Google I/O Extended

Kotlin Symbol Processor 技巧与实战

by 2BAB

June 2022

Android Kotlin Build/Tools

Google I/O Extended

2BAB

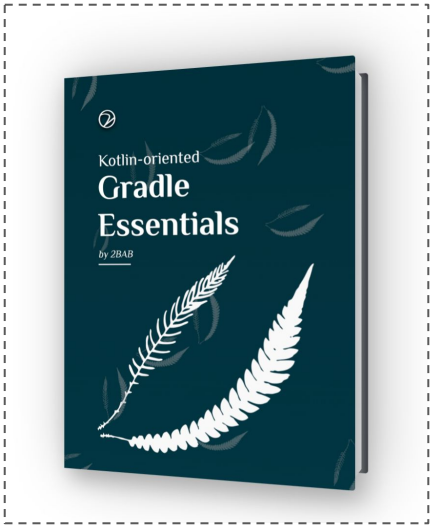
Android 高级工程师
关注基础架构、编译构建

xx2bab@gmail.com

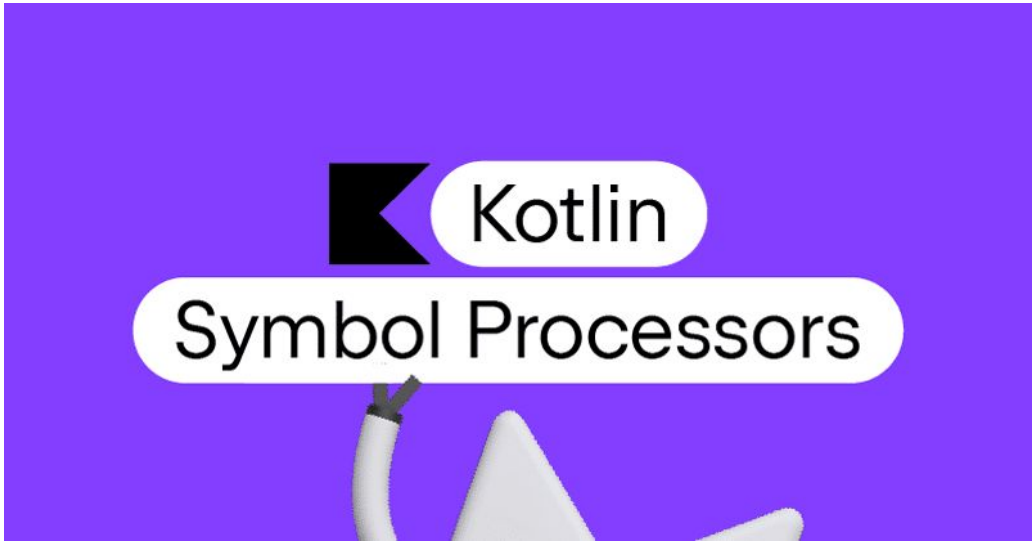
github.com/2BAB/

2bab.me/

binary.2bab.me/



Google I/O Extended



03

实战案例

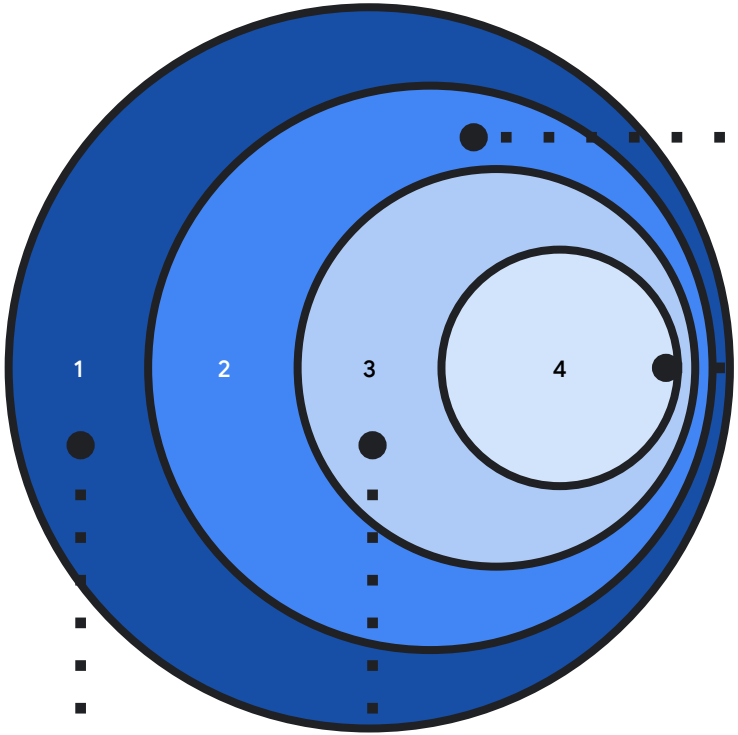
除了常规的 Annotation Processor 用例，还有什么可能？

Google I/O Extended

回顾

从 Alpha 到 1.0.x

KSP 基础概念



fun process(resolver)
fun finish()

File Visitor
Code Generator

SymbolProcessorProvider
SymbolProcessor

resolver.getSymbolsWithAnnotation()

前人指路

01

Kotlin 编译器插件：我们究竟在期待什么？

by BennyHuo

深入浅出讲解 KSP 的由来，对比 KAPT 的性能优势，在元编程中扮演的角色等。

https://www.bilibili.com/video/BV1Tf4y157ku

https://www.bilibili.com/video/BV1JY411H7pb

02

Kotlin 元编程：从注解处理器 KAPT到符号处理器 KSP

by BennyHuo

03

google/ksp/examples

涵盖多个完善的基 础 API 使用 Demo。

https://github.com/google/ksp/tree/main/examples

一年的变化

1. 紧跟 Kotlin 和 Gradle 的最新版本，拥有良好的兼容性。
2. 引入多平台支持，透出更多平台相关信息的 给开发者。
3. 修复超过 100 个 bug，稳定性提升显著，包括 Java 混编模块的相关细节问题。
4. 更细化的增量编译支持，稳步提高性能。
5. 增强了 Symbol 之间的联系，模块之间的联系，获取父节点声明、子模块依赖的声明等更加容易。

开源库支持

Library	Status
Room	Officially supported ↗
Moshi	Officially supported ↗
RxHttp	Officially supported ↗
Kotshi	Officially supported ↗
Lyricist	Officially supported ↗

Library	Status
Lich SavedState	Officially supported ↗
gRPC Dekorator	Officially supported ↗
EasyAdapter	Officially supported ↗
Koin Annotations	Officially supported ↗

Google I/O Extended

进阶

KSP 实用技巧

1. 复杂传参

自定义包名

不少基于 KSP 的处理器允许用户自定义生成的类包名（尤其是 1:1 生成的辅助类），方便后续的管理。而该包名参数需要从外部（Gradle 脚本）传输到处理器内部。

```
ksp {
  arg("key1", "value1")
  arg("packageName", "com.exam")
  ...
}
```

1. 复杂传参

参数若是文件..

倘若传递的参数较为复杂，例如传入 *.json / *.properties 的文件作为入参，或指定一个 *.log 类型的文件作为输出。

```
ksp {
  arg("procA.input", "in.json")
  arg("procB.output", "out.log")
  ...
}
```

应用错误的缓存

1. 复杂传参

思考

在 Gradle 环境内出现的问题，可以用 Gradle 的方式解决。最直观的办法是基于 KSP 的 Gradle Task 进行修改，增加额外的输入文件。

```
project.tasks.withType<com.google.devtools.ksp.gradle.KspTask> {
    inputs.files(project.layout
        .projectDirectory
        .file("local.properties")
    )
}
```

1. 复杂传参

能否再优雅一点..

新版本（1.0.5）的 KSP 实际上提供了一个特别的 `arg(CommandLineArgumentProvider)` API，从配置 DSL 的角度切入，提供耦合度更低、更稳定的文件传参体验。

https://github.com/google/ksp/pull/872/files

```
class InOutFilesProvider(
    @InputFile
    @PathSensitive(PathSensitivity.RELATIVE)
    val localProperties: File
) : CommandLineArgumentProvider {
    override fun asArguments(): Iterable<String> {
        return listOf("customProc.localProperties=${localProperties.path}")
    }
}
...
ksp {
    arg(InOutFilesProvider(rootProject.layout
        .projectDirectory.file("local.properties").asFile))
}
```

1. 复杂传参

修改文件后..

Task ':app:kspDebugKotlin' is not up-to-date because:

Input property 'commandLineArgumentProviders.$0.localProperties'
file /Users/2bab/Desktop/Koncat/sample/**local.properties has changed.**

The input changes require a **full rebuild** for incremental task ':app:kspDebugKotlin'.

1. 复杂传参

However, when using Android Gradle plugin 3.2.0 and higher, you need to pass processor arguments that represent files or directories using Gradle's `CommandLineArgumentProvider` interface.

Using `CommandLineArgumentProvider` allows you or the annotation processor author to improve the correctness and performance of incremental and cached clean builds by applying incremental build property type annotations to each argument.

For example, the class below implements `CommandLineArgumentProvider` and annotates each argument for the processor. The sample also uses the Groovy language syntax and is included directly in the module's `build.gradle` file.

https://developer.android.com/studio/build/dependencies

2. Variant 感知

添加 Debug 信息

利用 KSP 生成代码时，有时我们希望在 Debug 环境下插入一些额外的日志信息，或加入额外的调试代码；同时又不在 Pre-release，Release 环境中出现。我们需要一个开关来控制此类行为。

```
val isDebug = gradle.startParameter
        .taskRequests.toString()
        .contains("debug").toString()
ksp {
    arg("logEnabled", isDebug)
}
```

非有效策略

https://2bab.me/2021/12/21/enable-feature-by-variant

```
ksp {
    arg("logEnabled.debug", "true")
    arg("logEnabled.preRelease", "false")
    arg("logEnabled.release", "false")
}
```

2. Variant 感知

思考

实际上前面的方案只解决了一部分的问题：到了 KSP 处理器内部如何判断当前的 Variant 从而获取对应的配置？如果参数太多怎么办？

```
// Either
env.options["logEnabled.debug"]
// Or
env.options["logEnabled.release"]
```

```
class VariantAwareness(env: SymbolProcessorEnvironment) {

    val variantName: String

    init {
        val resourcesDir = CodeGeneratorImpl::class.memberProperties
            .first { it.name == "resourcesDir" }
            .also { it.isAccessible = true }
            .getter(env.codeGenerator as CodeGeneratorImpl)
            .toString()
        variantName = File(resourcesDir).parentFile.name
    }
    ...
}
```

https://github.com/google/ksp/issues/861

结合“文件传参”

```
val configPath = env.options["configFile.${awareness.variantName}"]
val config = Json.decodeFromString<Config>(File(configPath))
```

3. 不止于 AP

标记/符号

No expression. Symbol/Declaration only.

Kotlin Symbol Processor 名字很直观，其核心步骤之一是解析 Kotlin 源码构建出如右的声明结构，也因此它能做的事情实际上不止于 Annotation Processor。（当然，APT，AP，KAPT 等名字挂着 Annotation 的“前辈”，实际上也提供了类似的功能，只是不如 KSP 来的直接。

```
KSFile
  packageName: KSName
  fileName: String
  annotations: List<KSAnnotation>  (File annotations)
  declarations: List<KSDeclaration>
    KSClassDeclaration // class, interface, object
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      classKind: ClassKind
      primaryConstructor: KSFunctionDeclaration
      superTypes: List<KSTypeReference>
      declarations: List<KSDeclaration>
    KSFunctionDeclaration // top level function
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      …
    KSPropertyDeclaration // global variable
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      …
```

```
class ExtensionProcessor(...) : SymbolProcessor {

    override fun process(resolver: Resolver): List<KSAnnotated> {
//        resolver.getSymbolsWithAnnotation("com.example.Anno")
//            .forEach { ksAnnotated: KSAnnotated ->
//
//        }
        resolver.getNewFiles().forEach { ksFile: KSFile ->
            ksFile.accept(visitor, data)
        }
        return emptyList()
    }

}
```

```kotlin
override fun visitClassDeclaration(classDeclaration: KSClassDeclaration) {
    val className = classDeclaration.qualifiedName?.asString()
    if (className.isNullOrBlank()) { // Anonymous class is not supported
        return
    }
    classDeclaration.superTypes.forEach { superType ->
        val superKSType = superType.resolve()
        if (superKSType.toClassName().canonicalName == "kotlin.Any") {
            return@forEach
        }
        targetInterfaces.forEach { targetInterface ->
            if (superKSType.isAssignableFrom(targetInterface.type)) {
                ...
            }
        }
    }
}
```

4. 与依赖交互

探索依赖

`Resolver#getSymbolsWithAnnotation()` 或 Resolver#getNewFiles()，处理的范围都仅限于**本模块**的源码。但这无法解决一些**多模块**的场景，例如在 Application 模块收集、注册所有模块中被打上注解 `@ExportService` 的类。

```
// App 模块
fun onCreate() {
     registerServices(
         ServiceFromModuleA(),
         ServiceFromModuleB()
     )
}

// 其他模块

@ExportService
Class ServiceFromModuleA

@ExportService
Class ServiceFromModuleB
```

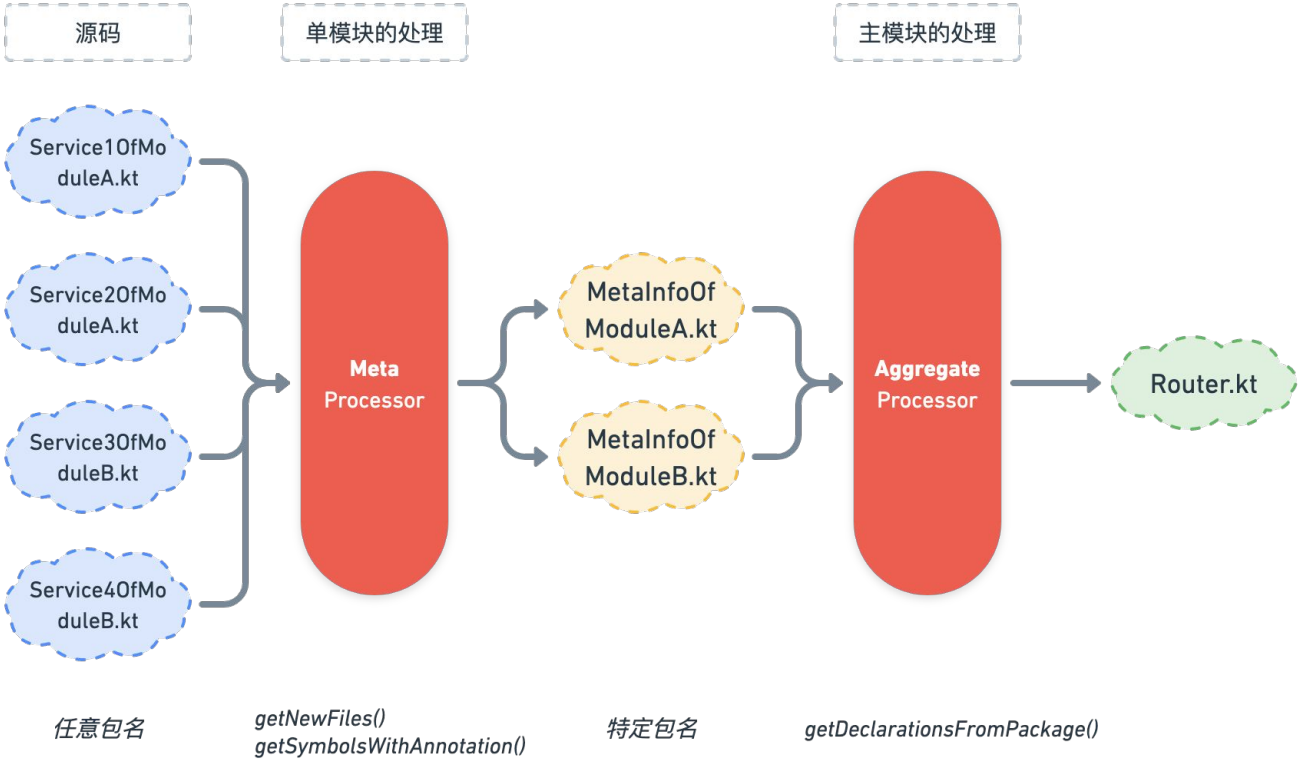
```
interface Resolver {
    fun getKSNameFromString(name: String): KSName

    fun getClassDeclarationByName(name: KSName): KSClassDeclaration?

    fun getFunctionDeclarationsByName(name: KSName, includeTopLevel:
Boolean = false): Sequence<KSFunctionDeclaration>

    fun getPropertyDeclarationByName(name: KSName, includeTopLevel:
Boolean = false): KSPropertyDeclaration?

    fun getDeclarationsFromPackage(packageName: String):
Sequence<KSDeclaration>
}
```

https://github.com/google/ksp/issues/344

Google I/O Extended

实战

利用 Koncat 生成路由表



Koncat：路由表生成与扩展

build.gradle.kts

```
koncat {
    annotations.addAll("me.xx2bab.koncat.sample.anno.ExportActivity")
    classTypes.addAll("me.xx2bab.koncat.sample.interfaze.DummyAPI")
    propertyTypes.addAll("org.koin.core.module.Module")
}
```

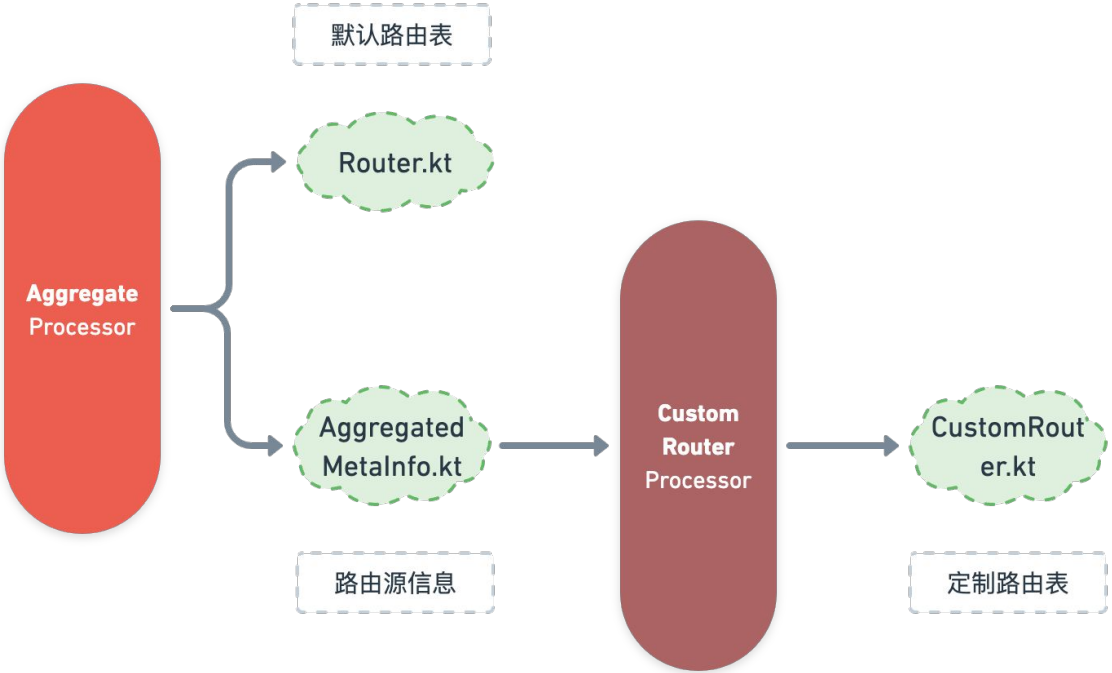
Router.kt

```
val annotatedClasses = mapOf<KClass<out Annotation>, List<ClassDeclarationRecord>>(...)
val interfaceImplementations = mapOf<KClass<*>, Any>(...)
val typedProperties = mapOf<KClass<*>, Any>(...)
```

Koncat：路由表生成与扩展

思考

Router 类是一个通用实现，无法满足不同开发者细枝末节的要求。
1. 开发者可能不满足 Router 收集的**元信息**；
2. 即便满足，其**数据结构**也未必是开发者所需要的；
能否提供**编译期的扩展方案**给第三方开发者，一步到位生成**定制的路由类**？



核心：利用 KSP 多轮处理的特性实现扩展

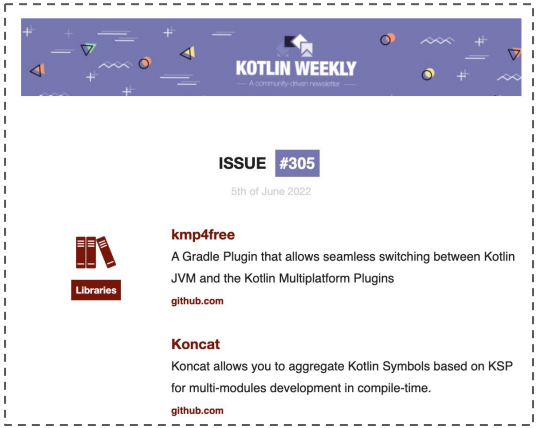
```
// AggregatedMetaInfo.kt

@KoncatExtend(metaDataInJson = """{
   "annotatedClasses":{
      "me.xx2bab.koncat.sample.annotation.ExportActivity":[
         {
            "name":"me.xx2bab.koncat.sample.android.AndroidLibraryActivity",
            "Annotations":[ … ]
         },
         …
      ]
   },
   "typedClasses":{
      "me.xx2bab.koncat.sample.interfaze.DummyAPI":[
         "me.xx2bab.koncat.sample.android.ExternalAndroidLibraryAPI",
         …
      ]
   },
   "typedProperties":{ … }
}
""")
val voidProp = null // DO NOT use voidProp directly
```

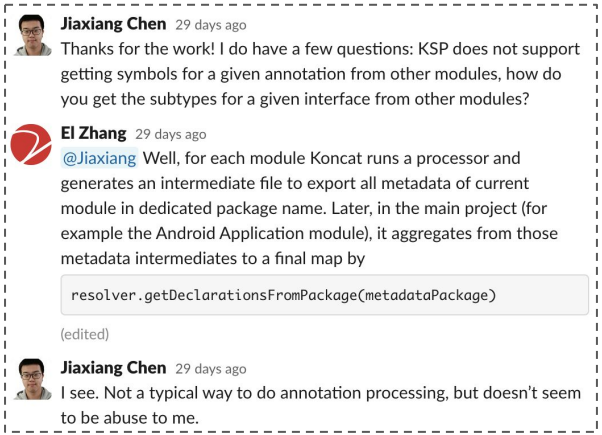
```kotlin
class ExtensionProcessor(private val koncat: KoncatProcAPI, …): SymbolProcessor {

    private var holder: KoncatProcMetadataHolder? = null

    override fun process(resolver: Resolver): List<KSAnnotated> {
        holder = koncat.syncAggregatedMetadata(resolver)
        return emptyList()
    }

    override fun finish {
        super.finish()
        holder?.apply {
            val fileSpec = RouterClassBuilder(resolve()).build()
            fileSpec.writeTo(codeGenerator, Dependencies(false, dependency))
        }
    }

    inner class RouterClassBuilder(private val data: KoncatProcMetadata) {
       …
    }
}
```

Koncat：路由表生成与扩展

https://github.com/2BAB/Koncat

不需要反射或字节码修改，也可以在编译期的源码阶段实现多模块的标记收集，路由表生成等。例如获取一个应用里某个接口的所有实现。





Google I/O Extended

Thank you!

2BAB

Android 高级工程师
关注基础架构、编译构建

xx2bab@gmail.com

github.com/2BAB/

2bab.me/

binary.2bab.me/