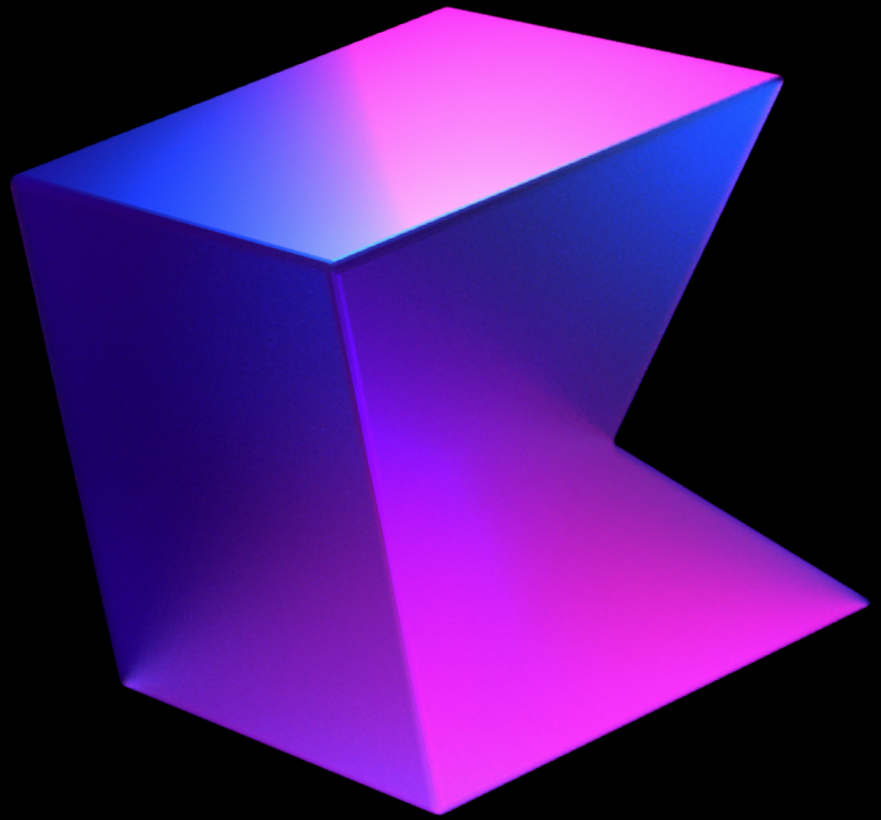
Kotlin

KSP
应用与技巧



2BAB

xx2bab@gmail.com

github.com/2BAB

@xx2bab

November 26, 2022

自我介绍 - 2BAB

关注基础架构、编译构建

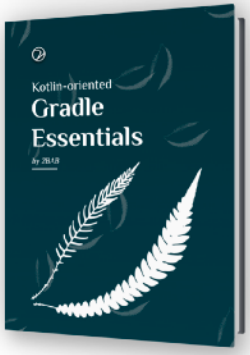
Android GDE（Google Developer Expert）

《Android 构建与架构实战》作者

《Kotlin oriented Gradle Essential》 作者

《二分电台》Podcast 主理人

https://2bab.me/about



Kotlin

什么是 KSP

Kotlin Symbol Processing（KSP）是一种**元编程**（Meta Programming）工具，基于 Kotlin Compiler Plugin（KCP）的源码分析能力。开发者可使用它编写 Processor 获取源码相关**符号**、**描述**等，还可基于描述信息进一步**生成新的代码**。

Kotlin

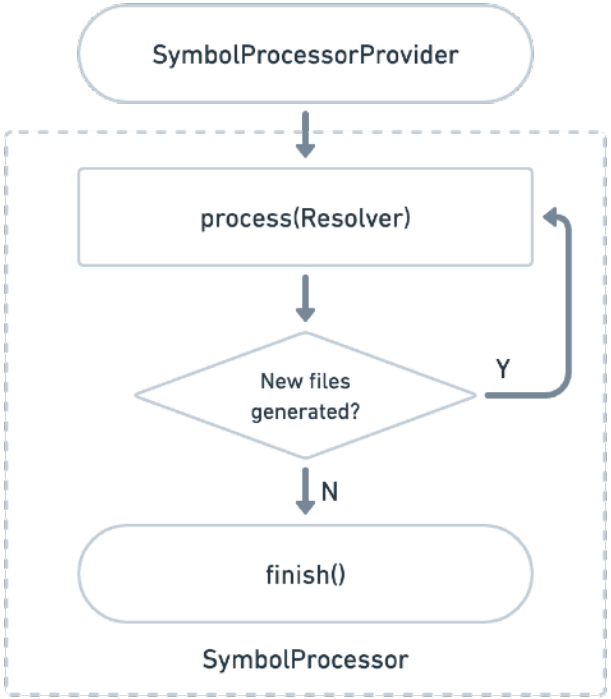
什么是 KSP

1. Kotlin Symbol
2. Processing

Kotlin

KSFile 结构 & KSP 主流程

```
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
      …
    KSPropertyDeclaration // global variable
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      …
```



Kotlin

核心场景：
Annotation Processor

SQLin 生成示例

```
// Source:
@DBRow(“person”)
data class Person(
    val age: Int,
    val name: String,
)

// KSP generated:
object PersonTable : Table<Person>(“person”) {
    val name: ClauseString get() {…}
    val age: ClauseNumber get() {…}
    var SetClause<Person>.name: String set(value) {…}
    var SetClause<Person>.age: Int set(value) {…}
}
```

Kotlin

核心代码

```
class ClauseProcessorProvider: SymbolProcessorProvider {

    override fun create(env: SymbolProcessorEnvironment): SymbolProcessor =
        ClauseProcessor(env.codeGenerator)
}

class ClauseProcessor(
    private val codeGenerator: CodeGenerator,
) : SymbolProcessor {
    override fun process(resolver: Resolver): List<KSAnnotated> {
        ..
    }
}
```

Kotlin

核心代码

```
override fun process(resolver: Resolver): List<KSAnnotated> {
    val allClassAnnotatedWhereProperty = resolver.getSymbolsWithAnnotation(…)  ①
        as Sequence<KSClassDeclaration>
    …
    for (classDeclaration in allClassAnnotatedWhereProperty) {
        if (classDeclaration.getAllSuperTypes().all { !it.isAssignableFrom(dbEntityType) }  ②
            || classDeclaration.annotations.all {
                !it.annotationType.resolve().isAssignableFrom(serializableType) }) {
            continue
        }
        val className = classDeclaration.simpleName.asString()
        val packageName = classDeclaration.packageName.asString()
        val objectName = "${className}Table"
③       val tableName = classDeclaration.annotations.find {
            it.annotationType.resolve().declaration.qualifiedName?.asString() == DB_ROW_NAME
        }?.arguments?.first()?.value?.takeIf { (it as? String)?.isNotBlank() == true } ?: className
        val outputStream = codeGenerator.createNewFile(…)  ④
        …
    }
}
```

Kotlin

更多开源库

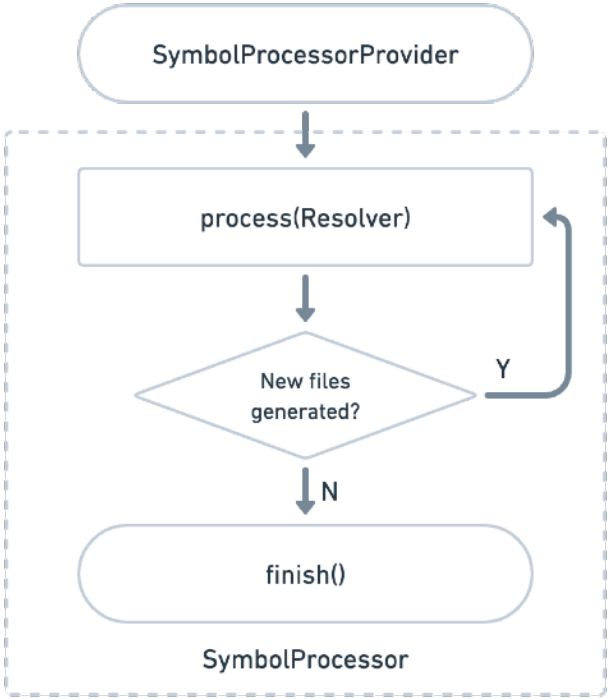
Library	Status
Room	Officially supported ↗
Moshi	Officially supported ↗
RxHttp	Officially supported ↗
Kotshi	Officially supported ↗
Lyricist	Officially supported ↗
Lich SavedState	Officially supported ↗
gRPC Dekorator	Officially supported ↗
EasyAdapter	Officially supported ↗
Koin Annotations	Officially supported ↗

Kotlin

Kotlin Symbol
边界扩展

KSFile 结构和运行流程

```
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
      …
    KSPropertyDeclaration // global variable
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      …
```



Kotlin

探索 getSymbolsWithAnnotation(…)

```
override fun getSymbolsWithAnnotation(..): Sequence<KSAnnotated> {
    val realAnnotationName =
        aliasingFqNs[annotationName]?.type
            ?.resolveToUnderlying()
            ?.declaration
            ?.qualifiedName
            ?.asString()
            ?: annotationName

    val ksName = KSNameImpl.getCached(realAnnotationName)
    val shortName = ksName.getShortName()
    ..
    val allAnnotated = if (inDepth) newAnnotatedSymbolsWithLocals else newAnnotatedSymbols
    return allAnnotated.asSequence().filter(::checkAnnotated)
}
```

Kotlin

探索 getSymbolsWithAnnotation(…)

```
private fun collectAnnotatedSymbols(...): Collection<KSAnnotated> {
    val symbols = arrayListOf<KSAnnotated>()
    val visitor = object : KSVisitorVoid() {...}
        for (file in newKSFiles) {
        file.accept(visitor, Unit)
    }
    return symbols
}
```

Kotlin

直接使用 getNewFiles()

```
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

自定义过滤器 - 以 isAssignableFrom 为例

```
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)) {...}
        }
    }
}
```

Kotlin

聚合场景 - 两种方案对比

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

// Other modules
@ExportService
Class ServiceFromModuleA
@ExportService
Class ServiceFromModuleB
```

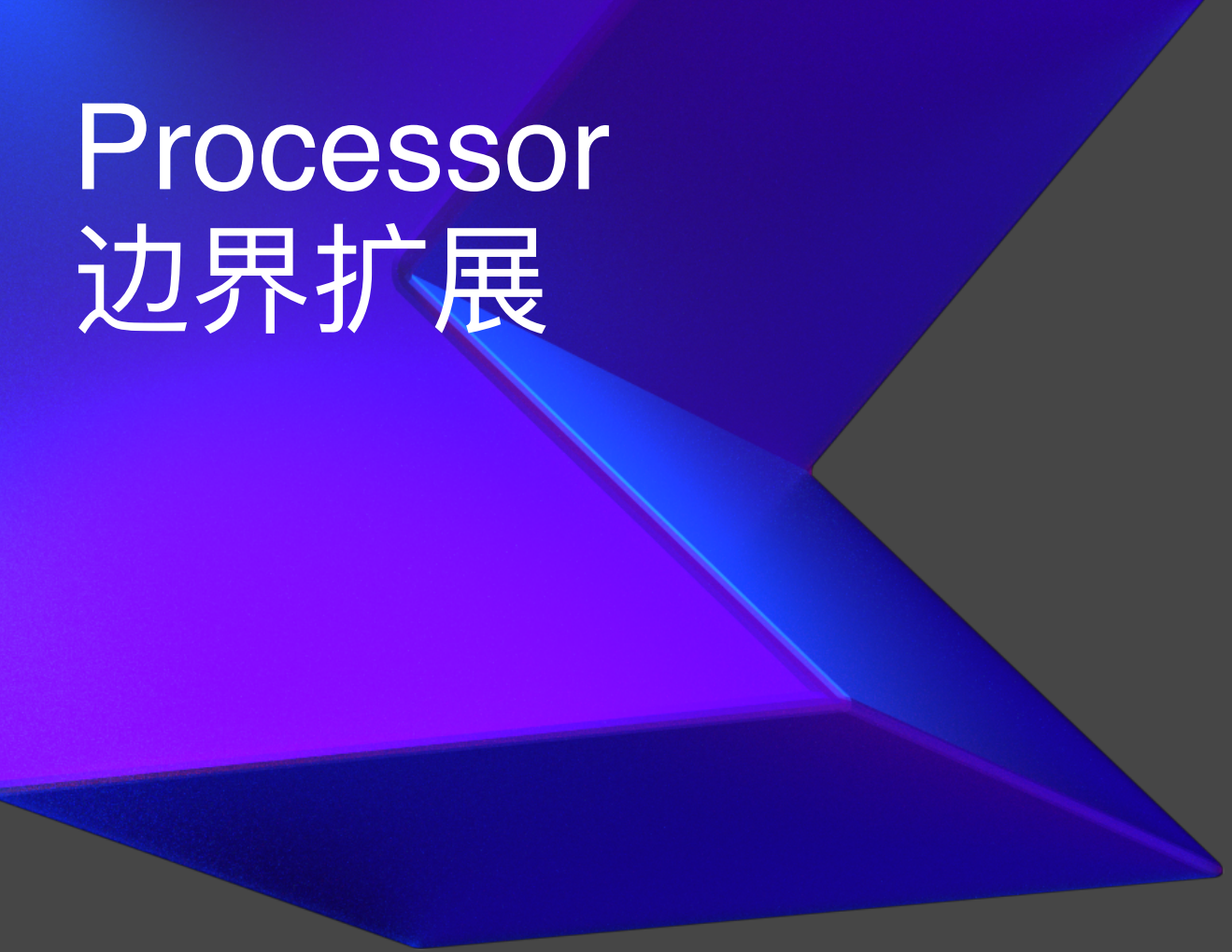
```
// Application
fun onCreate() {
    registerServices(
        ServiceFromModuleA(),
        ServiceFromModuleB()
    )
}

// Other modules
// @ExportService
Class ServiceFromModuleA: ServiceFlag
// @ExportService
Class ServiceFromModuleB: ServiceFlag
```

① 1:1 → N:1

② 多模块?

Kotlin

Processor
边界扩展



探索 Resolver

```
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>
    ..
}
```

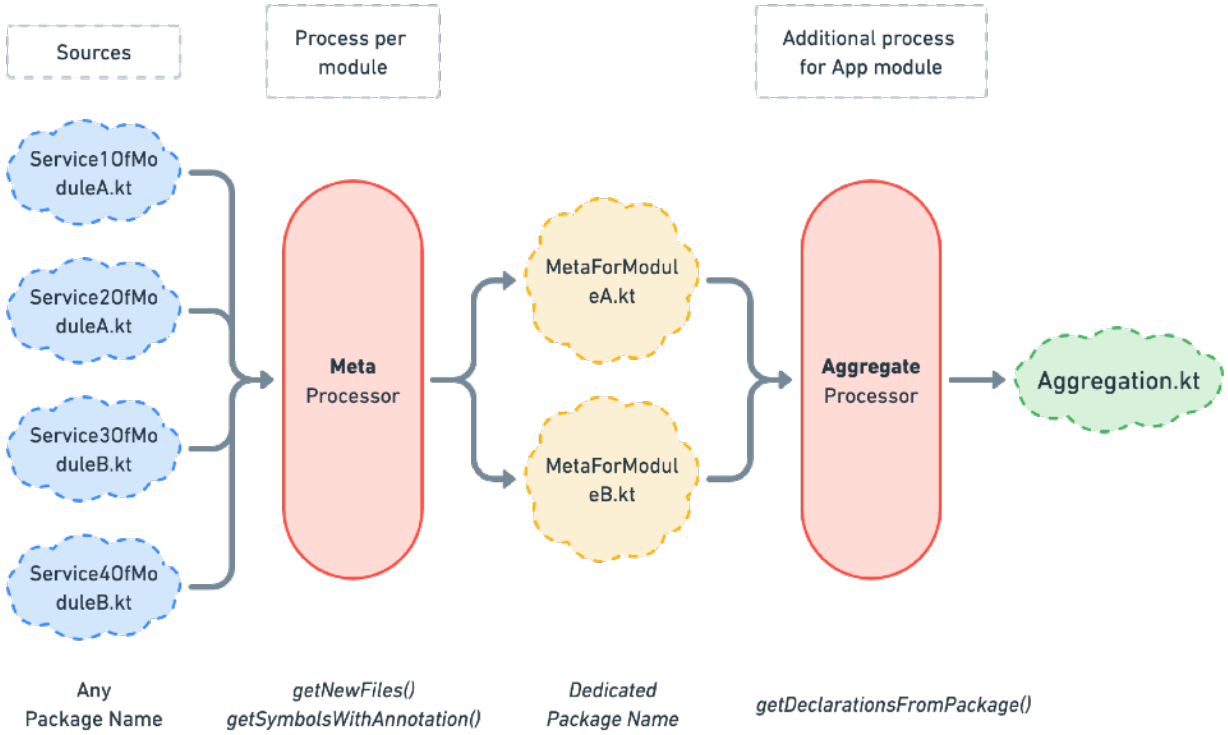
SearchScope: Sources + Deps

classes/functions/properties

无法强迫用户
使用某个包名..

Kotlin

多个 Processor

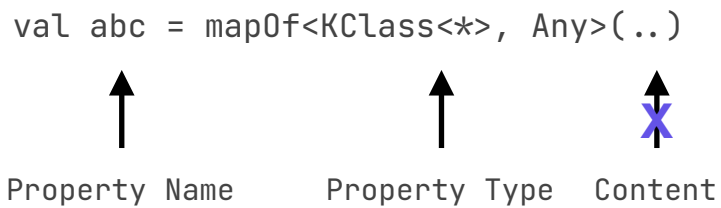


利用 KSP 多轮处理的特性

Kotlin

KSFile 可访问的元素（Element）

No Expression.
Symbol/Declaration Only



Kotlin

KSFile 可访问的元素（Element）

注解参数是唯一例外

(The only exception is annotation param)

Kotlin

利用注解参数传递数据（Data）

```
// MetaFor${ModuleName}.kt
@Meta(metaDataInJson = """{
   "annotatedClasses":{
      "examaple.ExportActivity":[
         {
            "name":"me.xx2bab.koncat.sample.android.AndroidLibraryActivity”,
            "Annotations":[ … ]
         },
         …
      ]
   },
   "typedClasses":{ … },
   "typedProperties":{ … }
}
""")
val voidProp = null // DO NOT use voidProp directly
```

Kotlin

最终结果

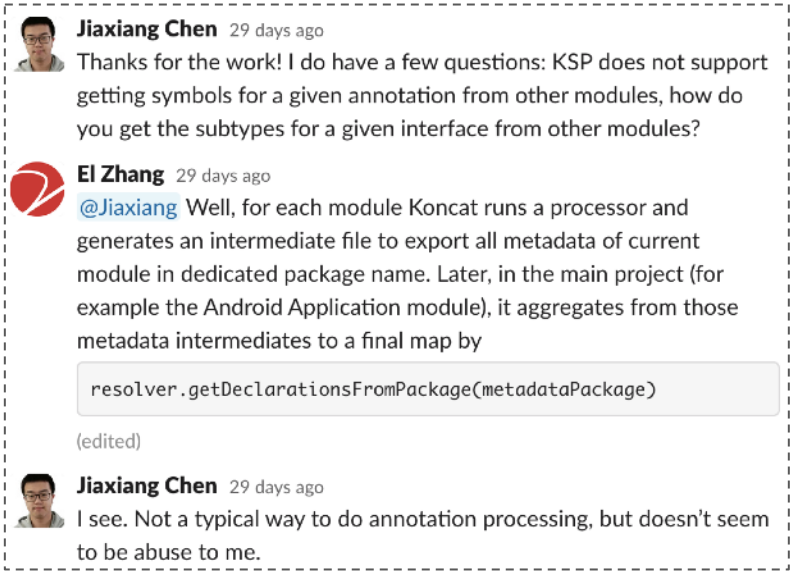
```
// Aggregation.kt
val annotatedClasses = mapOf<KClass<out Annotation>,
List<ClassDeclarationRecord>>(..)
val interfaceImplementations = mapOf<KClass<*>, Any>(..)
val typedProperties = mapOf<KClass<*>, Any>(..)
```

Kotlin

Koncat

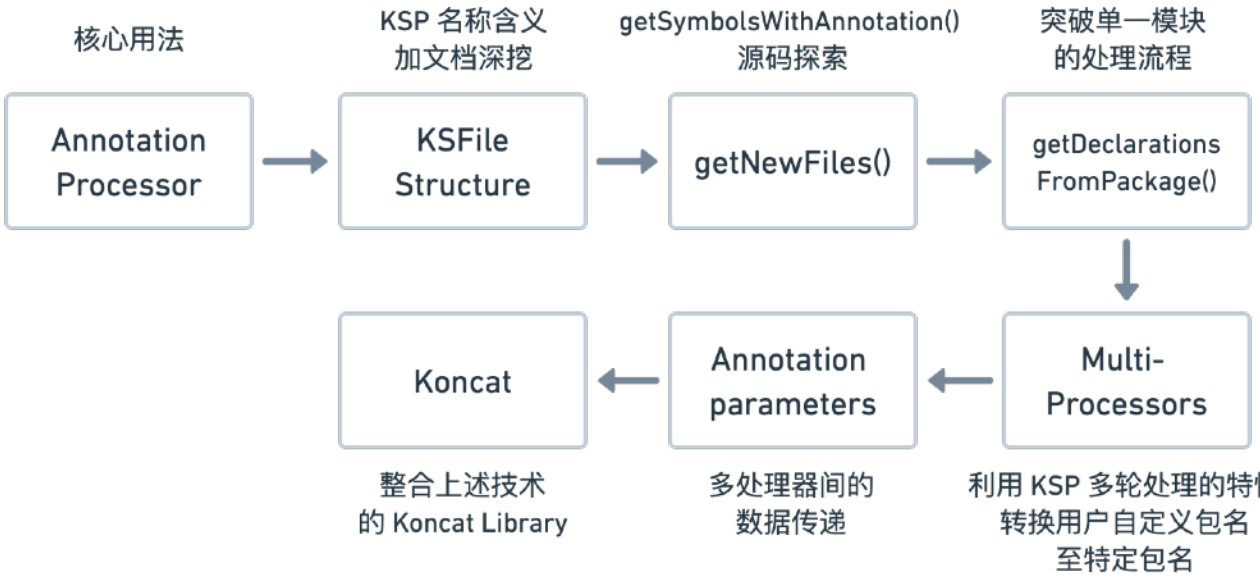
不需要反射（*Reflection*）或字节码修改（*Bytecode Transform*），也可以在编译期（*Compile-Time*）的源码阶段实现多模块的标记收集，路由表生成等。例如获取一个应用里某个接口的所有实现。

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



Kotlin

总结


核心用法
KSP 名称含义
加文档深挖
getSymbolsWithAnnotation()
源码探索
突破单一模块
的处理流程
Annotation Processor
KSFile Structure
getNewFiles()
getDeclarations FromPackage()
Multi-Processors
Annotation parameters
Koncat
利用 KSP 多轮处理的特性
转换用户自定义包名
至特定包名
多处理器间的
数据传递
整合上述技术
的 Koncat Library

Kotlin

彩蛋

Q：**为什么 Annotation Processing 是核心应用场景?**

A：**因为注解（Annotation）是为数不多可以逃脱出常规语法框架的元素。**

（直接解析 KSFile 时，要锁定某个类型或特征的类（Class）、函数（Function）、属性（Property）实际上是有其局限性的，例如我们可以判断一个类的类型（Class Type）或属性的类型（Property Type），但也局限于此。注解则没有上述的限制，它是灵活的、不与源码语法深度绑定的。）

Thanks!
Have a nice Kotlin

2BAB

xx2bab@gmail.com

github.com/2BAB

@xx2bab