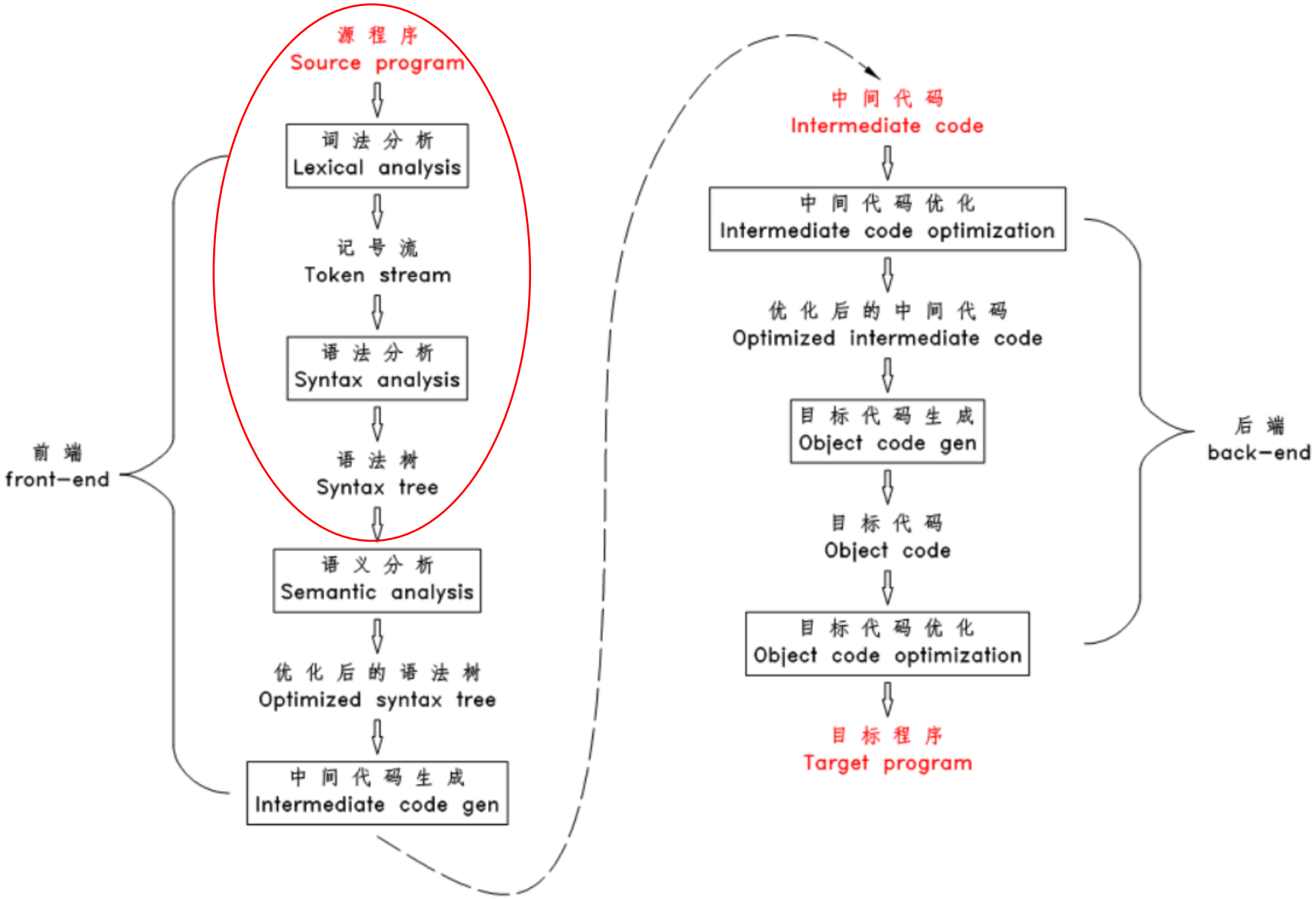
Simple Guidance to Spoon

CSTAR 王启凡

- 前置知识
- 使用场景
- 一些难点

前置知识-Compiler



前置知识-词法分析

编译器扫描源文件的字符流，过滤掉字符流中的空格、注释等，并将其分割为一个个的token

Token name	Sample token values
identifier	X, color,UP
keyword	if, wihile, return
separator	} (;
operator	+ * = ~
literal	true, 3.14e100, “what?”
comment	//todo: fix xxx leak bug

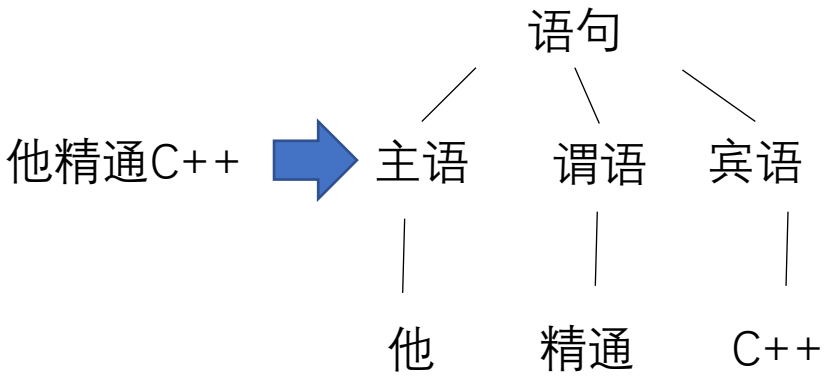
x = a + b * 2;

```
[(identifier, x), (operator, =),
(identifier, a), (operator, +),
(identifier, b), (operator, *),
(literal, 2), (separator, ;)]
```

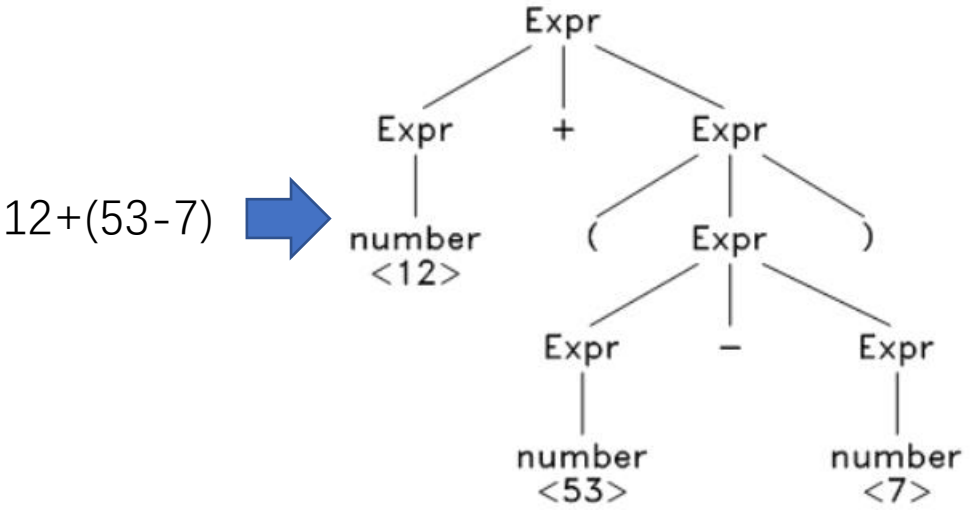
前置知识-语法分析

根据语言的语法规则来解析 token 流，生成抽象语法树。

语句 -> 主语 谓语 宾语
主语 -> 我 | 你 | 他 | 她 | 没人
谓语 -> 熟练 | 精通 | 爱
宾语 -> C++ | Java | Python



Expr -> Expr op Expr | (Expr) | number
op -> + - * /



前置知识-Spoon meta model

一个语言被实现的 meta models (元模型—描述模型的模型)是多样的，如javac与JDT不同。

抽象语法树 AST 可以视作 model，Spoon meta model 即是描述 Spoon建立的 AST 的model。

Spoon meta model three parts:

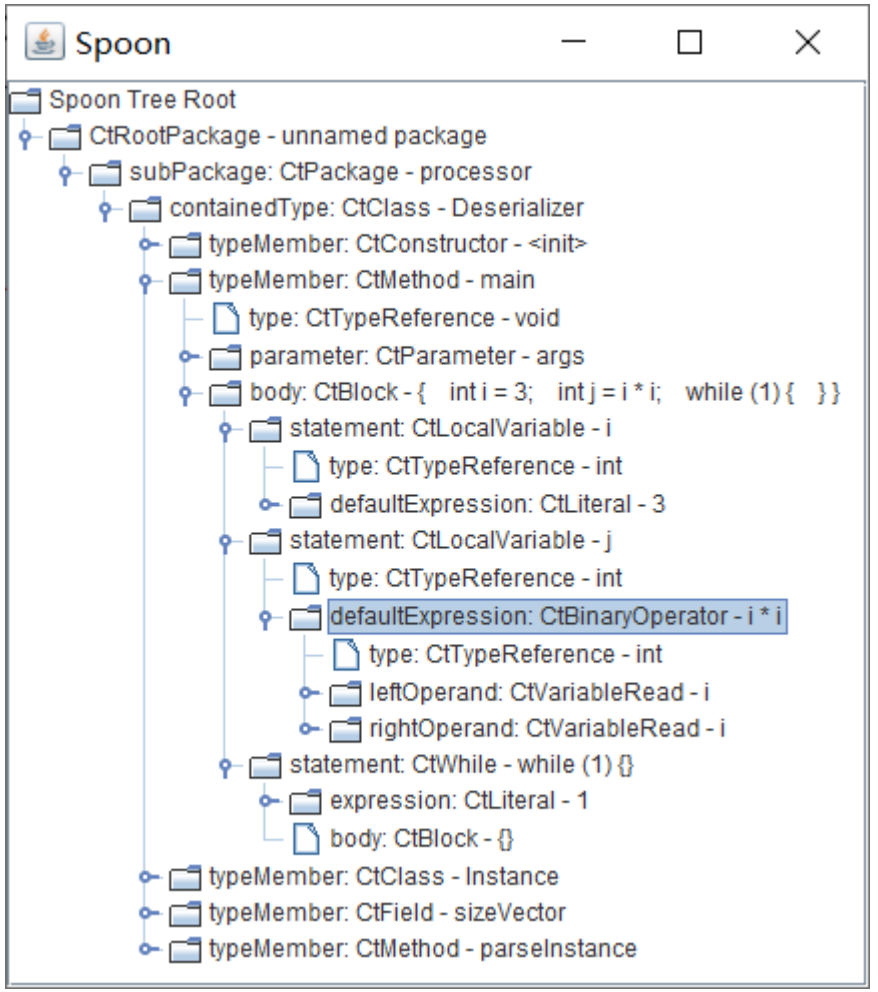
- The structural part contains the declarations of the program elements, such as interface, class, variable, method, annotation, and enum declarations.

- The code part contains the executable Java code, such as the one found in method bodies.

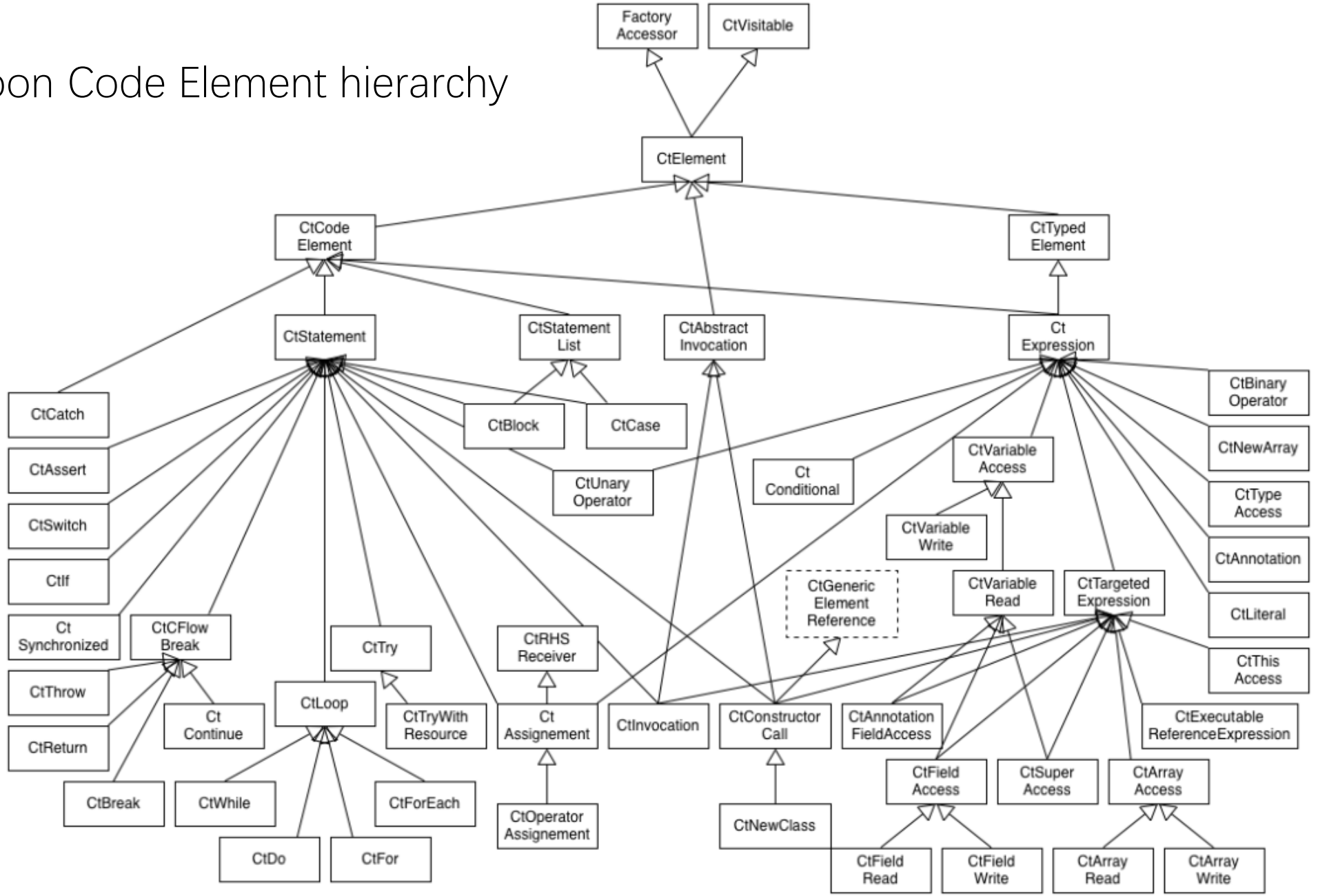
- The reference part models the references to program elements (for instance a reference to a type).

Windows D:\Java\jdk1.8.0_74\bin\java.exe
java -cp spoon-core-8.3.0-SNAPSHOT-jar-with-dependencies.jar spoon.Launcher -i .\Deserializer.java --gui

```java
package processor;

public class Deserializer {
    Run | Debug
    public static void main(String[] args) {
        int i = 3;
        int j = i * i;
        while(1) {

        }
    }

    private static class Instance {
        private final Vector<Double> vector = new Vector<>();

        public Instance(Vector<Double> v) {
            vector.addAll(v);
        }

        public Vector<Double> getVector() {
            return vector;
        }

        public int size() {
            return vector.size();
        }
    }

    static int sizeVector = 0;
    static void parseInstance(ArrayList<Instance> listInst) {
        sizeVector = 100;

        for(int i = 0; i < listInst.size(); ++i) {
            Instance inst = listInst.get(i);
            for(int j = 0; j < sizeVector; ++j) {
```



Spoon Code Element hierarchy



使用场景-位置信息/插入

Spoon 所有实现 SourcePositionHolder的 CodeElement 类型都提供了 getPosition 的接口，供用户获取语法树结点对应的源文件中的位置信息(文件名、行列号)。

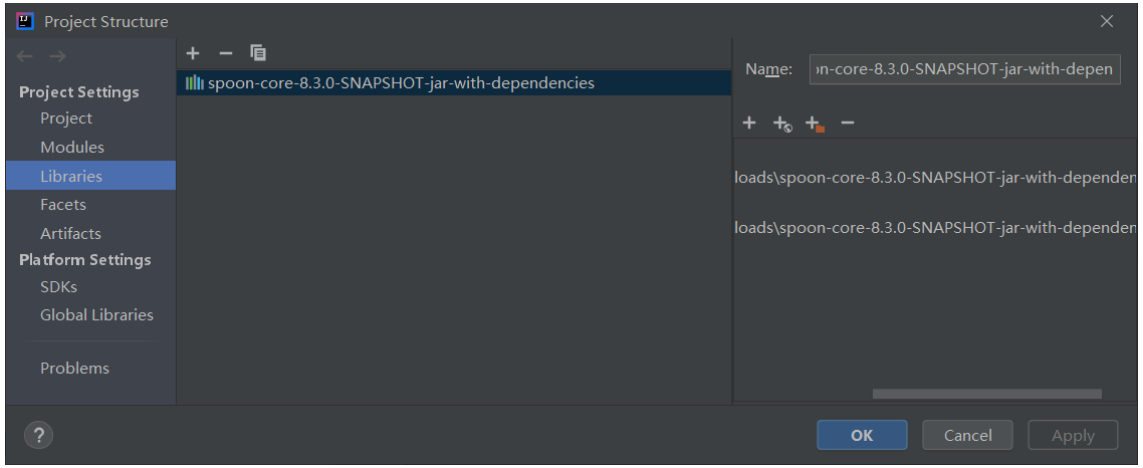
```
public interface SourcePositionHolder {
    SourcePosition getPosition();

    @Experimental
    default ElementSourceFragment getOriginalSourceFragment() {
        return ElementSourceFragment.NO_SOURCE_FRAGMENT;
    }
}
```

Spoon 所有实现 CtStatement 的 CodeElement 类型都提供了插入语句的相关接口，供用户修改语法树结点的相关结构。

```
public interface CtStatement extends CtCodeElement {
    <T extends CtStatement> T insertAfter(CtStatement var1) throws ParentNotInitializedException;

    <T extends CtStatement> T insertAfter(CtStatementList var1) throws ParentNotInitializedException;

    <T extends CtStatement> T insertBefore(CtStatement var1) throws ParentNotInitializedException;

    <T extends CtStatement> T insertBefore(CtStatementList var1) throws ParentNotInitializedException;
```

使用场景-命令行/Library

```
java -classpath /path/to/binary/of/your/processor.jar:spoon-core-8.3.0-with-dependencies.jar
spoon.Launcher -i /path/to/src/of/your/project -p processor.CustomedProcessor
```



pom.xml <dependencies>

```xml
<dependency>
    <groupId>fr.inria.gforge.spoon</groupId>
    <artifactId>spoon-core</artifactId>
    <version>8.3.0</version>
</dependency>
```

使用场景-静态分析/代码转换

使用 Spoon 提供的 Processor 根据语法树节点类型处理相关语法树结构。

```
SpoonAPI spoon = new Launcher();
//      spoon.addInputResource("./Deserializer.java");
spoon.addInputResource( s: "../GenJavaClass/src");
spoon.addProcessor(new CntLoop());
spoon.setSourceOutputDirectory("ReFac/");
spoon.addProcessor(new Insert());

spoon.buildModel();

spoon.process();

CntLoop.statistics();

spoon.prettyprint();
```

addInputResource

buildModel

addProccessor

process

prettyprint -Write the transformed files to disk

使用场景-Processor

Interface Processor<E extends CtElement>

All Superinterfaces:

FactoryAccessor

All Known Subinterfaces:

AnnotationProcessor<A,E>, FileGenerator<T>

All Known Implementing Classes:

AbstractAnnotationProcessor, AbstractManualProcessor, AbstractParallelProcessor, AbstractProcessor, ForceFullyQualifiedProcessor, ForceImportProcessor, ImportCleaner, ImportConflictDetector, JavaOutputProcessor, SpoonTagger

---

```
public interface Processor<E extends CtElement>
extends FactoryAccessor
```

This interface defines a generic code processor. To define a new processor, the user should subclass AbstractProcessor, the abstract default implementation of this interface. If a processor contains fields annotated with @Property, they can be set using a ProcessorProperties

使用场景-Processor

Modifier and Type	Method	Description
`Environment`	`getEnvironment()`	Gets the environment of this processor.
`java.util.Set<java.lang.Class<? extends CtElement>>`	`getProcessedElementTypes()`	Gets all the element types than need to be processed.
`TraversalStrategy`	`getTraversalStrategy()`	Gets the model's traversal strategy for this processor (default is `TraversalStrategy.POST_ORDER`).
`void`	`init()`	This method is upcalled to initialize the processor before each processing round.
`void`	`initProperties (ProcessorProperties properties)`	Initializes the properties defined by this processor by using the environment.
`void`	`interrupt()`	Interrupts the processing of this processor but changes on your AST are kept and the invocation of this method doesn't interrupt the processing of all processors specified in the `ProcessingManager`.
`boolean`	`isToBeProcessed(E candidate)`	Tells if this element is to be processed (returns `true` in the default implementation).
`void`	`process()`	A callback method upcalled by the manager so that this processor can manually implement a processing job.
`void`	`process(E element)`	A callback method upcalled by the meta-model scanner to perform a dedicated job on the currently scanned element.
`void`	`processingDone()`	This method is upcalled by the `ProcessingManager` when this processor has finished a full processing round on the program's model.

使用场景-Processor

在特定的位置上进行插桩。

```java
public class Insert extends AbstractProcessor<CtLoop> {

    @Override
    public void process(CtLoop ctLoop) {
        // we declare a new snippet of code to be inserted.
        CtCodeSnippetStatement snippet = getFactory().Core().createCodeSnippetStatement();

        // this snippet contains an if check.
        final String value = String.format( s: "System.out.println(\"%s\");", ctLoop.getPosition());
        snippet.setValue(value);

        ctLoop.insertBefore(snippet);
    }
}
```

使用场景-Processor

遍历特定的语法树结构。

Spoon 建立的语法树提供了丰富的API用于获取源代码信息与转移。

```java
// CtLoop
// All Known Subinterfaces:
// CtDo, CtFor, CtForEach, CtWhile
public class CntLoop extends AbstractProcessor<CtLoop> {

    static int totCount = 0;  // count all while block
    static int cntWhile = 0; // count all while block
    static int cntInnerLoop = 0; // count the loop that is in other loop

    @Override
    public void process(CtLoop ctLoop) {
        ++totCount;

        if(ctLoop instanceof CtWhile) {
            ++cntWhile;
        }

        List<CtLoop> inner = ctLoop.getBody().getElements(new TypeFilter<>(CtLoop.class));
        if(inner.size() > 0) {
            ++cntInnerLoop;
        }
    }

    public static void statistics() {
        System.out.println("totCount " + totCount);
        System.out.println("cntWhile " + cntWhile);
        System.out.println("cntInnerLoop " + cntInnerLoop);
    }
}
```

使用场景-Getters

大部分的CodeElemet 根据其类型特点都有相关 getter 接口。如 CtClass<T> 继承CtTyp<T>提供了获取Method的接口。
methods = ctClass.getMethods();

getMethods

```
java.util.Set<CtMethod<?>> getMethods()
```

Returns the methods that are directly declared by this class or interface. Derived from `getTypeMembers()`

最基础的 CtElement 提供了获取子节点的接口。
allDescendants = ctElement.getDirectChildren();

getDirectChildren

```
java.util.List<CtElement> getDirectChildren()
```

Returns:

a list of CtElement containing the element's direct children.

使用场景-Getters

CtRole 是一个枚举类，定义两个节点的"关系"(单向)，一般是父子节点的关系。

CtElement 提供的根据语法树节点关系(Role)提供的接口。
methods = ctClass.getValueByRole(CtRole.METHOD);

Enum Constant
ACCESSED_TYPE
ANNONYMOUS_EXECUTABLE
ANNOTATION
ANNOTATION_TYPE
ARGUMENT
ARGUMENT_TYPE
ASSIGNED
ASSIGNMENT
BODY
BOUND
BOUNDING_TYPE

getValueByRole

```
<T> T getValueByRole(CtRole role)
```

Parameters:

`role` - the role of the returned attribute with respect to this element. For instance, "klass.getValueByRole(CtRole.METHOD)" returns a list of methods. See `RoleHandlerHelper` for more advanced methods.

Returns:

a a single value (eg a CtElement), List, Set or Map depending on this `element` and `role`. Returned collections are read-only.

使用场景-Filters

Package spoon.reflect.visitor

Interface Filter<T extends CtElement>

Type Parameters:

T - the type of the filtered elements (an element belonging to the filtered element must be assignable from T).

All Known Implementing Classes:

`AbstractFilter`, `AbstractReferenceFilter`, `AnnotationFilter`, `CompositeFilter`, `DirectReferenceFilter`, `ExecutableReferenceFilter`, `FieldAccessFilter`, `InvocationFilter`, `LambdaFilter`, `LineFilter`, `NamedElementFilter`, `NameFilter`, `OverriddenMethodFilter`, `OverridingMethodFilter`, `ReferenceTypeFilter`, `RegexFilter`, `ReturnOrThrowFilter`, `SameFilter`, `SubtypeFilter`, `TemplateMatcher`, `TypeFilter`, `VariableAccessFilter`

```
public interface Filter<T extends CtElement>
```

This interface defines a filter for program elements.

matches

```
boolean matches(T element)
```

Tells if the given element matches.

Parameters:

`element` - - the element to be checked for a match. Parameter element is never null if `Query` is used.

使用场景-Getters & Filters

CtElement 提供getElements接口通过Filter进行筛选。

getElements

```
<E extends CtElement> java.util.List<E> getElements(Filter<E> filter)
```

Returns all the children elements recursively matching the filter. If the receiver (this) matches the filter, it is also returned

CtQuery 提供 filterChildren 接口通过筛选。

filterChildren

```
<R extends CtElement> CtQuery filterChildren(Filter<R> filter)
```

结合 Filter 使用 Getters

```
list1 = methodBody.getElements(new
TypeFilter(CtAssignment.class));

// collecting all deprecated classes
list2 = rootPackage.getElements(new
AnnotationFilter(Deprecated.class));

// creating a custom filter to select all public fields
list3 = rootPackage.filterChildren(
  new AbstractFilter<CtField>(CtField.class) {
    @Override
    public boolean matches(CtField field) {
      return field.getModifiers.contains(ModifierKind.PUBLIC);
    }
  }
).list();
```

使用场景-Getters & Filters

```
CtModel model = launcher.buildModel();
ExecutableProcessor.getInstance()
        .init(launcher.getFactory(), false,
                line.hasOption("askUser"), line.hasOption("middleInput"));

//get all methods(except Constructors)
for (CtMethod<?> cm : model.getRootPackage()
        .getElements(new TypeFilter<>(CtMethod.class))) {
    ExecutableProcessor.getInstance().addMethod(cm, cm.isStatic());
}
if(! line.hasOption("ignore-constructor")) {
    // get all Constructors
    for (CtConstructor<?> cc : model.getRootPackage()
            .getElements(new TypeFilter<>(CtConstructor.class))) {
        ExecutableProcessor.getInstance().addConstructor(cc);
    }
}
```

使用场景-Visitor

```java
package spoon.reflect.visitor;

public interface CtVisitable {
    void accept(CtVisitor var1);
}
```

所有的 CodeElement 都是实现了 CtVistable，提供了一个 accept 接口。
CtVisitor 是一个提供各种操作接口的intererface

```java
    void visitCtPackage(CtPackage var1);

    void visitCtPackageReference(CtPackageReference var1);

    <T> void visitCtParameter(CtParameter<T> var1);

    <T> void visitCtParameterReference(CtParameterReference<T> var1);

    <R> void visitCtReturn(CtReturn<R> var1);

    <R> void visitCtStatementList(CtStatementList var1);

    <S> void visitCtSwitch(CtSwitch<S> var1);

    <T, S> void visitCtSwitchExpression(CtSwitchExpression<T, S> var1);

    void visitCtSynchronized(CtSynchronized var1);

    void visitCtThrow(CtThrow var1);

    void visitCtTry(CtTry var1);

    void visitCtTryWithResource(CtTryWithResource var1);
```

访问者模式
在访问者模式（Visitor Pattern）中，使用访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。
根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。
分离数据与操作。

使用场景-Scanners/Iterator

Scanner 提供类似 Processor 的遍历方法，抽象类 CtScanner 实现了CtVisitor。所有实现了 CtVisitable 的类型(所有的CodeElement)类型都可以通过accept方法接受一个Scanner）。

灵活的是，它可以直接访问语法树节点而Processor不行。
使用类似于，
CounterScanner scanner = new CounterScanner();
ctClass.accept(scanner);

```
//Scanner counting the number of CtFieldWrite
static class CounterScanner extends CtScanner {
    private int visited = 0;
    @Override
    public <T> void visitCtFieldWrite(CtFieldWrite<T> fieldWrite) {
        visited++;
    }
}
```

Iterator 同样是实现了CtVisitor可以通过任意语法树节点构造，并进行遍历，可选择深度优先或广度优先。

```
CtIterator iter = new CtIterator(spoon.getModel().getRootPackage());
while (iter.hasNext()) {
    CtElement el = iter.next();
    //do something on each child of root
}
```

使用场景-Queries

```
public interface CtQueryable {
    <R extends CtElement> CtQuery filterChildren(Filter<R> var1);

    <I, R> CtQuery map(CtFunction<I, R> var1);

    <I> CtQuery map(CtConsumableFunction<I> var1);
}
```

Spoon 提供的 CtQuery 可以使用lambda表达式、流式且重用的函数式使用方式操作语法树。所有 CodeElement 均实现了CtQueriable。

CtQuery 提供了更多的函数式编程接口。
其中list为将结果返回为List。

```
// returns a list of String
list = package
.map((CtClass c) -> c.getSimpleName()).list();
```

```
// collecting all methods of deprecated classes
list2 = rootPackage
    .filterChildren(new AnnotationFilter(Deprecated.class)).list()
```

```
// a query which processes all not deprecated methods of all deprecated classes
rootPackage
  .filterChildren((CtClass clazz)->clazz.getAnnotation(Deprecated.class)!=null)
  .map((CtClass clazz)->clazz.getMethods())
  .map((CtMethod<?> method)->method.getAnnotation(Deprecated.class)==null)
  .list();
```

使用场景-代码生成

工厂模式
工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。
在工厂模式中，我们在创建对象时不会对用户暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

Spoon 所有实现 FactoryAccessor 的 CodeElement 类型都提供了 getFactory 的接口，供用户创建对象。

```
CtClass newClass = factory.createClass( s: "generated.java");
newClass.setSimpleName(i.getClass_name());

List<String> methodSignatures = i.getMethod_signatures();
for (String ms : methodSignatures) {

    Matcher matcher = r.matcher(ms);
    if(matcher.find() && matcher.groupCount() >= 2) {
        // function name
        String before = matcher.group( i: 1);
        CtMethod m = factory.createMethod();
        m.setBody(factory.createBlock());
        m.setSimpleName(before);

        // function parameters
        String after = matcher.group( i: 2);
        if(null != after) {
            String[] pList = after.split( s: ",");
            for(int ind = 0; ind < pList.length; ++ind) {
                CtParameter p = factory.createParameter();
                p.setSimpleName(pList[ind]);
                m.addParameter(p);
            }
        }

        newClass.addMethod(m);
    } else {
```

一些难点-第三方库

当使用第三方库时，通过 Spoon 的语法树有获取非源码 program element 的可能，无法获取。需要注入第三方库依赖。

依赖可知，通过JVM classpath注明或者在 launcher 中注明：
launcher.getEnvironment().setSourceClasspath("<classpath_project>");

存在依赖未知，通过launcher设置：

launcher.getEnvironment().setNoClasspath(true);

如果待分析工程是Maver，比较简单的方式，就是使用 MavenLauncher 通过 pom.xml 读取依赖

```
MavenLauncher launcher = new MavenLauncher("<path_to_maven_project>",
        MavenLauncher.SOURCE_TYPE.APP_SOURCE,
        new String[] {
            "/home/user/.m2/repository/org/my/jar/1.0/org-my-jar-1.0.jar"
        }
    );
```

```
// the second parameter can be APP_SOURCE / TEST_SOURCE / ALL_SOURCE
MavenLauncher launcher = new MavenLauncher("<path_to_maven_project>", MavenLauncher.SOURCE_TYPE.APP_SOURCE);
launcher.buildModel();
CtModel model = launcher.getModel();
```

一些难点-弱引用

Spoon提供的 CtReference 广泛存在于Spoon meta model 中，其表达了程序的。，获取的语法树节点引用为弱引用(weak reference)，即引用可能先于其引用的实例。

Case 1 (code available as source code): the reference points to a code element for which the source code is present. In this case, reference.getDeclaration() returns this code element (e.g. TypeReference.getDeclaration returns the CtType representing the given Java file). reference.getTypeDeclaration() is identical to reference.getDeclaration().

Case 2 (code available as binary in the classpath): the reference points to a code element for which the source code is NOT present, but for which the binary class is in the classpath (either the JVM classpath or the --source-classpath argument). In this case, reference.getDeclaration() returns null and reference.getTypeDeclaration returns a partial CtType built using runtime reflection. Those objects built using runtime reflection are called shadow objects, and you can identify them with method isShadow. (This also holds for getFieldDeclaration and getExecutableDeclaration).

Case 3 (code not available, aka noclasspath): the reference points to a code element for which the source code is NOT present, but for which the binary class is NOT in the classpath. This is called in Spoon the noclasspath mode. In this case, both reference.getDeclaration() and reference.getTypeDeclaration() return null. (This also holds for getFieldDeclaration and getExecutableDeclaration).

一些难点-比较问题

A `CtReference` targets its element in a weak way (Spoon references). It is not guaranteed that two references are the same object if they refer to the same program element.

slarse commented on 15 Jan — Collaborator

I agree with **@andrewbwogi** , it is correct that two identical methods are considered equal, even if they have different parents. Also worth noting is that hashing arbitrary Spoon elements is very demanding, as it is a recursive operation that traverses the entire subtree.

To create a set that hashes by identity, you can use `IdentityHashMap` in combination with `Collections.newSetFromMap()` . See this SO answer: https://stackoverflow.com/a/48096408

1

总结

简单使用 Spoon 的方式:

0. Build model(AST)
1. Traverse/transform AST with Visitor(Processor/Scanner)
2. Manipulate AST with Getter/Filter/Other APIs

适合 Spoon 的工作:

Static Code Analysis, Code Transformation,

Building Type System and Type Inference