



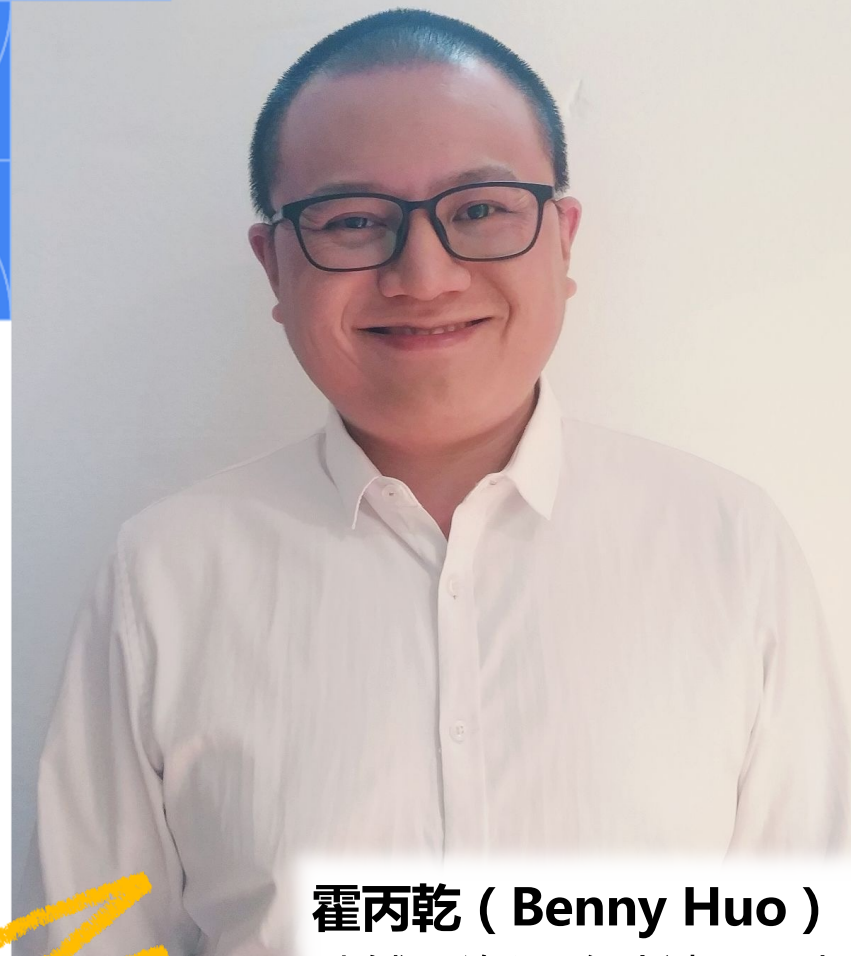
devfest



```
// You'll need  
// com.google.  
listRef.listAl  
.addOn  
prefixes.  
// All  
// You  
}  
it  
ach { item  
the items  
}  
}
```

使用 Kotlin 元编程技术提 升开发效率

 Google Developer Groups
Beijing

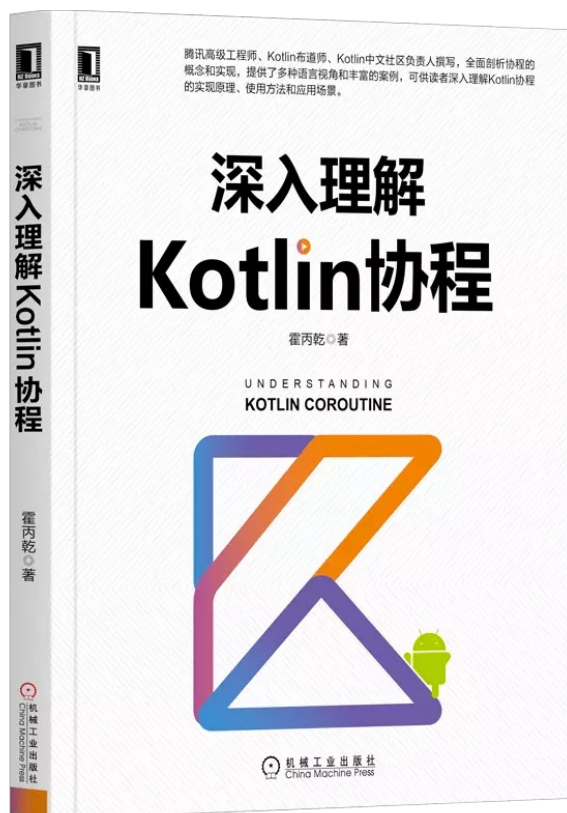


霍丙乾 (Benny Huo)
猿辅导资深移动端工程师
Google 开发者专家 (GDE)

讲者简介

- **霍丙乾 , Benny Huo**
- Google 开发者专家 (GDE , Kotlin 方向)
- 《深入理解 Kotlin 协程》《深入实践 Kotlin 元编程》作者
- Bilibili UP 主 (ID : **霍丙乾 bennyhuo**)
- 曾就职于**腾讯** , 现就职于**猿辅导**

《深入理解 Kotlin 协程》图书 2020 年 6 月出版



- 从工作机制、实现原理、应用场景、使用方法、实践技巧、标准库、框架、应用案例等多个维度全面讲解Kotlin协程的专著
- 同时提供了多语言视角，亦可通过本书了解其他语言的协程。

《深入实践 Kotlin 元编程》图书 2023 年 9 月出版



- 从工作机制、实现原理、应用场景、使用方法、实践技巧、应用案例等多个维度全面讲解 Kotlin 元编程的专著
- 涵盖 Java/Kotlin 反射、Java 注解处理器、Kotlin 符号处理器、Kotlin 编译器插件等方面内容
- 深入剖析 Jetpack Compose 编译器插件的工作机制

分享经历

2017.11	Android 技术大会	将 Kotlin 投入 Android 生产环境中
2018.11	JetBrains 北京开发者大会	优雅地使用 Kotlin 的 Data Class
2020.5	机械工业出版社	《深入理解 Kotlin 协程》
2020.5	GDG Android 11 Meetup	Kotlin 协程那些事儿
2020.10 / 11	GDG DevFest / 全球移动开发者峰会	Kotlin多平台在移动端应用与展望
2021.7	GDG 社区说	Kotlin 编译器插件：我们究竟在期待什么？
2021.11 / 12	GDG DevFest / Kotlin 中文开发者大会	从注解处理器 KAPT 到 符号处理器 KSP
2022.9	GDG 社区说	KLUE：统一 JS 调用 Native 函数的体验
2022.10	GDG DevFest	小猿口算 Android 项目优化实践
2023.4	GDG 社区说	如何开发一款 Kotlin 编译器插件
2023.5	KUG 北京 KotlinConf Global	你想知道的 Jetpack Compose 的编译器黑魔法
2023.6	Java 核心技术大会	Java 的现代化 - 包袱、挑战和革新
2023.10	JetBrains 码上道	Kotlin 开发者的首“锈”：Rust 到底香不香？

前情回顾 - 小猿口算的技术优化



小猿口算是一款免费帮助家长、老师减轻作业检查负担的学习工具类App，可通过拍照实现一秒检查小学作业，目前已全面覆盖小学阶段数学、语文、英语等各种题型。

前情回顾 - 小猿口算的技术优化



小猿口算

编译优化

包体积优化

KAE 迁移

前情回顾 - 小猿口算的技术优化

编译优化

KAPT、KSP

包体积优化

依赖分析

KAE 迁移

Kotlin 元编程


```
Text(  
  'Section Title',  
  style: TextStyle(  
    color: Colors.blue[200],  
  ),  
),  
),
```

devfest

```
s.star,  
r: Colors.blue[500],  
Text('23'),
```



元编程的基本概念

元

元

meta

表示 “关于...的”

元数据 metadata

描述数据的数据

元编程 metaprogramming

编写程序处理程序

元编程举例

- 宏 (C/C++ 预处理器)
- 代码的静态分析 (ktlint/detekt/eslint)
- 文档生成工具 (Javadoc/Dokka)
- 符号处理器 KAPT/KSP/Lombok/Manoid
- 编程器插件 (Parcelize/Jetpack Compose/Kace)
- 编译产物处理 (Proguard/Babel.js)

```
Text(  
  'Section Title',  
  style: TextStyle(  
    color: Colors.blue[200],  
  ),  
),  
),
```

devfest

```
s.star,  
r: Colors.blue[500],  
Text('23'),
```



元数据的常见类型

元数据的分类

- 注释 (comment)
- 注解 (annotation)
- @Metadata / *.kotlin_module
- PSI / FIR / IR
- 字节码

@Metadata

- **data1** : @Metadata 最核心的数据，包含类、文件及其成员的信息
- **data2** : data1 中使用到的类名、函数名等字符串字面量

```
public annotation class Metadata(  
    @get:JvmName("k")  
    val kind: Int = 1,  
  
    @get:JvmName("mv")  
    val metadataVersion: IntArray = [],  
  
    @get:JvmName("d1")  
    val data1: Array<String> = [],  
  
    @get:JvmName("d2")  
    val data2: Array<String> = [],  
  
    ...  
)
```

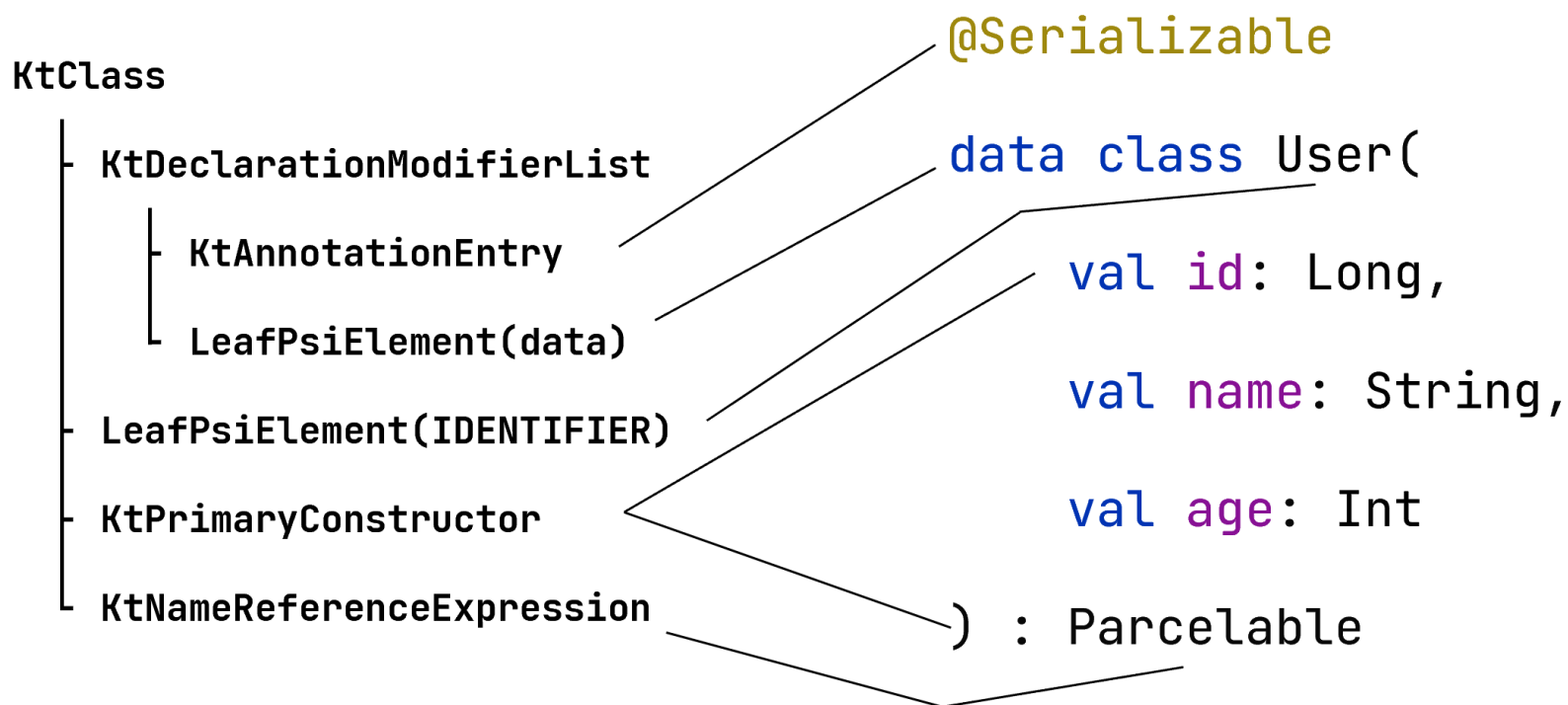
*.kotlin_module

- 存储了模块内 JVM 字节码不支持的顶级声明的信息
- 包括函数、属性、类型别名等

```
module {  
    package com.bennyhuo.lib {  
        com/bennyhuo/lib/CollectionsKt  
        com/bennyhuo/lib/ExecutorsKt  
        com/bennyhuo/lib/TaskKt  
    }  
}
```

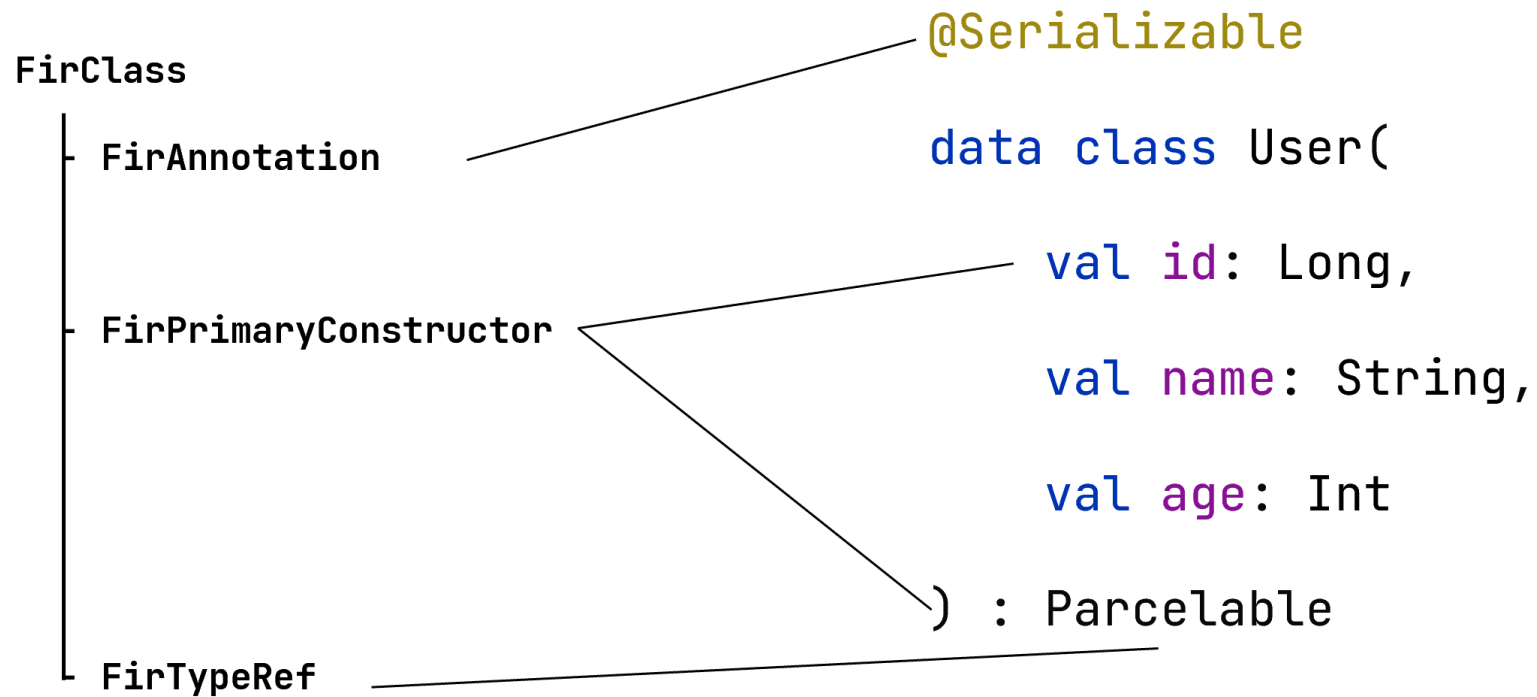
Program Structure Interface (PSI)

- IntelliJ 平台对各类编程语言语法树的统一抽象



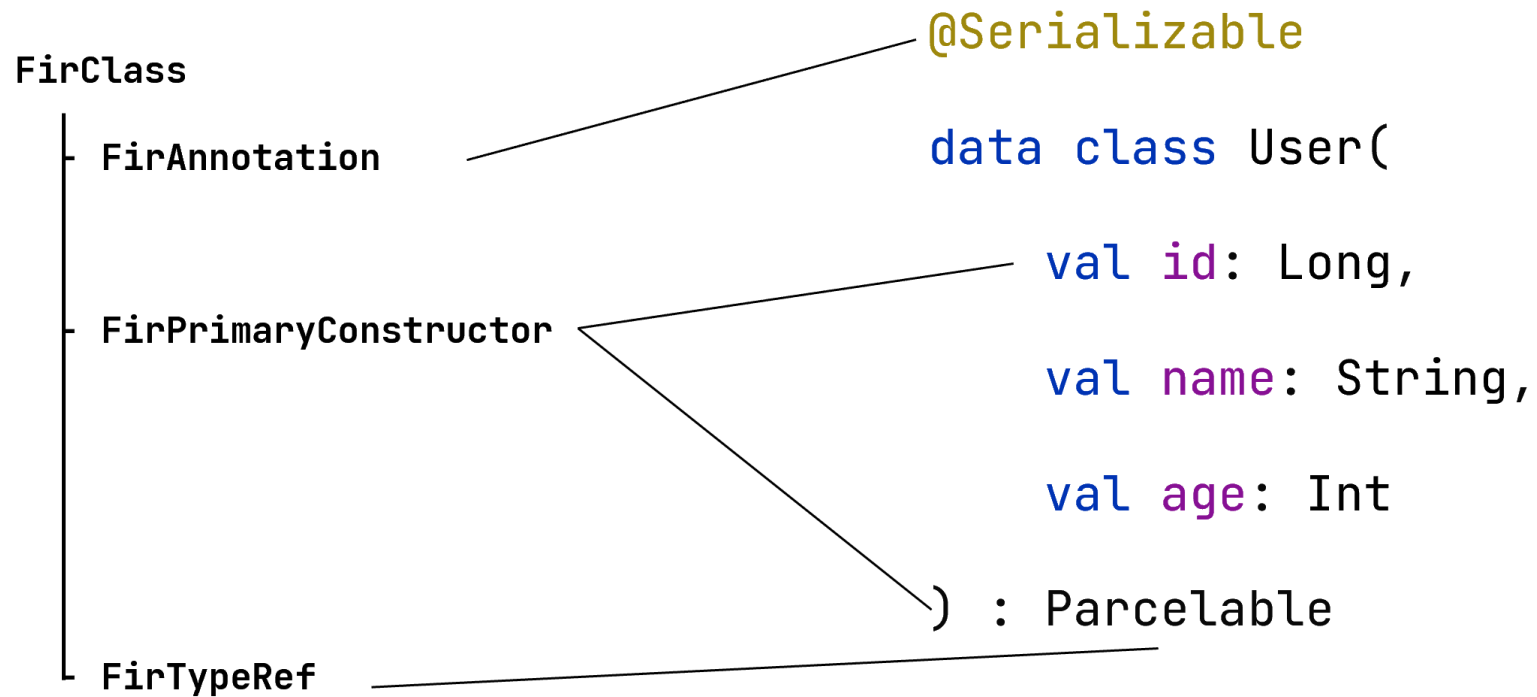
Frontend Intermediate Representation (FIR)

- K2 编译器的语法树结构，包含所有源码信息



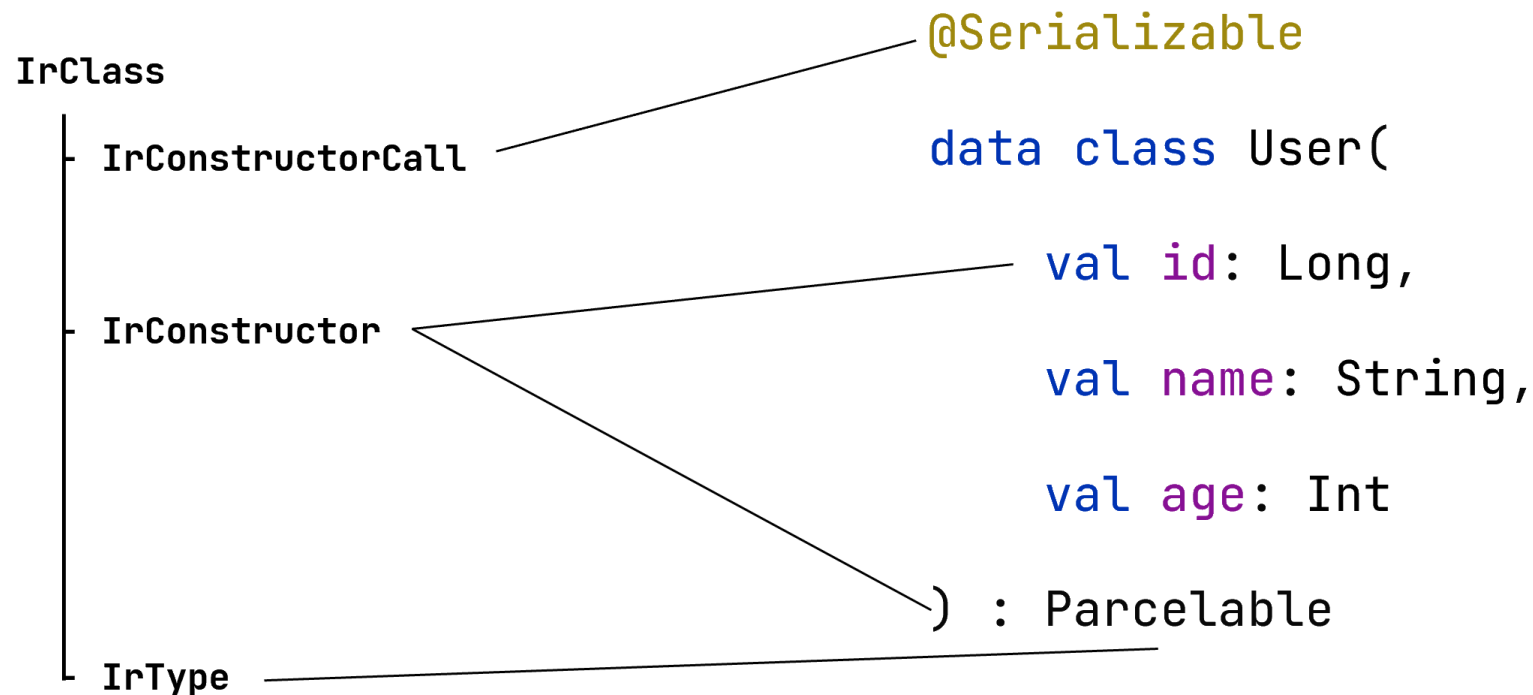
Frontend Intermediate Representation (FIR)

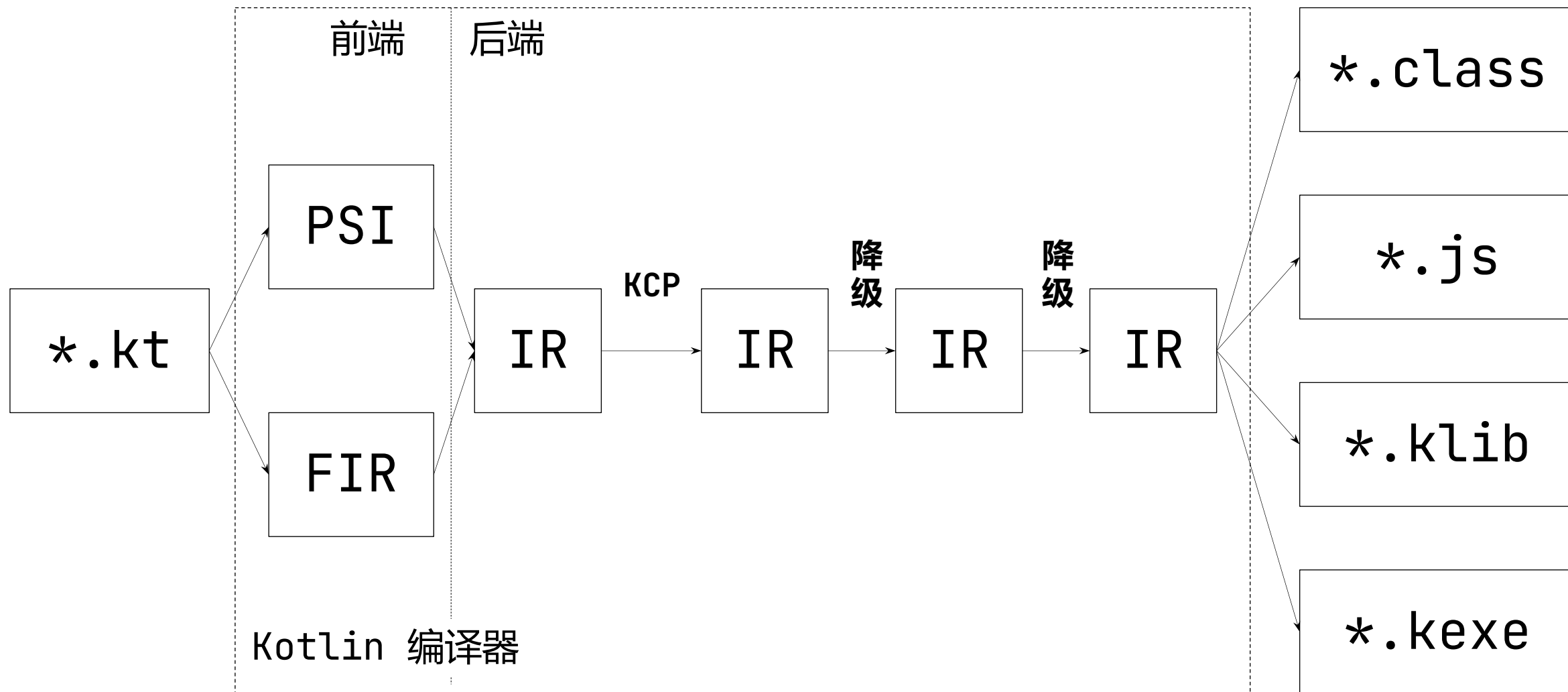
- K2 编译器的语法树结构，包含所有源码信息



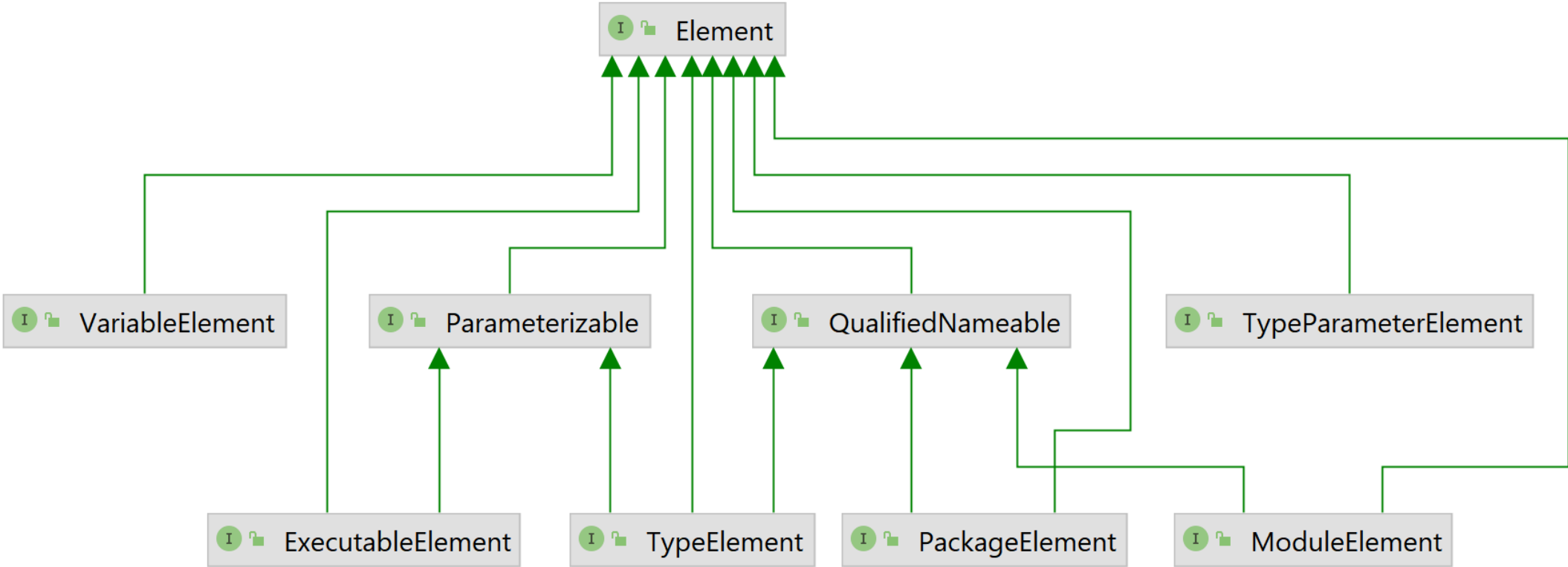
Intermediate Representation (IR)

- 编译器前端的输出，后端的输入，经过编译优化后用于生成目标产物



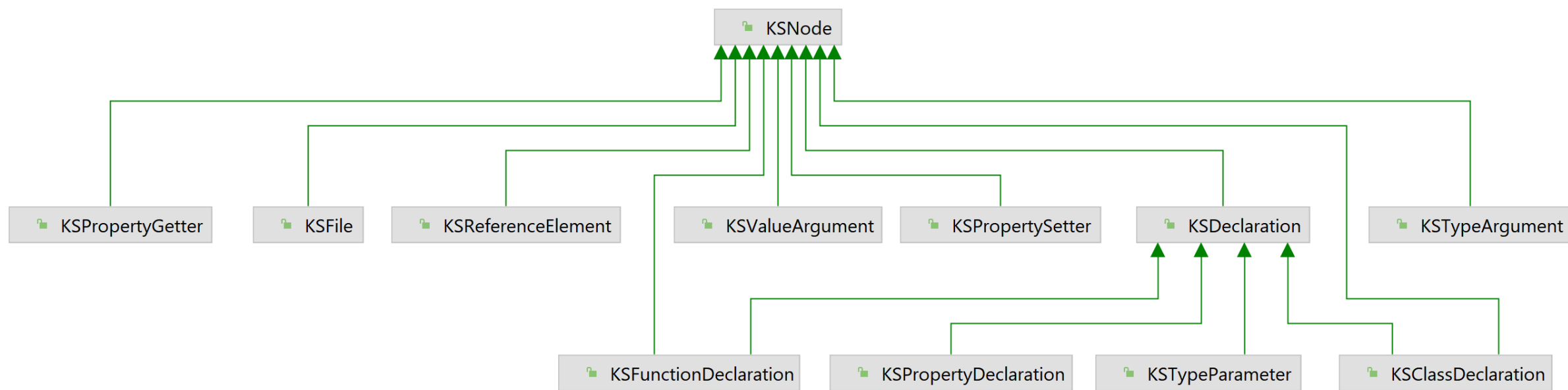


Java 符号树



* 参考《深入实践 Kotlin 元编程》第 2.5.5 节

Kotlin 符号树



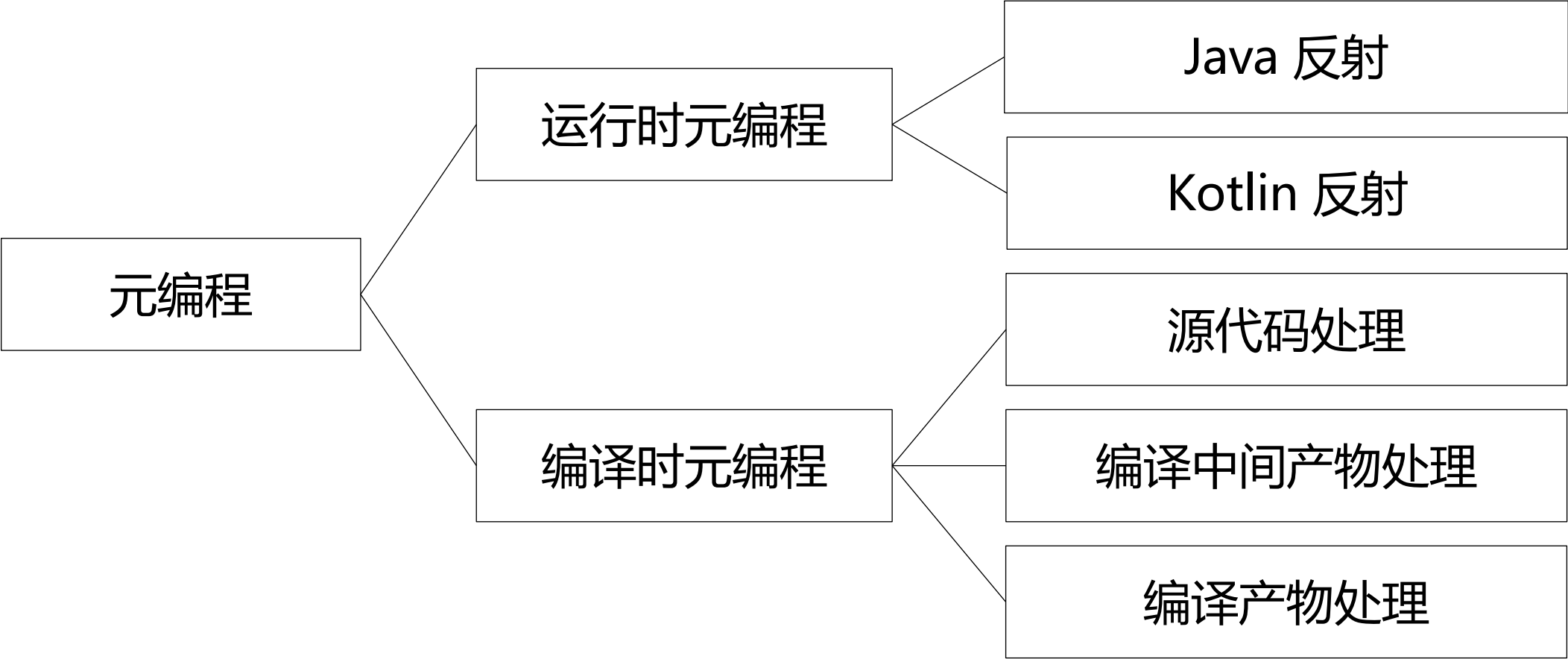
```
Text(  
  'Section Title',  
  style: TextStyle(  
    color: Colors.blue[200],  
  ),  
),  
),  
s.star,  
r: Colors.blue[500],  
Text('23'),
```

devfest



元编程的分类

元编程的分类

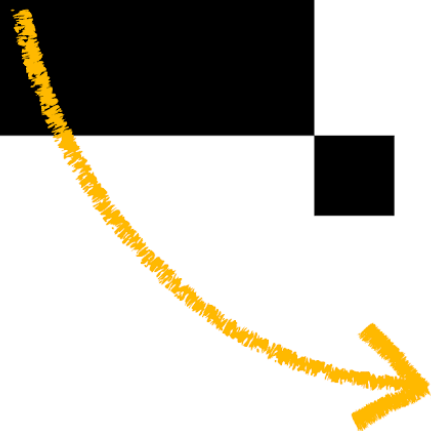


```
Text(  
    'Section Title',  
    style: TextStyle(  
        color: Colors.blue[200],  
    ),  
),  
),  
s.star,  
r: Colors.blue[500],  
Text('23'),
```

devfest



Google Developer Groups



DeepCopy

<https://github.com/bennyhuo/KotlinDeepCopy>

开发者编写的代码

```
data class District(var name: String)
```

```
data class Location(var lat: Double, var lng: Double)
```

```
data class Company(  
    var name: String,  
    var location: Location,  
    var district: District  
)
```

```
data class Speaker(var name: String, var age: Int, var company: Company)
```

```
data class Talk(var name: String, var speaker: Speaker)
```

使用 KAPT/KSP 生成的代码

```
fun Talk.deepCopy(  
    name: String = this.name,  
    speaker: Speaker = this.speaker)  
: Talk = Talk(name, speaker.deepCopy())
```

```
fun Speaker.deepCopy(  
    name: String = this.name,  
    age: Int = this.age,  
    company: Company = this.company  
): Speaker = Speaker(name, age, company.deepCopy())
```

```
fun Company.deepCopy(  
    name: String = this.name,  
    location: Location = this.location,  
    district: District = this.district  
): Company = Company(name, location.deepCopy(), district.deepCopy())
```

```
fun Location.deepCopy(  
    lat: Double = this.lat,  
    lng: Double = this.lng  
): Location = Location(lat, lng)
```

```
fun District.deepCopy(  
    name: String = this.name  
): District = District(name)
```

符号处理 (KAPT/KSP)

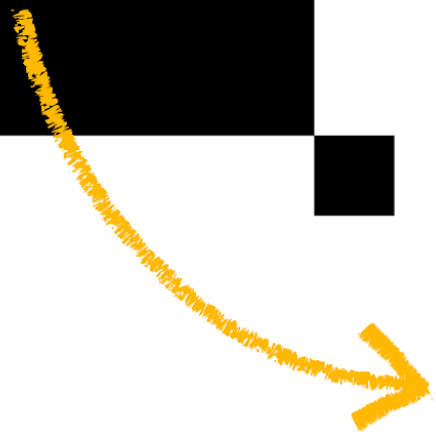
- ✓ 对符号进行规范检查
- ✓ 通过解析符号生成新的源码文件
- ✗ 访问方法/函数体
- ✗ 修改原始的符号树

```
Text(  
  'Section Title',  
  style: TextStyle(  
    color: Colors.blue[200],  
  ),  
),  
),  
s.star,  
r: Colors.blue[500],  
Text('23'),
```

devfest



Google Developer Groups



Lombok

修改 Java 语法树合成属性

```
public class User {  
    @Getter @Setter  
    private String firstName;  
    @Getter @Setter  
    private String lastName;  
  
    public String getFullName() { return firstName + " " + lastName; }  
}  
  
var user = new User();  
System.out.println(user.fullName);
```

Lombok 做了什么

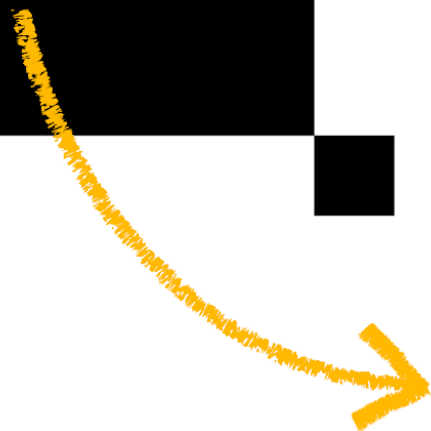
- 基于 APT 开发
- 使用了 Java 编译器内部 API 修改语法树

```
Text(  
    'Section Title',  
    style: TextStyle(  
        color: Colors.blue[200],  
    ),  
),  
),  
s.star,  
r: Colors.blue[500],  
Text('23'),
```

devfest



Google Developer Groups



TrimIndent

<https://github.com/bennyhuo/Kotlin-Trim-Indent>

```
val s1 = """
    if(a > 1) {
        return true
    }
""".trimIndent()
```

```
val s2 = """
    def test(a) {
        $s1
    }
""".trimIndent()
```

```
val s3 = """
    class Test {
        $s2
    }
""".trimIndent()
```

```
val s1 = ""
    if(a > 1) {
        return true
    }
"".trimIndent()
```

```
val s2 = ""
    def test(a) {
        $s1
    }
"".trimIndent()
```

```
val s3 = ""
    class Test {
        $s2
    }
"".trimIndent()
```

```
def test(a) {
    if(a > 1) {
    return true
}
}
```

```
class Test {
    def test(a) {
        if(a > 1) {
        return true
        }
    }
}
```

```
val s1 = ""
    if(a > 1) {
        return true
    }
"".trimIndent()
```

```
val s2 = ""
    def test(a) {
        $s1
    }
"".trimIndent()
```

```
val s3 = ""
    class Test {
        $s2
    }
"".trimIndent()
```

```
def test(a) {
    if(a > 1) {
        return true
    }
}
```

???

```
class Test {
    def test(a) {
        if(a > 1) {
            return true
        }
    }
}
```

???

```
val s1 = ""  
    if(a > 1) {  
        return true  
    }  
"".trimIndent()
```

```
val s2 = ""  
    def test(a) {  
        $s1  
    }  
"".trimIndent()
```

```
val s3 = ""  
    class Test {  
        $s2  
    }  
"".trimIndent()
```

```
val s1 = """
    if(a > 1) {
        return true
    }
    """
```

```
val s2 = """
    def test(a) {
        $s1
    }
    """
```

```
val s3 = """
    class Test {
        $s2
    }
    """
```



```
val s1 = ""  
    if(a > 1) {  
        return true  
    }  
    ""
```

```
val s2 = ""  
    def test(a) {  
        $s1  
    }  
    ""
```

```
val s3 = ""  
    class Test {  
        $s2  
    }  
    ""
```

```
val s1 = ""  
if(a > 1) {  
    return true  
}  
""
```

```
val s2 = ""  
def test(a) {  
    $s1  
}  
""
```

```
val s3 = ""  
class Test {  
    $s2  
}  
""
```

```
val s2 = ""  
def test(a) {  
  ${s1.prependIndent("  ")}  
}  
""
```


```
val s3 = ""  
class Test {  
  ${s2.prependIndent("  ")}  
}  
""
```

```
val s1 = ""  
    if(a > 1) {  
        return true  
    }  
"".trimIndent()
```


```
val s2 = ""  
    def test(a) {  
        $s1  
    }  
"".trimIndent()
```

```
val s3 = ""  
    class Test {  
        $s2  
    }  
"".trimIndent()
```

```
def test(a) {  
    if(a > 1) {  
        return true  
    }  
}
```



```
class Test {  
    def test(a) {  
        if(a > 1) {  
            return true  
        }  
    }  
}
```



TrimIndent 做了什么

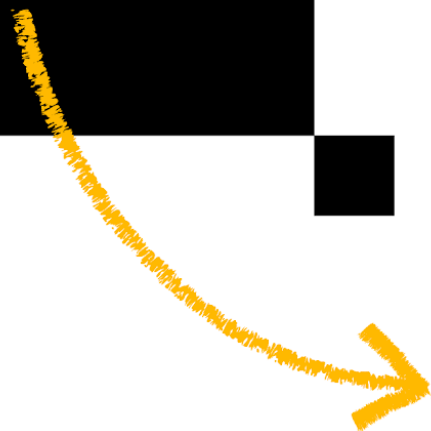
- 找到所有 `trimIndent` 函数调用
- 去掉所有 `trimIndent` 调用
- 编译时去除 `trimIndent` 的 `receiver` 的公共空白字符
- 为所有内嵌表达式添加 `prependIndent` 调用
- 通过编译器插件修改 IR 实现

```
Text(  
  'Section Title',  
  style: TextStyle(  
    color: Colors.blue[200],  
  ),  
),  
),  
s.star,  
r: Colors.blue[500],  
Text('23'),
```

devfest



Google Developer Groups



Kace

<https://github.com/kanyun-inc/Kace>

Kace 是什么

- Kotlin 1.8 废弃了 Kotlin Android Extensions
- Kace 可以完全替代 KAE，实现最小成本的代码迁移

Kace 的核心功能

📁 kace-compiler

③ Activity/Fragment 实现
AndroidExtensions 接口

📁 kace-gradle-plugin

① 根据 layout 生成属性源代码

📁 kace-runtime

② 通过 AndroidExtensions 接口
提供 findViewByIdCached 实现

生成扩展属性

```
<LinearLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
```

```
    android:orientation="vertical">
```

```
    <Button
```

```
        android:id="@+id/button1"
```

```
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
```

```
        android:text="button1" />
```

```
</LinearLayout>
```

生成扩展属性

```
private inline val AndroidExtensionsBase.button1
    get() = findViewById<Button>(this, R.id.button1, android.widget.Button::class.java)

internal inline val Activity.button1
    get() = (this as AndroidExtensionsBase).button1

internal inline val Fragment.button1
    get() = (this as AndroidExtensionsBase).button1
```

自动添加接口和实现（1）

```
class MainActivity : Activity()
```

自动添加接口和实现（2）

```
class MainActivity : Activity() ,  
    AndroidExtensions by AndroidExtensionsImpl() {  
    ...  
}
```

自动添加接口和实现（3）

```
class MainActivity : Activity() , AndroidExtensions {  
    private var impl = AndroidExtensionsImpl()  
    override fun <T: View> findViewById(  
        owner: AndroidExtensionsBase, id: Int, viewClass: Class<T>  
    ): T? {  
        return impl.findViewById(owner, id, viewClass)  
    }  
}
```

Kace 做了什么

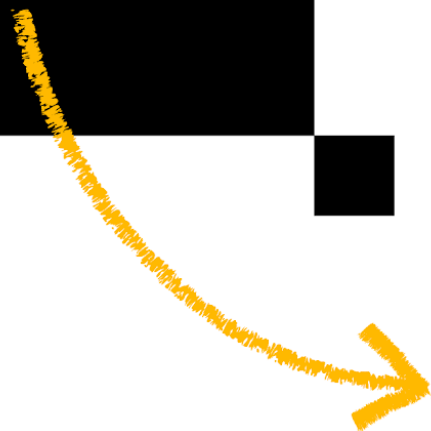
- 为所有布局文件生成合成的扩展属性
- 为所有 Activity、Fragment 添加 AndroidExtensions 接口和实现

```
Text(  
  'Section Title',  
  style: TextStyle(  
    color: Colors.blue[200],  
  ),  
),  
),  
s.star,  
r: Colors.blue[500],  
Text('23'),
```

devfest



Google Developer Groups



Kudos

<https://github.com/kanyun-inc/Kudos>

Kotlin utilities for deserializing objects (Kudos)

- 解决使用 Gson、Jackson 等框架反序列化 JSON 到 Kotlin 类时所存在的空安全问题和构造器默认值失效的问题。
- **kudos-gson** : 更安全的 Gson
- **kudos-jackson** : 更安全的 Jackson
- **kudos-android-json-reader** : 更安全更方便

Kudos 做了什么

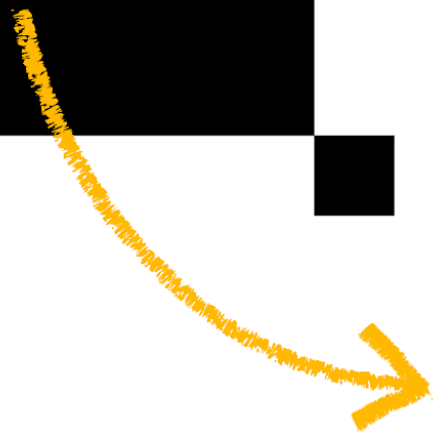
请关注 2023 Kotlin 中文开发者大会

```
Text(  
  'Section Title',  
  style: TextStyle(  
    color: Colors.blue[200],  
  ),  
),  
),  
s.star,  
r: Colors.blue[500],  
Text('23'),
```

devfest



Google Developer Groups



Jetpack Compose

Composable 函数不能同时是 suspend 函数

```
@Composable
suspend fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = name,
        modifier = modifier
    )
}
```

Suspend functions cannot be made Composable

```
@Composable
public suspend fun Greeting(
    name: String,
    modifier: Modifier = Modifier
): Unit
```

com.bennyhuo.compose
MainActivity.kt

ComposeDemo.app.main

Compose 的声明检查 (K1)

```
class ComposableDeclarationChecker : DeclarationChecker, ... {  
  
    override fun check(  
        declaration: KtDeclaration,  
        descriptor: DeclarationDescriptor,  
        context: DeclarationCheckerContext  
    ) {  
        when {  
            declaration is KtProperty && descriptor is PropertyDescriptor ->  
                checkProperty(declaration, descriptor, context)  
  
            declaration is KtPropertyAccessor && descriptor is PropertyAccessorDescriptor ->  
                checkPropertyAccessor(...)  
  
            declaration is KtFunction && descriptor is FunctionDescriptor ->  
                checkFunction(...)  
        }  
    }  
    ...  
}
```

Compose 的函数声明检查 (K1)

```
private fun checkFunction(  
    declaration: KtFunction,  
    descriptor: FunctionDescriptor,  
    context: DeclarationCheckerContext  
) {  
    val hasComposableAnnotation = descriptor.hasComposableAnnotation()  
    ...  
  
    if (descriptor.isSuspend && hasComposableAnnotation) {  
        context.trace.report(  
            COMPOSABLE_SUSPEND_FUN.on(declaration.nameIdentifier ?: declaration)  
        )  
    }  
    ...  
}
```

Composable 的函数变换

```
@Composable  
fun Greeting(name: String) {  
    Text(text = "Hello $name!")  
}
```

@Composable

```
fun Greeting(name: String, $composer: Composer?, $changed: Int) {
```

① 添加额外的参数

```
    $composer = $composer.startRestartGroup(<>)
```

```
    val $dirty = $changed
```

② 处理参数变化状态

```
    if ($changed and 0b1110 === 0) {
```

```
        $dirty = $dirty or if ($composer.changed(name)) 0b0100 else 0b0010
```

```
    }
```

```
    if ($dirty and 0b1011 !== 0b0010 || !$composer.skipping) {
```

```
        Text(name, $composer, 0b1110 and $dirty)
```

```
    } else {
```

```
        $composer.skipToGroupEnd()
```

```
    }
```

```
    $composer.endRestartGroup()?.updateScope { $composer: Composer?, $force: Int ->
```

```
        Greeting(name, $composer, updateChangedFlags($changed or 0b0001))
```

```
    }
```

③ 重组时的执行逻辑

```
}
```

currentComposer 的源码

```
val currentComposer: Composer  
    @ReadOnlyComposable  
    @Composable get() {  
        throw NotImplementedError("Implemented as an intrinsic")  
    }
```


currentComposer 的处理逻辑

```
override fun visitCall(expression: IrCall): IrExpression {  
    if (expression.symbol.owner.kotlinFqName == currentComposerIntrinsic) {  
        return expression.getValueArgument(0)  
            ?: error("Expected non-null composer argument")  
    }  
    return super.visitCall(expression)  
}
```

currentComposer 调用

```
@Composable  
fun Greeting(name: String) {  
    val composer = currentComposer  
}
```

Composable 变换结果 (1)

```
@Composable
```

```
fun Greeting(name: String, $composer: Composer?, $changed: Int) {
```

```
...
```

```
// 第二个参数是 $changed
```

```
val composer = <get-currentComposer>($composer, 0)
```

```
...
```

```
}
```

Composable 变换结果 (2)

```
@Composable
```

```
fun Greeting(name: String, $composer: Composer?, $changed: Int) {
```

```
    ...
```

```
    val composer = $composer
```

```
    ...
```

```
}
```

Compose 做了什么

- 对属性、函数的检查
- 对 Composable 函数做变换
 - 添加 \$composer 参数
 - 添加 \$default[n] 参数，用于支持函数默认值
 - 添加 \$changed[n] 参数，用于计算参数变化
 - 变换函数体，处理参数的变化状态并判断是否跳过重组
 -

Compose 的编译器插件是怎么实现的？



参考《[深入实践 Kotlin 元编程](#)》第 9 章

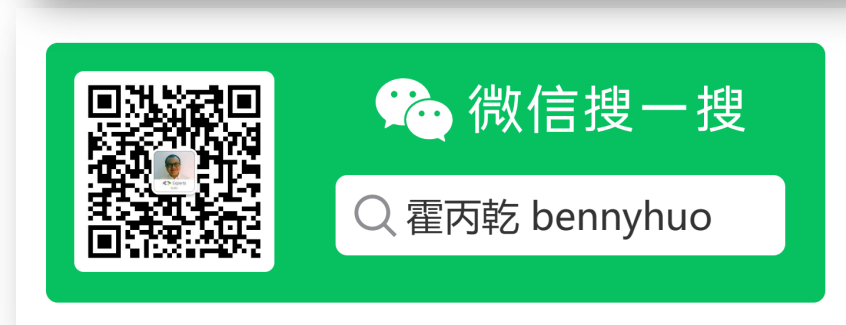
播放地址：<https://www.bilibili.com/video/BV1ck4y1j7Pa/>

Compose 能做什么？

等下请朱凯老师为大家解答

关注我

- Bilibili : [霍丙乾 bennyhuo](#)
- 微信公众号 : 霍丙乾 bennyhuo
- 掘金 : [霍丙乾 bennyhuo](#)
- YouTube : [霍丙乾 bennyhuo](#)
- 知识星球 : bennyhuo
- 微信 bennyhuo007 , 备注 “GDG”





谢谢大家