



# 静态代码分析 在体验优化中的探索实践

万钰臻，抖音 Android 基础技术

April, 2023

## 自我介绍




万钰臻

“

**2009** 毕业于中国海洋大学，  
**2010** 入职小米从事 MIUI、性能优化、稳定性等工作，  
**2021** 加入字节跳动，主要从事抖音通用优化和静态代码分析工作

”

- 
- 01 背景介绍
  - 02 静态分析介绍
  - 03 实践案例
  - 04 未来规划

# 背景介绍

抖音为什么做静态分析方向？！



# 痛点 1

灰度低级崩溃太多！

```
java.lang.NullPointerException: must not be null
```

```
    at com.ss.android.ugc.aweme.feed.desc.SearchEntityTextExtraRepo$transf  
ava:39)
```

```
    at android.os.Handler.handleCallback(Handler.java:900)
```

```
    at android.os.Handler.dispatchMessage(Handler.java:103)
```

```
    at android.os.Looper.loop(Looper.java:219)
```

```
    at android.os.HandlerThread.run(HandlerThread.java:67)
```

为什么集成测试无法发现？



# 举例

```
val cache = scene?.get(aweme.aid ?: "")
.....
if (cache == null) {
    QExecutor.work().post {
        aweme.textExtra.filter { it.type ==
TextExtraStruct.TYPE_SEARCH_TEXT }
        .let {
            if (!it.isNullOrEmpty()) {
                scene?.set(aweme.aid, it)
            }
        }
    }
}
return
}
```

空指针逻辑在特定条件执行，  
动态测试发现成本高，效率低！



结合静态分析更前置发现问题！



## 痛点 2

启动任务太多！

线程数量

300+

任务数  
量

1000

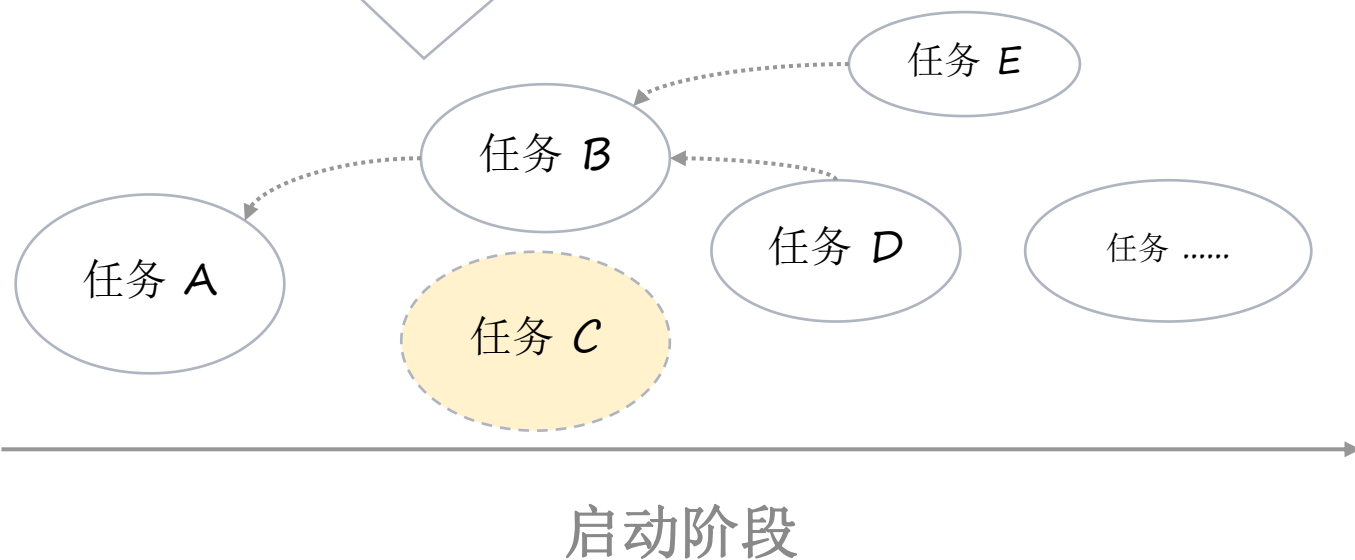
+

启动期间是否需要执行上千任务？



# 举例

- ⑩ 任务 A、B、E、D 一定在启动期间执行，且 E、D、B、A 存在依赖关系。
- ⑩ 任务 C 则无需在启动期间执行。



纯人工动态分析任务，  
费时费力且无法全面定位！

结合静态分析自动分析启动任务！



## 痛点 3

安装包体积太大！



200 M+, Dex 占  
55%

Dex 体积是否可以进一步压缩？



# 举例

抖音 Proguard Keep 有很多不合理使用，  
很多无需 Keep 类也被 Keep，导致 Dex 体积增大。

```
-keep class retrofit2.Converter.** { *; }  
-keep class org.kxml2.** { *; }  
-keep class com.google.gson.** { *; }
```

.....

纯人工动态分析 *Keep* 规则，  
费时费力且无法全面梳理！



结合静态分析自动分析 Keep 规则！

# 静态分析介绍

# 静态分析是什么

维基百科：是指在不运行程序的条件下，进行程序分析的方法。

相较动态分析，其优势如下：

- ⑩ 程序瑕疵检出成本小；
- ⑩ 可覆盖程序所有源码；
- ⑩ 不用依赖程序执行环境；

# 静态分析应用

**规范检查：** 检测代码是否符合编码规范要求

**质量分析：** 检测代码质量，例如空指针、除 0 异常等

**性能分析：** 检测代码性能问题，例如主线程 IO、重复计算等

**漏洞检测：** 检测代码安全漏洞，如缓冲区溢出、SQL 注入等

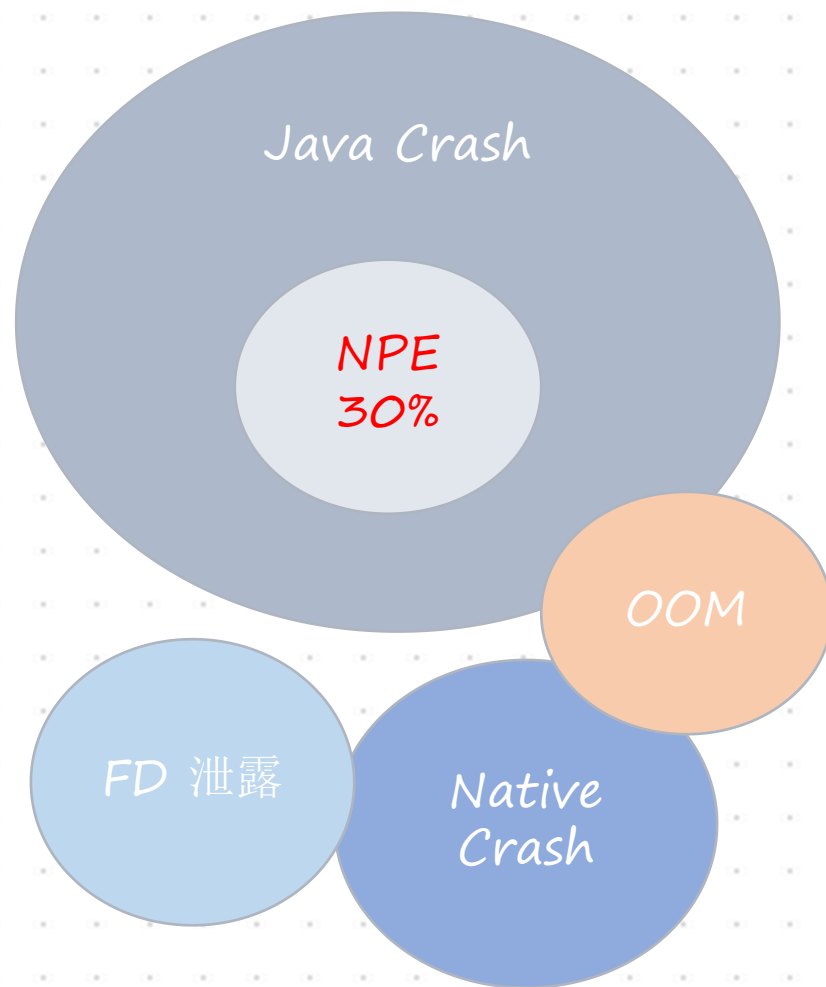
# 主流静态分析工具

名称	公司	优势	支持语言	开源
<i>Lint</i>	<i>google</i>	代码规范规则比较全，速度快	<i>java/kotlin/xml</i>	是
<i>Coverity</i>	<i>synopsys</i>	质量分析规则比较全，发现问题多	<i>java/kotlin/c++</i>	否
<i>Infer</i>	<i>facebook</i>	质量分析规则简单，速度快，准确率高	<i>java/c++</i>	是
<i>Soot</i>	学术研究	方法参数传递分析比较精准	<i>java/kotlin</i>	是

# 实践案例

# MR 稳定性分析

# 灰度常见稳定性问题





思路：在研发阶段，利用静态分析检测待合入 MR 稳定性

# 技术方案选型



Infe  
r


April, 2023

优势如下：

- ⑩ 代码开源，可定制开发；
- ⑩ 支持代码增量分析；
- ⑩ 静态代码分析效率高；

# 举例

```
String s1 = "a";  
String s2 = "b"  
String s1 = null;  
if (s2.equals("abc")) {  
    int length = s1.length();  
} else {  
    int length = s2.length() ;  
}
```



# Infer 分析 NPE

D1: s1 = "a";

D1 记录 S1 的值

D2: s2 = "b";

D2 记录 S2 的值

D3: s1 = null;

D3 导致 S1 被重新赋值，  
所以 D1 失效，D2、D3 有效。

s2.equals("abc")

if 分支判断语句，S2  
调用的时候读取 D2 的赋值，不为空

S1，调用的时候读取 D3 的赋值，  
此时为空，则抛出空指针异常！

int length = s1.length

int length = s2.length()

S2，调用的时候读取 D2 的赋值

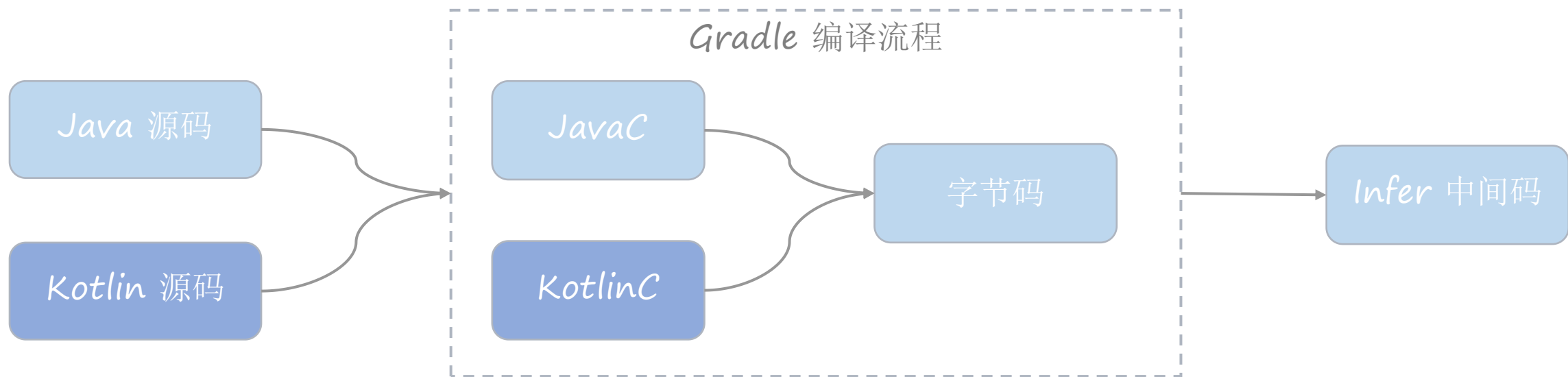
虽然 Infer 很优秀，但还是存在不足！

*Infer* 不足 1

不支持 Kotlin!

抖音业务开发基本都是 Kotlin

# 支持 Kotlin



# Infer 不足 2

不支持成员变量分析！

```
class HelloJava {  
    private String mHello;  
  
    int testField() {  
        return mHello.length();  
    }  
}
```

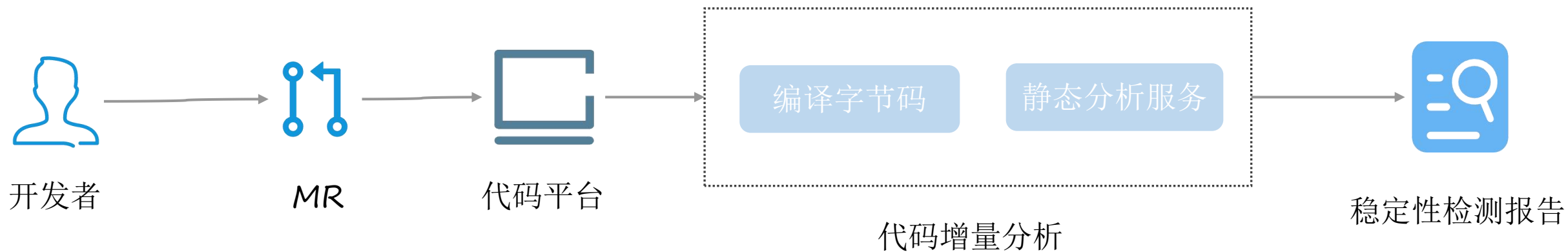
**mHello** 变量未初始化，  
NPE 异常



# 支持成员变量



# 整体方案



# 落地效果

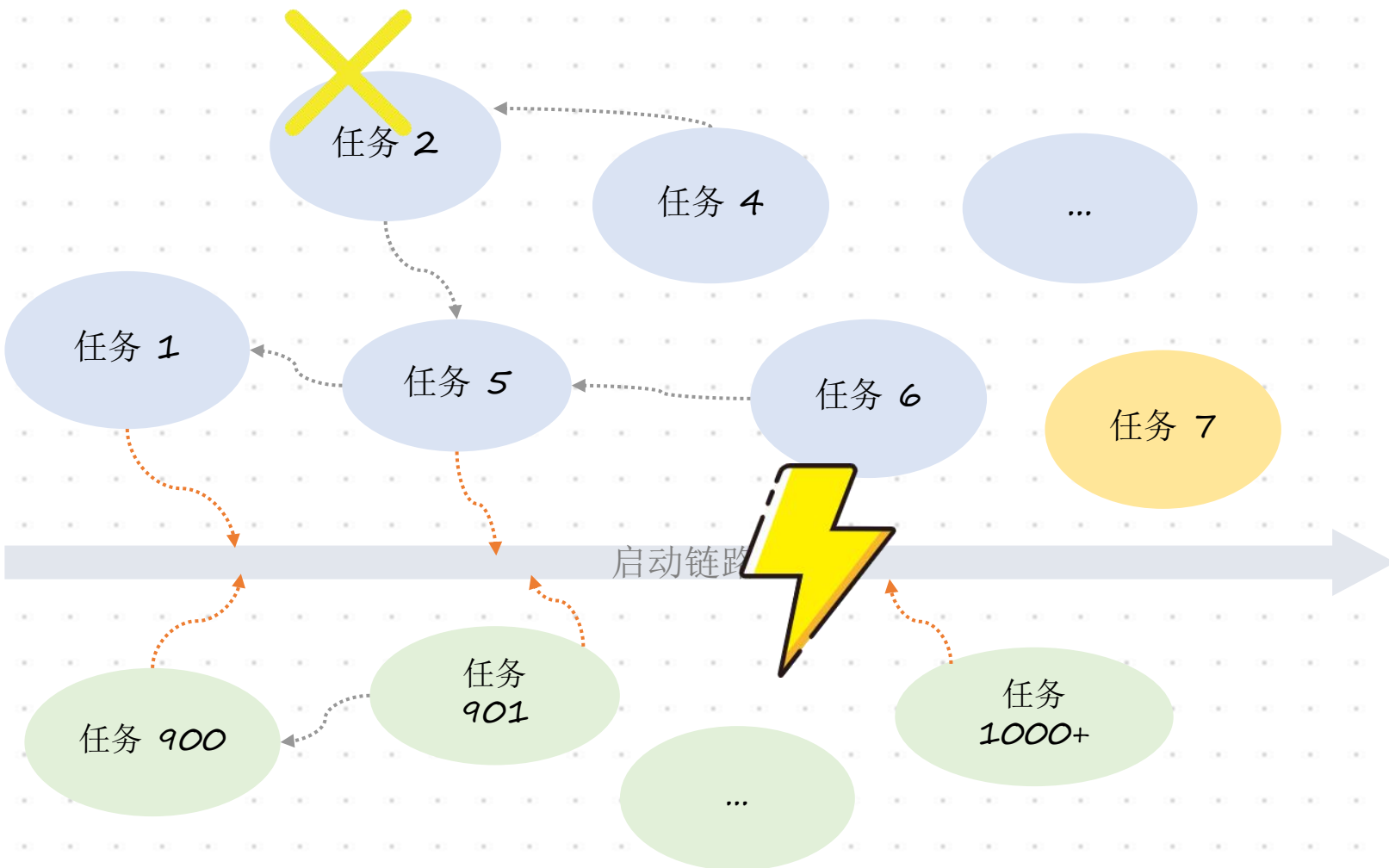
- ⑩ 灰度 *NPE* 量级  $-80\%$ ， 线上*NPE*事故报警  $-90\%$ 。
- 除 *NPE* 外，已支持数组越界、死锁、*FD* 泄露等 5 项能力。
- 累计发现异常问题 3.4 万+，已修复 2.7 万+，修复率 79%。

# 后期探索

## ⑩ 问题影响面和优先级评估

- *OOM* 检测
- *Native Crash* 检测

# 启动任务依赖分析



启动链路中需要执行 **1000+** 任务

任务之间的依赖关系错综复杂

我们尝试人工精简或调度任务，  
但经常引起启动崩溃！

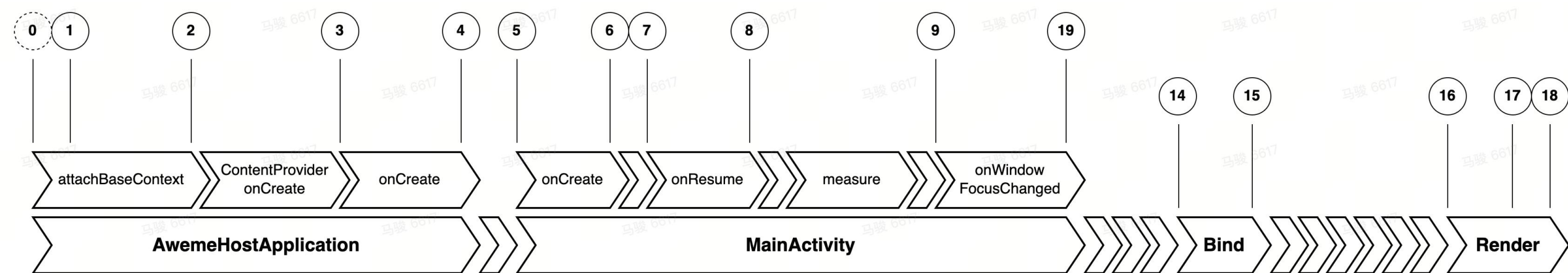
## 启动任务依赖分析重点要做好两件事：

- ⑩ 启动链路和任务之间的依赖关系。
- ⑩ 任务之间的依赖关系以及依赖顺序。

利用静态分析确定启动链路和任务的依赖关系。



# 什么是启动链路



# 什么任务

常见任务类型有：

⑩ *Runnable*;

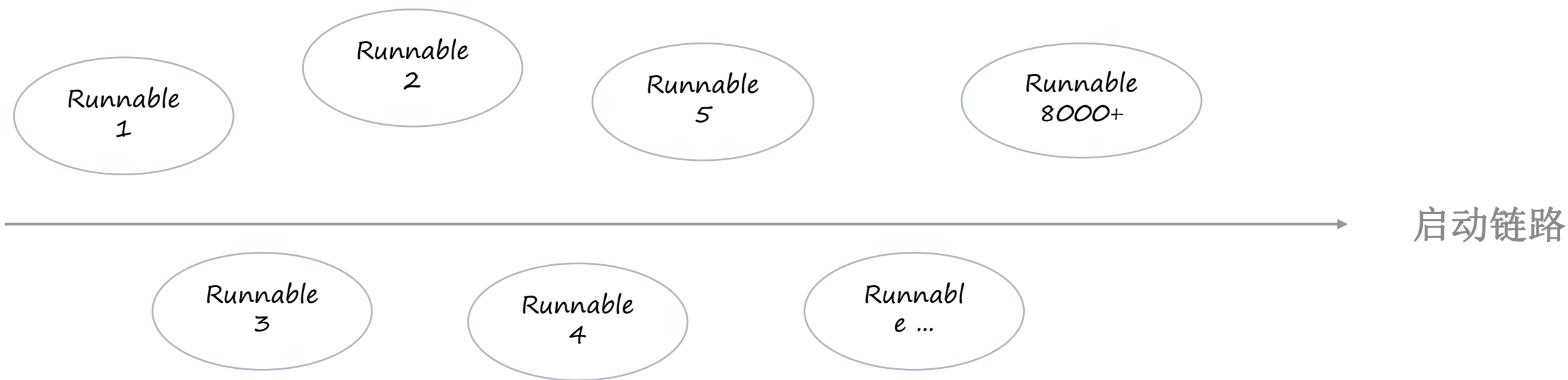
抖音有 8000+  
*Runnable*

⑩ *Callable*;

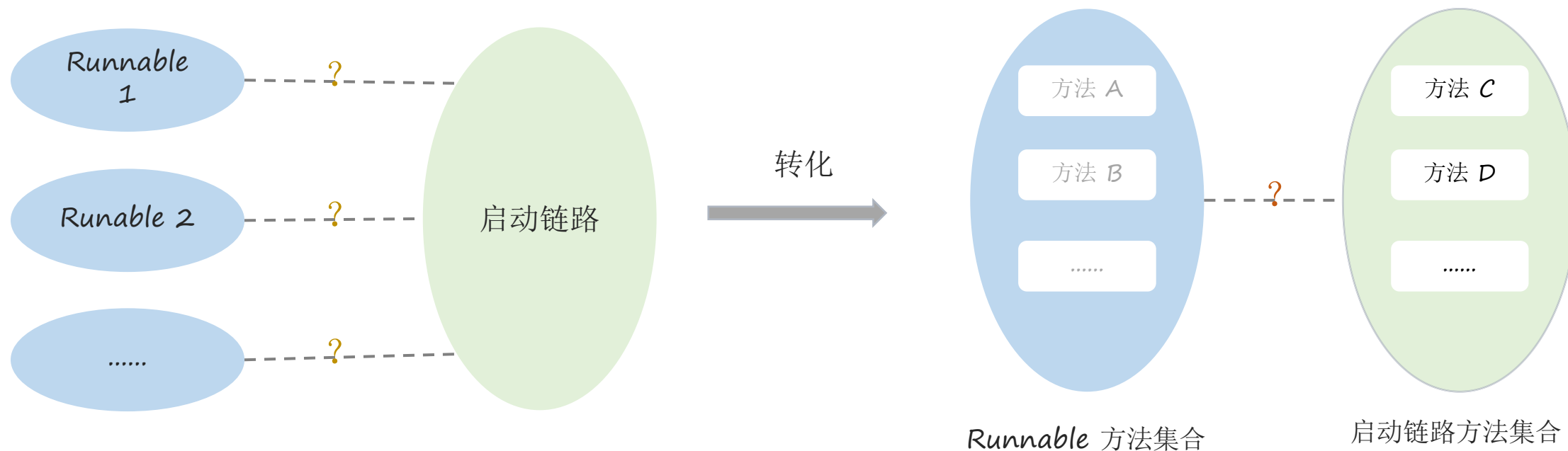
⑩ 自定义任务等;

# 如何确定启动链路和任务的依赖关系？

# 确定依赖关系

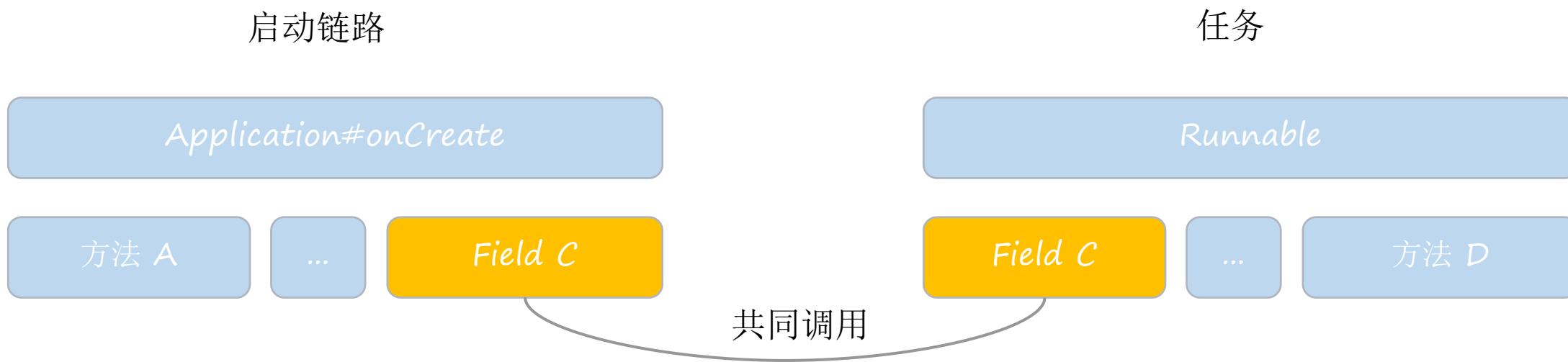


# 确定依赖关系



# 如何确定方法间的依赖关系？

# 确定依赖关系

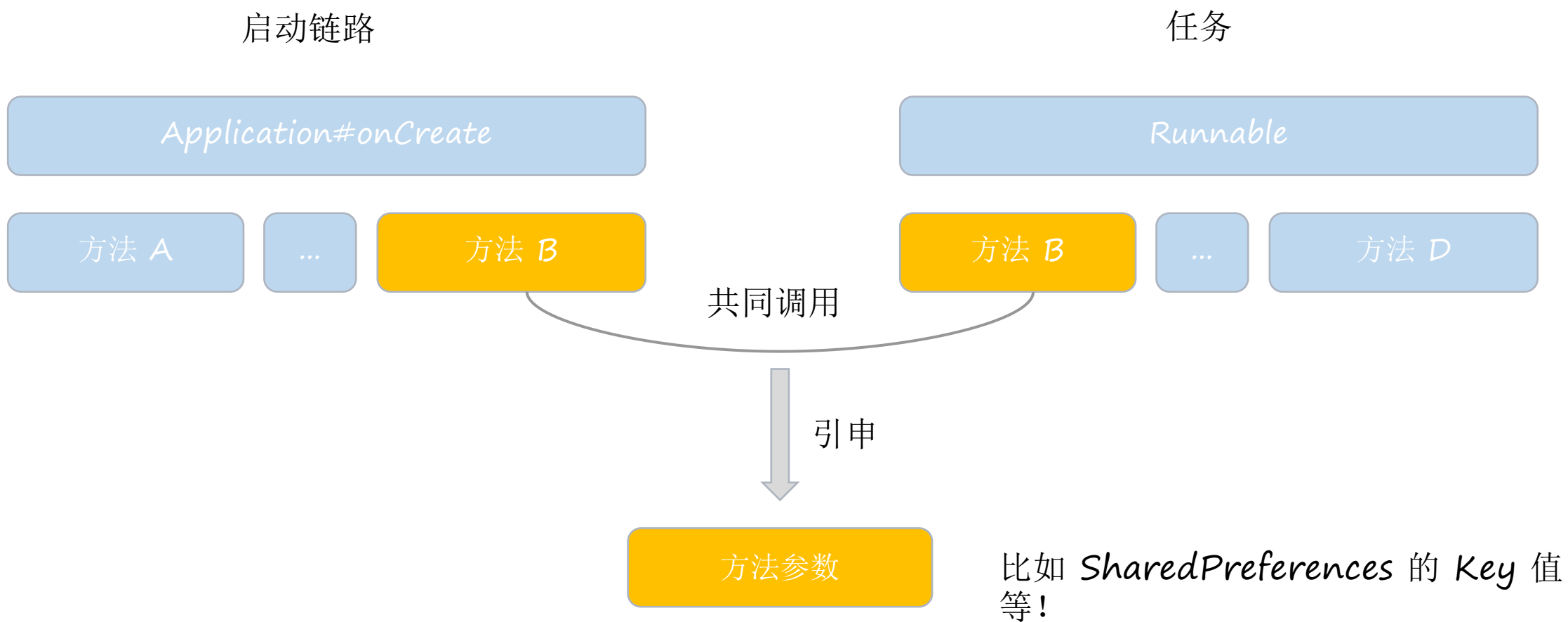


将依赖关系转换为共同调用

除成员变量，  
还有哪些规则可以确定依赖关系？



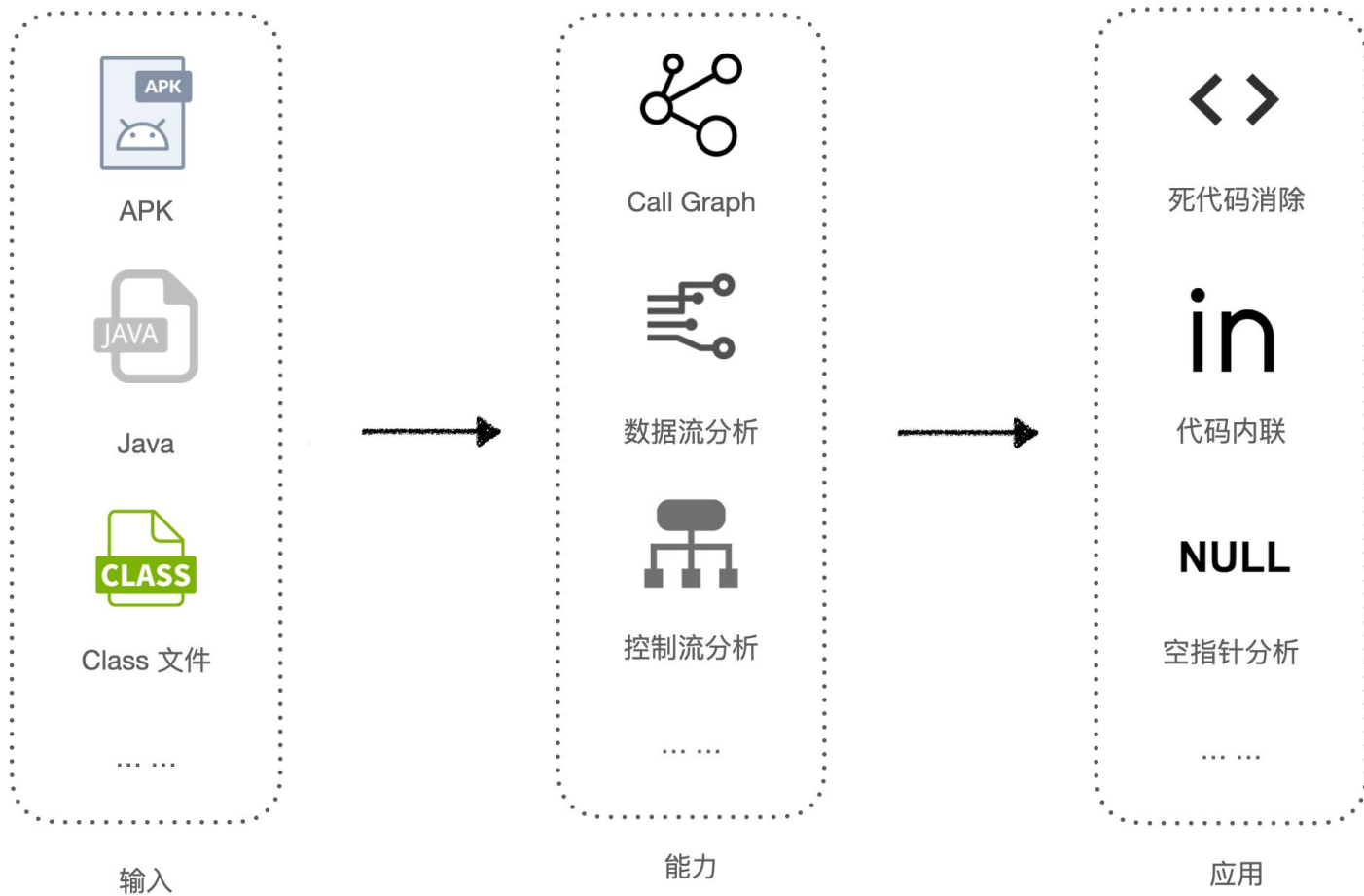
# 确定依赖关系



# 技术方案选型



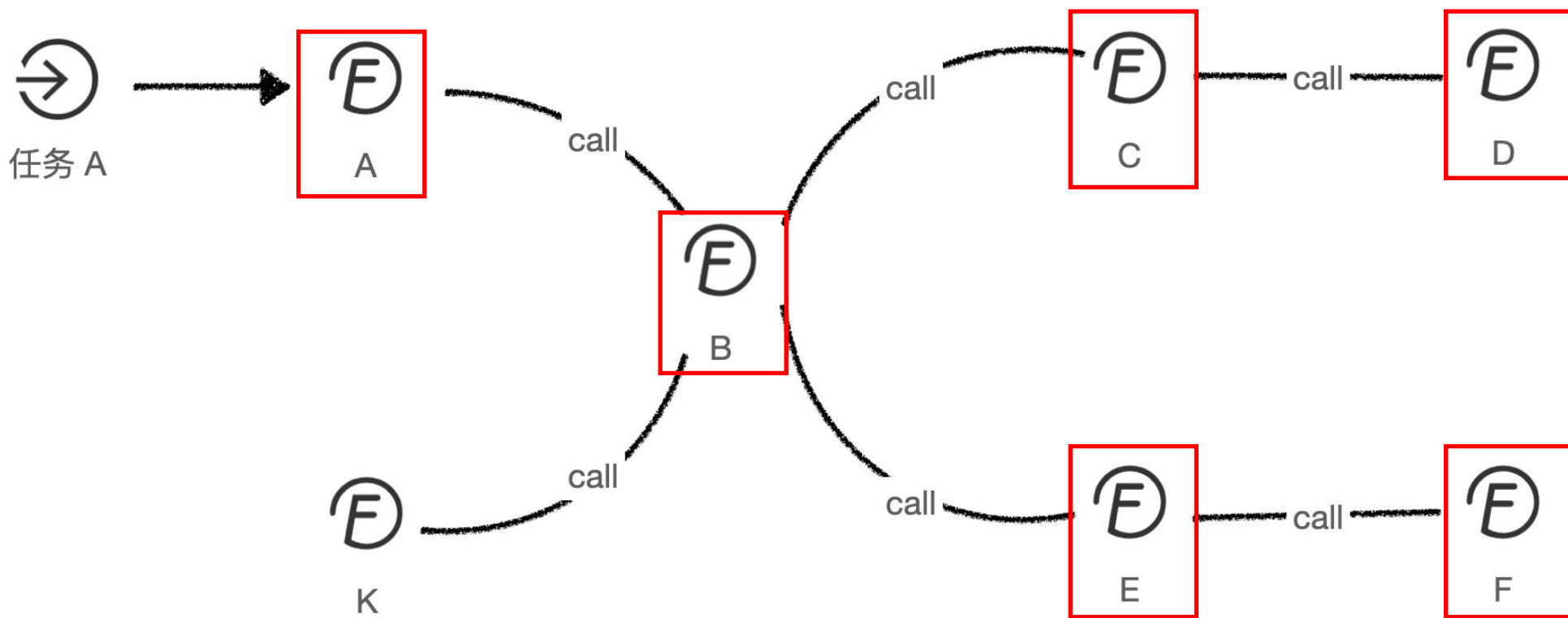
*Soot*



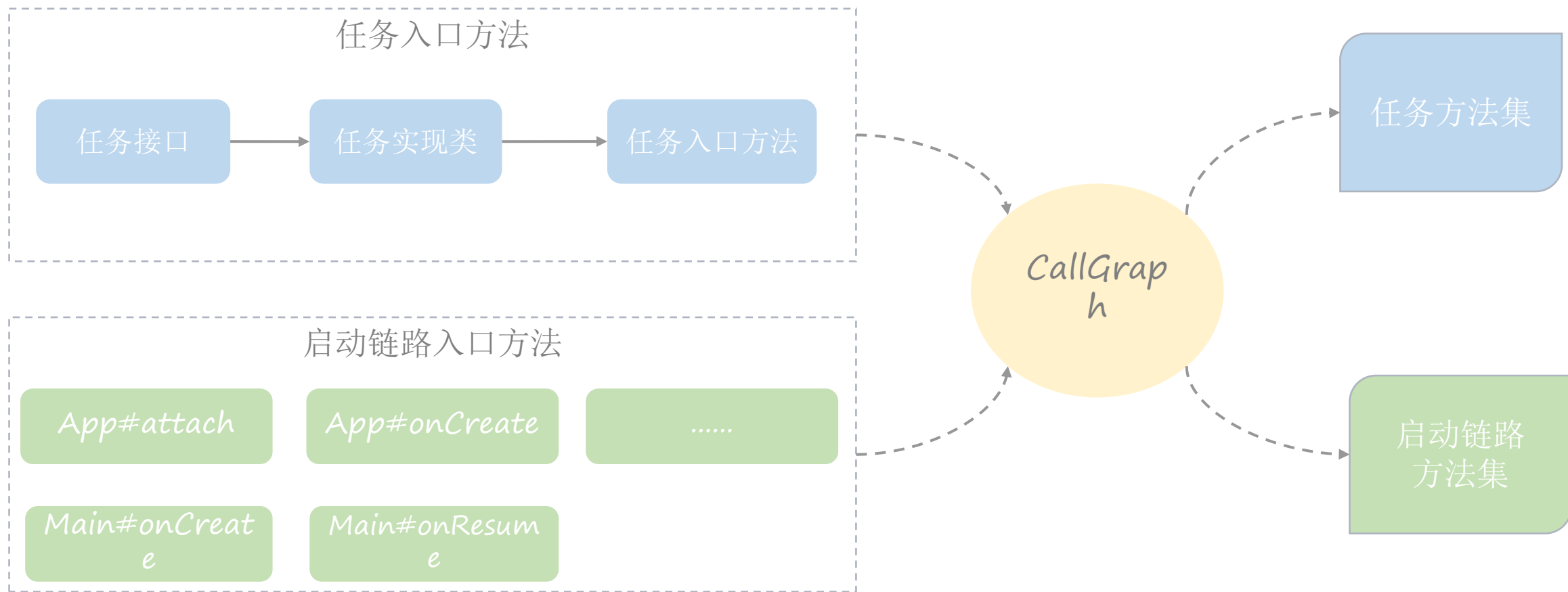
April, 2023

# CallGraph 介绍

CallGraph 用于展示代码方法间的调用关系。



# 方法集收集



# 难点分析

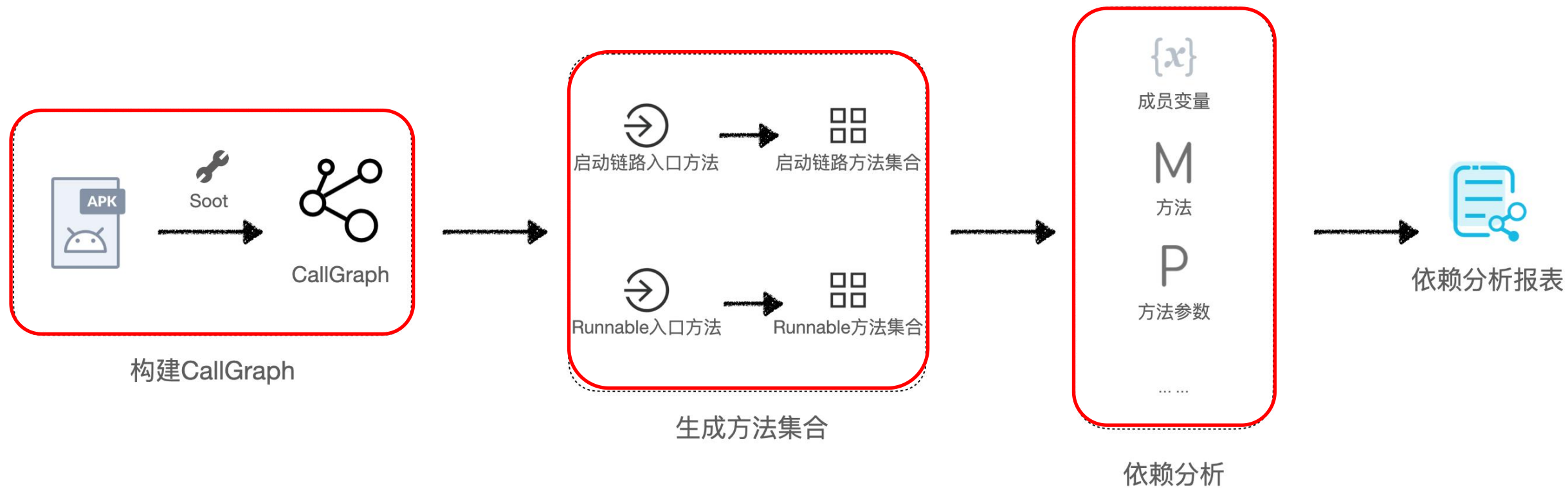
方法多态!

```
class A1 extends A {  
    @Override void a () {  
        ... ..  
    }  
}  
class A2 extends A {  
    @Override void a () {  
        ... ..  
    }  
}
```

```
public void b (Aa) {  
    ...  
    a.a();  
}
```

*a* 对象到底是 A1 还是 A2 呢?

# 整体方案



# 落地效果

- ⑩ 启动链路减少无依赖任务 240+。
- 启动指标 50 分位为减少 60+ ms。

# 后期探索

- ⑩ 不断完善启动链路和任务依赖类型，包括锁等。
- ⑩ 不断提高 *Callgraph* 多态分析准确率。
- ⑩ 静态分析任务间依赖，更合理调度任务。

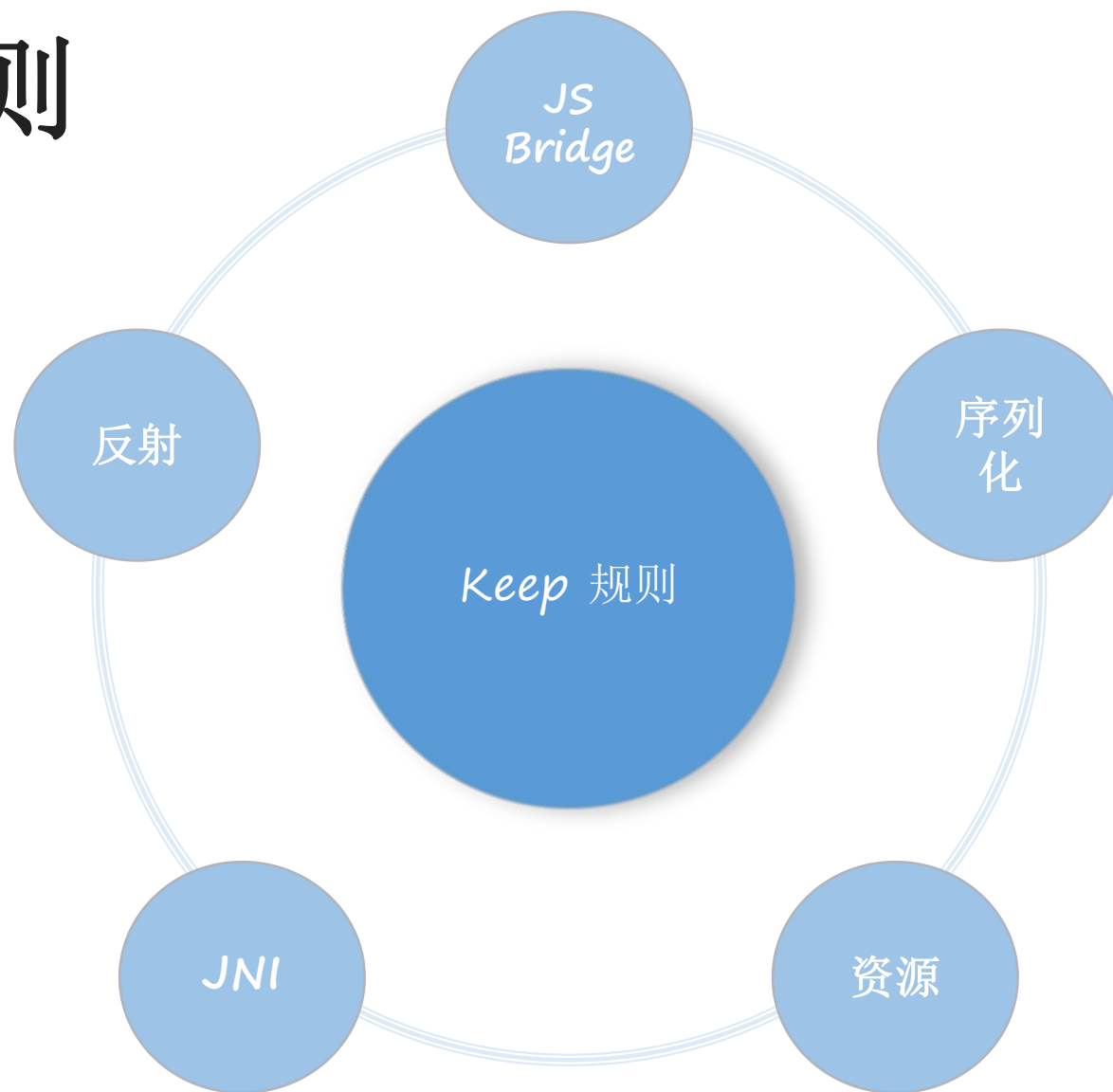


# 精准 *Proguard Keep* 规则

思路： 利用静态分析确定 Proguard Keep 规则。

哪些场景需要加入 *Keep* 规则呢？

# Keep 规则



# 技术方案选型



还是 *Soot*!

在方法数据流传递分析上有极大优势!

# *Keep* 规则之反射

# 反射 API

需要加入`Keep`规则的类、方法和属性都是通过参数传递进来的。

类名	方法名	方法签名	
java/lang/Class	forName	(Ljava/lang/String;)Ljava/lang/Class;	通过字符串获取类/方法/属性
	forName	(Ljava/lang/String;ZLjava/lang/ClassLoader;)Ljava/lang/Class;	
	getField	(Ljava/lang/String;)Ljava/lang/reflect/Field;	
	getMethod	(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;	
	getDeclaredField	(Ljava/lang/String;)Ljava/lang/reflect/Field;	
	getDeclaredMethod	(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;	
java/lang/ClassLoader	loadClass	(Ljava/lang/String;)Ljava/lang/Class;	通过传入类的字符串名来加载类
	loadClass	(Ljava/lang/String;Z)Ljava/lang/Class;	
	findClass	(Ljava/lang/String;)Ljava/lang/Class;	
	defineClass	(Ljava/lang/String;[B)Ljava/lang/Class;	
	defineClass	(Ljava/lang/String;[BILjava/security/ProtectionDomain;)Ljava/lang/Class;	
	defineClass	(Ljava/lang/String;Ljava/nio/ByteBuffer;Ljava/security/ProtectionDomain;)Ljava/lang/Class;	
	findSystemClass	(Ljava/lang/String;)Ljava/lang/Class;	
	findLoadedClass	(Ljava/lang/String;)Ljava/lang/Class;	

# 举例

```
String s1 = getClassName();
```

```
String s2 = s1
```

```
Class.forName(s2)
```

```
.....
```

```
fun getClassName(){
```

```
    return "com.A"
```

```
}
```



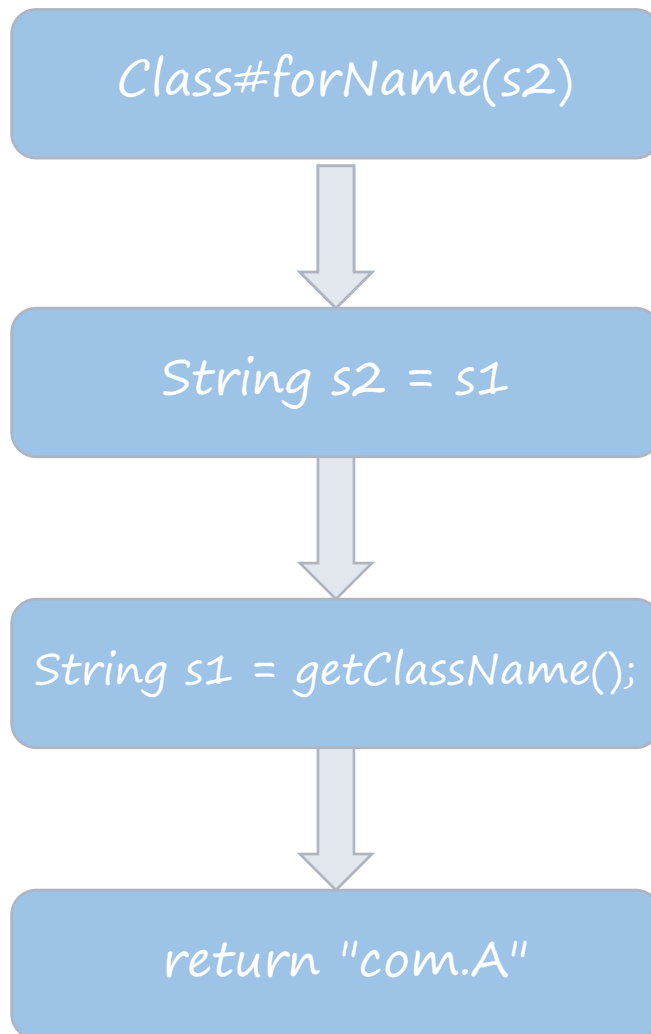
-keep class com.A



# 数据流分析

```
String s1 =  
  getClassName();  
String s2 = s1  
Class.forName(s2)  
  
fun getClassName(){  
  return "com.A"  
}
```

April, 2023



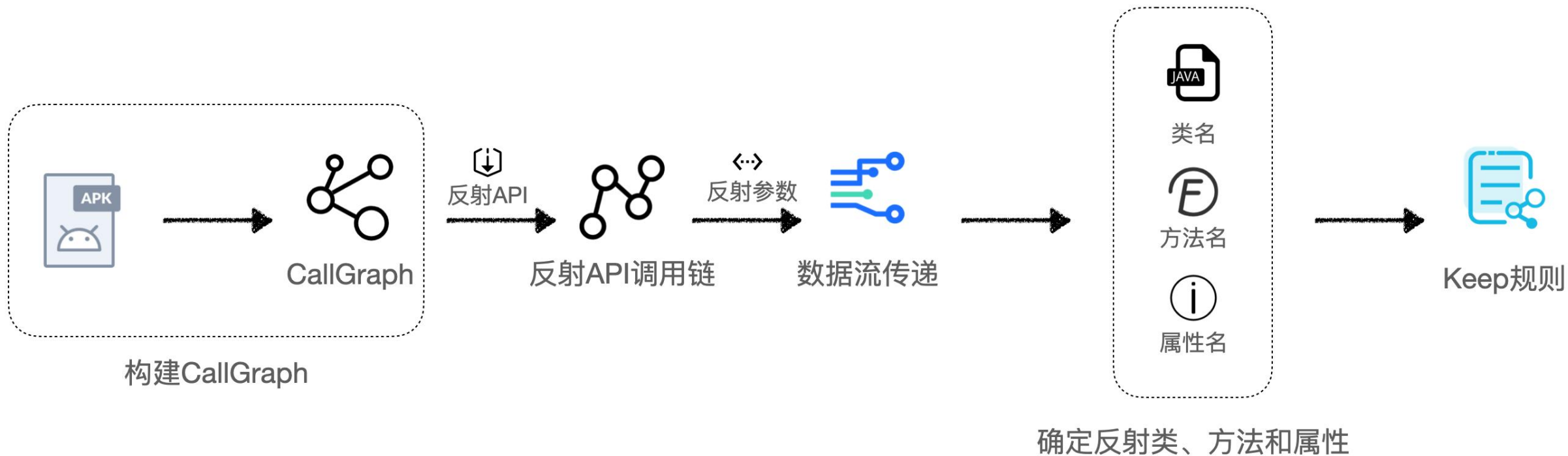
找到反射 API `Class.forName`  
记录参数 `s2`

发现 `s2` 是由 `s1` 赋值得来

`s1` 是被 `getClassName` 方法赋值。

`getClassName` 方法返回值是  
`String` 类型，停止数据流分析，发  
现反射类

# 反射整体方案



# *Keep* 规则之序列化

# 举例

// 定义一个 Java 类

```
public class Person {  
    String name;  
}
```

// 将 JSON 字符串对象转换为 Java

```
String json = "{\\"name\\":\\"Tom\\"}";  
person = gson.fromJson(json,  
    Person.class);
```



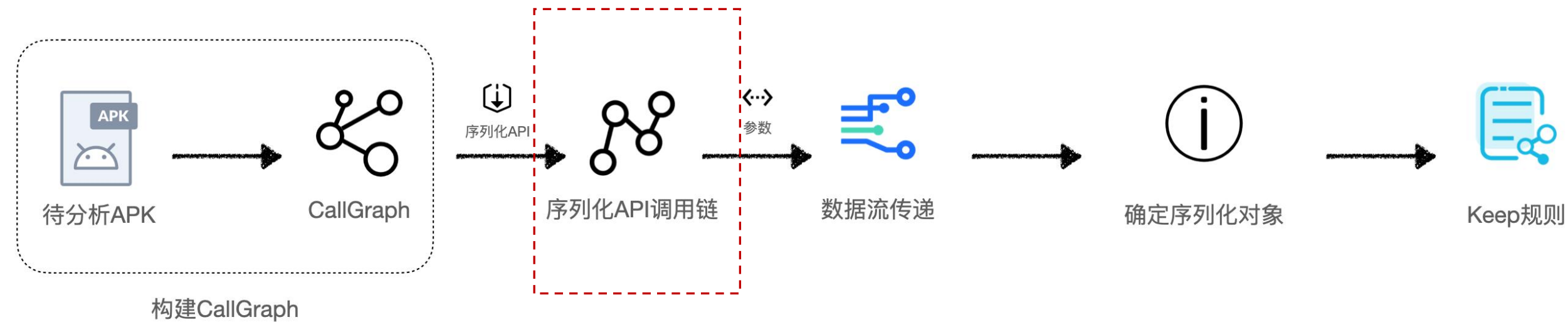
-keep class Person

# 序列化 API

序列化对象同样都是通过参数传递进来的。

类名	方法名	
com/google/gson/Gson	fromJson	(Ljava/io/Reader;Ljava/lang/Class;)Ljava/lang/Object;
		(Ljava/lang/String;Ljava/lang/Class;)Ljava/lang/Object;
		(Ljava/lang/String;Ljava/lang/reflect/Type;)Ljava/lang/Object;
		(Lcom/google/gson/JsonElement;Ljava/lang/Class;)Ljava/lang/Object;
		(Lcom/google/gson/JsonElement;Ljava/lang/reflect/Type;)Ljava/lang/Object;
		(Lcom/google/gson/stream/JsonReader;Ljava/lang/reflect/Type;)Ljava/lang/Object;
		(Ljava/io/Reader;Ljava/lang/reflect/Type;)Ljava/lang/Object;
	toJsonTree	(Ljava/lang/Object;)Lcom/google/gson/JsonElement;
		(Ljava/lang/Object;Ljava/lang/reflect/Type;)Lcom/google/gson/JsonElement;
	toJson	(Ljava/lang/Object;)Ljava/lang/String;
		(Ljava/lang/Object;Ljava/lang/reflect/Type;Ljava/lang/Appendable;)V
		(Ljava/lang/Object;Ljava/lang/reflect/Type;)Ljava/lang/String;
		(Ljava/lang/Object;Ljava/lang/reflect/Type;Lcom/google/gson/stream/JsonWriter;)V
		(Ljava/lang/Object;Ljava/lang/Appendable;)V
android/os/Parcel	writeParcelable	(Landroid/os/Parcelable;I)V
java/io/ObjectOutputStream	writeObject	(Ljava/lang/Object;)V

# 序列化方案



整体方案和反射一致，核心是明确序列化 *API*

# *Keep* 规则之资源

# 举例

// 创建自定义View

```
public class MyView extends View {
```

```
.....
```

```
}
```



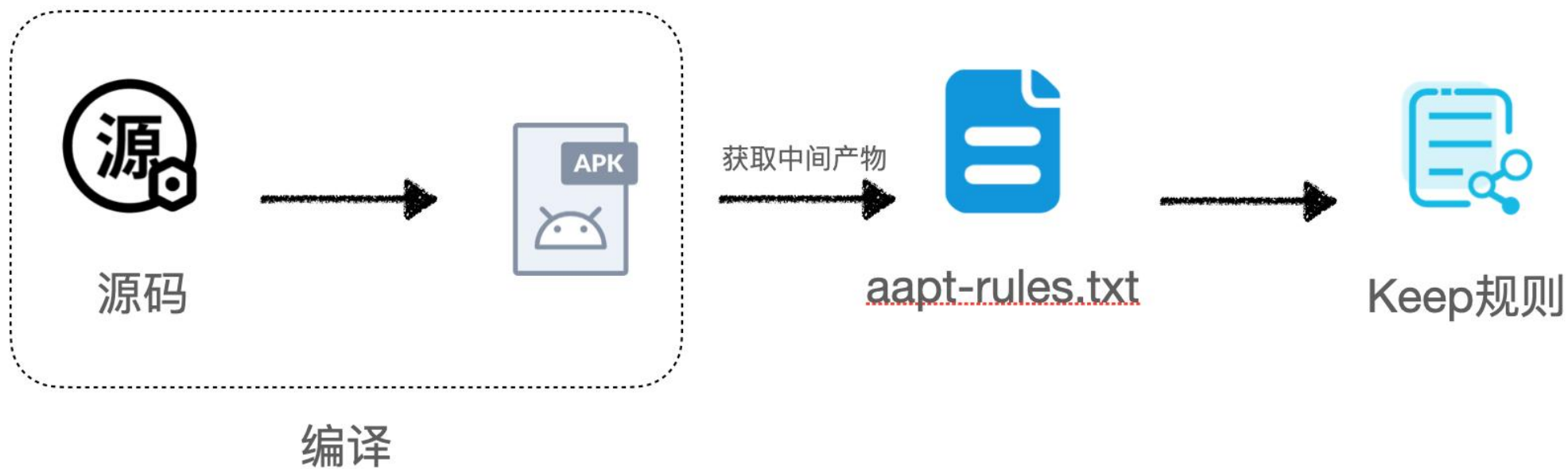
```
-keep class com.example.MyView
```

// 在 xml 文件中使用

```
<com.example.MyView/>
```

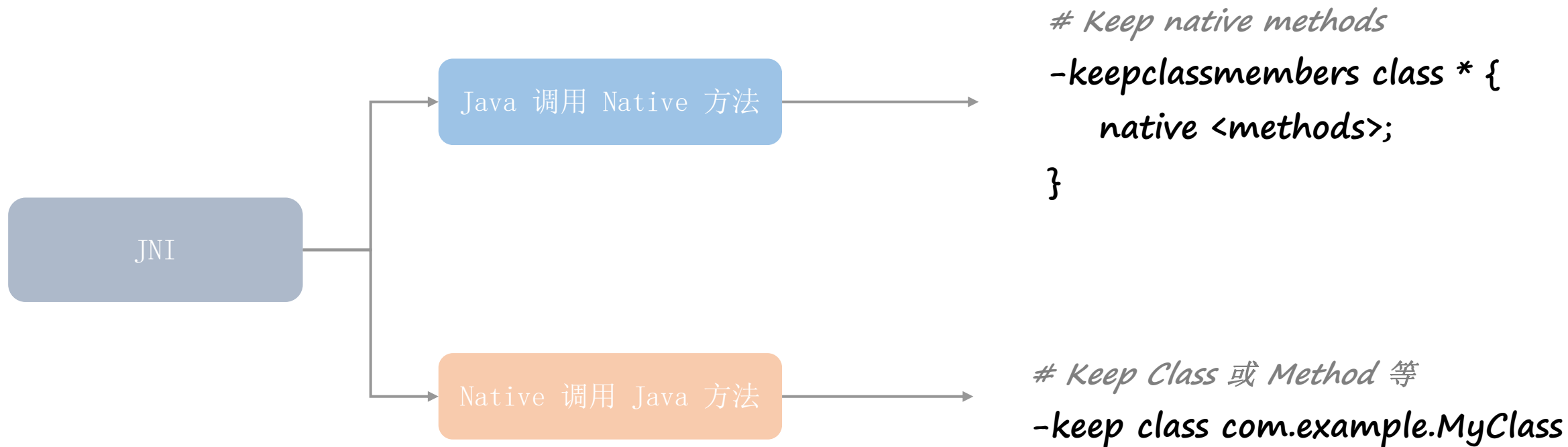


# 资源方案

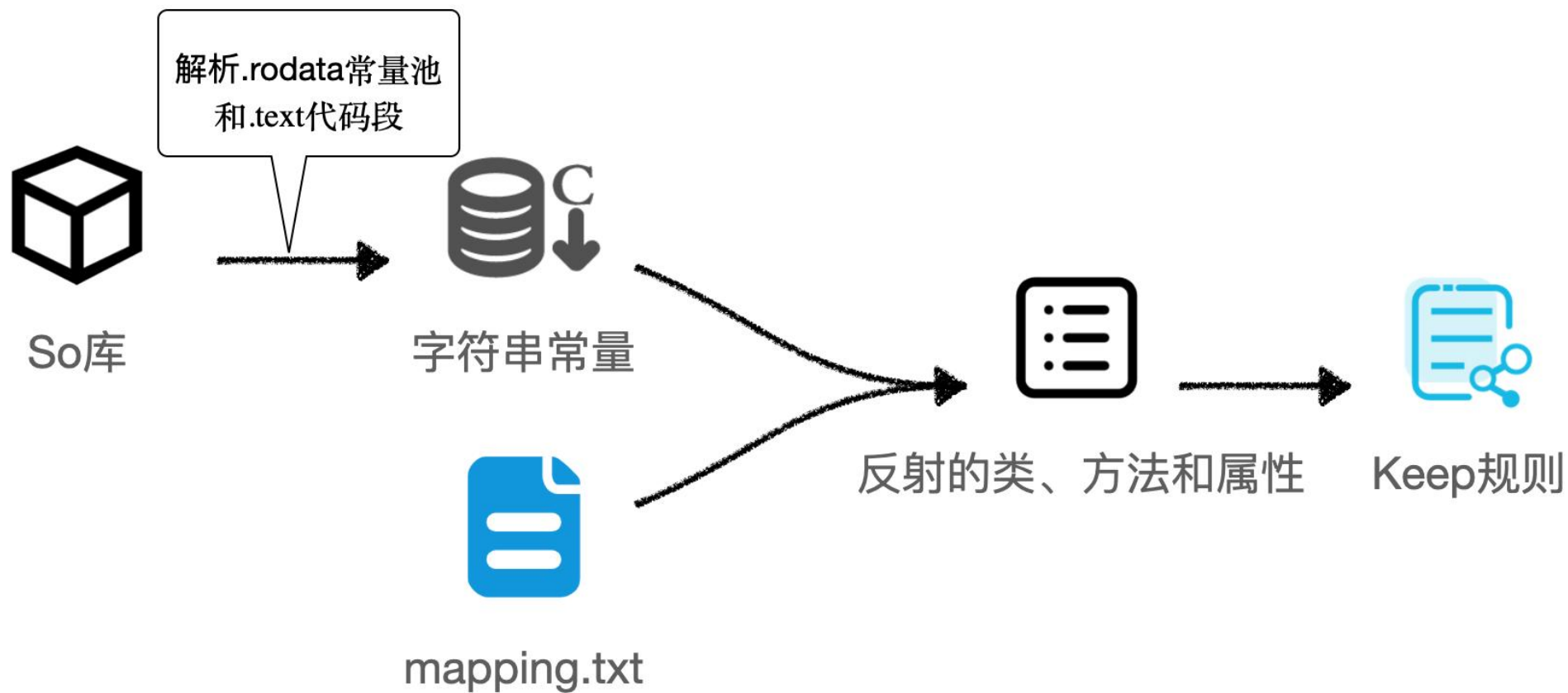


# *Keep* 规则之 JNI

# 举例



# JNI方案



# 落地效果

抖音包体积减少 2.3 M+

# 后期探索

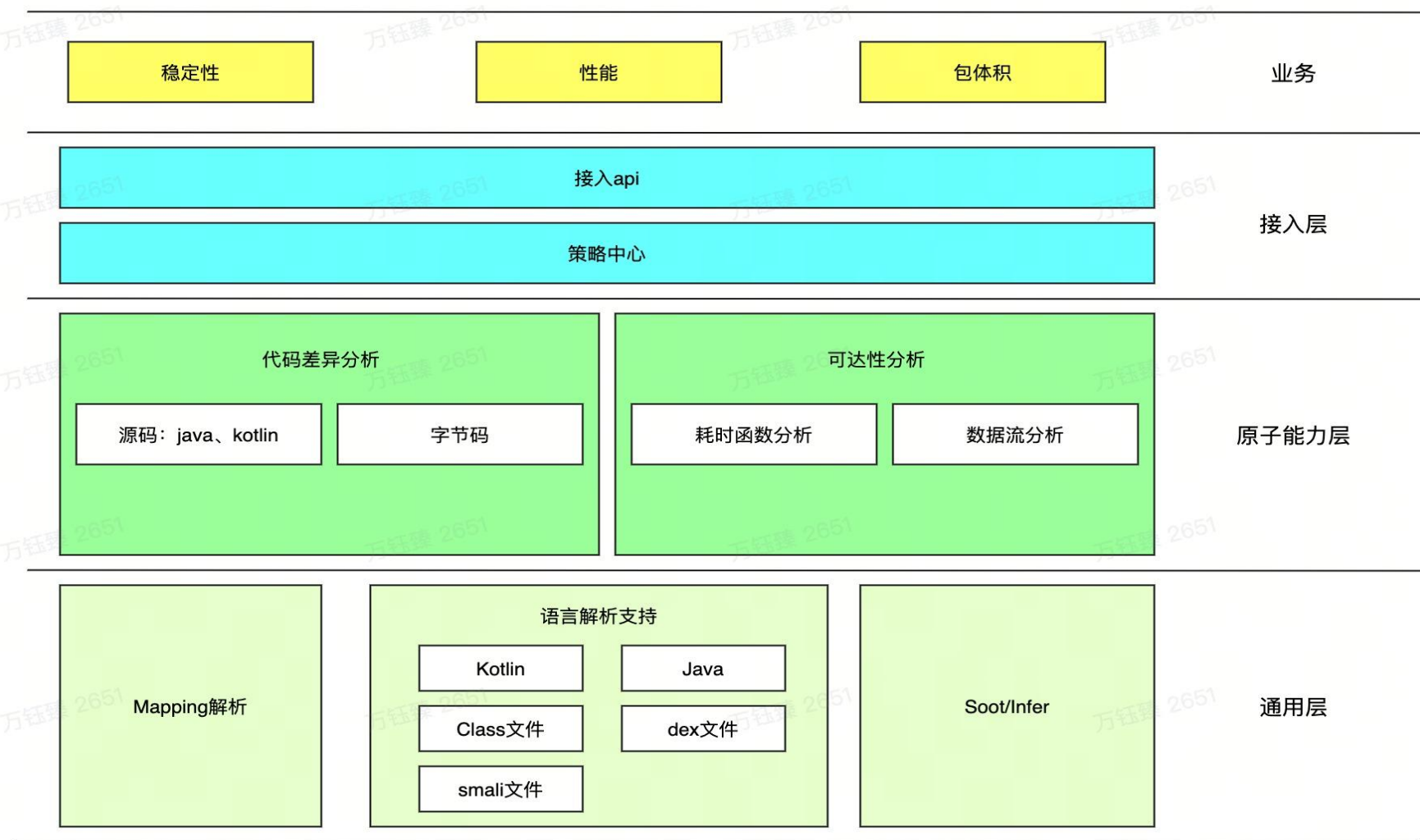
- ⑩ 完善更多 *Keep* 场景，如方法参数通过数组 *Index* 传递等。
- ⑩ 做好 *Keep* 规则防劣化流程。

# 05

## 未来规划

## 03\_未来规划-体系化

我们期望将静态代码  
分析能力整合成平台







*THANKS.*

*April, 2023*