前面介绍了关于编译时注解的使用方式，这里再补充一个关于注解处理器开发中的一些常用类、接口的使用方式和概念。

Element 和 TypeMirror

我觉得这两个是开发注解处理器最重要的两个概念，理解这两个的概念和使用是非常有必要的。

这两个都是接口，先来看一下 Element 的定义：

```
/**
 * 表示一个程序元素，比如包、类或者方法，有如下几种子接口：
 * ExecutableElement：表示某个类或接口的方法、构造方法或初始化程序（静态或实例），包括注解类型元素；
 * PackageElement：表示一个包程序元素；
 * TypeElement：表示一个类或接口程序元素；
 * TypeParameterElement：表示一般类、接口、方法或构造方法元素的形式类型参数；
 * VariableElement：表示一个字段、enum 常量、方法或构造方法参数、局部变量或异常参数
 */
public interface Element extends AnnotatedConstruct {
    /**
     * 返回此元素定义的类型
     * 例如，对于一般类元素 C<N extends Number>，返回参数化类型 C<N>
     */
    TypeMirror asType();

    /**
     * 返回此元素的种类：包、类、接口、方法、字段...,如下枚举值
     * PACKAGE, ENUM, CLASS, ANNOTATION_TYPE, INTERFACE, ENUM_CONSTANT, FIELD, PARAMETER, LOCAL_VARIABLE, EXCEPTION_PARAMETER,
     * METHOD, CONSTRUCTOR, STATIC_INIT, INSTANCE_INIT, TYPE_PARAMETER, OTHER, RESOURCE_VARIABLE;
     */
    ElementKind getKind();

    /**
     * 返回此元素的修饰符,如下枚举值
     * PUBLIC, PROTECTED, PRIVATE, ABSTRACT, DEFAULT, STATIC, FINAL,
     * TRANSIENT, VOLATILE, SYNCHRONIZED, NATIVE, STRICTFP;
     */
    Set<Modifier> getModifiers();

    /**
     * 返回此元素的简单名称,例如
     * 类型元素 java.util.Set<E> 的简单名称是 "Set";
     * 如果此元素表示一个未指定的包，则返回一个空名称;
     * 如果它表示一个构造方法，则返回名称 "<init>";
     * 如果它表示一个静态初始化程序，则返回名称 "<clinit>";
     * 如果它表示一个匿名类或者实例初始化程序，则返回一个空名称
     */
    Name getSimpleName();

    /**
```

```
    * 返回封装此元素的最里层元素。
    * 如果此元素的声明在词法上直接封装在另一个元素的声明中，则返回那个封装元素；
    * 如果此元素是顶层类型，则返回它的包；
    * 如果此元素是一个包，则返回 null；
    * 如果此元素是一个泛型参数，则返回 null.
    */
   Element getEnclosingElement();

   /**
    * 返回此元素直接封装的子元素
    */
   List<? extends Element> getEnclosedElements();

   /**
    * 返回直接存在于此元素上的注解
    * 要获得继承的注解，可使用 getAllAnnotationMirrors
    */
   @Override
   List<? extends AnnotationMirror> getAnnotationMirrors();

   /**
    * 返回此元素针对指定类型的注解（如果存在这样的注解），否则返回 null。注解可以是继承的，也可以是直接存在于此元素上的
    */
   @Override
   <A extends Annotation> A getAnnotation(Class<A> annotationType);
}
```

Element 代表程序的元素，在注解处理过程中，编译器会扫描所有的 Java 源文件，并将源码中的每一个部分都看作特定类型的 Element。它可以代表包、类、接口、方法、字段等多种元素种类，具体看 getKind()方法中所指代的种类，每个 Element 代表一个静态的、语言级别的构件。

Element 有五个直接子接口，它们分别代表一种特定类型的元素，如下：

PackageElement	表示一个包程序元素
TypeElement	表示一个类或接口程序元素
VariableElement	表示一个字段、enum 常量、方法或构造方法参数、局部变量或异常参数
ExecutableElement	表示某个类或接口的方法、构造方法或初始化程序（静态或实例），包括注解类型元素
TypeParameterElement	表示一般类、接口、方法或构造方法元素的泛型参数

在开发中 Element 可根据实际情况强转为以上 5 种中的一种，它们都带有各自独有的方法，来看个简单的例子：

```
package com.example;       // PackageElement

import java.util.List;

public class Sample        // TypeElement
        <T extends List> {  // TypeParameterElement

    private int num;       // VariableElement
    String name;           // VariableElement
```

```
10.
11.   public Sample() {}      // ExecuteableElement
12.
13.   public void setName(    // ExecuteableElement
14.         String name     // VariableElement
15.         ) {}
16.}
```

源码中的每个部分都作为一个 Element，而 TypeElement 对应着一种更具体的类型元素。根据上面的表格可以知道，一种特定的元素一般不止指代一种元素种类(ElementKind)，比如 TypeElement 可以指代类或接口，要知道一个元素的准确的种类需要调用 getKind()方法，该方法返回一个 ElementKind 枚举值来表示具体种类，如下：

```
1.  public enum ElementKind {
2.
3.   /** A package. */
4.   PACKAGE,
5.   /** An enum type. */
6.   ENUM,
7.   /** A class not described by a more specific kind (like {@code ENUM}). */
8.   CLASS,
9.   /** An annotation type. */
10.  ANNOTATION_TYPE,
11.  /** An interface not described by a more specific kind */
12.  INTERFACE,
13.
14.  // Variables
15.  /** An enum constant. */
16.  ENUM_CONSTANT,
17.  /** A field not described by a more specific kind */
18.  FIELD,
19.  /** A parameter of a method or constructor. */
20.  PARAMETER,
21.  /** A local variable. */
22.  LOCAL_VARIABLE,
23.  /** A parameter of an exception handler. */
24.  EXCEPTION_PARAMETER,
25.
26.  // Executables
27.  /** A method. */
28.  METHOD,
29.  /** A constructor. */
30.  CONSTRUCTOR,
31.  /** A static initializer. */
32.  STATIC_INIT,
33.  /** An instance initializer. */
34.  INSTANCE_INIT,
35.  /** A type parameter. */
36.  TYPE_PARAMETER,
37.
38.  /** An implementation-reserved element. This is not the element you are looking for. */
39.  OTHER,
40.  /**
```

```
     * A resource variable.
     * @since 1.7
     */
    RESOURCE_VARIABLE;
}
```

上面管 ElementKind 称作元素的种类，因为它和元素的类型 TypeMirror 很容易混掉。TypeMirror 表示的是 Java 编程语言中的类型，比如上面例子中的字段 String name，它的元素种类为 FIELD，而它的元素类型为 DECLARED 表示一个类类型，这里对应 Java 编程语言中的类型为 java.lang.String。Element 代表的是源代码上的元素，TypeMirror 代表的是 Element 对应 Java 编程语言中的类型。

```
/**
 * 表示 Java 编程语言中的类型
 */
public interface TypeMirror {
    /**
     * 返回此类型的种类，一个 TypeKind 枚举值：
     */
    TypeKind getKind();
}
```

TypeMirror 和 Element 一样有一个 getKind()方法来获取具体的类型，方法返回一个枚举值，如下：

```
public enum TypeKind {
    /** The primitive type {@code boolean}. */
    BOOLEAN,
    /** The primitive type {@code byte}. */
    BYTE,
    /** The primitive type {@code short}. */
    SHORT,
    /** The primitive type {@code int}. */
    INT,
    /** The primitive type {@code long}. */
    LONG,
    /** The primitive type {@code char}. */
    CHAR,
    /** The primitive type {@code float}. */
    FLOAT,
    /** The primitive type {@code double}. */
    DOUBLE,
    /** The pseudo-type corresponding to the keyword {@code void}. */
    VOID,
    /** A pseudo-type used where no actual type is appropriate. */
    NONE,
    /** The null type. */
    NULL,
    /** An array type. */
    ARRAY,
    /** A class or interface type. */
    DECLARED,
    /** A class or interface type that could not be resolved. */
    ERROR,
    /** A type variable. */
    TYPEVAR,
```

```
    /** A wildcard type argument. */
    WILDCARD,
    /** A pseudo-type corresponding to a package element. */
    PACKAGE,
    /** A method, constructor, or initializer. */
    EXECUTABLE,
    /** An implementation-reserved type. This is not the type you are looking for. */
    OTHER,
    /** A union type. */
    UNION,
    /** An intersection type. */
    INTERSECTION;
}
```

可以看到和 ElementKind 所描述的是不同的方面。不知道这样说明的清不清楚，其实这种概念上的东西自己用几次会有更好理解，这东西就说到这。

Element 的直接子接口

这里列一下 5 个 Element 子接口常用方法，大部分描述是从 JDK PAI 手册中截取。这东西你也没必要一次看完，大概了解一下，等到需要用的时候能够知道有这么个东西就行了。

TypeElement

```
/**
 * 表示一个类或接口程序元素
 */
public interface TypeElement {

    /**
     * 返回此类型元素的嵌套种类
     * 某一类型元素的嵌套种类 (nesting kind)。类型元素的种类有四种：top-level（顶层）、member（成员）、local（局部）和 anonymous（匿名）
     */
    NestingKind getNestingKind();

    /**
     * 返回此类型元素的完全限定名称。更准确地说，返回规范 名称。对于没有规范名称的局部类和匿名类，返回一个空名称.
     * 一般类型的名称不包括对其形式类型参数的任何引用。例如，接口 java.util.Set<E> 的完全限定名称是 "java.util.Set"
     */
    Name getQualifiedName();

    /**
     * 返回此类型元素的直接超类。如果此类型元素表示一个接口或者类 java.lang.Object，则返回一个种类为 NONE 的 NoType
     */
    TypeMirror getSuperclass();

    /**
     * 返回直接由此类实现或直接由此接口扩展的接口类型
     */
    List<? extends TypeMirror> getInterfaces();
```

```
27.
28.     /**
29.      * 按照声明顺序返回此类型元素的形式类型参数
30.      */
31.     List<? extends TypeParameterElement> getTypeParameters();
32. }
```

PackageElement

```
1.  /**
2.   * 表示一个包程序元素.
3.   */
4.  public interface PackageElement {
5.
6.      /**
7.       * 返回此包的完全限定名称。该名称也是包的规范名称
8.       */
9.      Name getQualifiedName();
10.
11.     /**
12.      * 如果此包是一个未命名的包，则返回 true，否则返回 false
13.      */
14.     boolean isUnnamed();
15. }
```

ExecutableElement

```
1.  /**
2.   * 表示某个类或接口的方法、构造方法或初始化程序（静态或实例），包括注解类型元素
3.   */
4.  public interface ExecutableElement {
5.
6.      /**
7.       * 获取按照声明顺序返回形式类型参数元素
8.       */
9.      List<? extends TypeParameterElement> getTypeParameters();
10.
11.     /**
12.      * 获取返回的类型元素
13.      */
14.     TypeMirror getReturnType();
15.
16.     /**
17.      * 获取形参元素
18.      */
19.     List<? extends VariableElement> getParameters();
20.
21.     /**
22.      * 如果此方法或构造方法接受可变数量的参数，则返回 true，否则返回 false
23.      */
24.     boolean isVarArgs();
25.
26.     /**
27.      * 按声明顺序返回此方法或构造方法的 throws 子句中所列出的异常和其他 throwable
```

```
    */
    List<? extends TypeMirror> getThrownTypes();

    /**
     * 如果此 executable 是一个注解类型元素，则返回默认值。如果此方法不是注解类型元素，或者它是一个没有默认值的注解类型元素，则返回 null
     */
    AnnotationValue getDefaultValue();
}
```

VariableElement

```
/**
 * 表示一个字段、enum 常量、方法或构造方法参数、局部变量或异常参数
 */
public interface VariableElement {

    /**
     * 如果此变量是一个被初始化为编译时常量的 static final 字段，则返回此变量的值。否则返回 null。
     * 该值为基本类型或 String，如果该值为基本类型，则它被包装在适当的包装类中（比如 Integer）。
     * 注意，并非所有的 static final 字段都将具有常量值。特别是，enum 常量不 被认为是编译时常量。要获得一个常量值，字段的类型必须是基本类型或 String
     */
    Object getConstantValue();
}
```

TypeParameterElement

```
/**
 * 表示一般类、接口、方法或构造方法元素的泛型参数.
 */
public interface TypeParameterElement {

    /**
     * 返回由此类型参数参数化的一般类、接口、方法或构造方法
     */
    Element getGenericElement();

    /**
     * 返回此类型参数的边界。它们是用来声明此类型参数的 extends 子句所指定的类型。
     * 如果没有使用显式的 extends 子句，则认为 java.lang.Object 是唯一的边界
     */
    List<? extends TypeMirror> getBounds();
}
```

注解处理器的辅助接口

在自定义注解处理器的初始化接口，可以获取到以下 4 个辅助接口：

```
public class MyProcessor extends AbstractProcessor {

    private Types typeUtils;
    private Elements elementUtils;
    private Filer filer;
```

```
    private Messager messager;

    @Override
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        typeUtils = processingEnv.getTypeUtils();
        elementUtils = processingEnv.getElementUtils();
        filer = processingEnv.getFiler();
        messager = processingEnv.getMessager();
    }
}
```

其中 Filer 之前有用过，一般我们会用它配合 JavaPoet 来生成我们需要的.java 文件，这里就不再提它的用法。

Messager

Messager 提供给注解处理器一个报告错误、警告以及提示信息的途径。它不是注解处理器开发者的日志工具，而是用来写一些信息给使用此注解器的第三方开发者的。在官方文档中描述了消息的不同级别中非常重要的是 Kind.ERROR，因为这种类型的信息用来表示我们的注解处理器处理失败了。很有可能是第三方开发者错误的使用了注解。这个概念和传统的 Java 应用有点不一样，在传统 Java 应用中我们可能就抛出一个异常 Exception。如果你在 process()中抛出一个异常，那么运行注解处理器的 JVM 将会崩溃（就像其他 Java 应用一样），使用我们注解处理器第三方开发者将会从 javac 中得到非常难懂的出错信息，因为它包含注解处理器的堆栈跟踪（Stacktace）信息。因此，注解处理器就有一个 Messager 类，它能够打印非常优美的错误信息。除此之外，你还可以连接到出错的元素。在像现在的 IDE（集成开发环境）中，第三方开发者可以直接点击错误信息，IDE 将会直接跳转到第三方开发者项目的出错的源文件的相应的行。

看下接口代码：

```
public interface Messager {

    void printMessage(Diagnostic.Kind kind, CharSequence msg);

    void printMessage(Diagnostic.Kind kind, CharSequence msg, Element e);

    void printMessage(Diagnostic.Kind kind, CharSequence msg, Element e, AnnotationMirror a);

    void printMessage(Diagnostic.Kind kind, CharSequence msg, Element e, AnnotationMirror a,
    AnnotationValue v);
}
```

方法都比较好懂，主要需要指定打印的信息类型和描述字符串。

Types

Types 是一个用来处理 TypeMirror 的工具，看下代码就好了，提供的方法如下：

```
/**
 * 一个用来处理 TypeMirror 的工具
 */
public interface Types {
    /**
```

```
     * 返回对应于类型的元素。该类型可能是 DeclaredType 或 TypeVariable。如果该类型没有对应元素，则返回 null.
     */
    Element asElement(TypeMirror t);

    /**
     * 测试两个 TypeMirror 对象是否表示同一类型.
     * 警告：如果此方法两个参数中有一个表示通配符，那么此方法将返回 false
     */
    boolean isSameType(TypeMirror t1, TypeMirror t2);

    /**
     * 测试一种类型是否是另一个类型的子类型。任何类型都被认为是其本身的子类型.
     *
     * @return 当且仅当第一种类型是第二种类型的子类型时返回 true
     * @throws IllegalArgumentException 如果给定一个 executable 或 package 类型
     */
    boolean isSubtype(TypeMirror t1, TypeMirror t2);

    /**
     * 测试一种类型是否可以指派给另一种类型.
     *
     * @return 当且仅当第一种类型可以指派给第二种类型时返回 true
     * @throws IllegalArgumentException 如果给定一个 executable 或 package 类型
     */
    boolean isAssignable(TypeMirror t1, TypeMirror t2);

    /**
     * 测试一个类型参数是否包含 另一个类型参数.
     *
     * @return 当且仅当第一种类型包含第二种类型时返回 true
     * @throws IllegalArgumentException 如果给定一个 executable 或 package 类型
     */
    boolean contains(TypeMirror t1, TypeMirror t2);

    /**
     * 测试一个方法的签名是否是另一个方法的子签名.
     *
     * @return 当且仅当第一个签名是第二个签名的子签名时返回 true
     */
    boolean isSubsignature(ExecutableType m1, ExecutableType m2);

    /**
     * 返回类型的直接超类型。interface 类型将出现在列表的最后（如果有）.
     *
     * @return 直接超类型；如果没有，则返回一个空列表
     * @throws IllegalArgumentException 如果给定一个 executable 或 package 类型
     */
    List<? extends TypeMirror> directSupertypes(TypeMirror t);

    /**
     * 返回删除状态的类型.
```

```
57.     *
58.     * @return 删除状态的给定类型
59.     * @throws IllegalArgumentException 如果给定一个 package 类型
60.     */
61.    TypeMirror erasure(TypeMirror t);
62.
63.    /**
64.     * 返回给定基本类型的装箱 (boxed) 值类型的类。即应用 boxing 转换.
65.     *
66.     * @param p  要转换的基本类型
67.     * @return 类型 p 的装箱值类型的类
68.     */
69.    TypeElement boxedClass(PrimitiveType p);
70.
71.    /**
72.     * 返回给定类型的拆箱 (unboxed) 值类型（基本类型）。即应用 unboxing 转换.
73.     *
74.     * @param t  要拆箱的类型
75.     * @return 类型 t 的拆箱值类型
76.     * @throws IllegalArgumentException 如果给定类型无法进行 unboxing 转换
77.     */
78.    PrimitiveType unboxedType(TypeMirror t);
79.
80.    /**
81.     * 对类型应用 capture 转换.
82.     *
83.     * @return 应用 capture 转换的结果
84.     * @throws IllegalArgumentException 如果给定 executable 或 package 类型
85.     */
86.    TypeMirror capture(TypeMirror t);
87.
88.    /**
89.     * 返回基本类型.
90.     *
91.     * @param kind  要返回的基本类型的种类
92.     * @return 一个基本类型
93.     * @throws IllegalArgumentException 如果 kind 不是基本种类
94.     */
95.    PrimitiveType getPrimitiveType(TypeKind kind);
96.
97.    /**
98.     * 返回 null 类型。该类型是 null 的类型.
99.     */
100.     NullType getNullType();
101.
102.     /**
103.      * 返回在实际类型不适用的地方所使用的伪类型。
104.      * 要返回的类型的种类可以是 VOID 或 NONE。对于包，可以使用 Elements.getPackageElement(CharSequence).asType() 替代.
105.      *
106.      * @param kind  要返回的类型的种类
107.      * @return 种类 VOID 或 NONE 的伪类型
```

```
108.     * @throws IllegalArgumentException 如果 kind 无效
109.     */
110.    NoType getNoType(TypeKind kind);
111.
112.    /**
113.     * 返回具有指定组件类型的数组类型.
114.     *
115.     * @throws IllegalArgumentException 如果组件类型对于数组无效
116.     */
117.    ArrayType getArrayType(TypeMirror componentType);
118.
119.    /**
120.     * 返回新的通配符类型参数。可以指定通配符边界中的一个，也可以都不指定，但不能都指定.
121.     *
122.     * @param extendsBound  扩展（上）边界；如果没有，则该参数为 null
123.     * @param superBound    超（下）边界；如果没有，则该参数为 null
124.     * @return 新的通配符
125.     * @throws IllegalArgumentException 如果边界无效
126.     */
127.    WildcardType getWildcardType(TypeMirror extendsBound,
128.                                 TypeMirror superBound);
129.
130.    /**
131.     * 返回对应于类型元素和实际类型参数的类型。例如，如果给定 Set 的类型元素和 String 的类型镜像，
        那么可以使用此方法获取参数化类型 Set<String>.
132.     *
133.     * 类型参数的数量必须等于类型元素的形式类型参数的数量，或者等于 0。如果等于 0，并且类型元素是
        泛型，则返回该类型元素的原始类型.
134.     *
135.     * 如果返回一个参数化类型，则其类型元素不得包含在一般外部类中。
136.     * 例如，首先使用此方法获取类型 Outer<String>，然后调用 getDeclaredType(DeclaredType, Type
        Element, TypeMirror...)，
137.     * 可以构造参数化类型 Outer<String>.Inner<Number>.
138.     *
139.     * @param typeElem  类型元素
140.     * @param typeArgs  实际类型参数
141.     * @return 对应于类型元素和实际类型参数的类型
142.     * @throws IllegalArgumentException 如果给定的类型参数太多或太少，或者提供不合适的类型参数
        或类型元素
143.     */
144.    DeclaredType getDeclaredType(TypeElement typeElem, TypeMirror... typeArgs);
145.
146.    /**
147.     * 根据给定的包含类型，返回对应于类型元素和实际类型参数的类型（它是给定包含类型的成员）.例子如
        上
148.     * 如果包含类型是一个参数化类型，则类型参数的数量必须等于 typeElem 的形式类型参数的数量。
149.     * 如果包含类型不是参数化的，或者为 null，则此方法等效于 getDeclaredType(typeElem, typeArg
        s）.
150.     *
151.     * @param containing  包含类型；如果没有，则该参数为 null
152.     * @param typeElem  类型元素
153.     * @param typeArgs  实际类型参数
```

```
     * @return 对应于类型元素和实际类型参数的类型，该类型包含在给定类型中
     * @throws IllegalArgumentException 如果给定的类型参数太多或太少，或者提供了不合适的类型参数、类型元素或包含类型
     */
    DeclaredType getDeclaredType(DeclaredType containing,
                                 TypeElement typeElem, TypeMirror... typeArgs);

    /**
     * 当元素被视为给定类型的成员或者直接由给定类型包含时，返回该元素的类型。
     * 例如，被视为参数化类型 Set<String> 的成员时，Set.add 方法是参数类型为 String 的 ExecutableType.
     *
     * @param containing  包含类型
     * @param element     元素
     * @return 从包含类型来看的元素的类型
     * @throws IllegalArgumentException 如果元素对于给定类型无效
     */
    TypeMirror asMemberOf(DeclaredType containing, Element element);
}
```

Elements

Elements 是一个用来处理 Element 的工具，提供的方法如下：

```
/**
 * 一个用来处理 Element 的工具
 */
public interface Elements {

    /**
     * 返回已给出其完全限定名称的包.
     *
     * @param name  完全限定的包名称；对于未命名的包，该参数为 ""
     * @return 指定的包；如果没有找到这样的包，则返回 null
     */
    PackageElement getPackageElement(CharSequence name);

    /**
     * 返回已给出其规范名称的类型元素.
     *
     * @param name  规范名称
     * @return 指定的类型元素；如果没有找到这样的元素，则返回 null
     */
    TypeElement getTypeElement(CharSequence name);

    /**
     * 返回注释元素的值，包括默认值.
     * 此值是以映射的形式返回的，该映射将元素与其相应的值关联。只包括那些注释中明确存在其值的元素，不包括那些隐式假定其默认值的元素。
     * 映射的顺序与值出现在注释源中的顺序匹配
     *
     * @see AnnotationMirror#getElementValues()
     * @param a  要检查的注释
```

```
29.     * @return 注释元素的值，包括默认值
30.     */
31.    Map<? extends ExecutableElement, ? extends AnnotationValue>
32.    getElementValuesWithDefaults(AnnotationMirror a);
33.
34.    /**
35.     * 返回元素的文档（"Javadoc"）注释文本
36.     *
37.     * @param e  将被检查的元素
38.     * @return 元素的文档注释；如果没有，则返回 null
39.     */
40.    String getDocComment(Element e);
41.
42.    /**
43.     * 如果元素已过时，则返回 true，否则返回 false.
44.     *
45.     * @param e  将被检查的元素
46.     * @return 如果元素已过时，则返回 true，否则返回 false
47.     */
48.    boolean isDeprecated(Element e);
49.
50.    /**
51.     * 返回类型元素的二进制名称.
52.     *
53.     * @param type  将被检查的类型元素
54.     * @return 二进制名称
55.     */
56.    Name getBinaryName(TypeElement type);
57.
58.    /**
59.     * 返回元素的包。包的包是它本身.
60.     *
61.     * @param type 将被检查的元素
62.     * @return 元素的包
63.     */
64.    PackageElement getPackageOf(Element type);
65.
66.    /**
67.     * 返回类型元素的所有成员，不管是继承的还是直接声明的。对于一个类，结果还包括其构造方法，但不包
       括局部类或匿名类.
68.     *
69.     * 注意，使用 ElementFilter 中的方法可以隔离某个种类的元素.
70.     *
71.     * @param type  将被检查的类型
72.     * @return 类型的所有成员
73.     */
74.    List<? extends Element> getAllMembers(TypeElement type);
75.
76.    /**
77.     * 返回元素的所有注释，不管是继承的还是直接存在的.
78.     *
79.     * @param e  将被检查的元素
```

```
80.     * @return 元素的所有注释
81.     */
82.    List<? extends AnnotationMirror> getAllAnnotationMirrors(Element e);
83.
84.    /**
85.     * 测试一个类型、方法或字段是否隐藏了另一个类型、方法或字段.
86.     *
87.     * @param hider   第一个元素
88.     * @param hidden  第二个元素
89.     * @return 当且仅当第一个元素隐藏了第二个元素时返回 true
90.     */
91.    boolean hides(Element hider, Element hidden);
92.
93.    /**
94.     * 测试一个方法（作为给定类型的成员）是否重写了另一个方法。当非抽象方法重写抽象方法时，还可以说
       成是前者实现 了后者.
95.     *
96.     * @param overrider  第一个方法，可能是 overrider
97.     * @param overridden  第二个方法，可能被重写
98.     * @param type   第一个方法是其成员的类型
99.     * @return 当且仅当第一个方法重写第二个方法时返回 true
100.     */
101.    boolean overrides(ExecutableElement overrider, ExecutableElement overridden,
102.                      TypeElement type);
103.
104.    /**
105.     * 返回表示基本值或字符串的常量表达式 文本。返回文本的形式是一种适合于表示源代码中的值的形式.
106.     *
107.     * @param value  基本值或字符串
108.     * @return 常量表达式的文本
109.     * @throws IllegalArgumentException 如果参数不是基本值或字符串
110.     *
111.     * @see VariableElement#getConstantValue()
112.     */
113.    String getConstantExpression(Object value);
114.
115.    /**
116.     * 按指定顺序将元素的表示形式打印到给定 writer。此方法的主要用途是诊断。输出的具体格式没有 指定
       并且是可以更改的.
117.     *
118.     * @param w 输出打印到的 writer
119.     * @param elements 要打印的元素
120.     */
121.    void printElements(java.io.Writer w, Element... elements);
122.
123.    /**
124.     * 返回与参数具有相同字符序列的名称.
125.     *
126.     * @param cs 将以名称形式返回的字符序列
127.     * @return 返回与参数具有相同字符序列的名称
128.     */
129.    Name getName(CharSequence cs);
```

```
130.
131.    /**
132.     * 如果类型是一个泛型接口则返回 true，否则返回 false
133.     *
134.     * @param type 将被检查的类型
135.     * @return 如果类型是一个泛型接口则返回 true，否则返回 false
136.     * @since 1.8
137.     */
138.    boolean isFunctionalInterface(TypeElement type);
139.}
```