的分析，可细化图3.1所给出的对应名字定义的名字表生成实体如图3.3

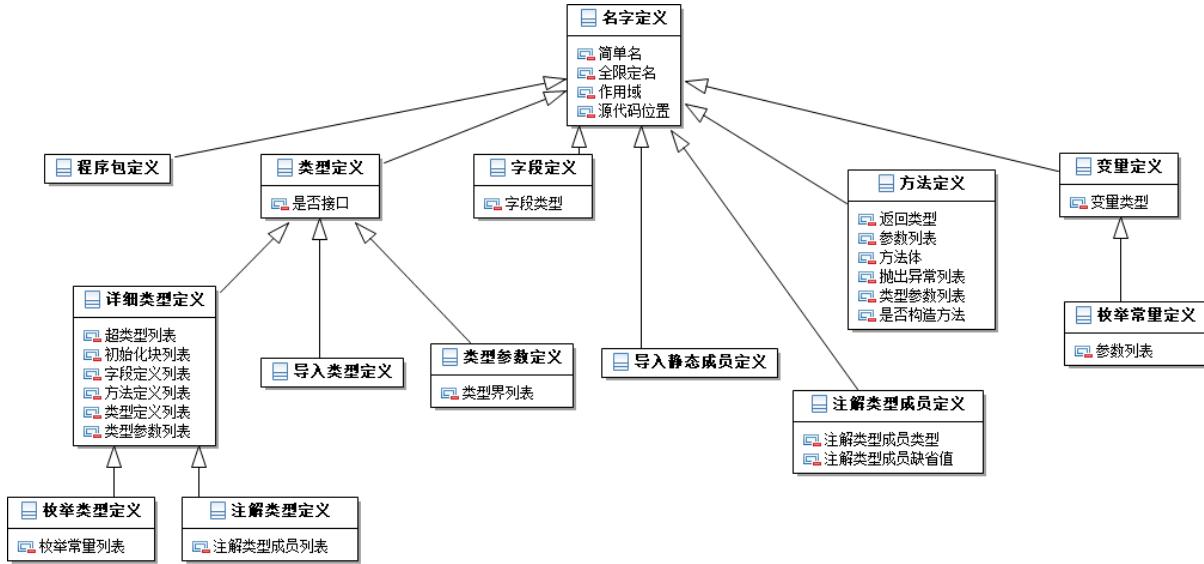


图 3.3 对应名字定义的名字表生成实体

3.2.6 名字作用域的属性

在分析各种名字定义的主要属性之后，这里进一步考虑名字作用域的属性。对于（一般的）名字作用域（NameScope）而言，我们认为有以下主要属性：

- (1). 作用域的名字（name）：每个作用域都有一个名字，可用于区分不同的作用域，以及用于输出作用域的基本信息；
- (2). 作用域的起始位置（startLocation）和终止位置（endLocation）：每个作用域在概念上都对应一个源代码文本区域，这个源代码文本区域有起始位置和终止位置，都是源代码位置（SourceCodeLocation）的实例；
- (3). 作用域的包围作用域（enclosingScope），即直接包围该作用域的作用域，也称为该作用域的父作用域（parent scope），图3.2给出的聚合关系给出了包围某个作用域的可能作用域类型；
- (4). 作用域的子作用域列表（subscopeList），则当前作用域包围哪些作用域，列表元素是名字作用域（NameScope），同样图3.2给出的聚合关系给出了一个作用域可能包围的作用域类型；

图3.2给出了六种具体的名字作用域，对这些具体作用域来说：

1. 系统作用域（SystemScope）可有一个特殊的名字，例如“<SystemScope>”，它的起始位置和终止位置都可以是null，但意味着它包含所有源代码文件；它的父作用域是null，意味着它没有父作用域；
2. 程序包作用域（PackageDefinition），注意实际上一个程序包定义就对应一个程序包作用域。它的作用域名字就是程序包定义的全限定名，它的起始位置和终止位置也可以是null，但意味着它包含所有属于它的编译单元。程序包作用域的父作用域就是系统作用域，子作用域列表则是它的编译单元所对应编译单元作用域列表；
3. 编译单元作用域（CompilationUnitScope），一个编译单元对应一个编译单元作用域。它的名字是该编译单元的单元名，它的起始位置是该编译单元的程序包声明语句之后的位置，终止位置是编译单元的文件末尾。编译单元作用域的父作用域就是它所属的程序包定义，子作用域列表则是顶层类型所对应的类型体作用域；

4. 类型体作用域(DetailedTypeDefinition), 一个类型体作用域对应一个详细类型定义, 因此一个详细类型定义就是一个类型体作用域。它的作用域名字是该详细类型的全限定名, 起始位置是声明该类型的起始位置, 终止位置是声明该类型的终止位置(即右花括的位置)。它的父作用域可能是编译单元作用域(对于顶层类), 也可能是另外一个类型体作用域, 甚至一个局部作用域。它的子作用域包括它所有的方法定义(MethodDefinition)、初始化块(是LocalScope的实例)和内部类(是DetailedTypeDefinition的实例);

5. 方法作用域(MethodDefinition), 一个方法作用域对应一个方法定义, 因此一个方法定义就是一个方法作用域。它的作用域名字是该方法的全限定名, 起始位置是声明该方法的起始位置, 终止位置是该方法的方法体的右花括号处。它的父作用域是一个类型体作用域(DetailedTypeDefinition的实例)。它的子作用域包括它的方法体对应的局部作用域(LocalScope)。

6. 局部作用域(LocalScope), 一个语句块对应一个局部作用域。我们可基于语句块的起始源代码位置设定局部作用域的名字, 例如"<Block>@列数:行号@单元名"的形式。局部作用域的起始位置就是该语句块的起始位置(左花括号处), 终止位置就是该语句块的终止位置(右花括号处)。局部作用域的父作用域可能是类型体作用域、方法作用域或者另外一个局部作用域。局部作用域的子作用域可能是类型体作用域或者另外一个局部作用域。

前面表3.1从名字定义的角度给出了名字定义所确定的作用域和所在的作用域, 这里我们再从作用域的角度讨论每个作用域都可以有哪些名字定义:

1. 系统作用域可以有的名字定义是程序包定义。
2. 程序包作用域(程序包定义)中不直接拥有名字定义, 顶层详细类型的作用域可理解为程序包作用域, 但这些详细类型也分散在各个编译单元中。
3. 编译单元作用域可以有的名字定义是详细类型定义和导入类型定义。顶层详细类型的作用域可理解为程序包, 但分散在各个编译单元中。导入类型定义的作用域是所在的编译单元。
4. 详细类型定义作为作用域可以有的名字定义是字段定义、方法定义、详细类型定义和类型参数定义。
5. 方法定义作为作用域可以有的名字定义是变量定义(方法参数)和类型参数定义。
6. 局部作用域可以有的名字是变量定义和详细类型定义。

表 3.2 名字作用域的主要属性及可能拥有的名字定义

名字作用域	可能的父作用域	可能的子作用域	可能拥有的名字定义
系统作用域	null	程序包定义	程序包定义
程序包作用域	系统作用域	编译单元作用域	
编译单元作用域	程序包作用域	详细类型定义	详细类型定义、导入类型定义
详细类型定义	编译单元作用域 详细类型定义 局部作用域	详细类型定义 方法定义 局部作用域	字段定义、方法定义 详细类型定义 类型参数定义
方法定义	详细类型定义	局部作用域	变量定义、类型参数定义
局部作用域	详细类型定义 方法定义 局部作用域	局部作用域 详细类型定义	变量定义 详细类型定义

表3.2给出六种作用域的可能父作用域(包围作用域)、可能的子作用域(子作用域列表的可能元素)以及可能拥有的名字定义。注意，每个名字作用域至多有一个父作用域，但是可能有多个子作用域。

图3.4给出了名字作用域的主要属性，名字作用域之间的泛化关系，以及名字作用域与名字定义和源代码位置之间的关联关系，其中名字作用域和名字定义域之间的关联关系表明，一个名字作用域可能有多个名字定义，但一个名字定义只有一个名字作用域，名字作用域与源代码位置之间的关联关系表明一个名字作用域有两个源代码位置(起始位置和终止位置)，名字定义和源代码位置之间的关联关系表明一个名字定义有一个源代码位置。

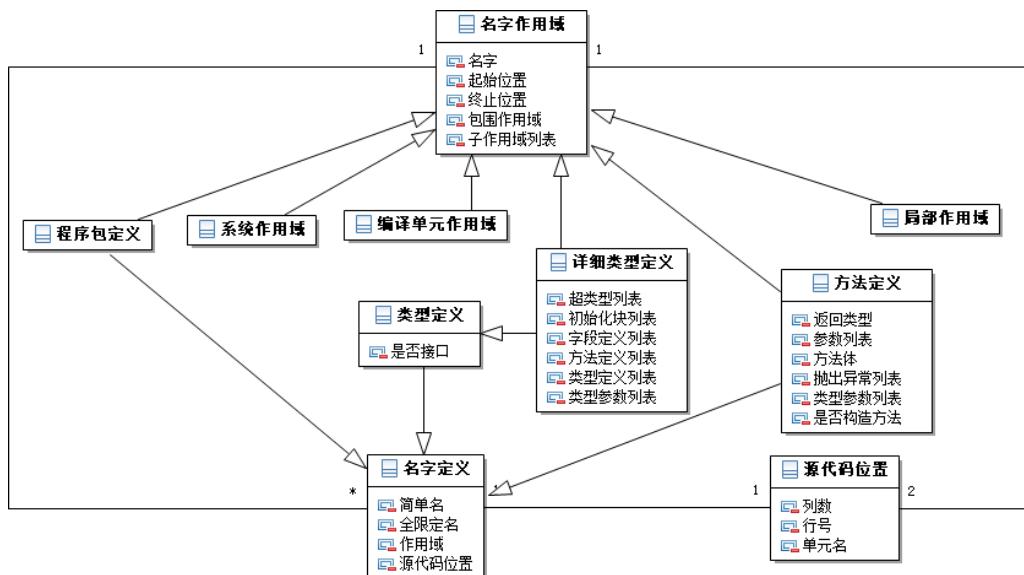


图 3.4 名字作用域的属性及其关联关系

3.2.7 程序中的名字定义、作用域和名字引用

名字引用(name reference)是指为使用某个已经声明的程序实体而给出的名字。实际上，在程序中出现的用户自定义标识符，除用于声明程序实体的名字定义外，其他都是名字引用。每个名字引用都引用某个程序实体，或者说都引用某个名字定义。将名字引用与它所引用的名字定义关联起来的过程称为**名字绑定**(name binding)，也称为**名字解析**(name resolving)。

图3.5给出了一个例子程序，我们通过该例子程序说明名字定义、名字引用以及名字作用域。

例子程序第1行声明了一个程序包sourceCodeAsTestCase，第3行声明了类SuperClass，第4行声明了字段fieldOne，第5行声明了方法methodOne，以及参数argOne和argTwo。第5行出现的SuperClass则是名字引用，引用在第3行声明的类SuperClass。第6,7行出现的用户自定义标识符(即除关键字if,else,return以及运算符之外的单词(token))都是名字引用。

表3.3给出了例子程序的所有名字定义。注意，虽然第4行和第11行的名字定义的简单名都是fieldOne，但它们的全限定名不同，坐在的作用域也不同。第5行和第18行的argOne虽然在表中的信息(除位置外)看起来一样，但是它们所在的作用域不同，虽然表中给出它们的作用域都是methodOne，但这两个方法的全限定名不同。

```
1 package sourceCodeAsTestCase;
2
3 class SuperClass {
4     protected int fieldOne = 0;
5     public int methodOne(int argOne, SuperClass argTwo) {
6         if (argOne < argTwo.fieldOne) return argOne + fieldOne;
7         else return argTwo.fieldOne + fieldOne;
8     }
9 }
10 class AnotherClass {
11     public int[] fieldOne;
12 }
13 class CNEexample extends SuperClass {
14     private int fieldTwo = 0;
15     private CNEexample fieldThree = null;
16     private AnotherClass[] fieldFour;
17
18     public int methodOne(int argOne, SuperClass argTwo) {
19         return argOne + argTwo.fieldOne + fieldOne;
20     }
21     public int methodTwo(int argOne, int argTwo) {
22         int result = 0;
23         fieldThree = new CNEexample();
24         fieldThree.fieldTwo = argOne;
25         fieldFour = new AnotherClass[argOne];
26         fieldTwo = 0;
27         while (fieldTwo < fieldThree.fieldTwo) {
28             fieldFour[fieldTwo] = new AnotherClass();
29             fieldFour[fieldTwo].fieldOne = new int[argTwo];
30             for (int index = 0; index < argTwo; index++) {
31                 fieldFour[fieldTwo].fieldOne[index] = fieldOne;
32                 result = result + fieldOne;
33             }
34             fieldTwo++;
35         }
36         return result;
37     }
38     public static void main(String[] args) {
39         SuperClass objOne = new CNEexample();
40         int result = objOne.methodOne(2, objOne);
41         CNEexample objTwo = (CNEexample) objOne;
42         result = result + objTwo.methodTwo(4, 8);
43
44         System.out.println("Result is " + result);
45     }
46 }
```

图 3.5 说明名字定义和名字引用的例子程序

表 3.3 图3.5的例子程序中的名字定义

位置	全限定名	简单名	种类	作用域
	sourceCodeAsTestCase	sourceCodeAsTestCase	程序包定义	<System>
3:0	sourceCodeAsTestCase.SuperClass	SuperClass	详细类型定义	CNExample.java
10:0	sourceCodeAsTestCase.AnotherClass	AnotherClass	详细类型定义	CNExample.java
13:0	sourceCodeAsTestCase.CNExample	CNExample	详细类型定义	CNExample.java
4:15	sourceCodeAsTestCase.SuperClass.fieldOne	fieldOne	字段定义	SuperClass
5:1	sourceCodeAsTestCase.SuperClass.methodOne	methodOne	方法定义	SuperClass
3:0	sourceCodeAsTestCase.SuperClass	SuperClass	方法定义	SuperClass
5:22	argOne	argOne	方法参数定义	methodOne
5:34	argTwo	argTwo	方法参数定义	methodOne
11:14	sourceCodeAsTestCase.AnotherClass.fieldOne	fieldOne	字段定义	AnotherClass
14:13	sourceCodeAsTestCase.CNExample.fieldTwo	fieldTwo	字段定义	CNExample
15:19	sourceCodeAsTestCase.CNExample.fieldThree	fieldThree	字段定义	CNExample
16:24	sourceCodeAsTestCase.CNExample.fieldFour	fieldFour	字段定义	CNExample
18:1	sourceCodeAsTestCase.CNExample.methodOne	methodOne	方法定义	CNExample
21:1	sourceCodeAsTestCase.CNExample.methodTwo	methodTwo	方法定义	CNExample
38:1	sourceCodeAsTestCase.CNExample.main	main	方法定义	CNExample
18:22	argOne	argOne	方法参数定义	methodOne
18:34	argTwo	argTwo	方法参数定义	methodOne
21:22	argOne	argOne	方法参数定义	methodTwo
21:34	argTwo	argTwo	方法参数定义	methodTwo
22:6	result	result	变量定义	<Block>@21:46
30:12	index	index	变量定义	<Block>@30:3
38:25	args	args	方法参数定义	main
39:13	objOne	objOne	变量定义	<Block>@38:40
40:6	result	result	变量定义	<Block>@38:40
41:12	objTwo	objTwo	变量定义	<Block>@38:40

表 3.4 图3.5的例子程序中的名字作用域

名字	起始位置	终止位置	父作用域	种类
<System>				系统作用域
sourceCodeAsTestCase			<System>	程序包定义
CNExample.java			sourceCodeAsTestCase	编译单元作用域
SuperClass	3:0	9:1	CNExample.java	详细类型定义
methodOne	5:1	8:2	SuperClass	方法定义
<Block>@5:53	5:53	8:2	methodOne	局部作用域
SuperClass	3:0	3:0	SuperClass	方法定义
AnotherClass	10:0	12:1	CNExample.java	详细类型定义
CNExample	13:0	46:1	CNExample.java	详细类型定义
methodOne	18:1	20:2	CNExample	方法定义
<Block>@18:53	18:53	20:2	methodOne	局部作用域
methodTwo	21:1	37:2	CNExample	方法定义
<Block>@21:46	21:46	37:2	methodTwo	局部作用域
<Block>@30:3	30:3	33:4	<Block>@21:46	局部作用域
main	38:1	45:2	CNExample	方法定义
<Block>@38:40	38:40	45:2	main	局部作用域
CNExample	13:0	13:0	CNExample	方法定义

表3.4给出了例子程序中的所有名字作用域。注意其中局部作用域使用<Block>加上局部作用域的起始位置命名。由于程序中的块语句很多，我们无需对每一个块语句都建立一个局部作用域，只需当其中有名字定义时才建立局部作用域，所以表中的局部作用域除每个方法的方法体都有一个局部作用域外，只有第30行的for循环语句建立了一个局部作用域，因为循环语句的初始化语句声明了变量index，该变量定义的作用域是这个局部作用域，而第27行开始的while循环的循环体则无需建立局部作用域。

表 3.5 图3.5的例子程序中的名字引用

名字	位置	类别	绑定结果	名字	位置	类别	绑定结果
argOne	6:6	变量引用	argOne@5:22	argTwo	6:15	变量引用	argTwo@5:34
fieldOne	6:22	字段引用	fieldOne@4:15	argOne	6:39	变量引用	argOne@5:22
fieldOne	6:48	变量引用	fieldOne@4:15	argTwo	7:14	变量引用	argTwo@5:34
fieldOne	7:21	字段引用	fieldOne@4:15	fieldOne	7:32	变量引用	fieldOne@4:15
argOne	19:9	变量引用	argOne@18:22	argTwo	19:18	变量引用	argTwo@18:34
fieldOne	19:25	字段引用	fieldOne@4:15	fieldOne	19:36	变量引用	fieldOne@4:15
fieldThree	23:2	变量引用	fieldThree@15:19	CNExample	23:19	方法引用	
fieldThree	24:2	变量引用	fieldThree@15:19	fieldTwo	24:13	字段引用	fieldTwo@14:13
argOne	24:24	变量引用	argOne@21:22	fieldFour	25:2	变量引用	fieldFour@16:24
AnotherClass	25:18	类型引用	AnotherClass@10:0	argOne	25:31	变量引用	argOne@21:22
fieldTwo	26:2	变量引用	fieldTwo@14:13	fieldTwo	27:9	变量引用	fieldTwo@14:13
fieldThree	27:20	变量引用	fieldThree@15:19	fieldTwo	27:31	字段引用	fieldTwo@14:13
fieldFour	28:3	变量引用	fieldFour@16:24	fieldTwo	28:13	变量引用	fieldTwo@14:13
AnotherClass	28:29	方法引用		fieldFour	29:3	变量引用	fieldFour@16:24
fieldTwo	29:13	变量引用	fieldTwo@14:13	fieldOne	29:23	字段引用	fieldOne@11:14
argTwo	29:42	变量引用	argTwo@21:34	index	30:23	变量引用	index@30:12
argTwo	30:31	变量引用	argTwo@21:34	index	30:39	变量引用	index@30:12
fieldFour	31:4	变量引用	fieldFour@16:24	fieldTwo	31:14	变量引用	fieldTwo@14:13
fieldOne	31:24	字段引用	fieldOne@11:14	index	31:33	变量引用	index@30:12
fieldOne	31:42	变量引用	fieldOne@4:15	result	32:4	变量引用	result@22:6
result	32:13	变量引用	result@22:6	fieldOne	32:22	变量引用	fieldOne@4:15
fieldTwo	34:3	变量引用	fieldTwo@14:13	result	36:9	变量引用	result@22:6
SuperClass	39:2	类型引用	SuperClass@3:0	CNExample	39:26	方法引用	
int	40:2	类型引用	int	objOne	40:15	变量引用	objOne@39:13
methodOne	40:15	方法引用	methodOne@5:1	objOne	40:35	变量引用	objOne@39:13
CNExample	41:2	类型引用	CNExample@13:0	CNExample	41:22	类型引用	CNExample@13:0
objOne	41:32	变量引用	objOne@39:13	result	42:2	变量引用	result@40:6
result	42:11	变量引用	result@40:6	objTwo	42:20	变量引用	objTwo@41:12
methodTwo	42:20	方法引用	methodTwo@21:1	System	44:2	类型引用	
out	44:9	变量引用		println	44:2	方法引用	
result	44:36	变量引用	result@40:6				

表3.5给出了例子程序中的名字引用，其中的绑定结果给出的是该名字引用是引用哪个名字定义。可以看到，程序中有很多名字引用，而且但从名字（标识符）来看也有很多同名的名字引用。名字定义可以根据全名来区分，但名字引用只能根据上下文来确定。特别地，其中第31行出现了两个fieldOne的名字引用，但第一个（表中在31:24位置的）fieldOne引用的绑定结果是在第11行定义的fieldOne，它是类AnotherClass 声明的字段，而第二个（表中在31:42 位置的）fieldOne引用的绑定结果是在第4行定义的fieldOne，它是类SuperClass声明的字段。这两个fieldOne实际上引用

的是不同的字段，这只能通过上下文来区分。

表3.5中也出现几个名字引用没有绑定结果，这是因为：(1)对于某些构造方法的调用由于是调用缺省的，即源代码中没有显式定义的构造函数，因此不能绑定到任何名字定义；(2)一些外部库的对象，如System，及其字段与方法，如println()和out等，都由于没有源代码而不能绑定。

这里需要再次声明的是，表3.3、表3.4和表3.5中给出的名字定义、作用域和名字引用的列位置与使用编辑器打开源代码文件查看时列位置可能不同，因为不同编辑器对于制表符相当于几个空白字符的定义不全相同。我们这里给出的列位置都是将制表符看做一个字符而计算出的列位置。不过这并不重要，列位置主要是用于区别同一行出现的多个同名的名字引用，从而列数大小我们可看出是第几个名字引用。

3.2.8 名字引用及其属性

名字引用就是对名字定义所对应的程序实体的使用，通常程序中除声明实体时给出的名字是名字定义之外，其他出现用户自定义标识符的地方都是名字引用，因此名字引用的数量比名字定义的数量要多得多。

基于所引用的名字定义，名字引用可分为以下几类：

1. 程序包引用(PackageReference)，它引用一个程序包定义。程序包引用很少单独出现，通常是作为类型的全限定名的一部分，或者是在按需导入类型时，星号“*”之前的内容都是程序包引用。
2. 类型引用(TypeReference)，它引用一个类型定义，可能是详细类型定义、导入类型定义甚至类型参数。随着Java语言的发展，引用一个类型的方式越来越多样，我们将计算结果是类型的表达式都称为类型引用。根据Java规范，Java语言可能有的类型表达式有：

2.1 简单类型引用(SimpleTypeReference)：语法形式是“标识符”，也即就使用一个标识符引用一个类型，也即这个标识符是某个类型的简单名（因为在该类型的作用域类，或者使用导入声明已经导入，从而可使用简单名进行引用）；

2.2 带限定名类型引用(QualifiedTypeReference)：语法形式是“{类型引用}.标识符”，即一个标识符前还有零个或多个类型引用。对一个类的内部类的引用可能需要这种形式。只有在确定前面带限定名的类型引用是引用哪个类型的基础上才能确定整个表达式引用的是哪个类型。简单类型引用可看做带限定名类型引用的特例。

2.3 带名字类型引用(NamedTypeReference)：语法形式是“{带限定的名字}.标识符”，即一个标识符前还有零个或多个带限定的名字(QualifiedName)。带限定的名字的语法形式是“{带限定的名字}.标识符”，即一个标识符前又可能有零个或多个带限定的名字。

带限定的名字、带限定名类型引用和带名字类型引用的区别在于：带限定的名字不一定是引用类型（即不一定是类型表达式，而可能是引用字段）；带名字类型引用和带限定名类型引用都肯定是引用类型（即都是类型表达式），但带名字类型引用的标识符前面是否是一个类型还不能确定（有可能是程序包名字，也有可能是类型名），但带限定名类型引用的标识符前可以确定是一个类型（而不是程序包）。

2.4 参数化类型引用(ParameterizedTypeReference)：语法形式是“类型引用<类型引用，类型引用>”，即一个类型引用后使用尖括号括起来一个或多个类型引用作为参数。在语法正确的Java程序中，第一个类型引用所引用的类型应该具有类型参数定义，并且类型参数的个数与这里尖括号里给出的类型引用个数相同。

2.5 通配符类型引用(WildcardTypeReference): 语法形式是“`? [(extends | super) 类型引用]`”，即是问号，且其后可使用关键字`extends`或`super`加跟一个类型引用。通配符类型引用主要用于声明方法参数。

2.6 交类型引用(IntersectionTypeReference): 语法形式是“`类型引用 & 类型引用 & { & 类型引用}`”，即是使用运算符`&`连接的两个或多个类型引用。交类型引用主要用于声明类型参数的界。

2.7 并类型引用(UnionTypeReference): 语法形式是“`类型引用 | 类型引用 | { | 类型引用}`”，即是使用运算符`|`连接的两个或多个类型引用。

通配符类型引用、交类型引用和并类型引用主要用于类型推导，是Java语言近年逐步引入的泛型(generic)程序设计所需要的特性，这些类型表达式的引入使得程序设计更为灵活，但也使得类型推导和类型匹配，特别是子类型关系更为复杂，目前在实际的应用程序还不是用得很普遍，主要是在容器类及其一些通用的算法实现中使用。

3. 方法引用(MethodReference): 就是对方法的调用，最基本的语法形式是“`标识符(实际参数列表)`”。方法引用根据标识符给出的方法名和参数列表绑定合适的方法定义。

4. 值引用(ValueReference): 就是对类的字段访问或对局部变量的访问。最基本的语法形式就是“`标识符`”，即给出一个标识符，它有可能是访问字段，也有是访问局部变量。我们将它们归为同样的引用，因为本质上它都是引用一片内存中的值。

5. 文字引用(LiteralReference): 就是程序中的文字常量。之所以将文字常量也看做引用，是因为在绑定一个名字引用时可能需要文字常量所给出的类型信息。

程序包引用、类型引用、方法引用、值引用和文字引用都是最基本的引用，即它们最终都是使用程序中的某个实体，因此都可绑定到名字定义（当然有些是使用程序源代码之外的第三方构件中的实体）。但为了确定一个名字引用到底绑定到哪个名字定义，还需要上下文来确定，这个上下文在Java语言中实际上就是表达式。特别是变量引用和方法引用（方法调用），它们通常是在表达式中出现，例如字段引用的基本形式是“`表达式.标识符`”引用字段，而方法调用的基本形式是“`表达式.标识符(参数列表)`”，要确定其中的标识符到底是引用哪个类的字段或哪个类的那个方法，我们需要首先确定表达式的类型。

因此，需要将相关的名字引用放在一起构成适当的上下文，在这个上下文下才能正确绑定其中的名字引用。我们使用名字引用组(NameReferenceGroup)标识相关的名字引用构成的上下文。后面我们将看到，对应Java语言中的每个表达式，我们都有相应的名字引用组，对应表达式的语法结构，名字引用组也呈现一种树状结构，即名字引用组中可能还有名字引用组，而最终的叶子节点则是上面所给出的程序包引用、类型引用、方法引用、值引用或文字引用这些最基本的名字引用。

图3.6给出了名字引用相关的实体及其泛化关系，其中名字引用组还将派生出多个实体，我们在后面进一步描述。

图3.6也给出了名字引用实体的主要属性。名字引用(NameReference)的共同属性包括：

1. 名字引用的名字(name): 即源代码中用于引用名字定义的标识符，或标识符序列（字符串）；
2. 名字引用的源代码位置(location): 即名字引用在源代码中的位置；
3. 名字引用的所在作用域(scope): 包含该名字引用的最内层的名字作用域；
4. 名字引用的绑定结果(definition): 对名字引用进行绑定之后的结果，也即该名字引用所引用的名字定义。

程序包引用(PackageReference)没有新的属性。对于类型引用，因为每个类型都可用于创建数

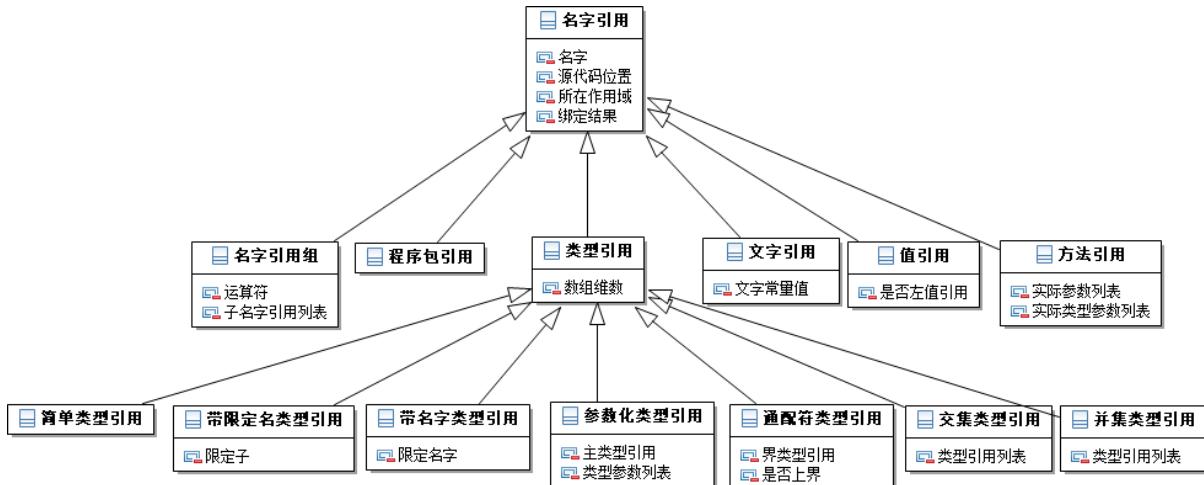


图 3.6 涉及名字引用的名字表生成实体

组，因此在类型引用中设置数组维数属性可区分在引用这个类型时是用于创建数组还是创建单个变量：

1. 类型引用的数组维数(dimension): 它是一个整数，说明使用该类型声明的数组是几维。当取0的时候表明没有使用该类型声明数组（而是单个变量）；

带限定名类型引用(QualifiedTypeReference)的主要属性是：

1. 带限定名类型引用的限定子(qualifier): 它是一个类型引用(TypeReference)，因为JDT Eclipse 在生成抽象语法树时能确认这个限定名（限定子）是类型引用；

带名字类型引用(NamedTypeReference)的主要属性类似：

1. 带名字类型引用的限定名字(qualifier): 但它是一个一般的名字引用(NameReference)，因为JDT Eclipse在生成抽象语法树时尚不能确定这个限定名字到底是类型还是程序包；

注意，带限定名类型引用和带名字类型引用的名字都是使用句点分隔的一系列标识符中的最后那个标识符。参数化类型引用(ParameterizedTypeReference)的主要属性就是：

1. 参数化类型引用的主类型引用(primaryType): 即放在尖括号之前的那个类型引用，它也可能是一个复杂的类型表达式；

2. 参数化类型引用的类型参数列表(argumentList): 一个参数化类型引用可能给出一个或多个实际类型参数，以匹配在类型参数定义中给出的形式类型参数。实际类型参数也都是类型引用，因此类型参数列表的元素是类型引用(TypeReference)；

通配符类型引用(WildcardTypeReference)的主要属性是：

1. 通配符类型引用的界类型引用(bound): 通配符类型引用中可使用extends或super指定界，界也是引用一个类型，因此也是类型引用；

2. 通配符类型引用的是否上界(isUpperBound): 对于通配符类型引用的界需要区分是上界（即使用extends声明）还是下界（即使用super声明）。

交集类型引用(IntersectionTypeReference)和并集类型引用(UnionTypeReference)的主要属性都是：

1. 类型引用列表(typeList): 给出了参与交集运算或并集运算的两个或多个类型，其元素也都是类型引用。

注意，参数化类型引用、通配符类型引用、交集类型引用和并集类型引用从名字引用和类型引用继承的属性可能并不适用，但因为在Java程序语法上，这些东西都可看做是类型，因此在JAnalyzer平台里，从概念上来说，这些也都是类型引用。

方法引用(MethodReference)对应程序中的方法调用，在面向对象程序中存在方法重载和重定义，因此为正确决定一个方法调用到底是调用那个方法（也即方法引用的绑定）还需要方法的实际参数列表：

1. 方法引用的实际参数列表(argumentList): 从语法上说，方法的实际参数是一个表达式，也是对名字的引用（或名字引用组），因此实际参数列表的元素是名字引用(NameReference)。
2. 方法引用的实际类型参数列表(typeArgumentList): 从语法上说，方法的参数类型参数是一个类型表达式，是对类型的引用，因此实际类型参数列表的元素是类型引用(TypeReference)。

注意，方法引用的名字就是在方法调用中给出的在左原括号之前的那个标识符，也即所要绑定方法的简单名。如果在方法调用中并不是只用简单名调用，例如System.out.println()，则这将对应一个名字引用组，其中关于Sysmte, out和println的引用都是该名字引用组的最基本的名字引用，而println则是一个方法引用，该方法调用的实际参数属于该方法引用。

值引用(ValueReference)意味着它引用的是一片可以存放值的内存区域，有时是左值，这时意味着要往这片内存区域存储数据，有时是右值引用，这时意味着要读取这片内存区域中的数据。因此值引用的主要属性是：

1. 是否左值引用(isLeftValue): 我们使用一个布尔类型的标记来区分左值和右值引用。

同样值引用中的名字也是对应变量定义或字段定义的简单名，如果使用obj.field这种形式访问字段，则它也对应一个名字引用组，对obj和field的引用都在该名字引用组中，只有名字为field的引用是值引用。

文字引用(LiteralReference)的主要属性当然是：

1. 文字引用的文字常量值(value)。

3.2.9 名字引用组

名字引用组(NameReferenceGroup)就是将上下文相关的一组名字引用汇集在一起，以便能根据上下文正确地绑定其中的名字引用。名字引用组对应Java语言中的表达式，将一个表达式中出现的所有名字引用按照表达式的语法结构组织在一起。因此名字引用组的主要属性是：

1. 名字引用组的运算符(operator): 是一个字符串，用于保存表达式，特别是算术表达式中的运算符；
2. 名字引用组的子名字引用列表(subreferenceList): 给出了该名字引用组所包含的子名字引用列表，其元素也是名字引用，特别地也可以是名字引用组，从而构成对应表达式语法结构的名字引用树。

例如，对于Java程序中的一个表达式 $x = y + z$ ，对应名字表中的一个名字引用组，其运算符是=，它包含两个子名字引用，其他一个是对于变量x的值引用(ValueReference)，而另外一个是对应表达式 $y+z$ 的名字引用组。对于表达式 $y+z$ 的名字引用组的运算符是+，它又包含两个子名字引用，分别是对于变量y和变量z的值引用。

对于名字引用组而言，它的名字就是在源代码中表达式这个串，它的绑定结果是该表达式的结果类型所对应的类型定义，例如对应表达式 $x=y+z$ 的名字引用组，它的名字就是"x=y+z"这个字符

串，它的绑定结果就是变量x的类型所对应的类型定义。

对应Java语言表达式，有以下具体名字引用组实体。这些实体的英文命名我们都以NRG开头：

1. 数组访问(NRGArrayAccess): 数组访问表达式的语法形式是“表达式[表达式]”，因此对应的名字引用组有两个子名字引用（组），分别对应语法中的两个表达式。该名字引用组的绑定结果是第一个表达式的结果类型（所对应的类型定义），也即从语义上来说是数组访问表达式的结果类型就是数组元素的类型（所对应的名字定义）；

2. 数组创建(NRGArrayCreation): 数组创建表达式有几种语法形式：

```
new 基本类型[表达式] { [表达式]}{[]}
new 类型名[<类型{, 类型}>] [表达式] { [表达式]}{[ ]}
new 基本类型[] {[]} 数组初始化表达式
new 类型名[<类型{, 类型}>] [] {[]} 数组初始化表达式
```

概括地说，数组创建表达式就是一个类型表达式后跟一个或多个数组维度表达式（用中括号括起来），若不给出维度表达式，则需要给出数组初始化表达式。因此对应的名字组有多个子名字引用，但都以一个类型引用（可能是参数化类型引用）开头，后跟零个或多个数组维度表达式对应的名字引用（组）（多数情况下就是整数类型的文字常量引用），以及零个或一个数组初始化表达式。整个数组创建表达式对应的名字引用组绑定到第一个类型引用的绑定结果，也即从语义上来说，数组创建表达式的结果类型就是这个类型引用所对应的类型（也是数组元素的类型）。

3. 数组初始化(NRGArrayInitializer): 数组初始化表达式的语法形式是：

```
{ [表达式{, 表达式} [, ]] }
```

也即是用花括号括起来的多个以逗号分隔的表达式。因此对应的名字引用组有多个子名字引用（组），每个子名字引用（组）对应一个表达式。该名字引用组的绑定结果是第一个表达式的结果类型（所对应的类型定义）。如果在语法上不存在第一个表达式，则无需建立对应的名字引用组。

4. 赋值(NRGAssignment): 赋值表达式的语法形式是“表达式赋值运算符表达式”，因此对应的名字引用组的运算符是赋值运算符，有两个子名字引用（组）。绑定的结果是第一个名字引用的结果类型（所对应的名字定义）。

这里要强调的是**名字引用的结果类型不等于名字引用的绑定结果**，特别是对于最基本的名字引用（例如值引用）。例如，假定程序中有变量声明MyClass obj，然后有赋值obj = aObj，则这个赋值对应的名字引用的绑定结果是obj的结果类型，也即MyClass（对应的类型定义），但obj作为变量引用（值引用）本身的绑定结果是声明语句MyClass obj对应的变量定义。

实际上，引入名字引用组的概念是为了在解析名字引用时有更好地上下文信息，这个上下文信息主要就是这个名字引用在怎样种类的表达式中出现，这个表达式的类型是什么。因此**名字引用组的绑定结果一定是一个类型定义，但基本名字引用的绑定结果则是对应某个声明语句的名字定义**。

5. 类型转换(NRGCast): 类型转换表达式的语法形式是“(类型) 表达式”，因此对应的名字引用有两个子名字引用（组），前一个必定是一个类型引用。整个名字引用组的绑定结果就是该类型引用的绑定结果。

6. 类实例创建(NRGClassInstanceCreation): 类实例创建表达式的语法形式是：

```
[表达式.] new [<类型, {类型}>] 类型([表达式{, 表达式}]) [匿名类声明]
```

也即是以零个或一个表达式加句点后跟关键字new，然后是可能的实际类型参数，然后是要创建实例的类型名字（实际上是调用该类型的构造方法），然后是圆括号括起来的零个或多个实际参数列表，最后是可能的匿名类声明。匿名类声明对应类型定义，它不属于名字引用，因此类实例创建表达式对应的名字引用组的子名字引用包括对应new前面表达式的名字引用，对应实际类型参数后面的类型的类型引用及其（构造）方法引用。

注意，构造方法的实际类型参数以及实际参数都将放在对应方法引用中，而不是直接隶属于类实例创建表达式对应的名字引用组。我们不仅为类实例创建中圆括号前面的类型设置类型引用也为其实例设置（构造）方法引用，这使得这个表达式更好理解：new前面的表达式，以及这个类型引用来确定所调用的构造方法到底在哪个类中，而方法引用及其实际类型参数和实际参数用于确定到底是调用这个类的那个构造方法。

对应类实例创建表达式的名字引用组的绑定结果是其中的类型引用的绑定结果（即该类型引用所绑定的类型定义）。

7. 条件表达式(NRGConditional): 条件表达式的语法形式是“表达式? 表达式: 表达式”，因此对应的名字引用组有三个子名字引用（组）。对应条件表达式的名字引用组的绑定结果是第二个名字引用的结果类型，除非第二个名字引用的结果类型是第三个名字引用的结果类型的子类型，这时整个名字引用组的绑定结果是第三个名字引用的结果类型。

8. 字段访问(NRGFieldAccess): 字段访问表达式的语法形式是“表达式.标识符”，因此对应的名字引用组有两个子名字引用（组）。第一个名字引用对应句点前的表达式，而第二个名字引用是一个值引用，因为在这个名字引用组中，这个标识符能确定是某个类型的字段。整个名字引用的绑定结果是该值引用的结果类型。

在Java程序中对详细类型定义的字段访问可能有多种形式：

(1) 一个单一的标识符可能就是对字段的访问，不过Eclipse JDT在生成抽象语法树时，对于单一的标识符并不能识别是字段访问（因为在生成抽象语法数时并不会去试图确定一个名字的含义），这时Eclipse JDT中只能生成一个类型为SimpleName的AST节点与之对应，而在JAnalyzer平台中，我们称之为值引用；

(2) 最普通的字段访问形式是"foo.bar"这种形式，对于这种形式，Eclipse JDT在生成抽象语法树时，可能会识别是字段访问（即生成为类型是FieldAccess的AST节点），也可能只生成一个类型为QualifiedName的AST节点与之对应，在JAnalyzer平台中，对这两种不同的节点，我们有两种不同的名字引用组，前者是NRGFieldAccess，后者是NRGQualifiedName。

(3) 形如"foo.this"的表达式，在Eclispe JDT生成抽象语法树时归结为类型是ThisExpression的AST节点，我们也相应地有名字引用组NRGThisExpression；

(4) 形如>this.foo"这种表达式，在Eclipse JDT生成抽象语法树时是得到一个ThisExpression的AST节点，然后有一个SimpleNameAST节点。在JAnalyzer平台中则会对应一个LiteralReference和一个ValueRefernece构成的名字引用组；

(5) 如果使用关键字super访问父类型的字段，在Eclipse JDT生成抽象语法树时是得到一个SuperFieldAccess 的AST节点，对应地在JAnalyzer平台中有名字引用组NRGSuperFieldAccess。

9. 中缀表达式(NRGInfixExpression): 语法形式是“表达式中缀运算符表达式{中缀运算符表达式}”，即使用某个中缀运算符连起来的多个表达式。因此对应的名字引用组的子名字引用对应这些表达式，名字引用组的运算符存放该中缀运算符。注意只有在所有中缀运算符都相同时，这多个

表达式才成为该名字引用组的（同一层）子名字引用，否则将使用不同的名字引用组来分层组织这些表达式所对应的子名字引用。整个名字引用组的绑定结果是整个中缀表达式的结果类型，在绑定时需要对整个中缀表达式的类型进行推断。

10. 实例判断表达式(NRGInstanceof): 语法形式是“表达式`instanceof` 类型”，因此对应的名字引用组有两个子名字引用，前者是一个一般的名字引用，后者是一个类型引用。整个名字引用组的绑定结果是`boolean`类型所对应的类型定义。

为处理Java语言的基本类型，我们需要将这些基本类型看做也是（自动）导入的类型，因此需要为这些基本类型生成相应的类型定义，这些类型定义被认为是导入类型定义（因为它们没有成员信息）。

11. 方法调用(NRGMethodInvocation): 方法调用表达式的语法形式是：

`[表达式.] [<类型{, 类型}>] 标识符([表达式{, 表达式}])`

即句点前面的表达式用于确定所调用的方法属于哪个类型，用尖括号括起来的类型列表给出的是方法的实际类型参数列表，圆括号前给出的标识符是所调用方法的简单名，圆括号中间给出的是方法的实际参数列表。与类实例创建(NRGClassInstanceCreation)类似，方法调用表达式对应的名字引用组至多有两个子名字引用，其中一个对应句点之前的表达式（如果存在的话则是第一个），另外一个（如果句点之前有表达式则是第二个，否则是第一个）是一个方法引用(MethodReference)，方法调用中的实际类型参数和实际参数都是该方法引用的一部分。

方法调用表达式对应的名字引用组的绑定结果是其中方法引用绑定结果（是一个方法定义）的返回类型（所对应的类型定义），因为从语法上来说，方法调用表达式的结果类型应该是所调用方法的返回类型。

12. 后缀表达式(NRGPostfixExpression): 语法形式是“表达式后缀运算符”，即一个表达式后带有后缀运算符。因此对应的名字引用组只有一个子名字引用对应这个表达式，名字引用组的运算符存放该后缀运算符。整个名字引用组的绑定结果是这个表达式的结果类型。

13. 前缀表达式(NRGPrefixExpression): 语法形式是“前缀运算符表达式”，即一个表达式前面有前缀运算符。因此对应的名字引用组只有一个子名字引用对应这个表达式，名字引用组的运算符存放该前缀运算符。整个名字引用组的绑定结果是这个表达式的结果类型。

14. 超类型字段访问(NRGSuperFieldAccess): 语法形式是“[类名.]`super`.标识符”，即一个可能的类名后跟关键字`super`，再跟一个标识符。因此对应的名字引用组有两个子名字引用，一个是类型引用，对应类名，一个是值引用，对应其中的标识符，它应该是类型引用所绑定的类型定义的超类型中的字段。整个名字引用组绑定的结果是这个字段的类型所对应的类型定义。

15. 超类型方法调用(NRGSuperMethodInvocation): 语法形式是

`[类名.]super. [<类型{, 类型}>] 标识符([表达式{, 表达式}])`

即类名加关键字`super`然后跟实际类型参数、方法简单名和实际参数。因此对应的名字引用组有两个子名字引用，一个是类型引用，对应类名，一个是方法引用，对应其中的标识符，实际类型参数和实际参数都是方法引用的一部分。整个名字引用组的绑定结果是其中方法引用绑定结果（是一个方法定义）的返回类型（所对应的类型定义）。

16. This表达式(NRGThisExpression): 语法形式是“[类名.]`this`”，即一个类名跟关键字`this`。对应的名字引用组有两个子名字引用，一个是类型引用，对应类名，一个是文字常量

引用(LiteralReference)，对应关键字this。整个名字引用组的绑定结果类型引用所对应的类型定义，如果没有类名，则（这时只有this这个关键字）根据表达式所在的类型作用域，绑定到这个类型作用域所对应的类型定义。

17. 类型常量(NRGTypeLiteral): 语法形式是“(类型— void).class”，即是一个类型名后跟关键字class，或是关键字void后跟关键字class。对于后一种情况，实际上没有名字引用，对于前者所对应的名字引用组只有一个子名字引用，而且是一个类型引用，对应其中的类型。整个名字引用组绑定的结果是java.lang.Class这个预定义类所对应的类型定义。同样，我们可将Java预定义类（即在java.lang这个程序包中的类）看做是导入类型定义，因为它们是自动导入的。

18. 变量声明(NRGVariableDeclaration): 语法形式是：

```
{修饰符} 类型变量声明片段{, 变量声明片段}  
变量声明片段 ::= 标识符{维数} {= 初始表达式}
```

变量声明通常是一个语句，但在for循环中它出现的地方是一个表达式该出现的地方，所以称为变量声明表达式。因此这个表达式主要是将声明标识符所对应的变量，也即这个标识符不是名字引用，而是名字定义。在这个表达式中出现名字引用的地方是类型，以及在变量声明片段中可能出现的维数表达式，和初始化表达式。因此对应的名字引用组的第一个子名字引用是类型引用，后面的子名字引用则对应每个变量声明片段中的维数和初始化表达式。整个名字引用组的绑定结果是第一个类型引用所对应的类型定义。

19. 带限定的名字(QualifiedName): 语法形式是“(带限定的名字|标识符).标识符”，也就是用句点分隔的两个或多个标识符。此名字引用组有两个子名字引用，第一个给出前面带限定名字对应的名字引用，第二个给出最后的标识符所对应的名字引用。

真如前面所说，使用句点分隔的两个或多个标识符，有可能根据Eclipse JDT在生成抽象语法树时的判断结果，分别对应带限定名类型引用、带名字类型引用、字段访问、或带限定的名字。前两者是能确定整个名字（表达式）是类型引用，字段访问是能确定最后的标识符是某个类型的字段，而带限定的名字则就是什么都不能确定的情况下选择，因此后面将谈到，要绑定带限定的名字则可能要将各种可能的情况都试一下，才能确定它到底绑定到那个名字定义。

表3.6简单地总结了各个名字引用组所对应的Java语言表达式的语法结果，以及这整个名字引用组的绑定结果。名字引用组的绑定结果通常是它所对应的表达式的结果类型（所对应的类型定义）。在Java语言的最新规范中还引入了方法表达式和Lambda表达式，方法表达式和Lambda表达式使得方法可以在不调用（不被执行）的情况下作为参数传递，从而在某种程度上实现函数式程序设计风范。由于目前的Java应用程序还比较少用这两个表达式，所以目前暂不考虑这两种表达式。

3.2.10 名字引用的解析

名字引用的解析就是指确定名字引用到底是访问那个程序实体的过程，也称为**名字绑定**(name binding)。名字绑定的结果就是将名字引用与它所想访问的程序实体对应的名字定义相关联。

最基本的名字引用有程序包引用、类型引用、方法引用和值引用，它们的绑定结果分别是程序包定义、类型定义、方法定义，以及字段定义或变量定义。但对名字引用的解析很多时候需要基于上下文，也即名字引用所在的表达式来确定绑定结果。针对不同的Java语言表达式，我们使用对应

表 3.6 名字引用组对应的表达式和绑定结果

名字引用组	对应表达式的语法结构	绑定结果
数组访问 NRGArrayAccess	表达式[下标表达式], 前一个表达式还可是数组表达式, 从而是多维数组元素访问	数组元素的类型(所对应的类型定义)
数组创建 NRGArrayCreation	<code>new</code> 类型表达式[维数表达式], 类型表达式可是参数化类型, 维数表达式可有多个	数组元素的类型(所对应的类型定义)
数组初始化 NRGArrayInitializer	用逗号分隔的多个表达式	第一个表达式的结果类型(所对应的类型定义)
赋值 NRGAssignment	表达式赋值运算符表达式	运算符之前表达式的结果类型(所对应的类型定义)
类型转换 NRGCast	(类型)表达式	类型引用的绑定结果
类实例创建 NRGClassInstanceCreation	[表达式.] 实际类型参数 类型(实际参数)	由[表达式.]类型所能绑定的类型(所对应的类型定义)
条件表达式 NRGConditional	表达式?表达式:表达式	第二个或第三个表达式的结果类型(所对应的类型定义)
字段访问 NRGFieldAccess	表达式.字段名	声明字段的类型(所对应的类型定义)
中缀表达式 NRGInfixExpression	表达式中缀运算符表达式	所有表达式的类型提升到一致的类型(所对应的类型定义)
实例判断表达式 NRGInstanceOf	表达式 <code>instanceof</code> 类型	<code>boolean</code> 类型所对应的导入类型定义
方法调用 NRGMethodInvocation	[表达式].实际类型参数 方法名(实际参数)	方法名所能绑定的方法的返回结果类型(所对应的类型定义)
后缀表达式 NRGPostfixExpression	表达式后缀运算符	其中表达式的结果类型(所对应的类型定义)
前缀表达式 NRGPrefixExpression	前缀运算符表达式	其中表达式的结果类型(所对应的类型定义)
超类型字段访问 NRGSuperFieldAccess	[类名]. <code>super</code> .字段名	字段名能绑定到的字段的类型(所对应的类型定义)
超类型方法调用 NRGSuperMethodInvocation	[类名]. <code>super</code> .实际类型参数方法名(实际参数)	方法名所能绑定到的方法的返回类型(所对应的类型定义)
This表达式 NRGThisExpression	[类名]. <code>this</code>	类名所能绑定到的类型, 或所在的类型体作用域对应的类型定义
类型常量 NRGTypeLiteral	(类型 <code>void</code>). <code>class</code>	<code>java.lang.Class</code> 所对应的导入类型定义
变量声明 NRGVariableDeclaration	修饰符类型名后跟多个变量名及其初始化表达式	类型名所能绑定到的类型(所对应的类型定义)
带限定的名字 NRGQualifiedName	用句点分隔的多个标识符	根据情况绑定到类型定义或字段定义

的名字引用组将出现在该表达式中的所有名字引用组织在一起，使得在对这些名字引用进行解析时有上下文信息可供使用。表3.6给出了不同的名字引用组的可能绑定结果。

名字绑定过程概括地说就是在合适的作用域里匹配对应种类的名字定义，所以名字绑定的要点有两点：(1) 如何匹配对应种类的名字定义；(2) 如何确定合适的作用域。

1. 名字引用与名字定义的匹配

真正将名字引用中的名字与名字定义中的简单名进行匹配发生在基本的名字引用，名字引用组的作用就是通过一些名字引用的绑定结果来确定其他一些名字引用应该在哪个作用域中进行匹配，因此真正的名字引用名字与名字定义名字的匹配只发生在名字引用组的叶子节点，名字引用组的内部节点的名字只是存储对应的表达式串，不会用于名字匹配。

程序包引用、类型引用和值引用的名字匹配过程比较简单，就是在合适的作用域中查找具有与程序包引用、类型引用或值引用中保存的名字相同的简单名的程序包定义、类型定义、变量定义或字段定义。对于值引用，如果是在局部作用域匹配则首先与变量定义的名字进行匹配，如果是在详细类型作用域则与字段定义的名字进行匹配。由于在同一作用域，同一种类的名字定义（除方法定义外）必然具有不同的简单名，因此如果匹配成功则绑定成功，否则绑定不成功。

对于方法引用，在Java语言中，包含方法调用的表达式包括对象实例构造表达式、方法调用表达式和超类型方法调用表达式，它们的语法结构都可归结为“[表达式].<实际类型参数> 标识符(实际参数)”，其中标识符给出要调用方法的简单名（对于对象实例构造就是类型的简单名）。为简单起见，这里只考虑不含实际类型参数的方法调用，它所绑定的方法定义也只能是非类属方法（即不声明形式类型参数的方法）。对于一个方法调用表达式，要确定它到底调用哪个方法（即我们所说的将方法引用绑定到方法定义）遵循如下步骤：

1. 首先根据（可能存在的）表达式确定在哪个类型（类或接口）的成员方法中寻找合适的方法定义。如果前面有表达式，则在表达式的结果类型（所对应的类型定义）中查找。如果前面没有表达式，则在包含该方法调用表达式的最内层的类型体作用域所对应的类型中查找。

2. 然后要确定在这个类型中有哪些以方法调用表达式中的标识符为简单名且可以访问和可以应用的方法，只有这些方法才可能被调用。所谓一个方法可以访问是指方法调用表达式出现的地方有访问该方法的访问控制权限，所谓一个方法可以应用是指满足下述两个条件的方法：

(1) 该方法的形式参数数目与上述调用表达式中的实际参数数目相同；

(2) 上述调用表达式的每一个实际参数的类型与该方法相应位置上的形式参数类型相同，或者能隐式类型转换为形式参数类型。

3. 最后在这些可以访问且可以应用的方法中确定最精确的方法(most specific method)。当然如果可以访问且可以应用的方法只有一个，那么这个方法就是最精确的方法。假设有两个方法，其形式参数列表分别为 $\langle S_1, S_2, \dots, S_n \rangle$ （下面称为方法一）和 $\langle T_1, T_2, \dots, T_n \rangle$ （下面称为方法二），其中 S_i 和 T_i 都是形式参数类型名（注意形式参数名对方法的匹配毫无影响），说方法一比方法二更精确，如果它们满足下面两个条件：

(1) 定义方法一的类型是定义方法二的类的子类型（注意继承成员也在考虑的范围内，因此这两个方法可能定义在不同的类中，但它们肯定一个是另外一个的子类型）；

(2) 对于任意的 i ，类型 S_i 与类型 T_i 相同，或者 S_i 的数据都能隐式类型转换为 T_i 。

如果能确定一个最精确的方法，则该方法调用表达式所调用的方法就是这个方法，如果最精确

的方法不只一个，则就发生了方法调用的二义性。不过，对于JAnalyzer平台而言，我们假定所分析的源代码都是语法正确（能通过编译器编译）的源代码，因此不考虑发生二义性的情况。

在Java语言中，类型S的数据如果能转换为类型T，我们不妨理解为类型S是类型T的子类型(subtype)，记为S <: T，这时也称类型T是类型S的超类型。子类型关系是直接子类型关系的自反、传递闭包。说类型S是类型T的直接子类型(direct subtype)，记为S < T。直接子类型关系如下定义：

(1) 对于基本数据类型，取值范围小的类型是取值范围大的类型的子类型，具体来说：

byte < short < int < long < float < double char < int < long < float < double

注意，char和byte及short间没有子类型关系，boolean类型与这些数值类型都没有子类型关系。

(2) 对于类或接口，如果类型T出现在类型S的声明中extends或implements中，则S < T。如果类型S没有extends或implements声明，则S < Object。

为简单起见，这里没有考虑类属类型（即带有类型参数的类或接口）。如果考虑类型的类属参数以及方法的类属参数，则上述子类型规则，以及绑定方法调用表达式的步骤都会更为复杂。

2. 名字引用绑定时的作用域确定

对于基本的名字引用，从它所在的作用域开始进行名字定义匹配，这样可以简单地处理名字遮掩或遮蔽，因为名字遮掩或遮蔽总是内层作用域的名字定义遮掩或遮蔽外侧作用域的名字定义。如果它所在的作用域不能匹配成功，则沿着作用域之间的包围关系在父作用域进行匹配，一直到系统作用域。

对于名字引用组，则其中的一些名字引用可能用于确定另外一些名字引用应该进行匹配的作用域，这样的名字引用组包括：

类实例创建： [表达式.]<(实际类型参数) 类型(实际参数)

方法调用： [表达式.]<(实际类型参数) 方法名(实际参数)

字段访问： [表达式.]字段名

超类型方法调用： [类名.]super.< 实际类型参数方法名(实际参数)

超类型字段访问： [类名.]super.字段名

This表达式： [类名].this

带限定的名字： 用句点分隔的多个标识符

实际上，对于这些名字引用组而言，除带限定的名字之外，其他名字引用组中的表达式或类名就是用于确定后面方法名、字段名的匹配作用域，也即后面的方法名（包括类实例创建的类型）、字段名需要在前面表达式的结果类型或类名所对应的类型定义中进行匹配。考虑到访问的可能是继承成员，那么更准确地说应该是表达式的结果类型或类型所对应的类型或其超类型中进行匹配。

对于带限定的名字，它可能最后实际是引用一个程序包、一个类型或者一个字段，可能需要从前面的标识符开始逐步尝试它是程序包引用、类型引用还是字段引用，并且基于前面标识符的绑定结果确定下一个标识符应该在哪个程序包定义或类型定义中进行匹配。

其他的名字引用组各个子名字引用组都在相同的作用域内进行匹配，实际上，这些名字引用组都绑定到对应表达式的结果类型，主要是用于在方法调用绑定时用于类型参数或方法参数的匹配。

3.2.11 名字表生成的领域模型总结

名字表中的实体比较多，这里对名字表生成相关的实体进行总结，实际上这些实体回答了“名字表到底有什么”这个问题。

1. 名字表中最重要的实体包括名字定义(`NameDefinition`)、名字作用域(`NameScope`)和名字引用(`NameReference`)。

2. 名字定义(`NameDefinition`)是指Java程序在声明一个程序包、类型、字段、方法、变量等的时候引入的名字，以对程序中的实体或说元素(即程序包、类型、字段、方法、变量)进行命名。因此在JAnalyzer的名字表生成构件的领域模型中主要有程序包定义(`PackageDefinition`)、类型定义(`TypeDefinition`)、方法定义(`MethodDefinition`)、字段定义(`FieldDefinition`)和变量定义(`VariableDefinition`)等具体的名字定义实体。

3. 对于类型定义，我们将那些在所分析的源代码中有详细的成员声明的类型定义称为详细类型定义(`DetailedTypeDefinition`)，而将那些从第三方构件库或由Java编译器自动导入的类型(包括基本类型)称为导入类型定义(`ImportedTypeDefinition`)。

4. 名字定义的主要属性包括它的简单名(即不含任何句点的名字)(`simpleName`)、完全限定名(`fullQualifiedName`)，所在的作用域(`scope`和它的源代码位置(`location`)。

5. 详细类型定义的主要属性还包括它的超类型列表(`superTypeList`)、初始化块列表(`initializerList`)、字段定义列表(`fieldList`)、方法定义列表(`methodList`)、类型定义列表(`typeList`)以及类型参数列表(`typeParameterList`)。

6. 方法定义的主要属性还包括它的返回类型(`returnType`)、参数列表(`parameterList`)、方法体(`body`)、抛出异常列表(`throwTypeList`)、类型参数列表(`typeParamterList`)和是否构造方法(`isConstructor`)。所以这里将构造方法和一般方法等同看待，构造方法的名字是类型名字，返回类型也是这个类型。

7. 图3.3给出了名字定义所涉及的主要实体及其之间的关系(主要是继承关系)。

8. 名字作用域(`NameScope`)是程序源代码文本的一片区域，在这个区域中声明的实体可使用它的简单名访问，除非它被同名的实体遮掩或遮蔽。

9. Java程序的作用域可理解为有系统作用域(`SystemScope`)、程序包作用域(`PackageDefinition`)、编译单元作用域(`CompilationUnitScope`)、类型体作用域(`DetailedTypeDefinition`)、方法作用域(`MethodDefinition`)和局部作用域(`LocalScope`)。

10. 程序包定义、详细类型定义和方法定义不仅是一个名字定义，而且也确定了一个作用域，因此我们通常用与名字定义相同的实体命名其确定的作用域。

11. 名字作用域的主要属性包括作用域的名字(`name`)，作用域的起始位置(`startLocation`)和终止位置(`endLocation`)，直接包围该作用域的父作用域(`enclosureScope`)，以及该作用域的子作用域列表(`subscopeList`)。通过子作用域列表，名字作用域根据嵌套关系构成了一颗多叉树，根是系统作用域。

12. 图3.4给出了名字作用域所涉及的主要实体及其之间的关系(包括与名字定义之间的关联关系)。

13. 名字引用(`NameReference`)是对程序实体(程序包、类型、字段、方法、变量)的使用，因此最基本的名字引用有程序包引用(`PackageReference`)、类型引用(`TypeReference`)、方法引用(`MethodReference`)和值引用(`ValueReference`)。程序中对变量和字段的使用都是访问一段内存的

值，或者设置一段内存的值，而且在缺乏上下文的情况下经常难以区分，因此将对变量和字段的引用都称为值引用。

14. 确定名字引用到底是对那个程序实体的使用这个过程称为**名字解析**(name resolving)，名字解析的结果是将名字引用与某个名字定义进行**绑定**(binding)。很多时候，需要通过上下文才能进行名字解析。我们将上下文相关的一组名字引用称为**名字引用组**(NameReferenceGroup)。名字引用组对应Java程序的表达式，因为不同的表达式给出了相关的名字引用的上下文信息。

15. 名字引用的主要属性是其中的名字(name)、源代码位置(location)、所在作用域(scope)，以及它的绑定结果(definition)。为处理对数组类型的引用，类型引用还有属性数组维数(dimension)。为正确绑定方法，方法引用还有属性实际参数列表(argumentList)和实际类型参数列表(typeArgumentList)。名字引用组的主要属性是某些表达式中的运算符(operator)以及在名字引用组中的子名字引用列表(subreferenceList)。

16. 图3.6给出了名字引用所涉及的主要实体及其之间的关系(主要是继承关系)。表3.6给出了主要的名字引用组，以及它对应的表达式的语法结构，和该名字引用组的可能绑定结果。

17. 名字引用的解析就是指确定名字引用到底是访问那个程序实体的过程，也称为**名字绑定**(name binding)。名字绑定的结果就是将名字引用与它所想访问的程序实体对应的名字定义相关联。名字绑定过程概括地说就是要在合适的作用域里匹配对应种类的名字定义。

18. 名字引用与名字定义的名字匹配的主要难点在于方法引用的绑定，它不仅要看方法名，而且还要考虑方法参数。简单地说，方法引用的实际参数类型要是方法定义的形式参数类型子类型才能绑定。如果有多个同名的、同参数个数的方法，则要寻找最精确的方法进行绑定。在绑定时要考虑继承成员。如果还要考虑多态性，则这个方法引用在运行时实际调用的方法还可能是所绑定的最精确方法所在类型的子类型中的重定义方法。

19. **子类型关系**(subtyping)是类型之间的一个自反、传递的关系。简单地说，对于基本数据类型，取值范围窄的数据类型可看做取值范围宽的数据类型的子类型；对于类或接口类型，类或接口之间的继承与实现关系给出了直接子类型关系，如果类型S的声明中extends或implements类型B，则类型A及其子类型都是类型B及其超类型的子类型。类和接口之间的子类型关系是直接子类型关系的自反传递闭包。

20. 名字引用的绑定需要在合适的作用域里匹配名字定义。对于基本的名字引用，从它所属的作用域开始匹配名字定义，然后再沿着作用域之间的包围关系在父作用域里匹配，一直到系统作用域。对于名字引用组，访问字段、调用方法和带限定的名字这些表达式中都可能需要基于其中一些子名字引用的绑定结果确定另外的子名字引用应该在哪个类型中进行匹配。

21. 总的来说，名字引用组的绑定结果是对应表达式的结果类型(所对应的类型定义)，它在访问字段、调用方法和带限定的名字这些名字引用组中用于确定另外的名字引用在哪个类型体作用域中进行匹配。在绑定方法引用时，实际参数也是表达式，对应名字引用组的绑定结果用于匹配方法引用的实际参数和方法定义的形式参数。

3.3 用例模型

在分析名字生成构件的领域模型，基本上弄清楚名字表有什么的基础上，这里再考虑名字表生成构件的用例模型。根据前面的需求概述，构件使用者对名字表的使用可概括为：(1) 名字定义的生

成、查找、遍历与访问；(2) 名字引用的生成、查找、遍历与访问。在领域模型中我们发现名字作用域也是一个重要实体，因此还应包括：(3) 名字作用域的生成、查找、遍历与访问。

注意，这里“查找”是指要求得到满足给定条件的某个名字定义或名字引用，“遍历”是指要求得到满足给定条件的所有名字定义或名字引用，“访问”是在得到名字定义和名字引用之后访问它的属性。

通过图3.5的简单例子程序可以看到名字引用多于名字定义，显然源代码规模越大，名字引用越多，而且在绑定名字引用时可能需要在各种作用域内匹配名字定义，但名字定义中只出现少数名字引用。基于这些因素，对于某个源代码文件集，可以一次生成所有的名字定义，但最好在需要的时候生成名字引用。至于名字作用域，因为是名字定义的重要属性，而名字引用绑定都是在某个合适作用域内匹配名字定义，因此也必须在生成名字定义时一起生成。

因此，下面先讨论名字定义和作用域的生成，然后讨论名字定义的查找、遍历与访问，然后再讨论名字作用域的查找、遍历与访问，最后讨论名字引用的生成、查找、遍历与访问。

3.3.1 名字定义和作用域的生成

从构件使用者的角度看，名字定义和作用域的生成非常简单：构件使用者提供源代码文件集，名字表生成构件生成名字定义和作用域，然后构件使用者获得访问名字定义和作用域的入口，我们把这个访问名字定义和作用域的入口称为名字表管理器 (NameTableManager)，而把名字表生成构件中具体负责生成名字定义和作用域的构件称为名字定义生成器 (NameDefinitionCreator)（注意，作用域总是在生成名字定义时同时生成）。

表 3.7 用例1. 名字定义和作用域生成

【用例名称】 名字定义和作用域生成
【用例场景】
参与者：构件使用者
【用例价值】
构件使用者获得可以访问名字定义和作用域的名字表管理器
【用例描述】
<ol style="list-style-type: none"> 1. 构件使用者指定一个源代码文件集 2. 名字定义生成器生成名字定义和作用域，并返回名字表管理器 3. 构件使用者可检查名字定义和作用域生成时是否有问题，若有问题，则： <ol style="list-style-type: none"> 3.1 可要求名字定义生成器返回问题源代码文件单元名及问题描述； 3.2 或者构件使用者指定一个文件句柄用于输出问题源代码文件单元名及问题描述。
【用例约束】
除非内存分配失败，否则返回的名字表管理器不会是null

表3.7给出了这个用例的描述。名字定义和作用域生成过程中可能存在问题，构件使用者可检查是否存在问题，如果有问题则可得到有问题的源代码文件单元名及相应问题描述的列表，或者将问题源代码文件单元名及其问题描述输出到指定文件。生成名字定义和作用域的存在问题通常是因为某个源代码文件存在编译错误（即Eclipse JDT不能为之生成正确的抽象语法树）。另外，除非内存分配失败，否则构件使用者得到的名字表管理器（名字表入口）不会是null，即使该源代码文件集

没有任何源代码文件，或者没有生成任何名字定义，因为从概念上来说，总是存在一个系统作用域，而且有一些自动导入的类型。

3.3.2 名字定义的查找、遍历与访问

名字定义的查找与遍历关键在于：(1) 在查找与遍历时可以设置怎样的条件；(2) 如何设置查找与遍历的条件。根据名字表生成构件的领域模型，名字定义的主要属性有简单名、全限定名、源代码位置和所在作用域。因此，查找与遍历时的条件主要是这些属性的组合，例如：

1. 基于简单名的条件：查找与遍历具有与指定字符串匹配的简单名的名字定义；
2. 基于全限定名的条件：查找与遍历具有与指定字符串匹配的全限定名的名字定义；
3. 基于源代码位置的条件：查找与遍历在某个源代码位置范围内的名字定义；
4. 基于作用域的条件：查找与遍历某个或满足某个条件的多个作用域内的名字定义。

不过有太多的条件组合方式，因此最好地方法是支持构件使用者自己定制查找与遍历的条件。为此引入名字定义过滤器 (NameDefinitionFilter) 的概念，它提供统一的接口判断一个名字定义是否符合构件使用者的查找或遍历条件。同时为查找与遍历名字定义，引入一个名字定义访问器 (NameDefinitionVisitor)，它使用名字定义过滤器，基于名字管理器提供的名字入口，查找与遍历满足条件的名字定义。

表 3.8 用例2. 名字定义的查找与遍历

【用例名称】 名字定义的查找与遍历
【用例场景】
参与者：构件使用者
【用例价值】
构件使用者获得满足条件的一个或多个名字定义
【用例描述】
<ol style="list-style-type: none"> 1. 构件使用者创建名字定义访问器 2. 构件使用者为名字定义访问器设置名字定义过滤器 3. 名字定义访问器基于名字管理器提供的名字表入口遍历名字表 4. 构件使用者从名字定义访问器获得满足条件的一个或多个名字定义
【用例约束】
<ol style="list-style-type: none"> 1. 如果是查找一个名字定义，则不存在满足条件的名字定义时返回null 2. 如果是遍历多个名字定义，则不存在满足条件的名字定义时返回大小为0的列表，不会返回null

表3.8给出了基于名字定义访问器与名字定义过滤器对名字定义进行查找与遍历的用例描述。我们约束，当遍历多个名字定义时，遍历的结果总不会是null，如果没有任何满足条件的名字定义，则返回一个大小为0的列表。

名字定义的访问是指构件使用者可以访问名字定义的主要属性，这只要各个名字定义提供访问属性的方法即可，因此这里无需对相关的用例做更多的描述。

3.3.3 名字作用域的查找、遍历与访问

名字作用域的主要属性是名字、起始位置、终止位置、父作用域与子作用域列表。因此查找与

遍历名字作用域的条件通常有：

1. 查找名字为指定字符串的作用域；
2. 查找包含某指定源代码位置的最内层作用域；
3. 遍历某一种类（例如所有详细类型定义、所有编译单元作用域等）的所有作用域；
4. 查找包含某一名字定义的最内层详细类型定义；
5. 查找包含某一名字定义的编译单元作用域。

也可引入名字作用域过滤器(NameScopeFilter)用于构件使用者自己定制查找与遍历名字作用域的条件，同时使用名字作用域访问器(NameScopeVisitor)支持构件使用者对名字作用域的访问。表3.9给出了基于名字作用域访问器与名字作用域过滤器对名字作用域进行查找与遍历的用例描述，该用例与基于名字定义访问器与名字定义过滤器对名字定义进行查找与遍历的用例基本相同。

表 3.9 用例3. 名字作用域的查找与遍历

【用例名称】 名字作用域的查找与遍历
【用例场景】
参与者：构件使用者
【用例价值】
构件使用者获得满足条件的一个或多个名字作用域
【用例描述】
<ol style="list-style-type: none"> 1. 构件使用者创建名字作用域访问器 2. 构件使用者为名字作用域访问器设置名字作用域过滤器 3. 名字作用域访问器基于名字管理器提供的名字表入口遍历名字表 4. 构件使用者从名字作用域访问器获得满足条件的一个或多个名字作用域
【用例约束】
<ol style="list-style-type: none"> 1. 如果是查找一个名字作用域，则不存在满足条件的名字作用域时返回null 2. 如果是遍历多个名字作用域，则不存在满足条件的名字作用域时返回大小为0的列表，不会返回null

3.3.4 名字引用的生成与使用

名字引用在一个软件项目中可能数量庞大，因此通常由构件使用者在需要的时候创建并使用，因此名字引用的生成、查找、遍历与访问通常是同时完成。名字引用的主要属性是名字、源代码位置、所在作用域和绑定结果。构件使用者通常需要：

1. 生成名字是指定字符串的所有名字引用；
2. 生成引用某个指定名字定义的所有名字引用（即绑定结果是该名字定义的名字引用）；
3. 生成某个指定作用域（特别是编译单元、类型或方法）的所有名字引用。

为生成名字引用，引入名字引用生成器(NameReferenceCreator)用于支持在已有名字管理器的基础上生成满足条件的名字引用。为使用条件过滤所生成的名字引用，引入名字引用过滤器(NameReferenceFilter)支持生成满足构件使用者定制条件的所有名字引用。因为是在使用时才生成，因此名字引用无需名字引用访问器。

表3.10给出了基于名字引用生成和名字引用过滤器生成指定作用域的满足条件的多个名字引用的用例描述，在该用例中构件使用可指定一个名字作用域，或者设置名字作用域过滤器在名字管理

表 3.10 用例4. 名字引用的生成与使用

【用例名称】 名字引用的生成与使用
【用例场景】
参与者：构件使用者
【用例价值】
构件使用者获得满足条件的一个或多个名字引用
【用例描述】
<ol style="list-style-type: none"> 1. 构件使用者基于名字管理器创建名字引用生成器 2. 构件使用者为名字引用生成器设定名字引用过滤器 3. 构件使用者设置名字作用域或名字作用域过滤器 4. 构件使用者从名字引用生成器获得满足条件的多个名字引用
【用例约束】
如果没有生成满足条件的名字引用，则返回大小为0的列表，而不会返回null

器中遍历得到多个名字作用域，生成所有这些名字作用域中的名字引用。为方便构件使用者的使用，用例的结果总是返回一个列表，即使没有生成满足条件的名字引用，也返回一个大小为0的列表，而不会返回空对象(null)。

3.3.5 名字表与抽象语法树

名字表是通过遍历源代码文件集每个源代码文件的抽象语法树而生成，因此构件使用者有时可能关心抽象语法树与名字表之间的一些联系，例如：

1. 查找指定名字定义（特别是详细类型定义、方法定义、字段定义）所在的抽象语法树节点；
2. 查找指定抽象语法树节点中满足条件的一个或多个名字定义；
3. 生成指定抽象语法树节点中满足条件的一个或多个名字引用。

为此使用名字表抽象语法树桥接器(NameTableASTBridge)，基于已有名字定义和作用域的名字表管理器完成与名字表与抽象语法树之间联系的管理。上述功能的用例事件流很简单，首先使用名字表管理器（及其源代码文件集）创建名字表抽象树桥接器，使用该桥接器就可为指定的名字定义查找所在的抽象语法树节点，查找指定抽象语法树节点中的名字定义，生成并返回指定抽象语法树节点中的名字引用等。为了使得到的名字定义和名字引用满足构件使用者定制的条件，构件使用者在查找名字定义或名字引用时提供名字定义过滤器或名字引用过滤器。

3.3.6 名字表生成构件功能总结

由于名字表生成构件是可重用构件，主要是提供一些API供构件使用者调用，因此与构件使用者的交互通常比较简单，因此这里不使用用例图描述名字表生成构件的用例模型，而是使用表3.11总结上面用例分析中所提到的名字表生成构件应该提供的功能，当然这个表还补充了一些功能（像遍历某一种类的所有名字定义），而且按照名字生成、名字定义使用、名字作用域使用、名字引用生成与使用、名字表与抽象语法树节点分别编号（以便可能补充更多功能）。

表 3.11 名字表生成生成构件功能列表

编号	功能描述	编号	功能描述
A1	生成所有的名字定义和名字作用域	A2	检查名字定义和作用域生成时的问题
A3	返回或输出名字定义和作用域生成时的问题		
B1	遍历某一种类的所有名字定义	B2	查找简单名与指定字符串匹配的首个名字定义
B3	遍历简单名与指定字符串匹配的所有名字定义	B4	查找全限定名与指定字符串匹配的名字定义
B5	遍历起始位置到终止位置间的所有名字定义	B6	遍历指定作用域的所有名字定义
B7	查找指定名字定义过滤器接受的首个名字定义	B8	遍历指定名字定义过滤器接受的所有名字定义
C1	查找名字为指定字符串的名字作用域	C2	查找包含指定源代码位置的最内层作用域
C3	遍历某一种类的所有名字作用域	C4	查找包含指定名字定义的最内层方法定义
C5	查找包含指定名字定义的最内层类型定义	C6	查找包含指定名字定义的最内层编译单元
C7	遍历名字作用域过滤器接受的所有名字作用域		
D1	生成名字是指定字符串的所有名字引用	D2	生成引用指定名字定义的所有名字引用
D3	生成指定作用域的所有名字引用		
E1	查找指定名字定义所在的抽象语法树节点	E2	查找指定抽象语法树节点满足条件的名字定义
E3	生成指定抽象语法树节点满足条件的名字引用		

3.3.7 名字生成构件领域模型补充

3.2节详细讨论名字生成构件的领域模型，但那里主要是分析名字表到底是什么、到底有什么。在上面进一步讨论名字生成构件功能时，为生成与使用（查找、遍历和访问）我们又引入了一些新的实体：

1. 名字表管理器(NameTableManager): 可看做是名字表的入口，大部分名字定义、作用域及名字引用的访问都应该通过名字表管理器进行。主要的属性是源代码文件集(sourceCodeFileSet)，一个名字表是针对一个源代码文件集而生成的。
2. 名字定义生成器(NameDefinitionCreator): 遍历源代码文件集的每个源代码文件，并生成名字定义和名字作用域。
3. 名字定义访问器(NameDefinitionVisitor): 遍历并返回满足条件的所有名字定义。
4. 名字定义过滤器(NameDefinitionFilter): 在遍历名字定义时指定名字定义的过滤条件。
5. 名字作用域访问器(NameScopeVisitor): 遍历并返回满足条件的所有名字作用域。
6. 名字作用域过滤器(NameScopeFilter): 在遍历名字作用域时指定名字作用域的过滤条件。
7. 名字引用生成器(NameReferenceCreator): 生成满足指定条件（通常是某个作用域）的所有名字引用。
8. 名字引用过滤器(NameReferenceFilter): 在生成名字引用时指定名字引用的过滤条件。
9. 名字表抽象语法树桥接器(NameTableASTBridge): 管理名字表（名字定义、名字作用域、名字引用）与生成名字表的抽象语法树之间的联系。

在多数时候，由于软件项目规模的问题，我们是先生成所有的名字定义和名字作用域，然后在需要的时候生成名字引用。不过对于一些小型的项目，我们也可能同时生成名字定义、名字作用域和名字引用，因此我们也可引入：

10. 名字表生成器(NameTableCreator): 同时生成名字定义、作用域和名字引用。

我们也为名字定义访问器、名字作用域访问器引入超类:

11. 名字表访问器(NameTableVisitor): 作为遍历名字表的访问器的共同基类;

同时也可为名字引用引入访问器, 以应对小型项目中同时生成的名字定义、名字作用域和名字引用的名字表:

12. 名字引用访问器(NameReferenceVisitor): 用于遍历名字引用。

我们也为名字定义过滤器、名字作用域过滤器、名字引用过滤器引入共同的超类:

13. 名字表过滤器(NameTableFilter): 作为设置查找条件的名字表项目的过滤器的共同基类。

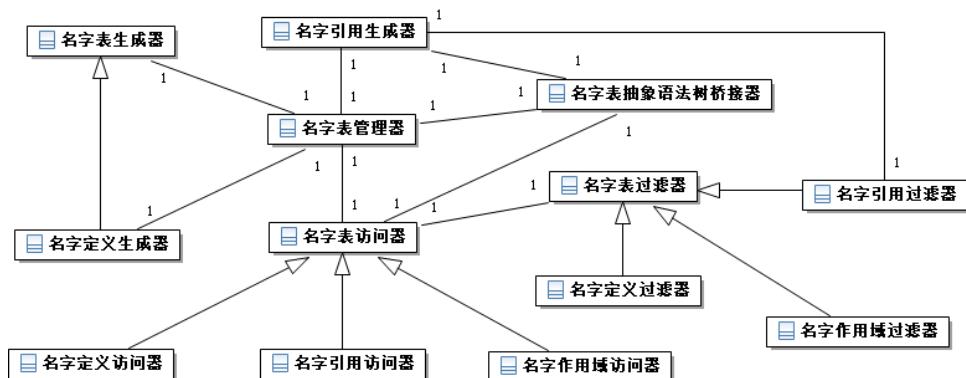


图 3.7 名字表构件的使用所涉及的实体

图3.7给出了上述实体及其之间基本的泛化和关联关系, 其中核心的实体是名字表管理器, 它使用名字表访问器(相应的子类)遍历名字表, 找到满足条件的名字表项目(名字定义、作用域或名字引用), 名字表访问器使用名字表过滤器(相应的子类)过滤满足条件的名字表项目。名字表管理器使用名字表生成同时生成名字定义、作用域和名字引用, 或者只使用名字定义生成器生成名字定义和名字作用域, 然后在需要的时候使用名字引用生成器生成名字引用。名字引用生成器在生成名字引用时可能使用名字引用过滤器过滤满足条件的名字引用。名字表抽象语法树桥接器可能使用名字表访问器遍历某抽象语法树节点的名字定义, 也可能使用生成器生成某抽象语法树节点的名字引用。

第四章 名字表生成构件的设计

根据??节的需求分析，名字表生成构件的主要实体可分为存储名字表的名字定义、名字作用域和名字引用，使用名字表的名字表管理器、访问器、过滤器、名字表抽象语法树桥接器，以及生成名字表的名字表生成器。这里在名字表生成构件领域的基础上，设计其中的类，主要是考虑各个类的职责，然后考虑如何基于类的职责实现用例。

4.1 名字定义与名字作用域

首先考虑名字定义和名字作用域，这两者同时生成，而且有些名字定义也是名字作用域。Java语言没有多继承，因此我们选择将名字作用域(NameScope)设计为接口，而名字定义(NameDefinition)为类。有多种不同的名字作用域和名字定义，因此我们引入两个枚举类型分别标识名字作用域和名字定义的类别(kind)。

4.1.1 枚举—名字作用域类别和名字定义类别

分别使用枚举类型名字作用域类别(NameScopeKind)和名字定义类别(NameDefinitionKind)标识名字作用域和名字定义的类别。表4.1给出了这两个枚举类型的枚举常量。注意，虽然我们没有将方法参数定义与变量定义在实体上分开（即没有单独为方法参数定义设置类），但我们仍用了不同的枚举常量来标识。

表 4.1 名字作用域类别和名字定义类别

名字作用域类别		名字定义类别	
常量	含义	常量	含义
NSK_SYSTEM	系统作用域	NDK_PACKAGE	程序包定义
NSK_PACKAGE	程序包作用域	NDK_TYPE	类型定义
NSK_COMPILATION_UNIT	编译单元作用域	NDK_FIELD	字段定义
NSK_TYPE	类型体作用域	NDK_METHOD	方法定义
NSK_METHOD	方法作用域	NDK_VARIABLE	变量定义
NSK_LOCAL	局部作用域	NDK_PARAMETER	方法参数定义
		NDK_TYPE_PARAMETER	类型参数定义
		NDK_ENUM_CONSTANT	枚举常量定义
		NDK_STATIC_MEMBER	导入静态成员定义

4.1.2 接口-名字作用域

名字作用域设计为Java语言接口，因此领域模型中所说的属性需要转换为职责（即返回该属性的操作或说方法）。名字作用域的主要职责包括：

- (RP1). 返回作用域名字(`getScopeName()`);
- (RP2-3). 返回作用域起始位置(`getScopeStart()`)，和终止位置(`getScopeEnd()`);
- (RP4). 返回包围作用域（即父作用域）(`getEnclosingScope()`);
- (RP5). 返回子作用域列表(`getSubScopeList()`)，即该作用域直接包围的作用域列表。

从概念上说，**一个Java源代码文件集的所有名字作用域以系统作用域为根节点，以作用域间的包围关系构成树**，通过子作用域列表可得到一个作用域的（直接）儿子节点，而通过包围作用域可得到它的（直接）父亲节点。

名字作用域是程序源代码的一片区域，可能需要判断某一程序位置是否在作用域中：

(RP6). 判断给定位置是否在作用域中(`containsLocation()`)，如果给定的位置在相同的源代码文件（编译单元）中，而且大于等于作用域起始位置的行号和列数，小于作用域终止位置的行号和列数，则在作用域中。

作用域的父作用域只是给出了直接包围该作用域的作用域，可能需要判断当前作用域是否在被直接或间接包含在另一作用域中：

(RP7). 判断是否在给定作用域中(`isEnclosedInScope()`)：如果给定的作用域在当前作用域到系统作用域（根节点）的路径上，则返回**true**，否则返回**false**。

每个作用域都应该可以返回它的作用域类别：

- (RP8). 返回名字作用域类别(`getScopeKind()`)。

从概念上说，名字定义是定义在作用域中，因此我们按作用域组织名字定义，而不是将所有的名字定义放在一个列表中，这样设计的好处在于在名字引用绑定时容易实现沿着名字作用域的包围关系由内层至外层匹配名字引用的名字与名字定义的简单名，缺点时可能查找与定位的效率不高。但我们生成名字表的主要用途是理解名字引用（即绑定名字引用）。因此名字作用域提供如何职责：

- (RP9). 在当前作用域定义名字(`define(NameDefition)`)。

名字引用的绑定是从名字引用所在的作用域开始解析，因此作用域需完成以下职责：

- (RP10). 在当前作用域中解析名字引用(`resolve(NameReference)`)。

4.1.3 抽象类-名字定义

我们将名字定义(`NameDefinition`)设计为抽象类，不允许创建对象实例，因为真正的名字定义应该是它的具体子类。根据前面的领域分析，名字定义的主要属性是：

- (AT1-2). 简单名(`simpleName`)、全限定名(`fullQualifiedName`);
- (AT3-4). 源代码位置(`location`)、所在作用域(`scope`)。

从概念上说，不同名字定义的全限定名应该是唯一的，但为了保险起见，我们认为名字定义的“简单名+源代码位置”是唯一的。不过由于导入类型定义（特别是对应基本数据类型，如`int`）没有在源代码中，因此我们**允许名字定义的源代码位置为空对象**。从而，名字定义的唯一标识是：**如果其原代码位置不空，则是“简单名+源代码位置”，否则是它的全限定名**。

从概念上，我们将名字定义设计为**不变类**(immutable class)，也即一旦创建一个名字定义（其任何具体子类）的对象实例之后不会再改变，或者更准确地说，在生成所有的名字定义和作用域之后，**名字定义和作用域（的任何具体实例）都不会再改变**，这意味着：

- (1). 不会为该源代码文件集中对该名字定义（或名字作用域）再创建其他对象实例；
- (2). 因此，**判断两个名字定义（或名字作用域）是否相等，直接使用对象引用相等比较（浅比较）即可**，无需通过唯一标识判断。

名字定义可能放在一些Java容器供快速查找与比较，因此它实现接口`Comparable<NameDefinition>`，并遵循上述原则实现`compareTo()`方法和`hashCode()`方法，即基于名字定义的唯一标识进行比较和计算散列值。

因此这个类使用构造函数设置上述四个属性的值，而不提供`setter`方法对这些属性值进行设置。这个类上述四个属性提供相应的`getter`方法。

这个类主要职责除上述属性的`getter`方法外，还有：

(RP1). 返回名字定义类别(`getDefinitionKind()`)：为确定名字定义的类别，这个类无需设置属性来保持这个类别，因为它的具体子类很容易确定自己的类别。因此这个方法在这个类中定义为**抽象方法**。

(RP2). 匹配名字引用(`match(NameReference)`)：将名字引用中的名字与当前名字定义的简单名进行比较。目前的比较方式就是精确比较。在名字引用解析中，所有对名字引用中名字的匹配最终都应该是调用这个方法，其他类不应该再提供匹配名字引用名字的功能。因此这个方法在这个类中定义为**终态**(final)方法。

如果一个源代码文件集的名字定义和名字作用域生成是正确的，那么对任意的名字定义，如果它的源代码位置不为空，则它的源代码位置应该包含在它的作用域内，即对任意名字定义对象`definition`，下面表达式应该返回`true`：

```
definition.scope.containsLocation(definition.location)
```

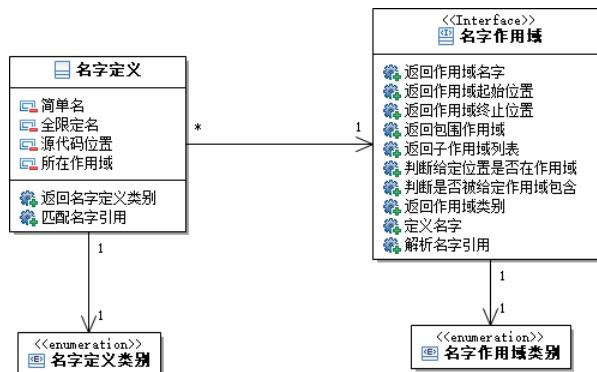


图 4.1 类名字定义和接口名字作用域

图4.1给出了类名字定义(`NameDefinition`)和接口名字作用域(`NameScope`)的类图，以及它们之间的关联关系。名字定义与名字作用域间的关联关系可概括为：每个名字定义都有所在的名字作用域，而一个名字作用域中可能定义多个名字定义。每个名字定义都有一个名字定义类别，而每个名字作用域也有一个名字作用域类别。

4.1.4 类—程序包定义

类程序包定义(PackageDefinition)继承类名字定义(NameDefinition), 并实现接口名字作用域(NameScope)。

除继承类名字定义的属性外, 程序包定义的主要属性是:

(AT1). 编译单元列表(unitList): 元素类型是CompilationScopeUnit, 作为作用域, 程序包所包含的编译单元是它的子作用域。

程序包定义的构造方法以程序包的简单名、全限定名和所属的系统作用域为参数。程序包定义对于名字作用域(NameScope) 接口中需要实现的方法:

(RP1). 返回作用域名字(getScopeName()): 以程序包定义的简单名为作用域名;

(RP2-3). 返回作用域起始位置(getScopeStart()), 和终止位置(getScopeEnd()): 都返回null;

(RP4). 返回包围作用域(即父作用域)(getEnclosingScope()): 返回程序包定义所在的作用域, 也即系统作用域。我们理解所有的程序包都定义在系统作用域, 即使是程序包的子程序包, 因为从作用域角度看, 程序包与子程序包没有包含(包围)关系;

(RP5). 返回子作用域列表(getSubScopeList()), 返回所包含的编译单元列表。

(RP6). 判断给定位置是否在作用域中(containsLocation()), 如果给定位置中的源代码文件单元名与程序包的编译单元列表中某个编译单元的单元名相同, 则返回true, 否则返回false。

(RP9). 在当前作用域定义名字(define(NameDefition)): 直接返回。我们不将任何名字定义直接放在程序包定义中。

(RP10). 在当前作用域解析名字引用(resolve(NameReference)): 虽然程序包定义中没有直接存放名字定义, 但是程序包级和公有的顶层类型是程序包的成员, 具有程序包作用域, 因此在程序包定义中解析名字引用就是将给定的名字引用与程序包中所有的编译单元作用域中的顶层类型进行匹配。

程序包定义对名字作用域接口要实现的另外两个方法 ((RP7)判断是否在给定作用域中、(RP8)返回作用域类别) 的实现很简单。

针对属性编译单元列表, 程序包定义应具有如下职责:

(RP11). 返回编译单元作用域列表(getCompilationUnitScopeList()): 这个方法与返回子作用域列表getSubScopeList()完全等价;

(RP12). 添加编译单元作用域(addCompilationUnitScope(CompilationUnitScope))。

最后, Java程序中可能有一个特殊的程序包, 即不具名程序包(unnamed package), 因此程序包定义提供如下职责:

(RP13). 返回是否是不具名程序包(isUnnamedPackage())。

4.1.5 类—编译单元作用域

编译单元作用域(CompilationUnitScope)实现接口名字作用域(SystemScope)。编译单元作用域对应Java软件的一个编译单元, 也即一个源代码文件。

编译单元作用域的名字就是源代码文件的单元名(即文件全名去掉源代码文件集的起始路径剩下的字符串)。起始位置和终止位置是该编译单元的文件起始位置和终止位置。包围它的作用域(父

作用域)是它所在的程序包定义,它的子作用域是定义在该编译单元的顶层类型,同时在编译单元作用域中定义的名字也是这些顶层类型。因此编译单元作用域有如下属性:

- (AT1). 单元名(unitName): 是编译单元作用域的作用域名字,也是源代码文件的单元名;
- (AT2-3). 起始位置(startLocatoin)和终止位置(endLocation);
- (AT4). 所属的程序包定义(enclosingPackage): 编译单元作用域的父作用域一个程序包定义;
- (AT5). (顶层)类型列表(typeList);

根据前面的讨论,我们需要为导入类型声明和静态成员导入声明创建类型引用和名字引用,因此有属性:

- (AT6). 导入类型引用列表(importedTypeList): 元素是类型引用(TypeReference);
- (AT7). 导入静态成员引用列表(importedStaticMemberList): 元素是名字引用。对于单个静态成员导入,为之创建名字引用,而对于按需静态成员导入,为之创建类型引用(因为按需静态成员导入除了最后的 "*" 之外,前面应该是一个类型引用)。

编译单元作用域的构造方法以单元名、起始位置、终止位置、所属程序包定义为参数。

编译单元作用域需要实现接口名字作用域(NameScope)所规定的职责:

(RP5). 返回子作用域列表(getSubScopeList()): 编译单元作用域的子作用域就是定义在该编译单元中的顶层向详细类型定义。

(RP8). 返回名字作用域类别(getScopeKind())。

(RP9). 在当前作用域定义名字(define(NameDefition)): 在编译单元作用域定义的名字就是它的顶层类型定义。

(RP10). 在当前作用域中解析名字引用(resolve(NameReference)): 在编译单元作用域解析名字需要将名字引用与顶层类型定义匹配,也要与导入类型引用列表中的类型引用所绑定的类型定义,以及导入静态成员引用列表中的名字引用所绑定的。注意,对应按需类型导入创建是程序包引用,那么要在该程序包引用所绑定的程序包定义中继续解析该名字引用。同样,对于对应按需静态成员导入所创建的类型引用,要在该类型引用所绑定的详细类型定义继续解析该名字引用。如果以上都不成功,则在该编译单元作用域所属的程序包定义域中再继续解析该名字引用。

注意,这种解析意味着如果与该编译单元属于同一程序包的其他编译单元定义了与导入类型声明同名的类型,则导入类型声明导入的类型遮蔽了其他编译单元定义的同名类型。

要实现的名字作用域的其他职责,包括(RP1)返回作用域名字、(RP2-3)返回作用域起始位置,和终止位置、(RP4)返回包围作用域(即父作用域)、(RP6)判断给定位置是否在作用域中、(RP7)判断是否在给定作用域中的实现都比较简单。

对于顶层类型列表,使用在当前作用域定义名字(define())这个操作添加元素到该列表,所以只提供getter方法:

(RP11). 返回顶层类型列表(getTypeList())。

对于导入类型引用列表和导入静态成员引用列表,提供如下操作:

(RP12). 添加导入类型引用(addImportedTypeReference(TypeReference));

(RP13). 返回导入类型引用列表(getImportedTypeList());

(RP14). 添加导入静态成员引用(addImportedStaticMemberReference(NameReference));

(RP15). 返回导入静态成员引用列表(getImportedStaticMemberList());

(RP16). 绑定导入声明(bindingImportDeclaration()): 将导入类型引用列表中的类型引用

以及导入静态成员引用列表中的名字引用进行绑定。这个应该在名字定义和作用域生成之后马上完成，其他名字引用的绑定应该完成这项功能之后才进行！

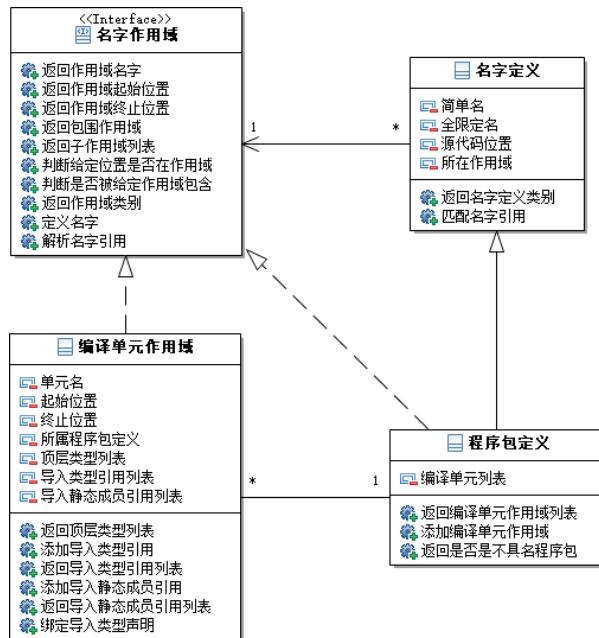


图 4.2 程序包定义和编译单元作用域

图4.2给出了类程序包定义(PackageDefinition)和编译单元作用域(CompilationUnitScope)的类图，以及它们之间的关联关系：每个程序包定义有多个编译单元作用域，而每个编译单元作用域必然属于某个程序包定义。

4.1.6 抽象类—类型定义

类型定义(TypeDefinition)是名字定义(NameDefinition)的直接子类。类型定义(TypeDefinition)主要是一个标志，标志它及它的派生类的对象实例都可看做是类型，可用于声明对象变量等。我们将类型定义也设计为抽象类，不能创建对象实例。

由于类型定义主要是标志，因此除继承名字定义的简单名、全限定名、源代码位置和所在作用域外，主要的属性是一个布尔类型标志用于区分一个类型是类还是接口，因为从名字引用绑定的角度看，类与接口没有实质差别，所以无需为类和接口分别引入类。

(AT1). 是否是接口(isInterface), 缺省值自然是false。

同时提供该属性的getter方法：

(RP1). 返回是否接口(isInterface())。

(RP2). 设置当前类型是接口(setInterface())。

类型定义的构造方法的参数与名字定义相同，以类型定义的简单名、全限定名、源代码位置和所在作用域为参数，供后代类调用以设置这些属性的值。

根据前面的领域模型，Java程序中类型之间最重要的关系是子类型关系(subtyping)，这个关系不仅适用于详细类型定义，而且也适用于导入类型定义之间的关系，因此判断子类型关系的基本职责应该属于类型定义：

(RP3). 是否是指定类型的子类型(isSubtypeOf(TypeDefinition)): 在这个类中实现有关自动导入类型之间的子类型关系判断(即int < long等的判断), 以及判断指定类型定义是否就是当前类型定义。

4.1.7 类-详细类型定义

详细类型定义(DetailedTypeDefinition)是类型定义(TypeDefinition)的直接子类, 而且实现接口名字作用域(NameScope)。可以认为详细类型定义所对应的作用域包括整个类型声明(从类型声明的修饰符开始, 到类型体右花括号处结束)。

除继承名字定义的属性简单名、全限定名、源代码位置和所在作用域外, 详细类型的主要属性是它的成员信息, 包括:

(AT1-3). 字段列表(fieldList)、方法列表(methodList)和(内部)类型列表(typeList);

注意, 这里说的字段、方法列表都是指在类型(类或接口)声明中直接给出的字段和方法成员, 不包括从超类型继承的成员。方法列表中包含构造方法。同样, 它的(内部)类型也是直接声明在类型体中的类型, 而不包括声明在方法中的局部类、匿名类。

在声明类型时还可能声明它继承某个类或实现多个接口, 继承的这个类和实现的接口都是这个类型的直接超类, 因此详细类型定义还有属性:

(AT4). (直接)超类型列表(superList): 继承的类和实现的接口都在这个列表中, 如果有继承的类, 则直接超类是列表的第一个元素。列表的元素是类型引用(TypeReference), 因为超类型声明实际上是对其他类型定义的引用。

类属类型(即带有形式类型参数的类型)还可能有类型参数, 因此详细类型定义还有属性:

(AT5). 类型参数列表(typeParameterList): 列表元素是类型参数定义(TypeParameterDefinition)。

在声明类型时还有一些修饰符(modifier), 用于确定这个类的访问权限(公有、程序包级、受保护、私有等), 以及是否是抽象类(abstract)、终态类(final)、静态类(static)等。声明修饰符有很多, 为简化存储, 我们使用一个整数标志存放这些修饰符:

(AT6). 类型声明修饰符(modifier): 一个用于标志这类的修饰符信息的整数。

Eclipse JDT中的类org.eclipse.jdt.core.dom.Modifier提供静态方法可用于判断这个整数包含怎样的修饰信息。

很多时候需要区分一个详细类型定义是否是顶层类, 我们使用一个布尔型标志记录这个类型是否是顶层类, 即是否是程序包的成员:

(AT7). 是否程序包成员(isPackageMember)。

详细类型定义在概念上也是一个作用域, 除了要给出它在源代码文件中的起始位置(使用继承自名字定义的位置(location)属性)外, 还要给出它的终止位置:

(AT8). 详细类型定义的源代码终止位置(endLocation)。

类型声明还可能含有多个初始化块(静态或非静态初始化块), 这些初始化块确定了一些局部作用域:

(AT9). 初始化块列表(initializerList)。

详细类型定义的构造方法以简单名、全限定名、源代码位置(起始位置)、终止位置和所在作用域为参数。这些属性一定设置就不能再改变, 因此无需为这些属性设置setter方法。

对于详细类型定义的职责, 首先是它要实现接口NameScope中规定的职责:

(RP1). 返回作用域名字(`getScopeName()`): 简单名可作为作用域名字;

(RP4). 返回包围作用域(即父作用域) (`getEnclosingScope()`): 就是该详细类型定义所在的作用域;

(RP5). 返回子作用域列表(`getSubScopeList()`): 它的子作用域包括它的初始化块、(内部) 类型和方法定义。

(RP9). 在当前作用域定义名字(`define(NameDefition)`): 详细类型定义作为作用域, 它所定义的名字包括它的字段定义、方法定义、(内部) 类型定义和类型参数定义。

(RP10). 在当前作用域中解析名字引用(`resolve(NameReference)`): 根据名字引用的类别, 在详细类型定义中匹配字段定义、方法定义、(内部) 类型定义和类型参数定义。注意, 当该名字引用在当前详细类型定义中无法绑定时, 需要优先考虑当前详细类型的超类型中对该名字引用继续绑定, 然后再考虑在包围当前详细类型定义的作用域(即当前详细类型定义所在的作用域)中继续绑定该名字引用。

其他要实现的职责((RP2-3)返回作用域起始位置和终止位置、(RP6)判断给定位置是否在作用域中、(RP7)判断是否在给定作用域中、(RP8)返回名字作用域类别)的实现都很简单。

除实现接口名字作用域中规定的职责外, 对于字段列表、方法列表和(内部)类型列表, 可以返回这些列表:

(RP11-13): 返回字段列表(`getFieldList()`)、返回方法列表(`getMethodList()`)、返回(内部)类型列表(`getTypeList()`)。

注意这些列表的元素添加(即添加字段、方法和(内部)类型)都是通过在详细类型定义中定义名字添加, 不提供额外的添加方法。

对于超类型列表(`superList`), 提供职责:

(RP14). 添加超类型引用(`addSuperType(TypeReference)`): 名字生成器保证如果有超类, 则是第一个增加的超类型。

(RP15). 返回超类型引用列表(`getSuperTypeList()`);

(RP16). 返回超类定义(`getSuperClassDefinition()`): 如果有超类, 返回超类引用所绑定的类型定义, 否则返回`null`。

对于类型参数列表, 提供职责:

(RP17). 返回类型参数列表(`getTypeParameterList()`)。

类型参数属于详细类型定义中的名字定义, 因此使用在作用域中定义名字(`define()`)这个方法增加类型参数, 不提供额外的方法增加类型参数。

基于修饰符, 提供有关类型的如下职责:

(RP18-21). 是否公有类型(`isPublic()`)、是否抽象类型(`isAbstract()`)、是否终态类型(`isFinal()`)、是否静态类型(`isStatic()`)。

(RP22). 是否程序包成员(`isPackageMember()`)。

对于初始化列表, 提供职责:

(RP23): 添加初始化块(`addInitializer(LocalScope)`): 初始化块被认为是一个局部作用域。

(RP24): 返回初始化块列表(`getInitializerList()`)。

详细类型定义为正确区别类与接口, 需要在生成名字定义时设置是否是接口:

(RP25). 设置为接口(`setInterface()`): 设置继承成员`isInterface`的值。

详细类型定义需重定义其直接超类类型定义(TypeDefinition)的判断子类型关系这个职责：

(RP26). 是否是指定类型的子类型(isSubtypeOf(TypeDefinition))：遍历超类型定义以及实现的接口定义列表，如果其中有指定的类型定义，则返回true，否则对于超类型定义以及实现的接口定义中的详细类型定义，判断它是否是指定类型定义的子类型（这里出现递归调用），如果是则返回true，否则返回false。

4.1.8 类—导入类型定义和导入静态成员定义

导入类型定义(ImportedTypeDefinition)是类型定义(TypeDefinition)的直接子类，它是编译单元中单个导入声明所导入的类型。导入类型声明中给出的名字是该导入类型定义的全限定名，而简单名则是全限定名去掉程序包名之后的名字，也是单个类型导入声明中最后的那个标识符。

从概念上说，导入类型声明的作用域是所在的编译单元作用域，但是可能在不同的编译单元会导入相同的类型，在每个编译单元创建导入类型定义不是好的设计，因此我们所有的导入类型定义放在系统作用域，而对应编译单元的每个单个类型导入声明，我们创建一个类型引用，该类型引用绑定到在系统作用域的导入类型定义，该类型引用的作用域是编译单元作用域，而导入类型定义本身的作用域是系统作用域，源代码位置为null。

因此，对于编译单元中的单个导入类型声明，我们实际上创建类型引用，并在名字定义和作用域生成之后，首先绑定这些类型引用。如果此导入类型声明导入的是源代码文件集中其他源代码文件声明的类型（即在名字表中有相应的详细类型定义），那么该引用将绑定到该详细类型定义，否则创建导入类型定义放在系统作用域，并将该类型引用绑定到该导入类型定义。基于这种设计，我们还可初步处理按需导入声明，即为按需导入声明创建程序包引用，将此程序包引用绑定到可能的程序包定义，那么当按需导入声明导入的程序包是源代码文件集中已有的程序包定义，则也可通过该程序包定义将此编译单元用到该程序包的类型引用绑定到正确的类型定义。

导入静态成员定义(ImportedStaticMemberDefinition)是名字定义(NameDefinition)的直接子类，它是编译单元单个静态导入声明所导入的静态成员（可能是数据成员也可能是方法成员）。单个静态导入声明中给出的名字是它的全限定名，而简单名则是去掉程序包名和类型名之后的名字（即单个静态导入声明中最后的那个标识符）。

我们也像单个导入类型声明那样处理单个静态导入声明：

- (1) 为单个静态导入声明创建名字引用(NameReference)，其作用域为所在的编译单元作用域；
- (2) 如果导入的静态成员所属类型不在源代码文件集的源代码中，则在系统作用域创建导入静态成员定义，并将创建的名字引用绑定到该静态成员定义，否则绑定到所属类型的静态成员。

因此导入类型定义和导入静态成员定义没有新的属性和方法（所有属性和方法都继承自超类型），而且它们的源代码位置都为null，所在作用域都为系统作用域。导入类型定义的属性是否接口isInterface由于没有更多信息而设置为缺省值false，也不重定义类型定义TypeDefinition的判断是否是给定类型的子类型(isSubtypeOf())这个方法。导入类型定义和导入静态成员定义的构造方法都以简单名、全限定名作为参数。

实际上，导入类型定义和导入静态成员定义存在的作用主要是：

- (1) 为基本数据类型（如int, double等）建立名字定义，使得基本类型可参与方法调用绑定时的类型推断；

(2) 为自动导入的类型(即java.lang程序包中的类型,如System, String等)建立名字定义,避免有太多名字引用(类型引用)无法绑定到合适的名字定义;

(3) 为导入的第三方构件库的类型(如Java容器类,像List之类)建立名字定义,也可以避免有太多名字引用无法绑定到合适的名字定义。

4.1.9 类-枚举类型定义和枚举常量定义

枚举类型定义(EnumTypeDefinition)是详细类型定义(DetailedTypeDefinition)的直接子类,因为Java语言的枚举类型被看做是一个类,它也可有构造方法、方法以及字段,并且可声明实现某些接口等。除此之外,它还有枚举常量列表:

(AT1). 枚举常量列表(constantList): 元素是枚举常量定义。枚举常量实际是该枚举类型的对象实例。

枚举类型的构造方法以简单名、全限定名、源代码位置(起始位置)、终止位置和所在作用域为参数。需要记住终止位置是因为枚举类型本身也是一个作用域。

对于枚举类型的职责,它因为是详细类型定义(DetailedTypeDefinition)的子类,因此也实现接口名字作用域(NameScope):

(RP1). 在当前作用域定义名字(define(NameDefition)): 枚举类型定义作为作用域,它所定义的名字包括枚举常量,原则上可以定义字段、方法和(内部)类型,但实际的Java程序中枚举类型中很少有字段和内部类型。

(RP2). 在当前作用域中解析名字引用(resolve(NameReference)): 对可能是引用枚举常量的名字引用(通常是字段引用或带限定的名字)进行绑定。

枚举类型的其他方法(包括实现名字作用域的方法)直接继承详细类型定义即可,无需重新实现。针对枚举常量列表,它需要增加操作:

(RP3). 返回枚举常量列表(getConstantList())。

注意枚举常量的增加通过实现名字作用域的在当前作用域定义名字(define())这个操作实现。

枚举常量定义(EnumConstantDefinition)作为名字定义(NameDefinition)的直接子类。虽然从概念上,枚举常量是枚举类型的对象实例,也是一个变量,但它与普通变量差别过大,所以目前暂时考虑使用直接作为名字定义子类的这种设计。

枚举常量中可以声明参数,实际上这些参数用于指定调用声明枚举类型时合适构造方法的实际参数,因此它有属性:

(AT1). 枚举常量参数列表(argumentList): 元素是名字引用。

枚举常量定义以简单名、全限定名(它的全限定名是它的简单名加上所属枚举类型定义的全限定名)、源代码位置和所在作用域(即它所属的枚举类型定义)为构造方法的参数。

对于参数列表,有如下操作:

(RP1). 返回枚举常量参数列表(getArgumentList())。

(RP2). 添加枚举常量参数(addArgument(NameReference))。

4.1.10 类-类型参数定义

类型参数定义是在声明类属类型或类属方法(包括构造方法)给出的形式类型参数,它也是类

型，可用于声明变量。因此它是类型定义TypeDefinition的直接子类，不过它的类别我们使用枚举常量NDK_TYPE_PARAMETER更准确地标识。它的简单名和全限定名相同，都是声明类型参数时的标识符，它的作用域是类型体作用域或方法作用域，从声明类型参数的地方开始，到类型体或方法体的结束处。它的源代码位置是声明类型参数的位置。

类型参数可以使用`extends`关键字声明多个类型的交集作为它的界，因此它的主要属性是：

(AT1). 类型参数的界类型列表(boundList)：界类型实际是对其他类型的一个引用，因此列表元素是类型引用(TypeReference)。

类型参数定义以简单名、源代码位置和所在作用域为它的构造方法参数，注意它的全限定名与简单名相同。

对于界类型列表，类型参数定义需要提供如下职责：

(RP1). 返回界类型列表(getBoundList())。

(RP2). 添加界类型引用(addBoundType(TypeReference))。

类型参数定义继承类型定义的属性`isInterface`，缺省值与类型定义一样为`false`。类型参数定义需要重定义类型定义的操作：

(RP3). 判断是否是指定类型定义的子类型(isSubtypeOf(TypeDefinition))：从概念上，我们暂时认为类型参数不是任何类型的子类型，因此总是返回`false`。类型参数的界是用来约束该（形式参数的）实际类型参数必须是这个界的子类型，不能认为类型参数本身是这个界的子类型。

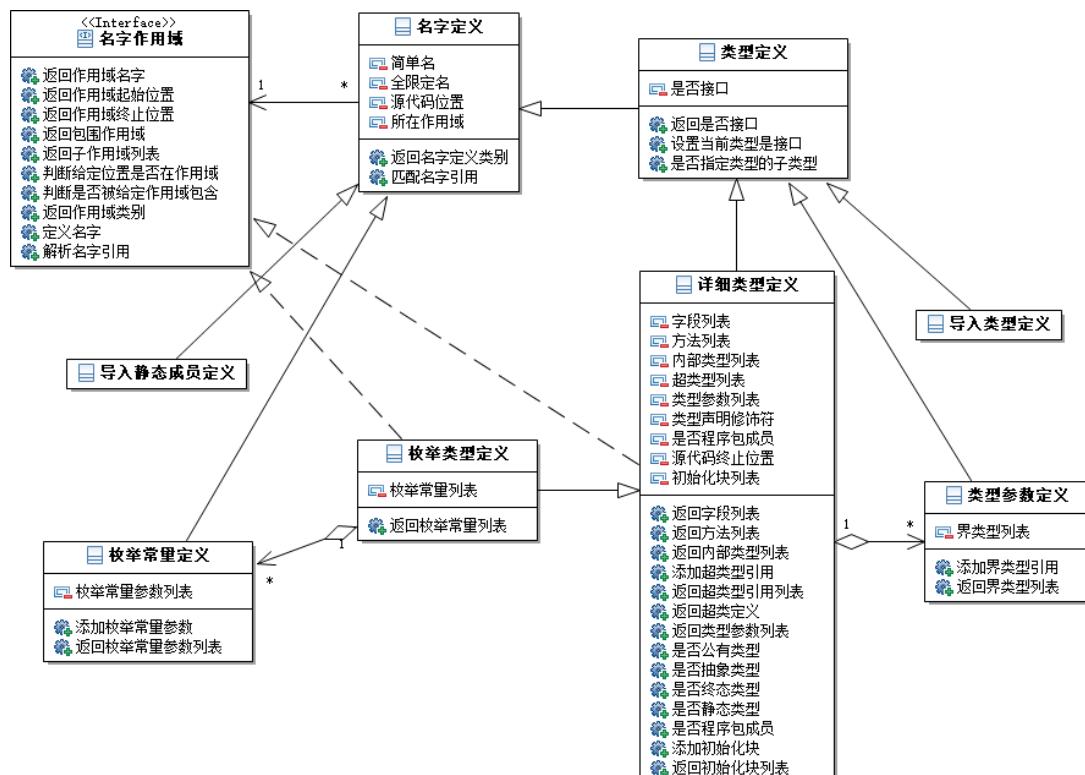


图 4.3 类型定义及其派生类与相关类

图4.3给出了类型定义及其派生类和几个相关类的类图。从图中可以看出，类型定义(TypeDefinition)的直接子类包括详细类型定义(DetailedTypeDefinition)、导入类型定义(ImportedTypeDefinition)和

类型参数定义(TypeParameterDefinition)。枚举类型定义(EnumTypeDefinition)又是详细类型定义的直接子类,其中只有详细类型定义和枚举类型定义是作用域,实现接口名字作用域(NameScope)。导入静态成员定义(ImportedStaticMemberDefinition)和枚举常量定义(EnumConstantDefinition)是名字定义(NameDefinition)的直接子类。枚举类型定义包含一个或多个枚举常量定义,而详细类型定义可能有零个或多个类型参数定义。

4.1.11 类-方法定义

方法定义(MethodDefinition)是名字定义(NameDefinition)的直接子类,而且实现接口名字作用域(NameScope)。可以认为方法定义所对应的作用域包括整个方法的声明(从方法声明的修饰符开始,到方法体的右花括号处结束)。

方法定义的主要属性包括:

(AT1). 返回类型(returnType): 这是一个类型引用(TypeReference);

(AT2). 参数列表(parameterList): 列表元素是变量定义(VariableDefinition),不过其名字定义类别为NDK_PARAMETER。

(AT3). 抛出类型列表(throwTypeList): 对应方法声明的抛出异常列表,元素是类型引用(TypeReference)。

(AT4). 类型参数列表(typeParameterList): 方法可以是类属方法,因此也可声明类型参数,元素是类型参数定义(TypeParameterDefinition)。

(AT5). 方法体(bodyScope): 方法体是一个局部作用域。

方法定义是一个作用域,除使用继承名字定义的属性位置(location)记录方法声明的起始位置外,还要记录作用域的终止位置:

(AT6). 终止位置(endLocation): 方法体的右花括号处是方法定义作为作用域的终止位置。

我们不为构造方法设计特别的类,而在方法定义中使用一个标记:

(AT7). 是否构造方法(isConstructor)。

在声明方法时还有一些修饰符(modifier),用于确定这个方法的访问权限(公有、程序包级、受保护、私有等),以及是否是抽象方法(abstract)、终态方法(final)、静态方法(static)等。与详细类型定义相同,我们使用一个整数标志存放这些修饰符:

(AT8). 方法声明的修饰符(modifier): 一个用于标志这类的修饰符信息的整数。

方法定义以简单名、全限定名(等于所属详细类型定义的全限定名加上简单名)、源代码位置(起始位置)、终止位置和所在作用域(即它所属的详细类型定义)为构造方法的参数。作为作用域,它需要记录终止位置。

方法定义是一个作用域,需要实现接口名字作用域(NameScope)指定的职责:

(RP1). 返回作用域名字(getScopeName()): 方法简单名作为作用域名字;

(RP4). 返回包围作用域(即父作用域)(getEnclosingScope()): 就是该方法定义所在的作用域;

(RP5). 返回子作用域列表(getSubScopeList()): 它的子作用域包括就是它的方法体。

(RP7). 判断是否在给定作用域中(isEnclosedInScope()): 如果给定的作用域在当前作用域到系统作用域(根节点)的路径上,则返回true,否则返回false。

(RP9). 在当前作用域定义名字(define(NameDefinition)): 方法定义作为作用域, 其中定义的名字包括方法参数和方法的类型参数。

(RP10). 在当前作用域中解析名字引用(resolve(NameReference)): 根据名字引用的类型匹配方法的参数和类型参数。

其他职责 ((RP2-3)返回作用域起始、终止位置、(RP6)判断给定位置是否在作用域中、(RP8)返回名字作用域类别) 的实现很简单。

对于方法的返回类型, 需要有操作:

(RP11). 设置方法的返回类型(setReturnType(TypeReference));

(RP12). 返回方法的返回类型(getReturnType());

(RP13). 返回方法返回类型定义(getReturnTypeDefinition()): 通常需要的是返回类型所对应的类型定义, 而不是对该类型的引用。

对于方法的参数列表, 需要有操作:

(RP14). 返回方法的参数列表(getParameterList()): 元素是变量定义(VariableDefinition)。如果方法没有参数则返回大小为0的列表, 不返回null。

注意, 由于方法参数是名字定义, 因此使用在当前作用域定义名字(define(NameDefinition))为方法增加参数, 不提供额外的增加参数的方法。

对于方法的抛出异常列表, 需要有操作:

(RP15). 返回抛出异常列表(getThrowTypeList()): 列表元素是TypeReference。如果没抛出异常列表, 则返回大小为0的列表, 不返回null。

(RP16). 添加抛出异常引用(addThrowType(TypeReference))。

对于方法的类型参数列表, 提供职责:

(RP17). 返回类型参数列表(getTypeParameterList())。

方法类型参数属于方法定义中的名字定义, 因此使用在作用域中定义名字(define())这个方法增加类型参数, 不提供额外的方法增加类型参数。

对于方法的方法体, 提供职责:

(RP18). 设置方法体(setBody(LocalScope));

(RP19). 返回方法体(getBody())。

使用如下操作设置方法是否是构造方法和返回方法是否构造方法:

(RP20). 设置为构造方法(setConstructor());

(RP21). 返回是否构造方法(isConstructor())。

基于方法的修饰符, 提供有关方法定义的如下职责:

(RP22-25). 是否公有方法(isPublic())、是否抽象抽象(isAbstract())、是否终态方法(isFinal())、是否方法类型(isStatic())。

在解析方法引用时, 要匹配方法引用和方法定义, 这个匹配不仅要检查名字是否相同, 而且还要匹配方法参数和方法类型参数, 我们将这个匹配的职责放在方法定义中:

(RP26). 匹配方法引用(matchMethod(MethodReference))。

为理解方法调用的多态性, 一个方法引用可能绑定到多个方法定义, 这些方法定义是静态绑定到的方法定义的重定义方法, 因此需要:

(RP27). 判断是否重定义指定方法(`isOverrideMethod(MethodDefinition)`): 如果当前方法重定义参数中指定的方法定义, 返回`true`。

4.1.12 类-字段定义和变量定义

字段定义(`FieldDefinition`)和变量定义(`VariableDefinition`)实质上是相同的, 都是确定一片存储区域用于存放程序要处理的数据。只是字段声明在类型定义中, 而变量声明在局部作用域中。

字段定义(`FieldDefinition`)是名字定义(`NameDefinition`)的直接子类。它的简单名是声明时给出的标识符, 全限定名包含所属的类型名。所在作用域是它所属的类, 源代码位置是声明时的位置。除这些属性之外, 字段定义还有:

(AT1). 字段类型(`type`): 即声明字段所使用的类型, 是一个类型引用(`TypeReference`)。

字段声明前也有修饰符, 因此也有:

(AT2). 字段声明的修饰符(`modifier`): 一个用于标志这类的修饰符信息的整数。

字段定义以简单名、全限定名(等于所属详细类型定义的全限定名加简单名)、源代码位置、所在作用域(即所属的详细类型定义)作为构造方法的参数。

对于字段类型, 字段定义应提供操作:

(RP1). 设置字段类型(`setType(TypeReference)`)。

(RP2). 返回字段类型(`getType()`): 返回声明字段时的类型引用。

(RP3). 返回字段类型定义(`getTypeDefinition()`): 通常我们更需要的是需要字段类型引用所绑定的类型定义。

基于字段的修饰符, 提供有关字段定义的如下职责:

(RP4-6). 是否公有字段(`isPublic()`)、是否终态字段(`isFinal()`)、是否静态字段(`isStatic()`)。

变量定义(`VariableDefinition`)也是名字定义(`NameDefinition`)的直接子类。它的简单名和全限定名相同, 都是声明时给出的标识符。所在作用域是声明该变量的局部作用域, 源代码位置是声明时的位置。同样, 变量定义也有:

(AT1). 变量类型(`type`): 即声明变量所使用的类型, 是一个类型引用(`TypeReference`)。

在Java语言中, 变量声明也可以有修饰符, 但目前我们暂时不关心局部变量的修饰符。我们将方法类型声明也看做变量定义, 但用不同的枚举常量标记, 因此变量定义还有属性:

(AT2). 变量类别(`kind`): 它的类型是`NameDefinitionKind`;

变量定义以简单名、源代码位置、所在作用域(即所属的局部作用域或方法作用域(对于方法参数))作为构造方法的参数。

同样对于变量类型, 变量定义提供操作:

(RP1). 设置变量类型(`setType(TypeReference)`)。

(RP2). 返回变量类型(`getType()`): 返回声明变量时的类型引用。

(RP3). 返回变量类型定义(`getTypeDefinition()`): 通常我们更需要的是需要变量类型引用所绑定的类型定义。

基于变量类别, 变量定义提供操作:

(RP4). 设置变量类别(`setKind(NameDefinitionKind)`);

(RP5). 是否方法参数(isMethodParameter()): 判断一个变量定义是否是方法的形式参数定义。

注意，变量定义需要根据属性变量列表(kind)重定义名字定义继承的操作返回名字定义类别(getDefinitionKind())。

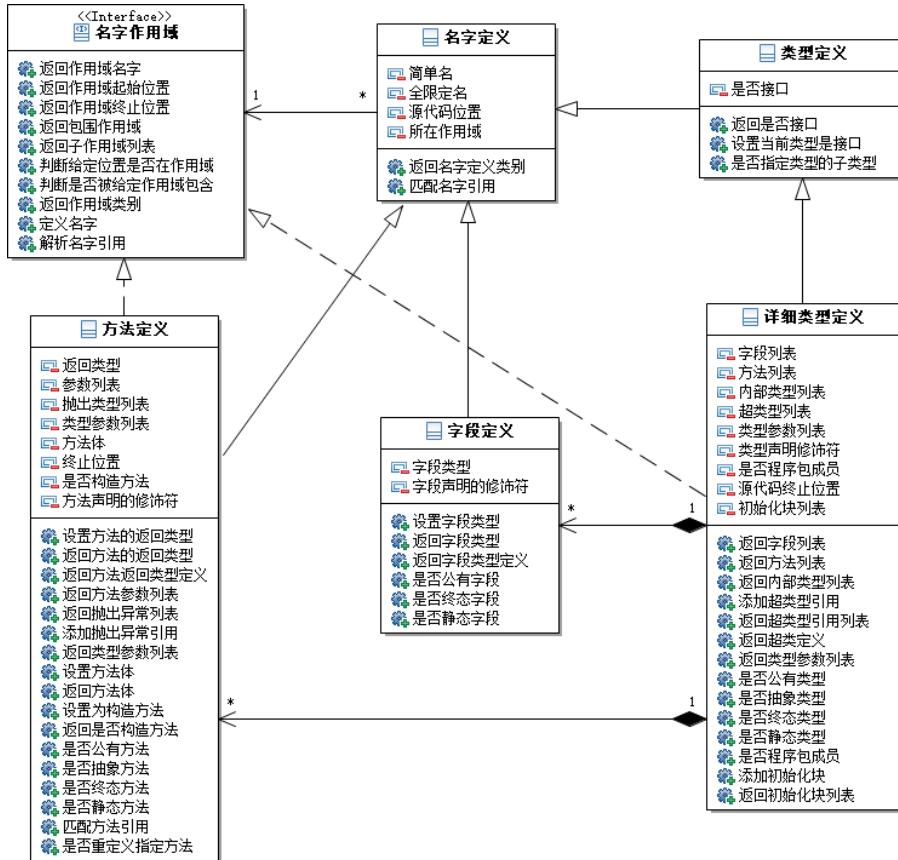


图 4.4 方法定义与字段定义

图4.4给出了类方法定义(MethodDefinition)和字段定义(FieldDefinition)。方法定义和字段定义都是名字定义的直接子类，其中方法定义实现接口名字作用域(NameScope)。详细类型定义(DetailedTypeDefintion)中有零个或多个方法定义或字段定义。

4.1.13 类-系统作用域

系统作用域(SystemScope)实现接口名字作用域(NameScope)。实际上，系统作用域是我们引入的概念，代表所有源代码构成的作用域，是作用域之间根据包含关系的构成的树结构的根。

系统作用域的名字是一个特别的字符串，例如"<System>"。系统作用域的起始位置和终止位置都是null，但是它包含任意合法的源代码位置。系统作用域的父作用域是null，它的子作用域包括所有的程序包定义。根据前面对导入类型和导入静态成员定义的设计，我们将这两者也定义在系统作用域中，因此系统作用域的主要属性有：

- (AT1). 程序包定义列表(packageList);
- (AT2). 导入类型定义列表(importedTypeList);

(AT3). 导入静态成员定义列表(`importedStaticMemberList`)。

系统作用域使用缺省的构造方法(即没有任何参数的构造方法)。系统作用域需要实现接口名字作用域所规定的职责:

(RP5). 返回子作用域列表(`getSubScopeList()`): 系统作用域的子作用域列表就是它的程序包定义列表。

(RP6). 判断给定位置是否在作用域中(`containsLocation()`): 总是返回`true`。

(RP7). 判断是否在给定作用域中(`isEnclosedInScope()`): 总是返回`false`。

(RP9). 在当前作用域定义名字(`define(NameDefinition)`): 根据名字定义的类别, 在系统作用域中增加程序包定义、导入类型定义和导入静态成员定义。

(RP10). 在当前作用域中解析名字引用(`resolve(NameReference)`): 根据名字引用的类别, 在系统作用域中匹配程序包定义、导入类型定义和导入静态成员定义。

其他职责((RP1)返回作用域名字、(RP2-3)返回作用域起始、终止位置、(RP4)返回包围作用域、(RP8)返回作用域类别)的实现都很简单。

对于上述三个列表, 系统作用域提供`getter`方法:

(RP11). 返回程序包定义列表(`getPackageList()`);

(RP12). 返回导入类型定义列表(`getImportedTypeList()`);

(RP13). 返回导入静态成员定义列表(`getImportedStaticMemberList()`)。

4.1.14 类-局部作用域

局部作用域(`LocalScope`)实现接口名字作用域(`NameScope`)。局部作用域对应Java程序中的一对花括号括起来的一个语句块(包括方法体、类的初始化块等)。通常Java程序中的语句块很多, 在生成名字表的时候应该只为那些内部有变量声明的语句块创建局部作用域对象实例。

局部作用域的主要属性有:

(AT1) 局部作用域的名字(`name`): 基于局部作用域左花括号的位置自动生成, 可采用"`<Block>+源代码位置串`的形式;

(AT2-3) 局部作用域的起始位置(`startLocation`)和终止位置(`endLocation`): 注意`for`语句(包括增强型`for`语句)的初始化表达式中声明的变量的局部作用域包括该初始化表达式;

(AT4) 所属作用域(`enclosingScope`): 局部作用域可能直接包含在详细类型定义(初始化块)、方法定义(方法体)或其他局部作用域中。

(AT5) 变量定义列表(`variableList`): 包括在局部作用域中声明的变量, 列表元素是变量定义`VariableDefinition`。

(AT6) 局部类型定义列表(`localTypeList`): 包括在局部作用域中声明的局部类和匿名类, 列表元素是详细类型定义(`DetailedTypeDefinition`)。匿名类的简单名可基于匿名类的起始源代码位置自动生成, 可采用"`<Class>+源代码位置串`的形式;

(AT7) 子局部作用域列表(`subLocalScopeList`): 包含在局部作用域中的子局部作用域, 列表元素是局部作用域。

局部作用域以名字、起始位置、终止位置和所属作用域为构造方法的参数。

局部作用域需要实现接口名字作用域规定的职责:

(RP5). 返回子作用域列表(getSubScopeList()): 它的子作用域包括局部类型定义和子局部作用域。

(RP9). 在当前作用域定义名字(define(NameDefition)): 在局部作用域中可定义的名字包括详细类型定义(局部类、匿名类)和变量定义。

(RP10). 在当前作用域中解析名字引用(resolve(NameReference)): 在局部作用域中匹配详细类型定义(局部类)和变量定义。

其他要实现的职责(RP1)返回作用域名字、(RP2-3)返回作用域起始位置、终止位置、(RP6)判断给定位置是否在作用域中、(RP7)判断是否在给定作用域中、(RP8)返回名字作用域类别)的实现都比较简单。

对于变量定义列表,增加变量定义使用实现名字作用域接口的操作在当前作用域定义名字(define())完成,但局部作用域需要提供getter方法:

(RP11). 返回变量定义列表(getVariableList());

同样对于局部类型定义列表,增加局部类型定义使用在当前作用域定义名字(define())这个操作完成,但局部作用域需提供操作:

(RP12). 返回局部类型定义列表(getLocalTypeList());

对于子局部作用域列表,需要提供职责:

(RP13). 添加子局部作用域(addSubLocalScope(LocalScope));

(RP14). 返回子局部作用域列表(getSubLocalScopeList())。

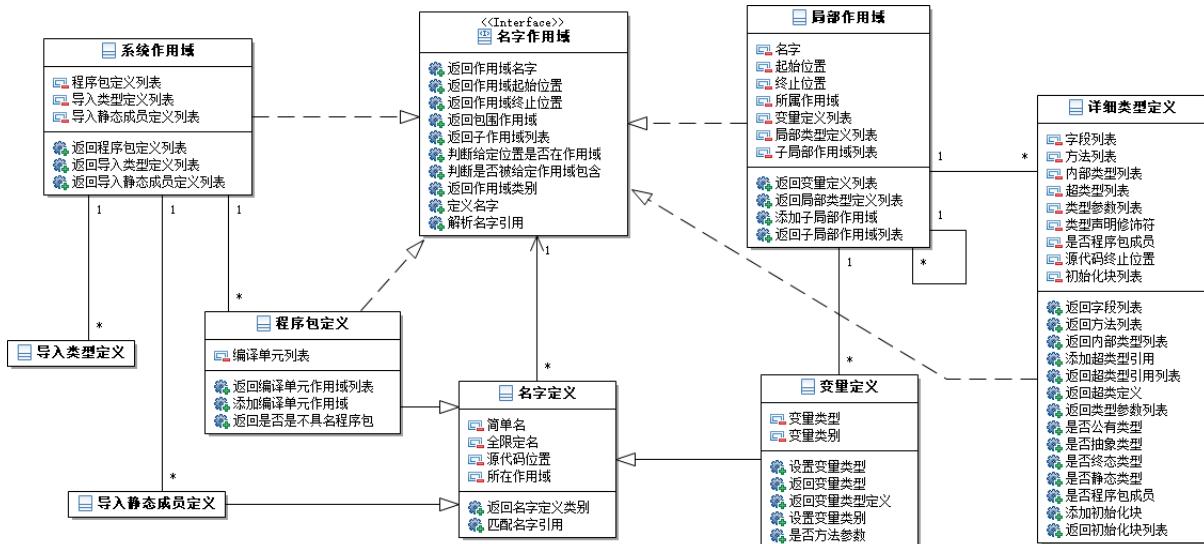


图 4.5 系统作用域、局部作用域和变量定义

图4.5给出了系统作用域和局部作用域,以及变量定义的类图。系统作用域包含零个或多个程序包定义、导入类型定义和导入静态成员定义。局部作用域包含零个或多个(局部)详细类型定义和子局部作用域,以及一个或多个变量定义。注意,图中导入类型定义、详细类型定义没有泛化关系指向名字定义是因为它们不是名字定义的直接子类。

4.1.15 名字定义和名字作用域相关类的设计小结

名字定义和名字作用域相关的类比较多，有必要在这里做一个简单的小结：

1. 图4.1、4.2、4.3、4.4和图4.5给出了这些类（和接口）的类图及它们之间的主要关系。

2. 名字定义(NameDefinition)的直接子类有：

- (1). 程序包定义(PackageDefinition)
- (2). 导入静态成员定义(ImportedStaticMemberDefinition)
- (3). 类型定义(TypeDefinition)
- (4). 方法定义(MethodDefinition)
- (5). 字段定义(FieldDefinition)
- (6). 变量定义(VariablenDefinition)
- (7). 枚举常量定义(EnumConstantDefinition)

3. 类型定义(TypeDefinition)的直接子类有：

- (1). 详细类型定义(DetailedTypeDefinition)
- (2). 导入类型定义(ImportedTypeDefinition)
- (3). 类型参数定义(TypeParameterDefinition)

4. 详细类型定义(DetailedTypeDefinition)的直接子类有：

- (1). 枚举类型定义(EnumTypeDefinition)

5. 实现接口名字作用域(NameScope)的类包括：

- (1). 系统作用域(SysemScope)
- (2). 程序包定义(PackageDefinition)
- (3). 编译单元作用域(CompilationUnitScope)
- (4). 详细类型定义(DetailedTypeDefinition)
- (5). 方法定义(MethodDefintion)
- (6). 局部作用域(LocalScope)

6. 注意，这些类的主要属性包括一些列表，针对这些列表，我们通常设置添加（单个）元素的方法，如果元素是作用域中的名字定义，则通过定义名字添加元素。返回这些列表的方法通常不返回null，即使没有元素也会返回大小为0的列表。

4.2 名字引用

名字引用是对程序实体（程序包、类型、字段、方法、变量）的使用，也有多种不同类别的名字引用，因此与名字作用域和名字定义类似，我们也使用一个枚举类型标记名字引用的类别。

4.2.1 枚举—名字引用类别

表4.2给出枚举类型名字引用类别(NameReferenceKind)中定义的枚举常量。因为Java程序中的实体主要有程序包、类型、方法、字段、变量，因此基本的名字引用是程序包引用、类型引用、方法引用、字段引用和变量引用。在名字解析时可能需要文字常量的类型推断整个表达式（名字引用组）的类型，因此有时需记录一些对文字常量的使用，我们将其归为文字引用类别。有些名字引用（主

要是带限定的名字)在生成时很难判断是哪个类别的引用,因此我们也引入了常量标记(暂时)未知类别的引用。

表 4.2 名字引用类别

常量	含义	常量	含义
NRK_PACKAGE	程序包引用	NRK_TYPE	类型引用
NRK_METHOD	方法引用	NRK_FIELD	字段引用
NRK_VARIABLE	变量引用	NRK_LITERAL	文字引用
NRK_GROUP	名字引用组	NRK_UNKNOWN	未知类别的引用

4.2.2 类-名字引用

类名字引用(*NameReference*)是所有表示名字引用的类的超类。与名字定义(*NameDefinition*)类不同,我们不将这个类设计为抽象类,实际上我们将名字引用这个类的对象实例作为最简单的名字引用,即对应一个标识符的名字引用。

根据领域分析,名字引用的主要属性包括:

- (AT1). 名字(*name*),对于基本名字引用(即不是名字引用组),通常是所使用实体的简单名(不含任何句点);对于名字引用组,是其对应的表达式(字符)串;
- (AT2). 源代码位置(*location*):名字引用中名字在源代码的位置,或者是名字引用组对应表达式在源代码的起始位置;
- (AT3). 所在作用域(*scope*):该名字引用出现的最内层作用域;
- (AT4). 绑定结果(*definition*):名字引用绑定的结果是一个名字定义,如果不能绑定则是*null*。
- (AT5). 名字引用类别(*kind*):注意,名字引用的类别在绑定的时候可能基于尝试的需要和结构而暂时改变。

不同的名字引用有不同的名字和源代码位置,因此名字引用以名字、源代码位置和所在作用域为构造方法的参数,一旦一个名字引用被创建,它的名字、源代码位置和所在作用域不再改变。名字引用之间的相等、比较和散列函数基于名字和源代码位置实现。由于同样的名字引用(同位置和同名字的引用)可能在不同的时候生成(因为名字引用是在需要时生成),因此名字引用的相等不能使用浅比较(即不能仅仅使用相等运算符==比较)。

名字引用提供的职责(操作)首先应该有:

- (RP1-3). 返回名字(*getName()*)、返回源代码位置(*getLocation()*)、返回所在作用域(*getScope()*);
- (RP4-5). 设置类别(*setReferenceKind(NameReferenceKind)*)、返回类别(*getReferenceKind()*);
- (RP6-7). 绑定到名字定义(*bindTo(NameDefinition)*)、返回绑定结果(*getDefinition()*);

名字引用的主要职责围绕名字解析和它所绑定的结果:

- (RP8). 解析名字引用(*resolveBinding()*):如果绑定成功则返回*true*,否则返回*false*;
- (RP9). 判断是否已解析(*isResolved()*):如果绑定结果不是*null*,返回*true*;

在解析名字引用,特别是解析名字引用组时,我们更关心绑定的结果类型:

(RP10). 返回绑定结果类型(getResultTypeDefinition()): 如果一个名字引用最终绑定到程序包定义或类型定义，则这个程序包定义或类型定义就是它的结果类型；如果一个名字引用最终绑定到字段定义或变量定义，则声明字段或变量的类型引用所能绑定的类型定义是它的结果类型；如果一个名字引用最终绑定到方法定义，则方法的返回类型引用所能绑定的类型定义是它的结果类型。

除为导入类型声明或导入静态成员声明所创建的名字引用不在类型的声明中外，其他名字引用（包括类型声明中的超类或实现接口而使用类型引用）都在某个详细类型定义中，更多时候需要关心名字引用所在的详细类型定义，因为经常需要在这个详细类型定义中解析名字引用：

(RP11). 返回所在的详细类型定义(getEnclosingTypeDefinition())。

4.2.3 类—程序包引用和文字引用

程序包引用(PackageReference)是名字引用(NameReference)的直接子类。程序包引用仅仅是一个标志，表明该名字引用已经确定是引用一个程序包定义。为方便起见，程序包引用中的名字可以是带句点的全限定名，以方便尽快匹配到一个程序包定义。

文字引用(LiteralReference)是名字引用(NameReference)的直接子类。文字引用通常用于名字引用组，以辅助判断其他名字引用或整个名字引用组的结果类型，所以对文字引用更关心它的类型，而不是它的文字常量值。因此，文字引用的名字存放的是对应文字常量的类型的名字，例如整数类型的文字引用的名字存放的是串“int”，我们将为基本数据类型int创建一个导入类型定义，其简单名就是“int”，整数类型的文字引用绑定到这个导入类型定义。

出现在表达式中的关键字this也创建对应的文字引用，这时它的名字是this，将绑定到这个文字引用所在的详细类型定义。

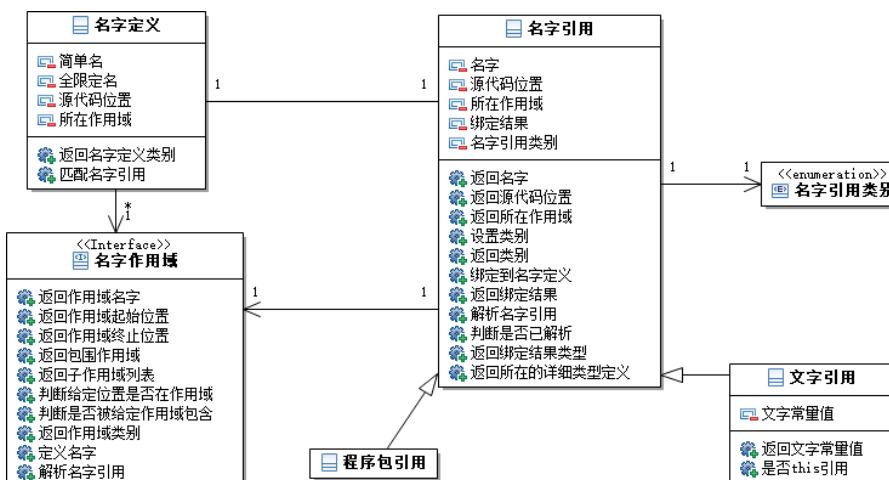


图 4.6 名字引用、程序包引用和文字引用

因此对于文字引用，它的属性有：

(AT1). 文字常量值(literal): 它是一个字符串，记录该文字常量的字面值；

文字引用以文字常量的类型名字、源代码位置、所在作用域和文字常量值为构造方法的参数。

文字引用除继承名字引用的职责外，它还应提供如下操作：

(RP1). 返回文字常量值(getLiteral()): 返回标识文字常量字面值的字符串；

(RP2). 是否this引用(isThisLiteral()): 返回是否是对应this关键字的文字引用; 文字引用需要重定义名字引用的解析名字引用(resolveBinding())操作:

(RP3). 解析名字引用(resolveBinding()): 如果是this引用, 则绑定到这个文字引用所在的详细类型定义; 否则利用文字常量的类型名字在系统作用域中匹配导入类型定义。

图4.6给出了名字引用、程序包引用和文字引用的类图。每个名字引用有一个名字引用类别, 而且名字引用与名字定义和名字作用域都有一对一的关联关系。

4.2.4 类—类型引用及其子类

类型引用(TypeReference)是名字引用(NameReference)的直接子类, 表示它所引用的一定是一个类型定义。随着Java语言的发展, 特别是类属机制的引入, 使用类型的方式越来越复杂, 因此类型引用有许多子类。为了区分这些子类, 我们也引入枚举类型类型引用类别(TypeReferenceKind)来标记类型引用的不同子类, 表4.3给出了类型引用类别中定义的常量。

表 4.3 类型引用类别

常量	含义	常量	含义
TRK_SIMPLE	简单类型引用	TRK_QUALIFIED	带限定名类型引用
TRK_NAMED	带名字类型引用	TRK_PARAMETRIZED	参数化类型引用
TRK_WILDCARD	通配符类型引用	TRK_INTERSECTION	交集类型引用
TRK_UNION	并集类型引用		

类型引用除继承名字引用的属性名字、源代码位置、所在作用域和绑定结果外, 还有属性:

(AT1). 数组维数(dimension): 如果引用该类型声明数组, 则记录所声明数组的维数, 否则为0;

(AT2). 类型引用类别(typeKind): 缺省值是TRK_SIMPLE。

类型引用也以名字、源代码位置和所在作用域为构造方法的参数。类型引用除继承名字引用的操作外, 它还应提供如下操作:

(RP1-2). 设置维数(setDimension(int))、返回维数(getDimension());

(RP3). 是否数组类型(isArrayType()): 如果数组维数大于等于1, 则返回true;

(RP4-5). 设置类型引用类别(setTypeKind(TypeKind))、返回类型引用类别(getTypeKind());

注意, 我们实际上不为简单类型引用(即只有一个标识符的类型引用)设置类, 而是将类型引用本身的对象实例看做是简单类型引用, 因此类型引用类别的缺省值是TRK_SIMPLE。因此类型引用这个类需要重定义名字引用中的解析名字引用操作实现对简单类型引用的解析:

(RP6). 解析名字引用(resolveBinding()): 如果是对基本数据类型的引用(这时属性名字记录的是基本数据类型的类型名字, 如"int"等), 则将其绑定到在系统作用域定义的导入类型定义, 否则在当前作用域匹配简单名等于类型引用的名字的类型定义。

带限定名类型引用(QualifiedTypeReference)是类型引用(TypeReference)的直接子类。它除继承类型引用(包括名字引用)的属性外, 还有:

(AT1). 限定类型(qualifier): 也是一个类型引用。

带限定名类型引用以名字、源代码位置和所在作用域为构造方法的参数。它除继承类型引用的操作之外，还有操作：

(RP1-2). 设置限定类型(`setQualifier(TypeReference)`)、返回限定子(`getQualifier()`)；

带限定名类型引用需要重定义：

(RP3). 解析名字引用(`resolveBinding()`)：先解析限定子，得到它的绑定结果（应该就是一个类型定义），然后在这个类型定义（如果是详细类型定义的话）中再匹配带限定名类型引用本身的名字，并且整个带限定名类型引用绑定到匹配到的类型定义。

带名字的类型引用(`NamedTypeReference`)是类型引用(`TypeReference`)的直接子类。它除继承类型引用（包括名字引用）的属性外，还有：

(AT1). 限定名字(`qualifier`)：它是一个名字引用（不能确定是否是类型引用）。

带名字的类型引用以名字、源代码位置和所在作用域为构造方法的参数。它除继承类型引用的操作之外，还有操作：

(RP1-2). 设置限定名字(`setQualifier(NameReference)`)、返回限定名字(`getQualifier()`)；

带名字的类型引用需要重定义：

(RP3). 解析名字引用(`resolveBinding()`)：先解析限定子，得到它的绑定结果类型（使用名字引用中定义的方法`getResultSetDefinition()`），如果返回的是详细类型定义，再在该详细类型定义中匹配带名字的类型引用本身的名字，并且整个带名字的类型引用绑定到匹配到的类型定义。

参数化类型引用(`ParameterizedTypeReference`)是对类属类型（带形式类型参数的类型）的引用，它是类型引用(`TypeReference`)的直接子类。它的属性名字将存放整个参数化类型引用（在源代码中的）字符串。除继承类型引用（包括名字引用）的属性外，还有：

(AT1). 主类型引用(`primaryType`)：也是一个类型引用(`TypeReference`)，它引用的类型应该是类属类型；

(AT2). 实际类型参数列表(`argumentList`)：是匹配类属类型的实际类型参数，每个类型参数都是一个类型引用，因此列表元素也是类型引用(`TypeReference`)。

参数化类型引用也以名字、源代码位置和所在作用域为构造方法的参数。参数化类型引用除继承类型引用的职责外，还有操作：

(RP1-2). 设置主类型引用(`setPrimaryType(TypeReference)`)、返回主类型引用(`getPrimaryType()`)；

(RP3-4). 添加类型参数(`addArgument(TypeReference)`)；返回类型参数列表(`getArgumentList()`)。

同样，参数化类型引用需要重定义：

(RP5). 解析名字引用(`resolveBinding()`)：解析主类型引用以及类型参数列表中的引用，然后整个参数化类型引用绑定到主类型引用的绑定结果。

通配符类型引用(`WildcardTypeReference`)是类型引用(`TypeReference`)的直接子类，它通常作为参数化类型引用的实际类型参数。它的属性名字也存放整个通配符类型引用（在源代码中的）字符串。除继承类型引用（包括名字引用）的属性外，还有属性：

(AT1). 通配符类型的界(`bound`)：它是一个类型引用，但可能是`null`；

(AT2). 是否上界(`isUpperBound()`)：如果是使用`extends`声明的界则为`true`，否则（即使用`super`声明的界）为`false`，缺省值是`true`。

通配符类型引用也以名字、源代码位置和所在作用域为构造方法的参数。除继承类型引用的职责外，还有操作：

- (RP1-2). 设置界(setBound(TypeReference))、返回界(getBound());
 (RP3-4). 设置为上界(setUpperBound(boolean))、返回是否为上界(isUpperBound());
 通配符类型引用需要重定义:

(RP5). 解析名字引用(resolveBinding()): 解析可能存在的界(类型引用), 最后整个通配符类型引用绑定到这个界的绑定结果, 如果不存在界(即界为null), 则绑定到java.lang.Object对应的导入类型定义。

交集类型引用(IntersectionTypeReference)和并集类型引用(UnionTypeReference)都是类型引用(TypeReference)的直接子类。它们的属性名字也都存放整个交集类型或并集类型(在源代码中的)字符串。除继承类型引用(包括名字引用)的属性外, 它们都还有属性:

(AT1). 类型引用列表(typeList): 参与交集运算或并集运算的两个或多个类型引用构成的列表, 列表元素是类型引用(TypeReference)。

这两个类都以名字、源代码位置和所在作用域为构造方法的参数。除继承类型引用的职责外, 还有操作:

- (RP1-2). 添加类型引用(addType(TypeReference))、返回类型引用列表(getTypeList())。

交集类型引用和并集类型引用都重定义:

(RP3). 解析名字引用(resolveBinding()): 解析所有参与运算的类型引用, 并将整个交集类型引用或并集类型引用绑定到第一个参与运算的类型引用的绑定结果(**因为我们目前对交集类型和并集类型的使用还不完全了解, 所以只能暂时这样!**)!

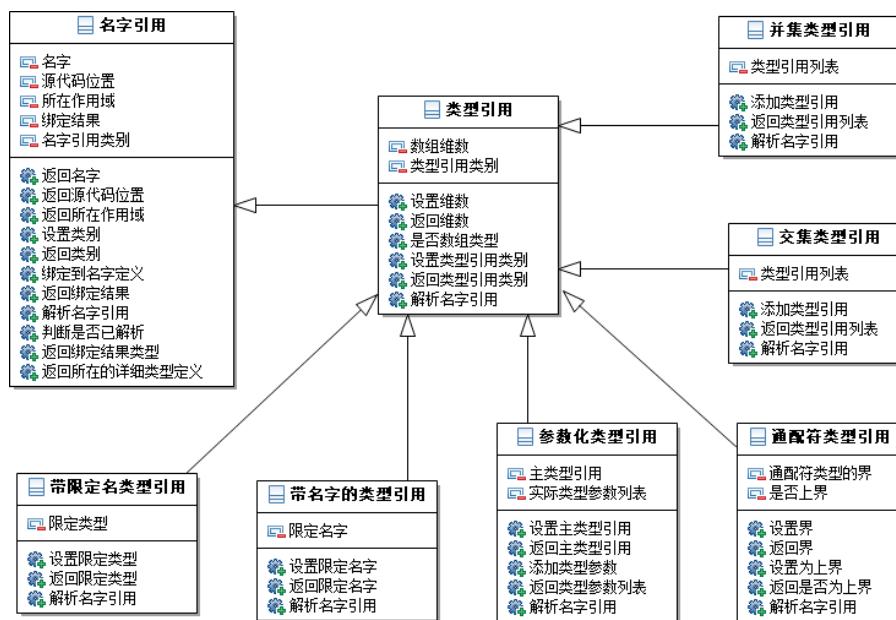


图 4.7 类型引用及其子类

图4.7给出了类型引用及其子类的类图。

4.2.5 类-方法引用

方法引用(MethodReference)是名字引用(NameReference)的直接子类, 对应Java程序中的方法调用(而不是Java语言最新引入的方法引用表达式)。除继承名字引用的名字(方法的简单名)、源

代码位置、所在作用域及绑定结果外，方法引用还有如下属性：

(AT1). 实际参数列表(argumentList): 在调用方法时，圆括号内给出的实际参数列表。实际参数实际上是表达式，因此列表元素是名字引用(NameReference);

(AT2). 实际类型参数列表(typeArgumentList): 在调用类属方法(即带有形式类型参数的方法)时，尖括号内给出的实际类型参数。实际类型参数实际上是类型(表达式)，因此列表元素是类型引用(TypeReference)。

由于Java语言的多态性，如果一个方法引用绑定到(即认为是调用)一个类型的某个方法定义，则它也可能是调用(即应该绑定到)这个类型的所有子类型中重定义这个方法的方法。简单地说，由于多态性，一个方法引用可能可以绑定到多个方法定义，因此方法引用除继承名字引用这个类的属性绑定结果(definition)存放静态绑定到的方法定义外，还有如下属性：

(AT3). 可能绑定的方法定义列表(alternativeList): 列表元素是方法定义，第一个元素与存放在属性绑定结果definition的方法定义相同。

方法定义以名字、源代码位置和所在作用域作为构造方法的参数。除继承名字引用的操作外，还提供如下操作：

(RP1-2): 设置实际参数列表(setArgumentList(List<NameReference>))、返回实际参数列表(getArgumentList());

(RP3-4): 设置实际类型参数列表(setTypeArgumentList(List<TypeReference>))、返回实际类型参数列表(getTypeArgumentList())。

由于通常方法调用的实际参数列表和实际类型参数列表都是在构造方法引用之前就已经设置好，因此不是将实际参数或实际类型参数一个一个添加至列表，而是整体设置。

对于可能绑定的方法定义列表，方法引用提供操作：

(RP5-6): 添加可能绑定的方法定义(addAlternative(MethodDefinition))、返回可能绑定的方法定义列表(getAlternativeList())。

对于单个方法引用本身的解析就是在所在的作用域(应该是详细类型定义)中的方法定义列表匹配方法引用，这是所有名字引用进行名字引用解析的基本方法，所以方法引用无需重定义名字引用的操作解析名字引用(resolveBinding())。但是这个匹配本身除了匹配简单名之外，还要匹配实际参数和实际类型参数(而且在匹配过程中还要判断子类型关系)，但这个匹配职责放在方法定义(MethodDefinition)，而不是在方法引用中。要获得所有可能绑定的方法定义，也需要后代类的一个方法是否是当前已经绑定的方法定义的重定义，这个职责也由方法定义完成。

4.2.6 类—值引用

值引用(ValueReference)是名字引用(NameReference)的直接子类。值引用表示对软件运行时一段内存区域中的数据值进行访问或修改，它可能是使用类型的字段，也可能是使用局部作用域中声明的变量。我们使用名字引用类别(NameReferenceKind)区分这两种情况。

有时我们需要区分是对字段或变量对应的这一段内存区域数据的读取还是修改，当是修改内存区域的数据时，这个值引用放在赋值运算的左边(或相当于放在赋值运算符的左边)，称为左值引用。因此值引用有如下属性：

(AT1) 是否左值引用(isLeftValue): 缺省值是false;

值引用以名字、源代码位置和所在作用域为构造方法的参数。除继承名字引用的操作外，它还有操作：

(RP1-2) 设置为左值引用(`setLeftValue()`)、返回是否左值引用(`isLeftValue()`)。

对于单个值引用本身的解析就是在所在的作用域（通常是详细类型定义或局部作用域）中的相应类别的名字定义（变量定义或字段定义）列表匹配值引用的简单名，这是所有名字引用进行名字引用解析的基本方法，所以值引用无需重定义名字引用的操作解析名字引用(`resolveBinding()`)。

4.2.7 抽象类—名字引用组及其子类

名字引用组(`NameReferenceGroup`)就是将上下文相关的一组名字引用汇集在一起，以便能根据上下文正确地绑定其中的名字引用。名字引用组对应Java语言中的表达式，将一个表达式中出现的所有名字引用按照表达式的语法结构组织在一起。

Java语言有许多表达式，因此也有多种类别的名字引用组，同样引入一个枚举类型名字引用组类别(`NameReferenceGroupKind`)用以区分不同的名字引用组类别。表4.4给出了名字引用组类别所定义的枚举常量。

表 4.4 名字引用组类别

常量	含义	常量	含义
<code>NRGK_ARRAY_ACCESS</code>	数组访问	<code>NRGK_ARRAY_CREATION</code>	数组创建
<code>NRGK_ARRAY_INITIALIZER</code>	数组初始化	<code>NRGK_ASSIGNMENT</code>	赋值表达式
<code>NRGK_CAST</code>	类型转换	<code>NRGK_CLASS_INSTANCE_CREATION</code>	类实例创建
<code>NRGK_CONDITIONAL</code>	条件表达式	<code>NRGK_FIELD_ACCESS</code>	字段访问
<code>NRGK_INFIX_EXPRESSION</code>	中缀表达式	<code>NRGK_INSTANCEOF</code>	实例判断
<code>NRGK_METHOD_INVOCATION</code>	方法调用	<code>NRGK_POSTFIX_EXPRESSION</code>	后缀表达式
<code>NRGK_PREFIX_EXPRESSION</code>	前缀表达式	<code>NRGK_SUPER_FIELD_ACCESS</code>	超类型字段访问
<code>NRGK_SUPER_METHOD_INVOCATION</code>	超类方法调用	<code>NRGK_VARTIALBE_DECLARATION</code>	变量声明表达式
<code>NRGK_THIS_EXPRESSION</code>	<code>this</code> 表达式	<code>NRGK_TYPE_LITERAL</code>	类型常量
<code>NRGK_QUALIFIED_NAME</code>	带限定的名字		

名字引用组(`NameReferenceGroup`)是名字引用(`NameReference`)的直接子类。它的名字是所对应表达式（在源代码中的）字符串。除继承名字引用的属性外，主要还有如下属性：

(AT1). 名字引用组的运算符(`operator`)：是一个字符串，用于保存表达式，特别是算术表达式中的运算符；

(AT2). 子名字引用列表(`subreferenceList`)：给出了该名字引用组所包含的子名字引用列表，其元素也是名字引用，特别地也可以是名字引用组，从而构成对应表达式语法结构的名字引用树。

名字引用组无需设置名字引用组类别这个属性，因为实际上每个具体的名字引用组子类对应一个类别。

名字引用组是一个抽象类，它作为具体名字引用组的共同基类，本身不创建实例。它以名字、源代码位置和所在作用域为构造方法的参数。除继承名字引用的操作外，它还应提供如下操作：

(RP1-2). 设置运算符(`setOperator(String)`)、返回运算符(`getOperator()`)；

- (RP3). 添加子名字引用(`addSubreference(NameReference)`);
- (RP4). 返回子名字引用列表(`getSubreferenceList()`);
- (RP5). 返回名字引用组类别(`getGroupKind()`)。

更多时候我们关心名字引用组中在叶子节点的名字引用，因为只有在叶子节点的名字引用才是最基本的名字引用：

(RP6). 返回叶子名字引用列表(`getReferenceAtLeaf()`)：返回在叶子节点的名字引用构成的列表，列表元素是名字引用(`NameReference`)，但不会是名字引用组的实例。

不同的名字引用组有不同的名字解析方法，因此名字引用组的继承名字引用的操作解析名字引用(`resolveBinding()`)是一个抽象方法。具体的名字引用组仅仅是给出操作解析名字引用(`resolveBinding()`)的具体实现，不增加属性和其他操作。基于领域分析，特别是领域模型中的表3.6下面概要说明名字引用组的直接子类对操作解析名字引用(`resolveBinding()`)的实现。

首先，下面的名字引用组具体子类的操作解析名字引用(`resolveBinding()`)的实现都比较简单，都是先解析子名字引用表中的所有名字引用，然后将整个名字引用组绑定到表达式的结果类型（通常是某个子名字引用的绑定结果类型）：

1. 数组访问(`NRGArrayAccess`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用，然后将整个名字引用绑定到第一个子名字引用的绑定结果类型；
2. 数组创建(`NRGArrayCreation`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用，然后将整个名字引用绑定到第一个子名字引用的绑定结果类型；
3. 数组初始化(`NRGArrayInitializer`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用，然后将整个名字引用绑定到第一个子名字引用的绑定结果类型；
4. 赋值(`NRGAssignment`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用，然后将整个名字引用绑定到第一个子名字引用的绑定结果类型；
5. 类型转换(`NRGCast`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用，然后将整个名字引用绑定到第一个子名字引用的绑定结果类型；
6. 条件表达式(`NRGConditional`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用，然后将整个名字引用绑定到第一个子名字引用的绑定结果类型。整个名字引用组的绑定结果是第二个名字引用的绑定结果类型，除非第二个名字引用的绑定结果类型是第三个名字引用的绑定结果类型的子类型，这时整个名字引用组的绑定结果是第三个名字引用的绑定结果类型。
7. 中缀表达式(`NRGInfixExpression`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用。整个名字引用组的绑定结果是整个中缀表达式的结果类型，在绑定时需要对整个中缀表达式的类型进行推断。
8. 实例判断表达式(`NRGInstanceof`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用。整个名字引用组的绑定结果是`boolean`类型所对应的导入类型定义。
9. 后缀表达式(`NRGPostfixExpression`)：先调用操作解析名字引用(`resolveBinding()`)解析子名字引用列表中的所有名字引用。整个名字引用组的绑定结果是这个后缀表达式的结果类型，在绑定时需要对这个后缀表达式的类型进行推断。
10. 前缀表达式(`NRGPrefixExpression`)：先调用操作解析名字引用(`resolveBinding()`)解析

子名字引用列表中的所有名字引用。整个名字引用组的绑定结果是这个前缀表达式的结果类型，在绑定时需要对这个前缀表达式的类型进行推断。

11. 变量声明(NRGVariableDeclaration): 先调用操作解析名字引用(resolveBinding())解析子名字引用列表中的所有名字引用，然后将整个名字引用绑定到第一个子名字引用的绑定结果类型；

12. 类型常量(NRGTypeLiteral): 调用操作解析名字引用(resolveBinding())解析子名字引用列表中的所有名字引用。整个名字引用组绑定的结果是java.lang.Class这个预定义类所对应的导入类型定义。

剩下的名字引用组都要基于某个子名字引用所确定的作用域来解析另外的名字引用。我们将上面十二个名字引用组称为**子名字引用间无隶属关系的名字引用组**，而将下面的名字引用组称为**子名字引用间有隶属关系的名字引用组**。

13. 类实例创建(NRGClassInstanceCreation): 基本语法形式是：“[表达式.] new [<类型参数>] 类型(实际参数)”。创建的名字引用组至多两个子名字引用。注意，其中的类型参数和实际参数都在第二个对应构造方法的方法引用中。如果有两个子名字引用，则先解析第一个子名字引用，它的结果应该是类型定义，如果是一个详细类型定义，则以该详细类型定义为作用域，在其中再解析第二个对应构造方法的子名字引用。整个名字引用组绑定到第一个名字引用的绑定结果。如果只有一个子名字引用，则在该子名字引用所在的作用域解析它，整个名字引用组绑定到这个名字引用的绑定结果类型。

14. 字段访问(NRGFieldAccess): 基本语法形式是“表达式.标识符”。创建的名字引用组有两个子名字引用。第一个名字引用对应句点前的表达式，而第二个名字引用是一个值引用。先解析第一个名字引用，它的结果应该是类型定义，如果是一个详细类型定义，则以该详细类型定义为作用域，在其中再解析第二个对应字段名的值引用。整个名字引用的绑定结果是该值引用的绑定结果类型（对应声明该字段的类型）。

15. 方法调用(NRGMethodInvocation): 基本语法形式是：“[表达式.] [<类型参数>] 标识符(实际参数)”。创建的名字引用组至多两个子名字引用。注意，其中的类型参数和实际参数都在第二个方法引用中。如果有两个子名字引用，则先解析第一个子名字引用，它的结果应该是类型定义，如果是一个详细类型定义，则以该详细类型定义为作用域，在其中再解析第二个方法引用。如果只有一个子名字引用，则就是方法引用，那么在该方法引用所在的作用域解析它。整个名字引用组总是绑定到方法引用的绑定结果类型（对应所调用方法的返回类型）。

16. 超类型字段访问(NRGSuperFieldAccess): 语法形式是“[类名.]super.标识符”。创建的名字引用组至多有两个子名字引用，其中对应标识符的是一个值应用。如果有两个名字引用，则先解析第一个名字引用，它的结果应该是类型定义，如果是一个详细类型定义，则获取这个详细类型的超类，如果它的超类也是详细类型定义，则以该详细类型定义为作用域解析第二个对应字段的值引用。如果只有一个子名字引用，则就是值引用，那么获取包含该值引用的详细类型定义，然后获取该详细类型定义的超类，如果它的超类也是详细类型定义，则以该详细类型定义为作用域解析这个值引用。整个名字引用组绑定的结果总是其中值引用的绑定结果类型（对应声明该字段的类型）。

17. 超类型方法调用(NRGSuperMethodInvocation): 语法形式是“[类名.]super.[<实际类型参数>] 标识符([实际参数])”。创建的名字引用组至多两个子名字引用。注意，其中的类型参数和实际参数都在第二个方法引用中。如果有两个子名字引用，则先解析第一个子名字引用，它的结果

应该是类型定义，如果是一个详细类型定义，则获取这个详细类型的超类，如果它的超类也是详细类型定义，则以该详细类型定义为作用域，在其中再解析第二个方法引用。如果只有一个子名字引用，则就是方法引用，那么获取包含该方法引用的详细类型定义，然后获取该详细类型定义的超类，如果它的超类也是详细类型定义，则以该详细类型定义为作用域解析这个方法引用。整个名字引用组总是绑定到方法引用的绑定结果类型（对应所调用方法的返回类型）。

18. This表达式(NRGThisExpression): 语法形式是“[类名.]this”。创建的名字引用组至多两个子名字引用，一个是类型引用，对应类名，一个是文字常量引用(LiteralReference)，对应关键字this。如果有两个子名字引用，则先解析第一个子名字引用，它的结果应该是类型定义，将第二个文字常量引用也绑定到这个类型定义，整个名字引用组也绑定到这个类型定义。如果只有一个名字引用，则是对应this的文字引用，获取包含该文字引用的详细类型定义，将这个文字引用绑定到该详细类型定义，将整个名字引用组也绑定到这个详细类型定义。

19. 带限定的名字(QualifiedName): 语法形式是“(带限定的名字 | 标识符).标识符”，也就是用句点分隔的两个或多个标识符。创建的名字引用组有两个子名字引用，第一个名字引用对应最后一个句点之前的限定名（可能是只有一个标识符，这时创建的是值引用，或者还是用句点分隔的两个或多个标识符，这时又是一个带限定的名字引用组），而第二个名字引用对应句点之后的标识符，是一个值引用。这时有几种可能的情况：

(1). 第一个名字引用引用的是一个程序包，整个带限定的名字是引用该程序包的一个类型（第二个名字引用是引用该类型的简单名）；

(2). 第一个名字引用引用的是一个类型，第二个名字引用引用的也是类型，整个带限定的名字是使用一个内部类；

(3). 第一个名字引用引用的是一个类型，第二个名字引用引用的是字段，整个带限定的名字是访问一个类的静态字段；

(4). 第一个名字引用引用的是一个对象，第二个名字引用引用的是字段，整个带限定的名字是访问一个对象的字段。

我们对这几种情况逐一进行尝试，其中上面第(2)、(3)两种情况可以合并成一种情况尝试。注意，整个名字引用组的绑定结果总是第二个名字引用的绑定结果类型（如果成功的话）：

(1) 首先将第一个名字引用的类别人为设置为程序包引用，在系统作用域内进行解析，如果绑定成功，则在绑定的结果（即，一个程序包定义）中解析第二个名字引用，如果绑定成功，则整个名字引用组解析完毕，否则恢复第一个名字引用的类别，并做下一尝试；

(2) 如果第一个名字引用不是名字引用组，则将其人为设置为类型引用(NRK_TYPE)，否则它仍是一个带限定的名字。解析第一个名字引用（如果是带限定的名字则会递归调用），如果绑定成功，结果应该是一个类型定义，如果是详细类型定义，则在该详细类型定义中解析第二个名字引用，如果绑定成功，则整个名字引用组解析完毕，否则恢复第一个名字引用的类别，并做下一尝试；

(3) 解析第一个名字引用（如果是带限定的名字则会递归调用，否则就是一个值引用），如果绑定成功，结果应该是一个类型定义，如果是详细类型定义，则在该详细类型定义中解析第二个名字引用，如果绑定成功，则整个名字引用组解析完毕，否则整个名字引用组的解析失败。

图4.8给出了方法引用、值引用和名字引用组的类图。名字引用组还有19个直接子类，但是这些子类没有属性，而只是重定义解析名字引用(resolveBinding())这个操作，因此不再给出。

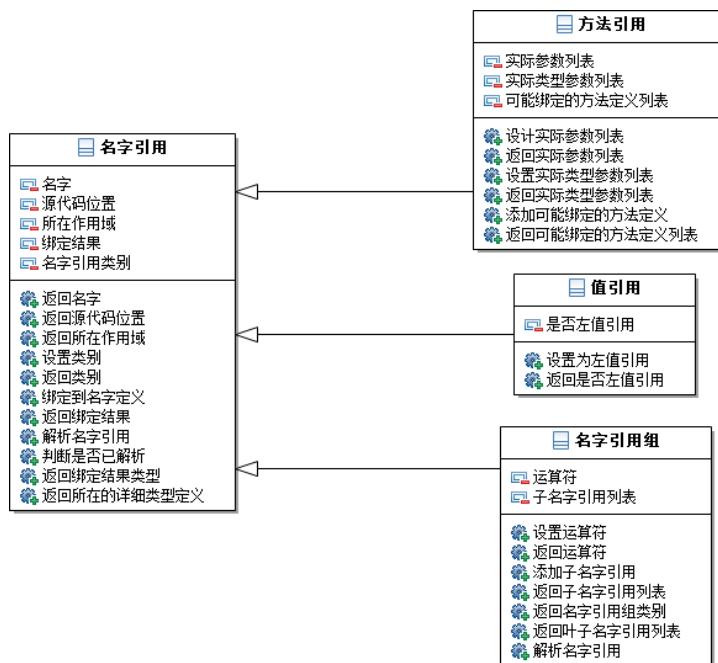


图 4.8 方法引用、值引用和名字引用组

4.2.8 名字引用解析过程

对名字引用进行解析是生成名字表的主要目的，而且名字引用解析过程比较复杂，这里对这个过程进行总结。如前所述，名字引用解析过程概括地说就是在合适的作用域里匹配对应种类的名字定义。所以名字引用解析过程主要分为两步：(1). 确定匹配名字引用的合适作用域；(2). 在作用域中匹配对应的名字定义。

1. 对于确定匹配名字引用的合适作用域，这根据名字引用的不同种类分为三种情况：
 - (1). 对于基本的名字引用（即非名字引用组），合适的作用域就是它所在的作用域；
 - (2). 对于子名字引用间无隶属关系的名字引用组，各子名字引用分别解析，然后根据各子名字引用的绑定结果类型进行类型推导，整个名字引用组绑定到类型推导所确定的类型定义；
 - (3). 对于子名字引用间有隶属关系的名字引用组，先解析其中用于确定作用域的子名字引用（通常是第一个），再在该子名字引用的绑定结果类型（通常是详细类型定义）中解析剩下的子名字引用（通常也只有一个），整个名字引用组绑定到剩下的那个子名字引用的绑定结果类型。

由于不同的名字引用（的子类）可能重定义解析名字引用(`resolveBinding()`)这个操作，因此实际上是通过Java语言的动态绑定机制实现上述不同情况的处理。

名字引用间无隶属关系的名字引用组通常是一些运算表达式（赋值、条件、前缀、后缀、中缀等），子名字引用是子表达式，整个名字引用组的绑定结果应该是整个运算表达式的结果类型（所对应的类型定义），但各个结果子表达式的结果类型可能不同，因此需要进行类型推导以确定整个表达式的结果类型。这个类型推导，简单地说：

- (1). 对于基本数据类型，都转换到取值范围最宽的数据类型，或根据具体的运算确定结果类型（例如关系运算的结果都是`boolean`类型，有`String`类型参与的加运算结果类型是`String`等）；
- (2). 对于非基本数据类型，各子表达式的结果类型之间应该有子类型关系，整个表达式结果类型是各个子表达式结果类型的共同超类型。

2. 对于在作用域中匹配对应的名字定义:

(1). 根据名字引用的类别在合适的名字定义中进行匹配, 如果匹配成功则绑定到该名字定义, 否则进行下一步;

(2). 如果当前作用域是详细类型定义, 那么在它的是引用详细类型定义的超类型(包括类和接口)中再尝试匹配, 如果在某个超类型中匹配成功, 则解析结束, 否则进行下一步;

(3). 在包围当前作用域的父作用域再尝试匹配, 如果匹配成功, 则解析结束, 否则绑定不成功。

在作用域中匹配对应的名字定义是利用名字作用域的在当前作用域中解析名字引用(`resolve()`)这个操作, 这个操作最终利用名字定义的匹配名字引用(`match(NameReference)`)这个操作进行匹配, 不过对于方法引用, 不仅仅是利用这个操作匹配, 关键是需要利用方法定义的匹配方法引用(`matchMethod(MethodReference)`)这个操作进行方法参数和类型参数的匹配, 并且在匹配成功之后需要找到后代类的重定义方法作为该方法引用的可能绑定结果。

表 4.5 名字引用的解析过程

【起点】	调用名字引用的解析名字引用(<code>resolveBinding()</code>)操作
【结果】	设置名字引用的属性绑定结果(<code>definition</code>)
【前置条件】	已生成名字定义和名字引用域, 并绑定了导入声明
【主要步骤】	<p>1. 动态调用不同名字引用的解析名字引用(<code>resolveBinding()</code>)操作</p> <p>1.1 对于非名字引用组: 调用所在作用域的解析名字引用(<code>resolve()</code>)操作</p> <p>1.2 对于子名字引用之间无隶属关系的名字引用组: 1.2.1 调用各子名字引用的解析名字引用(<code>resolveBinding()</code>)操作 1.2.2 根据各子名字引用的绑定结果类型进行类型推导 1.2.3 根据类型推导的结果绑定整个名字引用组</p> <p>1.3 对于子名字引用之间有隶属关系的名字引用组: 1.3.1 解析确定作用域的子名字引用 1.3.2 该子名字引用的绑定结果类型中解析另一个子名字引用 1.3.3 整个名字引用组的绑定到后一个名字引用的绑定结果类型</p> <p>2. 使用作用域的解析名字引用(<code>resolve()</code>)操作在作用域中匹配名字引用</p> <p>2.1 根据名字引用的类别在合适的名字定义中进行匹配, 成功则解析结束</p> <p>2.2 如果当前作用域是详细类型定义 2.2.1 在该详细类型定义的超类型中匹配名字引用, 成功则解析结束</p> <p>2.3 在包围当前作用域的父作用域中匹配名字引用</p>

表4.5简要总结了名字引用的解析过程。最后, 对于名字引用的解析过程还要说明两点:

1. 为避免循环解析(与匹配), 一旦确定在作用域中匹配名字引用与名字定义(即进入当前作用域中解析名字引用(`resolve()`)这个操作), 就不再调用名字引用的解析名字引用(`resolveBinding()`)这个操作;

2. 上述名字引用解析过程假定已经生成名字定义和名字作用域, 并对编译单元中的导入声明(所创建的名字引用)已经绑定, 因此没有考虑源代码可能存在语法错误的可能性。如果名字引用不能绑定成功, 其原因不是源代码存有语法错误, 而是该名字引用引用的程序实体没有源代码(而是第三方构件库或Java平台的构件库提供的程序实体)。

4.2.9 名字引用相关类设计小结

1. 图4.6、4.7和图4.8给出了名字引用相关类的类图。表4.5给出名字引用解析过程的总结。
2. 名字引用(NameReference)的直接子类包括：
 - (1) 程序包引用(PackageReference);
 - (2) 文字引用(LiteralReference);
 - (3) 类型引用(TypeReference);
 - (4) 方法引用(MethodReference);
 - (5) 值引用(ValueReference);
 - (6) 名字引用组(NameReferenceGroup)。
3. 类型引用(TypeReference)的直接子类又包括：
 - (1) 带限定名类型引用(QualifiedTypeReference);
 - (2) 带名字的类型引用(NamedTypeReference);
 - (3) 参数化类型引用(ParameterizedTypeReference);
 - (4) 通配符类型引用(WildcardTypeReference);
 - (5) 交集类型引用(IntersectionTypeReference);
 - (6) 并集类型引用(UnionTypeReference)。
4. 名字引用组(NameReferenceGroup)是一个抽象类，它有19个直接子类，这里不再一一给出。

4.3 名字表使用

名字表的使用主要是查找、遍历指定条件的名字表名字定义和名字作用域，并访问它的属性。访问某个名字定义和作用域的属性通过名字定义和名字作用域相关的类实现，而名字表定义和名字作用域的查找与遍历通过名字表管理器(NameTableManager)实现。

虽然在名字表生成时，我们支持同时生成名字定义、名字作用域和名字引用，但经过一段时间的实践，发现对名字表的访问，主要是对名字定义和名字作用域的访问，而名字引用在稍微大型的程序中的非常多，因此通常都只能是临时生成即时使用，所以我们不单独设置名字引用使用相关的类，而将名字引用的使用与名字引用的生成一起考虑。

名字表管理器的对象实例通过名字表生成器(NameTableCreator)或它的子类名字定义生成器(NameDefinitionCreator)创建，作为访问名字表的入口。名字定义按照作用域组织，作用域按照嵌套(包围)关系组织成树结构，因此查找和遍历名字定义和作用域都是遍历这个树结构，找到满足定义的名字表实体。

名字表使用的基本思路是使用访问者(visitor)模式，并在访问的过程中使用过滤器(filter)过滤满足条件的名字表实体。名字表访问器(NameTableVisitor)或其子类完成对作用域树结构的遍历，获取其中的名字表实体。名字表过滤器(NameTableFilter)或其子类完成名字表实体的过滤，使得遍历的结果可返回满足条件的名字表实体。

名字表通过遍历抽象语法树而生成，因此有时构件使用需要了解名字表与抽象语法树之间的对应管理，我们设计类名字表抽象语法树桥接器(NameTableASTBridge)完成名字表与抽象语法树之间对应关系的管理。

4.3.1 类-名字表访问器及其子类

我们参考Eclipse JDT的类`ASTVisitor`设计名字表访问器(`NameTableVisitor`)，它没有属性，只是提供一系列的访问方法：

(RP1) 进入作用域前的访问(`preVisit()`)：子类可重定义这个方法以给出所有作用域在调用访问`visit()`方法之前的相同的准备工作；

(RP2) 访问(系统作用域)(`visit(SystemScope)`)：对名字表的访问实际上对作用域的遍历，因此针对每种作用域(每个实现接口名字作用域(`NameScope`)的类)有一个访问方法`visit()`，这个方法返回`true`表示要继续访问子作用域，返回`false`表示不用继续访问子作用域；

(RP3). 访问(程序包定义)(`visit(PackageDefinition)`)；

(RP4). 访问(编译单元作用域)(`visit(CompilationUnitScope)`)；

(RP5). 访问(详细类型定义)(`visit(DetailedTypeDefinition)`)；

(RP6). 访问(方法定义)(`visit(MethodDefinition)`)；

(RP7). 访问(局部作用域)(`visit(LocalScope)`)；

(RP8-13) 遍历子作用域后的访问(`endVisit()`)：每种作用域都有`endVisit()`方法，这个方法在遍历当前作用域的子作用域之后再对当前作用域做收尾工作；

(RP14) 退出作用域时的访问(`postVisit()`)：子类可重定义这个方法以给出所有作用域在调用方法`endVisit()`之后的共同的收尾工作。

类名字表访问器(`NameTableVisitor`)是一个基类，所有对名字表的实际进行访问的类可重定义这个类的方法，这个类本身所有的方法都不做任何实际工作。我们设计它的几个子类来分别访问名字表中的名字定义和名字作用域。

类名字定义访问器(`NameDefinitionVisitor`)是名字表访问器(`NameTableVisitor`)的直接子类，重定义其中的6个访问不同种类作用域的访问(`visit()`)方法，以获取各名字作用域中的名字定义，并且使用名字表过滤器对名字定义进行过滤，以返回满足条件的名字定义，因此它的主要属性有：

(AT1) 名字定义过滤器(`filter`)：是类名字表过滤器(`NameDefinitionFilter`)的实例；

(AT2) 遍历结果(`result`)：是一个名字定义列表，元素是名字定义`NameDefinition`。

名字定义访问器(`NameDefinitionVisitor`)的主要职责除重定义名字表访问器(`NameTableVisitor`)的访问`visit()`方法外，还提供：

(RP1) 设置名字表过滤器(`setFilter()`)；

(RP2) 返回遍历结果(`getResults()`)：返回遍历结果列表，元素是名字定义(`NameDefinition`)；

(RP3) 将遍历结果作为树集返回(`getResultsAsTreeSet()`)：通常返回的名字定义列表可能还需要做被多次查找，列表的查找效率比较低，可能需要使用查找树存储这些名字定义，因此应提供这个方法。

类名字作用域访问器(`NameScopeVisitor`)是名字表访问器(`NameTableVisitor`)的直接子类，重定义其中的6个访问不同种类作用域的访问(`visit()`)方法，以获取作用域本身，并且使用名字表过滤器对名字作用域进行过滤，以返回满足条件的名字作用域，因此它的设计与名字定义访问器类似，主要属性有：

(AT1) 名字作用域过滤器(`filter`)：是类名字作用域过滤器(`NameScopeFilter`)的实例；

(AT2) 遍历结果(`result`)：是一个名字作用域列表，元素是名字定义`NameScope`。

名字作用域访问器(`NameScopeVisitor`)的主要职责除重定义名字表访问器(`NameTableVisitor`)的访问`visit()`方法外，还提供：

- (RP1) 设置名字作用域过滤器(`setFilter()`);
- (RP2) 返回遍历结果(`getResult()`)：返回遍历结果的名字作用域列表。

基于前面所说的原因，我们暂时不考虑使用名字引用访问器(`NameReferenceVisitor`)支持通过遍历作用域访问名字引用，而是建议在生成名字引用时即时处理（访问）名字引用。

4.3.2 类—名字定义过滤器和名字作用域过滤器

类名字定义过滤器(`NameDefinitionFilter`)用于设置过滤名字定义的条件，为支持多种条件的组合，我们为名字定义过滤器使用包装模式(decorate pattern)，即每个过滤器（的实例）还包含另外一个过滤器，从而形成一个过滤器链实现多个条件的组合过滤。因此名字定义过滤器的属性有：

(AT1). 包装的过滤器(`wrappedFilter`)：先使用包装的过滤器过滤，然后再使用当前过滤器进行过滤，形成过滤器链；

名字定义过滤器使用包装的过滤器为参数的构造方法，它只提供一个方法：

(RP1). 接受名字定义(`accept(NameDefinition)`)：给出是否接受名字定义的具体条件，如果接受则返回`true`，否则返回`false`。缺省的实现是返回包装的过滤器的`accept()`方法的返回结果，后代类需重定义这个方法以给出具体过滤条件。

最常用的名字定义访问是详细类型定义（不包括枚举类型定义）的访问，因此缺省提供定义类详细类型定义过滤器(`DetailedTypeDefinitionFilter`)，它提供两个构造方法，一个构造方法以一个包装的过滤器和一个字符串为参数，另外一个只以一个包装的过滤器为参数。它除继承包装的过滤器(`wrapperFilter`)之外，还设置属性：

(AT1) 名字串(`name`)：用以过滤简单名是否为该名字串的详细类型定义，因为很多时候我们可能要过滤详细类型定义的简单名。如果它为`null`，则对详细类型定义的简单名没有限制。

详细类型定义过滤器(`DetailedTypeDefinitionFilter`)重定义接受名字定义方法`accept()`，以只接受详细类型定义（不包括枚举类型定义），并在需要时过滤详细类型定义的简单名。

类名字作用域过滤器(`NameScopeFilter`)的设计与类名字定义过滤器类似，它的属性有：(AT1)。

(AT1). 包装的过滤器(`wrappedFilter`)：先使用包装的过滤器过滤，然后再使用当前过滤器进行过滤，形成过滤器链；

(AT2). 名字串(`name`)：用以过滤名字是否为该名字串的作用域，如果为`null`，则对作用域名字没有限制。

名字作用域过滤器提供两个构造方法，一个构造方法以一个包装的过滤器和一个字符串为参数，另外一个只以一个包装的过滤器为参数。它只提供一个方法：

(RP1). 接受名字作用域(`accept(NameDefinition)`)：给出是否接受名字作用域定义的具体条件。缺省的实现是先调用包装的过滤器的`accept()`方法，如果返回`true`，则在需要时判断要接受的作用域名字是否是指定名字串。后代类可重定义这个方法以给出更多过滤条件。

由于有些名字定义（如详细类型定义）也是名字作用域，对于这些名字定义，我们推荐使用名字定义过滤器，因为通过名字定义可访问更多信息。由于有些名字定义也是名字作用域，因此我们无法为名字定义过滤器和名字引用过滤器设置一个共同的基类，因为签

名(signature)为`accept(NameScope)`和`accept(NameDefinition)`的两个方法不能放在同一个类, 否则以诸如一个详细类型定义(DetailedTypeDefinition)为参数调用时会发生方法调用二义性!

4.3.3 类-名字表管理器

类名字表管理器(NameTableManager)可看做是访问名字表的入口, 实际上是保存源代码文件集及其名字表的作用域树结构的根节点—系统作用域, 从系统作用域开始, 使用名字表访问器(NameTableVisitor)可遍历名字表的所有实体。因此名字表管理器的主要属性是:

(AT1). 源代码文件集(codeFileSet): 类`SourceCodeFileSet`的实例, 管理的是该源代码文件集的名字表;

(AT2). 名字表作用域树结构根节点(systemScope): 是一个系统作用域(SystemScope)的实例。

名字表管理器以源代码文件集和系统作用域为构造方法的参数。名字表管理器的主要职责是提供对名字实体的常规访问:

(RP1). 返回所有程序包定义列表(`getAllPackageDefinitions()`);

(RP2). 返回所有编译单元作用域列表(`getAllCompilationUnitScopes()`);

(RP3). 返回所有详细类型定义列表(`getAllDetailedTypes()`);

(RP4). 查找给定简单名的首个名字定义(`findDefinitionOfSimpleName(String)`);

(RP5). 返回给定简单名的所有名字定义(`getAllDefinitionsOfSimpleName(String)`);

(RP6). 查找给定全限定名的名字定义(`findDefinitionOfFullQualifiedName(String)`);

(RP7). 返回给定位置间所有名字定义(`getAllDefinitionsBetweenLocations(SourceCodeLocation, SourceCodeLocation)`): 两个源代码位置(SourceCodeLocation)分别给出起始位置和终止位置, 返回这两个位置之间的所有名字定义;

(RP8). 返回给定作用域的所有名字定义(`getAllDefinitionsOfScope(NameScope)`);

(RP9). 查找过滤器接受的首个名字定义(`findDefinitionByFilter(NameDefinitionFilter)`);

(RP10). 返回过滤器接受的所有名字定义(`getAllDefinitionsByFilter(NameDefinitionFilter)`);

(RP11). 返回指定名字的作用域(`getScopeOfName(String)`);

(RP12). 返回包含源代码位置的最内层作用域(`getScopeOfLocation(SourceCodeLocation)`);

(RP13). 返回包含名字定义的方法定义(`getEnclosingMethodDefinition(NameDefinition)`);

(RP14). 返回包含名字定义的类型定义(`getEnclosingDetailedTypeDefinition(NameDefinition)`);

(RP15). 返回包含名字定义的编译单元(`getEnclosingCompilationUnitScope(NameDefinition)`);

(RP16). 返回包含名字引用的方法定义(`getEnclosingMethodDefinition(NameReference)`);

(RP17). 返回包含名字引用的类型定义(`getEnclosingDetailedTypeDefinition(NameReference)`);

(RP18). 返回包含名字引用的编译单元(`getEnclosingCompilationUnitScope(NameReference)`);

(RP19). 返回作用域过滤器接受的所有作用域(`getAllScopesByFilter(NameScopeFilter)`)。

(RP20). 接受名字访问器(`accept(NameTableVisitor)`): 接受该名字访问器, 也即使用该名字访问器对名字表进行访问, 访问结果可通过该名字访问器获得, 从而完成由用户自定义的名字表实体访问;

(RP21). 返回源代码文件集(`getSourceCodeFileSet()`);

(RP22). 返回系统作用域(`getSystemScope()`)。

上面主要是访问名字定义和名字作用域的方法。实际上，构件使用可利用接受名字访问器(`accept()`)方法实现对名字定义、名字作用域的访问，名字表管理器只是利用访问者缺省地实现了一些比较常用的访问名字定义、名字作用域的方法，而名字引用的访问使用名字引用生成器临时生成并即时处理。

4.3.4 类—名字表抽象语法树桥接器

类名字表抽象语法树桥接器(`NameTableASTBridge`)完成名字表与抽象语法树之间关联的一些功能，因此它有属性：

(AT1). 名字表管理器(`tableManager`): 类名字表管理器(`NameTableManager`)的对象实例。

类名字表抽象语法树桥接器以名字表管理器为构造方法的参数，这个类的主要职责有：

(RP1). 查找编译单元作用域对应的AST节点(`findASTNodeForCompilationUnitScope()`): 方法的参数是编译单元作用域(`CompilationUnitScope`)，返回的AST节点类型是编译单元(`CompilationUnit`)；

(RP2). 查找详细类型定义对应的AST节点(`findASTNodeForDetailedTypeDefinition()`): 方法的参数是详细类型定义(`DetailedTypeDefinition`)，返回的AST节点是类型声明(`TypeDeclaration`)；

(RP3). 查找方法定义对应的AST节点(`findASTNodeForMethodDefinition()`): 方法的参数是方法定义(`MethodDefinition`)，返回的AST节点类型是方法声明(`MethodDeclaration`)；

(RP4). 查找字段定义对应的AST节点(`findASTNodeForFieldDefinition()`): 方法的参数是字段定义(`FieldDefinition`)，返回的AST节点类型是字段声明(`FieldDeclaration`)；

(RP5). 查找类型声明对应的详细类型定义(`findDefinitionForTypeDeclaration()`): 方法的参数是类型为类型声明(`TypeDeclaration`)的AST节点，以及它所在的编译单元名，返回名字表中对应的详细类型定义(`DetailedTypeDefinition`)；

(RP6). 查找方法声明对应的方法定义(`findDefinitionForMethodDeclaration()`): 方法的参数是类型为方法声明(`MethodDeclaration`)的AST节点，以及它所在的编译单元名，返回名字表中对应的方法定义(`DetailedTypeDefinition`)；

(RP7). 查找字段声明对应的字段定义(`findDefinitionForFieldDeclaration()`): 方法的参数是类型为字段声明(`FieldDeclaration`)的AST节点，以及它所在的编译单元名，返回名字表中对应的字段定义(`DetailedTypeDefinition`)。

(RP8). 返回AST节点包含的所有名字定义(`getAllDefinitionsInASTNode()`): 方法的参数是AST节点，以及它所在的编译单元名，返回该AST节点所包含的所有名字定义列表；

(RP9). 生成AST表达式节点对应的名字引用组(`createAllReferencesInExpressionASTNode()`): 方法的参数是类型为表达式(`Expression`)的节点，以及它所在的编译单元名，返回该AST节点所包含的所有名字引用列表。

注意，由于一个编译单元的抽象语法树根节点并没有包含该根节点所对应的编译单元文件名信息，因此从AST节点查找对应的名字定义，生成名字定义或名字引用的方法都必须含有编译单元名这个参数，以便我们确定AST节点的源代码位置。名字表实体与AST 节点之间的对应关系需要依赖源代码位置来确定。从名字表实体查找AST节点则无需提供编译单元名，因为从名字表实体可以查询到包含它的编译单元作用域，从而确定编译单元名及相应的AST根节点。

图4.9给出了类名字表使用相关的类。名字表管理器使用名字表访问器和名字定义过滤器、名字作用域过滤器访问名字定义和名字作用域。名字定义访问器和名字作用域访问器都是名字表访问器

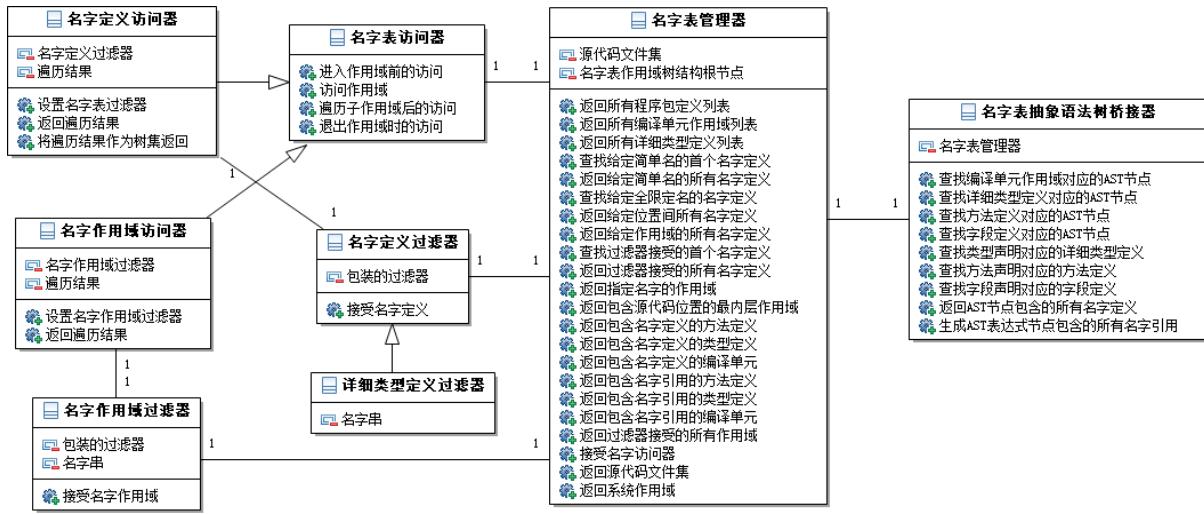


图 4.9 名字表使用相关的类

的直接子类，而详细类型定义过滤器是名字定义过滤器的直接子类。名字定义访问器使用名字定义过滤器，名字作用域访问器使用名字作用域过滤器。名字表抽象语法树桥接器基于名字表管理器管理名字表实体与AST节点之间的对应关系。

4.4 名字表生成

名字表的生成是通过遍历源代码文件集的每个源代码文件的抽象语法树而完成。我们设计两种生成名字表的方式：一种是同时生成所有的名字定义、名字作用域和名字引用；一种是只生成名字定义和名字作用域，而名字引用在需要时生成。由于目前生成全部在内存中完成，因此前一种生成方式只适合小型程序，大型程序的名字引用数量过于庞大，可能无法全部在内存中生成。

类NameTableCreator同时生成名字定义、作用域和名字引用，而类NameDefinitionCreator只生成名字定义和名字作用域，以及一些在名字定义中必须的名字引用。在详细讨论这些类的属性和职责之前，需要先讨论Java语言的语法结构，名字定义、作用域和名字引用可能对应Eclipse JDT生成的抽象语法树哪些节点，只有这样才能真正了解如何生成名字表。

4.4.1 Java程序语法结构

表4.6给出了Java程序的基本语法结构，圆括号内给出的名字是对应的抽象语法树节点类名（都是`ASTNode`的子类）。一个Java软件（对应我们说的源代码文件集）有多个Java编译单元（即源代码文件），Eclipse JDT为每个编译单元生成一棵抽象语法树(Abstract Syntax Tree, AST)，以编译单元节点(`CompilationUnit`)为根。

一个编译单元内有程序包声明(PackageDeclaration)、导入声明(ImportDeclaration)、类型声明(TypeDeclaration)（包括类、接口）、枚举声明(EnumDeclaration)（虽然是类的一种，但更强调其中的枚举常量声明）。

一个类型声明由类型参数(TypeParameter)、超类型声明(Type)(超类或实现的接口)和类型体(BodyDeclaration)。类型体内有初始化块(initializer)、字段声明(FieldDeclaration)、方法

表 4.6 Java 程序的基本语法结构

AST 节点类型	主要组成成分
编译单元(CompilationUnit)	程序包声明(PackageDeclaration) 导入声明(ImportDeclaration) 类型声明(TypeDeclaration) 枚举声明(EnumDeclaration) 标注类型声明(AnnotationTypeDeclaration)
类型声明(TypeDeclaration)	类型参数(TypeParameter) 超类型声明(Type) 类型体(BodyDeclaration)
类型体(BodyDeclaration)	初始化块(Initializer) 字段声明(FieldDeclaration) 方法声明(MethodDeclaration)
枚举声明(EnumDeclaration)	枚举常量声明(EnumConstantDeclaration)
字段声明(FieldDeclaration)	变量声明(VariableDeclaration)
方法声明(MethodDeclaration)	类型参数(TypeParameter) 返回类型(Type) 参数声明(SingleVariableDeclaration) 方法体(Block)

声明(MethodDeclaration)。特别地，枚举声明内有枚举常量声明(EnumConstantDeclaration)。方法声明有类型参数(TypeParameter)、返回类型(Type)、参数声明(SingleVariableDeclaration)和方法体(Block)。方法体实际上就是语句块，由零个或多个语句构成。

可以注意到，字段声明(FieldDeclaration)中的主要组成也是变量声明(VariableDeclaration)，而方法参数声明也是单个变量声明(SingleVariableDeclaration)。

对照前面的领域模型和名字定义、名字作用域和名字引用相关类的设计，可以看到，我们为这些AST节点都定义了相应的名字定义、作用域或名字引用。表4.7给出了这些语法结构所对应的名字表实体。

可以看到，多数语法结构对应的是名字定义，但也有些，如超类型声明、返回类型等实际上是类型引用，而编译单元对应编译单元作用域、初始化块和方法体对应局部作用域。对于导入声明/静态成员导入声明，我们先为之创建类型引用/名字引用，然后绑定导入类型定义(或详细类型定义)/静态成员定义。

其余的抽象语法树节点除标注类型和Javadoc注释外，主要的节点可分为三类：类型(Type)、语句(Statement)和表达式(Expression)。这里的AST节点类型(Type)实际上是对类型的引用，即我们所说的类型引用。类型、语句和表达式是Java程序中随处可见的语法元素。

表4.8给出了表示类型的AST节点类，它们都是类型(Type)节点的子类，表示Java程序中对类型的引用，因此我们为每种AST节点设计了对应的类型引用(TypeReference)，表4.8 的第三列给出了在名字表中每种AST节点对应的类型引用，注意，原子类型和简单类型我们都直接使用类TypeReference 表示。

表4.9给出了表示语句的AST节点类，它们都是AST节点类语句(Statement)的子类。与生成名

表 4.7 主要语法结构对应的名字定义、名字作用域和名字引用类

AST 节点类型	对应的名字表实体
编译单元(CompilationUnit)	编译单元作用域(CompilationUnitScope)
程序包声明(PackageDeclaration)	程序包定义(PackageDefinition)
类型导入声明(ImportDeclaration)	类型引用(TypeReference)
静态成员导入声明(ImportDeclaration)	名字引用(NameReference)
类型声明(TypeDeclaration)	详细类型定义(DetailedTypeDefinition)
枚举声明(EnumDeclaration)	枚举类型定义(EnumTypeDefinition)
类型参数(TypeParameter)	类型参数定义(TypeParameterDefinition)
超类型声明(Type)	类型引用(TypeReference)
初始化块(Initializer)	局部作用域(LocalScope)
字段声明(FieldDeclaration)	字段定义(FieldDefinition)
方法声明(MethodDeclaration)	方法定义(MethodDefinition)
枚举常量声明(EnumConstantDeclaration)	枚举常量定义(EnumConstantDefinition)
变量声明(VariableDeclaration)	变量定义(VariableDefinition)
返回类型(Type)	类型引用(TypeReference)
参数声明(SingleVariableDeclaration)	变量定义(VariableDefinition)
方法体(Block)	局部作用域(LocalScope)

表 4.8 表示引用类型的AST节点类

AST 节点类型	含义	类型引用
ArrayType	数组类型, 如int[]	
PrimitiveType	原子类型, 包括boolean, byte, char, int, short, long, double, float, void	TypeReference
NamedQualifiedType	带名字的类型, 如java.lang.String	NamedTypeReference
QualifiedType	带限定的类型, 如java.lang.String	QualifiedTypeReference
SimpleType	简单类型, 如String	TypeReference
WildcardType	通配符类型, 如<? extend Student>	WildcardTypeReference
ParameterizedType	参数化类型, 如List<String>	ParameterizedTypeReference
IntersectionType	交集类型, 如T & S	IntersectionTypeReference
UnionType	并集类型, 如T S	UnionTypeReference

表 4.9 表示语句的AST节点类

语句种类	AST 节点类型
基本语句	AssertStatement, ConstructorInvocation, EmptyStatement, ExpressionStatement, SuperConstructorInvocation, SynchronizeStatement, TypeDeclarationStatement, VariableDeclarationStatement
块语句	Block
分支语句	IfStatement, SwitchCase, SwitchStatement
循环语句	DoStatement, EnhancedForStatement, ForStatement, WhileStatement
控制转移	BreakStatement, ContinueStatement, LabelStatement, ReturnStatement
异常处理	ThrowStatement, TryStatement

字表有关的节点主要有：

- (1) 构造方法调用(ConstructorInvocation)语句，是使用关键字this调用当前类的（其他）构造方法；
- (2) 超类构造方法调用(SuperConstructorInvocation)语句，是使用关键字super调用超类的构造方法；
- (3) 类型声明语句(TypeDeclarationStatement)语句，通常是声明局部类型；
- (4) 变量声明语句(VariableDeclarationStatement)语句，声明变量。

表 4.10 表示Java表达式的AST节点类

表达式种类	AST 节点类型
名字	Name, 子类有SimpleName和QualifiedName
文字	BooleanLiteral, CharacterLiteral, NumberLiteral, StringLiteral, NullLiteral, TypeLiteral
基本表达式	Assignment, CastExpression, ConditionExpression, InstanceofExpression, VariableDeclarationExpression
数组访问等	ArrayAccess, ArrayCreation, ArrayInitializer
成员访问等	ClassInstanceCreation, FieldAccess, MethodInvocation, SuperFieldAccess, SuperMethodInvocation, ThisExpression
表达式构造	InfixExpression, ParenthesizedExpression, PostfixExpression, PrefixExpression

构造方法调用和超类构造方法调用语句可能需要生成对于构造方法的引用，但它又是特殊的构造方法引用（一般是通过实例创建表达式调用构造方法），**我们目前暂时不处理它**，因为它的方法名用关键字代替，从概念上说关键字不能看做引用！类型声明语句通常声明局部类型，需要为之生成详细类型定义。变量声明语句则要生成变量定义。

其他语句，包括for语句、增强型for语句，以及不是语句的AST节点CatchClause都可能包含变量声明，我们需要进行判断以生成合适的局部作用域，并在其中生成变量定义。

表4.10给出了表示Java表达式的AST节点类，它们都是AST节点类表达式Expression的子类。可以看到，我们实际上为除表示文字之外的表达式都设计了相应的名字引用组。

4.4.2 类—名字表生成器及相关类

名字表生成器(NameTableCreator)要完成的职责就是给定一个源代码文件集，然后生成名字定义、名字作用域和名字引用。由于Eclipse JDT在生成抽象语法树时只能针对一个源代码文件，因此名字表生成器需要遍历源代码文件集的每个源代码文件(SourceCodeFile)，为快速访问在生成名字表时所需的源代码文件及其抽象语法树根节点，我们设计一个编译单元文件类(CompilationUnitFile)，它是一个轻量级数据类，存储一些公有属性供名字生成时快速访问：

(AT1) 单元名(unitName)，也即源代码文件（即Java编译单元）的单元名（注意每个源代码文件有全名、单元名和简单名），单元名是全名中去掉源代码文件集的起始路径之后剩下的字符串。**在生成源代码位置时我们使用单元名；**

(AT2). 抽象语法树根节点(root)：即对应该源代码文件的AST根节点（AST节点类编译单元(CompilationUnit)的实例）；

(AT3). 错误信息(errorMessage)：在为该源代码文件生成抽象语法树，或构件名字表时产生的错误信息。

上述属性都是公有终态属性，因此无需提供getter和setter方法。类CompilationUnitFile只用于名字表生成，以及查询名字表生成是否有错误时使用。

名字表生成器针对一个源代码文件集进行名字表的生成，因此它有属性：

(AT1) 源代码文件集(codeFileSet)：要生成名字表的源代码文件集（对应一个Java软件项目的所有源代码）；

为记录在名字表生成过程时可能存在的错误，例如抽象语法树生成时的编译错误等，我们在名字表生成器中设置属性来记录出现错误的源代码文件：

(AT2) 错误编译单元列表(errorUnitList)：列表元素类型是类CompilationUnitFile。

名字表生成器以源代码文件集作为构造方法的参数。对于每个源代码文件（编译单元），名字表生成器根据Java程序的语法结构，扫描其程序包声明、导入声明、类型声明，因此名字表生成器应有操作：

(RP1). 扫描当前编译单元文件(scan(CompilationUnitFile, SystemScope))：类型SystemScope的参数是当前源代码文件集的系统作用域，用于组织扫描编译单元文件时生成的任何名字定义、名字作用域和名字引用；

(RP2). 扫描类型声明(scan(CompilationUnitFile, String, TypeDeclaration, NameScope))：参数String提供扫描参数TypeDeclaration给定的类型声明时，该类型声明应该有限定名，该字符串加上该类型声明的简单名构成该类型声明的全限定名。类型CompilationUnitFile的参数指定该类型声明所属的编译单元，而NameScope类型的参数则指定该类型声明所属的作用域。

(RP3). 扫描匿名类型声明(scan(CompilationUnitFile, AnonymousClassDeclaration, NameScope));

(RP4). 扫描枚举类型声明(scan(CompilationUnitFile, String, EnumTypeDeclaration, NameScope));

- (RP5). 扫描枚举常量声明(scan(CompilationUnitFile, String, EnumConstantDeclaration, NameScope));
- (RP6). 扫描字段声明(scan(CompilationUnitFile, String, FieldDeclaration, NameScope));
- (RP7). 扫描方法声明(scan(CompilationUnitFile, String, MethodDeclaration, NameScope));
- 最后,名字表生成器提供操作生成名字表,并提供处理错误信息的操作:
- (RP8). 生成名字表(create());
- (RP9). 是否有错误编译单元(hasError()): 如果错误编译单元列表的大小大于0返回true;
- (RP10). 返回错误编译单元列表(getErrorUnitList());
- (RP11). 打印错误编译单元列表(printErrorUnitList(PrintWriter)): 将错误编译单元列表打印到指定的输出设备,每一行给出一个编译单元名及错误信息,如果编译单元的语法错误信息有多行,则只输出第一行错误。

对方法声明扫描需要遍历方法体的语句,因为AST语句节点类型众多,我们设计抽象语法树的访问者,即Eclipse JDT的类ASTVisitor的子类处理每个可能在方法体中出现的每个AST节点。我们设计三个抽象语法树访问器类:

1. 类语句块访问器(BlockASTVisitor): 访问方法体语句块中的每个AST节点,生成名字定义和名字引用,并在需要时创建局部作用域;
2. 类类型访问器(TypeASTVisitor): 访问每个表示类型的AST节点,生成一个类型引用;
3. 表达式访问器(ExpressionASTVisitor): 访问每个表示表达式的AST节点,生成一个名字引用组,在需要时(对于变量声明表达式(VariableDeclarationExpression))生成名字定义。

类语句块访问器(BlockASTVisitor)的关键在于在需要时创建局部作用域,它需要维持一个作用域栈,使得最后创建的名字定义(通常是变量定义)放在紧包围它的作用域内,因此它有属性:

- (AT1). 作用域栈(scopeStack): 存放名字作用域的栈,每当创建局部作用域时入栈,从该局部作用域出来时退栈;
- (AT2). 编译单元(unitFile): 当前块语句属于的编译单元,类型是CompilationUnitFile;
- (AT3). 类型访问器(typeVisitor): 用于遍历表示类型的AST节点,以便生成名字定义时所必须的一些名字引用,例如类型定义的超类型声明、字段声明的类型、方法的返回类型等等;
- (AT4). 表达式访问器(expressionVisitor): 用于遍历表示表达式的AST节点,以便生成名字引用。
- (AT5). 名字表生成器(creator): 使用这个语句块访问器的名字表生成器,因为在块语句中可能存在(局部)类型声明和匿名类声明,需要使用名字表生成器的扫描方法来扫描对(局部)类型声明和匿名类声明!

语句块访问器以编译单元、名字表生成器、所在作用域作为构造方法的参数。它的主要职责是重定义类ASTVisitor的访问各抽象语法节点的visit()方法,在这些方法中,主要是如下几个方法需要注意:

- (1). 方法visit(Block): 访问块语句,需要判断是否创建局部作用域,如果要创建则将创建局部作用域入栈,然后再访问该块语句的语句序列,再访问完毕之后再退栈;
- (2). 方法visit(CatchClause): 访问try语句的catch子句,在子句中要声明捕获的异常,因此有变量声明,需要创建局部作用域;
- (3). 方法visit(EnhancedForStatement): 访问增强型for语句,其圆括号中有变量声明,因此

需要创建局部作用域，其起始位置是该变量声明之后的位置，而终止位置是整个语句的结束点。即使该for语句的循环体是空语句，也应该创建局部作用域；该for语句的循环体应该以创建的局部作用域为当前作用域进行访问。

(4). 方法visit(ForStatement): 访问for语句，它的初始化语句和循环体都可能有变量声明，因此可能创建局部作用域；

(5). 方法visit(TypeDeclarationStatement): 访问局部类型声明，需要调用名字表生成器的扫描类型声明(scan())方法进行扫描；

(6). 方法visit(AnonymousClassDeclaration): 访问匿名类声明，需要调用名字表生成器的扫描匿名类声明(scan())方法进行扫描。

类型访问器(TypeASTVisitor)遍历表示类型（引用）的AST节点，遍历结果是一个类型引用，因此它的属性有：

(AT1). 遍历结果result: 是一个TypeReference的实例；

当遍历数组类型时，需要将它的元素类型遍历完才能得到整个类型引用，但在遍历元素类型时已经没有数组维数信息，所以这个类需要记录这个类型引用中的维数信息：

(AT2). 数组维数dimension: 同样0表示不是数组类型；

基于同样的原因，类型引用往往需要在遍历完（返回到使用者）才能完整创建，因此需要记录创建类型引用的其他信息：

(AT3). 名字(name): 该类型引用的名字；

(AT4). 编译单元(unitFile): 该类型引用在这个编译单元中；

(AT5). 当前作用域(currentScope): 该类型引用在当前这个作用域中；

(AT6). 结果引用的源代码位置(resultLocation): 为这个引用找到最合适的源代码位置。

类型访问器以编译单元、当前作用域为构造方法的参数，它的主要职责就是重定义类ASTVisitor中表示类型节点的visit()方法，根据相应的节点，创建相应的类型引用，并获取所需的信息。除此之外，它还提供：

(RP1). 返回遍历结果(getResult()): 返回类型引用。使用者调用这个方法也意味着遍历的完成，所以很多类别类型的创建是在这个方法中才创建的，但也有些visit()方法可能可以直接创建最后的遍历结果。如果属性result为空则表示要在这个方法中创建结果，否则不需要！

(RP2). 重置访问器(reset(CompilationUnitFile, NameScope)): 以新的当前编译单元和当前作用域重置访问器。因为表示类型节点的遍历是一个很局部的遍历，因此类型访问器可以在整个名字生成过程中不断重用，而无需不断创建新的访问器对象实例！

表达式访问器(ExpressionASTVisitor)遍历表示表达式的AST节点，遍历结果是一个名字引用，因此它的属性有：

(AT1). 遍历结果(lastReference): 表达式的语法结构是一棵树，实际上在遍历的时候将是后序遍历这棵树，因此最后创建的名字引用（通常是名字引用组）是这棵树的根节点，也是遍历的结果；

同样表达式访问器要记住上下文信息：

(AT3). 编译单元(unitFile): 该表达式在这个编译单元中；

(AT4). 当前作用域(currentScope): 该表达式在当前这个作用域中；

表达式访问器以编译单元、当前作用域为构造方法的参数。它的主要职责就是重定义

类ASTVisitor中表示表达式节点的visit()方法，根据相应的节点，创建相应的名字引用（组），并获取所需的信息。除此之外，它还提供：

(RP1). 返回遍历结果(getResult()): 返回的是一个名字引用（组）；

(RP2). 重置访问器(reset(CompilationUnitFile, NameScope)): 以新的当前编译单元和当前作用域重置访问器。同样表示表达式节点的遍历是一个很局部的遍历，因此表达式访问器可以在整个名字生成过程中不断重用，而无需不断创建新的访问器对象实例！

4.4.3 类—名字定义生成器及相关类

名字定义生成器(NameDefinitionCreator)的职责与名字表生成器(NameTableCreator)类似，但只生成名字定义和名字作用域，也即，名字定义生成器完成名字表生成器的部分职责，因此可将名字定义生成器作为名字表生成器的直接子类，无需设置新的属性，只要重定义名字表生成器的各个scan()方法，在扫描编译单元、类型声明、枚举类型声明、枚举常量声明、字段声明和方法声明时只生成名字定义和作用域，以及名字定义所必须的名字引用（通常是类型引用）即可。为了在扫描时只生成名字定义、作用域和必须的名字引用，也使用不同的抽象语法树访问器：

1. 语句块名字定义访问器(BlockDefinitionASTVisitor): 访问语句块中的每个AST节点，生成名字定义和必须的名字引用，并在需要时创建局部作用域；

2. 表达式名字定义访问器(ExpressionDefinitionASTVisitor): 访问每个表示表达式的AST节点，生成名字引用组，且在需要时（对于变量声明表达式(VariableDeclarationExpression)）生成名字定义。

语句块名字定义访问器(BlockDefinitionASTVisitor)是语句块访问器(BlockASTVisitor)的直接子类，不设置新的属性，构造方法的参数也相同，只是重定义其中的一些visit()方法，去掉其中完成生成不必要的名字引用的语句。

表达式名字定义访问器(ExpressionDefinitionASTVisitor)是表达式访问器(ExpressionASTVisitor)的直接子类，不设置新的属性，构造方法的参数也相同，只是将除visit(VariableDeclarationExpression node)之外的visit()都改为直接返回（因为其他的表达式节点都不会包含名字定义）。

类型访问器无需重新定义，因为它只是用来生成类型引用，而且通常是在生成名字定义时需要使用类型访问器获得名字定义中必须的一些名字引用。

4.4.4 类—名字引用生成器及相关类

名字引用生成器(NameReferenceCreator)不是名字表生成器(NameTableCreator)的子类，它基于已有的名字表管理器，并使用名字表抽象语法树桥接器为各种作用域以及抽象语法树节点生成名字引用。因此它的主要属性有：

(AT1). 名字表管理器(tableManager): 是类名字表管理器(NameTableManager)的实例；

名字引用生成器以一个名字表管理器为构造方法的参数，主要职责是为各种作用域和抽象语法树节点生成名字引用：

(RP1). 生成指定编译单元作用域的所有名字引用(createReferences(CompilationUnitScope)): 生成指定编译单元中的所有名字引用（但不包括为导入声明创建的名字引用）构成的列表；

(RP2). 生成指定详细类型定义的所有名字引用(`createReferences(DetailedTypeDefinition)`)：生成详细类型定义中的所有名字引用（包括其内部类、局部类、匿名类中的引用）构成的列表；

(RP3). 生成方法定义的所有名字引用(`createReferences(MethodDefinition)`)：生成方法定义中的所有名字引用（包括局部类、匿名类中的引用）构成的列表；

(RP4). 生成字段定义的所有名字引用(`createReferences(FieldDefinition)`)：生成字段定义中的所有名字引用构成的列表，通常是出现在字段初始表达式中的名字引用；

上述操作返回的列表总是不为null，如果没有名字引用，或在指定名字表管理器没有找到对应的名字作用域或名字定义，则返回大小为0的列表。

实际上，生成指定编译单元的所有名字引用是最基本的操作，利用这个操作可完成以下操作：

(RP5). 生成指定名字串的所有名字引用(`createReferencesOfName(String)`)：遍历名字表管理器中的所有编译单元，生成其中的名字引用，并过滤其中名字是指定名字串的名字引用；

(RP6). 生成引用指定名字定义的所有名字引用(`createReferencesBindToDefinition(NameDefinition)`)：遍历名字表管理中的所有编译单元，生成其中的名字引用，并进行解析，过滤其中绑定结果是指定名字定义的名字引用；

(RP7). 生成满足过滤条件的名字引用(`createReferences(NameTableFilter)`)：遍历名字表管理中的所有编译单元，生成其中的名字引用，并进行解析，然后使用指定过滤器过滤其中的名字引用。

在生成指定编译单元作用域、详细类型定义、方法定义、字段定义中的名字引用时，都需要使用名字表抽象语法树桥接器(`NameTableASTBridge`)为名字表实体（编译单元作用域、详细类型定义、方法定义、字段定义）查找对应的抽象语法树节点（编译单元、类型声明、方法声明、字段声明），然后同名字表生成器中的扫描方法一样遍历抽象语法树节点，完成名字引用的生成。在遍历抽象语法树节点时，同样要使用到抽象语法树访问器：

1. 语句块名字引用访问器(`BlockReferenceASTVisitor`)：访问语句块中的每个AST节点，但只关注名字引用的生成；

2. 表达式名字引用访问器(`ExpressionReferenceASTVisitor`)：访问每个表示表达式的AST节点，生成名字引用组，与`ExpressionASTVisitor`的唯一差别是在访问`VariableDeclarationExpression`类型的节点时不生成变量定义。

语句块名字引用访问器(`BlockReferenceASTVisitor`)是类`ASTVisitor`的直接子类，它也类`BlockASTVisitor`有很大不同，因而不是它的子类。类`BlockASTVisitor`需要维持作用域栈，以便在需要创建局部作用域，使得生成的名字定义在合适的作用域。而类`BlockReferenceASTVisitor`的关键在于为每个遍历的AST节点在名字表中找到最合适的作用域，为该节点生成的名字引用属于该作用域。因此语句块名字引用访问器的属性有：

(AT1). 编译单元(`unitFile`)：该语句块在这个编译单元中；

(AT2). 语句块作用域(`blockScope`)：该语句块对应的（局部）作用域；

(AT3). 遍历结果(`resultList`)：遍历的结果是名字引用列表，我们不能将不同语句中的名字引用组成名字引用组，所以遍历结果名字引用列表；

(AT4). 类型访问器(`typeVisitor`)：类型访问器(`TypeASTVisitor`)的实例，用于遍历代表类型的节点，生成类型引用；

(AT5). 表达式访问器(`expressionVisitor`)：表达式名字引用访问器(`ExpressionReferenceVisitor`)的

实例，用以遍历代表表达式的节点，生成名字引用。

语句块名字引用访问器(BlockReferenceASTVisitor)以编译单元、语句块作用域为构造方法的参数。它的职责主要是重定义类ASTVisitor的visit()方法，完成名字引用的生成并返回给使用者，为此它还提供如下操作：

- (RP1). 返回遍历结果(getResult()), 返回遍历的结果, 即名字引用列表。
 - (RP2). 重置访问器(reset(CompilationUnitFile, LocalScope)): 以新的编译单元和局部作用域重置访问器。

(RP3). 返回包含源代码位置最内层作用域(`getNameScopeOfLocation(SourceCodeLocation)`)：这可以是一个私有方法，供内部使用以查找所生成的名字引用所在的最内层的作用域。

表达式名字引用访问器(ExpressionReferenceASTVisitor)是类表达式访问器ExpressionASTVisitor的直接子类，构造方法也以编译单元和作用域为参数，它不重新设置新的属性，只是重定义其中访问VariableDeclarationExpression类型的AST节点时，使其不生成名字定义。

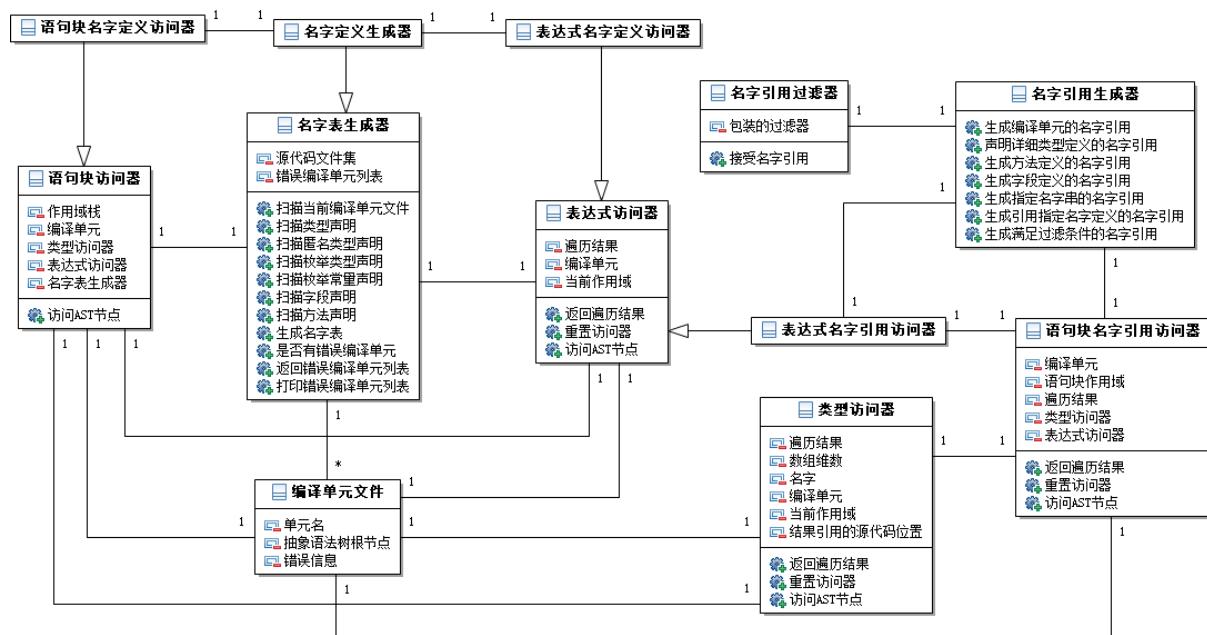


图 4.10 名字表生成相关的类

因为生成名字引用时也通常要使用这些名字引用,因此设计类名字引用过滤器(`NameReferenceFilter`)来生成和使用满足过滤条件的名字引用。

类名字引用过滤器(`NameReferenceFilter`)用于设置过滤名字引用的条件，它与名字定义过滤器的设计类似，使用包装模式(decorate pattern)。名字引用过滤器的属性有：

- (AT1). 包装的过滤器(wrappedFilter): 先使用包装的过滤器过滤, 然后再使用当前过滤器进行过滤, 形成过滤器链.

名字引用过滤器使用包装的过滤器为参数的构造方法，它只提供一个方法：

- (RP1). 接受名字引用(`accept(NameReference)`): 给出是否接受名字引用的具体条件, 如果接受则返回`true`, 否则返回`false`。缺省的实现是返回包装的过滤器的`accept()`方法的返回结果, 后代类可重定义这个方法以给出具体过滤条件。

图4.10给出了类名字表生成相关的类。名字表生成器使用语句块访问器、表达式访问器生成名字定义、名字作用域和名字引用。只生成名字定义和名字作用域时使用名字定义生成器，它使用语句块名字定义访问器和表达式名字定义访问器。名字引用生成器使用语句块名字引用访问器、表达式名字引用访问器生成名字引用。编译单元文件和类型访问器实际在这些类中都会被使用，但上面只展示了它们与主要类的关联关系。

4.5 用例实现

名字表生成构件主要是提供一些API供构件使用生成和访问名字表，不设计用户界面与终端用户进行交互，所以它的用例的交互都比较简单，主要是构件使用者提供所需的信息（包括指定条件），以生成和访问名字表。这里简单地讨论在需求分析中给出的几个用例的实现。

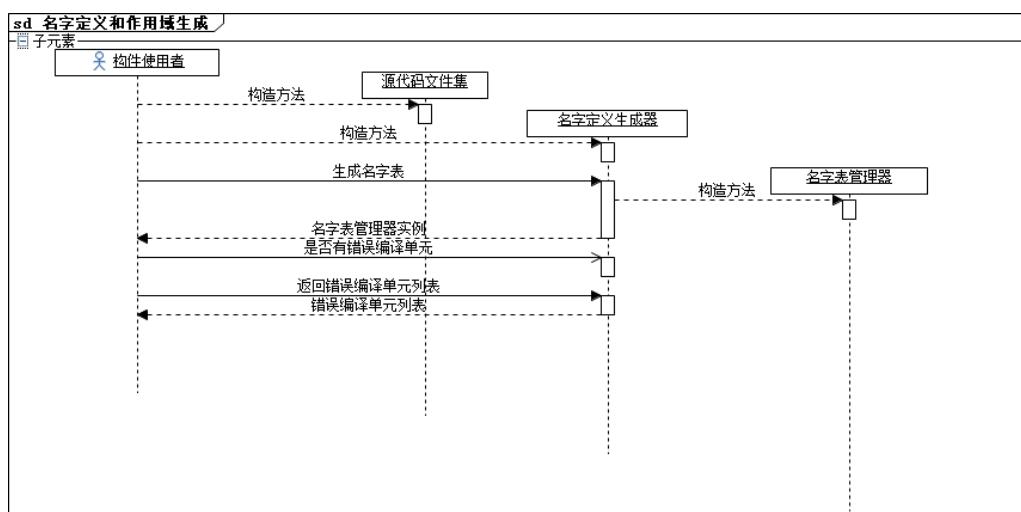


图 4.11 实现用例1. 名字定义和作用域生成

图4.11给出了实现用例1. 名字定义和作用域生成的顺序图。可以看到，该用例通过相关类的对象如下交互实现：

1. 构件使用者使用类源代码文件集(SourceCodeFileSet)的构造方法创建一个源代码文件集对象实例；
2. 以该源代码文件集对象为参数使用类名字定义生成器(NameDefinitionCreator)创建一个名字定义生成器对象实例；
3. 使用名字定义生成器的生成名字表(create())方法生成名字定义和作用域，在生成时，该方法创建名字表管理器(NameTableManager)的实例，并返回该实例；
4. 构件使用者可使用名字定义生成器的方法hasError()方法查询在生成时是否有错误编译单元；
5. 如果有错误编译单元，则使用名字定义生成器的方法返回错误编译单元列表(getErrorUnitList())得到这些错误编译单元的列表并进行处理。

虽然在名字定义和名字作用域生成过程中，名字定义生成器需要使用块语句访问器(BlockASTVisitor)、类型访问器(TypeASTVisitor)和表达式访问器(ExpressionASTVisitor)的实例遍历抽象语法树节点，但构件使用者无需了解名字生成这些构件之间的内部交互过程，所以上面没有画出这些交互。

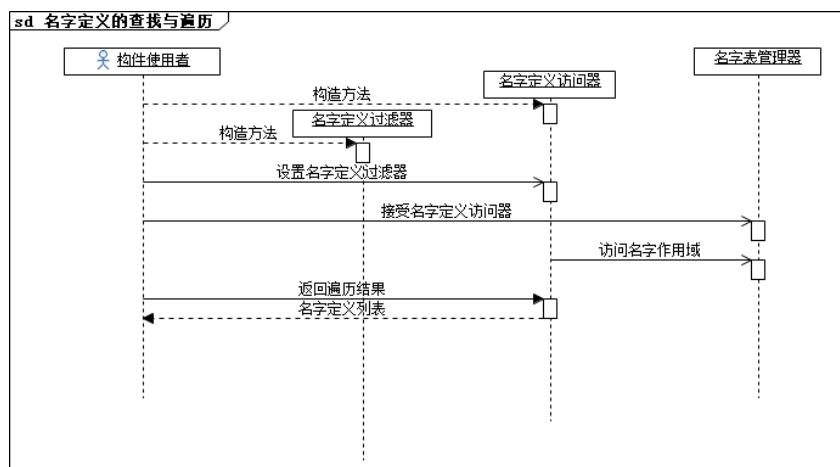


图 4.12 实现用例2. 名字定义的查找与遍历

图4.12给出了实现用例2. 名字定义的查找与遍历的顺序图。可以看到，该用例通过相关类的对象如下交互实现：

1. 构件使用者创建名字定义访问器(NameDefinitionVisitor)的对象实例；
2. 构件使用者创建名字定义过滤器(NameDefinitionFilter)的对象实例；
3. 构件使用者使用名字定义访问器的设置过滤器(setFilter())方法设置名字定义过滤器；
4. 构件使用者使用名字表管理器的接受名字表访问器(accept())方法；
5. 名字定义访问器的访问visit()方法遍历名字表的名字作用域树结构，并在遍历过程中使用过滤器过滤名字定义；
6. 构件使用者使用名字定义访问器的返回遍历结果(getResult())得到满足过滤条件的名字定义列表。

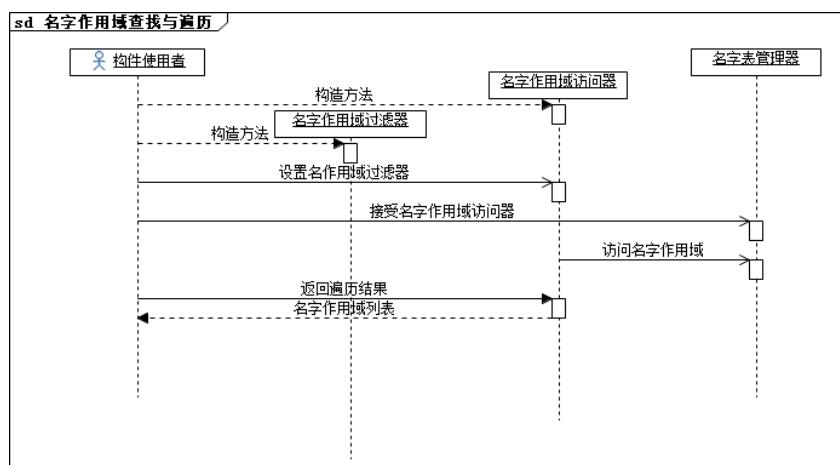


图 4.13 实现用例3. 名字作用域的查找与遍历

图4.14给出了实现用例3. 名字作用域的查找与遍历的顺序图。可以看到，该用例通过相关类的对象如下交互实现：

1. 构件使用者创建名字作用域访问器(NameScopeVisitor)的对象实例；
2. 构件使用者创建名字作用域过滤器(NameScopeFilter)的对象实例；

3. 构件使用者使用名字作用域访问器的设置过滤器(`setFilter()`)方法设置名字作用域过滤器;
4. 构件使用者使用名字表管理器的接受名字表访问器(`accept()`)方法;
5. 名字作用域访问器的访问`visit()`方法遍历名字表的名字作用域树结构，并在遍历过程中使用过滤器过滤名字作用域;
6. 构件使用者使用名字作用域访问器的返回遍历结果(`getResult()`)得到满足过滤条件的名字作用域列表。

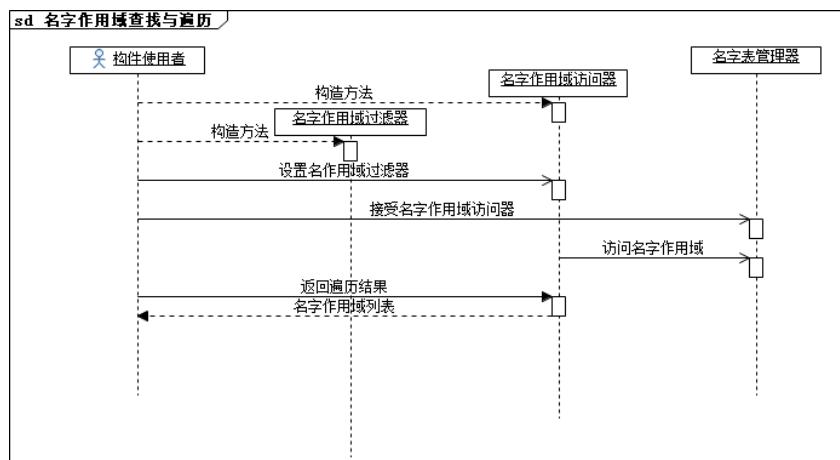


图4.14 实现用例3. 名字作用域的查找与遍历

图4.14给出了实现用例3. 名字作用域的查找与遍历的顺序图。可以看到，该用例通过相关类的对象如下交互实现：

1. 构件使用者创建名字作用域访问器(`NameScopeVisitor`)的对象实例;
2. 构件使用者创建名字作用域过滤器(`NameScopeFilter`)的对象实例;
3. 构件使用者使用名字作用域访问器的设置过滤器(`setFilter()`)方法设置名字作用域过滤器;
4. 构件使用者使用名字表管理器的接受名字表访问器(`accept()`)方法;
5. 名字作用域访问器的访问`visit()`方法遍历名字表的名字作用域树结构，并在遍历过程中使用过滤器过滤名字作用域;
6. 构件使用者使用名字作用域访问器的返回遍历结果(`getResult()`)得到满足过滤条件的名字作用域列表。

图4.15给出了实现用例4. 名字引用的生成与使用的顺序图。可以看到，该用例通过相关类的对象如下交互实现：

1. 构件使用者创建名字名字引用生成器(`NameReferenceCreator`)的对象实例;
2. 构件使用者创建名字引用过滤器(`NameReferenceFilter`)的对象实例;
3. 构件使用者使用名字引用生成器的设置过滤器(`setFilter()`)方法设置名字引用过滤器;
4. 构件使用者使用名字引用生成的生成名字引用(`createReference`)方法生成名字引用并返回名字引用：
 - 4.1. 名字引用生成器需要访问名字表管理器，遍历名字表的所有编译单元作用域；

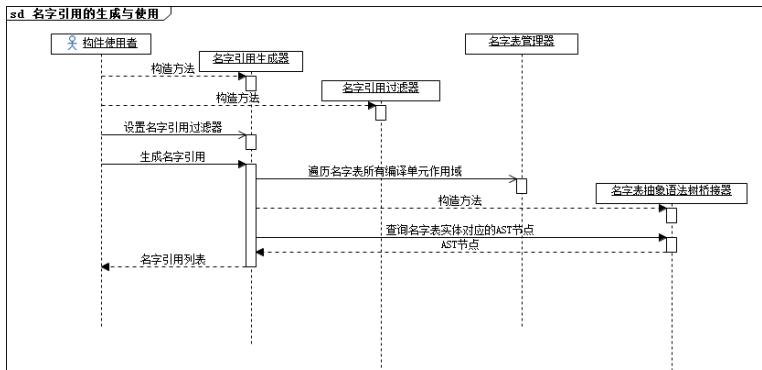


图 4.15 实现用例4. 名字引用的生成与使用

4.2. 名字引用生成器创建名字表抽象语法树桥接器，以查询名字实体对应的AST节点，名字引用生成器使用块语句名字引用访问器、表达式名字引用访问器和类型访问器生成名字引用。

不过顺序图并不能将名字引用的生成过程完整描述，上面只是给出了这些对象实例之间交互的简单示意图。

第五章 名字表生成构件的实现

基于本文档对名字表生成构件的设计，我们对以前实现的名字表生成构件进行了重构，主要的修改包括：

(1) 简化类NameTableManager的职责，删除了一些不常用的名字表访问方法，以及名字引用生成方法（名字引用生成职责该由类NameReferenceCreator完成，以及与抽象语法树相关的方法（相关职责由类NameTableASTBridge）完成。

(2) 简化了接口NameScope，将名字表访问、名字定义和名字引用打印等相关职责通过访问器模式实现，而不是在实现名字作用域接口(NameScope)的类中实现。

(3) 完善了在生成名字定义、作用域和名字引用时的抽象语法树访问器类，使得可处理类的初始块、匿名类等语法成分，并且根据Java语言的最新规范增加了一些抽象语法树节点（类ASTNode的子类）的访问方法，虽然其中有关Lambda表达式、方法引用表达式、标记(annotations)、Javadoc注释文档等相关的节点仍没有真正处理。

重构后的名字表生成构件包括8个程序包，如表5.1所示。代表名字表实体，即名字定义、名字引用和名字作用域的类分别放在程序包nameTable.nameDefinition、nameTable.nameReference、nameTable.nameReference.referenceGroup和nameTable.nameScope中。与使用名字表相关的类放在程序包nameTable、nameTable.visitor和nameTable.filter中。程序包nameTable.creator中的类与生成名字表实体（名字定义、引用和作用域）有关。

表 5.1 名字表生成构件的程序包

程序包	描述	程序包	描述
nameTable	名字表使用	nameTable.visitor	名字表访问器
nameTable.filter	名字表过滤器	nameTable.creator	名字表生成器
nameTable.nameDefinition	名字定义	nameTable.nameScope	名字作用域
nameTable.nameReference	名字引用	nameTable.nameReference.referenceGroup	名字引用组

下面我们仍然按照从名字表实体到名字使用表，再到名字表生成的顺序介绍这些程序包中的类的实现，主要是列出类的字段和方法，并对其中一些关键方法的实现要点进行说明。

5.1 名字作用域

代表名字作用域的是接口NameScope，实现该接口的类都代表一种名字作用域，包括：

(1) 类系统作用域(SystemScope): 代表系统作用域, 即包含整个源代码文件集所有源代码的作用域;

(2) 类编译单元作用域(CompilationUnitScope): 代表一个编译单元的所有源代码构成的作用域;

(3) 类局部作用域(LocalScope): 代表一个左右花括号括起来的语句块所构成的作用域。

这三个类在程序包nameTable.nameScope中, 下面的类也实现接口NameScope, 但同时也是名字定义, 因此在程序包nameTable.nameDefinition中:

(4) 类程序包定义(PackageDefinition): 代表一个程序包构成的作用域;

(5) 类详细类型定义(DetailedTypeDefinition): 代表一个类或接口所构成的作用域;

(6) 类枚举类型定义(EnumTypeDefinition): 代表一个枚举类型所构成的作用域;

(7) 类方法定义(MethodDefinition): 代表一个方法所构成的作用域;

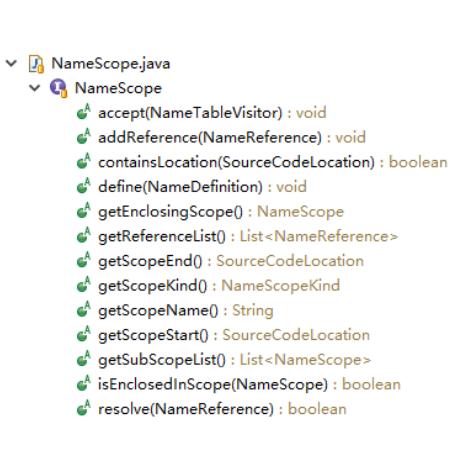
(8) 类导入类型定义(ImportedTypeDefinition)。

与设计阶段有点不同的是我们让类ImportedTypeDefinition也实现接口NameScope, 这是因为我们支持编写一些外部文件来提供更多有关导入类型的信息, 使得我们能“理解”(解析)更多的名字引用。这些外部文件可像写抽象类一样给出常用导入类型的重要字段(例如类System的out字段)和方法签名(signature)(例如类List的add()方法)。由于导入类型也可能有字段和方法, 因此我们也让它实现接口NameScope, 但它不算真正的作用域, 其中只有名字定义, 没有名字引用, 不属于我们所要分析和理解的源代码的一部分。

程序包nameTable.nameScope还有一个枚举类型NameScopeKind, 它用于标识不同的名字作用域种类, 其枚举常量在表4.1有定义(不过增加了常量NSK_IMPORTED_TYPE表示导入类型所对应的作用域), 因此这里不再介绍。

5.1.1 接口NameScope

代表名字作用域的接口NameScope在编译单元NameScope.java, 图5.1给出了它所定义的方法。



成员	描述
accept()	接受名字表访问器访问
addReference()	添加名字引用到作用域
containsLocation()	判断是否包含某位置
define()	定义名字到作用域
getEnclosingScope()	返回父作用域
getReferenceList()	返回名字引用列表
getScopeEnd()	返回作用域结束位置
getScopeKind()	返回作用域类别
getScopeName()	返回作用域名字
getScopeStart	返回作用域起始位置
getSubScopeList()	返回子作用域列表
isEnclosedInScope(NameScope)	判断是否真包含在另一作用域
resolve()	在当前作用域解析名字引用

图 5.1 接口NameScope定义的方法

与前面设计稍有不同的是，为支持当要分析的程序规模比较小时同时生成名字定义、作用域即名字引用，我们为名字作用域设计了添加名字引用(`addReference()`)和返回名字引用列表(`getReferenceList()`)（返回列表的元素类型是`NameReference`）两个方法。通常一个作用域保存直接出现在该作用域（不含子作用域）中的名字引用。

为实现访问名字表的访问器模式，每个名字作用域都需要实现`accept()`方法。这个方法在每个代表名字作用域类的实现都基本相同：先调用名字表访问器的`preVisit()`方法，然后调用`visit()`方法访问当前作用域，然后在该方法返回`true`时访问当前作用域所包含的每个子作用域，最后调用访问器的`endVisit()`方法。

每个代表名字作用域类所实现的`define()`方法都是在当前作用域添加名字定义，不同作用域可有不同的名字定义，因此相应的类有存放不同名字定义的列表。而相应的`resolve()`方法则是当前作用域解析名字引用，通常就是遍历存放在当前作用域的名字定义列表，利用类`NameDefinition`的`match()`方法匹配名字定义的简单名（或全限定名）与名字引用中保存的名字（当名字引用中保存的名字有圆点“.”时匹配全限定名，否则匹配简单名）。

5.1.2 类SystemScope

类`SystemScope`代表系统作用域。系统作用域是概念上存在的作用域，它包含所要分析的所有源代码，因此是实际上的名字表入口。所要分析的所有源代码文件构成一个源代码文件集，都处在某个文件路径下面，这个文件路径称为源代码文件集的起始路径。源代码文件集中的所有源代码文件定义的名字作用域按照包含关系构成树，系统作用域是这个树的根节点。

表 5.2 类`SystemScope`字段的描述

字段	类型	描述
<code>ROOT_OBJECT_NAME</code>	<code>String</code>	根对象名称，值为“Object”
<code>SYSTEM_PACKAGE_NAME</code>	<code>String</code>	系统包名称，值为“java.lang”
<code>SYSTEM_SCOPE_NAME</code>	<code>String</code>	系统作用域名称，值为“<System>”
<code>allDetailedTypeList</code>	<code>List<DetailedTypeDefinition></code>	所有详细类型定义列表
<code>importedStaticMemberList</code>	<code>List<ImportedStaticMemberDefinition></code>	导入静态成员列表
<code>importedTypeList</code>	<code>List<ImportedTypeDefinition></code>	导入类型列表
<code>packageList</code>	<code>List<PackageDefinition></code>	程序包列表
<code>referenceList</code>	<code>List<NameReference></code>	名字引用列表

表5.2给出了类`SystemScope`的字段，前面三个是内部所使用的静态常量，用来标识Java语言的根类`java.lang.Object`、系统包（自动导入的程序包）`java.lang`和我们给系统作用域的特别名字`<System>`。字段`allDetailedTypeList`记录所有详细类型定义构成的列表，它的作用主要是缓存，因为经常访问所有详细类型定义列表，缓存系统作用域后不用每次都遍历整个名字表才得到这个列表。剩下的字段则给出了定义在系统作用域中的程序包、导入类型和导入静态成员，以及用于存放在同时生成名字定义和名字引用时的名字引用列表。正常来说，没有任何名字引用是直接定义在系统作用域，因此系统作用域的名字引用列表应保持为`null`。

表5.3给出了类`SystemScope`的方法，多数方法是用于实现接口`NameScope`中定义的方法。系统作用域本身没有作用域起始位置和结束位置，因此返回这两个位置的方法都返回`null`，但它包含任

表 5.3 类SystemScope方法的描述

<code>getRootScope(NameScope) : SystemScope</code>	静态方法, 返回某个作用域所在的系统作用域
<code>accept(NameTableVisitor) : void</code> [实现接口NameScope的方法]	接受名字表访问器的访问
<code>addReference(NameReference) : void</code> [实现接口NameScope的方法]	添加引用
<code>containsLocation(SourceCodeLocation) : boolean</code> [实现接口NameScope的方法]	判断是否包含某源代码位置, 总是返回true
<code>define(NameDefinition) : void</code> [实现接口NameScope的方法]	在作用域中定义名字
<code>findPackageByName(String) : PackageDefinition</code>	基于程序包全名查找程序包定义
<code>getAllDetailedTypeDefinitions() : List<DetailedTypeDefinition></code>	返回所有详细类型定义构成的列表
<code>getAllOverrideMethods(DetailedTypeDefinition, MethodDefinition) : List<MethodDefinition></code>	返回给定详细类型定义的方法定义的所有重定义方法
<code>getAllOverrideMethods(ImportedTypeDefinition, MethodDefinition) : List<MethodDefinition></code>	返回给定导入类型定义的方法定义的所有重定义方法
<code>getEnclosingScope() : NameScope</code> [实现接口NameScope的方法]	返回包含当前作用域的作用域(即父作用域)
<code>getImportedStaticMemberList() : List<ImportedStaticMemberDefinition></code>	返回导入静态成员列表
<code>getImportedTypeList() : List<ImportedTypeDefinition></code>	返回导入类型列表
<code>getPackageList() : List<PackageDefinition></code>	返回程序包列表
<code>getReferenceList() : List<NameReference></code> [实现接口NameScope的方法]	返回引用列表, 对于系统作用域而言是空表
<code>getScopeEnd() : SourceCodeLocation</code> [实现接口NameScope的方法]	返回作用域的结束位置, 总是返回null
<code>getScopeKind() : NameScopeKind</code> [实现接口NameScope的方法]	返回作用域类别, 即返回NSK_SYSTEM
<code>getScopeName() : String</code> [实现接口NameScope的方法]	返回作用域名称, 即返回SYSTEM_SCOPE_NAME
<code>getScopeStart() : SourceCodeLocation</code> [实现接口NameScope的方法]	返回作用域起始位置, 总是返回null
<code>getSubScopeList() : List<NameScope></code> [实现接口NameScope的方法]	接受名字表访问器的访问
<code>getUnnamedPackageDefinition() : PackageDefinition</code>	返回不具名程序包对应的程序包定义(如果有的话, 否则返回null)
<code>isEnclosedInScope(NameScope) : boolean</code> [实现接口NameScope的方法]	判断是否真包含在另一作用域, 总是返回false
<code>resolve(NameReference) : boolean</code> [实现接口NameScope的方法]	在系统作用域中解析名字引用, 主要是匹配导入类型定义和导入静态成员定义

何程序位置，所以对于判断是否包含某源代码位置的方法总是返回`true`（并没有查验这个位置是否是有效位置）。

除实现接口`NameScope`定义的方法外，类`SystemScope`还提供方法`findPackageByName()`用于根据程序包的全名查找相应的程序包定义（类`PackageDefinition`的对象实例）。由于不具名程序包没有名字，因此提供了方法`getUnnamedPackageDefinition()`返回可能存在的不具备程序包对应的程序包定义（如果没有则返回`null`）。

类`SystemScope`为程序包列表、导入类型列表、导入静态成员列表提供了`getter`方法，注意为这些列表添加元素是在创建名字表时使用`define()`方法完成的。**名字表构件的使用者在创建名字表后，应不修改有关名字定义和名字作用域的任何对象**，因此名字表构件使用者无需关心这些列表的修改，也不要直接调用任何实现接口`NameScope`的`define()`方法。

类`SystemScope`的方法`getAllDetailedTypeDefinitions()`返回所有详细类型定义构成的列表（包括匿名类、局部类和内部类，但不含枚举类型），这个方法维护字段`allDetailedTypeList`（如果它不为空则直接返回该字段，否则使用名字表访问器访问系统作用域获得所有详细类型定义构成的列表）。

有两个`getAllOverrideMethods()`方法，分别为详细类型定义和导入类型定义的某个方法查找在它的后代类中重定义这个方法的所有方法构成的列表。在解析方法调用（类`MethodReference`的实例）时，由于面向对象语言的动态多态性，它可能调用一个方法在后代类的重定义方法。为确定一个方法调用所有可能调用的方法，需要利用类`SystemScope`的这两个`getAllOverrideMethods()`方法。对于详细类型定义的方法，通过在所有详细类型定义构成的列表中查找该详细类型定义的后代类（利用类`isSubtypeOf()`的方法确定后代类关系）中是否有重定义方法（利用类`MethodDefinition`的方法`isOverrideMethod()`判断两个方法定义是否是互为重定义的方法，即两个方法的返回类型、简单名、参数个数与对应类型都相同）。实现这个方法是在系统作用域中缓存所有详细类型定义构成的列表的主要原因之一，但目前没有缓存类（接口）之间的继承结构，因此每次都要确定后代类关系，因此实现效率可能仍比较低。对于导入类型定义的方法，则仅仅在系统作用域所存放的导入类型列表中查找重定义的方法。由于我们支持利用外部文件补充导入类型的信息，从而也可根据这些信息确定后代类关系和方法重定义信息。

最后，类`SystemScope`提供静态方法`getRootScope()`，对于任意作用域都可得到它的根，即它所在的系统作用域，这是通过接口`NameScope`的方法`getEnclosingScope()`遍历作用域的父作用域（包含它的作用域），直到不存在父作用域的作用域，则是系统作用域。

5.1.3 类`CompilationUnitScope`

类`CompilationUnitScope`在名字表中代表一个编译单元作用域，也即一个编译单元（一个Java源代码文件）的所有源代码构成的作用域。一个系统作用域包含多个程序包定义，每个程序包定义又包含多个编译单元作用域。

表5.4给出了类`CompilationUnitScope`的字段及说明，要注意的是，编译单元名是源代码文件的单元名，也即源代码文件包含路径的全名中去掉源代码文件集起始路径之后剩下的部分。另外，字段`referenceList`只存放直接在编译单元作用域中出现的名字引用，只在同时生成名字定义和名字引用时有用，而且通常也是空表。

表5.5给出了类`CompilationUnitScope`的方法及说明。除实现接口`NameScope`规定的方法外，还

表 5.4 类CompilationUnitScope字段的描述

字段	类型	描述
enclosingPackage	PackageDefinition	编译单元所属的程序包定义
endLocation	SourceCodeLocation	编译单元的结束位置(源代码文件最末尾)
importedStaticMemberList	List<NameReference>	为编译单元的静态导入创建的引用列表
importedTypeList	List<NameReference>	为编译单元导入声明创建的引用列表
referenceList	List<NameReference>	编译单元中出现的名字引用构成的列表
startLocation	SourceCodeLocation	编译单元的起始位置(源代码文件最开头)
typeList	List<TypeDefintion>	编译单元中的顶层类型定义列表
unitName	String	编译单元名, 对应源代码文件的单元名

提供了所声明字段的必要的getter方法, 以及各列表字段的添加元素方法。类CompilationUnitScope还实现接口Comparator<CompilationUnitScope>以支持按照编译单元名顺序存放在Java容器类实例中, 相应地也重定义了equals(), hashCode()和compareTo()方法。

方法bindImportDeclaration()用于将静态导入声明和导入声明所创建的名字引用绑定到合适的静态成员定义或导入类型定义。在各编译单元存放对静态成员定义和导入类型定义的引用而非定义本身是因为不同的编译单元可能导入相同的静态成员或类型, 因此我们最终采用在各编译单元存放名字引用(即将导入声明看做是名字引用而非名字定义), 而在系统作用域存放导入静态成员定义和导入类型定义, 这样既能通过这些名字引用以及它们绑定的定义解析在编译单元其他地方对导入静态成员和导入类型的引用, 又能避免同样的导入类型定义或导入静态成员定义在多个编译单元中存储。

方法bindImportDeclaration()目前只实现了对导入类型声明所对应引用的绑定:

(1) 对于单个导入类型声明, 所创建的名字引用是类型引用, 绑定时先确定其中的程序包全名, 然后找名字表是否存在这个程序包的程序包定义: 如果存在, 则说明导入类型的源代码存在于要分析的源代码(不是第三方库提供的类型), 则调用类PackageDefinition的方法matchTypeWithReference()在该程序包定义中匹配类型引用, 以确定要绑定的详细类型定义; 如果不存在, 则说明没有导入类型的源代码(是第三方库提供的类型), 则在系统作用域解析该类型引用, 也即在系统作用域存放的导入类型定义列表中匹配该类型引用, 如果匹配不成功, 则会创建一个导入类型定义存放在系统作用域, 并将该引用绑定到新创建的导入类型定义。

(2) 对于按需导入类型声明, 所创建的名字引用是程序包引用, 这时在名字表中查找是否存在对应的程序包定义, 如果存在则绑定到该程序包定义, 否则绑定不成功, 也即按需导入类型声明不能绑定到第三方库提供的程序包, 因为目前没有为第三方库提供的程序包创建相应的程序包定义。

方法match()会在上述绑定过程中由PackageDefinition的方法matchTypeWithReference()调用, 在某个程序包定义的编译单元中匹配为导入声明创建的类型引用。这个方法最终仍然是调用类NameDefinition的match()方法进行匹配。之所以没有使用resolve()方法进行解析, 而是直接匹配, 是因为解析的流程总是在当前作用域匹配不成功之后再在父作用域匹配, 而在绑定导入声明时无需如此。

5.1.4 类LocalScope

类LocalScope代表局部作用域, 即使用花括号括起来的一些语句构成的语句块。类的初始化

表 5.5 类CompilationUnitScope方法的描述

<code>CompilationUnitScope(String, PackageDefinition, SourceCodeLocation, SourceCodeLocation)</code>	构造方法，以编译单元名、所属程序包定义、起始位置和结束位置为参数
<code>accept(NameTableVisitor) : void</code> [实现接口NameScope的方法]	接受名字表访问器的访问
<code>addImportedStaticMemberReference(NameReference) : void</code>	添加静态导入声明所对应的名字引用
<code>addImportedTypeReference(NameReference) : void</code>	添加导入声明所对应的名字引用
<code>addReference(NameReference) : void</code> [实现接口NameScope的方法]	添加直接出现在编译单元中的名字引用（不含导入声明和静态导入声明）
<code>bindImportDeclaration() : void</code>	将对应导入声明的名字引用和对应静态导入声明的名字引用进行解析
<code>compareTo(CompilationUnitScope) : int</code>	实现两个编译单元作用域的比较，以便在容器类实例中可按单元名的顺序有序存放
<code>containsLocation(SourceCodeLocation) : boolean</code> [实现接口NameScope的方法]	判断是否包含某源代码位置
<code>define(NameDefinition) : void</code> [实现接口NameScope的方法]	在编译单元作用域中定义名字（即顶层类型所对应的详细类型定义或枚举类型定义）
<code>equals(Object) : boolean</code>	判断两个编译单元是否相同，只要编译单元名相同则相同
<code>getEnclosingPackage() : PackageDefinition</code>	返回所属的程序包定义
<code>getEnclosingScope() : NameScope</code> [实现接口NameScope的方法]	返回所属的作用域，也即返回所属的程序包定义
<code>getImportedStaticMemberList() : List<NameReference></code>	返回为静态导入声明所创建的名字引用列表
<code>getImportedTypeList() : List<NameReference></code>	返回为导入声明所创建的名字引用列表
<code>getReferenceList() : List<NameReference></code> [实现接口NameScope的方法]	返回直接出现在编译单元的名字引用构成的列表
<code>getScopeEnd() : SourceCodeLocation</code> [实现接口NameScope的方法]	返回编译单元的结束位置
<code>getScopeKind() : NameScopeKind</code> [实现接口NameScope的方法]	返回作用域类别，即NSK_COMPILATION_UNIT
<code>getScopeName() : String</code> [实现接口NameScope的方法]	返回作用域名字，也即返回编译单元名
<code>getScopeStart() : SourceCodeLocation</code> [实现接口NameScope的方法]	返回编译单元的起始位置
<code>getSubScopeList() : List<NameScope></code> [实现接口NameScope的方法]	返回子作用域，也即定义在编译单元中的所有顶层类型（详细类型或枚举类型）定义构成的列表
<code>getTypeList() : List<TypeDefintion></code>	返回顶层类型定义构成的列表
<code>getUnitName() : String</code>	返回编译单元名
<code>hashCode() : int</code>	用于存放在Java容器类实例时的散列函数
<code>isEnclosedInScope(NameScope) : boolean</code> [实现接口NameScope的方法]	判断是否包含在某作用域中
<code>match(NameReference) : boolean</code>	在编译单元中匹配顶层类型的简单名或全限定名与一个名字引用中的名字
<code>resolve(NameReference) : boolean</code> [实现接口NameScope的方法]	在编译单元作用域中解析名字引用，主要是匹配顶层类型定义

块、方法体等都是语句块，都确定了一个局部作用域。同样，分支、循环等多种语句中的块语句也可能确定一个局部作用域，但在创建名字定义和名字作用域时，只有当一个语句块中有名字定义时才会真正创建一个类LocalScope的对象。

表 5.6 类LocalScope字段的描述

字段	类型	描述
enclosingScope	NameScope	包含当前局部作用域的父作用域
endLocation	SourceCodeLocation	局部作用域的结束位置
localTypeList	List<DetailedTypeDefinition>	声明在局部作用域中的局部类型列表
referenceList	List<NameReference>	出现在局部作用域中的引用列表
startLocation	SourceCodeLocation	局部作用域的起始位置
subLocalscopeList	List<LocalScope>	被当前局部作用域包含的子局部作用域
variableList	List<VariableDefinition>	声明在当前局部作用域的变量定义列表

表5.6给出类LocalScope的字段及说明，表5.7给出类LocalScope的方法及说明。类LocalScope的字段和方法含义都比较清楚，无需太多说明。

局部作用域内能定义的名字有局部变量（我们用类VariableDefinition的实例表示）和局部类型（我们用类DetailedTypeDefinition的实例表示），因此方法define()支持在局部作用域内定义这两种类别的名字，分别在往字段variableList和localTypeList添加元素，相应地，方法reslove()在局部作用域内匹配名字引用保存的名字（通常是简单名）与这两种类别的名字定义。匿名类（类AnonymousClassDefinition的实例）也可能出现在局部作用域中，它被看做是特别的详细类型定义，因此也保存在字段localTypeList中。

局部作用域的子作用域可能包括嵌套在其中的局部作用域以及局部类型（包含匿名类）。只有那些内部有变量声明的块语句才会创建局部作用域，而且是只为包含变量声明的最内层块语句创建相应的局部作用域。

5.2 名字定义

所有代表名字定义的类都在程序包nameTable.nameDefinition中，类NameDefinition是这些类的基类。图5.2给出目前实现的代表名字定义的类及其它们之间的继承层次结构。

- nameTable.nameDefinition.NameDefinition (implements java.lang.Comparable<T>)
 - nameTable.nameDefinition.EnumConstantDefinition
 - nameTable.nameDefinition.FieldDefinition
 - nameTable.nameDefinition.ImportedStaticMemberDefinition
 - nameTable.nameDefinition.MethodDefinition (implements nameTable.nameScope.NameScope)
 - nameTable.nameDefinition.AutoGeneratedConstructor
 - nameTable.nameDefinition.PackageDefinition (implements nameTable.nameScope.NameScope)
 - nameTable.nameDefinition.TypeDefinition
 - nameTable.nameDefinition.DetailedTypeDefinition (implements nameTable.nameScope.NameScope)
 - nameTable.nameDefinition.AnonymousClassDefinition
 - nameTable.nameDefinition.EnumTypeDefinition (implements nameTable.nameScope.NameScope)
 - nameTable.nameDefinition.ImportedTypeDefinition (implements nameTable.nameScope.NameScope)
 - nameTable.nameDefinition.TypeParameterDefinition
 - nameTable.nameDefinition.VariableDefinition

图 5.2 代表名字定义的类的继承层次结构

在实现时我们在前面设计基础上增加了类AnonymousClassDefinition和类AutoGeneratedConstructor，前者代表匿名类，被看做特殊的详细类型定义，是类DetailedTypeDefinition的直接子类，后者表

表 5.7 类LocalScope方法的描述

<code>LocalScope(NameScope, SourceCodeLocation, SourceCodeLocation) : Constructor</code>	构造方法，以父作用域、起始位置和结束位置为参数
<code>accept(NameTableVisitor) : void [实现接口NameScope的方法]</code>	接受名字表访问器的访问
<code>addReference(NameReference) : void [实现接口NameScope的方法]</code>	添加出现在局部作用域的名字引用
<code>addSubLocalScope(LocalScope) : void</code>	添加局部作用域所包含的子局部作用域
<code>containsLocation(SourceCodeLocation) : boolean [实现接口NameScope的方法]</code>	判断是否包含某源程序位置
<code>define(NameDefinition) : void [实现接口NameScope的方法]</code>	在局部作用域中定义名字：局部类型定义或变量定义
<code>getAllLocalVariableDefinitions() : List<VariableDefinition></code>	返回局部作用域及其子局部作用域包含的所有变量定义
<code>getEnclosingScope() : NameScope [实现接口NameScope的方法]</code>	返回父作用域
<code>getLocalTypeList() : List<DetailedTypeDefinition></code>	返回局部类型定义列表
<code>getReferenceList() : List<NameReference> [实现接口NameScope的方法]</code>	返回引用列表
<code>getScopeEnd() : SourceCodeLocation [实现接口NameScope的方法]</code>	返回局部作用域结束位置
<code>getScopeKind() : NameScopeKind [实现接口NameScope的方法]</code>	返回作用域类别，即NSK_LOCAL
<code>getScopeName() : String [实现接口NameScope的方法]</code>	返回局部作用域的名字，是一个根据起始位置自动生成的名字
<code>getScopeStart() : SourceCodeLocation [实现接口NameScope的方法]</code>	返回局部作用域的起始位置
<code>getSubLocalScope() : List<LocalScope></code>	返回局部作用域所包含的子局部作用域列表
<code>getSubScopeList() : List<NameScope> [实现接口NameScope的方法]</code>	返回局部作用域的子作用域列表，包括子局部作用域以及局部类型定义
<code>getVariableList() : List<VariableDefinition></code>	返回直接包含在局部作用域的变量定义
<code>isEnclosedInScope(NameScope) : boolean [实现接口NameScope的方法]</code>	判断是否真包含在某个作用域内
<code>resolve(NameReference) : boolean [实现接口NameScope的方法]</code>	在局部作用域中解析名字引用，主要是匹配看是否是局部作用域中的变量或局部类型
<code>setEnclosingScope(NameScope) : void</code>	设置包含局部作用域的作用域（即设置父作用域）

示自动生成的构造方法（当类没有显式定义构造方法时意味着自动有一个无参数的缺省构造函数），前者被看做特殊的方法，是类MethodDefinition的直接子类。

5.2.1 类NameDefinition

类NameDefinition是所有代表名字定义的类的基类，给出了名字定义的一般特性，这些特性表现为这个类的字段，如表5.8所示，有名字定义的简单名、全限定名、源代码位置和所在作用域。除程序包定义的简单名和全限定名相同并都可能带有圆点(".")之外，其他名字定义的简单名都不含圆点。当名字定义也同时是作用域时，它就不仅有源代码起始位置，还有结束位置，这时字段location存放的是起始位置，结束位置将在实现作用域接口(NameScope)的名字定义类中另外使用字段存放。注意，**不是所有名字定义都有源代码位置**，没有为导入类型定义（类ImportedTypeDefinition的实例）提供额外信息时，导入类型定义没有源代码位置，这时它的location字段为null。

表 5.8 类NameDefinition字段的描述

字段	类型	描述
LOCATION_ID_BEGINNER	char	字符常量，用于标记名字定义唯一标识中源代码位置的开始，值为'@'
fullQualifiedName	String	名字定义的全限定名
location	SourceCodeLocation	名字定义的位置
scope	NameScope	名字定义所在的作用域
simpleName	String	名字定义的简单名

表5.9给出类NameDefinition的方法及说明。类NameDefinition实现接口Comparable<NameDefinition>以便将名字定义放在容器类实例时能够有序存放。我们为每个名字定义都确定一个唯一标识：

- (1) 当名字定义有源代码位置时，它的唯一标识由简单名和源代码位置（行号+列号+单元名）组成（中间有特定的分隔符）；
- (2) 当名字定义没有源代码位置时，它的唯一标识就是它的全限定名。

我们约束**只有在名字表创建阶段才设置名字定义和名字作用域的各种属性字段**，也即名字表创建完毕后，**名字定义和名字作用域对象实例都不再变化**，因此类NameDefinition在实现方法equals()时使用的是浅比较，即直接比较两个对象是否相等，而没有根据对象的属性字段进行比较。但在实现方法hashCode()和compareTo()时是基于构成名字定义唯一标识的成分计算散列值和比较结果，这样当名字定义在Java容器类实例中有序存放时是按照名字定义唯一标识的字典顺序存放。

类NameDefinition提供两个静态方法从一个被认为是名字定义唯一标识的字符串中提取可能的简单名或全限定名，以及标识源代码位置的字符串（这个字符串又可利用类SourceCodeLocation的静态方法getLocation()得到一个源代码位置对象，即类SourceCodeLocation的实例）。

类NameDefinition没有设置字段存放名字定义类别，由各种具体的名字定义类重定义方法getDefinitionKind() 确定名字定义的类别。为方便判定名字定义类别，类NameDefinition提供了一系列的is...()方法判断当前名字定义是否属于详细类型定义、枚举类型、字段定义、方法定义、类型定义、变量定义等等。

表 5.9 类NameDefinition方法的描述

<code>getDefinitionLocationStringFromId(String) : String</code>	静态方法，从名字定义唯一标识提取表示源代码位置的串
<code>getDefinitionNameFromId(String) : String</code>	静态方法，从名字定义唯一标识提取名字定义的简单名或全限定名
<code>NameDefinition(String, String, SourceCodeLocation, NameScope) : Constructor</code>	构造方法，以名字定义的简单名、全限定名、源代码位置和所在作用域为参数
<code>compareTo(NameDefinition) : int</code> [实现接口Compare<NameDefinition>的方法]	与名字定义进行比较，以便名字定义可有序地存放在Java容器类实例中
<code>equals(Object) : boolean</code>	名字定义一旦创建就不会再改变，因此可直接使用浅比较判断两个名字定义是否相同
<code>getDefinitionKind() : NameDefinitionKind</code>	返回名字定义的类别，这是一个抽象方法，由具体的名字定义实现
<code>getFullQualifiedName() : String</code>	返回名字定义的全限定名
<code>getLocation() : SourceCodeLocation</code>	返回名字定义的源代码位置
<code>getScope() : NameScope</code>	返回名字定义所在的作用域
<code>get SimpleName() : String</code>	返回名字定义的简单名
<code>getUniqueId() : String</code>	返回名字定义的唯一标识，当名字定义的源代码位置不为null时，唯一标识由简单名和源代码位置确定，否则只由全限定名确定
<code>hashCode() : int</code>	基于名字定义的唯一标识计算散列值
<code>isDetailedType() : boolean</code>	
<code>isEnumType() : boolean</code>	
<code>isFieldDefinition() : boolean</code>	
<code>isImportedStaticMember() : boolean</code>	
<code>isImportedType() : boolean</code>	
<code>isMethodDefinition() : boolean</code>	
<code>isTypeDefintion() : boolean</code>	
<code>isVariableDefinition() : boolean</code>	提供一些更直接的方法来判断一个名字定义是否是相应类别的名字定义，在类NameDefinition都返回false，由相应的类进行重定义（而返回true）
<code>match(NameReference) : boolean</code>	将名字定义与一个名字引用进行匹配，调用下面的matchString域名字引用中存放的名字串进行匹配，匹配成功将名字引用绑定到当前名字定义
<code>match(String) : boolean</code>	将名字定义与一个字符串进行匹配，当字符串含有圆点"."时比较全限定名与该字符串，否则比较简单名与该字符串，如果相等返回true
<code>toFullString() : String</code>	返回一个记录名字定义详细信息的字符串，包含名字定义类别、全限定名、简单名、所在作用域、源代码位置等信息
<code>toString() : String</code>	返回一个记录名字定义简略信息的字符串，只包含名字定义类别、名字定义唯一标识等信息

在名字引用解析过程中，最后将名字引用中保存的名字与名字定义中的简单名或全限定名匹配都是通过类NameDefinition 的方法match()完成，当名字引用中保存的名字含有圆点时，它基于名字定义的全限定名进行精确比较，否则基于名字定义的简单名进行精确比较。

类NameDefinition提供了两个方法toString()和toFullString()将一个名字定义转换为字符串，前者给出的信息比较简单，而后者给出的信息比较全，这些方法主要是在调试时用于输出名字定义的信息。

5.2.2 类TypeDefinition

类TypeDefinition是类NameDefinition的直接子类，是所有代表类型定义的类的基类。类型定义是类型引用的绑定对象，类型引用出现在变量声明、参数声明、类的超类声明、方法抛出的异常类型等许多地方。我们目前将Java程序中的类型定义分为四类：

(1) 详细类型定义（类DetailedTypeDefinition的实例）：指在所分析的程序中有源代码的类或接口类型，从而能得到该类型的字段、方法、内部类型等的详细信息；

(2) 枚举类型定义（类EnumTypeDefinition的实例）：虽然在Java规范中枚举被认为是特殊的类，但由于在目前的实际编程实践中还很少在枚举类型中声明方法，因此我们仍将其单独处理。枚举类型定义内部主要是一系列的枚举常量；

(3) 导入类型定义（类ImportedTypeDefinition的实例）：指在所分析的程序中没有源代码，但被所分析的程序导入的类型（类、接口或枚举）。我们目前支持使用额外的文件给出其中部分导入类型的部分信息，如部分字段的类型、部分方法的基调信息等帮助我们对所要分析的程序的理解；

(4) 类型参数定义（类ParameterTypeDefinition的实例）：指在声明类属类型（或说泛型类型，generic types）时给出的类型参数。类型参数也定义一个类型，可用于类属类型内部字段类型、方法参数类型、返回类型的声明。我们目前设计了这个类来标识类型参数定义，但在名字解析时还没有考虑如何利用类型参数以更精确地推导表达式（例如方法实际参数）的类型。

类TypeDefinition给出这些类型定义的一些共有的属性和职责（方法）。表5.10 给出类TypeDefinition的字段，除继承类NameDefinition的字段外使用两个布尔类型标记区分类型定义是否是接口，以及是否是顶层类型（即程序包成员类型）。

表 5.10 类TypeDefinition字段的描述

字段	类型	描述
isInterface	boolean	标记这个类型是否是接口(true)还是类(false)
isPackageMember	boolean	标记这个类型是否是顶层类型（程序包成员）
继承自类NameDefinition的字段：参见表5.8		
fullQualifiedName, SourceCodeLocation, scope, simpleName		

表5.11给出类TypeDefinition的方法及说明。除返回与设置两个字段isInterface和isPackageMember外的方法主要为具体的类型定义类提供缺省实现，包括返回类型所属程序包定义(getEnclosingPackage())、返回超类定义(getSuperClassDefinition())、返回超类型引用列表(getSuperList())和判断子类型关系(isSubtypeOf())。具体的类型定义类（主要是详细类型定义）重定义这些方法，给出更准确的实现。

类TypeDefinition的静态方法matchSubtypeRelationsOfPrimitiveTypes()用于判定两个原子类型(指int, double等)之间是否有子类型关系。方法isSubtypeOf()基于方法getSuperList()得到的超类型列表实现子类型关系的判断,当参数给出的类型定义是超类型列表的某个类型引用绑定的类型定义返回true,否则对于超类型列表中的每个类型引用绑定的类型定义递归调用方法isSubtypeOf()方法判断超类型定义与参数给定的类型定义是否有子类型关系,如果有则返回true。

表 5.11 类TypeDefinition方法的描述

<code>matchSubtypeRelationsOfPrimitiveTypes(String, String) : boolean</code>	静态受保护方法,判断第一个参数代表的基本类型(例如"int")是否是第二个参数代表的基本类型(例如"double")的子类型。
<code>TypeDefinition(String, String, SourceCodeLocation, NameScope) : Constructor</code>	构造方法,以类型简单名、全限定名、源代码位置、所在作用域为参数
<code>getDefinitionKind() : NameDefinitionKind [重定义类NameDefinition的方法]</code>	返回NameDefinitionKind.NDK_TYPE
<code>getEnclosingPackage() : PackageDefinition</code>	返回类型所属的程序包定义,缺省的实现是返回null
<code>getSuperClassDefinition() : TypeDefinition</code>	若当前类型是类类型,返回它的直接超类对应的类型定义,如果没有(即没有extends声明)则返回null,在TypeDefinition的缺省实现是返回null
<code>getSuperList() : List<TypeReference></code>	返回直接超类型(含接口和类)列表,列表元素是类型引用(TypeReference)(而不是绑定的类型定义),在TypeDefinition的缺省实现是返回null
<code>isInterface() : boolean</code>	判断这个类型是否是接口
<code>isPackageMember() : boolean</code>	判断这个类型是否是顶层类型(程序包成员)
<code>isPublic() : boolean</code>	判断这个类型是否是公有类型,缺省实现是返回true
<code>isSubtypeOf(TypeDefinition) : boolean</code>	判断这个类型是否是给定类型的子类型
<code>resolve(NameReference) : boolean</code>	在这个类型定义中解析名字引用。虽然一个类型定义不一定对应一个作用域(例如没有源代码和额外信息的导入类型)的,但在详细类型定义、枚举类型定义中解析名字引用非常常用,为避免过多的类型转换,所以直接在它们的基类设置这个方法
<code>setInterface(boolean) : void</code>	设置这个类型定义是否是接口
<code>setPackageMember(boolean) : void</code>	设置这个类型定义是否是顶层类型
继承自类NameDefinition的方法(不含已被重定义方法): 参见表5.9	
<code>compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int</code>	
<code>getFullQualifiedName():String; getSimpleName():String; getUniqueId():String</code>	
<code>getLocation():SourceCodeLocation; getScope():NameScope;</code>	
<code>isDetailedType():boolean; isEnumType():boolean; isTypeDefinition():boolean</code>	
<code>isImportedType():boolean; isImportedStaticMember():boolean</code>	
<code>isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean</code>	
<code>match(NameReference):boolean; match(String):boolean</code>	
<code>toFullString():String; toString():String</code>	

类TypeDefinition不实现接口NameScope，但也提供了一个resolve()方法，主要是因为在解析名字引用组时，经常会先解析一个子表达式得到一个类型定义，然后再在这个类型定义中解析其他的子表达式。若不在类TypeDefinition中设计resolve()方法，则在这种情况下需要将类型定义（类TypeDefinition的实例）转换为详细类型定义（类DetailedTypeDefinition的实例）等才能进行其他子表达式的解析。类TypeDefinition本身提供方法resolve()就可避免这样的类型转换，从而简化名字引用解析的实现。

5.2.3 类DetailedTypeDefinition与类AnonymousClassDefinition

类DetailedTypeDefinition是类TypeDefinition的直接子类，并实现接口NameScope。它的实例代表一个有详细信息的类型定义，包括类类型和接口类型。所谓有详细信息是指在所分析的程序中有源代码声明这个类型，从而有关于类型的字段类型、方法基调与方法实现、内部类声明等信息。实际上，对一个Java软件的分析和理解主要是指对这些类型的分析和理解。

表5.12给出类DetailedTypeDefinition的字段，除继承自类NameDefinition和类TypeDefinition的字段外，其他字段给出一个详细类型定义的主要成员包括字段列表、初始化块列表、方法列表、超类型列表、内部类列表和类型参数列表。为支持同时生成名字定义和名字引用，作为作用域，还有存放直接出现在类型作用域中的名字引用构成的列表。作为作用域，字段endLocation给出了类型声明的源代码结束位置，即类型声明的右花括号所在位置作为作用域的结束位置。

表 5.12 类DetailedTypeDefinition字段的描述

字段	类型	描述
endLocation	SourceCodeLocation	声明类型的源代码结束位置
fieldList	List<FieldDefinition>	类型声明的字段类别
initializerList	List<LocalScope>	类型的初始化块列表
methodList	List<MethodDefinition>	类型声明的方法列表
modifier	int	类型的修饰（表示公有、终态、静态等等整数标志）
referenceList	List<NameReference>	出现在类型声明中的引用列表
superList	List<TypeReference>	超类型引用列表
typeList	List<DetailedTypeDefinition>	在类型中声明的内部类列表
typeParameterList	List<TypeParameterDefinition>	类型的类型参数列表
继承自类TypeDefinition的字段：参见表5.10		
isInterface, isPackageMember		
继承自类NameDefinition的字段：参见表5.8		
fullQualifiedName, SourceCodeLocation, scope, simpleName		

表5.13和表5.14给出类DetailedTypeDefinition的方法及描述。虽然这个类的方法比较多，但大致可分为四类：(1) 实现接口NameScope规定的方法；(2) 类DetailedTypeDefinition中字段的getter和setter方法；(3) 用于判断类型的修饰符和类别的一些is...()方法；(4) 继承自类TypeDefinition和NameDefinition的方法。

作为作用域，详细类型定义表示类型作用域，可能定义的名字包括字段定义、方法定义、局部类型定义和类型参数定义，因此相应字段（即字段fieldList, methodList, typeList和typeParameterList）没有提供setter方法（或添加元素到列表的方法），这些字段是通过方法define()定义相应类别的名字定义而添加元素。

表 5.13 类DetailedTypeDefinition方法的描述

DetailedTypeDefinition(String, String, SourceCodeLocation, NameScope, SourceCodeLocation) : Constructor	构造方法，以类型简单名、全限定名、源代码起始位置、所在作用域、源代码结束位置为参数
accept(NameTableVisitor) : void [实现接口NameScope的方法]	接受名字表访问器的访问
addInitializer(LocalScope) : boolean	添加初始化块到初始化列表
addReference(NameReference) : void [实现接口NameScope的方法]	添加名字引用到名字引用列表
addSuperType(TypeReference) : void	添加超类型引用到超类型引用列表
containsLocation(SourceCodeLocation) : boolean [实现接口NameScope的方法]	判断是否包含指定的源代码位置
define(NameDefinition) : void [实现接口NameScope的方法]	在类型定义中定义名字，可能定义的名字包括字段定义、方法定义、(内部)类型定义
getEnclosingPackage() : PackageDefinition [重定义类TypeDefinition的方法]	返回包含当前详细类型定义的程序包
getEnclosingScope() : NameScope [实现接口NameScope的方法]	返回包含当前详细类型定义的作用域
getEndLocation() : SourceCodeLocation	返回声明当前详细类型的源代码结束位置
getFieldList() : List<FieldDefinition>	返回在当前详细类型定义中声明的字段列表
getInitializerList() : List<LocalScope>	返回当前详细类型的初始化块列表
getMethodList() : List<MethodDefinition>	返回在当前详细类型定义中声明的方法列表
getReferenceList() : List<NameReference> [实现接口NameScope的方法]	返回出现在当前详细类型定义中的名字引用列表
getScopeEnd() : SourceCodeLocation [实现接口NameScope的方法]	返回当前详细类型定义作为作用域的结束位置，等于getEndLocation()的返回值
getScopeKind() : NameScopeKind [实现接口NameScope的方法]	返回NameScopeKind.NSK_TYPE
getScopeName() : String [实现接口NameScope的方法]	返回详细类型定义的简单名作为作用域名字
getScopeStart() : SourceCodeLocation [实现接口NameScope的方法]	返回当前详细类型定义作为作用域的起始位置，等于getLocation()的返回值
getSubScopeList() : List<NameScope> [实现接口NameScope的方法]	返回当前详细类型定义包含的子作用域列表，包括初始化块、内部类型定义和方法定义
getSuperClassDefinition() : TypeDefinition [重定义类TypeDefinition的方法]	如果当前类型是类类型且有extends声明，则返回extends后声明的超类型引用所绑定的类型定义，否则返回null
getSuperList() : List<TypeReference> [重定义类TypeDefinition的方法]	返回超类型引用列表
getTypeList() : List<DetailedTypeDefinition>	返回内部类型列表
getTypeParameterList() : List<TypeParameterDefinition>	返回类型参数列表

表 5.14 类DetailedTypeDefinition方法的描述（续）

<code>isAbstract() : boolean</code>	判断当前详细类型定义是否是抽象类型
<code>isAnonymous() : boolean</code>	判断当前详细类型定义是否是匿名类
<code>isDetailedType() : boolean</code> [重定义类NameDefinition的方法]	
返回true	
<code>isEnclosedInScope(NameScope) : boolean</code> [实现接口NameScope的方法]	判断当前详细类型定义是否包含在给定作用域内
<code>isEnumType() : boolean</code> [重定义类NameDefinition的方法]	
返回false	
<code>isFinal() : boolean</code>	判断当前详细类型定义是否是终态类型
<code>isPrivate() : boolean</code>	判断当前详细类型定义是否是私有类型
<code>isProtected() : boolean</code>	判断当前详细类型定义是否是受保护类型
<code>isPublic() : boolean</code> [重定义类TypeDefinition的方法]	判断当前详细类型定义是否是公有类型
<code>isStatic() : boolean</code>	判断当前详细类型定义是否是静态类型
<code>resolve(NameReference) : boolean</code> [重定义类TypeDefinition的方法]	在当前详细类型定义中解析名字引用，主要是与字段、方法和内部类型匹配
<code>setModifierFlag(int) : void</code>	设置当前详细类型的修饰符标记
继承自类TypeDefinition的方法（不含已被重定义方法）：参见表5.11	
<code>getDefinitionKind():NameDefinitionKind; isInterface():boolean; isPackageMember():boolean</code>	
<code>isSubtypeOf(TypeDefinition):boolean; setInterface(boolean):void; setPackageMember(boolean):void</code>	
继承自类NameDefinition的方法（不含已被重定义方法）：参见表5.9	
<code>compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int</code>	
<code>getFullQualifiedName():String; getSimpleName():String; getUniqueId():String</code>	
<code>getLocation():SourceCodeLocation; getScope():NameScope;</code>	
<code>isTypeDefinition():boolean; isImportedType():boolean; isImportedStaticMember():boolean</code>	
<code>isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean</code>	
<code>match(NameReference):boolean; match(String):boolean</code>	
<code>toFullString():String; toString():String</code>	

对应地，在详细类型定义中解析名字引用的方法`resolve()`也主要是匹配在该作用域中的字段、方法、局部类型和类型参数，其中对字段、局部类型和类型参数的匹配都是利用类`NameDefinition`的`match()`方法进行匹配，但方法的匹配除了方法（简单）名的匹配之外，还要进行参数匹配，因此是调用类`MethodDefinition`的`matchMethod()`方法进行匹配，而且当匹配成功之后，考虑到方法调用的多态性，还将匹配成功的方法定义的所有在子类中重定义的方法作为候选方法列表记录在要解析的方法引用中。

如果在详细类型定义所代表的类型作用域中匹配名字引用不成功, `resolve()`会进一步在该详细类型定义的超类型列表的每个超类型所绑定的类型定义中解析该名字引用, 如果仍不成功则当该名字引用出现在当前详细类型定义(即名字引用所在作用域是当前详细类型定义或被真包含在当前详细类型定义)时继续在当前详细类型定义的父作用域中解析该名字引用。注意, 当该名字引用出现在当前详细类型定义的子类型定义时也会在当前详细类型定义中解析该名字引用, 但这时不能在当前详细类型定义的父作用域中解析该名字引用, 因为这时我们只能检查该名字引用是否是引用超类型定义的成员, 该名字引用本身不能引用超类型的父作用域(例如包含当前超类型定义的编译单元)中定义的名字(例如在编译单元中导入的类型定义)。

详细类型定义可能包含的子作用域包括初始化块对应的局部作用域、方法定义对应的方法作用域，以及局部类型定义对应的类型作用域。

类DetailedTypeDefinition的is...()方法主要是利用Eclipse JDT提供的Modifier类基于整数类型字段modifier存放的修饰符判断当前详细类型定义是否是公有、抽象、静态、终态、受保护、私有类型等。另外方法isAnonymous()判断当前详细类型定义是否是匿名类，我们将匿名类看做是特殊的详细类型定义。

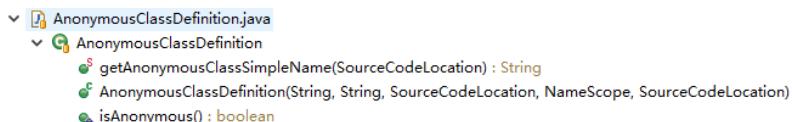


图 5.3 类AnonymousClassDefinition的成员

类AnonymousClassDefinition的实例代表一个匿名类，是类DetailedTypeDefinition的直接子类。图5.3给出它的成员信息。这个类本身十分简单，没有增加额外字段。类AnonymousClassDefinition的构造方法的参数与类DetailedTypeDefinition的构造方法参数相同，以简单名、全限定名、起始位置、所在作用域和结束位置为参数。方法isAnonymous()方法返回true表明这个类的实例是匿名类。方法getAnonymousClassSimpleName()是公有静态方法，基于源代码位置自动生成一个内部名字作为匿名类的简单名。

5.2.4 类EnumTypeDefinition与类EnumConstantDefinition

类EnumTypeDefinition的实例代表一个枚举类型。与我们在设计时有所差别，实现时我们让它作为类TypeDefinition的直接子类，而不是作为DetailedTypeDefinition的直接子类，因为在目前的Java语言编程实践中，枚举类型主要是定义一系列枚举常量，很少像其他类一样定义字段和方法等等。

表5.15给出类EnumTypeDefinition的字段及描述。表5.16和表5.17给出它的方法及描述。类EnumTypeDefinition的方法主要是实现接口NameScope的方法、设置和方法字段setter和getter方

法, 以及继承自类TypeDefinition和NameDefinition的方法。

表 5.15 类EnumTypeDefinition字段的描述

字段	类型	描述
constantList	List<EnumConstantDefinition>	包含的枚举常量定义列表
endLocation	SourceCodeLocation	声明类型的源代码结束位置
modifier	int	类型的修饰(表示公有、终态、静态等等整数标志)
referenceList	List<NameReference>	出现在类型声明中的引用列表
superList	List<TypeReference>	超类型引用列表
继承自类TypeDefinition的字段: 参见表5.10		
isInterface, isPackageMember		
继承自类NameDefinition的字段: 参见表5.8		
fullQualifiedName, SourceCodeLocation, scope, simpleName		

作为作用域, 枚举类型能定义的名字目前只支持枚举常量, 每个枚举常量是类EnumConstantDefinition的实例, 通过方法define()在枚举类型定义中增加枚举常量。对应地, 解析名字引用的方法resolve()也只是在枚举类型中匹配枚举常量的名字。枚举类型定义目前没有子作用域, 因此方法getSubScopeList()目前返回null。

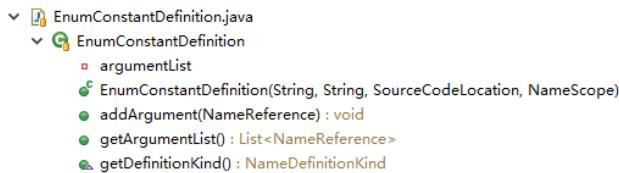


图 5.4 类EnumConstantDefinition的成员

类EnumConstantDefinition的实例代表一个枚举常量, 它是类NameDefinition的直接子类, 图5.4给出它的成员。这个类目前的实现很简单, 有元素类型为NameReference的列表字段argumentList记录构造枚举常量时的实际参数(但目前枚举类型定义中没有记录构造方法, 因此这些参数目前暂时没有进行检查与匹配)。枚举常量的构造方法以简单名、全限定名(即所在枚举类型的全限定名加枚举常量的简单名)、源代码位置和所在作用域为参数。实际上, 目前我们对枚举常量的处理仅仅是认为它提供了一个枚举常量简单名。

5.2.5 类ImportedTypeDefinition与类ImportedStaticMemberDefinition

类ImportedTypeDefinition是类TypeDefinition的直接子类, 它的实例代表一个导入类型定义。导入类型定义对应那些在所要分析的源程序中没有源代码, 但在导入声明中要导入的类型(包括自动导入的系统程序包java.lang中的类型)。

为更好的理解所要分析的源程序, 我们目前支持使用额外文件提供导入类型的更多信息, 包括导入类型有哪些字段、字段的类型, 有哪些方法、方法的基调(参数及其类型、返回类型)等。我们试验过, 只要遵循Java语言抽象类的语法给出导入类型的字段及其类型声明、方法及其基调声明, 就可以使用Eclipse JDT构建抽象语法树的方法为这些导入类型生成抽象语法树, 从而提取相应的信息。也即, 我们可为导入类型编写一个代码骨架, 从而为所要分析的源程序提供更多有关导入类型的信息, 使得所要分析的源程序中名字引用可以更准确地绑定。

表 5.16 类EnumTypeDefinition方法的描述

<code>EnumTypeDefinition(String, String, SourceCodeLocation, NameScope, SourceCodeLocation) : Constructor</code>	构造方法，以类型简单名、全限定名、源代码起始位置、所在作用域、源代码结束位置为参数
<code>accept(NameTableVisitor) : void</code> [实现接口NameScope的方法]	接受名字表访问器的访问
<code>addConstant(EnumConstantDefinition) : void</code>	添加枚举常量定义
<code>addReference(NameReference) : void</code> [实现接口NameScope的方法]	添加名字引用到名字引用列表
<code>addSuperType(TypeReference) : void</code>	添加超类型引用到超类型引用列表
<code>containsLocation(SourceCodeLocation) : boolean</code> [实现接口NameScope的方法]	判断是否包含指定的源代码位置
<code>define(NameDefinition) : void</code> [实现接口NameScope的方法]	在枚举类型定义中定义名字，可能定义的名字只有枚举常量定义
<code>getConstantList() : List<EnumConstantDefinition></code>	返回枚举常量定义列表
<code>getEnclosingPackage() : PackageDefinition</code> [重定义类TypeDefinition的方法]	返回包含当前枚举类型定义的程序包
<code>getEnclosingScope() : NameScope</code> [实现接口NameScope的方法]	返回包含当前枚举类型定义的作用域
<code>getEndLocation() : SourceCodeLocation</code>	返回声明当前枚举类型的源代码结束位置
<code>getScopeEnd() : SourceCodeLocation</code> [实现接口NameScope的方法]	返回当前枚举类型定义作为作用域的结束位置，等于getEndLocation()的返回值
<code>getScopeKind() : NameScopeKind</code> [实现接口NameScope的方法]	返回NameScopeKind.NSK_TYPE
<code>getScopeName() : String</code> [实现接口NameScope的方法]	返回枚举类型定义的简单名作为作用域名字
<code>getScopeStart() : SourceCodeLocation</code> [实现接口NameScope的方法]	返回当前枚举类型定义作为作用域的起始位置，等于getLocation()的返回值
<code>getSubScopeList() : List<NameScope></code> [实现接口NameScope的方法]	枚举类型定义没有子作用域，返回null
<code>getSuperClassDefinition() : TypeDefinition</code> [重定义类TypeDefinition的方法]	如果当前类型是类类型且有extends声明，则返回extends后声明的超类型引用所绑定的类型定义，否则返回null
<code>getSuperList() : List<TypeReference></code> [重定义类TypeDefinition的方法]	返回超类型引用列表
<code>isDetailedType() : boolean</code> [重定义类NameDefinition的方法]	返回false
<code>isEnclosedInScope(NameScope) : boolean</code> [实现接口NameScope的方法]	判断当前枚举类型定义是否包含在给定作用域内
<code>isEnumType() : boolean</code> [重定义类NameDefinition的方法]	返回true
<code>isInterface() : boolean</code>	返回false
<code>isPublic() : boolean</code> [重定义类TypeDefinition的方法]	判断当前枚举类型定义是否是公有类型
<code>resolve(NameReference) : boolean</code> [重定义类TypeDefinition的方法]	在当前详细类型定义中解析名字引用，主要是与字段、方法和内部类型匹配
<code>setModifierFlag(int) : void</code>	设置当前详细类型的修饰符标记

表 5.17 类EnumTypeDefinition方法的描述（续）

继承自类TypeDefinition的方法（不含已被重定义方法）：参见表5.11
getDefinitionKind():NameDefinitionKind; isPackageMember():boolean isSubtypeOf(TypeDefinition):boolean; setInterface(boolean):void; setPackageMember(boolean):void
继承自类NameDefinition的方法（不含已被重定义方法）：参见表5.9
compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int getFullQualifiedName():String; getSimpleName():String; getUniqueId():String getLocation():SourceCodeLocation; getScope():NameScope; isTypeDefinition():boolean; isImportedType():boolean; isImportedStaticMember():boolean isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean match(NameReference):boolean; match(String):boolean toFullString():String; toString():String

```

package java.util;

interface List<E> extends java.util.Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    E get(int index);
}
class ArrayList<E> implements java.util.List<E> {
    ArrayList(int c);
    ArrayList();
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    E get(int index);
}
class TreeSet<E> {
    boolean add(E e);
}
class Set<E> {
    boolean add(E e);
}

```

图 5.5 描述导入类型定义信息的示例

图5.5给出一个描述导入类型定义信息的例子。在这个例子中我们首先声明下面所有的类都属于程序包java.util，由于这不是所要分析程序的源代码，因此这个声明不会在为所要分析程序的名字表生成一个程序包定义（注意，所有的导入类型定义都放在系统作用域）。这个程序包声明只是用于构成下面类型的全限定名的一部分。

在这个例子中我们给出了多个导入类型的信息，每个导入类型的声明像声明一个抽象类（或接口），只是给出了方法的基调而没有给出实现，我们也没有给出每个导入类型的全部方法，只是给出了在所要分析程序中最可能被引用的一些成员。关于导入类型的这些额外信息有助于我们解析所要分析程序中的名字引用，从而对所要分析程序有更多理解。

表 5.18 类ImportedTypeDefinition字段的描述

字段	类型	描述
endLocation	SourceCodeLocation	声明类型的结束位置
fieldList	List<FieldDefinition>	类型声明的字段类别
methodList	List<MethodDefinition>	类型声明的方法列表
superList	List<TypeReference>	超类型引用列表
typeList	List<ImportedTypeDefintion>	在类型中声明的内部类列表
typeParameterList	List<TypeParameterDefinition>	类型的类型参数列表

表5.18给出类ImportedTypeDefinition的字段。由于要支持额外文件提供的有关导入类型的信息，因此它也有fieldList, methodList, typeList, superList, typeParameterList等字段。甚至为方便起见，我们让这个类也实现NameScope接口，并增加了一个新的作用域类别NameScopeKind.NSK_IMPORTED_TYPE。

表5.19和表5.20给出类ImportedTypeDefinition的方法及说明。简单地说，类ImportedTypeDefinition目前的实现可看做类DetailedTypeDefinition的实现的一个简化版本。只是类ImportedTypeDefinition的实例代表的是没有源代码的类型，很可能也没有额外文件提供它的信息，因此所有字段，包括源代码位置等都可能为null，而且导入类型定义中没有方法体、没有语句，也没有初始化块，从而没有局部作用域等等。除在生成名字定义和名字作用域时会为有额外文件的导入类型生成一些引用（主要是类型引用）之外，不会有其他名字引用出现在导入类型中。

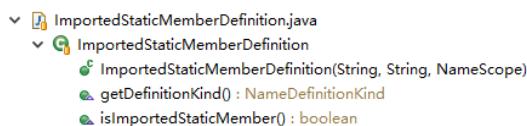


图 5.6 类ImportedStaticMemberDefinition的成员

Java程序还有静态成员导入声明，我们设计类`ImportedStaticMemberDefinition`代表静态成员导入声明所引入的静态导入成员定义，图5.6给出这个类的成员。目前这个类的实现比较简单，只是重定义方法`getDefinitionKind()`和`isImportedStaticMember()`以表明它是静态导入成员定义。实际上，目前这个类的对象实例仅仅是一个标记，我们也没有对静态导入成员的解析做完整的测试，也即对静态导入成员的处理还很不完善。

表 5.19 类ImportedTypeDefinition方法的描述

<code>ImportedTypeDefinition(String, String, SourceCodeLocation, NameScope, SourceCodeLocation) : Constructor</code>	构造方法，以类型简单名、全限定名、起始位置、所在作用域、结束位置为参数，适用于构造有额外信息的导入类型定义实例
<code>ImportedTypeDefinition(String, String, NameScope) : Constructor</code>	构造方法，以类型简单名、全限定名、所在作用域为参数，适用于构造没有额外信息的导入类型定义实例
<code>accept(NameTableVisitor) : void [实现接口NameScope的方法]</code>	接受名字表访问器的访问
<code>addReference(NameReference) : void [实现接口NameScope的方法]</code>	导入类型定义内没有名字引用，因此此方法直接返回
<code>addSuperType(TypeReference) : void</code>	添加超类型引用到超类型引用列表
<code>containsLocation(SourceCodeLocation) : boolean [实现接口NameScope的方法]</code>	判断是否包含指定的源代码位置
<code>define(NameDefinition) : void [实现接口NameScope的方法]</code>	在类型定义中定义名字，可能定义的名字包括字段定义、方法定义、(内部)类型定义
<code>getEnclosingScope() : NameScope [实现接口NameScope的方法]</code>	返回包含当前导入类型定义的作用域
<code>getEndLocation() : SourceCodeLocation</code>	返回声明当前导入类型的源代码结束位置
<code>getFieldList() : List<FieldDefinition></code>	返回在当前导入类型定义中声明的字段列表
<code>getMethodList() : List<MethodDefinition></code>	返回在当前导入类型定义中声明的方法列表
<code>getReferenceList() : List<NameReference> [实现接口NameScope的方法]</code>	返回出现在当前导入类型定义中的名字引用列表，由于导入类型定义没有名字引用，因此直接返回null
<code>getScopeEnd() : SourceCodeLocation [实现接口NameScope的方法]</code>	返回当前导入类型定义作为作用域的结束位置，等于getEndLocation()的返回值
<code>getScopeKind() : NameScopeKind [实现接口NameScope的方法]</code>	返回NameScopeKind.NSK_IMPORTED_TYPE
<code>getScopeName() : String [实现接口NameScope的方法]</code>	返回导入类型定义的简单名作为作用域名字
<code>getScopeStart() : SourceCodeLocation [实现接口NameScope的方法]</code>	返回当前导入类型定义作为作用域的起始位置，等于getLocation()的返回值
<code>getSubScopeList() : List<NameScope> [实现接口NameScope的方法]</code>	返回当前导入类型定义包含的子作用域列表，包括内部类型定义和方法定义
<code>getSuperClassDefinition() : TypeDefinition [重定义类TypeDefinition的方法]</code>	如果当前类型是类类型且有extends声明，则返回extends后声明的超类型引用所绑定的类型定义，否则返回null
<code>getSuperList() : List<TypeReference> [重定义类TypeDefinition的方法]</code>	返回超类型引用列表
<code>getTypeList() : List<ImportedTypeDefinition></code>	返回内部类型列表
<code>getTypeParameterList() : List<TypeParameterDefinition></code>	返回类型参数列表

表 5.20 类 ImportedTypeDefinition 方法的描述 (续)

<code>isEnclosedInScope(NameScope) : boolean</code> [实现接口 NameScope 的方法] 判断当前导入类型定义是否包含在给定作用域内
<code>isImportedType() : boolean</code> [重定义类 NameDefinition 的方法] 返回 true
<code>resolve(NameReference) : boolean</code> [重定义类 TypeDefinition 的方法] 在当前导入类型定义中解析名字引用，主要是与字段、方法和内部类型匹配
继承自类 TypeDefinition 的方法 (不含已被重定义方法): 参见表 5.11 <code>getDefinitionKind():NameDefinitionKind; isInterface():boolean; isPackageMember():boolean</code> <code>getEnclosingPackage():PackageDefinition; isPublic():boolean</code> <code>isSubtypeOf(TypeDefinition):boolean; setInterface(boolean):void; setPackageMember(boolean):void</code>
继承自类 NameDefinition 的方法 (不含已被重定义方法): 参见表 5.9 <code>compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int</code> <code>getFullQualifiedName():String; getSimpleName():String; getUniqueId():String</code> <code>getLocation():SourceCodeLocation; getScope():NameScope; isTypeDefinition():boolean</code> <code>isDetailedType():boolean; isEnumType():boolean; isImportedStaticMember():boolean</code> <code>isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean</code> <code>match(NameReference):boolean; match(String):boolean</code> <code>toFullString():String; toString():String</code>

5.2.6 类 TypeParameterDefinition

类 TypeParameterDefinition 是类 TypeDefinition 的直接子类，它的实例代表一个类型参数定义（即形式类型参数）。在 Java 程序中定义类属类（泛型类，generic classes）或类属方法（泛型方法，generic methods）时使用尖括号给出（形式）类型参数。类属类或类属方法可以有多个类型参数，每个类型参数可使用 extends 给出它的一个或多个界（bound），界用于约束类属类或类属方法的实际类型参数必须是这个界的子类型。如果有多个界，则类类型必须是第一个界，剩下的界必须是接口类型，这意味着实际类型参数必须是第一个界的子类型，而且要实现界列表中的所有接口类型。

不过实际上，我们目前对于类型参数并没有过多处理，对于类属类的使用（即参数化类型引用）和类属方法的调用也没有对实际类型参数与类型参数定义（即形式类型参数）进行匹配和处理。我们目前只是引入了类 TypeParameterDefinition 表示类型参数，对于它的分析（例如类属类和类属方法的实例化）留待以后实现。

表 5.21 给出类 TypeParameterDefinition 的字段，除继承类 TypeDefinition 和类 NameDefinition 的字段外，只有一个界类型列表，其元素是类型引用，因为界的声明实际上是对其他类型（定义）的使用（引用）。

表 5.21 类 TypeParameterDefinition 字段的描述

字段	类型	描述
<code>boundList</code>	<code>List<TypeReference></code>	类型参数的界类型引用列表
继承自类 TypeDefinition 的字段: 参见表 5.10		
<code>isInterface, isPackageMember</code>		
继承自类 NameDefinition 的字段: 参见表 5.8		
<code>fullQualifiedName, SourceCodeLocation, scope, simpleName</code>		

表5.22给出类TypeParameterDefinition的方法，这些方法的实现目前都非常简单。

表 5.22 类TypeParameterDefinition方法的描述

TypeParameterDefinition(String, String, SourceCodeLocation, NameScope) : Constructor 构造方法，以简单名、全限定名、源代码位置、所在作用域为参数
addBoundType(TypeReference) : boolean 添加类型参数的界类型引用
getBoundList() : List<TypeReference> 返回类型参数的界类型引用列表
getDefinitionKind() : NameDefinitionKind [重定义类TypeDefinition的方法] 返回NameDefinitionKind.NDK_Type_PARAMETER
getSuperList() : List<TypeReference> [重定义类TypeDefinition的方法] 返回超类型，这里也返回界类型引用列表
isInterface() : boolean [重定义类TypeDefinition的方法] 返回false
isPackageMember() : boolean [重定义类TypeDefinition的方法] 返回false
继承自类TypeDefinition的方法（不含已被重定义方法）：参见表5.11 getEnclosingPackage():PackageDefinition; getSuperClassDefinition():TypeDefinition; isPublic() : boolean isSubtypeOf(TypeDefinition):boolean; setInterface(boolean):void; setPackageMember(boolean):void
resolve(NameReference):boolean 继承自类NameDefinition的方法（不含已被重定义方法）：参见表5.9 compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int getFullQualifiedName():String; getSimpleName():String; getUniqueId():String getLocation():SourceCodeLocation; getScope():NameScope; isTypeDefinition():boolean isDetailedType():boolean; isEnumType():boolean isImportedType():boolean; isImportedStaticMember():boolean isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean match(NameReference):boolean; match(String):boolean toFullString():String; toString():String

5.2.7 类MethodDefinition和类AutoGeneratedConstructor

类MethodDefinition是类NameDefinition的直接子类，它的实例代表一个方法。表5.23 给出它的字段，主要是给出一个方法的重要组成部分，包括它的方法体（对应一个局部作用域）、方法参数列表（元素是变量定义）、返回类型（是一个类型引用）、抛出异常列表（元素是类型引用）、类型参数列表（元素是类型参数定义）。

类MethodDefinition实现接口NameScope，代表一个方法作用域，因此除了使用继承自类NameDefinition 的字段location保存方法作用域的起始位置外，还使用字段endLocation保存结束位置。方法作用域除包含方法体之外，还包含方法参数和类型参数、抛出异常类型声明等。为支持在作用域中存放名字引用，也设计了字段referenceList。

我们将类的构造方法看做特殊的方法，其简单名与类名相同。目前没有为构造方法单独设计一个类，而是在类MethodDefinition设置一个标志字段constructorFlag标记一个方法定义是否是构造方法。

表5.24和5.25 给出类MethodDefinition的方法及描述。除继承自类NameDefinition的方法

表 5.23 类MethodDefinition字段的描述

字段	类型	描述
bodyScope	LocalScope	方法体对应的局部作用域
constructorFlag	boolean	是否是构造方法的标记
endLocation	SourceCodeLocation	方法声明的源代码结束位置
modifier	int	声明方法时的修饰符
parameterList	List<VariableDefinition>	方法的参数定义列表，每个参数是一个变量定义
referenceList	List<NameReference>	直接出现在方法声明中的名字引用列表
returnType	TypeReference	方法的返回类型引用
throwTypeList	List<TypeReference>	方法声明抛出的异常类型引用
typeParameterList	List<TypeParameterDefinition>	方法的类型参数列表，每个参数是一个类型参数定义
继承自类NameDefinition的字段：参见表5.8		
fullQualifiedName, SourceCodeLocation, scope, simpleName		

外，类MethodDefinition 的方法也主要是维护它的字段的getter和setter方法，以及实现接口NameScope规定的方法。

作为作用域，直接定义在方法作用域的名字是它的方法参数和类型参数，因此方法参数列表(parameterList)和类型参数列表(typeParameterList)的元素由方法define()添加。相应地，在方法作用域解析名字引用 (resolve() 方法的实现) 也是在这两个列表中进行匹配。方法作用域的子作用域只有它的方法体 (对应的局部作用域)，因为只有方法体被直接包含在方法作用域中。

类MethodDefinition的方法getReturnType()只是给出代表当前方法返回类型的一个类型引用，而getReturnTypeDefinition()才给出当前方法返回类型 (所绑定的类型定义)。当返回类型是一个参数化类型引用时，例如List<String>，我们有时既要知道方法返回的主类型 (即List)，也要知道它的参数类型 (即String，是列表的元素类型，理解和分析方法的返回值可能更关注这个元素类型)，因此带参数的getReturnTypeDefinition()方法在参数为true时既可返回主类型也可返回参数类型 (它们一起构成的列表)，而不带参数的同名方法则只是返回主类型。

方法matchMethod()将当前方法定义与一个方法引用进行匹配，这时方法引用中应已经存有方法的实际参数列表 (每个实际参数是一个名字引用)，匹配除了匹配方法定义的简单名与方法引用中保存的名字外，还匹配参数个数与对应参数的类型。实际参数 (是名字引用，代表一个表达式) 的结果类型必须是形式参数 (变量定义) 的声明类型的子类型才会匹配成功。类NameReference的方法getResultTypeDefinition()可给出一个名字引用 (代表一个表达式) 的结果类型。

方法isOverloadMethod()和isOverrideMethod()用于判定当前方法与给定方法是否是互为重载和互为重定义方法。两个具有相同简单名的方法如果参数列表 (参数个数或对应参数类型) 不同则是互为重载方法，如果参数列表 (参数个数和对应参数类型) 完全相同则是互为重定义方法。当然严格意义上的互为重载只有这两个方法是同一个类的成员 (可能是继承成员) 时才成立，而重定义则只有后代类的方法重定义超类的方法。因此需要结合子类型关系才能真正判定两个方法是否是重载或重定义。

类MethodDefinition也可能代表一个构造方法。当一个Java类没有显式定义的构造方法时，Java编译器会假定有一个没有任何参数的缺省构造方法。为处理这种情况 (从而可解析对缺省构造方法的调用)，我们设计类AutoGeneratedConstructor 代表自动生成的缺省构造方法，它是类MethodDefinition的直接子类。图5.7 给出这个类的成员。

表 5.24 类MethodDefinition方法的描述

<code>MethodDefinition(String, String, SourceCodeLocation, NameScope, SourceCodeLocation) : Constructor</code>	构造方法，以简单名、全限定名、起始位置、所在作用域、结束位置为参数
<code>accept(NameTableVisitor) : void</code> [实现接口NameScope的方法]	接受名字访问器的访问
<code>addReference(NameReference) : void</code> [实现接口NameScope的方法]	添加名字引用到名字引用列表
<code>addThrowType(TypeReference) : void</code>	添加一个抛出异常类型引用
<code>containsLocation(SourceCodeLocation) : boolean</code> [实现接口NameScope的方法]	判断是否包含指定的源代码位置
<code>define(NameDefinition) : void</code> [实现接口NameScope的方法]	在方法作用域中定义名字，包括定义参数和类型参数
<code>getBodyScope() : LocalScope</code>	返回方法体对应的局部作用域
<code>getDefinitionKind() : NameDefinitionKind</code> [重定义类NameDefinition的方法]	返回NameDefinitionKind.NDK_METHOD
<code>getEnclosingScope() : NameScope</code> [实现接口NameScope的方法]	返回当前方法所在的作用域
<code>getEnclosingType() : DetailedTypeDefinition</code>	返回声明当前方法的详细类型定义
<code>getEndLocation() : SourceCodeLocation</code>	返回方法声明的源代码结束位置
<code>getParameterList() : List<VariableDefinition></code>	返回方法的参数列表
<code>getReferenceList() : List<NameReference></code> [实现接口NameScope的方法]	返回方法的名字引用列表
<code>getReturnType() : TypeReference</code>	返回方法的返回类型引用
<code>getReturnTypeDefinition() : TypeDefinition</code>	返回方法的返回类型引用所绑定的类型定义，当绑定的类型定义是参数化类型定义时，只返回主类型定义
<code>getReturnTypeDefinition(boolean) : List<TypeDefinition></code>	返回方法的返回类型引用所绑定的类型定义，当绑定的类型定义是参数化类型定义时，将根据指定的参数是只返回主类型（指定参数为false）还是将所有的参数类型也一起返回（指定参数为true）。
<code>getScopeEnd() : SourceCodeLocation</code> [实现接口NameScope的方法]	返回方法声明的源代码结束位置，返回值域getEndLocation()相同
<code>getScopeKind() : NameScopeKind</code> [实现接口NameScope的方法]	返回NameScopeKind.NSK_METHOD
<code>getScopeName() : String</code> [实现接口NameScope的方法]	返回方法简单名作为作用域名
<code>getScopeStart() : SourceCodeLocation</code> [实现接口NameScope的方法]	返回方法声明的源代码起始位置，返回值域getLocation()相同
<code>getSubScopeList() : List<NameScope></code> [实现接口NameScope的方法]	返回子作用域列表，方法声明只有一个子作用域就是它的方法体对应的局部作用域
<code>getThrowTypeList() : List<TypeReference></code>	返回方法声明的抛出异常类型引用列表

表 5.25 类MethodDefinition方法的描述（续）

<code>isAbstract() : boolean</code>	返回方法是否是抽象方法
<code>isAutoGenerated() : boolean</code>	返回方法是否是自动生成的构造方法
<code>isConstructor() : boolean</code>	返回方法是否是构造方法
<code>isEnclosedInScope(NameScope) : boolean</code> [实现接口NameScope的方法]	判断方法作用域是否真包含在指定作用域中
<code>isFinal() : boolean</code>	返回方法是否是终态方法
<code>isOverloadMethod(MethodDefinition) : boolean</code>	判断当前方法与参数指定的方法是否是互为重载的方法，即有相同的简单名，但方法参数列表不同
<code>isOverrideMethod(MethodDefinition) : boolean</code>	判断当前方法与参数指定的方法是否是互为重定义的方法，即有相同的简单名，且有相同的方法参数（即个数和类型相同）列表
<code>isPrivate() : boolean</code>	判断方法是否是私有方法
<code>isProtected() : boolean</code>	判断方法是否是受保护方法
<code>isPublic() : boolean</code>	判断方法是否是公有方法
<code>isStatic() : boolean</code>	判断方法是否是静态方法
<code>matchMethod(MethodReference) : boolean</code>	与一个方法引用进行匹配，除匹配方法定义的简单名与方法引用中保存的名字是否相同之外，还要匹配方法引用的实际参数与方法定义的形式参数：方法引用的实际参数的类型定义要是形式参数的类型定义的子类型才能匹配成功（即返回true）
<code>resolve(NameReference) : boolean</code> [实现接口NameScope的方法]	在方法作用域中解析名字，主要是匹配方法参数（变量定义）和方法类型参数
<code>setBodyScope(LocalScope) : void</code>	设置方法的方法体对应的局部作用域
<code>setConstructor(boolean) : void</code>	设置方法是否是构造方法
<code>setConstructor() : void</code>	设置方法为构造方法
<code>setModifierFlag(int) : void</code>	设置代表方法各种修饰符的整数
<code>setReturnType(TypeReference) : void</code>	设置方法的返回类型引用
继承自类NameDefinition的方法（不含已被重定义方法）：参见表5.9	
<code>compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int</code>	
<code>getFullQualifiedName():String; getSimpleName():String; getUniqueId():String</code>	
<code>getLocation():SourceCodeLocation; getScope():NameScope;</code>	
<code>isDetailedType():boolean; isEnumType():boolean; isTypeDefinition():boolean</code>	
<code>isImportedType():boolean; isImportedStaticMember():boolean</code>	
<code>isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean</code>	
<code>match(NameReference):boolean; match(String):boolean</code>	
<code>toFullString():String; toString():String</code>	



图 5.7 类AutoGeneratedConstructor的成员

类AutoGeneratedConstructor的实现很简单，只是重定义方法isAutoGenerated()返回true表示它是自动生成的，且重定义方法isAbstract()返回true表示它没有方法体。

5.2.8 类FieldDefinition

类FieldDefinition是类NameDefinition的直接子类，它的实例表示一个字段，即类的数据成员，包括静态或非静态数据成员。字段定义本身的组成很简单，主要是需要关注它的声明类型。表5.26给出类FieldDefinition的字段说明，其中type表示字段定义的声明类型，是一个类型引用。

表 5.26 类FieldDefinition字段的描述

字段	类型	描述
<code>modifier</code>	<code>int</code>	声明字段时的修饰符
<code>type</code>	<code>TypeReference</code>	字段的类型，这是一个对类型定义的引用
继承自类 <code>NameDefinition</code> 的字段：参见表5.8		
<code>fullQualifiedName</code> , <code>SourceCodeLocation</code> , <code>scope</code> , <code>simpleName</code>		

表5.27给出类FieldDefinition的方法，这些方法的实现都比较简单。对于字段定义的类型声明，方法getType()返回声明时的类型引用，而getTypeDefinition()方法返回该类型引用所能绑定到的类型定义。与类MethodDefinition的方法getReturnTypeDefinition()类似，由于字段声明时的类型可能是参数化类型，因此带参数的getTypeDefinition()方法在参数为true可返回主类型和类型参数所能绑定到的类型定义构成的列表，而在参数为false时只返回主类型（引用）绑定到的类型定义构成的列表。没有参数的getTypeDefinition()返回也只返回主类型（引用）绑定到的类型定义。

5.2.9 类VariableDefinition

类VariableDefinition是类NameDefinition的直接子类，它的实例表示一个变量定义，包括局部变量定义和方法参数定义。我们将方法形式参数的声明也看做变量定义，但使用枚举类型NameDefinitionKind的不同枚举常量加以区别，因此在类VariableDefinition设计字段kind存放不同的变量定义列表。表5.28给出类VariableDefinition的字段。

与类FieldDefinition类似，类VariableDefinition最重要的属性是变量声明时的类型，它存放在字段type是一个类型引用。在Java程序中，可使用一个类型同时声明多个变量，例如int i, j;，我们将为其中的每个变量（即i和j）都创建一个类VariableDefinition的实例。

表5.29给出类VariableDefinition的方法。这些方法的实现也都比较简单。同样，对于变量定义的类型声明，方法getType()返回声明时的类型引用，而getTypeDefinition()方法返回该类型

表 5.27 类FieldDefinition方法的描述

<code>FieldDefinition(String, String, SourceCodeLocation, NameScope) : Constructor</code>	构造方法，以简单名、全限定名、源代码位置和所在作用域为参数
<code>getDefinitionKind() : NameDefinitionKind</code> [重定义类NameDefinition的方法]	返回NameDefinitionKind.NDK_FIELD
<code>getEnclosingType() : DetailedTypeDefinition</code>	返回该字段所属的详细类型定义
<code>getType() : TypeReference</code>	返回该字段的类型，这个函数返回的是一个类型引用
<code>getTypeDefinition() : TypeDefinition</code>	返回该字段的类型，这个函数返回的字段类型引用所绑定的类型定义，若声明字段的类型为参数化类型，则返回主类型对应的类型定义。当该字段类型引用绑定不成功时返回null
<code>getTypeDefinition(boolean) : List<TypeDefinition></code>	返回该字段的类型，当给定参数为true时且声明字段的类型为参数化类型时将返回主类型与类型参数对应的类型定义构成的列表，如果给定参数为false则返回只含主类型对应类型定义的列表。当绑定不成功时返回空表
<code>isFinal() : boolean</code>	判断字段是否是终态字段
<code>isPrivate() : boolean</code>	判断字段是否是私有字段
<code>isProtected() : boolean</code>	判断字段是否是受保护字段
<code>isPublic() : boolean</code>	判断字段是否是公有字段
<code>isStatic() : boolean</code>	判断字段是否是静态字段
<code>setModifierFlag(int) : void</code>	设置字段的修饰符，用一个整数表示
<code>setType(TypeReference) : void</code>	设置字段的类型(引用)
<code>toDeclarationString() : String</code>	将字段的信息转换为在程序源代码中声明的样式，主要是当字段类型是数组类型时要添加合适的中括号
继承自类NameDefinition的方法(不含已被重定义方法): 参见表5.9	
<code>compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int</code>	
<code>getFullQualifiedName():String; getSimpleName():String; getUniqueId():String</code>	
<code>getLocation():SourceCodeLocation; getScope():NameScope;</code>	
<code>isDetailedType():boolean; isEnumType():boolean; isTypeDefinition():boolean</code>	
<code>isImportedType():boolean; isImportedStaticMember():boolean</code>	
<code>isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean</code>	
<code>match(NameReference):boolean; match(String):boolean</code>	
<code>toFullString():String; toString():String</code>	

表 5.28 类VariableDefinition字段的描述

字段	类型	描述
<code>kind</code>	<code>NameDefinitionKind</code>	变量定义可能对应局部变量NDK_VARIABLE也可能存放方法参数NDK_PARAMETER
<code>type</code>	<code>TypeReference</code>	字段的类型，这是一个对类型定义的引用
继承自类NameDefinition的字段: 参见表5.8		
<code>fullQualifiedName, SourceCodeLocation, scope, simpleName</code>		

引用所能绑定到的类型定义。特别地，带参数的`getTypeDefinition()`方法在参数为`true`可返回主类型和类型参数所能绑定到的类型定义构成的列表，而在参数为`false`时只返回主类型（引用）绑定到的类型定义构成的列表。没有参数的`getTypeDefinition()`返回也只返回主类型（引用）绑定到的类型定义。

表 5.29 类VariableDefinition方法的描述

<code>VariableDefinition(String, String, SourceCodeLocation, NameScope) : Constructor</code>
构造方法，以简单名、全限定名、源代码位置和所在作用域为参数
<code>getDefinitionKind() : NameDefinitionKind [重定义类NameDefinition的方法]</code>
返回变量定义的名字定义列表，可能是 <code>NameDefinitionKind.NDK_VARIABLE</code> 代表局部变量声明，也可能是 <code>NameDefinitionKind.NDK_PARAMETER</code> 代表方法的形式参数声明
<code>getType() : TypeReference</code>
返回该变量（或方法参数）声明时的类型，这个函数返回的是一个类型引用
<code>getTypeDefinition() : TypeDefinition</code>
返回该变量（或方法参数）声明时的类型，这个函数返回的变量类型引用所绑定的类型定义，若声明变量的类型为参数化类型，则返回主类型对应的类型定义。当该变量类型引用绑定不成功时返回 <code>null</code>
<code>getTypeDefinition(boolean) : List<TypeDefinition></code>
返回该变量（或方法参数）的类型，当给定参数为 <code>true</code> 时且声明变量的类型为参数化类型时将返回主类型与类型参数对应的类型定义构成的列表，如果给定参数为 <code>false</code> 则返回只含主类型对应类型定义的列表。当绑定不成功时返回空表
<code>setDefinitionKind(NameDefinitionKind) : void</code>
设置变量定义的名字定义类别
<code>setType(TypeReference) : void</code>
设置该变量（或方法参数）声明时的类型（引用）
<code>toDeclarationString() : String</code>
将变量定义的信息转换为在程序源代码中声明的样式，主要是当变量类型是数组类型时要添加合适的中括号
继承自类NameDefinition的方法（不含已被重定义方法）：参见表5.9
<code>compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int</code> <code>getFullQualifiedName():String; getSimpleName():String; getUniqueId():String</code> <code>getLocation():SourceCodeLocation; getScope():NameScope;</code> <code>isDetailedType():boolean; isEnumType():boolean; isTypeDefinition():boolean</code> <code>isImportedType():boolean; isImportedStaticMember():boolean</code> <code>isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean</code> <code>match(NameReference):boolean; match(String):boolean</code> <code>toFullString():String; toString():String</code>

5.2.10 类PackageDefinition

类`PackageDefinition`是类`NameDefinition`的直接子类，它的实例代表一个程序包定义。表5.30给出类`PackageDefinition`的字段。程序包定义由Java程序的编译单元中使用`package`语句而得到，多个编译单元可属于同一个程序包，因此程序包定义的最重要属性是属于该程序包定义的编译单元列表，即表5.30中的字段`unitList`，其元素是类`CompilationUnitScope`的实例。

类`PackageDefinition`实现接口`NameScope`，作为作用域，可理解程序包定义由属于它的所有编译单元作用域构成。为存放可能出现在程序包作用域的名字引用，设计了字段`referenceList`。不过正常情况下，应该没有名字引用直接出现在程序包定义中。

每个Java程序允许出现一个不具名程序包(unnamed package)，它包含那些没有`package`语句的编译单元。为处理不具名程序包，我们定义一个特定的字符串常量`UNNAMED_PACKAGE_NAME`作为

不具名程序包的简单名（和全限定名），从而可以像普通程序包一样处理不具名程序包。

在概念上，Java语言的程序包还可包括子程序包，但程序包和子程序包只是当编译单元在操作系统文件组织管理中起作用，子程序包对应的目录是程序包对应目录的子目录，从作用域的角度看，程序包与子程序包并没有包含关系，因此在JAnalyzer 平台中，**所有程序包都处于同样级别，全部直接包含在系统作用域中**，每个程序包都使用全名作为它的简单名和全限定名。

表 5.30 类PackageDefinition字段的描述

字段	类型	描述
UNNAMED_PACKAGE_NAME	String	一个字符串常量，作为不具名程序包的程序包名字
referenceList	List<NameReference>	出现在程序包中的名字引用列表，正常情况下应为null
unitList	List<CompilationUnitScope>	属于该程序包的编译单元对应的编译单元作用域列表
继承自类NameDefinition的字段：参见表5.8		
fullQualifiedName, SourceCodeLocation, scope, simpleName		

表5.31给出类PackageDefinition的方法。每个程序包都使用（可能带圆点的）全名作为它的简单名和全限定名，因此类PackageDefinition的构造方法最多有一个字符串类型的参数指定程序包的名字，如果没有给出字符串类型的参数，则创建不具名程序包对应的程序包定义（对象实例），方法isUnnamedPackage()用于判定当前程序包定义是否是不具名程序包。

除继承自类NameDefinition的方法外，类PackageDefinition本身声明的方法主要是实现接口NameScope 规定的方法。作为作用域，没有名字直接定义在程序包，因此方法define()的实现抛出异常。而在程序包定义中解析名字引用，主要要实现跨编译单元匹配该程序包作用域中的顶层类型，因此方法resolve()的实现实际上是在当前程序包所属的所有编译单元中匹配顶层类型，当匹配不成功则在所属作用域（也即系统作用域）中继续匹配（即继续在导入类型定义和导入静态成员中进行匹配）。

但有时我们需要只在程序包所属编译单元中匹配顶层类型，不能继续在系统作用域中匹配（例如在绑定导入声明语句所创建的类型引用时），因此设计了方法matchTypeReference()，这个方法完成与resolve()类似的功能，只是当匹配不成功时不再激发在系统作用域中的匹配。

5.3 名字引用

程序包nameTable.nameReference和nameTable.nameReference.referenceGroup包括所有代表名字引用的类，后者包括类NameReferenceGroup及具体代表名字引用组的类，前者包括剩余的代表名字引用的类。类NameReference是所有代表名字引用的类的基类，图5.8给出这些类的继承层次结构。

5.3.1 类NameReference

类NameReference是所有代表名字引用的类的基类，给出与名字引用相关的共同属性和方法。类NameReference 不是抽象类，它的实例可代表最简单的对一个名字的引用。表5.32给出这个类的字段，包括名字引用类别(kind)（枚举类型NameReferenceKind定义了所有可能的类别）、源代码位置(location)、名字(name)和所在作用域(scope）。名字引用的名字是Java程序源代码中一段

表 5.31 类PackageDefinition方法的描述

<code>PackageDefinition(NameScope) : Constructor</code>	构造方法，以所属作用域（即系统作用域）为参数，用于构造不具名程序包对应的程序包定义
<code>PackageDefinition(String, NameScope) : Constructor</code>	构造方法，以程序包名字、所属作用域（即系统作用域）为参数。程序包的简单名与全限定名相同（都是带圆点的全限定名）
<code>accept(NameTableVisitor) : void [实现接口NameScope的方法]</code>	接受名字表访问器的访问
<code>addCompilationUnitScope(CompilationUnitScope) : void</code>	添加编译单元作用域到程序包定义
<code>addReference(NameReference) : void [实现接口NameScope的方法]</code>	添加名字引用
<code>containsLocation(SourceCodeLocation) : boolean [实现接口NameScope的方法]</code>	判断是否包含给定源代码位置，当参数给定的源代码位置的编译单元属于当前程序包时返回true，否则返回false
<code>define(NameDefinition) : void [实现接口NameScope的方法]</code>	正常情况下程序包中不能定义名字，因此调用这个方法会抛出IllegalNameDefinition异常
<code>getAllDetailedTypeDefinitions() : List<DetailedTypeDefinition></code>	返回该程序包所有编译单元中顶层详细类型定义构成的列表
<code>getCompilationUnitScopeList() : List<CompilationUnitScope></code>	返回属于该程序包的所有编译单元作用域构成的列表
<code>getDefinitionKind() : NameDefinitionKind [重定义类NameDefinition的方法]</code>	返回NameDefinitionKind.NDK_PACKAGE
<code>getEnclosingScope() : NameScope [实现接口NameScope的方法]</code>	返回所属作用域
<code>getReferenceList() : List<NameReference> [实现接口NameScope的方法]</code>	返回名字引用列表
<code>getScopeEnd() : SourceCodeLocation [实现接口NameScope的方法]</code>	返回null，对于程序包定义而言调用这个方法没有意义
<code>getScopeKind() : NameScopeKind [实现接口NameScope的方法]</code>	返回NameScopeKind.NSK_PACKAGE
<code>getScopeName() : String [实现接口NameScope的方法]</code>	返回程序包的名字作为作用域名
<code>getScopeStart() : SourceCodeLocation [实现接口NameScope的方法]</code>	返回null，对于程序包定义而言调用这个方法没有意义
<code>getSubScopeList() : List<NameScope> [实现接口NameScope的方法]</code>	返回子作用域列表，对于程序包定义，就是返回所有编译单元作用域构成的列表
<code>isEnclosedInScope(NameScope) : boolean [实现接口NameScope的方法]</code>	判断是否真包含在给定作用域
<code>isUnnamedPackage() : boolean</code>	判断当前程序包定义是否代表不具名程序包。每个系统作用域中只有一个不具名程序包定义
<code>matchTypeWithReference(NameReference) : boolean</code>	将名字引用中保存的名字看做是类型名，在当前程序包所包含的编译单元匹配顶层类型名字。
<code>resolve(NameReference) : boolean [实现接口NameScope的方法]</code>	在当前程序包中解析名字引用。完成与函数matchTypeWithReference()类似的功能，但在匹配不成功时会激发继续在系统作用域中匹配
继承自类NameDefinition的方法（不含已被重定义方法）：参见表5.9	
<code>compareTo(NameDefinition):int; equals(Object):boolean; hashCode():int</code>	
<code>getFullQualifiedName():String; getSimpleName():String; getUniqueId():String</code>	
<code>getLocation():SourceCodeLocation; getScope():NameScope;</code>	
<code>isDetailedType():boolean; isEnumType():boolean; isTypeDefinition():boolean</code>	
<code>isImportedType():boolean; isImportedStaticMember():boolean</code>	
<code>isFieldDefinition():boolean; isMethodDefinition():boolean; isVariableDefinition():boolean</code>	
<code>match(NameReference):boolean; match(String):boolean</code>	
<code>toFullString():String; toString():String</code>	

```

- nameTable.nameReference NameReference (implements java.lang.Comparable<T>)
  - nameTable.nameReference.LiteralReference
  - nameTable.nameReference.MethodReference
  - nameTable.nameReference.referenceGroup.NameReferenceGroup
    - nameTable.nameReference.referenceGroup.NRGArrayAccess
    - nameTable.nameReference.referenceGroup.NRGArrayCreation
    - nameTable.nameReference.referenceGroup.NRGArrayInitializer
    - nameTable.nameReference.referenceGroup.NRGAssignment
    - nameTable.nameReference.referenceGroup.NRGCast
    - nameTable.nameReference.referenceGroup.NRGClassInstanceCreation
    - nameTable.nameReference.referenceGroup.NRGConditional
    - nameTable.nameReference.referenceGroup.NRGFieldAccess
    - nameTable.nameReference.referenceGroup.NRGInfixExpression
    - nameTable.nameReference.referenceGroup.NRGInstanceof
    - nameTable.nameReference.referenceGroup.NRGMethodInvocation
    - nameTable.nameReference.referenceGroup.NRGPostfixExpression
    - nameTable.nameReference.referenceGroup.NRGPrefixExpression
    - nameTable.nameReference.referenceGroup.NRGQualifiedName
    - nameTable.nameReference.referenceGroup.NRGSuperFieldAccess
    - nameTable.nameReference.referenceGroup.NRGSuperMethodInvocation
    - nameTable.nameReference.referenceGroup.NRGThisExpression
    - nameTable.nameReference.referenceGroup.NRGTypeLiteral
    - nameTable.nameReference.referenceGroup.NRGVariableDeclaration
  - nameTable.nameReference.PackageReference
  - nameTable.nameReference.TypeReference
    - nameTable.nameReference.IntersectionTypeReference
    - nameTable.nameReference.NamedTypeReference
    - nameTable.nameReference.ParameterizedTypeReference
    - nameTable.nameReference.QualifiedTypeReference
    - nameTable.nameReference.UnionTypeReference
    - nameTable.nameReference.WildcardTypeReference
  - nameTable.nameReference.ValueReference

```

图 5.8 代表名字引用的类的继承层次结构

代码片段（字符串），即在语法上用于引用某个名字的一段代码。对于最简单的名字引用，其中名字字段name保存的通常是所要访问的名字定义的简单名，而对于对应表达式的名字引用（组），字段name会存储整个表达式。

类NameReference的字段definition记录该名字引用所能绑定到的名字定义，在名字解析之前的缺省值是null，在调用类NameReference的解析方法resolveBinding()之后存放能绑定到的名字定义。

表 5.32 类NameReference字段的描述

字段	类型	描述
definition	NameDefinition	名字引用所绑定到的名字定义
kind	NameReferenceKind	名字引用类别
location	SourceCodeLocation	名字引用所在源代码位置
name	String	所引用的名字
scope	NameScope	名字引用所在的作用域

表5.33给出类NameReference的方法及说明。类NameReference实现接口Comparable<NameReference>以便在Java容器类实例中有序存放，因此我们基于名字引用所保存的名字以及源代码位置自动生成名字引用的唯一标识，方法getUniqueId()可确定名字引用的唯一标识，而方法compareTo()、equals()以及hashCode()也基于名字引用所保存的名字以及源代码位置进行有序比较、相等比较和计算散列值，使得当名字引用有序存放在Java容器类时能按名字引用的唯一标识的字典顺序存放。

类NameReference的方法getReferencesAtLeaf()用于返回一个名字引用内可能包含的多个最基本的名字引用构成的列表。最基本的名字引用是指那些不再包含其他名字引用的名字引用。目前主要是有三种情形：

(1) 名字引用组代表一个表达式，对应表达式的语法树结构包含多个名字引用。类NameReferenceGroup的

方法`getReferencesAtLeaf()`将返回在表达式语法结构的叶子节点上的名字引用；

(2) 一个参数化类型引用实例不仅包括主类型引用而且包括类型参数引用。类`ParameterizedTypeReference`的方法`getReferencesAtLeaf()`将返回主类型引用和类型参数引用构成的列表；

(3) 一个方法引用实例不仅包括对方法本身的引用，而且为了方法匹配还包括方法参数、方法类型参数引用。类`MethodReference`的`getReferencesAtLeaf()`将返回方法引用本身、方法参数、方法类型参数构成的名字引用列表。

其他名字引用类使用类`NameReference`对方法`getReferencesAtLeaf()`的缺省实现，即返回以自己为唯一元素的名字引用列表（即不再将当前名字引用实例拆分成多个名字引用构成的列表）。

类`NameReference`的方法`bindTo()`将当前名字引用绑定到给定名字定义，方法`resolveBinding()`则调用所在作用域的`resolve()`解析当前名字引用。类`NameReference`的后代类重定义方法`resolveBinding()`以完成更复杂的名字引用解析工作。如果解析成功，则方法`resolveBinding()`返回`true`。所谓解析成功是指字段`definition`被赋予不等于`null`的值。为提高效率，一个名字引用实例的名字解析只做一次，也即当字段`definition`的值不是`null`时，方法`resolveBinding()`将直接返回（不做真正的解析工作）。方法`isResolved()`判断当前名字引用是否已经解析，当字段`definition`不为`null`时返回`true`，否则返回`false`。

方法`getResultTypeDefinition()`返回名字引用的结果类型。名字引用的结果类型定义为：

(1) 若名字引用是类型引用，可能绑定的名字定义就是类型定义，因此它的结果类型就是这个类型定义；

(2) 若名字引用是方法引用，可能绑定的名字定义是方法定义，它的结果类型是该方法定义的返回类型（所绑定的类型定义）；

(3) 若名字引用是值引用，可能绑定的名字定义是字段定义、变量定义（含方法参数定义），它的结果类型是该字段或变量定义的声明类型（所绑定的类型定义）；

(4) 其他情况（指文字引用、程序包引用），结果类型没有定义，方法`getResultTypeDefinition()`将返回`null`。

类`NameReference`的方法`setLeftValueReference()`用于将当前名字引用设为左值引用。所谓左值引用是指能放在赋值运算符左边的表达式，也即实际上是用这个表达式的结果所在的内存区域而不是结果本身。本质上只有值引用（类`ValueReference`的实例）才能作为左值引用，但是值引用可能处于名字引用组的某个叶子节点，为避免在设置左值引用时还要进行类型转换及遍历名字引用组，我们在类`NameReference`实现方法`setLeftValueReference()`，从而可更方便地设置一个名字引用为左值引用（但最终只可能将值引用设为左值引用）。相应地，方法`isLeftValue()`用于判断一个名字引用是否是左值引用。

类`NameReference`实现了几个方法来将一个名字引用转换为可用于输出的信息串。方法`toString()`给出类别和唯一标识构成的字符串；方法`toFullString()`给出类别、名字、源代码位置、所在作用域等构成的字符串；方法`toMultilineString()`则给出更详细的信息，包括当名字引用（组）含有子名字引用时同时给出子名字引用的信息，从而成为一个可能含有多行信息的字符串。

5.3.2 类`TypeReference`

类`TypeReference`是类`NameReference`的直接子类，它的实例代表一个类型引用，即使用类型

表 5.33 类NameReference方法的描述

<code>NameReference(String, SourceCodeLocation, NameScope) : Constructor</code>	构造方法，以名字、源代码位置、所在作用域为参数
<code>NameReference(String, SourceCodeLocation, NameScope, NameReferenceKind) : Constructor</code>	构造方法，以名字、源代码位置、所在作用域和名字引用列表为参数
<code>bindTo(NameDefinition) : void</code>	将当前名字引用绑定到给定名字定义
<code>bindedDefinitionToString() : String</code>	将名字引用所绑定的名字定义转换为合适字符串信息
<code>compareTo(NameReference) : int</code>	与另一个名字引用比较以便在Java容器类实例中有序存放名字引用
<code>equals(Object) : boolean</code>	基于名字引用的唯一标识比较两个名字引用是否相同
<code>getDefinition() : NameDefinition</code>	返回名字引用所绑定到的名字定义
<code>getEnclosingTypeDefinition() : TypeDefinition</code>	给出名字引用出现在哪个类型定义
<code>getLocation() : SourceCodeLocation</code>	返回名字引用的源代码位置
<code>getName() : String</code>	返回名字引用中所保存的名字
<code>getReferenceKind() : NameReferenceKind</code>	返回名字引用类别
<code>getReferencesAtLeaf() : List<NameReference></code>	返回当前名字引用所包含的最基本的名字引用
<code>getReturnTypeDefinition() : TypeDefinition</code>	返回名字引用的结果类型(所绑定到的类型定义)
<code>getScope() : NameScope</code>	返回名字引用所在的作用域
<code>getUniqueId() : String</code>	返回名字引用的唯一标识
<code>hashCode() : int</code>	基于名字引用的唯一标识计算散列码
<code>isGroupReference() : boolean</code>	
<code>isLiteralReference() : boolean</code>	
<code>isMethodReference() : boolean</code>	
<code>isTypeReference() : boolean</code>	判断名字引用是否是特定类别的名字引用
<code>isLeftValue() : boolean</code>	判断名字引用是否代表左值引用
<code>isResolved() : boolean</code>	判断名字引用是否已经被解析，实际上是返回 <code>definition != null</code> 的值
<code>resolveBinding() : boolean</code>	解析名字引用，缺省的行为是从名字引用所在的作用域开始解析名字引用，但对名字引用组而言需要根据表达式的语法对其子名字引用逐个解析
<code>setLeftValueReference() : void</code>	设置名字引用为左值引用
<code>setReferenceKind(NameReferenceKind) : void</code>	设置名字引用的类别
<code>toFullString() : String</code>	将名字引用转换为用于输出的信息字符串，包含类别、名字、源代码位置、所在作用域等比较详细内容
<code>toMultilineString(int, boolean) : String</code>	将名字引用转换为用于输出的信息字符串，除类别、名字、源代码位置、所在作用域等信息外，还可能将包含的子名字引用的信息也都输出，因此可能有多行信息
<code>toString() : String</code>	将名字引用转换为类别和唯一标识的比较简单的用于输出的信息串

声明变量、参数、字段等等。类型引用是程序中最常见的引用之一。Java程序有丰富的类型引用方式，因此这个类有很多子类。我们使用枚举类型TypeReferenceKind中的枚举常量区分它的不同子类。类TypeReference是所有代表类型引用的类的基类，它不是抽象类，它的一个实例可代表最简单的类型引用（即只有一个简单名字的类型引用）。

表5.34给出类TypeReference的字段。除继承类NameReference的字段外，类TypeReference有字段typeKind记录类型引用类别，字段dimension记录引用类型是的数组维数。由于任何类型可声明相应的数组类型，因此数组类型本身不能作为类型定义，而只能是作为类型引用时的属性（即使用类型定义来声明变量、字段时的属性），当数组维数为0时表明该类型引用是普通类型引用（非数组类型引用）。

表 5.34 类TypeReference字段的描述

字段	类型	描述
dimension	int	引用类型时的数组维数
typeKind	TypeReferenceKind	类型引用的类别
继承自类NameReference的字段：参见表5.32		
definition, kind, location, name, scope		

表5.35给出类TypeReference的方法。除使用方法getTypeKind()可确定类型引用的类别外，也有一系列is...()方法判定类型引用的类别，包括方法isArrayType()可判断当前的类型引用是否是数组类型引用。

方法resolveBinding()提供类型引用（包括最简单的类型引用）在解析时的缺省工作，即当类型名是原子类型或自动导入的类型（即程序包java.lang中的类型）时直接在系统作用域解析当前类型引用，否则在所属作用域内解析。

类TypeReference重定义了类NameReference的几个to...String()方法，主要是当类型引用是数组类型时，在名字后面加上适当数目（基于字段dimension的值）中括号以表明当前类型引用是数组类型。进一步，也提供了方法toDeclarationString()这个方法将类型因为转换为声明时字符串，即类型名字加上适当数目中括号构成的字符串。

5.3.3 类ParameterizedTypeReference

类ParameterizedTypeReference是类TypeReference的直接子类，它的实例代表一个参数化类型引用，即在一个类型名后使用尖括号给出一个或多个类型参数形式的类型引用，例如List<String>，其中尖括号前的类型名是主类型引用，它所绑定（引用）的类型应该是类属类型（泛型类型），尖括号给出的类型引用是实际类型参数。

表5.36给出类ParameterizedTypeReference的字段，除继承自类TypeReference和NameReference的字段外，它的字段主要是记录参数化类型引用的主类型和类型参数列表。

表5.37给出类ParameterizedTypeReference的方法，其中resolveBinding()会调用主类型和类型参数的resolveBinding()方法进行解析，然后将整个参数化类型引用绑定到主类型所绑定的类型定义。方法getDefinition(boolean)在给定参数为true会返回包括主类型和所有类型参数所绑定到的类型定义构成的列表，而将名字引用转换为用于输出的信息字符串的几

表 5.35 类TypeReference方法的描述

TypeReference(String, SourceCodeLocation, NameScope) : Constructor	构造方法, 以名字、源代码位置、所在作用域为参数
TypeReference(TypeReference) : Constructor	拷贝构造方法, 拷贝另外一个类型引用的信息而构造新的示例
getDimension() : int	返回引用类型时的维数
getTypeKind() : TypeReferenceKind	返回类型引用类别, 缺省值为TRK_SIMPLE
isArrayType() : boolean	判断是否是数组类型, 当字段dimension大于0时返回true
isNamedQualifiedType() : boolean	判断是否带名字类型引用
isParameterizedType() : boolean	判断是否是参数化类型引用
isQualifiedType() : boolean	判断是否带限定类型引用
isTypeReference() : boolean [重定义类NameReference的方法]	返回true
resolveBinding() : boolean [重定义类NameReference的方法]	重定义名字引用解析方法, 当类型名是原子类型或者是自动导入的类型(例如String)时直接在系统作用域解析当前类型引用, 否则在所在作用域解析当前类型引用
setDimension(int) : void	设置引用类型时的数组维数, 缺省值为0
setTypeKind(TypeReferenceKind) : void	设置类型引用类别
toDeclarationString() : String	将类型引用转换为声明时的字符串, 主要是名字再加上适当数目的中括号以处理数组类型
toFullString() : String [重定义类NameReference的方法]	将名字引用转换为用于输出的信息字符串, 包含类别、名字(含适当中括号)、源代码位置、所在作用域等比较详细内容
toMultilineString(int, boolean) : String [重定义类NameReference的方法]	将名字引用转换为用于输出的信息字符串, 除类别、名字(含适当中括号)、源代码位置、所在作用域等信息外, 还可能将包含的子名字引用的信息也都输出, 因此可能有多行信息
toString() : String [重定义类NameReference的方法]	将名字引用转换为类别和唯一标识的比较简单的用于输出的信息串, 但类型名字中含适当中括号
继承自类NameReference的方法(不含已被重定义方法): 参见表5.33	
bindTo(NameDefinition):void; bindedDefinitionToString():String; compareTo(NameReference):int; equals(Object):boolean; getDefinition():NameDefinition; getEnclosingTypeDefintion():TypeDefinition; getLocation():SourceCodeLocation; getName():String; getReferenceKind():NameReferenceKind; getReferencesAtLeaf():List<NameReference>; getResultTypeDefintion():TypeDefinition; getScope():NameScope; getUniqueId():String; hashCode():int; isGroupReference():boolean; isLeftValue():boolean; isLiteralReference():boolean; isMethodReference():boolean; isResolved():boolean; setLeftValueReference():void; setReferenceKind(NameReferenceKind):void	

表 5.36 类ParameterizedTypeReference字段的描述

字段	类型	描述
argumentList	List<TypeReference>	类型参数列表，列表元素是类型引用
primaryType	TypeReference	主类型引用
继承自类TypeReference的字段：参见表5.34		
dimension, typeKind		
继承自类NameReference的字段：参见表5.32		
definition, kind, location, name, scope		

个to...String()方法以及toDeclaration()方法则在输出名字时不仅给出主类型的名字，而且使用尖括号给出类型参数名字。

5.3.4 类NamedTypeReference和类QualifiedTypeReference

类NamedTypeReference是类TypeReference的直接子类，它的实例代表一个带名字的类型引用，即一个名字引用（使用圆点隔开）后跟一个类型引用。类QualifiedTypeReference也是类TypeReference的直接子类，它的实例代表一个带限定的类型引用，即一个类型引用（使用圆点隔开）后跟一个类型引用。

实际上，带名字的类型引用、带限定的类型引用，以及带限定的名字（类NRGQualifiedNames的实例）都对应由两个或多个使用圆点隔开的标识符构成的字符串，如cn.com.MyClass.InternalClass。对于这种字符串，Eclipse JDT在生成抽象语法树时可能有足够的信息来区别它是对类型的引用还是对字段的引用，但也可能没有足够的信息，仅仅能确定它是名字引用。如果在生成抽象语法树节点能确定是类型引用，则相应地我们生成类NamedTypeReference或QualifiedTypeReference的实例，如果是对字段的引用则生成NRGFieldAccess的实例，其他情况则生成类NRGQualifiedNames的实例。具体是生成类NamedTypeReference还是QualifiedTypeReference的实例又要取决于在生成抽象语法树时能否确定最后一个圆点前的名字串是否是类型引用，如果能确定则生成后者的实例，否则生成前者的实例。表4.8给出了不同的抽象语法树节点所对应的不同的类型引用。

表5.38给出类NamedTypeReference的字段。我们使用字段fullQualifiedName存放带名字类型引用的全限定名，例如对于源代码中出现的名字引用cn.com.myClass.InternalClass，该字段存放整个字符串"cn.com.myClass.InternalClass"，而继承自类NameReference的字段name仅仅存放字符串"InternalClass"。字段qualifier则存放对应名字cn.com.myClass部分的名字引用（注意这个名字引用又可能是带名字的类型引用）。

在生成类NamedTypeReference实例时我们将获取该带名字类型引用对应的全限定名。保存该全限定名的作用是在解析该带名字类型引用时我们可先尝试基于整个全限定名在所属作用域进行匹配，尝试不成功再解析字段qualifier对应的名字引用，该名字引用所绑定的结果应该是程序包或类型，然后再在得到的程序包或类型中匹配字段name所保存的名字。目前类NamedTypeReference的方法resolveBinding()就是使用这样的实现方法。

为什么要先基于整个全限定名进行匹配呢？这是因为我们的程序包定义并没有分层存放，也即对于名字cn.com.myClass，可能并没有对应cn的程序包定义，而只有对应cn.com的程序包定义（当程序包cn下没有实际的类时我们就不会创建对应cn的程序包定义），因此先解析前面的限定名cn是

表 5.37 类ParameterizedTypeReference方法的描述

ParameterizedTypeReference(String, SourceCodeLocation, NameScope) : Constructor 构造方法，以名字、源代码位置、所在作用域为参数
ParameterizedTypeReference(ParameterizedTypeReference) : Constructor 拷贝构造方法，基于另一个参数化类型引用构造新的示例
addArgument(TypeReference) : void 添加类型参数引用
getArgumentList() : List<TypeReference> 返回类型参数列表，列表元素是类型引用
getDefinition(boolean) : List<TypeDefinition> 当给定参数为true时返回主类型和所有类型参数构成的列表，为false返回只有主类型构成的列表
getPrimaryType() : TypeReference 返回主类型（引用）
getPrimaryTypeDefinition() : TypeDefinition 返回主类型引用所能绑定到的类型定义
getReferencesAtLeaf() : List<NameReference> [重定义类NameReference的方法] 返回主类型以及类型参数中所包含的所有基本类型引用构成的列表
resolveBinding() : boolean [重定义类TypeReference的方法] 解析当前类型引用，依次解析主类型和所有类型参数，但整个类型引用绑定到主类型所能绑定到的类型定义
setArgumentList(List<TypeReference>) : void 设置类型参数列表
setPrimaryType(TypeReference) : void 设置主类型引用
toDeclarationString() : String [重定义类TypeReference的方法] 转换为类型声明形式的字符串，主要是给出主类型以及尖括号括起来的多个类型参数
toFullString() : String [重定义类TypeReference的方法] 将名字引用转换为用于输出的信息字符串，包含类别、名字（含用尖括号括起来的类型参数）、源代码位置、所在作用域等 比较详细内容
toMultilineString(int, boolean) : String [重定义类TypeReference的方法] 将名字引用转换为用于输出的信息字符串，除类别、名字（含用尖括号括起来的类型参数）、源代码位置、所在作用域等信息外，还可能将包含的子名字引用的信息也都输出，因此可能有多行信息
toString() : String [重定义类TypeReference的方法] 将名字引用转换为类别和唯一标识的比较简单的用于输出的信息串，但类型名字中含用尖括号括起来的类型参数
继承自类TypeReference的方法（不含已被重定义方法）：参见表5.35 getDimension():int; getTypeKind():TypeReferenceKind; isArrayType():boolean; isNamedQualifiedType():boolean; isParameterizedType():boolean; isQualifiedType():boolean; isTypeReference():boolean; setDimension(int):void; setTypeKind(TypeReferenceKind):void
继承自类NameReference的方法（不含已被重定义方法）：参见表5.33 bindTo(NameDefinition):void; bindedDefinitionToString():String; compareTo(NameReference):int; equals(Object):boolean; getDefinition():NameDefinition; getEnclosingTypeDefinition():TypeDefinition; getLocation():SourceCodeLocation; getName():String; getReferenceKind():NameReferenceKind; getResultTypeDefinition():TypeDefinition; getScope():NameScope; getUniqueId():String; hashCode():int; isGroupReference():boolean; isLeftValue():boolean; isLiteralReference():boolean; isMethodReference():boolean; isResolved():boolean; setLeftValueReference():void; setReferenceKind(NameReferenceKind):void

不会成功的，而必须至少cn.com一起解析才可能成功。所以先尝试整个全限定名进行解析可以避免因程序包定义没有分层存放而掉入不存在的程序包中导致解析不成功。程序包定义无法分层存放，因为从Java程序作用域组织看，子程序包不是程序包的子作用域！

表5.38给出类NamedTypeReference的方法，这个类的构造方法必须给出全限定名作为参数之一，而必须重定义类NameReference的方法resolveBinding()以实现上面所说的名字解析过程。

表 5.38 类NamedTypeReference字段的描述

字段	类型	描述
fullQualifiedName	String	这个带名字类型引用的全限定名
qualifier	NameReference	除最后的简单名的限定名构成的名字引用
继承自类TypeReference的字段：参见表5.34		
dimension, typeKind		
继承自类NameReference的字段：参见表5.32		
definition, kind, location, name, scope		

表 5.39 类NamedTypeReference方法的描述

NamedTypeReference(String, String, SourceCodeLocation, NameScope) : Constructor 构造方法，以简单名、全限定名、源代码位置、所在作用域为参数
NamedTypeReference(NamedTypeReference) : Constructor 拷贝构造方法，拷贝另一个带名字类型引用的实例构造一个新实例
getQualifier() : NameReference 返回限定名部分构成的名字引用
resolveBinding() : boolean [重定义类TypeReference的方法] 解析带名字的类型引用，先基于全限定名作为整体在当前作用域进行解析，不成功再解析限定名构成的名字引用，并基于其结果（程序包定义或类型定义中）匹配字段name中的名字
setQualifier(NameReference) : void 设置限定名部分构成的名字引用
继承自类TypeReference的方法（不含已被重定义方法）：参见表5.35
getDimension():int; getTypeKind():TypeReferenceKind; isArrayType():boolean; isNamedQualifiedType():boolean; isParameterizedType():boolean; isQualifiedType():boolean; isTypeReference():boolean; setDimension(int):void; setTypeKind(TypeReferenceKind):void toDeclarationString():String; toFullString() : String; toMultilineString(int, boolean):String; toString():String
继承自类NameReference的方法（不含已被重定义方法）：参见表5.33
bindTo(Definition):void; bindedDefinitionToString():String; compareTo(NameReference):int; equals(Object):boolean; getDefinition():NameDefinition; getEnclosingTypeDefinition():TypeDef; getLocation():SourceCodeLocation; getName():String; getReferenceKind():NameReferenceKind; getResultTypeDefinition():TypeDef; getScope():NameScope; getUniqueId():String; hashCode():int; isGroupReference():boolean; isLeftValue():boolean; isLiteralReference():boolean; isMethodReference():boolean; isResolved():boolean; setLeftValueReference():void; setReferenceKind(NameReferenceKind):void

目前类QualifiedTypeReference的实现与类NamedTypeReference的实现基本相同，图5.9给出类QualifiedTypeReference的成员（不含继承成员），可以看到，除字段qualifier的类型是TypeReference，构造方法名称不同外，其他都与类NamedTypeReference相同，而且目前它的方法resolvebinding()与

类QualifiedTypeReference的同名方法的实现也相同，不管qualifier的类型是什么，我们都在它所属的作用域内进行解析，然后绑定的结果都要么是程序包要么是类型定义（如果是其他名字定义都不会是合法的Java程序，而我们假定所分析的程序是合法的Java程序）。

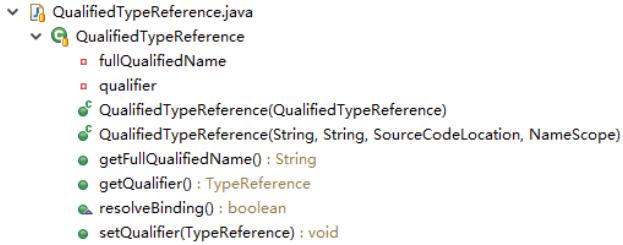


图 5.9 类QualifiedTypeReference的成员

5.3.5 类IntersectionTypeReference和类UnionTypeReference

类IntersectionTypeReference和类UnionTypeReference都是类TypeReference的直接子类，其实例分别代表交集类型和并集类型，即几个类型做交集运算和并集运算。交集类型和并集类型是Java最新引入的特性，在实际编程实践中还没有太多应用，因此目前这两个类的实现也比较简单，而且在目前的类型推导中也没有发挥作用，即在匹配方法的形式参数类型和实际参数类型时我们忽略交集类型或并集类型，实际上我们只关心其中第一个参与运算的类型引用，因此交集类型引用或并集类型引用目前在名字解析时都绑定到第一个参与运算的类型引用所能绑定到的类型定义。

图5.10给出这两个类的成员，可以看到这两个类有相同的成员，字段typeList给出参与交集运算或并集运算的类型（引用）列表。构造方法中的名字则记录在程序源代码中出现的整个交集类型或并集类型（例如"Class1 & Class2 & Class3"等），它们的resolveBinding()方法都是讲参与运算的所有类型引用都进行解析，然后将整个交集或并集类型引用绑定到第一个参与运算的类型引用所绑定到的类型定义。

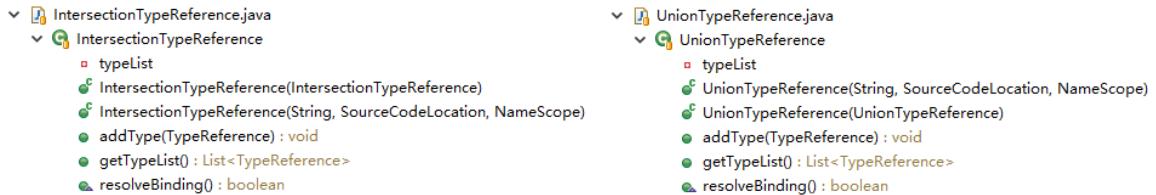


图 5.10 类IntersectionTypeReference和类UnionTypeReference的成员

5.3.6 类WildcardTypeReference

类WildcardTypeReference是类TypeReference的直接子类，它的实例代表一个通配符类型引用，即使用尖括号括起来，其中含有问号'?'的类型引用，并且可以使用extends或super给出它的上界或下界。实际上，如果没有界（即只有问号）则没有引用任何名字，我们不会为其创建类WildcardTypeReference的实例，因此目前类WildcardTypeReference主要是记录其中的界，并使用标记区分是上界还是下界。对于通配符类型引用在类型推导中的作用我们还需进一步研究，因此目前通配符类型引用也没有在类型推导中发挥作用，即在匹配方法的形式参数类型和实际参数

类型时我们暂时忽略通配符类型引用，而在名字引用解析时，通配符类型引用会绑定到它的界类型（引用）所能绑定到的类型定义。

图5.11给出类WildcardTypeReference的成员，它的字段主要是界类型引用bound以及布尔类型标记isUpperBound标记这个界类型引用是否上界（即是否是使用关键字extends声明）。

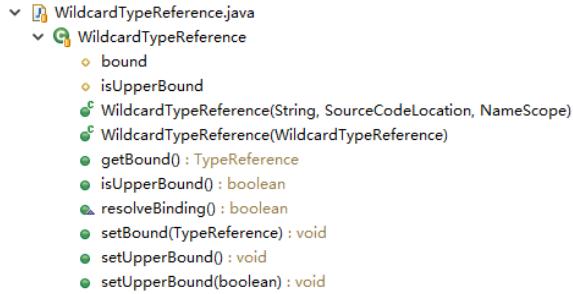


图 5.11 类WildcardTypeReference的成员

类WildcardTypeReference的构造方法的字符串参数记录程序源代码中通配符类型引用的整个字符串，例如“<? extends Collection>”，作为通配符引用的名字保存在字段name中。这个类重定义方法resolveBinding()，它目前的实现是解析界类型引用，并将整个通配符类型引用绑定到界类型引用所绑定到的类型定义。

5.3.7 类MethodReference

类MethodReference是类NameReference的直接子类，它的实例代表一个方法引用，也即代表对一个方法的调用。在Java程序中，调用方法除给出方法名之外，还要给出方法实际参数，对于类属方法（泛型方法）还要给出实际类型参数。方法实际参数可能是表达式，因此我们使用名字引用（组）表示，实际类型参数是类型表达式，因此使用类型引用表示。

表5.40给出类MethodReference的字段，主要是包含实际参数列表（元素是名字引用）argumentList、实际类型参数列表（元素是类型引用）typeArgumentList。字段alternativeList则给出方法引用所能绑定到的所有方法定义列表。由于多态性，一个方法调用在编译期间可能确定调用某个方法，但是在运行期间实际调用该方法在子类中的重定义方法，从而从程序静态分析角度看，一个方法调用可能绑定到多个方法定义，因此我们使用alternativeList保存可能绑定的多个方法定义，其中第一个是根据调用方法的对象的静态类型所确定的方法，这个方法也保存在字段definition（继承自类NameReference），列表中其他方法都是第一个方法在后代类中的重定义方法。

表 5.40 类MethodReference字段的描述

字段	类型	描述
alternativeList	List<MethodDefinition>	方法引用所能绑定到的所有方法定义列表
argumentList	List<NameReference>	方法（实际）参数列表，每个参数是一个名字引用（表达式）
typeArgumentList	List<TypeReference>	方法（实际）类型参数类别，每个类型参数是一个类型引用
继承自类NameReference的字段：参见表5.32		
definition, kind, location, name, scope		

表5.41给出类MethodReference的方法，这些方法主要是它的字段的getter和setter方法，其

中方法`getReferencesAtLeaf()`除了将方法引用本身添加到返回的列表之外，还将方法引用中保存的方法参数列表和方法类型参数列表中的所有元素添加到返回的列表。类`MethodReference`没有重定义名字解析方法`resolveBinding()`，因为方法引用本身的解析只需在它所属作用域内（通常是类型声明作用域）进行匹配即可，不过这种匹配最后会调用类`MethodDefinition`的`matchMethod()`方法，不仅匹配方法名，而且会匹配方法的参数（不过目前的实现暂时没有考虑方法的类型参数的匹配）。

表 5.41 类`MethodReference`方法的描述

<code>MethodReference(String, SourceCodeLocation, NameScope) : Constructor</code>	构造方法，以方法引用的名字、源代码位置、所在作用域为参数
<code>addAlternative(MethodDefinition) : void</code>	添加该方法引用能绑定到方法定义，由于多态性，一个方法引用可能调用（绑定到）多个方法定义
<code>addAlternative(List<MethodDefinition>) : void</code>	添加该方法引用能绑定到的多个方法定义
<code>getAlternativeList() : List<MethodDefinition></code>	返回该方法引用所能绑定到的多个方法定义
<code>getArgumentList() : List<NameReference></code>	返回方法实际参数列表，每个实际参数是一个名字引用（表达式）
<code>getReferencesAtLeaf() : List<NameReference> [重定义类NameReference的方法]</code>	返回方法引用所包含的所有名字引用，包括对方法本身的引用、方法参数和类型参数的引用所构成的列表
<code>getTypeArgumentList() : List<TypeReference></code>	返回方法实际类型参数列表，每个类型参数是一个类型引用
<code>isMethodReference() : boolean [重定义类NameReference的方法]</code>	返回true
<code>setArgumentList(List<NameReference>) : void</code>	设置方法的实际参数列表
<code>setTypeArgumentList(List<TypeReference>) : void</code>	设置方法的实际类型参数列表
继承自类<code>NameReference</code>的方法（不含已被重定义方法）：参见表5.33	
<code>bindTo(NameDefinition):void; bindedDefinitionToString():String;</code>	
<code>compareTo(NameReference):int; equals(Object):boolean; getDefinition():NameDefinition;</code>	
<code>getEnclosingTypeDefinition():TypeDefintion; getLocation():SourceCodeLocation;</code>	
<code>getName():String; getReferenceKind():NameReferenceKind; getResultTypeDefinition():TypeDefintion;</code>	
<code>getScope():NameScope; getUniqueId():String; hashCode():int;</code>	
<code>isGroupReference():boolean; isLeftValue():boolean; isLiteralReference():boolean;</code>	
<code>isResolved():boolean; isTypeReference():boolean; setLeftValueReference():void;</code>	
<code>setReferenceKind(NameReferenceKind):void; toFullString():String</code>	
<code>toMultilineString(int, boolean):String; toString():String</code>	

5.3.8 类`ValueReference`

类`ValueReference`是类`NameReference`的直接子类，它的实例代表一个值引用，也即对程序中一片内存区域的使用。在程序中有时是使用一片内存区域的地址，即想要将数据存放到这一片内存区域，这时它称为左值引用（由于可放在赋值运算符的左边而得名），也有时是读取这一片内存区域存放的数据值，这时称为右值使用。

图5.12给出类`ValueReference`的成员，它的实现比较简单，除继承自类`NameReference`的成员

外，字段`isLeftValue`是布尔类型的标记，标记是否是左值引用，相应地有该字段的`setter`和`getter`方法，其中方法`setLeftValueReference()`重定义类`NameReference`的同名方法，实际上只有在这个类这个方法才真正设置是否左值引用。类`ValueReference`也重定义方法`toMultilineString()`，主要作用是在输出时标记值引用是左值引用还是右值引用。

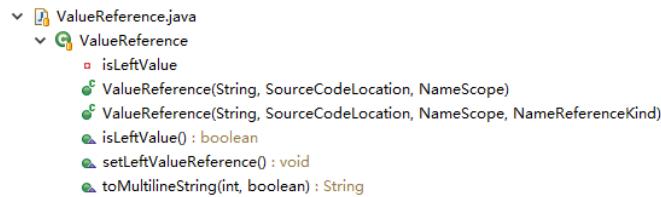


图 5.12 类ValueReference的成员

类ValueReference的一个构造方法以引用时的名字、所在源代码、所在作用域和引用类别作为参数，之所以引用类别也作为构造方法的参数，是因为值引用有时能确定就是引用字段（例如在字段名字引用组NRGFieldAccess中），但有时不能确定，能确定是引用字段，则值引用的类别是NRK_FIELD，否则就是NRK_VARIABLE，缺省时是NRK_VARIABLE，不带引用类别为参数的构造方法即使用该缺省值。

类ValueReference没有重定义名字解析方法resolveBinding(), 因为对于值引用, 使用类NameReference 的缺省实现, 即调用所在作用域的resolve()方法匹配值引用中的名字即可。

5.3.9 类LiteralReference

类LiteralReference是类NameReference的直接子类，它的实例代表对文字常量的引用。严格意义上说，文字常量的使用不是名字引用，但是在表达式（对应的名字引用组中）我们可能需要文字常量的类型来推导整个表达式的类型，因此需要在名字引用组中保存文字常量的引用。因此，只有在名字引用组中才会出现LiteralReference的实例，我们不会为单独的某个文字常量创建类LiteralReference的实例。

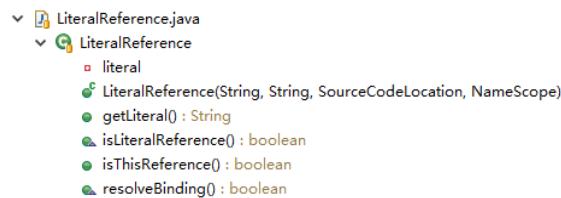


图 5.13 类 LiteralReference 的成员

图5.13给出类LiteralReference的成员，这个类的设计非常简单，它使用字符串类型的字段literal存放文字常量的字面值（例如对于整数常量200，存放"200"）。我们不使用继承自NameReference的字段name存放字面值，因为在解析时我们使用字段name匹配名字定义，而对于文字常量，我们需要它的类型以便推导它所在表达式的结果类型，因此我们将文字常量绑定到文字常量的类型（所对应的类型定义，而非它的值），所以类LiteralReference的name字段存放的是文字常量的类型（Eclipse JDT在生成抽象语法树节点时会提供文字常量的类型信息）。

类LiteralReference的实例还可能代表关键字this。出现在Java程序中的关键字this也被看做是一种特殊的常量，用于代表当前对象。当关键字this单独出现时我们不为之创建名字引用（正

如单独出现的文字常量也不创建名字引用), 但在表达式中的关键字this(例如this.value)需要创建名字引用以便解析整个表达式(只有解析this之后才能解析后面的value这个名字引用), 因此这时我们也创建一个类LiteralReference表示对应this的名字引用, 它的字段name和literal都存放"this"这个字符串, 方法isThisReference()用于判断一个文字常量引用是否是代表关键字this的引用。

类LiteralReference重定义类NameReference的名字解析方法resolveBinding(): 对于代表关键字this的引用, 查找包含这个引用的最里层的详细类型定义, 认为this指代这个详细类型定义, 因此将这个引用绑定到这个详细类型定义; 对于其他的文字常量引用, 则在系统作用域匹配与字段name相同的导入类型定义(因为文字常量的类型要么是原子类型, 要么是String类型)。

5.3.10 类PackageReference

类PackageReference是类NameReference的直接子类, 它的实例代表对程序包的引用。实际上, 我们只会为按需导入声明创建类PackageReference的实例, 因为只有在这时能从语法上确定是对程序包的引用。Java程序其他地方对程序包名字的使用通常是和类型引用或字段引用在一起, 例如java.lang.String等, 程序其他地方不会单独使用程序包名字, 而对于与类型引用或字段引用在一起的程序包使用, Eclipse JDT在生成抽象语法树时会根据情况生成带名字的类型引用, 或者字段引用, 或者带限定的名字等语法树节点, 我们也相应地创建类NamedTypeReference, NRGFieldAccess或NRGQualifiedName的实例, 这时程序包的引用会作为这些实例的一部分, 而且直接使用类NameReference的实例存放程序包的名字, 因为在生成抽象语法树节点时没有明确确定这一部分名字就是程序包名字, 我们需要在绑定时通过匹配才能确定, 所有在生成名字引用时是直接使用类NameReference的实例。

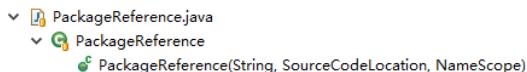


图 5.14 类PackageReference的成员

因此, 类PackageReference的实现非常简单, 仅仅是提供一个构造方法来标记一个引用是程序包引用。图5.14给出它的成员, 它的构造方法以引用时的名字、源代码位置和所在作用域为参数。

5.3.11 类NameReferenceGroup及其子类

类NameReferenceGroup是类NameReference的直接子类, 它的实例代表名字引用组。名字引用组按照Java程序的表达式的语法结构将多个名字引用构成一组, 从而一些名字引用可以为另外一些名字引用提供名字解析的上下文。

类NameReferenceGroup是代表名字引用组的类的基类。我们为每个Java语言表达式(准确地说, 是对应抽象语法树的每个表达式节点, 即抽象语法树类Expression的每个子类)都设计一个名字引用组子类。类NameReferenceGroup是抽象类。类NameReferenceGroup及其子类都在程序包nameTable.nameReference.referenceGroup中。

表5.42给出类NameReferenceGroup的字段, 它为Java语言的每个运算符定义一个对应的字符串常量, 因为在推断某些类型表达式(名字引用组)的结果类型(名字引用需要绑定到结果类型所对应的类型定义)需要表达式中的运算符(例如如果是算术运算符, 则结果通常是int或double这种数

值类型，而如果是逻辑运算符或关系运算符，则结果是boolean等）。字段operator存放当前名字引用组的运算符，当然这个字段只对某些名字引用组是有用的。注意，每个名字引用组至多有一个运算符（因为名字引用组是按照表达式的语法结构组织，对应每个含有运算符的表达式都有相应的名字引用组）。字段subreferences给出它的子名字引用，子名字引用还可能是名字引用组，从而对应表达式的语法结构（通常是树状结构）。

例如，对于表达式this.value*x+2，整个表达式对应类NRGInfixExpression（中缀表达式）的实例，子表达式this.value*x又对应一个类NRGInfixExpression的实例，常量2对应一个类LiteralReference的实例，这两个实例是整个表达式对应实例的子名字引用。对应子表达式this.value*2的名字引用组又有两个实例，前者对应this.value，是类NRGFieldAccess的实例，后者对应x，是类ValueReference的实例。最后对应this.value的名字引用组有两个子名字引用，分别是对应this的LiteralReference实例和对应value的ValueReference实例。

表 5.42 类NameReferenceGroup字段的描述

字段	类型	描述
OPERATOR_AND	String	静态字符串常量（值为“&”），表示逻辑与 类NameReference为Java语言的每个运算符都定义了字符串常量，这里不一一列出
OPERATOR_XOR	String	静态字符串常量（值为“^”），表示逻辑异或
operator	String	以字符串表示的当前名字引用组的运算符，值为上述常量之一
subreferences	List<NameReference>	名字引用组的子名字引用列表
继承自类NameReference的字段：参见表5.42		
definition, kind, location, name, scope		

表5.43给出类NameReferenceGroup的方法，它的构造方法以名字引用组所对应的源代码字符串、源代码位置和所在作用域为参数。源代码字符串存放在继承自类NameReference的字段name中，因此名字引用组对应的整个表达式存放在字段name，而它的子表达式则存放在它的子名字引用的字段name中。类NameReferenceGroup实现名字引用组的共有的方法，包括字段operator和subreferences的getter和setter方法，以及重定义NameReference的方法以在输出引用信息和绑定信息时按名字引用组的语法结构进行输出。

类NameReferenceGroup的方法getReferencesAtLeaf()返回处于名字引用组语法结构中叶子节点位置的名字引用，这些引用被看做是基本的名字引用。例如，对于表达式this.value*x+2，它返回对应this,value,x和2的名字引用，这些名字引用处于这个表达式结构的叶子节点位置。

类NameReferenceGroup的方法resolveBinding()是抽象方法，它的子类必须重定义这个方法以完成对具体名字引用组的解析。实际上，类NameReferenceGroup的所有子类的实现主要就是重载这个方法。图5.15给出子类NRGArrayAccess的成员，它的构造方法的参数与类NameReferenceGroup相同（实际上该构造方法的实现就是调用类NameReferenceGroup的构造方法），方法getGroupKind()返回相应的名字引用组类别（枚举类型NameReferenceGroupKind的常量），最后方法resolveBinding()根据数组访问的语法结构进行名字引用解析。

类NameReferenceGroup的其他子类的实现也基本相同，因此不再这里赘述。类NameReferenceGroup的完整的子类列表可参看图5.8，而不同子类，即不同名字引用组的解析方法我们已经在设计阶段有详细说明（参见4.2.7节）。

表 5.43 类NameReferenceGroup方法的描述

<code>NameReferenceGroup(String, SourceCodeLocation, NameScope) : Constructor</code>	构造方法，以名字引用组所对应的源代码（表达式）字符串、源代码位置、所在作用域为参数
<code>addSubReference(NameReference) : void</code>	添加子名字引用到名字引用组
<code>bindedDefinitionToString() : String [重定义类NameReference的方法]</code>	按照名字引用组的语法结构输出它及其子名字引用的绑定信息（可能是多行信息）
<code>getGroupKind() : NameReferenceGroupKind</code>	返回名字引用组的组类别
<code>getOperator() : String</code>	返回名字引用组的运算符
<code>getReferenceKind() : NameReferenceKind [重定义类NameReference的方法]</code>	返回NameReferenceKind.NRK_GROUP
<code>getReferencesAtLeaf() : List<NameReference> [重定义类NameReference的方法]</code>	返回在名字引用组语法结构中处于叶子节点的所有名字引用构成的列表，这些是整个组中最基本的名字引用
<code>getSubReferenceList() : List<NameReference></code>	返回子名字引用列表
<code>isArithematicOperator() : boolean</code>	判断当前名字引用组存放的运算符是否是算术运算符，有时需要基于运算符的种类推断表达式的类型
<code>isGroupReference() : boolean [重定义类NameReference的方法]</code>	返回true
<code>isShiftOperator() : boolean</code>	判断当前名字引用组存放的运算符是否是移位运算符，有时需要基于运算符的种类推断表达式的类型
<code>resolveBinding() : boolean [重定义类NameReference的方法]</code>	抽象方法，具体的名字引用组子类必须重定义这个方法以对名字引用组进行解析
<code>setLeftValueReference() : void [重定义类NameReference的方法]</code>	设置整个名字引用组为左值引用，实际上是设置处于叶子节点的值引用为左值引用
<code>setOperator(String) : void</code>	设置名字引用组的运算符
<code>toFullString() : String [重定义类NameReference的方法]</code>	将名字引用组转换为信息比较完整的输出字符串（可能多行），会根据名字引用组的语法结构输出所有的子名字引用
<code>toMultilineString(int, boolean) : String [重定义类NameReference的方法]</code>	将名字引用组转换为信息最为完整的输出字符串（可能多行），会根据名字引用组的语法结构输出所有的子名字引用
<code>toString() : String [重定义类NameReference的方法]</code>	将名字引用组转换为信息比较简略的输出字符串（可能多行），会根据名字引用组的语法结构输出所有的子名字引用
继承自类NameReference的方法（不含已被重定义方法）：参见表5.33	
<code>bindTo(NameDefinition):void; compareTo(NameReference):int; equals(Object):boolean;</code>	
<code>getDefinition():NameDefinition; getEnclosingTypeDefinition():TypeDefintion;</code>	
<code>getLocation():SourceCodeLocation; getName():String; getResultTypeDefinition():TypeDefintion;</code>	
<code>getScope():NameScope; getId():String; hashCode():int; isLeftValue():boolean;</code>	
<code>isLiteralReference():boolean; isMethodReference():boolean;</code>	
<code>isResolved():boolean; isTypeReference():boolean; setReferenceKind(NameReferenceKind):void;</code>	



图 5.15 类NRGArrayAccess的成员

5.4 名字表使用

构件使用者主要是通过类NameTableManager使用名字表，类NameTableASTBridge则辅助类NameTableManager 提供名字表与抽象语法树之间的桥接功能，构件使用者在需要考察名字表的实体（主要是名字定义和名字作用域）与抽象语法节点对应关系时才需要类NameTableASTBridge的功能。这两个类在程序包nameTable中。

为辅助类NameTableManager遍历和查找名字表实体（名字定义、作用域和名字引用），我们参照访问器模式设计访问器类对名字表进行访问，所有的访问器类都在程序包nameTable.visitor中。为了在访问时可以设置各种条件，我们设计过滤器配合访问器遍历和查找满足过滤条件的名字表实例，所有过滤器类都在程序包nameTable.filter中。

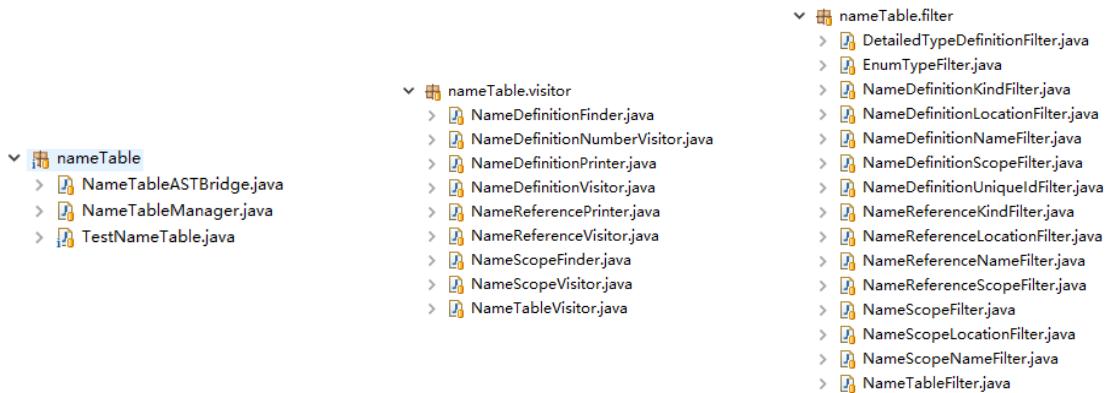


图 5.16 与名字表使用相关的程序包及类

图5.16给出与名字表使用相关的类，包括访问器类和过滤器类。下面我们先给出访问器类和过滤器类的实现，然后再给出类NameTableManager和类NameTableASTBridge的实现。

5.4.1 名字表访问器类

类NameTableVisitor是所有代表名字表访问器类的基类，程序包nameTable.visitor中的其他类都是这个类的直接子类。

类NameTableVisitor参照Eclipse JDT的类ASTVisitor的设计，分别设计方法preVisit()实现在访问节点前的统一操作、方法postVisit()实现在访问节点后的统一操作、方法visit()实现访问不同节点的操作以及方法endVisit()实现访问节点之后的操作，不同节点可以有不同的操作。对于名字表访问而言，由于名字表实体按作用域存放，节点就是作用域，因此类NameTableVisitor对不同作用域类别（系统作用域、程序包定义、类型定义、访问定义和局部作用域）有不同的visit()和endVisit()方法。注意，我们针对详细类型定义、枚举类型定义和导入类型定义分别有visit()和endVisit()方法。

根据访问器模式的设计思想，每个节点（即作用域）都实现accept()方法，该方法以访问器类的实例为参数，它先调用访问器类的preVisit()方法，然后再调用visit()方法。若visit()方法返回true，则针对作用域的每个子作用域再调用accept()方法访问子作用域；若visit()方法返回false，则不访问子作用域。最后作用域的accept()方法再调用endVisit()方

法和postVisit()方法，完成访问当前节点之后的工作。因此，实质上整个遍历是前序遍历，当然如果每个节点都将实质性访问工作放在endVisit()中也可实现后序遍历。

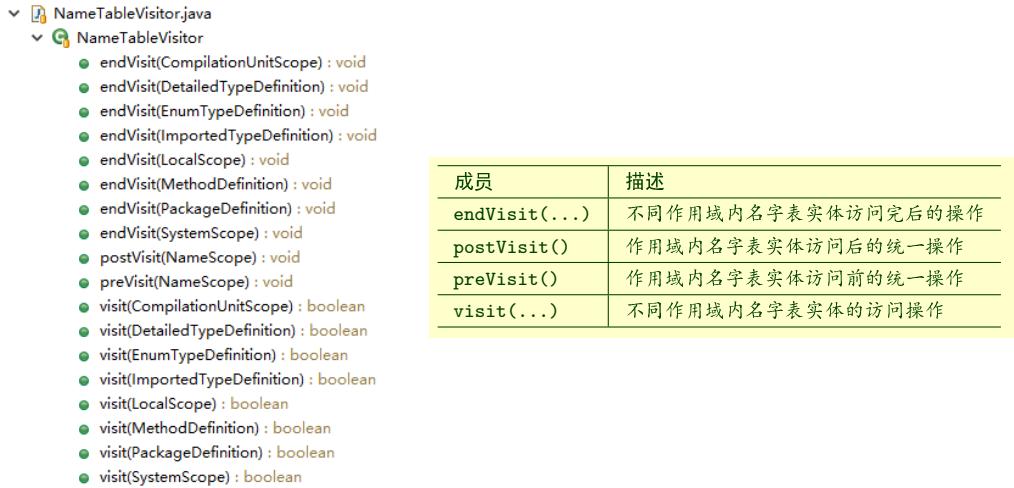


图 5.17 类NameTableVisitor定义的方法

对于类NameTableVisitor而言，它对这些方法的实现都是没有做任何实质性工作（因为它是基类，还没有确定到底访问怎样的名字表实体），对于visit()方法，都是直接返回true，对于其他方法则只有语句return。注意，按照访问器设计模式，方法visit()返回true表示要进一步访问当前节点的子节点，返回false则表示不再访问当前节点的子节点。

对于名字定义的访问，目前实现了四种访问器类：

1. 类NameDefinitionVisitor(): 遍历并返回满足条件的所有名字定义构成的列表；
2. 类NameDefinitionFinder(): 遍历并返回满足条件的第一个名字定义；
3. 类NameDefinitionPrinter(): 遍历并打印满足条件的所有名字定义，打印时会分作用域呈树状形式打印；
4. 类NameDefinitionNumberVisitor(): 遍历并计数满足条件的名字定义个数。

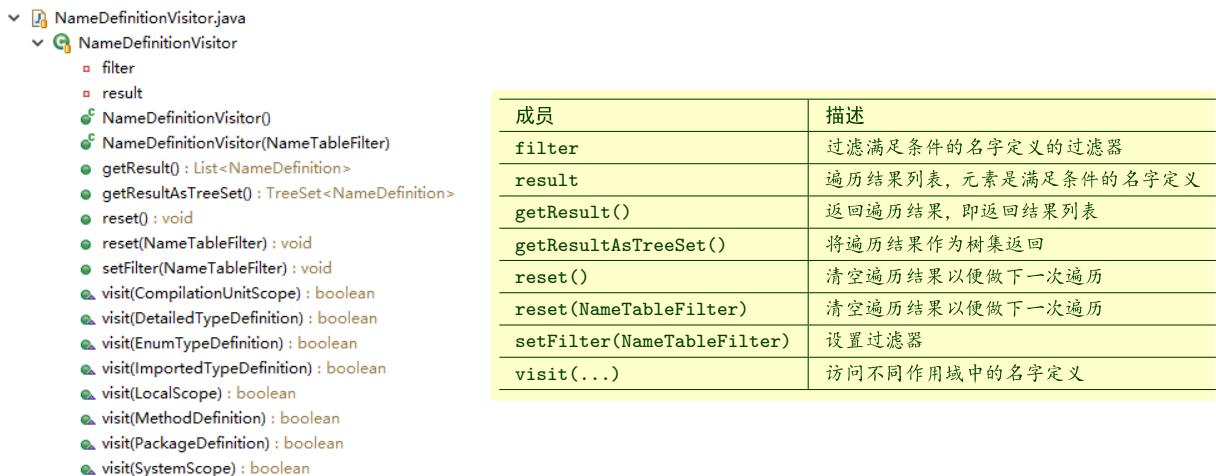


图 5.18 类NameDefinitionVisitor的成员

图5.18给出类NameDefinitionVisitor的成员，它的字段filter存放类NameTableFilter（或其后代类）的实例，用于在遍历过程中过滤满足条件的名字定义。字段filter可以为null，表示没有过滤条件（即所有名字定义都符合要求）。字段result是一个元素为名字定义（类NameDefinition或其后代类的实例）的列表，存放遍历的结果，即获取的满足过程器过滤条件的所有名字定义所构成的列表。

类NameDefinitionVisitor的方法getResult()以列表形式返回遍历结果，而方法getResultSetASTreeSet()则将遍历结果转换为树集返回。以树集形式存放的遍历结果按照名字定义的唯一标识有序存放，而且可以高效地确定某个名字定义是否在遍历结果中。方法reset()清空字段result（甚至重新设置过滤器）以使用同一个访问器实例做新的遍历。类NameDefinitionVisitor重定义各个visit()方法，提取作用域中的名字定义加到遍历结果字段result，并返回true以获取子作用域中的名字定义。

类NameDefinitionFinder的实现与类NameDefinitionVisitor类似，图5.19给出它的成员。与类NameDefinitionVisitor不同的是，类NameDefinitionFinder只要碰到第一个满足过滤条件的名字定义，则将其放在字段result，然后终止对名字表的遍历，所以为了在遇到满足条件的名字定义时更方便终止遍历，类NameDefinitionFinder使用私有方法accept基于过滤器filter判断一个名字定义是否可接受。注意，字段filter可以为null，这时NameDefinitionFinder将返回第一个碰到的名字定义。

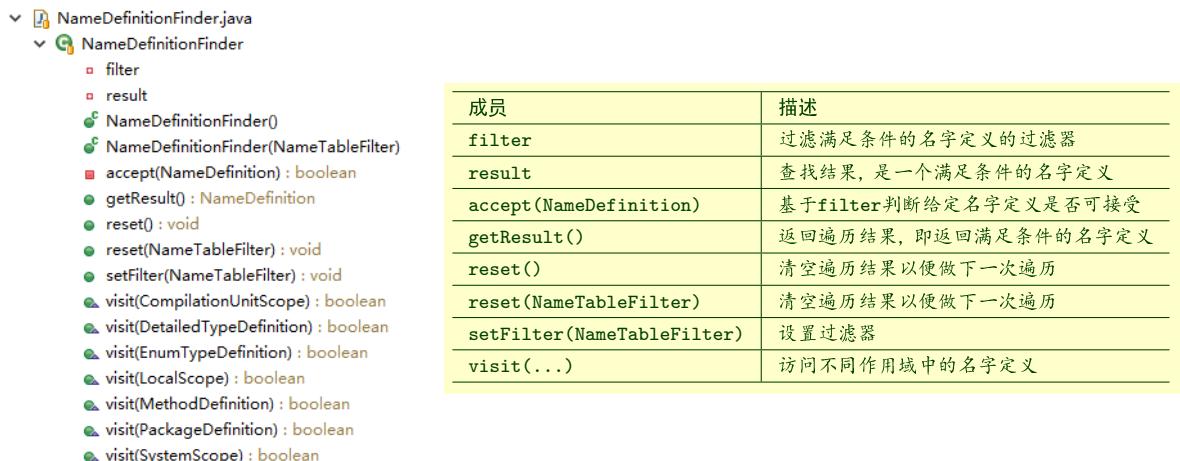
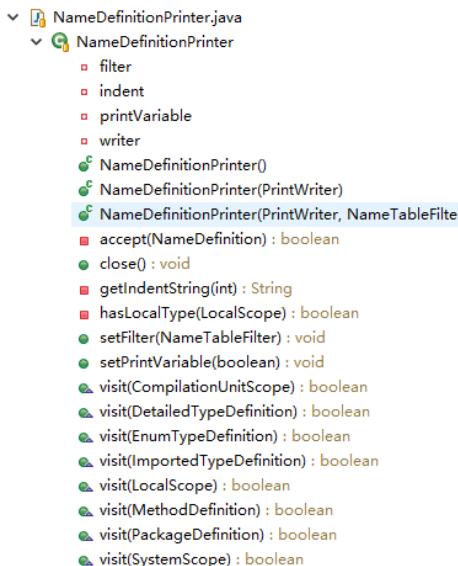


图 5.19 类NameDefinitionFinder的成员

类NameDefinitionNumberVisitor的实现与类NameDefinitionVisitor类似，它遍历名字表，计数满足过滤条件的名字定义数目，因此它的方法getResult()返回遍历结果（即满足条件的名字定义数目），而它的visit()方法则在遍历作用域时计算其中名字定义的数目。除这两点之外，它与类NameDefinitionVisitor完全相同。

类NameDefinitionPrinter的实现也与类NameDefinitionVisitor类似，它遍历名字表，将其中满足过滤条件的名字定义按照所属作用域的层次打印出来。图5.20给出它的成员。由于要按作用域分层次打印，因此字段indent记录在遍历到当前作用域时打印信息缩进的字符数，私有方法getIndentString()则基于这个字符数生成用于缩进的空白字符串。由于变量定义可能非常多，我们设计一个布尔字段printVariable让用户控制是否将局部作用域内的变量打印出来，方

法`setPrintVariable()`设置这个字段的值。当不打印局部作用域变量时，可能整个局部作用域因为内容可打印就不应该输出，所以私有方法`hasLocalType()`用于判断局部作用域是否还有局部类型，如有局部类型则即使不打印变量定义也需要对这个局部作用域进行遍历，否则这个局部作用域在不打印变量定义时就可以不再访问。字段`writer`记录构件使用者所提供的输出流设备，它的类型是`PrintWriter`，方法`close()`用于关闭这个流设备。各个`visit()`方法则根据不同作用域提取其中的名字定义，并根据当前应该缩进的字符数将其中的名字定义的信息分层打印到输出流设备。



成员	描述
<code>filter</code>	过滤满足条件的名字定义的过滤器
<code>indent</code>	打印当前作用域名字定义时的起始缩进字符数
<code>printVariable</code>	是否打印变量定义
<code>writer</code>	打印信息输出流
<code>accept(NameDefinition)</code>	基于 <code>filter</code> 判断是否打印给定的名字定义
<code>close()</code>	关闭打印信息输出流设备
<code>getIndentString(int)</code>	生成指定长度的空白串用于缩进
<code>hasLocalType(LocalScope)</code>	判断局部作用域内是否有局部类
<code>setFilter(NameTableFilter)</code>	设置过滤器
<code>setPrintVariable(boolean)</code>	设置是否打印变量定义
<code>visit(...)</code>	访问并打印不同作用域中的名字定义

图 5.20 类`NameDefinitionPrinter`的成员

对名字作用域的遍历，目前实现了两种访问器类：

1. 类`NameScopeVisitor`: 遍历作用域，获取所有满足过滤条件的作用域；
2. 类`NameScopeFinder`: 遍历作用域，获取满足过滤条件的第一个作用域。

因此这两个类的实现与类`NameDefinitionVisitor`及`NameDefinitionFinder`的实现类似，图5.21给出这两个类的成员。注意到，类`NameScopeVisitor`的方法`getResult()`的返回类型是以名字作用域为元素的列表，而类`NameScopeFinder`的方法`getResult()`则是返回一个名字作用域。类`NameScopeFinder`有私有方法`accept()`判断一个作用域是否满足过滤器条件。这两个类的各`visit()`方法都是在遍历作用域时判断当前作用域是否满足过滤条件，如果满足则类`NameScopeFinder`将其保存在字段`result`，并终止遍历，而类`NameScopeVisitor`将其加入到字段`result`的列表，然后继续遍历当前作用域的子作用域。

对于名字作用域，目前我们没有提供类似类`NameDefinitionPrinter`打印名字作用域的类，因为在打印名字定义时我们会同时分层打印各名字作用域。同样也没有提供对名字作用域的计数。

对于名字引用，目前我们也实现了类`NameReferenceVisitor`和类`NameReferencePrinter`，分析遍历并获取多个满足过滤条件的名字引用，以及分作用域层次打印满足过滤条件的名字引用。这两个类的实现分别与类`NameDefinitionVisitor`和类`NameDefinitionPrinter`类似。

注意到类`NameReferencePrinter`使用两个私有方法`accept()`分别基于过滤器判断是否接受名字定义和名字引用，之所以判断名字定义是否接受，是因为类型定义、方法定义、程序包定义等本身既是名字定义，也是作用域。我们目前的实现允许过滤器`filter`对这些定义也

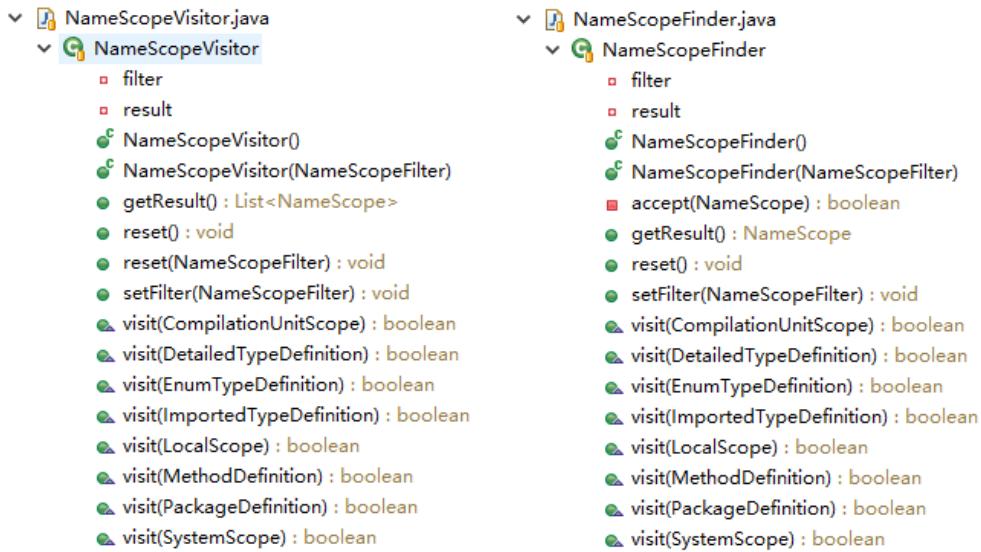


图 5.21 类NameScopeVisitor和类NameScopeFinder的成员

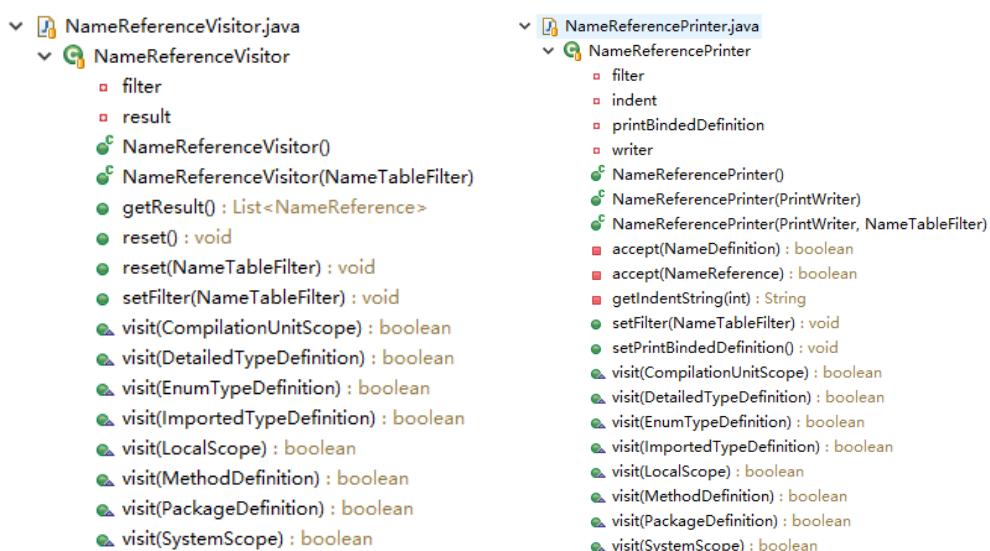


图 5.22 类NameReferenceVisitor和类NameReferencePrinter的成员

进行过滤，也即可打印指定程序包、类型和方法中的名字引用。类NameReferencePrinter的字段printBindedDefinition也是一个布尔型标记，用于指定在打印名字引用时是否也打印该引用所绑定的名字定义信息。

注意，只有在同时生成名字定义、作用域和名字引用时，各作用域才会存放名字引用，这时才能使用类NameReferenceVisitor 或类NameReferencePrinter通过遍历名字表（实际上就是遍历名字作用域）访问和打印名字引用。当只生成名字定义和名字作用域时不能使用者两个类对名字引用进行访问和打印。在这种情况下，名字引用是使用类NameReferenceCreator 临时生成并返回给构件使用者，这时构件使用者再对得到的名字引用列表进行访问和打印。

最后，需要说明的是，目前虽然只提供了以上访问器类，但很容易根据需要扩展类NameTableVisitor，基于名字定义、名字作用域和名字引用的属性和方法，实现更多的访问器类（正如我们扩展类ASTVisitor实现更多的对抽象语法树节点的访问功能）。

5.4.2 名字表过滤器类

名字表过滤器类有两个基本的类，类NameTableFilter和类NameScopeFilter。类NameTableFilter是所有用于过滤名字定义和名字引用的过滤器类的基类，而类NameScopeFilter是用于过滤名字作用域的过滤器类的基类。之所以有两个基本类，而不是一个，是因为有些名字定义也是名字作用域，它既可作为名字定义进行过滤也可作为作用域进行过滤，因此我们设计两个不同的类来明确区别对这种名字定义是按名字作用域进行过滤还是按名字定义进行过滤。

图5.23给出这两个类的实现。这两个类的实现很简单，类NameTableFilter定义两个accept()方法，分别表示是否接受一个名字定义和名字引用，而类NameScopeFilter 定义一个accept()方法，表示是否接受一个名字作用域。在这两个类中，这些方法的缺省实现都是返回false（表示不接受），具体的过滤器类根据所设置的条件重定义这些accept()方法。



图 5.23 类NameTableFilter和类NameScopeFilter的成员

具体的名字表过滤器类使用装饰模式，即它可包装另外一个过滤器类的实例，从而形成过滤器链，所遍历的名字表实体（名字定义、名字引用或名字作用域）要同时满足过滤器链上的所有过滤条件才会被接受。例如，我们可以创建如下的名字表过滤器实例：

```
new NameDefinitionNameFilter(  
    new NameDefinitionKindFilter(NameDefinitionKind.NDK_METHOD), "resolve")
```

表示接受名字等于"resolve"的方法定义。

由于对名字定义的遍历最为常用，因此我们为名字定义的过滤实现了多个过滤器类，这些类都是类NameTableFilter的直接子类：

1. `ClassNameDefinitionNameFilter`: 基于名字定义的简单名或全限定名进行过滤;
 2. `ClassNameDefinitionUniqueIdFilter`: 基于名字定义的唯一标识进行过滤;

3. 类NameDefinitionLocationFilter: 基于名字定义的源代码位置进行过滤;
4. 类NameDefinitionScopeFilter: 基于名字定义所属的作用域进行过滤;
5. 类NameDefinitionKindFilter: 基于名字定义的类别进行过滤。

这些过滤器类的实现非常类似:

1. 都有字段wrappedFilter, 它是所包装的过滤器, 是类NameTableFilter(或其后代类) 的实例, 其余的字段则存放用于过滤的具体条件, 例如对于类NameDefinitionNameFilter则存放一个字符串, 用于匹配简单名或全限定名, 为区别是匹配简单名还是全限定名, 还设置一个布尔类型字段useFullQualifiedName来区分这两个情况;
2. 每个过滤器类都要重定义NameTableFilter的accept(NameDefinition)方法, 根据所设置的具体条件对名字定义进行过滤;
3. 过滤器类的其他方法主要用于设置用于过滤的具体条件, 因此都提供灵活的构造方法和一些setter方法设置条件。

由于这些过滤器类的实现都非常类似, 我们以其中最为复杂的类NameDefinitionLocationFilter为例, 对它的成员做更为详细的说明。图5.24给出它的成员, 除字段wrappedFilter 存放所包装的过滤器外, 它还有两个字段start和end分别存放作为过滤条件的源代码起始位置和终止位置。它的accept()方法先调用wrappedFilter(如果它不为null) 的accept()方法, 当返回true 再判断给定的名字定义的源代码位置是否大于等于start而且小于end, 若是才返回true (表示接受该名字定义), 其他情况都返回false。字段start和end都可以为null, 这时表示不考虑相应的条件。

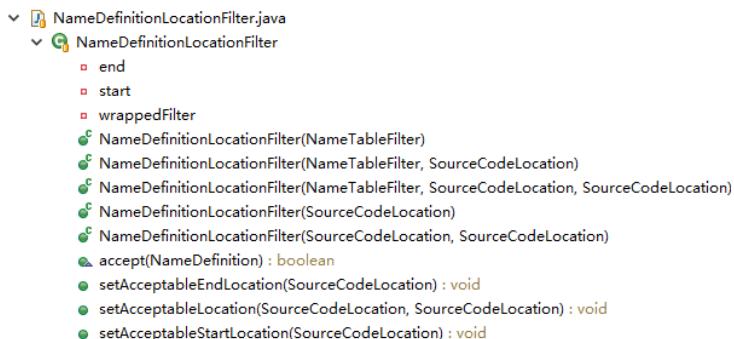


图 5.24 类NameDefinitionLocationFilter的成员

类NameDefinitionLocationFilter提供多个构造方法设置字段wrappedFilter、start和end, 至少设置其中一项条件(因为如果所有条件都不设置则只要直接使用NameTableFilter的实例即可)。构造方法中这些参数的顺序设置原则是: 如果要设置所包装的过滤器, 则它总是第一个参数; 如果只设置一个源代码位置参数, 则代表设置起始位置这个条件。类NameDefinitionLocationFilter还提供一些set...()方法, 可以在创建实例之后对过滤条件进行补充设置。

类NameDefinitionNameFilter针对名字定义的简单名或全限定名进行过滤, 除字段wrappedFilter之外, 它还有字符串类型字段acceptableName指定名字条件和布尔类型字段useFullQualifiedName指明是否使用全限定名。当指定的名字不为空时, 方法accept()将字段acceptableName与名字定义的简单名(当字段useFullQualifiedName 为false时)或全限定名进行精确匹配(使用String类的equals()方法比较), 如果相等则返回true。

类NameDefinitionUniqueIdFilter针对名字定义的唯一标识进行过滤, 它有字符串类型字

段`acceptableUniqueId` 指定唯一标识条件。当指定的唯一标识不为空时，方法`accept()`将字段`acceptableUniqueId`与名字定义的唯一标识进行精确匹配（使用`String`类的`equals()`方法比较），如果相等则返回`true`。

类NameDefinitionScopeFilter针对名字定义所属作用域进行过滤,它有字段acceptableScope指定作用域条件,当该字段不为null时,方法accept()将它与名字定义所属作用域(使用类NameDefinition的getScope()方法得到)进行相等比较(使用运算符==),如果相等则返回true。

类NameDefinitionKindFilter针对名字定义的类别进行过滤，它有字段acceptableKind指定类别条件，当该字段不为null时，方法accept()将它与名字定义的列表（使用类NameDefinition的getDefinitionKind()方法得到）进行相等比较（使用运算符==），如果相等则返回true。

特别地,由于详细类型定义被经常遍历与访问,我们实现了类DetailedTypeDefinitionFilter,它除含有字段wrappedFilter表示所包装的过滤器外,只是重定义方法accept()在调用wrappedFilter的accept()的基础判断给定的名字定义是否是详细类型字段(基于类NameDefinition的isDetailedType()方法判断)。对于枚举类型,我们也实现了类EnumTypeFilter,用于接受枚举类型。

对于名字引用，目前实现了如下过滤器类，这些类都是类NameTableFilter的直接子类：

1. 类NameReferenceNameFilter: 基于名字引用的名字进行过滤;
 2. 类NameReferenceLocationFilter: 基于名字引用的源代码位置进行过滤;
 3. 类NameReferenceScopeFilter: 基于名字引用所属的作用域进行过滤;
 4. 类NameReferenceKindFilter: 基于名字引用的类别进行过滤。

这些类的实现与相应的名字定义过滤器类的实现基本相同（字段的设置和方法accept()的实现策略），只是对于名字引用的过滤是重定义类NameTableFilter的accept(NameReference)方法（名字定义过滤器类是重定义其中的accept(NameDefinition)方法）。

对于名字作用域，目前实现了如下过滤器类，这些类都是类NameScopeFilter的直接子类：

1. 类NameScopeNameFilter: 基于名字作用域的名字进行过滤;
 2. 类NameScopeLocationFilter: 基于名字作用域的源代码位置进行过滤。

类NameScopeNameFilter的实现基本与类NameDefinitionNameFilter以及类NameReferenceNameFilter相同。不过类NameScopeLocationFilter的实现有一点特别。图5.25给出它的成员。

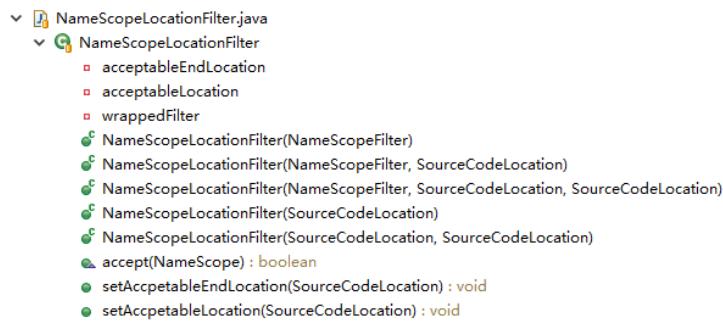


图 5.25 类NameScopeLocationFilter的成员

除字段wrappedFilter存放所要包装的过滤器类外，类NameScopeLocationFilter还有字段acceptableLocation 和acceptableEndLocation用于设置源代码位置条件，但是方法accept()是按如下方式使用这两个字段：

(1) 如果`acceptableEndLocation`不为`null`, 则给定的作用域的起始位置和结束位置要分别等于字段`acceptableLocation`和`acceptableEndLocation`才返回`true`, 除非给定作用域的起始位置或结束位置本身就为`null` (例如系统作用域或程序包定义);

(2) 如果`acceptableEndLocation`为`null`, 则给定作用域包含位置`acceptableLocation`时 (使用接口`NameScope`的`containsLocation()`进行判断) 返回`true`。

最后需要指出的, 构件使用者完全可以根据自己的需要扩展(`extends`)类`NameTableFilter`实现新的名字定义和名字引用过滤器类, 或扩展类`NameScopeFilter`实现新的名字作用域过滤器类。

5.4.3 类`NameTableManager`

类`NameTableManager`是管理名字表的主类, 它将源代码文件集与名字表的入口 (即系统作用域) 关联到一起, 而且名字表构件的使用者通过这个类可实现与名字表使用相关的大部分功能。类`NameTableManager`只有两个字段:

- (1) 字段`codeFileSet`: 它是类`SourceCodeFileSet`的实例, 存放名字表的源代码文件集;
- (2) 字段`systemScope`: 它是类`SystemScope`的实例, 存放名字表的系统作用域, 即名字表的入口, 通过系统作用域可遍历所有的名字定义和名字作用域。

这两个字段通过类`NameTableManager`的构造函数设置。构件使用者通常不使用类`NameTableManager`的构造函数得到类`NameTableManager`的实例, 而是使用名字表生成器的名字表生成方法得到。

表5.44和表5.45给出了类`NameTableManager`的方法。类`NameTableManager`给出一系列`find...()`方法查找满足条件的首个名字定义, 以及一系列`getAll...()`方法获取满足条件的所有名字定义和名字作用域。除获取程序包定义、编译单元作用域的方法外, 这些方法也是通过名字表访问器类实现。这些方法中`getAllDetailedTypeDefinitions()`是直接调用系统作用域的同名方法得到, 注意我们在系统作用域中缓冲了所有详细类型定义构成的列表, 以便在名字解析时可以快速确定子类型关系和方法的多态调用。查找首个和返回所有名字定义或名字作用域都支持使用名字表过滤器 (名字作用域过滤器), 用于可自定义过滤器而获取所需要的首个或多个名字定义或名字作用域。

注意, 返回单个名字定义或作用域的方法可能返回`null` (在不存在满足条件的名字定义或作用域时), 而获取所有名字定义或作用域列表的方法即使在不存在满足条件的名字定义或作用域时也会返回大小为0的空列表, 而不会返回`null`。

类`NameTableManager`提供一系列`getEnclosing...()`方法获取包含某个名字定义或名字引用的最内层的编译单元、类型定义或方法定义, 这些方法都可能返回`null`。这些方法的实现都是从名字定义或名字引用所属的作用域 (这是包含该名字定义或名字引用的最内层作用域), 追溯它的父作用域, 直到找到第一个编译单元作用域、详细类型定义或方法定义, 作为包含该名字定义或名字引用的最内层编译单元、类型定义或方法定义。

方法`getCorrespondingFilePath(NameDefinition)`返回给定名字定义所在的源代码文件的全路径名, 这是通过查找包含该名字定义的编译单元作用域, 然后基于编译单元作用域的编译单元名在源代码文件集中查找源代码文件, 并获得它的全路径名。实际上, 对于任意的名字引用也可按这种方法获得包含该名字引用的源代码文件信息 (不过类`NameTableManager`暂时没有提供这种方法)。

方法`getScopeOfLocation()`返回包含给定源代码位置所在的最内层作用域, 它的实现是从系统作用域开始, 判断是否有子作用域还是包含该源代码位置, 直到一个作用域包含该源代码位置,

表 5.44 类NameTableManager方法的描述

<code>NameTableManager(SourceCodeFileSet, SystemScope) : Constructor</code>	构造方法，以源代码文件集和系统作用域为参数
<code>accept(NameTableVisitor) : void</code>	接受一个名字表访问器进行访问
<code>findCompilationUnitScopeByName(String) : CompilationUnitScope</code>	通过单元名查找相应的编译单元作用域
<code>findDefinitionByFilter(NameTableFilter) : NameDefinition</code>	使用名字表过滤器查找满足条件的(首个)名字定义
<code>findDefinitionById(String) : NameDefinition</code>	查找唯一标识是给定字符串的名字定义
<code>findDefinitionOfQualifiedName(String) : NameDefinition</code>	查找全限定名是给定字符串的名字定义
<code>findDefinitionOfSimpleName(String) : NameDefinition</code>	查找简单名是给定字符串的(首个)名字定义
<code>findPackageByName(String) : PackageDefinition</code>	查找全限定名是给定字符串的程序包定义
<code>getAllCompilationUnitScopes() : List<CompilationUnitScope></code>	返回名字表中所有的编译单元作用域构成的列表
<code>getAllDefinitionsBetweenLocations(SourceCodeLocation, SourceCodeLocation) : List<NameDefinition></code>	返回给定两个源代码位置(大于等于前者而小于后者)之间的所有名字定义
<code>getAllDefinitionsByFilter(NameTableFilter) : List<NameDefinition></code>	返回名字表过滤器所能接受(满足过滤条件)的所有名字定义构成的列表
<code>getAllDefinitionsOfScope(NameScope) : List<NameDefinition></code>	返回给定作用域(包括其子作用域)中的所有名字定义构成的列表
<code>getAllDefinitionsOfSimpleName(String) : List<NameDefinition></code>	返回简单名是给定字符串的所有名字定义
<code>getAllDetailedTypeDefinitions() : List<DetailedTypeDefinition></code>	返回所有的详细类型定义(包括局部类型和匿名类型)
<code>getAllDetailedTypeDefinitions(NameScope) : List<DetailedTypeDefinition></code>	返回给定作用域(包括其子作用域)中的所有详细类型定义(包括局部类型和匿名类型)
<code>getAllPackageDefinitions() : List<PackageDefinition></code>	返回名字中所有程序包定义
<code>getAllScopesByFilter(NameScopeFilter) : List<NameScope></code>	返回名字作用域过滤器所能接受的所有名字作用域构成的列表
<code>getAllScopesOfName(String) : List<NameScope></code>	返回名字是给定字符串的所有名字作用域构成的列表
<code>getCorrespondingFilePath(NameDefinition) : String</code>	返回给定名字定义所在的源代码文件的路径全名(不是单元名)
<code>getEnclosingCompilationUnitScope(NameDefinition) : CompilationUnitScope</code>	返回包含给定名字定义的编译单元作用域
<code>getEnclosingCompilationUnitScope(NameReference) : CompilationUnitScope</code>	返回包含给定名字引用的编译单元作用域
<code>getEnclosingDetailedTypeDefinition(NameDefinition) : DetailedTypeDefinition</code>	返回包含给定名字定义的最内层的详细类型定义
<code>getEnclosingDetailedTypeDefinition(NameReference) : DetailedTypeDefinition</code>	返回包含给定名字引用的最内存的详细类型定义
<code>getEnclosingMethodDefinition(NameDefinition) : MethodDefinition</code>	返回包含给定名字定义的最内层的方法定义
<code>getEnclosingMethodDefinition(NameReference) : MethodDefinition</code>	返回包含给定名字引用的最内层的方法定义

表 5.45 类NameTableManager方法的描述（续）

<code>getScopeOfLocation(SourceCodeLocation) : NameScope</code>	返回给定源代码位置所在的最内层作用域
<code>getScopeOfStartAndEndLocation(SourceCodeLocation, SourceCodeLocation, NameScope) : NameScope</code>	返回指定作用域中起始位置和结束位置是给定位置的子作用域
<code>getScopeOfStartAndEndLocation(SourceCodeLocation, SourceCodeLocation) : NameScope</code>	返回起始位置和结束位置是给定位置的作用域
<code>getSourceCodeFileSet() : SourceCodeFileSet</code>	返回名字表所对应的源代码文件集
<code>getSystemPath() : String</code>	返回名字表所对应的源代码文件集的起始路径
<code>getSystemScope() : SystemScope</code>	返回名字表的系统作用域
<code>getTotalNumberOfDefinitions(NameDefinitionKind) : int</code>	返回给定类别的名字定义总数
<code>getTotalNumberOfDefinitions(NameTableFilter) : int</code>	返回满足给定过滤器给定条件的名字定义总数

而且没有子作用域还包含该源代码位置，则是包含该源代码位置的最内层作用域。

有NameScope类型参数的方法`getScopeOfStartAndEndLocation()`以所给定的作用域为起始作用域，查找它是否有一个子作用域（或者子作用域的子作用域）的起始位置和结束位置等于方法参数中给定的位置，这个方法是通过作用域访问器类`NameScopeFinder`的实例，使用作用域过滤器类`NameScopeLocationFilter`的实例访问起始作用域而实现。没有NameScope类型参数的方法`getScopeOfStartAndEndLocation()`则以系统作用域为起始作用域。

方法`getTotalNumberOfDefinitions()`可计算某种类别或满足某种过滤条件的名字定义的数目，它是通过使用名字定义访问器类`NameDefinitionNumberVisitor`的实例实现。

方法`getSystemScope()`返回名字表的系统作用域，构件使用者可对系统作用域使用名字表访问器（及名字表过滤器）而完成任意的名字表访问功能，但我们建议直接使用类`NameTableManager`的方法`accept()`，接受名字表访问器对名字表进行访问，这个方法内部仍是调用系统作用域的`accept()`方法。

5.4.4 类NameTableASTBridge

类`NameTableASTBridge`提供名字表中的名字表实体与抽象语法树节点之间的关联，它分担类`NameTableManager`的这方面的功能，因此它只有一个字段，保存类`NameTableManager`的实例，基于该实例管理名字表实体与抽象语法树节点之间的关联关系。

表5.46给出类`NameTableASTBridge`的方法，它的构造方法以类`NameTableManager`的实例为参数，这意味着构件使用者只有在获得类`NameTableManager`的实例之后才能构造类`NameTableASTBridge`的实例，完成名字表实例与抽象语法树节点之间的关联关系的管理。

类`NameTableASTBridge`提供一系列`findASTNode...()`方法查找编译单元作用域、详细类型定义和方法定义所对应抽象语法树AST节点。

实际上，通过编译单元作用域查找编译单元抽象语法树AST节点比较容易：编译单元作用域的

表 5.46 类NameTableASTBridge方法的描述

<code>NameTableASTBridge(NameTableManager) : Constructor</code>	构造方法，以类NameTableManager的实例为参数
<code>findASTNodeForCompilationUnitScope(CompilationUnitScope) : CompilationUnit</code>	查找给定编译单元作用域所对应的AST节点（类型是CompilationUnit）
<code>findASTNodeForCompilationUnitScope(String) : CompilationUnit</code>	查找给定编译单元名所对应的AST节点（类型是CompilationUnit）
<code>findASTNodeForDetailedTypeDefinition(DetailedTypeDefinition) : TypeDeclaration</code>	查找给定详细类型定义所对应的AST节点（类型是TypeDeclaration）
<code>findASTNodeForFieldDefinition(FieldDefinition) : FieldDeclaration</code>	查找给定字段定义所对应的AST节点（类型是FieldDeclaration）
<code>findASTNodeForMethodDefinition(MethodDefinition) : MethodDeclaration</code>	查找给定方法定义所对应的AST节点（类型是MethodDeclaration）
<code>findDefinitionForAnonymousClassDeclaration(String, AnonymousClassDeclaration) : AnonymousClassDefinition</code>	查找在由编译单元名指定的编译单元中的匿名类声明AST节点所对应的匿名类定义
<code>findDefinitionForAnonymousClassDeclaration(LocalScope, SourceCodeLocation, AnonymousClassDeclaration) : AnonymousClassDefinition</code>	查找给定局部作用域、源代码位置的匿名类声明AST节点所对应匿名类定义
<code>findDefinitionForEnumDeclaration(String, EnumDeclaration) : EnumTypeDefinition</code>	查找在由编译单元名指定的编译单元中的枚举类型声明AST节点所对应的枚举类型定义
<code>findDefinitionForEnumDeclaration(CompilationUnitScope, SourceCodeLocation, EnumDeclaration) : EnumTypeDefinition</code>	查找在给定编译单元作用域、源代码位置的枚举类型声明AST节点所对应的枚举类型定义
<code>findDefinitionForMethodDeclaration(String, MethodDeclaration) : MethodDefinition</code>	查找在由编译单元名指定的编译单元中的方法声明AST节点所对应的方法定义
<code>findDefinitionForMethodDeclaration(DetailedTypeDefinition, SourceCodeLocation, MethodDeclaration) : MethodDefinition</code>	查找在给定详细类型定义、源代码位置的方法声明AST节点所对应的方法定义
<code>findDefinitionForTypeDeclaration(String, TypeDeclaration) : DetailedTypeDefinition</code>	查找在由编译单元名指定的编译单元中的类型声明AST节点所对应的详细类型定义
<code>findDefinitionForTypeDeclaration(CompilationUnitScope, SourceCodeLocation, TypeDeclaration) : DetailedTypeDefinition</code>	查找在给定编译单元作用域、源代码位置的类型声明AST节点所对应的详细类型定义
<code>findDefinitionsForFieldDeclaration(String, FieldDeclaration) : List<FieldDefinition></code>	查找在由编译单元名指定的编译单元中的字段声明AST节点所对应的字段定义构成的列表
<code>findDefinitionsForFieldDeclaration(DetailedTypeDefinition, SourceCodeLocation, SourceCodeLocation, FieldDeclaration) : List<FieldDefinition></code>	查找在给定详细类型定义、源代码位置的字段声明AST节点所对应的字段定义构成的列表
<code>getAllDefinitionsInASTNode(String, ASTNode) : List<NameDefinition></code>	返回在某个AST节点中的所有名字定义构成的列表

作用域名字就是编译单元的单元名，根据单元名在源代码文件集中可找到对应的源代码文件，即类`SourceCodeFile`的实例，其中有该源代码文件的抽象语法树根节点，即编译单元作用域所对应的抽象语法树节点。

查找详细类型定义相对应的AST节点则要先找到该详细类型定义所属的编译单元作用域，然后找到该编译单元的AST节点，这是该编译单元的抽象语法树根节点，然后从该根节点开始，我们使用类`TypeDeclarationASTVisitor`的实例对抽象语法树进行遍历，获取在该编译单元中声明的所有类型声明节点（含局部类型声明），然后通过比较简单名和源代码位置找到相应的类型声明AST节点。

查找方法定义相对应的AST节点，也需要先找到该方法定义所属的编译单元作用域，得到抽象语法树根节点，并且找到该方法定义所属的详细类型定义，根据该详细类型定义找到相应的类型声明AST节点，然后从类型声明节点中得到它的方法声明列表，并通过比较方法声明的简单名和源代码位置得到方法定义相应的方法声明AST节点（抽象语法树根节点需要用于计算方法声明AST节点的源代码位置）。

类`NameTableASTBridge`提供一系列`findDefinition...`方法为AST节点（包括匿名类声明、枚举类型声明、类型声明、方法声明和字段声明AST节点）找到对应的名字定义。每种节点有两个方法，一种基于AST节点所在的编译单元的单元名查找，一种基于可能包含该AST节点的上一层名字定义，以及AST节点的源代码位置查找。前一方法需要花费更多时间遍历名字表查找相应的名字定义，而后者则可基于更小的范围内查找。这些方法的实现都是基于AST节点的源代码位置和名字定义的简单名在合适的范围查找相应的名字表实体。

5.5 名字表生成类

与生成名字表相关的类都在程序包`nameTable.creator`中，图5.26给出这些类的列表。实际上，在这些类中，类`TypeDeclarationASTVisitor`没有用于名字表的生成，它是辅助类`NameTableASTBridge`获取某一编译单元中所有的类型声明，只是因为它也是Eclipse JDT抽象语法树遍历类`ASTVisitor`的子类，所以被放在这个程序包中。

注意，这一节所谈到的抽象语法树节点类或访问器类都在程序包`org.eclipse.jdt.core.dom`，由Eclipse JDT提供。

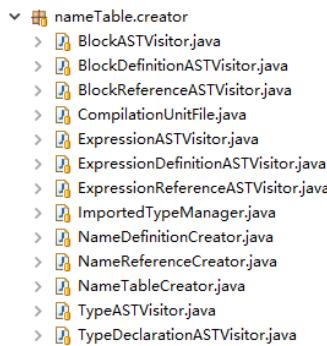


图 5.26 与生成名字表相关的类

根据我们的设计，生成名字表的方式有两种：

1. 同时生成名字定义、名字作用域和名字引用，生成的名字引用存放在名字作用域的字段referenceList中，这种方法只适用于比较小型（名字引用不太多）的程序；
2. 只生成名字定义和名字作用域，名字引用由于比较多，因此通常是临时针对某个范围单独生成，生成的名字引用不存放在名字作用域中（从而使用名字引用访问器类NameReferenceVisitor不能访问到）。

类NameTableCreator完成名字表生成的第一种方式，同时生成名字定义、作用域和名字引用，而类NameDefinitionCreator则只生成名字定义和名字作用域，构件使用者需要在生成名字定义和名字作用域获得类NameTableManager实例的基础上再使用类NameReferenceCreator生成指定作用域范围内的名字引用。

名字表的生成都是对源代码文件集的每个源代码文件，即每个编译单元的抽象语法树进行扫描，找到其中的类型定义、字段定义和方法定义等，然后扫描方法体中的语句找到变量定义、局部类型定义、匿名类定义，以及各种名字引用，并创建合适的作用域存放存放这些名字定义和名字引用。

实际上，我们是在类NameTableCreator或类NameDefinitionCreator中扫描编译单元，找到类型定义、字段定义和方法定义，但是对方法体的扫描则由Block...ASTVisitor这种类完成从方法体开始的抽象语法树节点的遍历。命名为Block...ASTVisitor的类主要处理方法体的语句，在碰到表达式节点时则使用Expression...ASTVisitor进行对抽象语法树节点做进一步的遍历。因为名字引用多数出现在表达式节点，而且不同的表达式可能需要创建不同的名字引用组，我们将表达式节点的遍历分离出来，以免Block...ASTVisitor的实现过于复杂。在扫描和遍历抽象语法树节点的过程中，经常会碰到各种类型引用，我们使用类TypeASTVisitor完成类型引用节点的遍历以创建合适的类型引用实例。

简单地说：

- (1) 类NameTableCreator使用类BlockASTVisitor和类ExpressionASTVisitor完成对编译单元的扫描，并且同时生成名字定义、作用域和名字引用。
- (2) 类NameDefinitionCreator只生成名字定义、作用域和必要的名字引用（主要是类型引用），因此使用类ExpressionDefinitionASTVisitor和类BlockDefinitionASTVisitor完成对编译单元的扫描。类NameDefinitionCreator需要扫描表达式的类ExpressionDefinitionASTVisitor，因为少数名字定义被包装在表达式语句中（例如for语句初始化表达式列表通常包含变量定义）。
- (3) 类NameReferenceCreator只生成名字引用，使用类ExpressionReferenceASTVisitor和类BlockReferenceASTVisitor完成对编译单元的扫描。

由于类型引用在生成名字定义、作用域和名字引用时都可能使用到，类TypeASTVisitor都会被这些类使用到。在生成名字表时，这些类都要用到编译单元名及其抽象语法树根节点的信息（例如用于创建源代码位置信息），我们将这两项信息，连同可能的错误信息封装在数据类CompilationUnitFile中。

下面先给出类CompilationUnitFile的实现，然后是TypeASTVisitor的实现，类Block...ASTVisitor的实现，以及类Expression...ASTVisitor的实现，最后再给出类Name...Creator的实现，以及类ImportedTypeManager的实现。类ImportedTypeManager扫描为导入类型编写的额外信息文件，以获取有关导入类型的更多信息，它与类NameDefinitionCreator的实现有很多类似的地方。

5.5.1 类CompilationUnitFile

在生成名字表时经常会同时使用编译单元的单元名及其抽象语法根节点，因为单元名是源代码位置的一部分，而计算抽象语法树节点在源文件的行号与列号需要抽象语法树根节点。因此为了简化参数的传递，我们将编译单元名及其抽象语法树根节点封装在数据类`CompilationUnitFile`。

图5.27给出类`CompilationUnitFile`的成员，字段`root`存放编译单元的抽象语法树根节点（类型是`CompilationUnit`），字段`unitName`存放编译单元的单元名。我们还设置了字段`errorMessage`存放该编译单元在创建名字表时可能发生的错误信息。

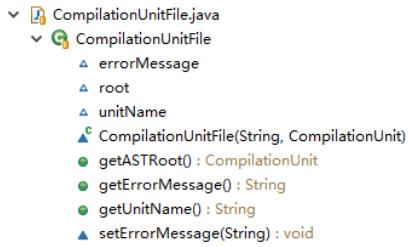


图 5.27 类`CompilationUnitFile`的成员

类`CompilationUnitFile`的构造方法以单元名和根节点为参数，一旦创建，它们不会再被改变，因此字段`root`和`unitName`只有getter方法。字段`errorMessage`可以被改变，因此有getter和setter方法。

5.5.2 类TypeASTVisitor

类`TypeASTVisitor`是Eclipse JDT的抽象语法树访问器类`ASTVisitor`的直接子类，它的实例用于访问抽象语法树中表示引用类型的节点，这些节点都是抽象语法树节点类`Type`的后代类。

图5.28给出类`TypeASTVisitor`的成员。字段`currentScope`的类型是`NameScope`，存放正遍历节点所在的作用域，也是遍历结果得到的类型引用所属作用域。字段`unitFile`是类`CompilationUnitFile`的实例，存放编译单元文件信息，即单元名和抽象语法树根节点。类`TypeASTVisitor`的构造函数以编译单元信息和所在作用域为参数，在构造实例时设置这两个字段的信息。

整数类型字段`dimension`是数组类型的维数，当为0时不是数组类型声明，大于0时为数组类型，数组维数信息可能要整个节点遍历完毕才能确定，所以需要存起来。同样的原因，字符型字段`name`存放类型引用中的名字，`resultLocation`存放类型引用的源代码位置。有些代表类型引用的节点需要先访问其子节点得到部分信息，而在访问完整个节点才能创建最后的结果类型引用，所以我们将这些信息先暂存起来，在方法`getResult()`被调用时才创建结果类型引用（调用这个方法表明肯定已经访问完整个节点）。字段`result`代表最后的访问结果，即用于返回的类型引用，它可能在方法`getResult()`中才创建。

类`TypeASTVisitor`的`reset()`方法支持重新设置作用域，甚至重新设置编译单元文件信息以使用相同的实例遍历其他代表类型引用的节点，同时这个方法也将代表结果的字段`result`重置为`null`。实际上，在一些`visit()`方法中，也是使用类`TypeASTVisitor`的当前实例（在`reset()`后）访问当前节点的子节点。也就是说，类`TypeASTVisitor`的实例是可以重用的，它对两个不同节点的两次访问之间是不会互相影响的。

```

    TypeASTVisitor.java
    + TypeASTVisitor
        - currentScope
        - dimension
        - name
        - result
        - resultLocation
        - unitFile
    + TypeASTVisitor(CompilationUnitFile, NameScope)
    + getResult() : TypeReference
    + reset(CompilationUnitFile, NameScope) : void
    + reset(NameScope) : void
    + visit(ArrayType) : boolean
    + visit(IntersectionType) : boolean
    + visit(NameQualifiedType) : boolean
    + visit(ParameterizedType) : boolean
    + visit(PrimitiveType) : boolean
    + visit(QualifiedName) : boolean
    + visit(QualifiedType) : boolean
    + visit(SimpleName) : boolean
    + visit(SimpleType) : boolean
    + visit(UnionType) : boolean
    + visit(WildcardType) : boolean

```

成员	描述
currentScope	正遍历节点所在的作用域，也是遍历结果应该属于的作用域
dimension	数组类型的维数，大于0时才代表真正是数组类型
name	类型引用的名字
result	遍历的最后结果，也即整个节点所应创建的类型引用
resultLocation	结果类型引用的源代码位置
unitFile	编译单元文件信息，类CompilationUnitFile的实例
getResult()	返回遍历的最后结果
reset(...)	重置访问器以遍历新的节点，result字段将被重置为null
visit(...)	访问代表类型引用的AST节点

图 5.28 类TypeASTVisitor的成员

类TypeASTVisitor的每个visit()方法处理一种代表引用类型的AST节点，实际上除少数节点（ArrayType、PrimitiveType、SimpleType和SimpleName）外，其他节点我们都设计了相应的类型引用类（即类TypeReference的子类），因此相应的visit()也就是创建相应的类型引用类的实例。而ArrayType节点主要用于确定数组维数，PrimitiveType和SimpleName节点代表简单的类型引用，直接创建类TypeReference的实例。SimpleType节点实际上有可能是带限定名的类型，因此需要进一步访问其子节点才能确定创建何种类型引用。

5.5.3 类BlockASTVisitor

类BlockASTVisitor是Eclispe JDT的AST访问器类ASTVisitor的直接子类，它的实例遍历以方法体的块语句（AST 节点类型是Block）为根节点的抽象语法（子）树，提取其中的名字定义和名字引用，并在合适的时候生成局部作用域。注意，方法体中只可能存在局部作用域，每个局部作用域都对应一个花括号括起来的块语句，但我们不会为每个块语句都创建局部作用域，只有当块语句有变量声明，或说有变量的作用域是该块语句时才会创建局部作用域。

图5.29给出类BlockASTVisitor的部分成员，之所以是部分因为这个类有92个visit()方法，重定义类ASTVisitor的每个访问（不同）抽象语法树节点的visit()方法，这里没有列出所有的visit()方法。

类BlockASTVisitor有类型为NameTableCreator的字段creator，是使用类BlockASTVisitor进行名字表生成的生成器实例，之所以需要生成器实例，是因为在方法体中可能存在局部类和匿名类，需要调用生成器的方法扫描局部类和匿名类。类BlockASTVisitor的字段unitFile存放正在扫描的编译单元信息，是类CompilationUnitFile 的实例。

类BlockASTVisitor有类型为ExpressionASTVisitor的字段expressionVisitor，及类型为ASTVisitor的字段typeVisitor，是因为在扫描方法体时需要处理表达式和类型引用。这两个字段分别是表达式AST节点访问器和类型引用AST节点访问器，分别访问类型为AST节

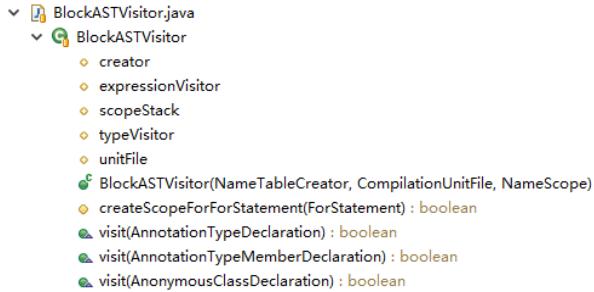


图 5.29 类BlockASTVisitor的部分成员

点类Expression和Type的子类的AST节点。这两个访问器是可以重入的，即可不断调用它们的reset()方法重置访问器而访问不同的表达式节点或类型引用节点，无需每次在需要时都创建不同的访问器实例。注意到Java程序中类型引用和表达式可能会非常多，这也许可减少内存的分配和回收次数（之所以说也许，因为对这一点我们还没有做过严谨的实证分析）。

类BlockASTVisitor有类型为Stack的字段scopeStack。类是在程序包util中的一个简单的实现堆栈数据结构的类（Java语言本身提供的堆栈类有考虑同步等复杂因素，为简单起见我们做了自己的实现）。由于创建的局部作用域可能是嵌套的，即一个局部作用域被包含在另外一个局部作用域中，而我们需要获取直接包含变量声明（乃至局部类、匿名类）的最内层局部作用域，因此使用栈scopeStack存放这些嵌套的局部作用域，使得最内层局部作用域总是在栈顶。

由于在类BlockASTVisitor维护创建的局部作用域栈，因此类BlockASTVisitor是不可重入的，也即为安全起见我们在扫描不同方法的方法体时是创建不同的类BlockASTVisitor实例完成，而不是重置已经创建的实例。这是因为，在扫描一个类的方法体过程中可能扫描局部类的方法的方法体，显然扫描局部类的方法体时不能重用包含这个局部类的类的方法体的BlockASTVisitor实例，因为后者的局部作用域栈不能被破坏。

类BlockASTVisitor的构造方法以名字表生成器实例、编译单元信息和方法体对应的作用域为参数，它将名字表生成实例和编译单元信息分别存放在字段creator和unitFile中，而将作用域压入到scopeStack的栈顶。

类BlockASTVisitor的visit()方法实际访问抽象语法树节点，它重定义了目前类ASTVisitor的所有visit()方法，以保证能正确处理每个抽象语法树节点（虽然有些节点不可能出现在以Block为根的子节点中）。

按照所访问的抽象语法树节点，类BlockASTVisitor的visitor()方法大致可分为如下几种：

1. 访问表达式节点（类Expression的子类）的visit()方法基本上利用类ExpressionASTVisitor的实例访问表达式节点，使用这个类的getResult()方法获得访问结果（一个名字引用），将该名字引用加入到局部作用域栈顶作用域；
2. 访问可能需要创建局部作用域的抽象语法树节点的visit()方法。可能需要创建局部作用域的抽象语法树节点类型有Block、CatchClause、EnhancedForStatement、ForStatement类型节点：

(1). Block代表块语句（方法体中还有许多块语句），我们调用生成器的needCreateLocalScope()方法判断该块语句是否要创建局部作用域。方法needCreateLocalScope()实际上是检查Block中的语句列表中是否有变量声明语句或类型声明语句，如果有则需要创建，否则不需要。

(2). `CatchClause`是异常处理语句的捕获语句块，它要声明所捕获的异常类型，这个声明的作用域是该语句块，所以需要创建局部作用域；

(3). `EnhancedForStatement`是增强型`for`循环语句，它其中有循环空值变量声明，作用域是循环体，所以对应循环体需要创建局部作用域；目前为了简化，当循环体不是块语句时我们实际上没有创建局部作用域，而是将这个声明放在它上一层作用域（这样是否合适？如果上层声明了相同名字的变量时名字引用解析是否准确？还有待进一步测试）；

(4). `ForStatement`是`for`循环语句，因为它的初始化表达式可能含有变量声明，而且该声明的作用域是循环体，所以需要创建局部作用域。我们调用私有方法`createScopeForForStatement()`检查初始化表达式是否有变量声明，以及循环体中有变量声明，如果有则创建局部作用域。

注意，对于其他循环语句或分支语句，其对应的`visit()`方法都直接返回`true`，即访问它的子节点，如果它的循环体是块语句，则会在访问`Block`类型的节点时决定是否创建局部作用域，上面(2)-(4)都是变量声明语句在块语句外，因此需要特别处理。

3. 访问可能含有变量声明的抽象语法树节点的`visit()`方法。这些节点有`SingleVariableDeclaration`、`VariableDeclarationExpression`、`VariableDeclarationFragment`、`VariableDeclarationStatement`等，这些节点对应的`visit()`方法会在栈顶作用域定义变量（即添加变量定义），这都通过调用生成器的方法`defineVariable()`完成。

4. 访问含有类型声明的抽象语法节点`TypeDeclarationStatement`的`visit()`方法，调用生成器（字段`creator`）的合适的`scan()`方法扫描其中的类型声明或枚举声明。

5. 其他`visit()`方法要么是返回`false`忽略该节点及其子节点，要么是返回`true`表示继续访问其子节点。

由于类`BlockASTVisitor`的`visit()`方法非常多，我们不对每个方法的实现一一进行说明。

5.5.4 类`BlockDefinitionASTVisitor`

类`BlockDefinitionASTVisitor`是类`BlockASTVisitor`的直接子类，它完成的职责与类`BlockASTVisitor`类似，但是只向局部作用域栈顶作用域添加名字定义（及名字定义中需要用到的名字引用，例如变量定义中的类型引用），而不添加（独立的）名字引用。

图5.30给出类`BlockDefinitionASTVisitor`的部分成员（同样没有给出全部的`visit()`方法）。可以看到，类`BlockDefinitionASTVisitor`没有额外定义字段，全部是继承自类`BlockASTVisitor`的字段，构造方法的参数也与类`BlockASTVisitor`的构造方法参数完全相同。



图 5.30 类`BlockDefinitionASTVisitor`的部分成员

类`BlockDefinitionASTVisitor`除构造方法及重定义`visit()`方法外，没有其他方法。它重定义了38个`visit()`方法，这些方法主要是：

1. 访问表达式节点（类`Expression`的子类）的`visit()`方法不再需要遍历以获取其中的名字引用，因此返回`true`（有些节点其中可能含有变量声明子节点，所以要进一步访问子节点）或`false`（有些节点，例如类型引用节点，肯定不会含有任何名字定义，所以无需进一步访问子节点）。但其中处

理节点类型为ClassInstanceCreation的visit()方法比较特殊，它调用生成器的scan()方法完成对匿名类的扫描。对于类BlockASTVisitor而言，这项功能在由类ExpressionASTVisitor完成。

2. 访问可能需要创建局部作用域的抽象语法树节点的visit()方法。这些方法的实现与类BlockASTVisitor的同参数方法基本相同，主要是在需要时创建局部作用域。与类BlockASTVisitor同参数方法不同的是删除了处理名字引用相关的语句（例如处理变量声明的初始化表达式中名字引用的语句）；

3. 访问可能含有变量声明的抽象语法树节点的visit()方法。这些方法的实现也与类BlockASTVisitor的同参数方法基本相同，但删除了处理名字引用相关的语句，特别是处理变量声明的初始化表达中名字引用的语句；

其他类型的AST节点则使用类BlockASTVisitor的visit()方法处理。同样由于这个类的visit()方法也比较多，我们也不在这里一一地详细说明。

5.5.5 类BlockReferenceASTVisitor

类BlockReferenceASTVisitor是类ASTVisitor的直接子类，它的实例遍历以方法体的块语句（AST 节点类型是Block）为根节点的抽象语法（子）树，提取其中的名字引用。注意，类BlockReferenceASTVisitor 不是类BlockASTVisitor的子类，而是直接扩展类ASTVisitor，这是因为它不创建局部作用域，与类BlockASTVisitor的职责在这一点有很大不同。生成名字引用时不能改变已经生成的名字定义与名字作用域。

图5.31给出类BlockReferenceASTVisitor的部分成员。同样这里并没有给出它的所有visit()方法。类BlockReferenceASTVisitor目前有90个visit()，基本上重定义了类ASTVisitor的所有visit()方法，除可能漏掉的两个新增的方法引用表达式节点的visit() 方法外（注意，我们目前实际上还没有正确处理Java 8所引用的方法引用表达式和Lambda表达式，所以暂时没有增加这两个方法，留待以后统一处理方法引用表达式和Lambda表达式时再一起更正这个问题）。

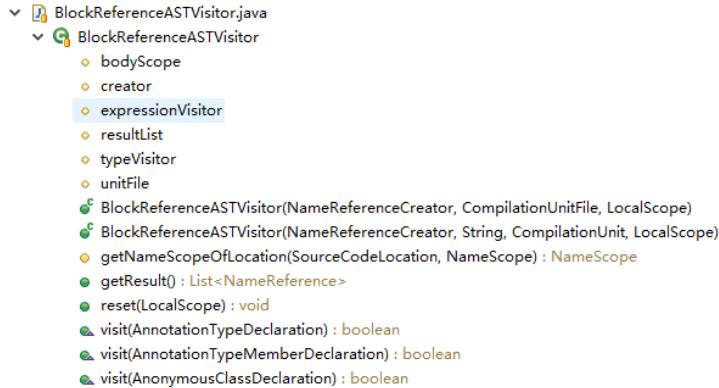


图 5.31 类BlockReferenceASTVisitor的部分成员

类BlockReferenceASTVisitor的字段creator是类NameReferenceCreator的实例，是名字引用生成器，字段unitFile是类CompilationUnitFile的实例，保存正在扫描的编译单元信息，字段bodyScope是所扫描的方法体所对应的局部作用域，所生成的名字引用都在该局部作用域（或其子作用域）中。

类BlockReferenceASTVisitor的构造方法以名字引用生成器实例、编译单元信息和对应方法

体的局部作用域为参数，它将这些信息存放在对应的字段中，并初始化其他字段，其中一个构造方法将编译单元信息包装在类`CompilationUnitFile` 的实例中，另一个构造方法则直接使用单元名及其抽象语法树根节点为参数。

类`BlockReferenceASTVisitor`的字段`expressionVisitor`是类`ExpressionReferenceASTVisitor`的实例，负责访问表达式类型的抽象语法树节点，提取其中的名字引用（组）。字段`typeVisitor`是类`TypeASTVisitor`的实例，负责访问代表类型引用的抽象语法树节点，提取其中的类型引用。

类`BlockReferenceASTVisitor`的字段`resultList`是扫描方法体生成的所有名字引用构成的列表，包括方法体中声明的局部类、匿名类中的名字引用。方法`getResult()`返回该列表。

类`BlockReferenceASTVisitor`在生成名字引用时需要确定包含该名字引用的最内层作用域（因为名字引用解析需要从最内层作用域开始解析），私有方法`getNameScopeOfLocation()`基于所访问的抽象语法树节点，基于编译单元的抽象语法树根节点计算该节点对应的源代码位置，然后从给定的作用域（通常是所扫描的方法体对应的局部作用域，即字段`bodyScope`）开始，在该作用域中找包含该位置的最内层作用域，作为所访问的抽象语法树节点所生成的名字引用所属的（最内层）作用域。

类`BlockReferenceASTVisitor`的其他方法都是`visit()`方法，为方便与类`BlockASTVisitor`对比，我们也分为如下几种类别的`visit()`方法进行讨论：

按照所访问的抽象语法树节点，类`BlockASTVisitor`的`visitor()`方法大致可分为如下几种：

1. 访问表达式节点（类`Expression`的子类）的`visit()`方法利用类`ExpressionReferenceASTVisitor`的实例访问表达式节点，使用这个类的`getResult()`方法获得访问结果（一个名字引用），将该名字引用加入到局部作用域栈顶作用域。后面可以看到，类`ExpressionReferenceASTVisitor`与类`ExpressionASTVisitor`的唯一差别就在于处理变量声明表达式节点方面，前者只关注名字引用，而后者要在当前作用域中添加变量定义；
2. 访问可能需要创建局部作用域的抽象语法树节点的`visit()`方法。可能需要创建局部作用域的抽象语法树节点指`Block`、`CatchClause`、`EnhancedForStatement`、`ForStatement`四个类型的节点。对于类`BlockReferenceASTVisitor`而言，不创建局部作用域，因此这些方法也只是返回`true`，进一步访问其子节点而已。
3. 访问可能含有变量声明的抽象语法树节点的`visit()`方法。这些节点有`SingleVariableDeclaration`、`VariableDeclarationExpression`、`VariableDeclarationFragment`、`VariableDeclarationStatement`等。对于类`BlockReferenceASTVisitor`而言，只是提取中声明变量类型的类型引用，以及声明变量时可能存在的初始化表达式中的名字引用。
4. 访问含有类型声明的抽象语法节点`TypeDeclarationStatement`的`visit()`方法，也是调用生成器（字段`creator`）的合适的`scan()`方法扫描其中的类型声明或枚举声明，以获得其中的名字引用构成的列表，并将其中的元素全部添加到字段`resultList`中。
5. 其他`visit()`方法与类`BlockASTVisitor`相同，要么是返回`false`忽略该节点及其子节点，要么是返回`true`表示继续访问其子节点。

上面概述了实现类`BlockReferenceASTVisitor`的`visit()`方法的基本思想，由于这样的方法太多，这里不再对每个方法的实现做更详细的介绍。

5.5.6 类ExpressionASTVisitor

类ExpressionASTVisitor是类ASTVisitor的直接子类，它的实例被类BlockASTVisitor和类NameTableCreator用于扫描方法体中出现的抽象语法树表达式节点，以生成对应的名字引用（组）。抽象语法树表达式节点都是在Expression的子节点，除表4.10列出的26个类外，Java 8引入了5个名为...Reference的表示方法引用表达式的节点和一个名为LambdaExpression的Lambda表达式节点，共32个类表示抽象语法表达式节点。

图5.32给出类ExpressionASTVisitor的部分成员。这个类有31个visit()方法（对于表达式节点类ParenthesizedExpression我们使用类ASTVisitor的缺省实现，即直接访问其子节点，即括号内的表达式节点即可），图5.32没有全部列出。

类ExpressionASTVisitor的字段creator是名字表生成器，即类NameTableCreator的实例。在处理抽象语法树节点类ClassInstanceCreation时，其中可能有匿名类，需要调用名字生成器的scan()方法对匿名类进行扫描，生成其中的名字引用。

类ExpressionASTVisitor的字段unitFile是类CompilationUnitFile的实例，保存编译单元信息。字段scope的类型是NameScope，给出生成的名字引用应该所属的作用域。

类ExpressionASTVisitor的构造方法以名字生成器、编译单元信息和作用域为参数，将这些信息存放在相应的字段。

类ExpressionASTVisitor的字段typeVisitor是类TypeASTVisitor的实例，用于访问在表达式节点中出现的代表类型引用的节点，生成相应的类型引用。

类ExpressionASTVisitor的字段lastReference是访问表达式节点后最终生成的名字引用（可能是名字引用组）。类ExpressionASTVisitor的实例是可重入的，在访问一个表达式节点时，可能需要访问它的子表达式（子节点），这时我们仍然使用同一实例进行访问，并且也将访问的结果存入字段lastReference，但这个结果会立即保存到合适的名字引用组的子名字引用列表中。总的来说，类ExpressionASTVisitor（包括它的子类）最终总是只生成一个名字引用（但可能是名字引用组）。方法getResult()返回最后生成的名字引用，即字段lastReference。

字段createReferenceForLiteral是一个布尔类型的开关，表示是否为文字常量生成引用。我们只为那些处于表达式中的文字常量生成文字常量引用，而不为单独存在的文字常量（例如return 0中的0）生成文字常量。

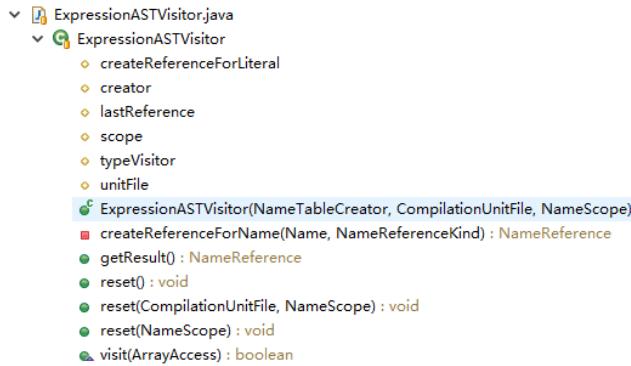


图 5.32 类ExpressionASTVisitor的部分成员

类ExpressionASTVisitor实例的可重入通过方法reset()实现。这个类提供几种参数形

式的`reset()`方法，对访问结果字段(`lastReference`)、作用域字段(`scope`)和编译单元信息字段(`unitFile`)进行重置，其中访问结果字段必须重置，而作用域字段和编译单元信息字段是否重置由使用者(通过调用不同的`reset()`方法)决定。

私有方法`createReferenceForName()`为一个简单名字创建名字引用，它被`visit(QualifiedName)`和`visit(FieldAccess)`方法用于为其中的简单名创建值引用实例，而其名字引用类别分别是`NRK_VARIABLE`和`NRK_FIELD`。

类`ExpressionASTVisitor`有31个`visit()`方法，对应处理每一种抽象语法树表达式节点。这些`visit()`方法可分为以下几类：

1. 访问文字常量的`visit()`方法，这些方法都是在当字段`createReferenceForLiteral`时创建文字常量引用，保存至字段`lastReference`；
2. 对于Java 8新增加的有关方法引用表达式和Lambda表达式的6个节点，目前只是暂时返回`false`；
3. 对于`VariableDeclarationExpression`节点，提取其中的变量声明，生成相应的名字定义，加入到当前作用域，并为其中变量的类型声明和初始化表达式生成类型引用和名字引用加入到名字引用组(类`NRGVariableDeclaration`的实例)；
4. 对于`ClassInstanceCreation`节点，除生成名字引用组(类`NRGClassInstanceCreation`的实例)存放与实例创建相关的名字引用外，如果其中存在匿名类声明，则调用名字表生成器(`creator`)的`scan()`方法扫描匿名类声明。注意，这时在匿名类中的名字引用将存放对应匿名类声明的作用域类，而不是在类`NRGClassInstanceCreation`的实例中；
5. 其他抽象语法树节点基本上与名字引用组的子类是一一对应的，对它们的访问则根据其语法结构创建相应的名字引用组。

由于类`ExpressionASTVisitor`中的`visit()`方法比较多，这里不再对每个方法的实现进行详细的说明。

5.5.7 类`ExpressionDefinitionASTVisitor`

类`ExpressionDefinitionASTVisitor`是类`ExpressionASTVisitor`的直接子类，它的实例被类`BlockDefinitionASTVisitor`用于扫描方法体中出现的抽象语法树表达式节点，以提取可能的名字定义。由于只有在类型为`VariableDeclarationExpression`的节点中可能存在名字定义，因此这个类的实现实际上十分简单：

1. 将除处理类型为`VariableDeclarationExpression`节点外的`visit()`都重定义为直接返回`false`，因为这些节点中没有名字定义，也无需继续访问它的子节点。虽然类型为`ClassInstanceCreation`的节点可能含有匿名类声明，但类`BlockDefinitionASTVisitor`已经对这种节点中的匿名类声明做了处理，而类`ExpressionDefinitionASTVisitor`的实例被类`BlockDefinitionASTVisitor`使用，所以无需再对处理类型为`ClassInstanceCreation`中的匿名类声明。
2. 对于`VariableDeclarationExpression`节点，提取其中的变量声明，生成相应的名字定义，加入到当前作用域，这里会调用名字表生成器(这时应该是类`NameDefinitionCreator`的实例)的`defineVariable()`方法。

图5.33给出类`ExpressionDefinitionASTVisitor`的部分成员(实际上给出了除一些`visit()`之外的所有成员)。可以看到，类`ExpressionDefinitionASTVisitor`没有额外定义新的字段，而且



图 5.33 类ExpressionDefinitionASTVisitor的部分成员

有与类ExpressionASTVisitor相同的构造方法参数，除重定义visit()方法之外，也重定义了方法getResult()，由于不关心名字引用的生成，因此这个方法直接返回null（注意，处理类型为VariableDeclarationExpression时提取的名字定义是直接添加到作用域中，而不作为访问的结果返回）。

5.5.8 类ExpressionReferenceASTVisitor

类ExpressionReferenceASTVisitor是类ExpressionASTVisitor的直接子类，它的实例被类BlockReferenceASTVisitor和类NameReferenceCreator用于扫描方法体中出现的抽象语法树表达按时节点，以生成对应的名字引用组。由于表达式节点中基本上是生成名字引用，因此类ExpressionReferenceASTVisitor基本上是重用类ExpressionASTVisitor的功能，实际上，它只重定义了处理类型为VariableDeclarationExpression节点和类型为ClassInstanceCreation节点的visit()方法，去掉了这些方法在类ExpressionASTVisitor中实现的提取名字定义的语句，而只保留了生成名字引用的功能。

图5.34给出类ExpressionReferenceASTVisitor的（所有）成员，可以看到它只重定义了两个visit()方法，其他visit()方法均直接使用类ExpressionASTVisitor 的实现。

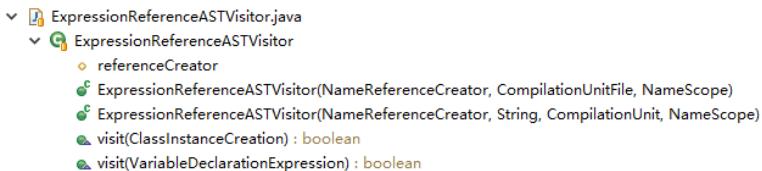


图 5.34 类ExpressionReferenceASTVisitor的成员

类ExpressionReferenceASTVisitor重新使用字段referenceCreator存放类型为NameReferenceCreator的名字引用生成器，而不是继承自类ExpressionASTVisitor的字段creator，因为类NameReferenceCreator不是类NameTableCreator的子类。处理类型为ClassInstanceCreation的visit()在扫描其中的匿名类声明时要调用名字引用生成器的scan()方法，并将生成的名字引用列表全部添加到遍历结果字段lastReference（继承自类ExpressionASTVisitor）。所以对于匿名类中的名字引用，在使用类NameTableCreator生成名字表时，这些名字引用是添加在对应匿名类的名字作用域中，而在使用类NameReferenceCreator时，这些名字引用是放在名字引用组NRGClassInstanceCreation的实例中（因为我们不能存储在对应匿名类的作用域，而需要返回给构件使用者临时使用）。

类ExpressionReferenceASTVisitor的构造方法以名字生成器、编译单元信息和作用域为参数，将这些信息存放在相应的字段（继承自类ExpressionASTVisitor的字段creator总是置为null），其中一个构造方法将编译单元信息包装在类CompilationUnitFile的实例中，另一个构造方法则直接使用单元名及其抽象语法树根节点为参数。

5.5.9 类NameTableCreator

类NameTableCreator的实例是名字表生成器，同时生成名字定义、作用域和名字引用，名字定义和名字引用都按其所属的（最内层）作用域存放。

表5.47给出类NameTableCreator的字段，其中字段codeFileSet给出源代码文件集，所生成的名字表就是该源代码文件集的名字表。字段expressionVisitor是抽象语法树表达式节点访问器，因为在扫描字段声明时会碰到初始化表达式，扫描枚举常量声明时会碰到其参数（表达式），这两者都需要使用表达式节点访问器得到相应的名字引用。字段typeVisitor是抽象语法树类型引用节点访问器，在扫描类型声明、字段声明、方法声明时都碰到各种类型引用，所以需要类型引用节点访问器。由于这两种访问器都是可重入的，所以在类NameTableCreator的实例中对这两个访问器只分别创建一个实例，并且在需要时不断重用。

表 5.47 类NameTableCreator字段的描述

字段	类型	描述
codeFileSet	SourceCodeFileSet	建立名字表的源代码文件集
errorUnitList	List<CompilationUnitFile>	生成名字表过程中出现错误的编译单元文件的信息
expressionVisitor	ExpressionASTVisitor	抽象语法树表达式节点访问器
typeVisitor	TypeASTVisitor	抽象语法树类型引用节点访问器

表5.48给出类NameTableCreator的方法，它的构造方法以源代码文件集为参数。类NameTableCreator对名字表生成构件的使用者主要是提供两种方法：

(1) 一系列的createNameTableManager()方法用以生成名字表，构件使用者由此得到类NameTableManager的实例，从而可进一步访问和使用名字表实体。在生成名字表时，使用者可提供输出流用于输出扫描的编译单元文件信息，以展示名字表生成的过程，也可提供一个文件名（字符串）数组，这些文件用于提取导入类型定义的更多信息。输出流和文件名数组这两个参数都是可选的，可以为null。

(2) 几个与字段errorUnitList相关的方法，包括方法hasError()判断名字表生成过程中是否有错误编译单元、方法getErrorUnitList()返回出现错误的编译单元文件列表、方法getErrorUnitNumber()返回出现错误的编译单元文件数目和printErrorUnitList()打印出现错误的编译单元文件信息到指定输出流。

方法createNameTableManager()生成名字表，它调用程序包级方法create(PrintWriter)，后者扫描源代码文件集中的每个编译单元文件（调用方法scanCurrentCompilationUnit()），记录出现错误的编译单元文件信息，并将所扫描的编译单元文件信息打印到给定的输出流设备。方法scanCurrentCompilationUnit()扫描一个编译单元，处理程序包声明和导入声明，然后扫描其中声明的每个顶层类型（调用方法scan(..., TypeDeclaration, ...)或方法scan(..., EnumDeclaration, ...)方法），这些方法又调用其他的scan()方法扫描枚举常量、字段声明、初始化块和方法声明。这些scan()方法生成相应的名字定义、作用域和名字定义，且在扫描初始化块和方法声明时使用类BlockASTVisitor的实例访问以Block类型节点为根抽象语法（子）树，提取其中的名字定义、作用域和名字引用。

类NameTableCreator的扫描匿名类的scan(..., AnonymousClassDeclaration, ...)方法则用于类Block...ASTVisitor或类Expression...ASTVisitor在处理类型为ClassInstanceCreation的抽象语法树节点时扫描匿名类声明。

表 5.48 类NameTableCreator方法的描述

<code>NameTableCreator(SourceCodeFileSet) : Constructor</code>	构造方法, 以源代码文件集为实例
<code>create(PrintWriter) : SystemScope</code>	生成名字表, 并将生成过程中所扫描的编译单元文件信息输出到指定的输出流(如果不为null的话)
<code>createLocalScope(SourceCodeLocation, SourceCodeLocation, NameScope) : LocalScope</code>	使用指定的起始位置和结束位置创建局部作用域, 且该局部作用域的父作用域是方法参数中给定的作用域
<code>createNameTableManager(PrintWriter, String[]) : NameTableManager</code>	生成名字表, 给定的输出流(如果不为null)用于输出扫描的编译单元文件信息(因为生成过程可能非常久, 需要一些输出信息来填补程序运行时的用户等待时间)。字符串数组给出一些额外的文件(全名), 这些文件用于提取有关导入类型定义的更多信息
<code>createNameTableManager(String[]) : NameTableManager</code>	生成名字表, 字符串给出一些额外的文件全名, 这些文件用于提取有关导入类型定义的更多信息
<code>createNameTableManager(PrintWriter) : NameTableManager</code>	生成名字表, 给定的输出流(如果不为null)用于输出扫描的编译单元文件信息
<code>createNameTableManager() : NameTableManager</code>	生成名字表, 既不输出扫描编译单元文件信息, 也没有额外文件用于提取导入类型定义的更多信息
<code>defineField(CompilationUnitFile, String, VariableDeclaration, TypeReference, NameScope, int) : void</code>	基于给定的类型为VariableDeclaration的抽象语法树节点, 在指定的作用域中定义一个字段(即添加一个字段定义)。其他参数分别给出编译单元信息、字段全限定名中的限定名(即除简单名外的名字), 该字段声明时的类型, 及字段声明的修饰符信息
<code>defineParameter(CompilationUnitFile, SingleVariableDeclaration, NameScope) : void</code>	基于给定的类型为SingleVariableDeclaration的抽象语法树节点, 在指定的作用域中定义一个参数(注意实际上是添加一个类VariableDefinition的实例)
<code>defineVariable(CompilationUnitFile, VariableDeclaration, TypeReference, NameScope) : void</code>	基于给定的类型为VariableDeclaration的抽象语法树节点, 在指定的作用域中定义一个变量(即添加一个VariableDefinition的实例), 类型为TypeReference的参数给出该变量声明时的类型
<code>getErrorUnitList() : List<CompilationUnitFile></code>	返回在生成名字表过程中有错误的编译单元文件信息列表, 列表的每个元素包括单元名以及错误信息。通常的错误包括为该编译单元生成抽象语法树出错, 或者该编译单元中根本不包含任何类型声明
<code>getErrorUnitNumber() : int</code>	返回在生成名字表过程中有错误的编译单元文件数目
<code>hasError() : boolean</code>	返回在生成名字表过程中是否有错误的编译单元文件
<code>needCreateLocalScope(Block) : boolean</code>	给定一个类型为Block的抽象语法树节点, 判断是否要为该节点创建相应的局部作用域
<code>printErrorUnitList(PrintWriter) : void</code>	在指定的输出流中打印生成名字表过程中有错误的编译单元文件信息
<code>scan(CompilationUnitFile, String, TypeDeclaration, NameScope) : void</code>	扫描一个类型声明(类型为TypeDeclaration抽象语法树节点)以生成名字定义和名字引用, 其他参数分别给出编译单元信息、类型定义的全限定名中的限定名(即除简单名外的名字)和类型定义所属的作用域
<code>scan(CompilationUnitFile, Initializer, DetailedTypeDefinition) : void</code>	扫描一个类型声明的初始化块, 该类型声明所对应详细类型定义已知, 并由类型为DetailedTypeDefinition的参数给出
<code>scan(CompilationUnitFile, String, EnumDeclaration, NameScope) : void</code>	扫描一个枚举类型, 该枚举声明所属的作用域由参数为NameScope给出
<code>scan(CompilationUnitFile, String, FieldDeclaration, NameScope) : void</code>	扫描一个字段声明(类型为FieldDeclaration的抽象语法树节点), 其他参数分别给出的编译单元信息, 字段的全限定名中的限定名, 以及所属的作用所域
<code>scan(CompilationUnitFile, String, MethodDeclaration, NameScope) : void</code>	扫描一个方法声明(类型为MethodDeclaration的抽象语法树节点), 其他参数分别给出的编译单元信息, 方法的全限定名中的限定名, 以及所属的作用所域
<code>scan(CompilationUnitFile, String, EnumConstantDeclaration, NameScope) : void</code>	扫描一个枚举常量声明, 其他参数分别给出的编译单元信息, 枚举常量的全限定名中的限定名, 以及所属的作用所域
<code>scan(CompilationUnitFile, String, AnonymousClassDeclaration, NameScope, TypeReference) : void</code>	扫描一个匿名类型声明, 其他参数分别给出编译单元信息, 限定名信息, 所属作用域, 以及声明该匿名类时所创建实例的所属类型(引用)
<code>scanCurrentCompilationUnit(CompilationUnitFile, SystemScope) : int</code>	扫描一个编译单元, 其信息(即单元名和抽象语法树根节点)由类型为CompilationUnitFile的参数指定, 类型为SystemScope给出该编译单元所属的系统作用域

类NameTableCreator()提供三个define...()方法分别在指定作用域中定义字段、参数和变量。由于在一个变量声明中可能使用一个类型（引用）声明多个变量，因此这些方法都需要类型为TypeReference的参数预先指定字段、参数或变量的声明类型，这些方法只是从类型为VariableDeclaration或SingleVariableDeclaration的抽象语法树节点中提取一个变量的名字以及其中可能存在的数组维数信息。

在所有的方法scan()和方法define...()中，我们都使用字符串类型的参数来给出一个要添加到作用域的名字定义的限定名部分，名字定义的限定名是在扫描的过程中产生，例如一个字段或方法的限定名是它所属类型的全限定名。同时，在这些方法中都需要编译单元信息（类CompilationUnitFile的实例）以确定名字定义（作用域、名字引用）的源代码位置。

5.5.10 类NameDefinitionCreator

类NameDefinitionCreator是类NameTableCreator的直接子类，它的实例是名字定义（和作用域）生成器，只生成名字定义、作用域和必要的名字引用（主要是一些类型引用）。它没有声明额外的字段，而且除构造方法外，只是重定义类NameTableCreator中的部分scan()方法和define...()方法，去掉这些方法中生成名字引用的语句，而只保留生成名字定义、作用域以及必要的类型引用的语句。图5.35给出类NameDefinitionCreator的成员。



图 5.35 类NameDefinitionCreator的成员

5.5.11 类NameReferenceCreator

类NameReferenceCreator不是类NameTableCreator的子类，它的实例是名字引用生成器，在已经生成名字定义（和作用域）的基础上，生成指定范围内的名字引用。这个类只有一个字段是名字表管理器类NameTableManager 的实例，构造方法也以一个名字表管理器类的实例为参数。

表5.49给出类NameReferenceCreator的方法，它主要是提供了一系列的createReferences()方法生成编译单元、详细类型定义、枚举类型定义、初始化块、字段定义、方法定义、匿名类声明中的名字引用。这些createReferences()可分为两类：

(1). 只需提供一个参数，分别生成指定编译单元作用域、详细类型定义、字段定义和方法定义中的名字引用；

(2). 需要提供要生成名字引用的名字表实体及其对应的抽象语法节点的更详细信息。在这些方法中，前两个参数分别指定编译单元名及其抽象语法树根节点，后面的参数给定该名字表实体及其所属的作用域，或者对应的抽象语法树节点。

表 5.49 类NameReferenceCreator方法的描述

<code>NameReferenceCreator(NameTableManager) : Constructor</code>	构造方法，以名字表管理器类的实例为参数
<code>createReferenceForExpressionASTNode(String, Expression) : NameReference</code>	为指定的表达式节点生成名字引用，字符串类型的参数指定节点所在的编译单元名
<code>createReferences(CompilationUnitScope) : List<NameReference></code>	生成指定的编译单元（作用域）中的所有名字引用
<code>createReferences(String) : List<NameReference></code>	生成编译单元名是指定字符串的编译单元中的所有名字引用
<code>createReferences(String, CompilationUnit, DetailedTypeDefinition, TypeDeclaration) : List<NameReference></code>	生成指定的详细类型定义中的所有名字引用，其他参数分别给出该详细类型定义所属的编译单元名、编译单元抽象语法树根节点，和给详细类型定义对应的类型声明（抽象语法树）节点
<code>createReferences(DetailedTypeDefinition) : List<NameReference></code>	生成指定的详细类型定义中的所有名字引用
<code>createReferences(String, CompilationUnit, DetailedTypeDefinition, Initializer) : List<NameReference></code>	生成指定的初始化块中的所有名字引用，其他参数分别给出该初始化块所属的编译单元名、编译单元抽象语法树根节点和所属的详细类型定义
<code>createReferences(String, CompilationUnit, EnumTypeDefinition, EnumDeclaration) : List<NameReference></code>	生成指定的枚举类型定义中的所有名字引用，其他参数分别给出该枚举类型所属的编译单元名、编译单元抽象语法树根节点和对应的枚举类型声明（抽象语法树节点）
<code>createReferences(String, CompilationUnit, NameScope, AnonymousClassDeclaration) : List<NameReference></code>	生成指定的匿名类型声明（抽象语法树节点中）的所有名字引用，其他参数分别给出该匿名类声明所属的编译单元名、编译单元抽象语法树根节点和所属的作用域
<code>createReferences(String, CompilationUnit, DetailedTypeDefinition, FieldDeclaration) : List<NameReference></code>	生成指定的字段定义中的所有名字引用，其他参数分别给出该字段所属的编译单元名、编译单元抽象语法树根节点和所属的详细类型定义
<code>createReferences(FieldDeclaration) : List<NameReference></code>	生成指定的字段中的所有名字引用
<code>createReferences(String, CompilationUnit, MethodDefinition, MethodDeclaration) : List<NameReference></code>	生成指定的方法定义中的所有名字引用，其他参数分别给出该方法定义所属的编译单元名、编译单元抽象语法树根节点和对应的方法声明（抽象语法树节点）
<code>createReferences(MethodDefinition) : List<NameReference></code>	生成指定的方法定义中的所有名字引用
<code>createReferencesBindedToDefinition(NameDefinition) : List<NameReference></code>	生成绑定到指定名字定义的所有名字引用
<code>createReferencesByFilter(NameTableFilter) : List<NameReference></code>	生成为指定名字表过滤器所接受的所有名字引用
<code>createReferencesForASTNode(String, ASTNode) : List<NameReference></code>	生成指定的抽象语法树节点中的所有名字引用，注意该抽象语法树节点必须在局部作用域中，字符串参数指定该抽象语法树节点所属的编译单元名
<code>createReferencesOfName(String) : List<NameReference></code>	生成名字是字段字符串的所有名字引用

我们为最常用的编译单元、详细类型定义、字段定义和方法定义给出了只需提供一个参数的最简单的生成名字引用的方法，构件使用者应尽量使用这些方法生成名字引用，除非是构件使用者自己扫描编译单元，能逐步获得编译单元中详细类型定义、字段定义和方法定义的信息（这时由于已经有更多信息就可以调用更多参数的`createReferences()`方法生成名字引用，避免重复查找这些信息带来的时间开销）。实际上，只需一个参数的生成名字引用的方法内部是通过类`NameTableManager`的实例查找所属的编译单元信息，并通过类`NameTableASTBridge`的实例查找相应的抽象语法树节点，再调用对应的需要多个参数的`createReferences()`方法而实现的。

通常构件使用者不应单独生成某个初始化块或匿名类型定义中的名字引用，因为要找到初始化块和匿名类型定义对应的抽象语法树节点不容易。生成详细类型定义和方法定义中名字引用的方法`createReferences()`所生成的名字引用列表中包含该详细类型定义中的初始化块中的名字引用，或方法定义中的匿名类声明中的名字引用。目前暂时也没有为枚举类型定义提供只需一个参数的生成名字引用的方法，因为枚举类型定义中通常并不存在名字引用。

类`NameReferenceCreator`还有方法`createReferencesBindedToDefinition()`方法生成绑定到指定名字定义的所有名字引用，这个方法可能所需的时间开销非常大，因为它是通过遍历所有的编译单元，生成每个编译单元中的所有名字引用，然后解析这些名字引用，并检查所绑定的名字定义是否指定的名字定义而实现。所返回的列表只包含最基本的名字引用（即不包含名字引用组），因为只有处于叶子位置的名字引用才通常是我们真正关心的名字引用，特别是关注名字引用所绑定的名字定义时。

类`NameReferenceCreator`的方法`createReferencesByFilter()`和`createReferencesOfName()`也都是通过遍历所有编译单元，生成每个编译单元中的所有名字引用，然后对生成的名字引用进行过滤（或检查其名字）而实现，因此所需的时间可能也会比较长。

类`NameReferenceCreator`的方法`createReferenceForExpressionASTNode()`为某个表达式节点生成其对应的一个名字引用（组），它直接使用类`ExpressionReferenceASTVisitor`的实例访问该表达式节点得到生成的名字引用（组）。方法`createReferencesForASTNode()`则为某个抽象语法树节点生成其中的名字引用，这时需要指定该语法树节点所属的编译单元名。这个方法使用类`BlockReferenceASTVisitor`的实例访问该节点得到生成的名字引用列表，由于这个实例需要局部作用域作为参数创建，所以该抽象语法树节点必须属于某个局部作用域。这个方法通过类`NameTableManager`的方法`getScopeOfLocation()`确定一个抽象语法树节点所属的作用域。编译单元名用于查找相应的抽象语法树根节点，并确定该抽象语法树节点的源代码（起始）位置。

5.5.12 类`ImportedTypeManager`

类`ImportedTypeManager`的实例用于从额外的文件中读取有关导入类型的更多信息，以便能对不在所分析的源代码中的导入类型有更多信息而使得名字解析更为准确。正如图5.5所给出的描述导入类型信息的文件例子所看到的，描述导入类型信息可以将导入类型看做是抽象类，给出所属程序包、类型声明、字段声明和不含方法体的方法声明即可，而且可以只给出部分字段和部分方法（的基调）。

图5.36给出类`ImportedTypeManager`的成员，它提供了一系列的静态方法供类`NameTableCreator`或类`NameDefinitionCreator`使用以管理有关导入类型的更多信息。

类`ImportedTypeManager`只有一个静态字段`typeVisitor`用于访问导入类型信息中的类型

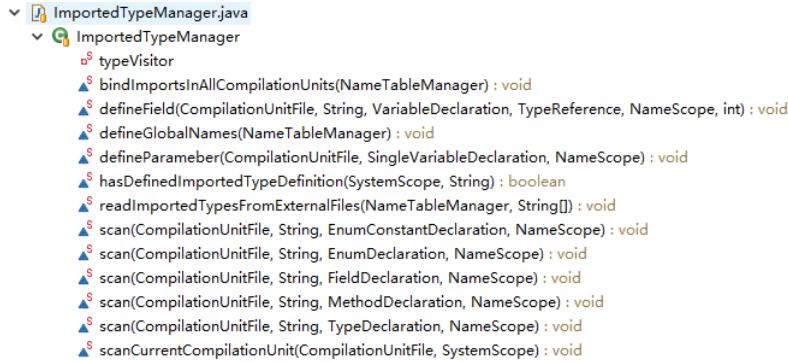


图 5.36 类ImportedTypeManager的成员

引用节点，生成类型引用。类ImportedTypeManager的实现有点类似类NameTableCreator，方法readImportedTypesFromExternalFiles()将字符串数组中给出的文件看做是一个编译单元，生成它的抽象语法树，然后调用方法scanCurrentCompilationUnit()方法扫描这个编译单元，后者的实现与类NameTableCreator的同名方法类似，调用其他的scan()方法完成类型声明、方法声明、字段声明的扫描以获得导入类型信息中的名字定义（和名字作用域，如类型作用域、方法作用域等），以及必要的类型引用信息。当然导入类型信息文件中没有方法体，所以无需使用类Block...ASTVisitor的实例扫描方法体，也无需类Expression...ASTVisitor的实例扫描表达式节点生成名字引用。

类ImportedTypeManager的方法hasDefinedImportedTypeDefinition()方法用于判断一个导入类型定义是否已经在系统作用域中，如果已经存在则不再添加以避免存在重复的导入类型定义。方法bindImportsInAllCompilationUnits()用于将每个编译单元中的导入声明所创建的名字引用绑定到导入类型定义，以便在名字解析时能使用导入类型定义的信息。方法defineGlobalNames()在系统作用域中添加Java语言自动导入的类型（如java.lang.String）以及原子类型对应的导入类型定义（例如int等）。

类NameTableCreator的createNameTableManager()方法在调用create()方法生成名字表实体之后，则使用类ImportedTypeManager的方法readImportedTypesFromExternalFiles()读取导入类型信息文件（如果有的话），然后使用方法bindImportsInAllCompilationUnits()绑定导入类型定义，在绑定过程中如果遇到在系统作用域中不存在的导入类型定义，则会自动生成一个最简单的导入类型定义，最后使用defineGlobalNames()定义自动导入的类型和原子类型所对应的导入类型定义，完成这些准备工作之后，才能进行其他名字引用的绑定。

第六章 名字表生成构件的使用

名字表生成构件的使用者通常先利用类NameTableCreator或类NameDefinitionCreator的实例生成名字表，得到类NameTableManager的实例，然后通过名字表访问器和名字表过滤器获取所需要的名字表实体，然后对名字表实体的成员进行访问和使用。当使用类NameDefinitionCreator只生成名字定义和作用域时，需要访问名字引用时，在通过类NameReferenceCreator生成指定范围的名字引用。

因此名字表生成构件的使用者主要是使用名字表构件中下面的一些类：

1. 使用程序包nameTable.creator中的类NameTableCreator、类NameDefinitionCreator和类NameReferenceCreator的生成名字表实体；
2. 使用程序包nameTable中的类NameTableManager作为名字表入口访问名字表，在访问时可能会用到程序包nameTable.visitor中的名字表访问器和程序包nameTable.filter中的名字表过滤器。当需要建立名字表实体与抽象语法树节点之间的关联时，还需使用程序包nameTable中的类NameTableASTBridge；
3. 在得到名字表实体之后需要使用代表名字定义的类、代表名字作用域的类和代表名字引用的类，访问这些类的成员以使用名字表实体。不过对于名字引用来说，最常用的是解析名字引用得到它所绑定的名字定义。通常在使用名字引用时不用关心具体的名字引用组，也即构件使用者通常不会需要直接使用名字引用组类NameReferenceGroup的子类。

程序包nameTable中的编译单元文件TestNameTable.java给出了如何使用名字表的一些例子，下面给出的例子程序都来自于该文件。

6.1 打印所有名字表实体

首先我们介绍如何生成（简单）程序的名字表，包括名字定义、名字作用域和名字引用。编译单元TestNameTable.java 的方法printAllNames(String, String)可打印生成名字表，并打印所有的名字定义、作用域和名字引用。

图6.1给出了方法printAllNames()的源代码。这个方法有两个参数，path给出源代码文件集的起始路径，resultFile给出将名字表实体打印到哪个文件。

在方法printAllNames()中，第111行语句（行号都在文件TestNameTable.java中的行号为准）创建源代码文件集类SourceCodeFileSet的实例，基于这个实例可创建名字表生成器NameTableCreator的实例。第113行的字符串数组给出了含有导入类型信息的额外文件名数组（这一行由于图形大小的缘故没有截全）。

```

110 public static void printAllNames(String path, String resultFile) throws IOException {
111     SourceCodeFileSet parser = new SourceCodeFileSet(path);
112     NameTableCreator creator = new NameTableCreator(parser);
113     String[] fileNameArray = {"C:\\ZxcWork\\ToolKit\\data\\javalang.txt", "C:\\ZxcWork\\ToolKit\\data\\java";
114
115     Debug.setStart("Begin creating system, path = " + path);
116     NameTableManager manager = creator.createNameTableManager(new PrintWriter(System.out), fileNameArray);
117     if (creator.hasError()) {
118         System.out.println("There are " + creator.getErrorUnitNumber() + " error unit files:");
119         creator.printErrorUnitList(new PrintWriter(System.out));
120         System.out.println();
121     }
122     Debug.time("End creating.....");
123     Debug.flush();
124
125     PrintWriter writer = new PrintWriter(new File(resultFile));
126     NameDefinitionPrinter definitionPrinter = new NameDefinitionPrinter(writer);
127     definitionPrinter.setPrintVariable(true);
128     manager.accept(definitionPrinter);
129
130     writer.println();
131
132     NameReferencePrinter referencePrinter = new NameReferencePrinter(writer);
133     referencePrinter.setPrintBindedDefinition();
134     manager.accept(referencePrinter);
135     writer.close();
136 }
```

图 6.1 方法printAllNames()的源代码

第116行调用`creator`的`createNameTableManager()`方法生成名字表，得到类`NameTableManager`的实例，然后第117到121行的if语句查看生成过程中是否有错误的编译单元，如果有则打印到控制台。注意，即使有错误的编译单元也不妨碍名字表的使用。名字表将包含那些没有错误的编译单元的名字定义、作用域和名字引用。

方法`printAllNames()`的第125行到128行使用程序包`nameTable.visitor`的类`NameDefinitionPrinter`的实例按作用域分层次打印名字表中的所有名字定义，这只要使用输出流设备`writer`创建类`NameDefinitionPrinter` 的实例，然后名字表管理器`manager`接受（调用方法`accept()`该实例即可）。

类似地，第132行到134行使用`NameReferencePrinter`的实例按作用域分层次打印名字表中的所有名字引用，这也只要使用输出流设备创建这个类的实例，然后调用名字表管理器的`accept()`方法即可。

方法`printAllNames()`的运行结果将在名为`resultFile`指定的字符串的文本文件中输出起始路径由`path` 指定的源代码文件集中的所有的名字定义和名字引用，而且这些名字定义和名字引用按作用域分层次打印（这同时也输出了名字作用域信息）。由于输出的信息很多，这里不给出输出的样例，读者可自行运行`TestNameTable.java`文件并查看相应的运行结果。

6.2 打印所有名字定义

编译单元`TestNameTable.java`的方法`printAllDefinitions(String, String)`只生成名字定义和名字作用域，然后使用类`NameDefinitionPrinter`的实例按作用域分层次打印所有的名字定义，其关键语句域方法`printAllNames()`相同。图6.2给出这个方法的源代码。

编译单元`TestNameTable.java`的另一个方法`printAllDefinitionsToTable(String, String)`也只生成名字定义和名字作用域，但不是使用类`NameDefinitionPrinter`的实例打印名字定义，而是以自定义的表格（实际上是用制表符分隔名字定义的重要信息）形式打印名字定义，从而展示有关名字表访问的更多方式。

```

140@ public static void printAllDefinitions(String path, String resultFile) throws IOException {
141    SourceCodeFileSet parser = new SourceCodeFileSet(path);
142    NameTableCreator creator = new NameDefinitionCreator(parser);
143    String[] fileNameArray = {"C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\javlang.txt", "C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\javutil.txt"};
144
145    Debug.setStart("Begin creating system, path = " + path);
146    NameTableManager manager = creator.createNameTableManager(new PrintWriter(System.out), fileNameArray);
147    Debug.time("End creating.....");
148    Debug.flush();
149
150    NameDefinitionPrinter printer = new NameDefinitionPrinter(new PrintWriter(new File(resultFile)));
151    printer.setPrintVariable(true);
152    manager.accept(printer);
153    printer.close();
154}

```

图 6.2 方法printAllDefinitions()的源代码

图6.3给出方法printAllDefinitionsToTable()的源代码。第215到219行以方法参数中的起始路径创建源代码文件集，然后创建名字定义生成器，最后生成名字表，得到名字管理器类NameTableManager的实例。为简便起见，后面介绍的方法都不再处理生成名字表时出错的编译单元文件信息。

```

214@ public static void printAllDefinitionsToTable(String path, String resultFile) throws IOException {
215    SourceCodeFileSet parser = new SourceCodeFileSet(path);
216    NameTableCreator creator = new NameDefinitionCreator(parser);
217    String[] fileNameArray = {"C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\javlang.txt", "C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\javutil.txt"};
218
219    final NameTableManager manager = creator.createNameTableManager(new PrintWriter(System.out), fileNameArray);
220
221    PrintWriter writer = new PrintWriter(new File(resultFile));
222    writer.println("Location\tFullQualifiedName\tSimpleName\tKind\tScope");
223
224    NameDefinitionVisitor visitor = new NameDefinitionVisitor();
225    visitor.setFilter(new NameTableFilter() {
226        // Exclude those simple definitions defined in system scope!
227        public boolean accept(NameDefinition definition) {
228            if (definition.isTypeDefinition() && definition.getScope().equals(manager.getSystemScope())) return false;
229            return true;
230        }
231    });
232
233    manager.accept(visitor);
234    List<NameDefinition> definitionList = visitor.getResult();
235
236    for (NameDefinition definition : definitionList) {
237        System.out.println("Write definition " + definition);
238
239        String fullName = definition.getFullQualifiedName();
240        String simpleName = definition.getSimpleName();
241        SourceCodeLocation location = definition.getLocation();
242        String locationString = "~";
243        if (location != null) locationString = "(" + location.toString() + ")";
244        String scope = definition.getScope().getScopeName();
245        String kind = definition.getDefinitionKind() + "";
246        writer.println(locationString + "\t" + fullName + "\t" + simpleName + "\t" + kind + "\t" + scope);
247    }
248    writer.close();
249}

```

图 6.3 方法printAllDefinitionsToTable()的源代码

方法printAllDefinitionsToTable()的第224行创建一个名字定义访问器visitor，第225行使用匿名类声明并为该名字定义访问器设置一个名字定义过滤器，当一个名字定义属于系统作用域并且是类型定义时不被过滤器接受。设置过滤器后，调用名字表管理器的accept()方法则使用该名字定义访问器遍历名字表，调用名字定义访问器的getResult()方法得到遍历的结果，它是哪些除系统作用域中的类型定义外的所有名字定义构成的列表。

方法printAllDefinitionsToTable()的第236到第247行的for循环打印遍历结果列表中的名字定义，输出名字定义的源代码位置、全限定名、简单名、类别和所属作用域信息，每一行对应一个名字定义，这些信息直接使用制表符分隔，第222行给出了表头信息。很容易将打印得到的文本文件拷贝到Excel等电子表格软件中得到真正的表格。注意，有些名字定义没有源代码位置（即会返回

为null的源代码位置对象实例)。名字定义源代码位置信息串的缺省值是"~~"，这是因为直接使用空白字符串会与制表符混淆，而波浪号在LATEX中就是生成空白串。实际上，前面表3.3给出的例子程序中的名字定义列表就是使用方法printAllDefinitionsToTable()打印的结果再转换为LATEX表格而得到。

6.3 打印所有名字作用域

编译单元TestNameTable.java的方法printAllNameScopesToTable(String, String)只生成名字定义和名字作用域，然后使用名字作用域访问器和过滤器，按表格形式打印出所有的名字作用域。图6.4给出这个方法的源代码。

```

255@ public static void printAllNameScopesToTable(String path, String resultFile) throws IOException {
256    SourceCodeFileSet parser = new SourceCodeFileSet(path);
257    NameTableCreator creator = new NameDefinitionCreator(parser);
258    String[] fileNameArray = {"C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\javalang.txt", "C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\java\\\\lang\\\\String.txt"};
259
260    NameTableManager manager = creator.createNameTableManager(new PrintWriter(System.out), fileNameArray);
261
262    PrintWriter writer = new PrintWriter(new File(resultFile));
263    writer.println("Name\\tStart\\tEnd\\tParent\\tKind");
264
265    NameScopeVisitor visitor = new NameScopeVisitor();
266    SystemScope rootScope = manager.getSystemScope();
267
268    rootScope.accept(visitor);
269    List<NameScope> scopeList = visitor.getResult();
270
271    for (NameScope scope : scopeList) {
272        String name = scope.getScopeName();
273        SourceCodeLocation start = scope.getScopeStart();
274        String startString = "~~";
275        if (start != null) startString = "(" + start.toString() + ")";
276
277        SourceCodeLocation end = scope.getScopeEnd();
278        String endString = "~~";
279        if (end != null) endString = "(" + end.toString() + ")";
280
281        NameScope parentScope = scope.getEnclosingScope();
282        String parentString = "~~";
283        if (parentScope != null) parentString = parentScope.getScopeName();
284
285        String kind = scope.getScopeKind() + "";
286
287        writer.println(name + "\\t" + startString + "\\t" + endString + "\\t" + parentString + "\\t" + kind);
288    }
289    writer.close();
290}

```

图 6.4 方法printAllNameScopesToTable()的源代码

方法printAllNameScopesToTable()的实现与方法printAllDefinitionsToTable()类似，只是在第265行创建的是名字作用域访问器类NameScopeVisitor的实例，而且没有设置过滤器(这意味着打印所有的名字作用域)。第271到第288行的for循环遍历名字作用域访问器visitor的访问结果(名字作用域列表)，输出名字作用域的名字、起始位置、结束位置、父作用域和类别信息。实际上，前面的表3.4给出的名字作用域信息就是基于这个方法的打印结果。

注意，方法printAllNameScopesToTable()的第266行使用类NameTableManager的方法getSystemScope()得到名字表的系统作用域，然后调用系统作用域的accept()方法，接受名字作用域访问器的实例访问名字表得到所需的名字作用域列表。实际上，直接调用manager的accept()也完成完全相同的功能，这里只是展示不同的方式而已，不过我们推荐直接使用NameTableManager的accept()方法，因为这使得代码更简短。

6.4 打印所有名字引用

编译单元TestNameTable.java的方法printAllReferencesToTable()先使用类NameDefinitionCreator的实例声明名字定义和名字作用域，并利用名字定义访问器和过滤器在名字表中找到指定简单名的详细类型定义，然后使用类NameReferenceCreator的实例生成这些详细类型定义中的名字引用，并以表格形式打印到指定的文本文件。

```

160 public static void printAllReferencesToTable(String path, String resultFile) throws IOException {
161     SourceCodeFileSet parser = new SourceCodeFileSet(path);
162     NameTableCreator creator = new NameDefinitionCreator(parser);
163     String[] fileNameArray = {"C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\javalang.txt", "C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\java\\\\lang\\\\NameTableManager.txt"};
164     NameTableManager manager = creator.createNameTableManager(new PrintWriter(System.out), fileNameArray);
165
166     PrintWriter writer = new PrintWriter(new File(resultFile));
167     writer.println("Type\tName\tLocation\tKind\tScope\tDefinition");
168     NameDefinitionVisitor visitor = new NameDefinitionVisitor();
169     NameTableFilter filter = new DetailedTypeDefinitionFilter() {
170         public boolean accept(NameDefinition definition) {
171             if (definition.isDetailedType() &&
172                 definition.getSimpleName().equals("NameDefinition")) return true;
173             return false;
174         }
175     };
176     visitor.setFilter(filter);
177     manager.accept(visitor);
178     List<NameDefinition> typeList = visitor.getResult();
179     NameReferenceCreator referenceCreator = new NameReferenceCreator(manager);
180
181     for (NameDefinition definition : typeList) {
182         DetailedTypeDefinition type = (DetailedTypeDefinition)definition;
183         System.out.println("Write references in type " + type.getFullQualifiedName());
184
185         List<NameReference> referenceList = referenceCreator.createReferences(type);
186         for (NameReference reference : referenceList) {
187             reference.resolveBinding();
188             List<NameReference> leafReferenceList = reference.getReferencesAtLeaf();
189             for (NameReference leafReference : leafReferenceList) {
190                 if (leafReference.isLiteralReference()) continue;
191
192                 String name = leafReference.getName();
193                 String location = "(" + leafReference.getLocation().toString() + ")";
194                 String kind = leafReference.getReferenceKind() + "";
195                 String scopeName = leafReference.getScope().getScopeName();
196                 NameDefinition bindDef = leafReference.getDefinition();
197                 String bindString = "~~";
198                 if (bindDef != null) bindString = bindDef.getUniqueId();
199                 writer.println(type.getFullQualifiedName() + "\t" + name + "\t" + location + "\t" +
200                             kind + "\t" + scopeName + "\t" + bindString);
201             }
202         }
203     }
204     writer.close();
205 }
```

图 6.5 方法printAllReferencesToTable()的源代码

方法printAllReferencesToTable()的第161行到第164行使用类NameDefinitionCreator的实例生成名字定义和名字作用域，得到名字表管理器NameTableManager的实例。第168行到第178行创建名字定义访问器实例，并使用匿名类的方式创建一个名字表过滤器，只接受简单名为"NameDefinition"的详细类型定义，列表typeList是遍历结果（即简单名为"NameDefinition"的详细类型定义，可能多个，但在我们的程序例子中只有一个）。

第179行使用名字表管理器创建名字引用生成器类NameReferenceCreator的实例。第181行到第203行的for循环遍历typeList中的名字定义，并将其（安全地）转换为详细类型定义（因为过滤器中已经确定只接受详细类型定义）。第185行生成该详细类型定义中的所有名字引用构成的列表。第186行到第202行的for循环遍历生成的名字引用列表，先调用resolveBinding()对名字引用进行解析（第187行），第188行使用方法getReferencesAtLeaf()得到该名字引用（因为可能是名字引用

组、方法引用或参数化类型引用) 中最基本名字引用的列表(避免将名字引用组或复杂的名字引用一起输出)。第189行到第201行的for循环输出这些最基本名字引用的所属的详细类型定义、名字引用中的名字、源代码位置、名字引用类别以及所绑定的名字定义等信息,其中第190行跳过文字常量引用(因为用户一般不关心对文字常量的引用)。

6.5 打印引用给定名字定义的所有名字引用

编译单元TestNameTable.java的方法printReferenceBindToDefinition()打印引用给定名字定义的所有名字引用。图6.6给出这个方法的部分源代码。

```

294 public static void printReferenceBindToDefinition(String path, String resultFile) throws IOException {
295     SourceCodeFileSet parser = new SourceCodeFileSet(path);
296     NameTableCreator creator = new NameDefinitionCreator(parser);
297     String[] fileNameArray = {"C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\javalang.txt", "C:\\\\ZxcWork\\\\ToolKit\\\\data\\\\jav
298
299     NameTableManager manager = creator.createNameTableManager(new PrintWriter(System.out), fileNameArray);
300
301     PrintWriter writer = new PrintWriter(new File(resultFile));
302     writer.println("CompilationUnit\\tType\\tMethod\\tReference\\tLocation\\tDefinition");
303
304     NameDefinitionFinder finder = new NameDefinitionFinder(new DetailedTypeDefinitionFilter(
305         new NameDefinitionNameFilter("NameDefinition")));
306     manager.accept(finder);
307     NameDefinition definition = finder.getResult();
308     if (definition == null) return;
309
310     NameReferenceCreator referenceCreator = new NameReferenceCreator(manager);
311     List<NameReference> referenceList = referenceCreator.createReferencesBindedToDefinition(definition);
312
313     for (NameReference reference : referenceList) {
314         CompilationUnitsScope unitScope = manager.getEnclosingCompilationUnitsScope(reference);
315         String unitString = "~~";
316         if (unitScope != null) unitString = unitScope.getUnitName();
317
318         DetailedTypeDefinition type = manager.getEnclosingDetailedTypeDefinition(reference);
319         String typeString = "~~";
320         if (type != null) typeString = type.getFullQualifiedName();
321
322         MethodDefinition method = manager.getEnclosingMethodDefinition(reference);
323         String methodString = "~~";
324         if (method != null) methodString = method.getSimpleName();
325
326         String name = reference.getName();
327         String location = "(" + reference.getLocation().toString() + ")";
328         NameDefinition bindDef = reference.getDefinition();
329
330         String bindString = "~~";
331         if (bindDef != null) bindString = bindDef.getUniqueId();
332         System.out.println("Writer reference " + name + ", location " + location + ", refer to definition
333         writer.println(unitString + "\\t" + typeString + "\\t" + methodString + "\\t" + name + "\\t" + locatio
334     }
335
336     finder = new NameDefinitionFinder(new NameDefinitionNameFilter(
337         new NameDefinitionKindFilter(NameDefinitionKind.NDK_METHOD), "resolveBinding"));
338     manager.accept(finder);
339     definition = finder.getResult();

```

图 6.6 方法printReferenceBindToDefinition()的部分源代码

方法printReferenceBindToDefinition()的第295到299行生成名字定义和名字作用域,得到名字表管理器。第304到308行利用名字定义访问器类NameDefinitionFinder的实例查找简单名为"NameDefinition"的详细类型定义,这里不再声明新的名字表过滤器,而是使用类DetailedTypeDefinitionFilter和类NameDefinitionNameFilter的组合进行过滤。

第310行基于名字表管理器创建名字引用生成器,第311行生成绑定到指定名字定义(即变量definition)的所有名字引用构成的列表。注意,该列表仅仅包含最基本的名字引用,因为最基本的名字引用所绑定的名字定义才是最值得关注的。所以第313行到第334行的循环在打印名字引用

时，不用再调用方法`getReferencesAtLeaf()`得到最基本的名字引用。这个循环中使用名字表管理器的`getEnclosing...()`方法获得名字引用所属的编译单元、详细类型定义和方法定义的信息，通过这些信息（比只使用名字引用的唯一标识或源代码位置）更容易定位这个名字引用。

方法`printReferenceBindToDefinition()`的第336到339行使用名字定义访问器查找简单名为"resolveBinding"的方法（注意在给出简单名时不要带上圆括号），后面的源代码与第311行到第334行类似，打印引用（即调用）该方法的名字引用（应该是方法引用）的信息。为简便起见，图6.6没有给出后面的源代码。

6.6 通过控制流图打印名字定义和名字引用

编译单元`TestNameTable.java`的方法`printNamesInCFG()`通过方法的控制流图打印其中的名字定义和名字引用信息。目前控制流图的创建直接基于抽象语法树，控制流图的每个节点称为可执行点（类`ExecutionPoint`的实例），其中保存对应的抽象语法树节点信息（通过类`ExecutionPoint`的方法`getAstNode()`可得到）。

图6.7给出方法`printNamesInCFG()`的第一部分源代码，其中第372行到第375行创建名字表（的名字定义和名字作用域），得到名字表管理器。第378行到第395行利用名字定义访问器和名字定义类别过滤器得到所有方法定义，第383行到第395行的循环通过计算方法定义的结束位置行号减去起始位置行号找到源代码行最多的方法定义。

```

371@ public static void printNamesInCFG(String path, String resultFile) throws IOException {
372     SourceCodeFileSet parser = new SourceCodeFileSet(path);
373     NameTableCreator creator = new NameDefinitionCreator(parser);
374     String[] fileNameArray = {"C:\\ZxcWork\\ToolKit\\data\\javalang.txt", "C:\\ZxcWork\\ToolKit\\data\\javautil.txt", "C:\\";
375     NameTableManager manager = creator.createNameTableManager(new PrintWriter(System.out), fileNameArray);
376
377     // Find the method which has the maximal code lines
378     NameDefinitionVisitor visitor = new NameDefinitionVisitor(new NameDefinitionKindFilter(NameDefinitionKind.NDK_METHOD));
379     manager.accept(visitor);
380     List<NameDefinition> resultList = visitor.getResult();
381     MethodDefinition maxMethod = null;
382     int maxLineNumber = 0;
383     for (NameDefinition definition : resultList) {
384         MethodDefinition method = (MethodDefinition)definition;
385         if (maxMethod == null) {
386             maxMethod = method;
387             maxLineNumber = maxMethod.getEndLocation().getLineNumber() - maxMethod.getLocation().getLineNumber();
388         } else {
389             int currentLineNumber = method.getEndLocation().getLineNumber() - method.getLocation().getLineNumber();
390             if (currentLineNumber > maxLineNumber) {
391                 maxMethod = method;
392                 maxLineNumber = currentLineNumber;
393             }
394         }
395     }
396
397     System.out.println("Find the maximal method " + maxMethod.getFullQualifiedName());
398     CompilationUnitScope unitScope = manager.getEnclosingCompilationUnitScope(maxMethod);
399     String unitName = unitScope.getUnitName();
400     CompilationUnit root = parser.findSourceCodeFileASTRootByFileUnitName(unitName);
401     CFGCreator cfgCreator = new CFGCreator(unitName, root);
402
403     NameTableASTBridge bridge = new NameTableASTBridge(manager);
404     MethodDeclaration methodDeclaration = bridge.findASTNodeForMethodDefinition(maxMethod);
405     if (methodDeclaration == null) {
406         System.out.println("Can not find AST node for method definition " + maxMethod.getUniqueId());
407         return;
408     }
409     DetailedTypeDefinition type = manager.getEnclosingDetailedTypeDefinition(maxMethod);
410     String typeName = type.getSimpleName();
411     // Create control flow graph (CFG) for this maximal method!
412     ControlFlowGraph cfg = cfgCreator.create(methodDeclaration, typeName);
413     if (cfg == null) {
414         System.out.println("Can not create control flow graph for method " + maxMethod.getUniqueId());
415         return;
416     }
417 }
```

图 6.7 方法`printNamesInCFG()`的第一部分源代码

方法`printNamesInCFG()`的第398行到第401行找到该方法定义所属的编译单元及其抽象语

法树根节点，并使用编译单元名和抽象语法树根节点创建控制流图生成器类CFGCreator的实例cfgCreator。第403行到第408行利用类NameTableASTBridge的实例找到该方法定义所对应的方法声明（抽象语法树节点），及其所属的详细类型定义。第412行使用cfgCreator生成该方法的控制流图（类ControlFlowGraph的实例）。

图6.8给出方法printNamesInCFG()的第二部分源代码。第423行得到控制流图中的所有节点列表（列表元素实际上也是类ExecutionPoint的实例），第424行到第459行的循环遍历这个列表，输出控制流图节点中的名字定义和名字引用信息。

```

417
418     NameReferenceCreator referenceCreator = new NameReferenceCreator(manager);
419     PrintWriter writer = new PrintWriter(new File(resultFile));
420     writer.println("Method: " + maxMethod.getUniqueId());
421
422     // Write definitions and references in each execution point (i.e. the node of control flow graph)
423     List<GraphNode> pointList = cfg.getAllNodes();
424     for (GraphNode node : pointList) {
425         ExecutionPoint point = (ExecutionPoint)node;
426         ExecutionPointType pointType = point.getType();
427
428         writer.println("\tExecutionPoint: " + point.getStartLocation() + "--" + point.getEndLocation() + " " + point.getLabel());
429         // Do not write definitions and references for virtual node of CFG, since its AST node frequently include all states
430         // branch or a loop statement.
431         if (pointType.isVirtual()) continue;
432
433         ASTNode astNode = point.getAstNode();
434         List<NameDefinition> definitionList = bridge.getAllDefinitionsInASTNode(unitName, astNode);
435         if (definitionList != null) {
436             if (definitionList.size() > 0) writer.println("\t\tDefinitions: ");
437             for (NameDefinition definition : definitionList) {
438                 System.out.println("Writer definition " + definition.getSimpleName());
439                 writer.println("\t\t\t" + definition.getLocation() + " " + definition.getDefinitionKind() + " " + definition.getName());
440             }
441         }
442
443         List<NameReference> referenceList = referenceCreator.createReferencesForASTNode(unitName, astNode);
444         if (referenceList != null) {
445             if (referenceList.size() > 0) writer.println("\t\tReferences: ");
446             for (NameReference reference : referenceList) {
447                 reference.resolveBinding();
448                 List<NameReference> leafReferenceList = reference.getReferencesAtLeaf();
449                 for (NameReference leafReference : leafReferenceList) {
450                     if (leafReference.isResolved()) {
451                         writer.println("\t\t\t" + leafReference.getLocation() + " " + leafReference.getReferenceKind() + " " +
452                     } else {
453                         writer.println("\t\t\t" + leafReference.getLocation() + " " + leafReference.getReferenceKind() + " " +
454                     }
455                     System.out.println("Writer reference " + leafReference.getName());
456                 }
457             }
458         }
459     }
460     writer.close();
461 }
```

图 6.8 方法printNamesInCFG()的第二部分源代码

方法printNamesInCFG()的第426行得到控制流图的节点类型，跳过控制流图中的虚拟节点（即不是真正的语句，而是为了生成控制流图方便而创建的一些虚拟节点，例如方法控制流图的起始和终止节点，分支和循环的汇合节点等等）。第433行得到控制流图节点对应的抽象语法树节点，然后第434行可通过类NameTableASTBridge的实例bridge得到抽象语法树节点中的所有名字定义，第435行到第442行输出这些名字定义的信息（有些语句太长而没有截全）。第443行使用名字引用生成器实例referenceCreator生成控制流图节点中的所有名字引用，第444行到第459行输出这些名字引用的信息（同样有些语句太长而没有截全）。

总的来说，目前控制流图基于抽象语法树生成，因此需要找到对应方法定义所属的编译单元及其抽象语法树根节点，并且使用类NameTableASTBridge找到对应的方法声明（抽象语法树）节点，以及获得控制流图节点中的名字定义，而对于名字引用，则需要使用类NameReferenceCreator的方法生成。

注意，在使用源代码文件集类SourceCodeFileSet的实例parser找编译单元的抽象语法树

根节点时会创建该编译单元的抽象语法树，在生成和使用控制流图的时候需要抽象语法树驻于内存，当遍历多个编译单元、多个方法时这可能会占用很多内存，因此构件使用者最好使用类SourceCodeFileSet的方法releaseAST()和releaseFileContent() 方法释放抽象语法树和源代码文件内容所占用的内存。

6.7 更为复杂的名字表使用简介

程序包nameTable中还包含两个编译单元文件TestNameTable1.java和TestNameTable2.java,这两个文件的代码实现了更为复杂的名字表使用功能。编译单元文件TestNameTable1.java先打印出详细类型定义中的方法，然后在该方法下打印出所有调用该方法的方法，图6.9给出这个程序的输出结果示例。

```

1 In type: nameTable.NameTableManager
2     References to method: nameTable.NameTableManager.NameTableManager
3         In type: nameTable.creator.NameTableCreator
4             Method: createNameTableManager@86:1
5                 Reference new NameTableManager(codeFileSet,systemScope), location 93:27
6             Method: createNameTableManager@101:1
7                 Reference new NameTableManager(codeFileSet,systemScope), location 108:27
8             Method: createNameTableManager@116:1
9                 Reference new NameTableManager(codeFileSet,systemScope), location 123:27
10            Method: createNameTableManager@130:1
11                Reference new NameTableManager(codeFileSet,systemScope), location 136:27
12            References to method: nameTable.NameTableManager.getSystemScope
13                In type: analyzer.qualitas.ClassMetricCollector
14                    Method: modifyQualitasDetailedTypeMeasure@123:1
15                        Reference manager.getSystemScope(), location 144:26
16                    Method: collectQualitasDetailedTypeMeasure@290:1
17                        Reference manager.getSystemScope(), location 314:26
18                    Method: collectQualitasDetailedTypeMeasure@354:1
19                        Reference manager.getSystemScope(), location 367:26
20                In type: analyzer.qualitas.CompilationUnitMetricCollector
21                    Method: collectQualitasCompilationUnitMeasure@63:1
22                        Reference manager.getSystemScope(), location 79:26

```

图 6.9 程序TestNameTable1.java的输出

图6.9的第一层（第1行）给出类（的全名），第二层缩进（例如第2行、第12行）给出这个类下面的方法，第三层缩进（例如第3行、第13行、第20行）给出调用这个方法的类，第四层缩进给出第三层缩进给出的类的那个方法调用第二层缩进给出的方法，第五层缩进给出具体调用的位置，例如图6.9的第5行表明，在类NameTableCreator的方法createNameTableManager()的93行27列由于创建类NameTableManager的对象实例而调用了类NameTableManager的构造方法（第二层缩进给出的是构造方法）。

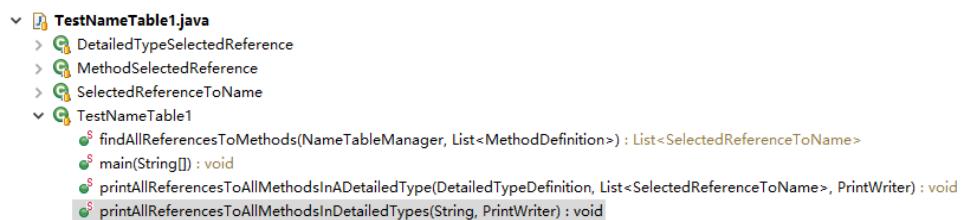


图 6.10 编译单元TestNameTable1.java的成员

图6.10给出编译单元TestNameTable1.java中的成员。为实现上述功能，类SelectedReferenceToName存放一个名字定义，以及一个列表，其元素是类DetailedTypeSelectedReference的实例，后者又有三个字段（下面所说的引用都是指引用该名字定义的引用）：

(1) 字段`type`是详细类型定义(类`DetailedTypeDefinition`)的实例,表明这个类所存放的引用都是该详细类型定义中的引用。

(2) 列表`methodReferenceList`,其元素是类`MethodSelectedReference`的实例,给出在这个详细类型定义的方法中出现的引用。类`MethodSelectedReference`有两个字段,其中`method`是一个方法定义,属于该详细类型定义的方法定义,另外一个是元素类型为`NameReference`的列表,给出这个方法定义中引用上面所说的那个名字定义的引用。

(3) 列表`otherReferenceList`,其元素是类`NameReference`的实例,给出这个详细类型定义的其他地方(不在方法体)中的引用。

也即类`SelectedReferenceToName`存储的是图6.9给出的输出结果示例中第三层缩进(例如第3行)到第五层缩进(例如第5行)的内容(即分方法、类给出调用第二层缩进给出的方法的名字引用)。

编译单元`TestNameTable1.java`的静态方法`findAllReferencesToMethds()`给出一个详细类型定义中的所有方法定义构成的列表,然后生成一个元素类型为`SelectedReferenceToName`的列表,也即其中每个元素分方法、类给出调用该详细类型定义的一个方法定义的所有引用。静态方法`printAllReferencesToAllMethodsInADetailedType()`则打印一个详细类型定义的所有方法的`SelectedReferenceToName`列表(由方法`findAllReferencesToMethds()`得到)。静态方法`printAllReferencesToAllMethodsInDetailedTypes()`则指定源代码文件集的起始路径,生成该源代码文件集的名字表,使用名字表管理器、名字表访问器和过滤器找到满足条件的一些详细类型定义,然后调用方法`printAllReferencesToAllMethodsInADetailedType()`分方法、类打印调用该这些详细类型定义的每个方法的所有引用。

这里不再详细给出编译单元`TestNameTable1.java`中的方法的具体实现代码,构件使用者可结合这个程序的输出结果,及其源代码了解名字表构件的复杂使用方式。

```

1 \begin{tabular}{c}
2 \hline \hei{ 描述} \\
3 \hline \hei{ 类\pname{TypeDefinition}本身声明的字段} \\
4 \hline \pname{isInterface} : \pname{boolean} \\
5 \hline \pname{isPackageMember} : \pname{boolean} \\
6 \hline \hei{ 继承自类\pname{NameDefinition}的字段} \\
7 \hline \pname{fullQualifiedName} : \pname{String} \\
8 \hline \pname{location} : \pname{SourceCodeLocation} \\
9 \hline \pname{scope} : \pname{NameScope} \\
10 \hline \pname{simpleName} : \pname{String} \\
11 \hline \hei{ 类\pname{TypeDefinition}本身声明的方法} \\
12 \hline \pname{matchSubtypeRelationsOfPrimitiveTypes(String, String) : boolean} \\
13 \hline \pname{TypeDefinition(String, String, SourceCodeLocation, NameScope) : Constructor} \\
14 \hline \pname{getDefinitionKind() : NameDefinitionKind}~[重定义类\pname{NameDefinition}的方法] \\
15 \hline \pname{getEnclosingPackage() : PackageDefinition} \\
16 \hline \pname{getSuperClassDefinition() : TypeDefinition} \\
17 \hline \pname{getSuperList() : List<TypeReference>} \\
18 \hline \pname{isInterface() : boolean} \\
19 \hline \pname{isPackageMember() : boolean} \\
20 \hline \pname{isPublic() : boolean} \\
21 \hline \pname{isSubtypeOf(TypeDefinition) : boolean} \\
22 \hline \pname{resolve(NameReference) : boolean} \\
23 \hline \pname{setInterface(boolean) : void} \\
24 \hline \pname{setPackageMember(boolean) : void} \\
25 \hline \hei{ 继承自类\pname{NameDefinition}的方法(不含已被重定义方法) } \\
26 \hline \pname{compareTo(NameDefinition) : int} \\

```

图 6.11 程序`TestNameTable2.java`的输出

编译单元`TestNameTable2.java`打印一个详细类型定义的成员信息。图6.11给出它的一个输出实例,给出了类`TypeDefinition`的成员信息(实际上,前面表5.10 和表5.11就是基于这个输出得到的。前面讨论名字表构件实现中以表格形式给出类的字段和方法都是使用这个程序的输出信

息)。这个程序输出的是一个可用于 \LaTeX 文件中的表格, $\text{\begin\{tabular\}}$ 和 $\text{\end\{tabular\}}$ 给出的是 \LaTeX 文件中的一个表格(环境), 以 \backslash 开头的串都是 \LaTeX 的命令。

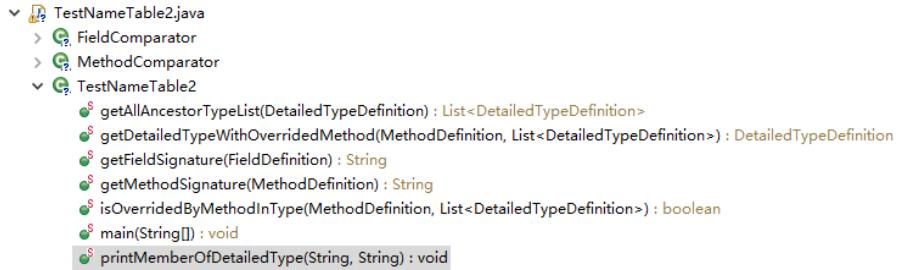


图 6.12 编译单元 `TestNameTable2.java` 的成员

图 6.12 给出编译单元 `TestNameTable2.java` 的成员。实际上, 编译单元 `TestNameTable2.java` 的输出中最关键的是要给出这个详细类型定义中的继承成员信息, 以及其中声明的方法是否重定义祖先类的方法, 重定义哪个方法。为实现这些功能, 编译单元 `TestNameTable2.java` 的静态方法 `getAllAncestorTypeList()` 给出一个详细类型定义的所有祖先类(也是详细类型定义)构成的列表。

静态方法 `getDetailedTypeWithOverridedMethod()` 则给定一个方法定义和一个详细类型定义列表, 该详细类型定义类别中的元素被认为是该方法定义所在类的祖先类(实际上就是通过调用方法 `getAllAncestorTypeList()` 得到的列表), 在这个详细定义列表中查找是否有一个类, 这个方法定义重定义了这个类的一个方法。我们假定该详细定义列表中的类按照继承层次排列, 离参数中指定的方法定义所属类越近的类排得越靠前(实际上方法 `getAllAncestorTypeList()` 的实现会保证这一点)。

静态方法 `isOverridedByMethodInType()` 则判断给定的方法定义是否为给定的详细类型定义列表中的某个类的方法所重定义, 这个详细类型定义列表被假定由该方法定义所在的类的后代类构成。

静态方法 `printMemberOfDetailedType()` 给定源代码文件集的起始路径和输出文件名, 生成该源代码文件集的名字表, 然后使用名字表管理器、访问器和过滤器查找一个指定的详细类型定义, 然后先打印该详细类型定义自己声明的成员信息, 然后打印它的祖先类的成员信息, 这样在打印过程中可收集一个类的后代类信息(即在打印这个类之前已经打印过的类就是它的后代), 这个信息可用于静态方法 `isOverridedByMethodInType()` 判断一个方法定义是否已经被重定义(已经被重定义不再打印)。

简单地说, 方法 `getDetailedTypeWithOverridedMethod()` 可实现在输出一个方法时确定这个方法是否重定义了某个祖先类的方法。而方法 `getAllAncestorTypeList()` 可得到一个类的所有祖先类, 因此可打印这个类从祖先类继承的成员, 在打印祖先类的成员时通过 `isOverridedByMethodInType()` 方法判断这个成员是否已经被后代类重定义(已经被重定义不再打印)。

编译单元 `TestNameTable2.java` 的方法 `getFieldSignature()` 和 `getMethodSignature()` 分别给出字段定义和方法定义声明时的字符串, 以便简洁地输出一个字段定义和方法定义的信息。

编译单元 `TestNameTable2.java` 还声明两个类 `FieldComparator` 和 `MethodComparator`, 它们用来对一个详细类型定义中的字段列表和方法列表进行排序, 将静态成员排在非静态成员的前面, 静态成员和非静态成员之间则按照它们的简单名顺序进行排序(在生成名字表时, 详细类型定义的字

段列表和方法列表按照在源代码中出现的顺序排序)。

同样，这里不再详细给出编译单元TestNameTable1.java中的方法的具体实现代码，构件使用者可结合这个程序的输出结果，及其源代码了解名字表构件的复杂使用方式。

6.8 名字表构件使用总结

总的来说，名字表构件中代表名字定义、名字作用域和名字引用的类给出这些名字表实体的存储，类NameTableCreator、类NameDefinitionCreator和类NameReferenceCreator用于名字表实体的生成，类NameTableManager作为名字表的入口，构件使用者主要通过这个类的实例，然后使用名字表访问器、过滤器可实现名字表访问的基本功能，在需要建立与抽象语法节点之间关联时则要借助类NameTableASTBridge的方法。至于更复杂的名字表访问，则需要构件使用者根据需要基于已有的名字表访问方式自己实现。

第七章 其他程序包简介

7.1 概述

目前JAnalyzer的程序包如图7.1所示，其中程序包sourceCodeAST基于Eclipse JDT构件Java源文件的抽象语法树，并进行管理，而程序包nameTable及其子程序包为以某个目录为项目根目录下的所有Java源文件构成的软件项目构建名字表，以确定软件项目中的名字定义和名字引用的绑定（解析）。

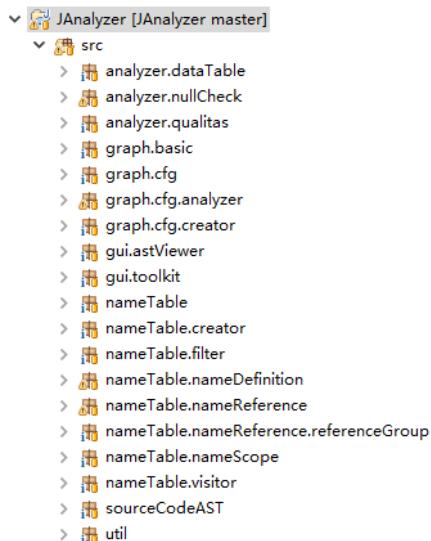


图 7.1 JAnalyzer 目前包含的程序包

程序包sourceCodeAST以及nameTable（及其子程序包）已经在前面做了详细的介绍，在其他的程序包中：

- (1) 程序包util提供了一个堆栈类Stack，方便程序调试类Debug，比较两个目录下Java源文件版本类SystemVersionComparator和对一个目录下Java源文件进行整理的类SourceFilePackingManager。这些都是一些不太适合放到其他程序包的通用工具类；
- (2) 程序包gui及其子程序包主要用来放与图形用户界面有关的程序包和类。目前子程序包gui.toolkit给出了一个类MainFrame，实现了一些基于SWT启动图形用户主界面的功能，而另外一个类FileChooserAndOpener封装了一个文件选择器，可实现通过图形用户界面选择并打开一个文件的功能。子程序包astViewer则实现了对一个Java源文件的抽象语法树和控制流图的文本化浏览功能，运行其中的TestASTViewer的主函数可得到一个图形化界面，在该界面中选择并打开一

个Java源代码，生成它的抽象语法树和控制流图，不过都是以文本形式进行展示；

(3) 程序包graph及其子程序包主要用来实现与源程序分析相关的图的生成。目前子程序包basic给出了图的一些基础类，而子程序包cfg及其子程序包实现了程序控制流图的生成及一些基于控制流图的数据流分析功能。后面会对控制流图的生成、数据流分析等功能的实现和使用做更多的介绍；

(4) 程序包analyzer及其子程序包主要用来实现一些更复杂的源程序分析功能。目前子程序包dataTable实现了一个简单的二维表数据管理功能，可以方便我们将源程序分析的结果写入到以制表符分隔的文本形式的二维表，也可从这种形式的二维表读入内容，并管理它的行、列和单元格的内容。子程序包qualitas是针对我们下载的一个Java源程序库Qualitas，提供了分析这个库的Java源文件的元信息及目录信息的一些功能。Qualitas是一些软件工程研究者收集的Java源程序库，包含了多个Java开源项目及其演化版本，被多个软件工程研究项目所使用。子程序包nullCheck实现了与空值检测相关的一些功能，这些功能还是实现与完善之中。

由于时间关系，下面暂时只给出程序包graph及其子程序包的介绍，以便大家进一步了解JAnalyzer中控制流图的生成和使用。

7.2 控制流图的生成与使用

程序包graph及其子程序包主要用来实现与Java源程序分析相关的视图生成，目前有如下子程序包：

- (1) 子程序包basic给出了图的一些基础类和接口，包括类AbstractGraph、接口GraphNode、接口GraphEdge和工具类GraphUtil；
- (2) 子程序包graph.cfg给出用于存储控制流图(control flow graph, 简称CFG)的类；
- (4) 子程序包graph.cfg.creator给出生成控制流图的类；
- (3) 子程序包graph.cfg.analyzer给出基于控制流图实现一些简单的数据流分析的类。

7.2.1 程序视图基础类

程序包graph.basic给出了实现程序视图的一些基础类，包括接口GraphNode、接口GraphEdge、类AbstractGraph和工具类GraphUtil。

接口GraphNode要求每个作为图的顶点(节点)的对象都需要提供唯一标识(id)、标签(label)和描述(descript)，其中标识需要在整个图中唯一，标签通常用于显示，描述是对这个顶点的详细描述。

接口GraphEdge要求每个作为图的边的对象都需要提供起始顶点(startNode)、终止顶点(endNode)、是否是有向边(isDirected)、标签(label)和描述(descript)。

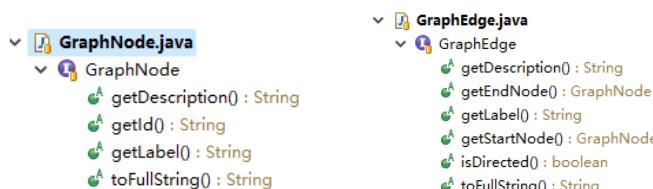


图 7.2 接口GraphNode和接口GraphEdge

图7.2给出了接口GraphNode和接口GraphEdge声明的方法。这两个接口中的方法toFullString()都要求对象在实现方法toString()之外给出一种将整个对象转换为更详细描述的字符串。

类AbstractGraph作为任何程序视图的基类，提供了一些与图相关的基本方法。类AbstractGraph目前的实现保存了两个链表nodes和edges，类型分别是List<GraphNode>和List<GraphEdge>，分别记录图的所有顶点和所有边。类AbstractGraph还有一个字段id作为图的标识，目前这个标识没有要求唯一，只是一个任意的字符串。

表7.1给出了类AbstractGraph的方法描述，这些方法都比较简单。注意，这个类目前的实现中没有对边是否是有向边做任何约束，边的方向取决于类的使用者的解释。**目前的实现中环（两个端点都是同一顶点的边）在计算顶点的度数中还没有被计算为2次。**

类GraphUtil是一个工具类以提供静态方法实现对图的复杂操作。目前实现了多个方法extractSubGraph...()来以某个顶点为中心提取子图。目前的实现中方法simplyWriteToDotFile()可将图写成可由GraphViz这个软件进行可视化的.dot文件。实际上，生成的.dot文件很简单，只需指定图的顶点和边即可，GraphViz软件会自动实现图的顶点和边的布局从而进行图的可视化绘制。GraphViz软件的自动布局效果比较好，绘制的图形效果也比较好。类GraphUtil的方法getLegalToken()方法将一个字符串转换为合乎GraphViz语法的图的顶点或边的标识，因为目前类AbstractGraph中的顶点或边的标识、标签等可能包含GraphViz中不能使用的字符。目前的方法getConnectedComponentNumber()可基于一个图的邻接矩阵调用方法depthSearchFirstTravel()进行深度优先搜索而计算图的连通分支数。

7.2.2 控制流图的存储

程序包graph.cfg中的类用于存储生成的控制流图。实际上，控制流图的使用者主要是在生成控制流图之后，利用这个程序包中的类提供的方法对控制流图进行访问。图7.3给出了这个程序包中的类。

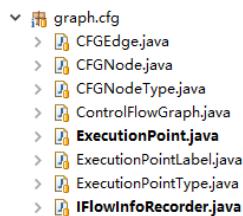


图 7.3 程序包graph.cfg中的类

在程序包graph.cfg中，CFGNodeType是一个枚举，标识控制流图可能的节点类型。在我们的设想中，控制流图的节点可以是可执行点(N_EXECUTION_POINT)，或基本块(N_BASIC_BLOCK)，或者是一个子控制流图(N_SUM_CFG)，不过目前生成的控制流图的节点都只是可执行点。CFGNode是扩展接口GraphNode的一个接口，用于表示控制流图的节点，主要是声明了一些方法来判断一个控制流图节点是否是起始节点(isStart())、正常终止节点(isNormalEnd())、异常终止节点(isAbnormalEnd())，谓词节点(isPredicate())，或是虚拟节点(isVirtual())。

图7.4给出了JAnalyzer生成的一个方法（方法名是hasTypeReference()）的控制流图示例：首先每个方法都有一个起始节点作为方法的入口、一个正常终止节点作为方法正常返回的出口和一个异常终止节点作为方法抛出异常的出口。图7.4中使用八边形框住的就是这些节点（这个图中没有异

表 7.1 类AbstractGraph方法的描述

<code>addEdge(GraphEdge) : void</code>	往图中增加一条边
<code>addNode(GraphNode) : void</code>	往图中增加一个顶点
<code>adjacentFromNode(GraphNode) : List<GraphNode></code>	给出以参数指定顶点为起点的所有邻接顶点列表
<code>adjacentNodes(GraphNode) : List<GraphNode></code>	给出所有与参数指定顶点相邻的顶点邻标
<code>adjacentToNode(GraphNode) : List<GraphNode></code>	给出以参数指定顶点为终点的所有邻接顶点列表
<code>findById(String) : GraphNode</code>	基于顶点的标识从图的顶点列表中查找顶点
<code>findByLabel(String) : GraphNode</code>	基于顶点的标签从图的顶点列表中查找顶点
<code>getAdjacentMatrix() : int[] []</code>	得到图的邻接矩阵，即矩阵的行与列都代表顶点，从某个顶点到某个顶点有边为1，否则为0
<code>getAllNodes() : List<GraphNode></code>	返回图的所有顶点构成的列表
<code>getDegree(GraphNode) : int</code>	返回指定顶点的度数，即所有与该顶点相连的边数
<code>getEdges() : List<GraphEdge></code>	返回图的所有边构成的列表
<code>getGeneratedSubGraph(List<GraphNode>) : AbstractGraph</code>	返回以参数指定的顶点列表为顶点集的生成子图，即生成子图的顶点集为指定顶点集，边集由两个端点都在指定顶点集中的边构成
<code>getId() : String</code>	返回图的标识
<code>getInDegree(GraphNode) : int</code>	返回指定顶点的入度
<code>getOutDegree(GraphNode) : int</code>	返回指定顶点的出度
<code>hasEdge(GraphEdge) : boolean</code>	判断图中是否有指定的边
<code>hasEdge(GraphNode, GraphNode) : boolean</code>	判断图中是否有从（前一参数）指定的起始顶点到（后一参数）指定的顶点的边
<code>hasNode(GraphNode) : boolean</code>	判断图中是否有指定的顶点
<code>setAllEdges(ArrayList<GraphEdge>) : void</code>	将参数中指定的边列表进行拷贝而设置图的所有边构成的列表
<code>setAllNodes(ArrayList<GraphNode>) : void</code>	将参数中指定的顶点列表进行拷贝而设置图的所有顶点构成的列表
<code>setEdges(List<GraphEdge>) : void</code>	设置图的所有边构成的列表（不会拷贝列表中的边实例）
<code>setNodes(List<GraphNode>) : void</code>	设置图的所有顶点构成的列表（不会拷贝列表中的顶点实例）
<code>toFullString() : String</code>	返回图的一个字符串表示，其中的节点信息更为详细
<code>toString() : String</code>	返回图的一个字符串表示

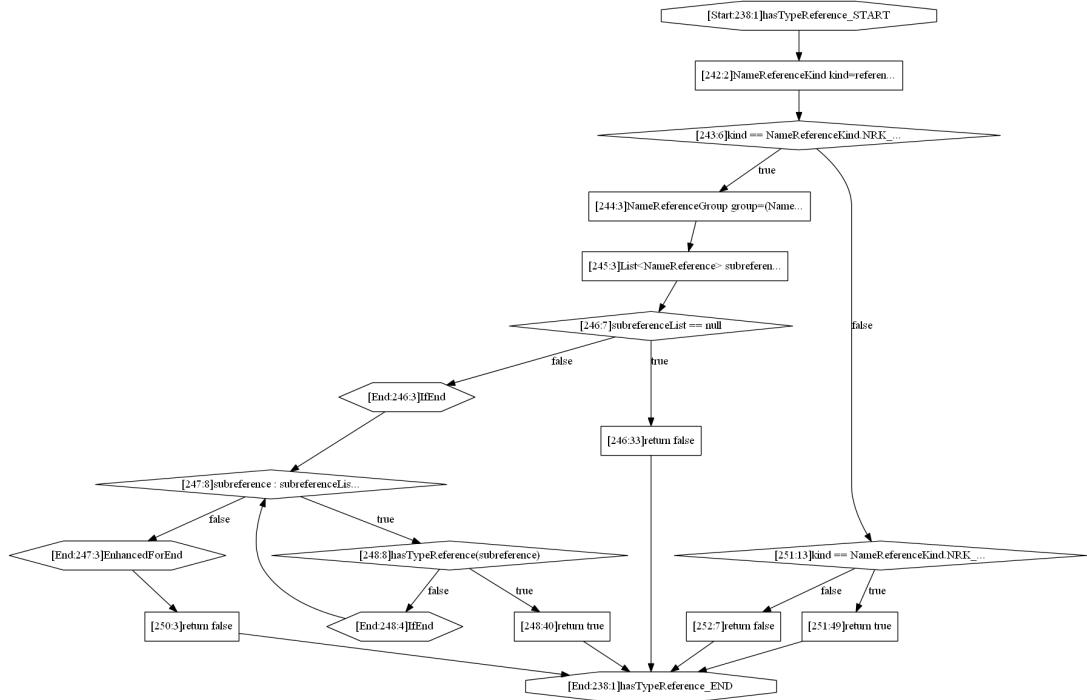


图 7.4 JAnalyzer 生成的控制流图示例

常终止节点)。为了简化控制流图的生成,我们针对分支、循环等复杂语句构建了代表这些语句入口和出口的虚拟节点,图7.4中使用六边形框住的节点都是虚拟节点。图7.4中四边菱形框则代表谓词节点,而长方形框则是真正包含可执行语句的控制流图节点。

类CFGEdge是实现接口GraphEdge的类(而不是接口),它以实现接口CFGNode的类的对象为顶点,并且给出了接口GraphEdge所声明的方法的简单实现。

实际上,目前所生成的控制流图的节点都是可执行点,这用类ExecutionPoint的对象实例存储。类ExecutionPoint存储该可执行点所对应的抽象语法树节点(astNode),以及作为图形顶点的标识(id)、标签(label)和描述(description),可执行点的类型(type)、这个可执行点所对应语句(更准确地说抽象语法树节点)在源代码的起始位置(startLocation)和终止位置(endLocation),以及这个可执行点可能存储的数据流信息(recorder)。

枚举ExecutionPointType定义了可执行点的类型,类ExecutionPoint中的type字段存放的就是这个枚举的值。枚举ExecutionPointLabel则给出了一些字符串常量用于作为可执行点的标签(存放在类ExecutionPoint中的label字段)。通过类ExecutionPoint的type和label字段可准确地确定一个可执行点是否是起始节点、终止节点、虚拟节点、谓词节点或普通节点,以及所对应的语句。**目前所有可执行点都是通过程序包graph.cfg.creator 的类ExecutionPointFactory来创建的,通过阅读这个类的实现源代码很容易确定不同抽象语法树所创建的可执行点及其类型和标签,这里由于时间关系暂时不做详细说明。**

类ExecutionPoint的字段astNode存放可执行点对应的抽象语法树节点信息,startLocation和endLocation作为SourceCodeLocation的实例给出该抽象语法树节点在源代码中的起始位置和终止位置信息。这里需要注意两点:(1)对应循环、分支等复杂语句的虚拟节点会存放整个语句所对应的抽象语法树节点,而循环体(分支块)中的语句所对应的节点还会存放相应的抽象语

法树节点（这个抽象语法树节点当然是虚拟节点中所存放抽象语法树节点的子节点）；(2) 由于控制流图的可执行节点存放了抽象语法树节点，因此控制流图可能占用大量内存。目前的控制流图只能针对一个方法生成（即没有生成方法间的控制流图），因此建议指向一个方法的控制流图应尽量是局部变量，而且在处理完一个方法，或者一个编译单元的控制流图之后应调用类SourceCodeFileSet的releaseAST()和releaseFileContent()方法将类SourceCodeFileSet的实例中指向相应编译单元的抽象语法树根和文件内容的对象引用置为null，以便JVM的垃圾回收机制尽快回收它们所占用的内存，降低内存占用率，以便能分析处理源代码规模大的软件项目。

类ExecutionPoint的字段recorder的类型是接口IFlowInfoRecorder，这个接口没有定义任何方法，只是表明实现该接口的类的实例用于存储基于控制流图的数据流分析信息。在基于控制流图进行数据流分析时，往往要将一些信息存放在控制流图的节点中，以便对数据流分析方程进行迭代求解。类ExecutionPoint的字段recorder可用于存储数据流分析信息。具体要存储什么信息，以及如何使用这些信息可通过实现接口IFlowInfoRecorder来完成，类ExecutionPoint只是一个提供存储和读取数据流分析信息的载体。

类ExecutionPoint的对象实例通过字段id和label一起来进行区分，也即这个类重定义了方法equals()和hashCode()基于这两个字段判断两个对象实例是否相等以及生成哈希值。

类ExecutionPoint提供的方法主要是对于它的字段的读取和设置，以及实现接口CFGNode所要求实现的确定可执行节点作为控制流图节点类型（虚拟节点、起始节点、终止节点或谓词节点等）的方法。

表 7.2 类ControlFlowGraph字段的描述

字段	类型	描述
abnormalEndNode	CFGNode	控制流图的异常终止节点
className	String	控制流图对应方法所属类的类名
description	String	对控制流图的描述串
endNode	CFGNode	控制流图的正常终止节点
factory	ExecutionPointFactory	用于生成控制流图节点（可执行点）的工厂对象
label	String	控制流图的标签
method	MethodDeclaration	控制流图对应方法的抽象语法树节点
methodName	String	控制流图对应方法的方法名
startNode	CFGNode	控制流图的起始节点
unitRecorder	CompilationUnitRecorder	控制流图对应方法所在的编译单元记录
继承自类AbstractGraph的字段		
edges, id, nodes		

类ControlFlowGraph的实例代表某个方法的控制流图，表7.2给出了这个类所声明的字段，其中最重要的字段是method，它存储控制流图对应方法的抽象语法树节点，控制流图的生成是通过该节点遍历方法体中的语句而进行创建。字段unitRecorder则给出相应方法所在的编译单元的信息，包括编译单元的全名和编译单元的抽象语法树根节点，这个信息主要用于生成方法体的语句的源代码位置。Eclipse JDT的抽象语法树节点本身不存储这个节点对应的源代码在源文件中的行列位置，而是存储以字节数为度量单位的偏移量，需要通过编译单元抽象语法树根节点将此偏移量转换为源文件行列位置。字段factory是用于生成控制流图节点（对象实例），目前都是可执行点的工厂对象。字段className和methodName给出相应的类名和方法名，但实际上类ControlFlowGraph并没有

严格检查字段**methodName**与**method**的一致性，这两个串只是示意性的，主要用于生成起始节点和终止节点的标签。

注意类**ControlFlowGraph**继承了类**AbstractGraph**，并实现了接口**CFGNode**，以便我们以后可将一个控制流图本身整体作为一个控制流图节点（利用在方法调用时，将被调用的方法的控制流图作为调用者的控制流图的一个节点，从而可展示过程间控制流图）。

类**ControlFlowGraph**的方法都比较简单，表7.3和从7.4给出了它所声明的方法及简介。实际上，我们目前通常是在生成控制流图之后，获得控制流图的所有顶点或/和边的列表，然后对控制流图进行分析。类**ControlFlowGraph**的方法**simplyWriteToDotFile()**可将控制流图写成.**.dot**文件以便使用**GraphViz**软件进行可视化展示，图7.4就是使用这个方法得到的.**.dot**文件，然后使用**GraphViz**得到的控制流图图片。

表 7.3 类**ControlFlowGraph**方法的描述

<code>getAbnormalEndNode() : CFGNode</code>	返回控制流图的异常终止节点
<code>getCFGNodeType() : CFGNodeType</code> [实现接口 CFGNode 的方法]	返回控制流图整体作为节点的类型，实际上是返回 N_SUB_CFG
<code>getClassName() : String</code>	返回控制流图对应方法所属类的类名
<code>getCompilationUnitRecorder() : CompilationUnitRecorder</code>	返回控制流图对应方法所在的编译单元信息
<code>getCompilationUnitRoot() : CompilationUnit</code>	返回控制流图对应方法所在的编译单元的抽象语法树根节点
<code>getDescription() : String</code> [实现接口 GraphNode 的方法]	返回对控制流图进行描述的串
<code>getEndNode() : CFGNode</code>	返回控制流图的正常终止节点
<code>getExecutionPointFactory() : ExecutionPointFactory</code>	返回用于生成控制流图节点的工厂对象
<code>getFileUnitName() : String</code>	返回控制流图对应方法所在编译单元的全名
<code>getId() : String</code> [重定义类 AbstractGraph 的方法]	返回控制流图本身作为图形节点的标识
<code>getLabel() : String</code> [实现接口 GraphNode 的方法]	返回控制流图本身作为图形节点的标签
<code>getMethod() : MethodDeclaration</code>	返回控制流图对应方法的抽象语法树节点
<code>getMethodName() : String</code>	返回控制流图对应方法的方法名
<code>getStartNode() : CFGNode</code>	返回空值流图的起始节点

7.2.3 控制流图的生成

生成控制流图的类都在程序包**graph.cfg.creator**之中，其中重要的类或接口包括：

- (1) 可执行点工厂类**ExecutionPointFactory**，它提供了一系列的**creat...()**方法，这些方法

表 7.4 类ControlFlowGraph方法的描述（续）

<code>isAbnormalEnd() : boolean [实现接口CFGNode的方法]</code>	
返回 <code>false</code>	
<code>isNormalEnd() : boolean [实现接口CFGNode的方法]</code>	
返回 <code>false</code>	
<code>isPredicate() : boolean [实现接口CFGNode的方法]</code>	
返回 <code>false</code>	
<code>isStart() : boolean [实现接口CFGNode的方法]</code>	
返回 <code>false</code>	
<code>isVirtual() : boolean [实现接口CFGNode的方法]</code>	
返回 <code>false</code>	
<code>setAndAddAbnormalEndNode(CFGNode) : void</code>	设置控制流图的异常终止节点，并将其添加到节点列表中
<code>setAndAddEndNode(CFGNode) : void</code>	设置控制流图的正常终止节点，并将其添加到节点列表中
<code>setAndAddStartNode(CFGNode) : void</code>	设置控制流图的起始节点，并将其添加到节点列表中
<code>setCompilationUnitRecorder(String, CompilationUnit) : void</code>	设置控制流图对应方法所在的编译单元信息
<code>setCompilationUnitRecorder(CompilationUnitRecorder) : void</code>	设置控制流图对应方法所在的编译单元信息
<code>setDescription(String) : void</code>	设置描述控制流图的描述串
<code>setExecutionPointFactory(ExecutionPointFactory) : void</code>	设置用于生成控制流图节点的工厂对象
<code>setLabel(String) : void</code>	设置控制流图的标签
<code>setMethod(String, String, MethodDeclaration) : void</code>	设置控制流图对应方法的类名、方法名和方法对应的抽象语法树节点
<code>simplyWriteToDotFile(PrintWriter) : void</code>	将控制流图写成.dot文件以便使用GraphViz软件进行可视化展示
继承自类AbstractGraph的方法（不含已被重定义方法）：参见表7.1	
<code>addEdge(GraphEdge) : void; addNode(GraphNode) : void;</code>	
<code>adjacentFromNode(GraphNode) : List<GraphNode>; adjacentToNode(GraphNode) : List<GraphNode>;</code>	
<code>findById(String) : GraphNode; findByLabel(String) : GraphNode; getAdjacentMatrix() : int[][];</code>	
<code>getAllNodes() : List<GraphNode>; getDegree(GraphNode) : int; getEdges() : List<GraphEdge>;</code>	
<code>getGeneratedSubGraph(List<GraphNode>) : AbstractGraph; getInDegree(GraphNode) : int;</code>	
<code>getOutDegree(GraphNode) : int; hasEdge(GraphEdge) : boolean;</code>	
<code>hasEdge(GraphNode, GraphNode) : boolean; hasNode(GraphNode) : boolean;</code>	
<code>setAllEdges(ArrayList<GraphEdge>) : void; setAllNodes(ArrayList<GraphNode>) : void</code>	
<code>setEdges(List<GraphEdge>) : void; setNodes(List<GraphNode>) : void</code>	
<code>toFullString() : String; toString() : String</code>	

针对不同的语句（抽象语法树节点类Statement的子类对象）创建不同的可执行点作为控制流图的，包括创建虚拟节点、谓词节点、普通节点等；

(2) 抽象类StatementCFGCreator，它声明了抽象方法creat(ControlFlowGraph, Statement, List<PossiblePrecedeNode>, String)，其含义是在指定的控制流图中为指定语句创建控制流图节点，该节点的可能前驱节点在类型为List<PossiblePrecedeNode>列表中。这个方法返回一个同样类型的列表作为后续控制流图节点的可能前驱节点列表。这个方法中的字符串类型参数指明正在创建控制流图节点的语句可能存在的标号(label)。实际上，Java程序中只有循环语句前面的标号有用。在创建节点时记录整个循环语句的标号以便在循环体中出现带标号的break和continue时能得到正确的控制流；

(3) 类PossiblePrecedeNode，它用于表示一个控制流图节点可能的前驱节点，它包含字段node、reason和label。这里node的类型是CFGNode记录可能的前驱节点，而reason的类型是枚举类型PossiblePrecedeReasonType的值，用于表示所记录的前驱节点与后继节点之间的控制流关系是顺序还是由于break, continue或返回、抛出异常等产生的控制流关系（即控制流图中的边）。label是一个字符串，用于表示相应控制流图边的标签，这些标签可能用于描述分支语句的分支（即是条件表达式的true分支还是false分支），以及break, continue的目标语句标号等；

(4) 枚举PossiblePrecedeReasonType用于描述控制流图中前驱节点和其后继节点之间的控制流关系，包括枚举常量(PPR_SEQUENCE)、PPR_BREAK、PPR_CONTINUE、PPR_RETURN和PPR_THROW，分别表示是由于顺序执行还是由于break, continue或返回、抛出异常等产生的控制流关系；

(5) 类CFGCreator是生成控制流图的主类。使用者只需利用这个类进行控制流图的生成即可。这个类即可当做实例类使用，也可当成静态类使用。当成实例类使用时可利用编译单元的全名和抽象语法树根节点创建这个类的对象，然后调用方法create(MethodDeclaration, String)生成指定抽象语法树节点对应方法的控制流图，其中的字符串类型参数给出该方法的方法名，也可调用create()方法生成这个编译单元中所有方法的控制流图列表。这个类当做静态类使用时，则可调用其静态方法create(NameTableManager, CompilationUnitRecorder, MethodDefinition)，创建某个指定方法的控制流图，这时需要给出该方法所在的名字表管理器和编译单元信息，或者调用其静态方法create(NameTableManager, MethodDefinition)生成指定方法的控制流图，这时编译单元信息则会在这个方法内部通过方法定义来获得。类CFGCreator还提供了由可执行点（准确地说根据可执行点的源代码起始位置和终止位置）匹配相应抽象语法树节点的功能，这是为了以后将控制流图持久化时能恢复节点中的抽象语法树节点信息（因为对控制流图的持久化不可能将节点中的抽象语法树节点也写入外存）。

(6) 一系列的以...CFGCreator命名的控制流图（节点）生成类，这些类主要是针对不同的语句而设置，例如IfCFGCreator用于为if语句创建控制流图，而SimpleStatementCFGCreator为所有的简单语句（不含复杂的控制流结构的语句）创建控制流图（节点）等等。这些类都继承抽象类StatementCFGCreator，给出其中声明的create()方法的具体实现，针对不同的语句创建具有不同控制流结构的控制流图；

(7) 类StatementCFGCreatorFactory创建了不同控制流图生成类...CFGCreator()的实例，并使用静态方法getCreator()基于不同的语句返回相应的控制流图生成类实例。因此控制流图生成类的实例是可重用的，即在生成控制流图时，针对if语句总是用同一个IfCFGCreator的实例，而不是为每个if语句临时创建这样一个实例。

(8) 类StatementCFGCreatorHelper是一个辅助类，主要用于生成控制流图节点标签时将抽象语法树节点转换为字符串，以及提供一些其他静态方法辅助控制流图的生成。

实际上，当需要生成控制流图的时候只需调用类CFGCreator的creat()方法。最初我们在生成控制流图的时候是不需要名字表，直接基于抽象语法树节点，即类MethodDeclaration的实例生成，后来在类CFGCreator中增加了静态的creat()方法，它基于名字表中的方法定义，即类MethodDefinition的实例生成控制流图。在生成控制流图时要记录每个节点的起始源代码位置和终止源代码位置，因此需要编译单元全名和编译单元抽象语法树根节点信息。

类CFGCreator的create()方法（不管是实例方法还是静态方法）生成控制流图的主要步骤都是：

(1) 创建类ControlFlowGraph的实例作为最后要返回的结果，基于编译单元信息创建可执行点工厂类ExecutionPointFactory实例，并设置方法名、类名等一些用于标识、标签的信息；

(2) 使用可执行点工厂实例创建方法的整个控制流图的起始节点，将其加入到初始的可能前驱节点列表；

(3) 基于方法的抽象语法树根节点（类MethodDeclaration的实例）获得方法体（抽象语法树节点类Block的实例），使用控制流图生成器工厂类StatementCFGCreatorFactory获得相应的...CFGCreator类的实例，以初始的可能前驱节点（即整个方法的起始节点作为前驱节点）调用它的creat()方法；

(4) 每个...CFGCreator类的creat()都是使用可执行点工厂类ExecutionPointFactory的方法创建合适的可执行点（包括虚拟节点）。对于复杂语句则再使用类StatementCFGCreatorFactory获得相应的...CFGCreator类的实例，调用其create()方法进一步生成控制流图。在生成节点时主要处理前驱节点列表中的节点与当前生成的控制流图节点之间的边，并得到新的前驱节点列表为后续生成控制流图做准备。不同...CFGCreator类的create()方法的具体实现这里不再详细说明，需要了解控制流图生成的细节可阅读相应的源代码；

(5) 在为整个方法体生成控制流图节点之后，再生成一个正常终止节点和一个异常终止节点，用于处理最后得到的前驱节点列表中所有原因分别为PPR_RETURN和PPR_THROW的前驱节点，生成这些节点到正常终止节点或异常终止节点的边，从而完成方法的整个控制流图的生成。

可以看到，目前控制流图的生成不是通过遍历抽象语法树节点，而是使用工厂类针对不同的语句进行处理。我们尝试过使用遍历抽象语法树节点的方法实现控制流图的生成，但我们发现基于遍历的实现相当于将不同...CFGCreator的方法create()合并到一个Java源文件，并没有太多的好处，而且基于遍历的实现会有稍微低一些的时间效率，因此最后我们还是保留了目前的实现策略。

程序包graph.cfg.creator中的TestCFGCreator展示了如何使用类CFGCreator的成员方法create()生成控制流图，并测试了其中匹配抽象语法树节点方法的正确性，其中核心的测试方法是testMatchASTNode()，图7.5给出这个方法的方法体。

在图7.5，第60行基于一个路径创建了一个源代码文件集，第61到98行的循环测试了每个编译单元控制流图生成。第62行和第68行分别获得一个编译单元的全名和抽象语法树根节点，然后在第69行构造一个类CFGCreator的实例，第70行调用create()方法得到整个编译单元所有方法的控制流图列表。第71到95行的循环测试每个控制流图。第73行调用simplyWriteToDotFile()将控制流图写入到指定的文件以便使用GraphViz软件可视化。第77行调用类getAllNodes()可获得一个控制流图所有节点的列表，这个列表可能为空（因为一个方法可能是空方法体）。第79行到93行处理每个

```

60     SourceCodeFileSet parser = new SourceCodeFileSet(path);
61     for (SourceCodefile codeFile : parser) {
62         String fileName = parser.getFileUnitName(codeFile);
63         System.out.println("Scan file: " + fileName);
64         if (!codeFile.hasCreatedAST()) {
65             System.out.println("Can not create AST for code file " + fileName);
66             continue;
67         }
68         CompilationUnit root = codeFile.getASTRoot();
69         CFGCreator creator = new CFGCreator(fileName, root);
70         List<ControlFlowGraph> cfgs = creator.create();
71         for (ControlFlowGraph cfg : cfgs) {
72             try {
73                 cfg.simplyWriteToDotFile(output);
74             } catch (Exception exc) {
75                 exc.printStackTrace();
76             }
77             List<GraphNode> nodes = cfg.getAllNodes();
78             if (nodes != null) {
79                 for (GraphNode node : nodes) {
80                     ExecutionPoint point = (ExecutionPoint)node;
81
82                     ASTNode astNode = point.getAstNode();
83                     ASTNode matchedNode = creator.matchASTNode(point);
84                     if (astNode == matchedNode) {
85                         Debug.println("Matched AST node for execution point [" + point.getDescription() + "] at ["
86                     } else {
87                         Debug.println("DO NOT matched AST node for execution point [" + point.getDescription() + "] "
88                         if (astNode != null) Debug.println("\tAST Node from point: " + astNode.toString());
89                         else Debug.println("\tAST node is null!");
90                         if (matchedNode != null) output.println("\tAST node matched: " + matchedNode.toString());
91                         else Debug.println("\tMatched node is null!");
92                     }
93                 }
94             }
95         }
96         codeFile.releaseAST();
97         codeFile.releaseFileContent();
98     }

```

图 7.5 控制流图的生成及节点与抽象语法树的匹配

节点，第80行将其转换为可执行点，因为目前所有的控制流图节点都是可执行点，第82和第83行分别直接获得可执行点中的抽象语法树节点和通过匹配获得抽象语法树节点信息，并在第81行到92行将它们进行比较。第96行和第97行释放当前编译单元的文件内容和抽象语法树所占用的内存，以避免在处理大规模源代码时内存不足。

简单地来说，在提供编译单元全名、编译单元抽象语法树节点和方法的抽象语法节点信息之后就可通过类CFGCreator的create()方法生成控制流图，然后可获得控制流图的所有节点（和边）的信息。如果调用类CFGCreator的静态方法create()，则只需提供名字表管理器和方法定义，有时调用者可能已经找到该方法定义所属的编译单元信息，那么可提供编译单元信息以便重复查找。后面基于控制流图的数据流分析中我们将展示如何基于静态方法create()生成控制流图。

7.2.4 基于控制流图的数据流分析

多数数据流分析都要基于方法的控制流图进行迭代求解，因此我们将一些数据流分析功能的实现放在程序包graph.cfg.analyzer中。目前这个程序包实现了初步的定值到达分析和支配节点分析。

由于我们可能要完成多种数据流分析功能，而且进行数据流分析时，都需要将数据流分析方程求解的中间结果放在控制流图的节点中，以便进行迭代求解。我们目前在作为控制流图节点的可执行点类ExecutionPoint中设置了字段recorder记录数据流分析方程求解的中间结果（当然也包括最终的结果），它的类型统一为接口IFlowInfoRecorder，这个接口本身没有声明任何方法，只是一个标志，并作为所有数据流分析方程求解（中间）结果的基类型。

当我们要实现一种数据流分析功能时：

(1) 首先，需要为存储在可执行点的数据流分析中间结果定义一个接口，这个接口继承`IFlowInfoRecorder`，并根据要存储的数据流分析中间结果定义相应的读写结果信息的方法；

(2) 其次，需要为存储在可执行点的数据流分析中间结果定义一个类实现(1)中所说的接口，这个类的对象示例真正存放数据流分析（中间）结果；

(3) 最后，我们为这种数据流分析功能实现一个分析器类，这个类利用(2)中所定义的数据流分析中间结果类在控制流图的可执行点存放中间结果，并对数据流分析方程进行迭代求解，求解最终的结果存储在每个可执行点的`recorder`字段（所指向的对象实例），供其他程序使用。

之所以既然为数据流分析结果定义一个接口又要定义一个类（来实现这个接口）是因为Java语言的类只支持单继承，而接口则可多继承。利用这种接口+类的方式，我们可以为每种数据流分析功能（的结果读写）定义一个接口，而一个真正实现数据流分析功能的类则可实现多个接口以同时完成多个数据流分析功能及其结果的读写。

目前程序包`graph.cfg.analyzer`初步实现了到达定值分析和支配节点分析，下面对这两种分析的含义及其在`JAnalyzer`的实现进行简单的说明。

到达定值分析

对于一个方法中的名字引用，我们需要静态分析它可能的值，或者更准确地说，在某个源代码位置的名字引用，当程序的控制流到达这个点时，这个名字引用所绑定的名字定义可能在此前哪些地方被赋值，这些赋值在数据流分析称为一个名字定义（即局部变量、方法参数和字段）的定值（defined value）。给定某个源代码位置处的名字引用，分析通过控制流能到达该源代码位置处对该名字引用所绑定（引用）的名字定义的可能定值，在数据流分析中称为到达定值(reaching definitions)分析。

为简单起见，这里使用程序控制流图(control flow graph, CFG)的节点表示源代码位置。给定CFG节点 n ，它的到达定值是三元组 (m, v, exp) 的列表，其中 m 是一个CFG节点， v 是一个名字定义（代表一个局部变量、方法参数或字段）， exp 是一个表达式。三元组 (m, v, exp) 的含义是在CFG节点 m 对 v 使用表达式 exp 进行定值。定值三元组 (m, v, exp) 是节点 n 的到达定值（属于节点 n 的到达定值列表），表示从节点 m 到 n 存在一条控制流图路径，且在这条路径上没有对名字定义 v 的定值（直观地就是在这条路径上没有改变 v 在 m 处所赋的值，从而在 m 处对 v 的赋值（定值）可以到达 n ）。

对于CFG节点 n 处的名字引用`nameref`，它的到达定值是节点 n 的到达定值列表的子集，即若节点 n 的到达定值 (m, v, exp) 中的名字定义 v 是`nameref`所绑定的名字定义，则 (m, v, exp) 是`nameref`的到达定值。

到达定值分析是经典的数据流分析，给定一个方法的控制流图，假定它有一个虚拟的起始节点 s 和终止节点 e 。为了求每个节点的到达定值列表，首先要初始化每个节点 n 的生成定值列表`gen(n)`和杀死定值列表`kill(n)`，其次进行迭代求每个节点的到达定值列表：设控制流进入该节点时的到达定值列表是`IN(n)`，而控制流离开该节点时的到达定值列表是`OUT(n)`，通过迭代求下面的

数据流方程得到每个节点的 $IN(n)$ 和 $OUT(n)$:

$$OUT(n) = gen(n) \cup (IN(n) - kill(n))$$

$$IN(n) = \bigcup_{m \in Pred(n)} OUT(m)$$

直观地, 控制流离开节点 n 的到达定值列表 $OUT(n)$ 等于进入该节点的到达定值列表删除该节点的杀死定值列表再并上该节点的生成定值列表。而进入该节点 n 的到达定值列表 $IN(n)$ 等于 n 的所有前驱节点 m 的 $OUT(m)$ 的并, 这里 $Pred(n)$ 给出在控制流图中 n 的所有前驱节点 m , 即 m 有有向边到 n 。

实际实现时, 我们可将 $OUT(n)$ 看做节点 n 的所有到达定值列表, 并在初始化时计算每个节点 n 的生成定值列表 $gen(n)$ 。而节点 n 的杀死定值列表 $kill(n)$ 则可在迭代时通过 $gen(n)$ 推算: 对 n 的每个前驱节点 m 的 $OUT(m)$ 中的每个到达定值 (p, v, exp) , 如果在节点 n 生成名字定义 n 的定值 (n, v, exp') (即存在 (n, v, exp') 属于 $gen(n)$), 那么 (p, v, exp) 属于 $kill(n)$ 。

由于控制流图的虚拟起始节点 s 没有前驱, 因此 $OUT(s)$ 的定义是迭代求上述数据流方程的初始条件。对于到达定值分析, $OUT(s)$ 定义为方法所有参数对应的定值, 即为方法的每个参数 par 引入一个哑定值 $(s, par, -)$, $OUT(s)$ 为所有参数的哑定值构成的列表。对于控制流图的其他节点 n , 多数节点的 $gen(n)$ 都是空列表, 除了下面几种类型的节点:

- (1) 如果节点 n 是赋值语句 (在抽象语法树中是Eclipse JDT中类Assignment的对象实例) $lexp = rex$, 则 $lexp$ 必然绑定到一个名字定义 v (对于合法的Java程序而言), 则 $gen(n)$ 含有定值 (n, v, rex) ;
- (2) 如果节点 n 是变量声明语句 (对应类VariableDeclarationStatement) 或变量声明表达式 (对应类VariableDeclarationExpression), 则对它的每个有初始化表达式的声明片段 (对应类VariableDeclarationFragment), 假定其形式是“类型变量 $v =$ 初始化表达式 exp ”, 则 $gen(n)$ 含有定值 (n, v, exp) ;
- (3) 对于增强型for语句中的形式变量声明, 即`for (v : exp) ...`中声明的名字定义 v , 我们将 $(v : exp)$ 也看做循环语句中的谓词条件表达式, 在生成控制流图时生成节点 n , 该节点的 $gen(n)$ 含有定值 (n, v, exp) ;
- (4) 如果节点 n 是有副作用的一元运算++和--的表达式 $lexp ++, ++ lexp, -- lexp, lexp --$, 则对于合法的Java程序 $lexp$ 必然绑定到某个名字定义 v , 从而 $gen(n)$ 含有定值 $(n, v, lexp)$ 。如果我们只是考虑声明的类型为对象类型的名字定义, 则由于对象类型不能做这样的一元运算, 因此就可不考虑这种情况。

对于这些节点按照上述方式得到 $gen(n)$, 并且按照上述讨论确定 $kill(n)$, 从而可使用迭代求解上述方程。所谓的迭代求解, 则可设置一个布尔变量`change`, 其初始值为`false`, 只要在迭代过程中有一个节点的 $OUT(n)$ 值进行了改变, 则`change`置为`true`。当循环一次后没有任何节点的 $OUT(n)$ 值做了改变, 即循环一次后`change`的值如果还是`false`, 则迭代结束, 最后得到的每个节点 $OUT(n)$, 则是该节点的到达定值列表。基于每个节点的到达定值列表, 可基于前面的讨论确定**在节点 n 中每个对象表达式 exp 的到达定值列表 $RD(n, exp)$** 。

在JAnalyzer中, 我们使用以下类与接口实现到达定值分析:

- (1) 类ReachNameDefinition的实例代表一个三元组 $\langle node, name, value \rangle$ (即这个类声明了这三个字段), 其中`node`是类ExecutionPoint的实例, `name`是类NameDefinition的实例, 而`value`是类NameReference的实例。类ReachNameDefinition的实例代表一个定值: 即在控制流图节点`node`对名字定义`name`做的`value`给出的赋值, 也即上面所说的定值三元组 (m, v, exp) ;

(2) 接口 `IReachNameRecorder`, 它扩展接口 `IFlowInfoRecorder`, 声明了方法 `addGeneratedName()` 添加一个定值 (以构成当前节点的 $gen(n)$ 集); 方法 `addReachName()` 添加一个到达定值 (以得到当前节点的所有到达定值列表, 可理解为上面所说的 $OUT(n)$); 方法 `getGeneratedNameList()` 和方法 `getReachNameList()` 分别得到当前节点生成的定值列表和所有到达定值列表;

(3) 类 `ReachNameRecorder`, 实现接口 `IReachNameRecorder`, 有两个类型为 `List<ReachNameDefinition>` 的列表 `generatedNameList` 和 `definedNameList` 分别存放当前节点生成的定值列表和到达定值列表, 并实现了接口 `IReachNameRecorder` 的方法;

(4) 类 `ReachNameAnalyzer`, 它实现到达定值分析功能, 准确地说它提供的静态方法 `reachNameAnalysis()` 实现了到达定值分析功能, 下面我们对这个类的实现做更多的说明。图 7.6 给出了这个类声明的方法 (所有方法都是静态方法, 因此这个类实际上是一个静态工具类)。

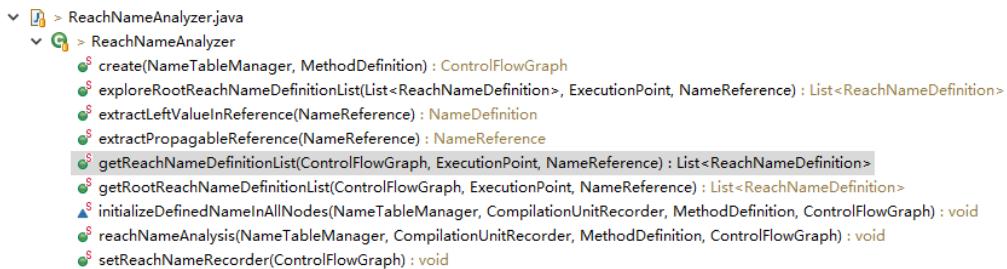


图 7.6 类 `ReachNameAnalyzer` 声明的方法

类 `ReachNameAnalyzer` 的静态方法 `reachNameAnalysis()` 依次需要名字表管理器 (类 `NameTableManager`)、编译单元信息记录 (类 `CompilationUnitRecorder`)、方法定义 (类 `MethodDefinition`) 和控制流图 (`ControlFlowGraph`) 的对象实例作为参数。调用者需保证实际参数控制流图是实际参数方法定义所指定方法的控制流图, 而且所给出的方法定义属于实际参数编译单元信息记录所给出的编译单元, 同时该编译单元属于所给定的名字表管理器。该方法大致实现了上面所说的到达定值分析过程: 它调用静态方法 `initializeDefinedNameInAllNodes()` 为每个控制流图的节点初始化其生成的定值列表, 然后使用一个循环对于到达定值分析进行迭代求解。注意每个节点存放的到达定值列表相当于前面所说的 $OUT(n)$, 由于我们针对每个节点进行分析, 所以无需 $IN(n)$ (实际上, 上一次迭代的 $OUT(n)$ 就是这一次迭代的 $IN(n)$), 并且我们使用 $gen(n)$ 来推算 $kill(n)$ 。

类 `ReachNameAnalyzer` 的静态方法 `initializeDefinedNameInAllNodes()` 初始化每个控制流图节点的生成定值列表, 实现了上面所说的 $gen(n)$ 的求解过程, 对含有赋值语句、有初始化表达式的变量声明语句 (包括 `for` 循环中的初始化表达)、增强 `for` 语句中的变量声明以及带有副作用的 `++`, `--` 运算符表达式的控制流图节点初始化其生成定值列表。

类 `ReachNameAnalyzer` 的静态方法 `create(NameTableManager, MethodDefinition)` 在给定名字表管理器和方法定义信息的情况下创建该方法定义所对应方法的控制流图, 并且该控制流图的每个节点都含有到达定值列表信息。这个方法基于类 `CFGCreator` 的静态 `create()` 方法创建给定方法的控制流图, 然后调用类 `ReachNameAnalyzer` 的静态方法 `setReachNameRecorder()` 方法初始化该控制流图的每个节点 (可执行点) 的 `recorder` 字段引用类 `ReachNameRecorder` 的对象实例, 最后调用方法 `reachNameAnalysis()` 完成到达定值的分析, 分析的结果存放在控制流图的各个节点中。

我们将初始化控制流图每个节点 (可执行点) 用于记录数据流分析结构的字段 `recorder` 的方法 `setReachNameRecorder()` 与方法 `reachNameAnalysis()` 分开的目的在于, 后者在读写节点中的

生成定值和到达定值信息时是基于recorder引用的是实现接口IReachNameRecorder的类的对象实例，而不是类ReachNameRecorder的实例，也要求节点有生成定值和到达定值的信息，但不限于其仅仅存放生成定值和到达定值的信息。所以，调用方法reachNameAnalysis()的前提是每个控制流图的节点已经设置了用于保存数据分析结果信息的对象实例，这个对象实例是实现了接口IReachNameRecorder的类的对象实例。

类ReachNameAnalyzer的静态方法getReachNameDefinitionList(ControlFlowGraph, ExecutionPoint, NameReference)给返回给定控制流图的给定可执行点中某个名字引用的到达定值列表，当然其前提是该控制流图的节点中存放了到达定值信息（例如是使用这个类提供的create()方法创建的控制流图）。

虽然经典的数据流分析可得到每个对象表达式（所绑定的名字定义）的可能定值，但是上述分析没有考虑定值的传播，例如，对于下面的程序片段：

```
(1) Class obj1 = fun()
(2) Class obj2 = obj1
(3) if (obj2 == null) { ... }
```

在第(3)行，对象表达式obj2的到达定值是((2), obj2, obj1)，也即我们能得到obj2指向的是obj1所指向的对象，但是却没有进一步的信息知道对obj2的空值检测实际上检测的是方法fun()的返回值。我们希望在分析时能考虑定值的简单传播，从而对空值检测的目标以及对象表达式真正指向的值有更好的分析。为此我们并不做复杂的别名分析，而是在到达定值分析的基础上做一些扩充，将一些简单传播的表达式（语句）考虑在内。

对于节点n处的对象表达式 $objexp$ 的一个到达定值 (m, v, exp) ，这里 v 是 $objexp$ 所绑定的名字定义，在节点m处使用 exp 对 v 进行了定值。我们考虑 exp 的两种简单情形：

- (1) 用于定值 v 的表达式 exp 本身绑定到局部变量或参数 u ，则认为节点m处的每个形如 (k, u, exp') 的到达定值可传播给到达定值 (m, v, exp) ，记为 $(k, u, exp') \rightarrow (m, v, exp)$ ；
- (2) 用于定值 v 的表达式 exp 是类型转换表达式(type) exp' ，而 exp' 本身绑定到局部变量或参数 u ，则也认为节点m处每个形如 (k, u, exp'') 的到达定值可传播给定值 (m, v, exp) ，记为 $(k, u, exp'') \rightarrow (m, v, exp)$ 。

从而→给出了到达定值三元组之间的传播关系，记 \rightarrow^* 为这个关系的传递闭包，则节点n处的对象表达式 $objexp$ 的一个根源到达定值(root reaching definitions) (m, v, exp) 满足：存在 $(k, u, exp') \in RD(n, exp)$ ，使得 $(m, v, exp) \rightarrow^* (k, u, exp')$ ，且不再存在定值 (l, w, exp'') 使得 $(l, w, exp'') \rightarrow (m, v, exp)$ ，也即 exp 不再绑定到局部变量或参数，也不是类型换表达式(type) exp'' ，其中 exp'' 绑定到局部变量或参数。记 $RRD(n, exp)$ 为节点n处的对象表达式 exp 的所有根源到达定值列表。

上面这个定义看起来有点复杂，但实际上使用算法实现求节点n处的对象表达式 exp 的所有根源到达定值列表比较简单，只要对 $RD(n, exp)$ 中的每个到达定值 (m, v, exp') ，基于 exp 的形式针对上面两种简单情形不断迭代直到最后得到的定值 (k, u, exp'') 中的表达式 exp'' 不绑定到局部变量或参数，也不是符合要求的类型转换表达式即可。但这个实现要注意避免死循环的情况，例如对于下面的程序片段，其中 $obj1, obj2$ 是对象引用， b, c 是条件表达式。按照定义代码行（控制流图节点）(4)处的对象引用 $obj2$ 的到达定值是：

$$RD((4), obj2) = \{\langle (3), obj2, obj1 \rangle\}$$

```

(1) obj1 = new Class();
(2) while (b) {
(3)     obj2 = obj1;
(4)     obj2.m()
(5)     if (c) obj1 = obj2;
(6) }
```

这里表达式obj1绑定到一个局部变量，节点(3)处，obj1的到达定值是：

$$RD((3), \text{obj1}) = \{\langle(1), \text{obj1}, \text{new Class}()\rangle, \langle(5), \text{obj1}, \text{obj2}\rangle\}$$

因此有：

$$\begin{aligned} \langle(1), \text{obj1}, \text{new Class}()\rangle &\longrightarrow \langle(3), \text{obj2}, \text{obj1}\rangle \\ \langle(5), \text{obj1}, \text{obj2}\rangle &\longrightarrow \langle(3), \text{obj2}, \text{obj1}\rangle \end{aligned}$$

注意到节点(5)处obj2的到达定值又是 $\langle(3), \text{obj2}, \text{obj1}\rangle$ ，从而有：

$$\begin{aligned} \langle(1), \text{obj1}, \text{new Class}()\rangle &\longrightarrow \langle(3), \text{obj2}, \text{obj1}\rangle \\ \langle(5), \text{obj1}, \text{obj2}\rangle &\longrightarrow \langle(3), \text{obj2}, \text{obj1}\rangle \\ \langle(3), \text{obj2}, \text{obj1}\rangle &\longrightarrow \langle(5), \text{obj1}, \text{obj2}\rangle \end{aligned}$$

这是上述程序片段的所有到达定值传播关系，它的闭包是：

$$\begin{aligned} \langle(1), \text{obj1}, \text{new Class}()\rangle &\longrightarrow \langle(3), \text{obj2}, \text{obj1}\rangle \quad \langle(1), \text{obj1}, \text{new Class}()\rangle \longrightarrow \langle(5), \text{obj1}, \text{obj2}\rangle \\ \langle(5), \text{obj1}, \text{obj2}\rangle &\longrightarrow \langle(3), \text{obj2}, \text{obj1}\rangle \quad \langle(5), \text{obj1}, \text{obj2}\rangle \longrightarrow \langle(5), \text{obj1}, \text{obj2}\rangle \\ \langle(3), \text{obj2}, \text{obj1}\rangle &\longrightarrow \langle(5), \text{obj1}, \text{obj2}\rangle \quad \langle(3), \text{obj2}, \text{obj1}\rangle \longrightarrow \langle(3), \text{obj2}, \text{obj1}\rangle \end{aligned}$$

从而根据上述定义，节点(4)处对象引用obj2的根源定值只有 $\langle(1), \text{obj1}, \text{new Class}()\rangle$ 。

如果严格按照定义求出所有定值传播关系，并求它的闭包这对于求某个对象引用的所有根源定值而言显然是低效的。但如果按照上述所说的递归算法，求节点(4)处对象引用obj2的根源定值就需要递归求节点(3)处对象引用obj1的根源定值，而求节点(3)处对象引用obj1的根源定值又需要递归求节点(5)处对象引用obj2的根源定值，但节点(5)处对象引用obj2的到达定值又包括 $\langle(3), \text{obj2}, \text{obj1}\rangle$ ，这又会需要递归求节点(3)处对象引用obj1的根源定值，从而陷入死循环。为了避免这种死循环，我们实现时在求某个节点处对象引用的根源定值时会记录已经探索过的到达定值，不会对已经探索过到达定值重复探索，也即，对于上述例子而言，在求节点(4)处对象引用obj2的根源定值时，当递归调用到求节点(5)处对象引用obj2的根源定值时，不会再考虑它的到达定值 $\langle(3), \text{obj2}, \text{obj1}\rangle$ ，因为这个到达定值在求节点(4)处对象引用obj2的根源定值的一开始已经探索过，从而避免陷入循环递归调用。

类ReachNameAnalyzer的静态方法getRootReachNameDefinitionList()给返回给定控制流图的给定可执行点中某个名字引用的根源到达定值列表（其形式参数与方法getReachNameDefinitionList()相

同), 当然其前提也是该控制流图的节点中存放了到达定值信息。实际上, 这个方法是调用静态方法`exploreRootReachNameDefinitionList()`按照上面所说的过程探索一个名字引用的根源到达定值。方法`getRootReachNameDefinitionList()`实际上是给出一个探索的起点, 而方法`exploreRootReachNameDefinitionList()`则会利用方法`extractPropagableReference()`提取定值中用于赋值的表达式中的可能进行定值传播的简单名字引用表达式或类型转换表达式, 从而递归地调用方法`exploreRootReachNameDefinitionList()`自己进行根源到达定值的探索, 在探索过程中按照上面所说的策略避免陷入死循环。

最后类`ReachNameAnalyzer`的静态方法`extractLeftValueInReference(NameReference)`用于提取一个名字引用(表达式)中的左值表达式, 例如赋值表达式中被赋值的名字(及左值)。这个方法用于到达定值分析中确定一个表达式到底是对哪个名字引用(及其绑定的名字定义)进行定值。

支配节点分析

说控制流图节点 m 是节点 n 的支配节点是指从方法的起始节点 s 到节点 n 的每条程序执行路径都必然经过 m 。给定控制流图, 支配其中一个节点 n 的所有节点构成的集合记为 $DOM(n)$, 也可通过数据流分析得到:

- (1) 初始化时, 起始节点 s 的 $DOM(s) = \{s\}$, 其他节点 n 的 $DOM(n)$ 为控制流图中所有节点;
- (2) 在迭代求解时, 节点 n 的 $DOM(n)$ 为它的所有前驱节点 m 的 $DOM(m)$ 的交再并上 $\{n\}$, 直到所有节点 n 的 $DOM(n)$ 不再改变。

类似于到达定值分析, JAnalyzer使用以下接口和类实现支配节点分析:

- (1) 接口`IDominateNodeRecorder`, 它扩展接口`IFlowInfoRecorder`, 声明方法`getDominateNodeList()`获取支配当前节点的所有节点列表; 方法`setDominateNodeList()`设置支配当前节点的所有节点列表;
- (2) 类`DominateNodeRecorder`, 实现接口`IDominateRecorder`, 有一个类型为`List<GraphNode>`的列表`dominateNodeList`存放支配当前节点的所有节点的列表。之所以选用类型`GraphNode`而不是`CFGNode`是因为实际上在一般的有向图中也可以有支配节点的概念;
- (3) 类`DominateNodeAnalyzer`, 它提供的静态方法`dominateNodeAnalysis()`实现支配节点分析功能。下面我们将对这个类的实现做更多的说明。图7.7给出了这个类声明的方法(所有方法都是静态方法, 因此这个类也是一个静态工具类)。类`DominateNodeAnalyzer`的静态方

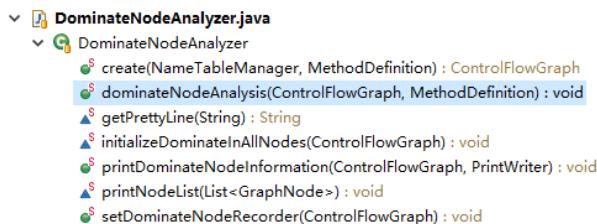


图 7.7 类`DominateNodeAnalyzer`声明的方法

法`dominateNodeAnalysis()`调用方法`initializeDominateInAllNodes()`初始化每个节点的支配节点, 然后再按照上面所说的迭代求解过程求得每个节点的所有支配节点列表。调用这个方法的前提是控制流图的每个节点设置了用于保存数据流分析结果信息的对象实例, 这个对象实例必须是实现了接口`IDominateNodeRecorder`的类的对象实例。类`DominateNodeAnalyzer`的静态方

法`setDominantNodeRecorder()`即可为给定控制流图的每个节点设置类`DominantNodeRecorder`的对象实例用于保存数据流分析结果（准确地说就是支配节点分析结果）信息。

类`DominantNodeAnalyzer`的静态方法`create()`在给定名字表管理器和方法定义信息下的情况创建该方法定义所对应方法的控制流图，并且该控制流图的每个节点都含有支配节点列表信息。这个方法基于类`CFGCreator`的静态`create()`方法创建给定方法的控制流图，然后调用类`DominantNodeAnalyzer`的静态方法`setDominantNodeRecorder()`方法初始化该控制流图的每个节点（可执行点）的`recorder`字段引用类`DominantNodeRecorder`的对象实例，最后调用方法`dominateNodeAnalysis()`完成支配节点的分析，分析的结果存放在控制流图的各个节点中。

类`DominantNodeAnalyzer`的静态方法`printDominantNodeInformation()`用于输出支配节点信息以方便调试，而静态方法`printNodeList()`可输出一个节点列表中的所有节点信息，静态方法`getPrettyLine()`可为节点标签或描述信息输出固定长度的信息（而省略其中更长的信息），以美化整个节点信息的输出。

基于控制流图分析的应用示例

程序包`graph.cfg.analyzer`的编译单元`TestCFGCreator.java`给出了一些测试程序展示控制流图的生成并基于控制流图进行数据流分析。图7.8给出了其中方法`testCreateCFGWithReachName()`的核心代码片段。

```

247     ControlFlowGraph cfg = ReachNameAndDominantNodeAnalyzer.create(tableManager, method);
248
249     List<GraphNode> nodeList = cfg.getAllNodes();
250     System.out.println("Before write execution point " + nodeList.size() + " nodes!");
251     for (GraphNode graphNode : nodeList) {
252         if (graphNode instanceof ExecutionPoint) {
253             ExecutionPoint node = (ExecutionPoint)graphNode;
254             ReachNameRecorder recorder = (ReachNameRecorder)node.getFlowInfoRecorder();
255             List<ReachNameDefinition> definedNameList = recorder.getReachNameList();
256             for (ReachNameDefinition definedName : definedNameList) {
257                 NameDefinition name = definedName.getName();
258                 NameReference value = definedName.getValue();
259                 if (definedName.getValue() != null) {
260                     output.println("[" + graphNode.getId() + "]\t" + name.getSimpleName() + "\t" + value.toString());
261                 } else {
262                     output.println("[" + graphNode.getId() + "]\t" + definedName.getName().getSimpleName() + "\t");
263                 }
264             }
265         } else {
266             output.println(graphNode.getId() + "\t~~\t~~\t~~\t~~");
267             System.out.println("Found none execution point with defined name node!");
268         }
269     }
}

```

图 7.8 类`DominantNodeAnalyzer`声明的方法

方法`testCreateCFGWithReachName()`以给定软件项目根目录创建名字表管理器对象`tableManager`（类`NameTableManager`的对象实例），并使用名字表访问器类`NameDefinitionVisitor`的对象实例找到该项目中源代码行数最多的方法`method`（类`MethodDefinition`）的对象实例，然后在第247行使用类`ReachNameAndDominantNodeAnalyzer`的静态方法`create()`创建一个其节点既有到达定值信息又有支配节点信息的控制流图。第249行获得该控制流图的所有节点列表，然后第251到第269行的循环可对其中的每个节点进行处理。第252行的判断节点是否是可执行点（实际上就目前的实现来说所有控制流图的节点都是可执行点，该判断可省略），第253行将节点的类型转换为可执行节点，第254行利用可执行点的`getFlowInfoRecorder()`方法获得保存数据流分析结果的记录信息，并将其类型转换为`ReachNameRecorder`。这里获得的记录信息到底能转换成怎样的信息取决于用户为可执行点设置的信息类型，类`ReachNameAndDominatedNodeAnalyzer`使

用它的`setReachNameAndDominantNodeRecorder()`方法为每个可执行点设置的记录信息类型为类`ReachNameAndDominantNodeRecorder`, 这个类继承类`ReachNameRecorder`, 并实现了接口`IDominantNodeRecorder`, 因此可认为它既包含到达定值信息也包含支配节点信息, 也即图7.8的第254行调用`node.getFlowInfoRecorder()`的返回结果也可转换为`IDominantNodeRecorder`类型的变量, 从而访问该节点的支配节点信息。

类图`ReachNameAndDominatedNodeAnalyzer`只有两个方法, 一个方法是`create()`, 创建既有到达定值信息又有支配节点信息的控制流图, 另一方方法是`setReachNameAndDominantNodeRecorder()`, 为每个控制流图的节点设置保存数据流分析结果信息的对象实例, 所设置的对象实例是类`ReachNameAndDominantNodeRecorder`的对象实例。

类`ReachNameAndDominantNodeRecorder`继承类`ReachNameRecorder`并实现接口`IDominantNodeRecorder`, 为实现该接口它声明了字段`dominateNodeList`保存支配节点信息, 并实现方法`getDominantNodeList()`和`setDominantNodeList()`读取和设置支配节点信息, 至于到达定值信息的读写则使用继承的字段和继承的方法实现。因此使用接口声明数据分析结果应读写的信息可灵活地支持同时进行多种数据分析, 而不会被Java语言的类只能单继承而限制。

图7.9给出了类`ReachNameAndDominantNodeRecorder`的`create()`方法的源代码, 其中第25行到第30行主要是根据名字管理器及方法定义找到编译单元信息, 第33行创建控制流图, 使用类`CFGCreator`的方法`create()`创建的控制流图并不含数据流分析信息, 第36行设置保存数据流分析结果的记录对象, 然后第37行和第38行分别调用类`ReachNameAnalyzer`的`reachNameAnalysis()`方法和类`DominantNodeAnalyzer`的`dominantNodeAnalysis()`方法完成到达定值分析和支配节点分析。实际上, 类`ReachNameAnalyzer`的`create()`方法, 以及类`DominantNodeAnalyzer`的`create()`有极为相似的源代码, 只是所设置的保存数据流分析结果的记录对象不同而已。图7.9的源代码如果不调用`dominantNodeAnalysis()`方法则只进行到达定值分析, 同样不调用`reachNameAnalysis()`方法则只完成支配节点分析。类似地, 如果要同时做更多的数据流分析, 则只要保存数据分析结果的记录对象能存储相应的信息, 则可通过实现更多的...`Analyzer`类的...`Analysis()`方法来完成这样的数据分析功能。

```

24@ public static ControlFlowGraph create(NameTableManager nameTable, MethodDefinition method) {
25    CompilationUnitScope unitScope = nameTable.getEnclosingCompilationUnitScope(method);
26    if (unitScope == null) return null;
27    String sourceFileName = unitScope.getUnitName();
28    CompilationUnit astRoot = nameTable.getSourceCodeFileSet().findSourceCodeASTRootByFileUnitName(sourceFileName);
29    if (astRoot == null) return null;
30    CompilationUnitRecorder unitRecorder = new CompilationUnitRecorder(sourceFileName, astRoot);
31
32    // Create a ControlFlowGraph object
33    ControlFlowGraph currentCFG = CFGCreator.create(nameTable, method);
34    if (currentCFG == null) return null;
35
36    setReachNameAndDominantNodeRecorder(currentCFG);
37    ReachNameAnalyzer.reachNameAnalysis(nameTable, unitRecorder, method, currentCFG);
38    DominantNodeAnalyzer.dominantNodeAnalysis(currentCFG, method);
39
40    return currentCFG;
41}

```

图 7.9 类`ReachNameAndDominantNodeRecorder`的`create()`方法

参考文献

- [1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification (Java SE 8 Edition)*. Oracle America, Inc., 2015. 中译本: 陈h 鹏译. Java 语言规范—基于Java SE 8. 机械工业出版社, 2016.