抛砖引玉——自定义Lint

为什么要用它?

人会犯错，机器不会。这就很需要用机器帮我们规避一些问题，构建更健壮的代码。

自定义Detector

首先来区分一下**Issues vs Detectors**这两个概念。Issue 代表您想要发现并提示给开发者的一种问题，包含描述、更全面的解释、类型和优先级等等。官方提供了一个 Issue 类，您只需要实例化一个 Issue，并注册到 IssueRegistry 里。

另外您还需要实现一个 Detector。Detector 负责扫描代码并找到有问题的地方，然后把它们报告出来。一个 Detector 可以报告多种类型的 Issue，您可以针对不同类型的问题使用不同的严重程度，这样用户可以更精确地控制他们想要看到的内容。

以创建一个检查实现序列化的Detector为例：

```
/**
 * @author : LeeZhaoXing
 * @date   : 2021/1/11
 * @desc   : 检查Bean序列化
 */
class SerializableClassDetector  : Detector() ,Detector.UastScanner{
    companion object {
        private const val REPORT_MESSAGE = "该对象必须要实现Serializable接口，因为外部类
实现了Serializable接口"
        private const val CLASS_SERIALIZABLE = "java.io.Serializable"
        val ISSUE = Issue.create(
            "SerializableClassCheck",
            REPORT_MESSAGE,
            REPORT_MESSAGE,
            Category.CORRECTNESS,
            10,
            Severity.ERROR,
            Implementation(SerializableClassDetector::class.java,
Scope.JAVA_FILE_SCOPE)
        )
    }

    override fun applicableSuperClasses(): List<String>? {
        return listOf(CLASS_SERIALIZABLE)
```

```
        }

        override fun visitClass(context: JavaContext, declaration: UClass) {
            for (field in declaration.fields) {
                //字段是引用类型，并且可以拿到该class
                val psiClass = (field.type as? PsiClassType)?.resolve() ?: continue
                if (!context.evaluator.implementsInterface(psiClass, CLASS_SERIALIZABLE,
    true)) {
                    context.report(ISSUE, context.getLocation(field.typeReference!!),
    REPORT_MESSAGE)
                }
            }
        }
    }
```

先看Issue.create()方法，其参数定义如下：

1. **id**: 唯一的 id，简要表达当前问题。
2. **briefDescription**: 简单描述当前问题。
3. **explanation**: 详细解释当前问题和修复建议。
4. **category**: 问题类别，在 Android 中主要有如下六大类:
   - SECURITY: 安全性。例如在 AndroidManifest.xml 中没有配置相关权限等。
   - USABILITY: 易用性。例如重复图标，一些黄色警告等。
   - PERFORMANCE: 性能。例如内存泄漏，xml 结构冗余等。
   - CORRECTNESS: 正确性。例如超版本调用 API，设置不正确的属性值等。
   - A11Y: 无障碍 (Accessibility)。例如单词拼写错误等。
   - I18N: 国际化 (Internationalization)。例如字符串缺少翻译等。
5. **priority**: 优先级，从 1 到 10，10 最重要。
6. **severity**: 严重程度，包括 FATAL、ERROR、WARNING、INFORMATIONAL 和 IGNORE。
7. **implementation**: Issue 和哪个 Detector 绑定，以及声明检查的范围。Scope的枚举如下：
   - RESOURCE_FILE（资源文件)
   - BINARY_RESOURCE_FILE（二进制资源文件）
   - RESOURCE_FOLDER（资源文件夹）
   - ALL_RESOURCE_FILES（所有资源文件）
   - JAVA_FILE（Java文件）
   - ALL_JAVA_FILES（所有Java文件）
   - CLASS_FILE（class文件）
   - ALL_CLASS_FILES（所有class文件）
   - MANIFEST（配置清单文件）
   - PROGUARD_FILE（混淆文件）
   - JAVA_LIBRARIES（Java库）
   - GRADLE_FILE（Gradle文件）

- PROPERTY_FILE(属性文件)
- TEST_SOURCES（测试资源）
- OTHER(其他)

实现Scan接口

扫描并发现代码中的Issue,自定义 Detector 还需要实现一个或多个以下接口:

- UastScanner: 扫描 Java 文件和 Kotlin 文件
- ClassScanner: 扫描 Class 文件
- XmlScanner: 扫描 XML 文件
- ResourceFolderScanner: 扫描资源文件夹
- BinaryResourceScanner: 扫描二进制资源文件
- OtherFileScanner: 扫描其他文件
- GradleScanner: 扫描 Gradle 脚本

因为我们要扫描的是Java/Koltin文件，所以实现UastScanner接口。

自定义Register

用于注册要检查的Issue(规则)，只有注册了Issue,该Issue才能被使用。例如注册上文的实现序列化规范规则。

```
    class CustomIssueRegistry :  IssueRegistry() {
        override val issues: List<Issue>
            get() = listOf(
                SerializableClassDetector.ISSUE
            )

        override val api: Int
            get() = CURRENT_API

    }
```

Lint Debug

首先看下依赖：

```
val lintVersion = "27.1.1"

// Lint
compileOnly("com.android.tools.lint:lint-api:${lintVersion}")
compileOnly("com.android.tools.lint:lint-checks:${lintVersion}")

// Lint testing
testImplementation("com.android.tools.lint:lint:${lintVersion}")
testImplementation("com.android.tools.lint:lint-tests:${lintVersion}")
```

然后在test文件夹下编写测试代码（java目录下无法导入测试的api）：

```
@Suppress("UnstableApiUsage")
class ExampleUnitTest {
    @Test
    fun test() {
        lint()
            .allowMissingSdk()
            .files(
                TestFiles.kotlin(
                    """
                        package me.dawn.lintcheck
                        import java.io.Serializable

                        class SerializableBean : Serializable {
                            private var serializableField: InnerSerializableBean? =
null
                        }

                        class InnerSerializableBean : Serializable {
                            private var commonBean: CommonBean? = null
                        }

                        class CommonBean{
                            private var s: String = "abc"
                        }
                    """
                )
            )
            .issues(SerializableClassDetector.ISSUE)
            .run()
            .expect(
                "No warnings."
```

```
            )
        }
    }
```

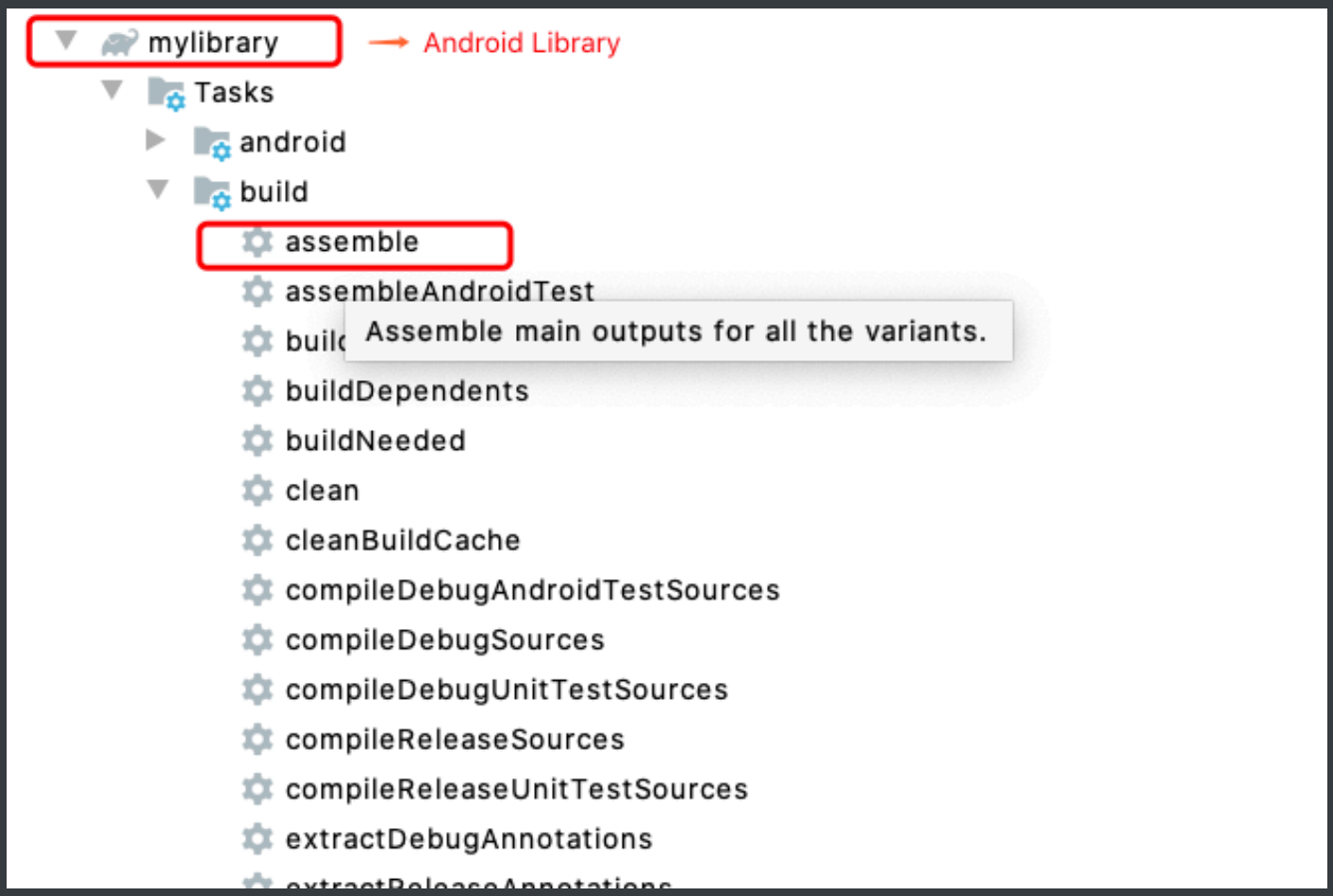
在执行测试代码的时候编译器报错，添加**.allowMissingSdk( )**交保护费。

打包成AAR

新建Android Library，添加gradle配置：

```
    dependencies {
        ...
        lintPublish project(path: ':lib_lint_check')
    }
```

执行assemble任务成功后,在**myibrary->build->outputs->aar**文件夹就能得到AAR文件：mylibrary-release.aar



编码实时检查

新建一个Android项目，使用上面自定义的Lint规则

```
    repositories {
        flatDir {
            dirs 'libs'
        }
        jcenter()
    }

    dependencies {
        ...
        api files('libs/mylibrary-release.aar')
    }
```

故意写一个错误的对象，就能看到IDE给出的提示。

编译时检查

配置Gradle脚本即可实现编译时检查，好处是每次编译时可以进行检查及时发现错误，坏处是会拖慢编译速度。

编译Android需要执行assemble任务，我们只需要使assemble依赖上lint任务即可在每次编译的时候进行lint检查

```
    android.applicationVariants.all { variant ->
        variant.outputs.each { output ->
            def lintTask = tasks["lint${variant.name.capitalize()}"]
            output.assemble.dependsOn lintTask
        }
    }
```

LintOption加上配置

```
android.lintOptions {
    abortOnError true
}
```