ByteTech

抖音Android包大小代码优化实践

魏馥檀

讲师自我介绍

魏馥檀

2018年本科毕业于南开大学，2019年加入字节跳动抖音Android客户端基础技术团队，长期从事于抖音/抖音极速版包大小相关工作，目前专注于抖音包体积技术优化、包体积平台的开发和技术演进。对Android包体积优化有着较为丰富的经验。

背景介绍

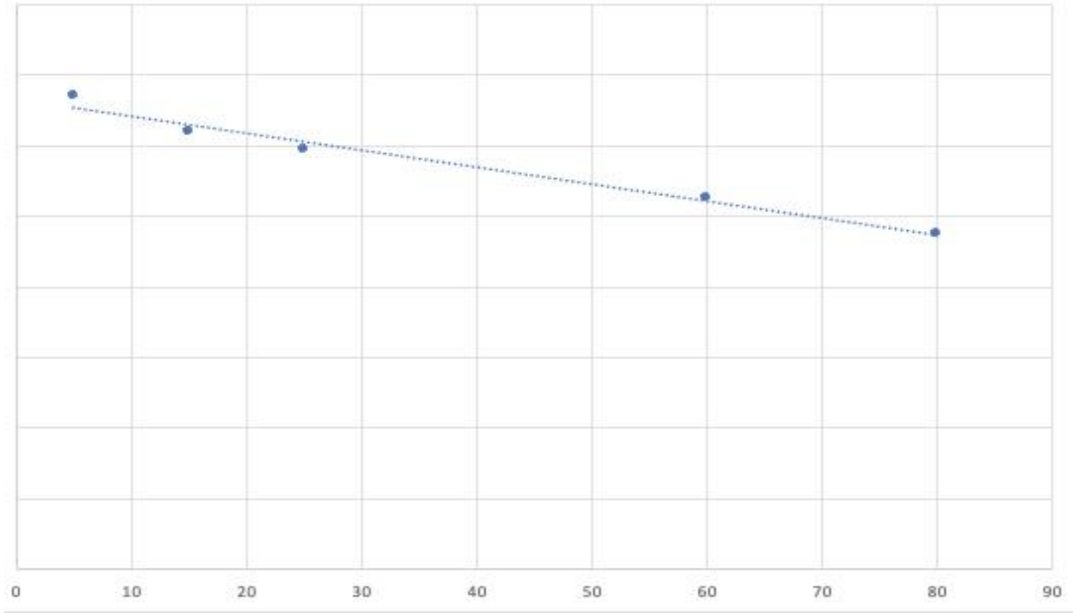
技术优化

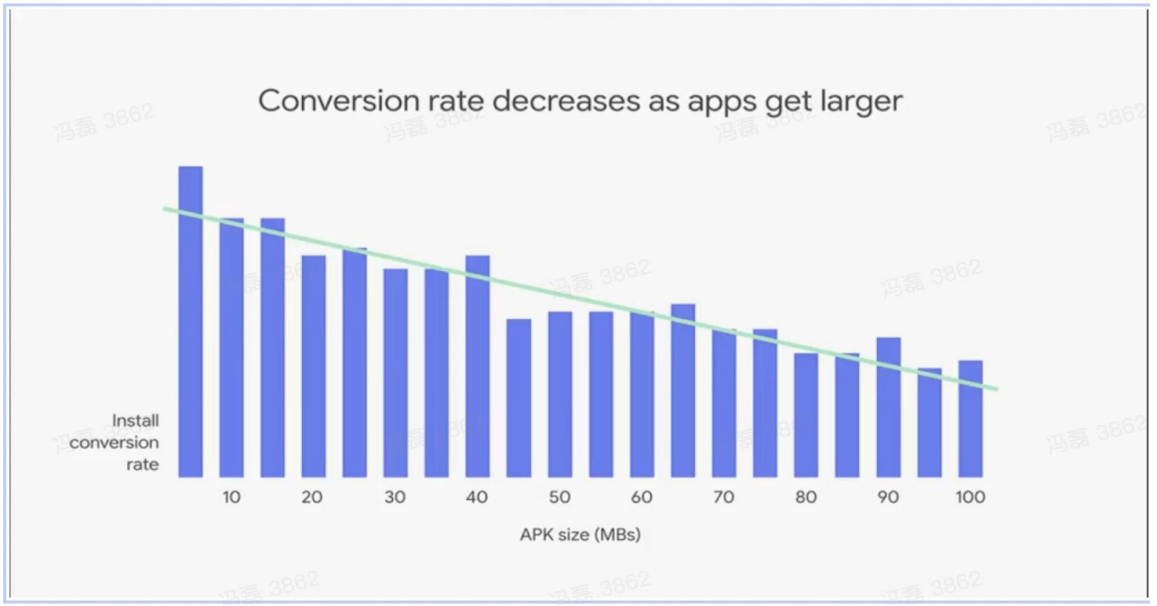
业务优化

总结

01_包体积影响

抖音包体积实验：
- 在大包80M体积情况下，每M的体积增大会减少约0.3%~0.5%的激活量
- 在小包5~15M体积情况下，每M的体积增大会减少约0.7%的激活量





Google包体积数据：
- 整体呈现包体积增大转化率下降的趋势
- 平均每增加1M，激活率下降0.17%，在低端设备较多的区域，该值还会上升

01_为什么从代码开始

- 代码大小占比大，占抖音总大小55%以上
- 当前代码优化工具(ProGuard)能力不如人意
- 针对Dex优化的方案仍然初级
- 代码承载业务需求能力，长时间有大量无用代码产生

代码优化收益大

01_代码优化思路

线下分析　　线上监控

背景介绍

技术优化

业务优化

总结

02_Dex特性

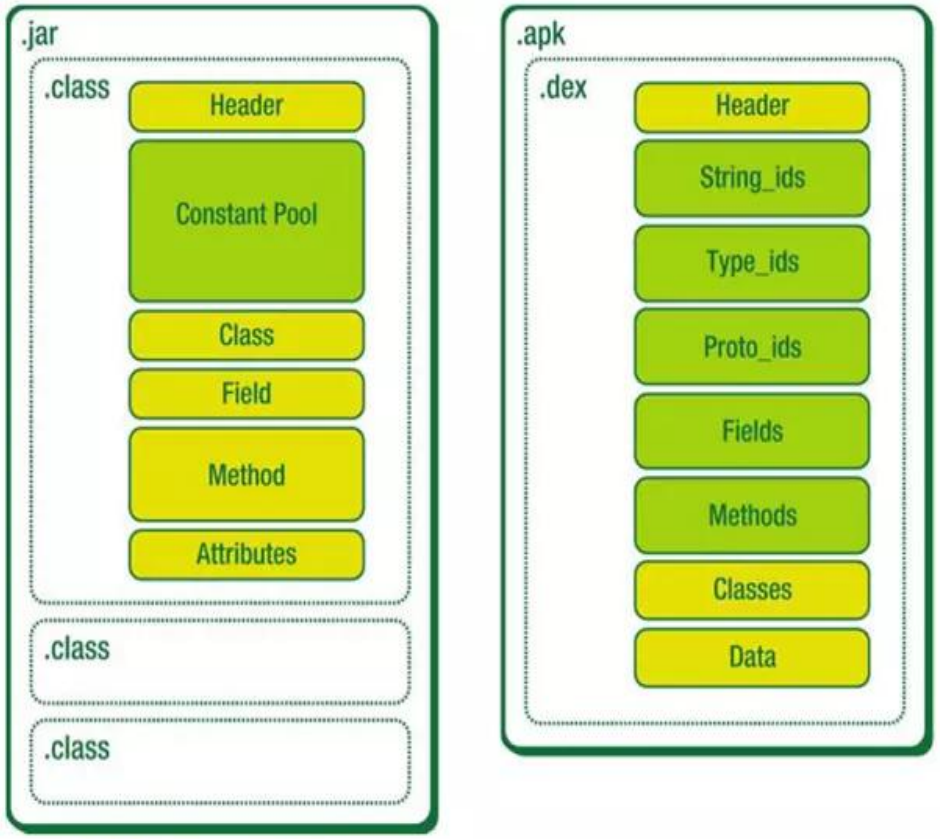
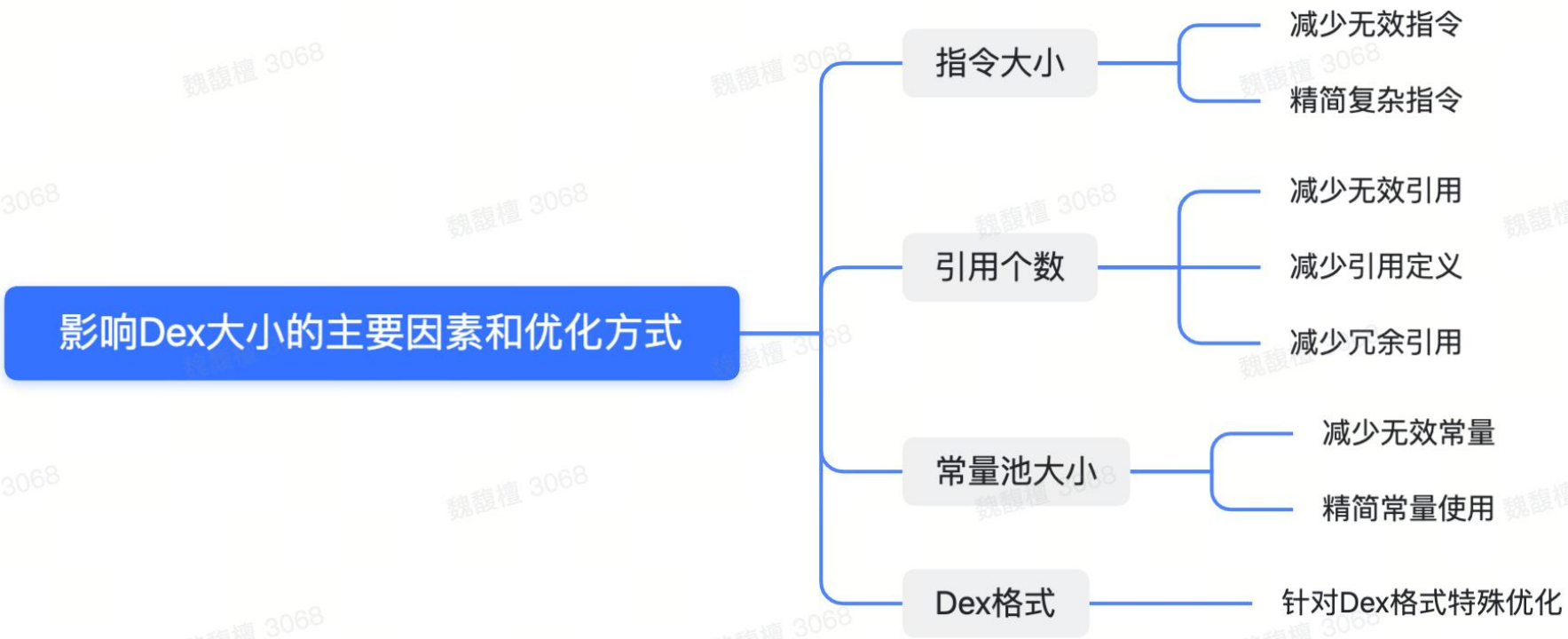


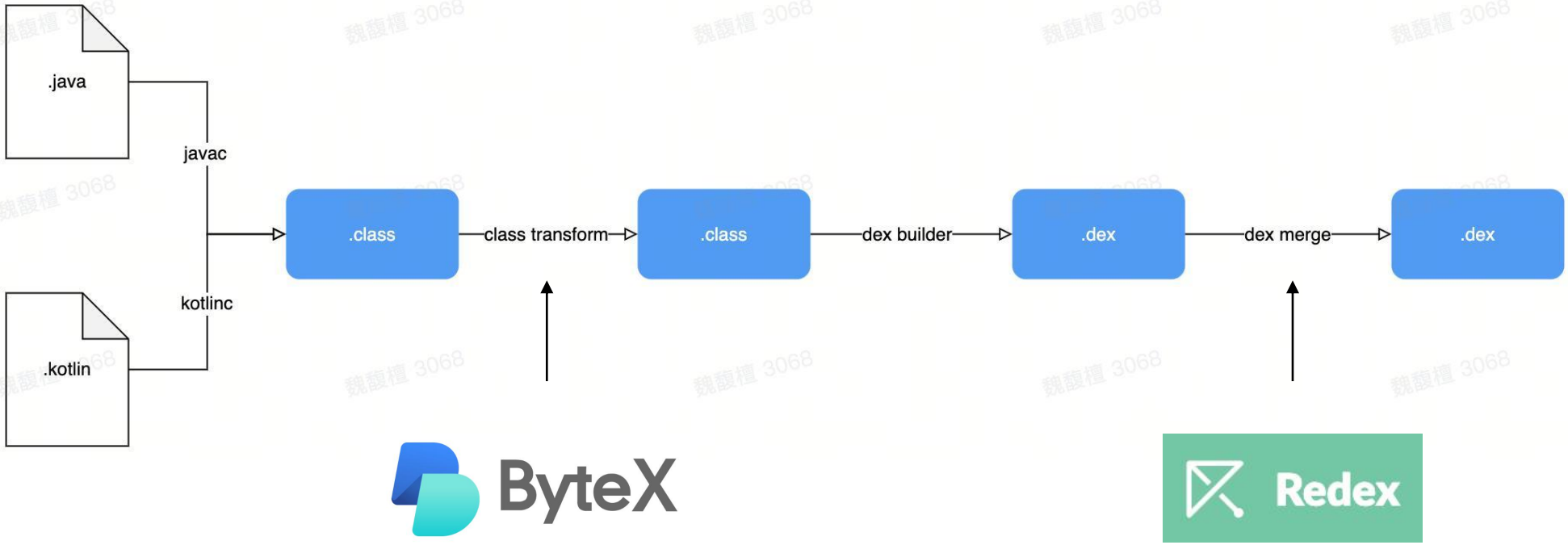
Figure 3-2. *Class file vs DEX file*

- 独特的格式，仅支持Android平台
- 高效的压缩，将Class拆分指令、引用、常量复用
- 单个Dex有引用不超过65536的限制

02_影响因素和优化方式

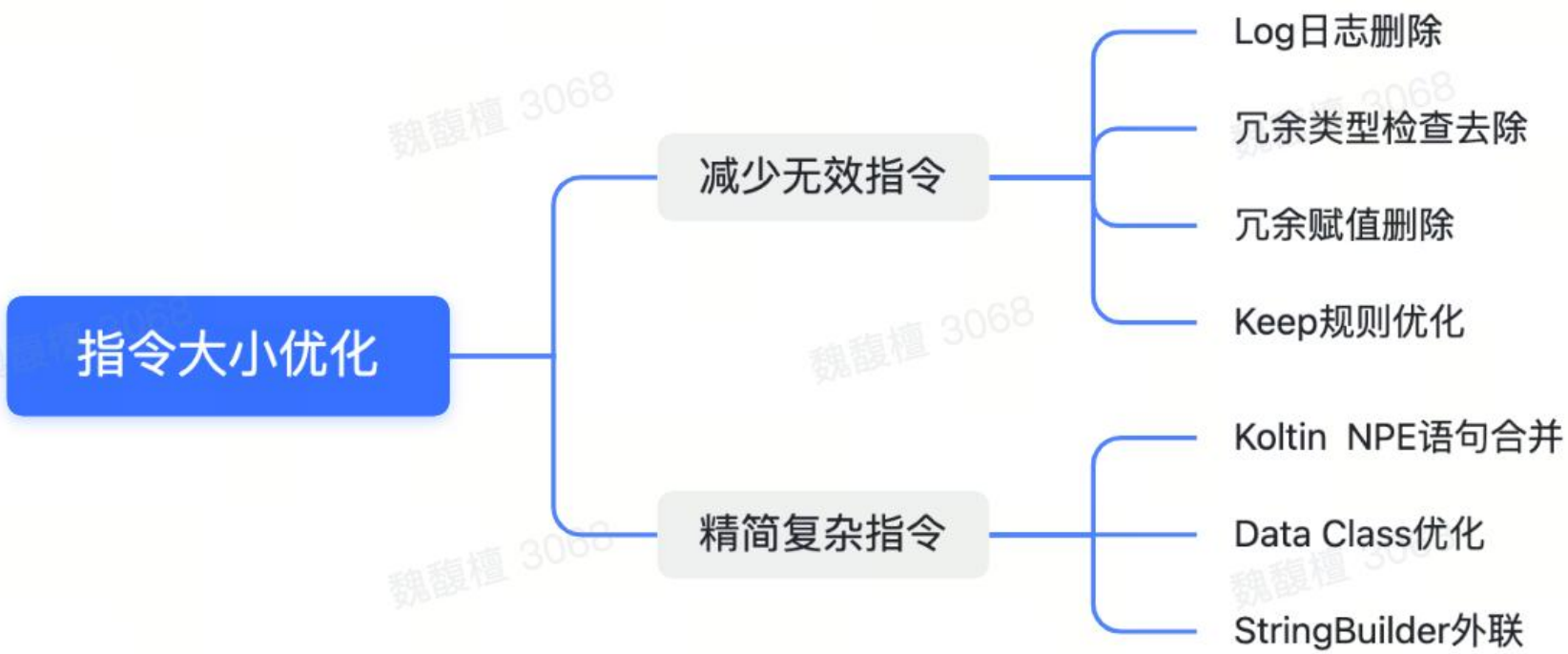


02_优化工具


.java
javac
.kotlin
kotlinc
.class
class transform
.class
dex builder
.dex
dex merge
.dex
ByteX
Redex

02_指令大小优化

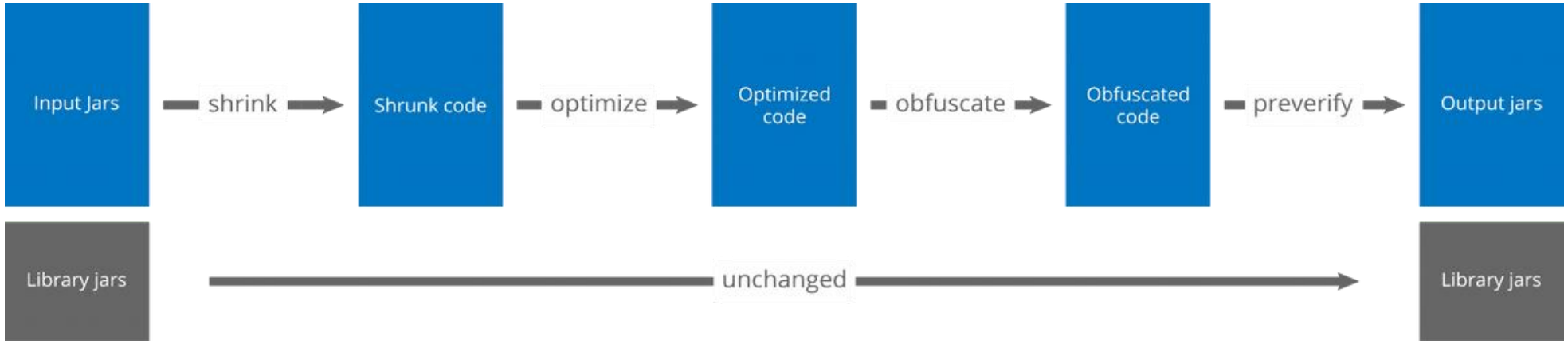
Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

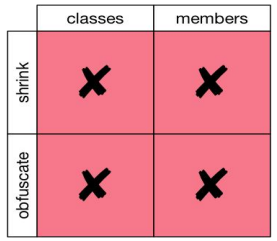


02_Case1 - Keep规则优化

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

proguard流程





Keep的影响

```
-keep class com.ss.android.ugc.** {
    *;
}
```

不合理的keep导致优化效率骤降

❿ 如何发现？

❿ 如何优化？

02_Case1 - Keep规则优化

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

实现发现不合理keep的方案：

修改ProGuard源码，新增Keep影响记录ClassVisitor

```
private void checkKeepThreshold() {
    if (configuration.verbose) {
        System.out.println("Start check whether keep rule reach threshold...");
    }

    File thresholdOutput = null;
    if (configuration.printMapping != null) {
        thresholdOutput = new File(configuration.printMapping.getParentFile(),
"threshold.txt");
    }

    KeepThresholdStatistics keepThresholdStatistics = new
KeepThresholdStatistics(configuration.keep);

programClassPool.accept(keepThresholdStatistics.createClassPoolVisitor(false,
false, true));
libraryClassPool. accept(keepThresholdStatistics.createClassPoolVisitor(false,
false, true));

    keepThresholdStatistics.checkThreshold(thresholdOutput);
}
```

❿ Keep规则的影响范围

```
The keep rule:

-keep class * extends com.squareup.wire.** {
    <fields>;
    <methods>;
}
reach the threshold: [100(class), 1000(field), 1000(method)], actually keep: [1339(class), 6930(field), 12649(method)]
------------
The keep rule:

-keepclasseswithmembers,allowshrinking class * {
    public <init>(android.content.Context,android.util.AttributeSet);
}
reach the threshold: [100(class), 1000(field), 1000(method)], actually keep: [1273(class), 0(field), 1263(method)]
------------
```

⓫ 实现Keep监控

API keep规则检查机器人 BOT 检查新增keep规则及不合预期的keep规则

检测到Keep规则新增/减少：
base commit: 8719de729304bb020206ce8391fe2aa093a99c52
compare_commit: f8ada7ccdcc8839b44316f7230e535e041df895a
link MR: https://cony.bytedance.net/api/workflow/mr/1153/52179
新增/减少详细请参照： https://sf1-hscdn-tos.pstatp.com/obj/developer-baas/baas/tt27sz/72166611c175f5b6_1581494721782.txt

检测到Keep规则新增/减少：
base commit: 8719de729304bb020206ce8391fe2aa093a99c52
compare_commit: f8ada7ccdcc8839b44316f7230e535e041df895a
link MR: https://cony.bytedance.net/api/workflow/mr/1153/52179
新增/减少详细请参照： https://sf3-hscdn-tos.pstatp.com/obj/developer-baas/baas/tt27sz/282eb9f1ef4f2da5_1581494723262.txt

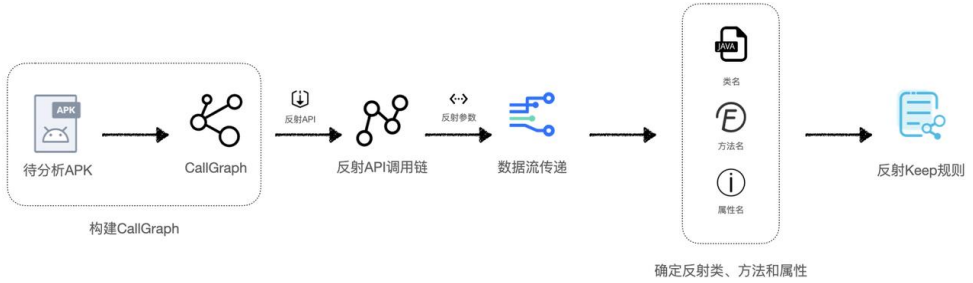
02_Case1 - Keep规则优化

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

早期使用的方案：人工确认优化

- ⑩ 优化不合理规则7条
- ⑩ 减少Dex数量1个
- ⑩ 优化包大小3.5M

更为彻底的方案：静态代码分析



02_Case2 - data class 优化

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

问题背景：

Kotlin data class编译时，会额外生成hashcode，equals，toString等基础方法，包含大量指令且无法自动移除

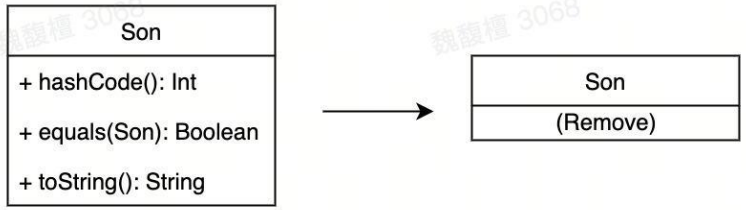
```
data class Son(val name: String, val teacher: String, var age: Int = 0)
```

```
class Son {
    public String name;
    public String teacher;
    public int    age;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Son son = (Son) o;
        if (age != son.age) return false;
        if (!name.equals(son.name)) return false;
        return teacher.equals(son.teacher);
    }

    @Override
    public int hashCode() {
        int result = name.hashCode();
        result = 31 * result + teacher.hashCode();
        result = 31 * result + age;
        return result;
    }

    @Override
    public String toString() {
        return "Son{" +
                "name='" + name + '\'' +
                ", teacher='" + teacher + '\'' +
                ", age=" + age +
                '}';
    }
}
```

02_Case2 - data class 优化

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

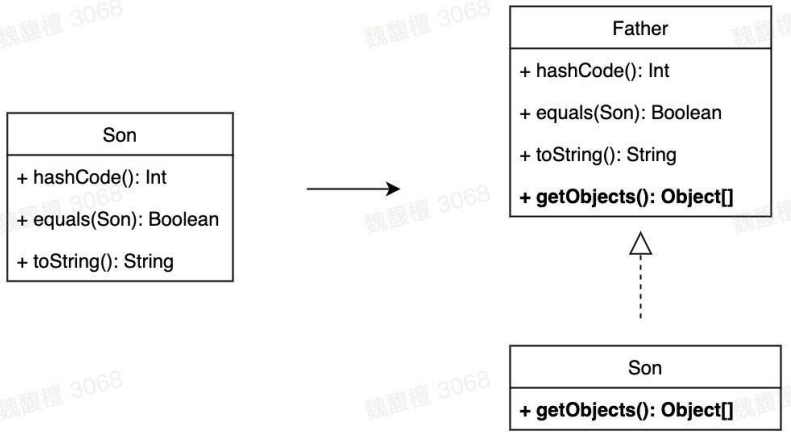
优化方案1：直接移除



优点：优化更彻底

缺点：逻辑不等价，维护成本高

优化方案2：父类实现



优点：逻辑等价，维护成本低

缺点：仍然保留少量指令

02_Case2 - data class 优化

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

最终优化效果：

每个data class 减少2个方法定义，和大量冗余指令，逻辑保持不变。

```
class Son {
    public String name;
    public String teacher;
    public int    age;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Son son = (Son) o;
        if (age != son.age) return false;
        if (!name.equals(son.name)) return false;
        return teacher.equals(son.teacher);
    }

    @Override
    public int hashCode() {
        int result = name.hashCode();
        result = 31 * result + teacher.hashCode();
        result = 31 * result + age;
        return result;
    }

    @Override
    public String toString() {
        return "Son{" +
                "name='" + name + '\'' +
                ", teacher='" + teacher + '\'' +
                ", age=" + age +
                '}';
    }
}
```

优化约300K

```
public abstract class AbstractFather {

    public abstract Object[] getObjects();

    public boolean equals(Object other) {
        if (this == other) {return true;}
        if (other == null) {return false;}
        if (!(this.getClass().isAssignableFrom(other.getClass()))) {
            return false;
        }
        Object[] origins = (this).getObjects();
        Object[] others = ((AbstractFather) other).getObjects();
        for (int i = 0; i < origins.length; i++) {
            if (!Intrinsics.areEqual(origins[i], others[i])) {
                return false;
            }
        }
        return true;
    }

    public int hashCode() { return Objects.hash(getObjects()); }

    public String toString() {
        Object[] objects = getObjects();
        int length = objects.length;
        StringBuilder sb = new StringBuilder(this.getClass().getName());
        for (int i = 0; i < length; i++) {
            if (i == 0) {
                sb.append(":%s");
            } else {
                sb.append(",%s");
            }
        }
        return new Formatter().format(sb.toString(), objects).toString(); // "Son2:%s,%s,%s"
    }
}

public class Son2 extends AbstractFather {
    public String name;
    public String teacher;
    public int    age;
    @Override
    public Object[] getObjects() { return new Object[]{name, teacher, age}; }
}
```

02_Case3 - StringBuilder外联

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

问题背景：

在kotlin/java中使用字符串拼接时，会产生大量的StringBuilder.append()方法的调用。

```
var log = "A" + 1 + "B" + 1.0f + other_var;
```

编译

```
StringBuilder builder = new StringBuilder();
builder.append("A"); builder.append(1);
builder.append("B"); builder.append(1.0f);
builder.append(other_var);
builder.toString();
```

完全相同的代码格式

```
StringBudiler builder = new StringBuilder();
builder.append(A);
builder.append(B);
...
builder.append(Z);
String result = builder.toString();
```

```
String result = StringOutliner.outline(A, B, ..., Z);
```

把公共代码抽取成一个函数进行复用 -- 外联复用

02_Case3 - StringBuilder外联

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

优化方案1：

生成多个类型明确的Outline方法，类型相同的字符串拼接语句可以Outline

```
var log_1 = "A" + 1 + "B" + 1.0f + other_var;
var log_2 = "C" + 2 + "D" + 2.0f + other_var;
var log_3 = "E" + "G" + "F" + 2.0L + other_var;
```

外联

```
// StringOutliner.outline(String, Int, String, Float, Object);
var log_1 = StringOutliner.outline("A", 1, "B", 1.0f, other_var);
var log_2 = StringOutliner.outline("C", 2, "D", 2.0f, other_var);

// StringOutliner.outline(String, String, String, Long, Object);
var log_3 = StringOutliner.outline("E", "G", "F", 2.0L, other_var);
```

- 生成大量Outline方法
- 单个Outline方法复用率较低

劣化约200K

02_Case3 - StringBuilder外联

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

优化方案2：

生成固定个数Object类型的Outline方法，数量相同的字符串拼接语句可以Outline

```
var log_1 = "A" + 1 + "B" + 1.0f + other_var;
var log_2 = "C" + 2 + "D" + 2.0f + other_var;
var log_3 = "E" + "G" + "F" + 2.0L + other_var;
```

外联

```
// StringOutliner.outline(Object, Object, Object, Object, Object);
var log_1 = StringOutliner.outline("A", Int.valueOf(1), "B", Float.valueOf(1.0f),
other_var);
var log_2 = StringOutliner.outline("C", Int.valueOf(2), "D", Float.valueOf(2.0f),
other_var);
var log_3 = StringOutliner.outline("E", "G", "F", Long.valueOf(2.0L), other_var);
```

- 生成少量Outline方法
- 单个Outline方法复用率很高

装箱问题如何解决？

02_Case3 - StringBuilder外联

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

装箱问题Balance：

```
// StringOutliner.outline(int);
const/4 v0 1
invoke-static {v0} LStringOutliner.outline(I);Ljava/lang/String;
move-result-object {v2}
```

VS

```
// StringOutliner.outline(Object);
const/4 v0 1
invoke-static {v0} Ljava/lang/Integer.valueOf(I);Ljava/lang/Integer;
move-result-object {v1}
invoke-static {v0} LStringOutliner.outline(Ljava/lang/Object;);Ljava/lang/String;
move-result-object {v2}
```

基本类型装箱会增加2条指令

移除一个引用类型拼接时

-1

移除一个基本类型拼接时

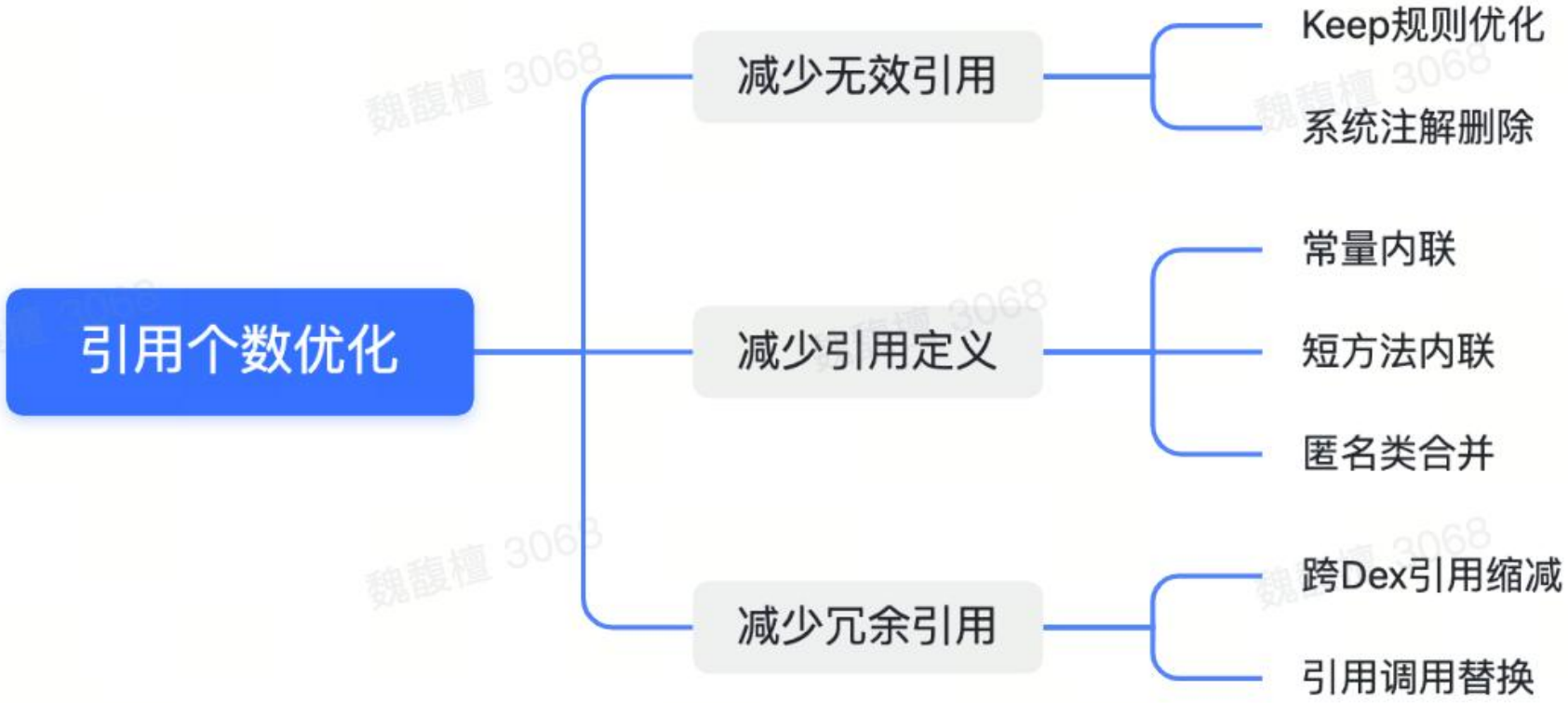
-1 append , +1 box, +1 move-

当且仅当引用类型数量大于基本类型数时，
进行StringBuilder外联

优化约200K

02_引用大小优化

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

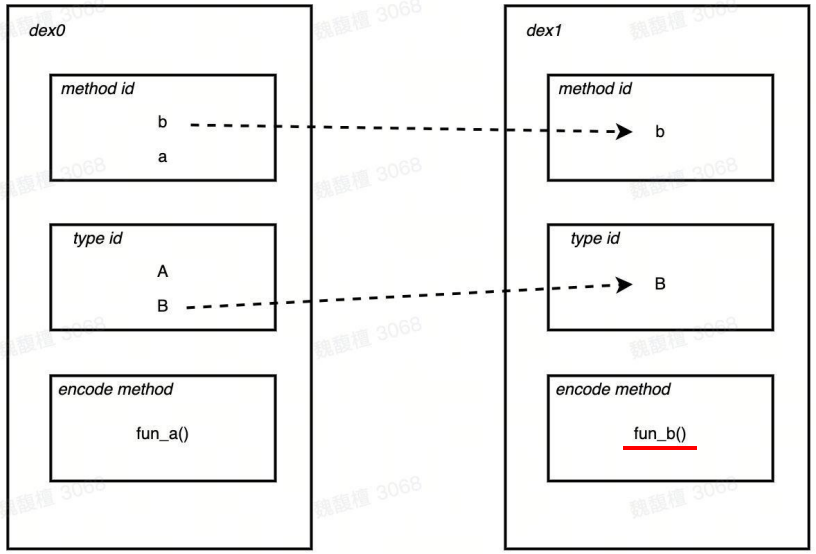


02_Case4 - 跨Dex引用缩减

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

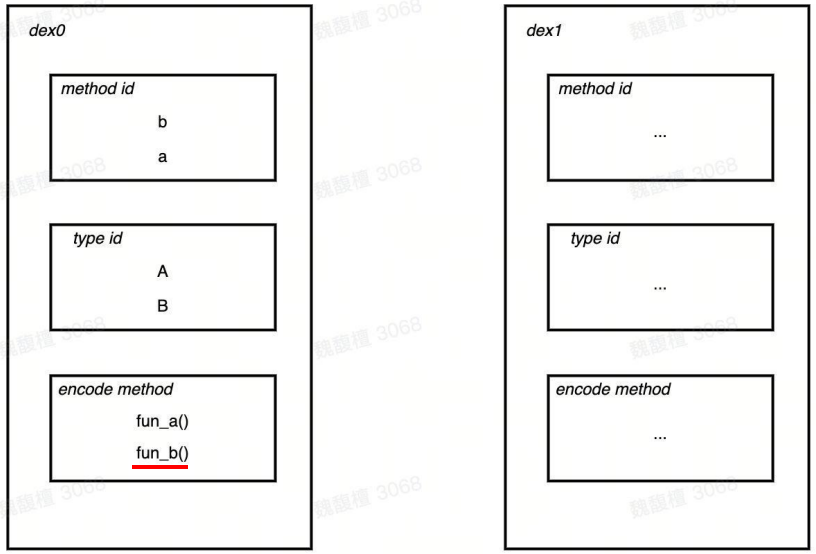
问题背景：

Dex分包问题导致的冗余引用产生



如何优化？

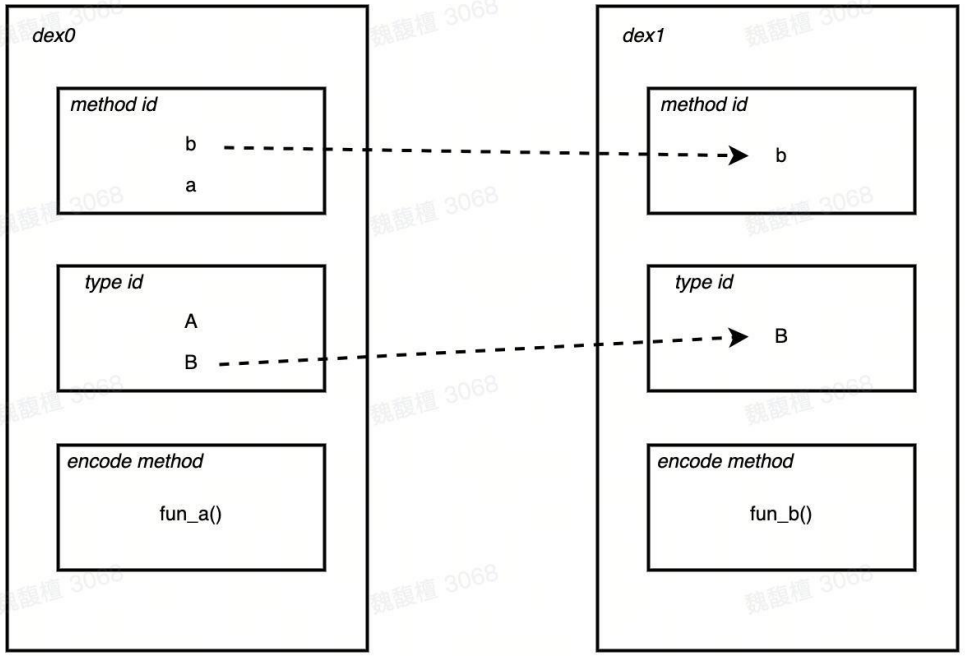
根据引用关系对Dex进行重排序



将Class B的定义转移到dex0中，将减少2个引用定义

02_Case4 - 跨Dex引用缩减

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

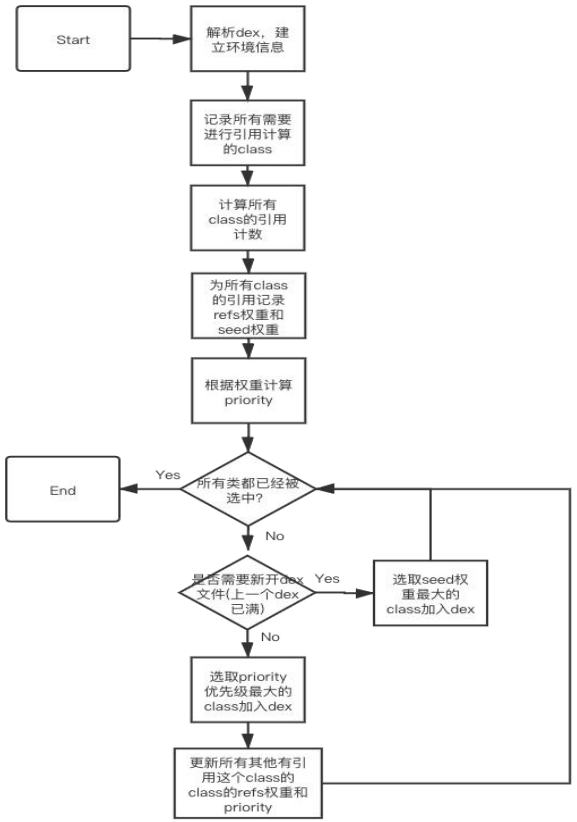


优化核心：Dex中引入Class时，增加已复用引用大小，减少未复用引用大小

Class引入优先级 = 已引入引用大小 / 未引入引用大小

02_Case4 - 跨Dex引用缩减

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化



每次选择最高优先级Class放入Dex

↓

贪心算法：容易陷入局部最优解

其他启发式算法？

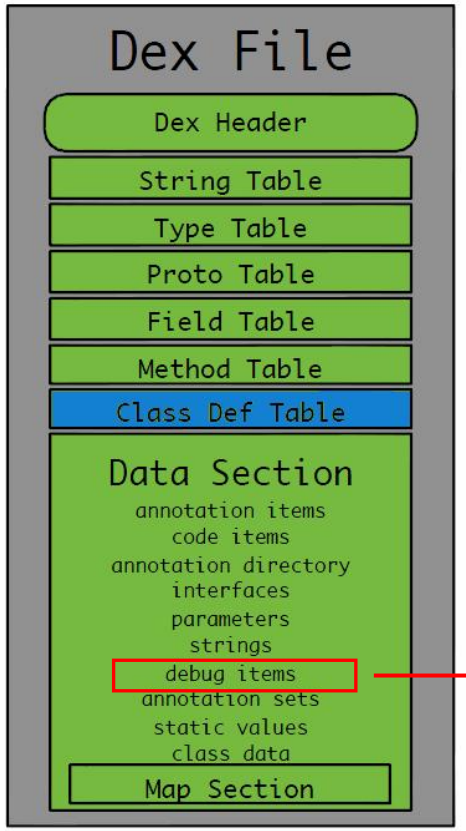
抖音优化

500K

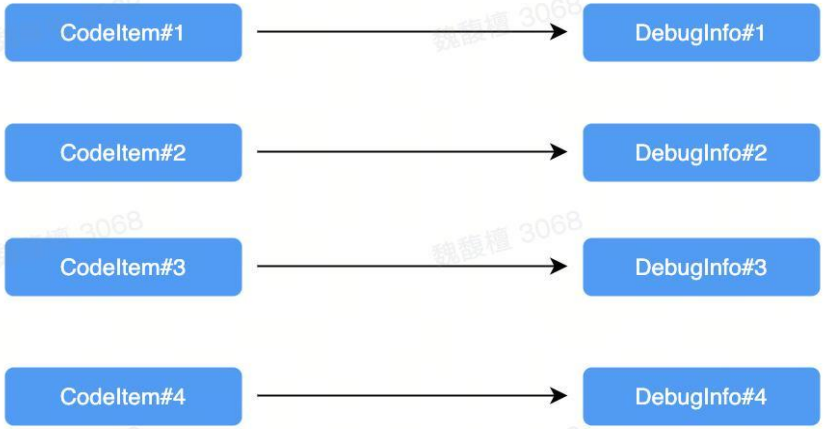
02_Dex格式优化

02_Case5 - DebugInfo缩减

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化



为什么这么大？



- 每个CodeItem对应一个DebugInfo

DebugInfo是Dex中用于单步调试和堆栈行号信息获取的二进制段

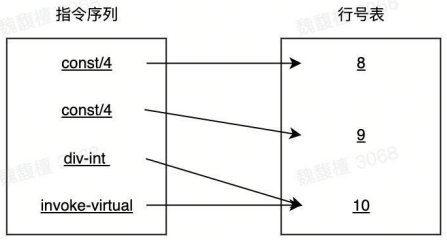
大小约占Dex大小的5.5%，抖音约占5M大小

02_Case5 - DebugInfo缩减

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

关键点：行号偏移转指令偏移

行号偏移

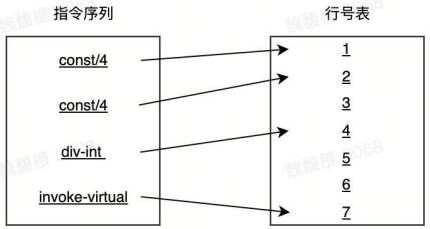



```
java.lang.ArithmeticException: / by zero
        at com.bytedance.android.test.getDiv(GetDivTest.java:10)
        at com.bytedance.android.test.GetDivTest.getDivTest(GetDivTest.java:15)
```

div-int指令崩溃，对应行号10

```
7    private void getDiv() {
8        int a = 0;
9        int b = 1;
10       System.out.println(b / a);
11   }
```

指令偏移




```
java.lang.ArithmeticException: / by zero
        at com.bytedance.android.test.getDiv(GetDivTest.java:4)
        at com.bytedance.android.test.GetDivTest.getDivTest(GetDivTest.java:15)
```

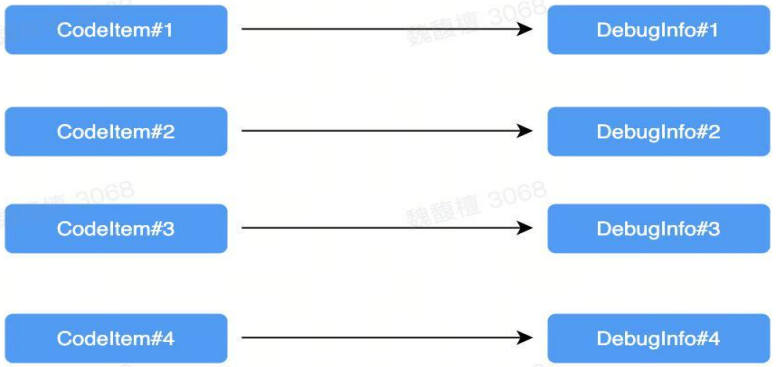
对应崩溃发生指令偏移4

崩溃堆栈行号	实际源码行号
0 - 1	8
2	9
3-7	10
>8	无效

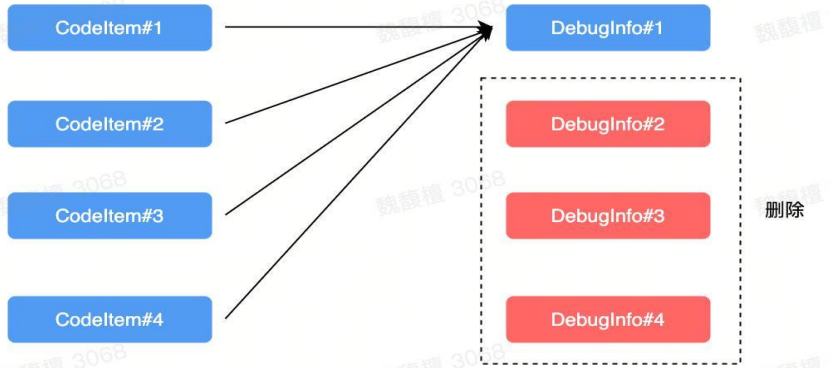
解析崩溃行号为10

02_Case5 - DebugInfo缩减

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化



DebugInfo复用




```
DSLN INTEGRATED MAPPING FILE
DSLN Mapping Start
DSLN version = 0.1.32
GIT SHA:1c4654a0824d610ef32bbee3a0d55325e859697d
BUILD TIME:2023-03-16 08:07:37
J.N <init> [V@] [0:11:4:-1]
ShootScanQrCodeResolution$heightArray$2 invoke [Ljava/lang/Object;@] [196611:1356:196626:-1]
ShootScanQrCodeResolution$value$2 invoke [Ljava/lang/Object;@] [262144:350:262179:1363:262202:-1]
ShootScanQrCodeResolution$widthArray$2 invoke [Ljava/lang/Object;@] [196611:1352:196626:-1]
X.FQi$a <init> [V@LX/FQi;] [16842752:643:16842758:-1]
X.FQi$a LIZ [I@I] [16908288:669:16908297:-1]
X.FQi$a LIZ [Ljava/util/ArrayList;@LX/FQi$a;] [16777216:643:16777219:-1]
X.FQi$a LIZIZ [Ljava/util/ArrayList;@LX/FQi$a;] [16777216:643:16777219:-1]
X.FQi$a LIZJ [Ljava/util/ArrayList;@LX/FQi$a;] [16777216:643:16777219:-1]
X.FQi$a LIZLLL [Ljava/util/ArrayList;@LX/FQi$a;] [16777216:643:16777219:-1]
X.FQi$a LJ [Ljava/util/ArrayList;@LX/FQi$a;] [16777216:643:16777219:-1]
```

行号解析表

包含指令偏移和行号映射的Mapping文件

实现DebugInfo复用：

- 一个足够大的DebugInfo
- 每个方法独立的行号解析表

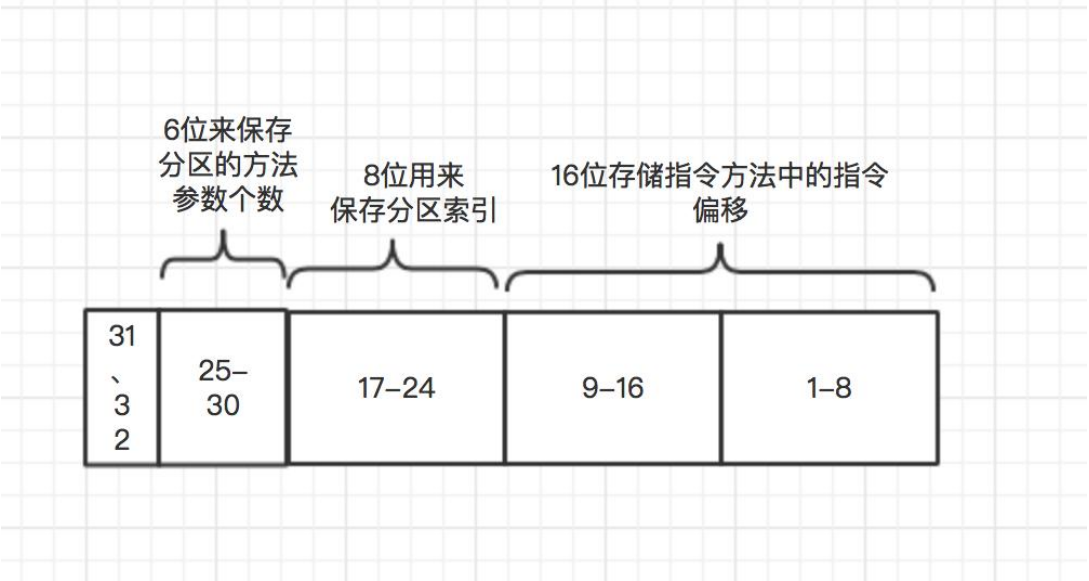
02_Case5 - DebugInfo缩减

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

分区问题：

- 5.x，6.x机型，DebugInfo参数个数与方法参数个数不对应时，不返回崩溃行号
- 同类同名方法崩溃时，需要通过崩溃行号区分具体崩溃方法
- 过大的DebugInfo会增加dex2Oat执行的效率和内存占用

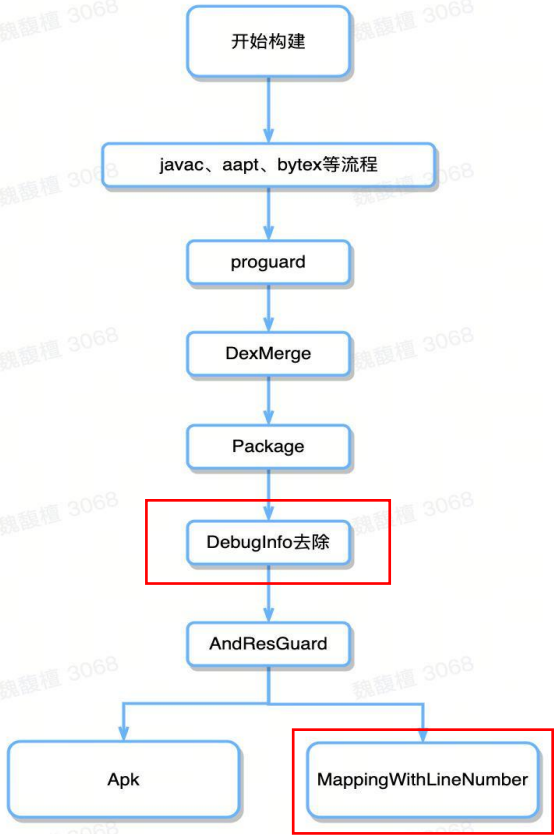
17-30位记录分区信息，不同分区信息使用不同DebugInfo



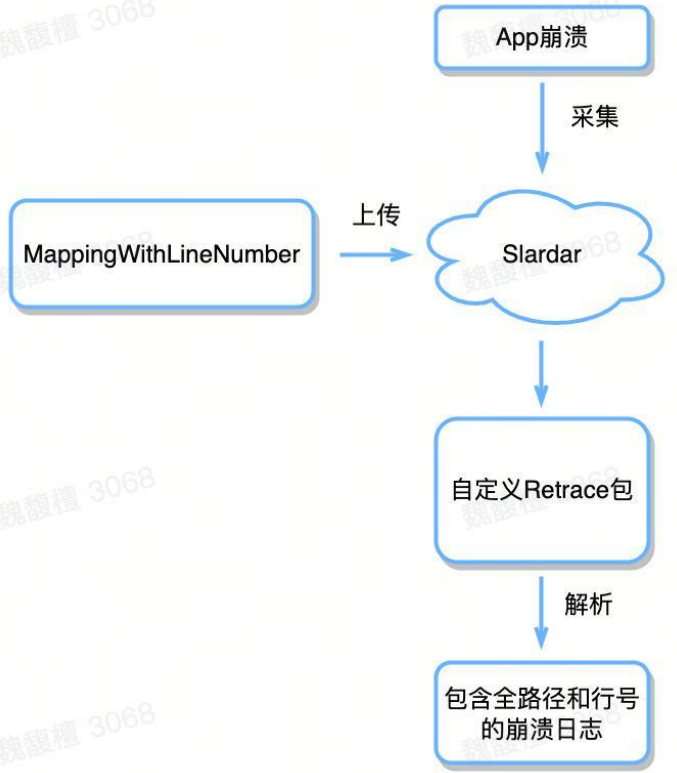
02_Case5 - DebugInfo缩减

Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

构建流程

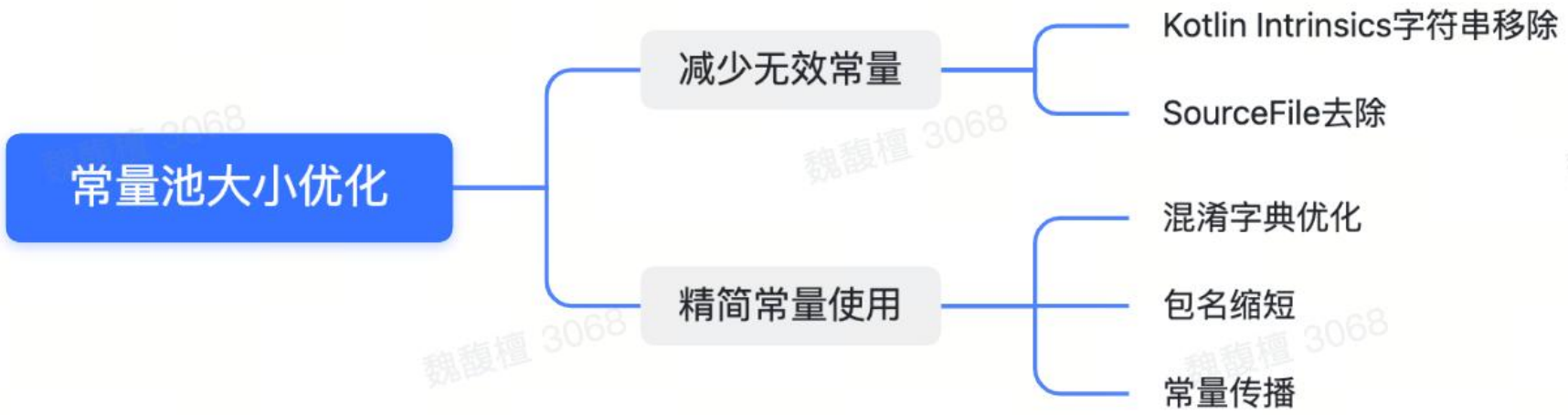


解析流程



02_ 常量池大小优化

Dex优化 → 指令优化 → 引用优化 → 格式优化 → 常量优化



02_Case6 - 包名缩短

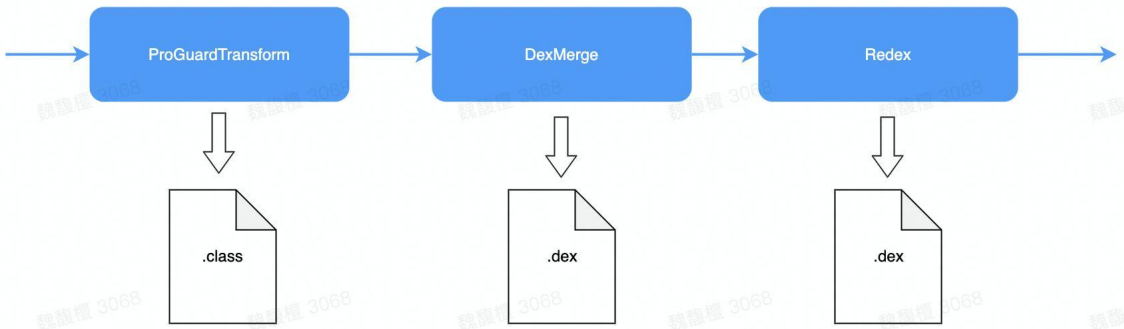
Dex优化 | 指令优化 | 引用优化 | 格式优化 | 常量优化

```
androidx.activity.OnBackPressedCallback -> androidx.activity.c:
    boolean mEnabled -> a
    java.util.concurrent.CopyOnWriteArrayList mCancellables -> b
    46:56:void <init>(boolean) -> <init>
    91:94:void remove() -> a
```

缩短被混淆类的包名长度

```
androidx.activity.OnBackPressedCallback -> X.c:
    boolean mEnabled -> a
    java.util.concurrent.CopyOnWriteArrayList mCancellables -> b
    46:56:void <init>(boolean) -> <init>
    91:94:void remove() -> a
```

为什么不用ProGuard repackage?



ProGuard在Dex生成前缩短包名，会改变Dex中的Class集合导致优化**结果不稳定**。

Redex在Dex生成后缩短包名，不会改变Dex中的Class集合，优化**结果更稳定**。

抖音优化**1M+**

背景介绍

技术优化

业务优化

总结

03_线上代码覆盖率

原理：在所有Class的静态方法块中插桩，在类被加载时将对应Index的01状态记录到Bitmap中

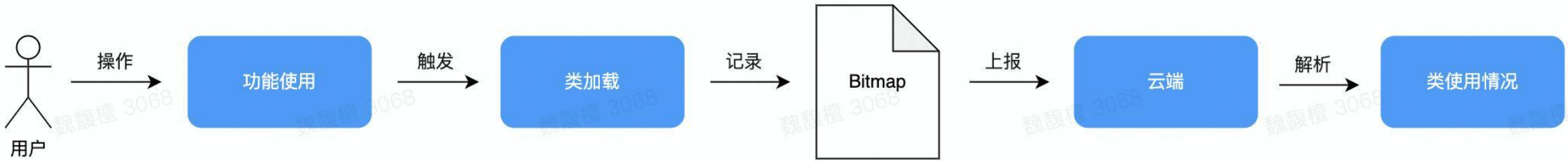
Index	0 1 2 3 4 5 6 7 8
Bitmap	1 0 0 1 0 0 0 1 0

0/1状态表示对应类的使用情况

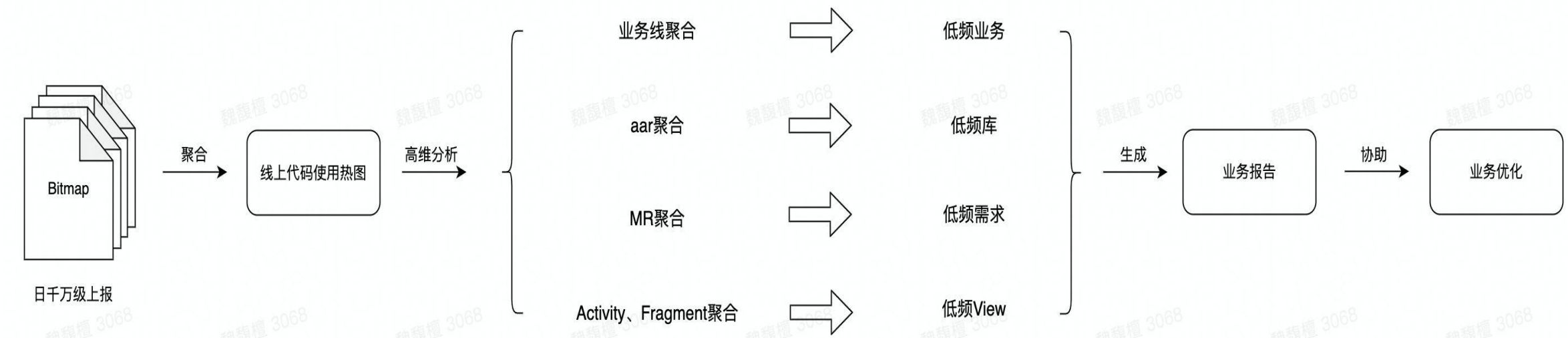
```
public class DouyinPlayer {

    public Player mPlayer;

    public void start() {
        mPlayer.start();
    }
}
```

插桩

```
public class DouyinPlayer {

    static {
        Covode.collect(1002);
    }

    public Player mPlayer;

    public void start() {
        mPlayer.start();
    }
}
```



03_线上代码覆盖率

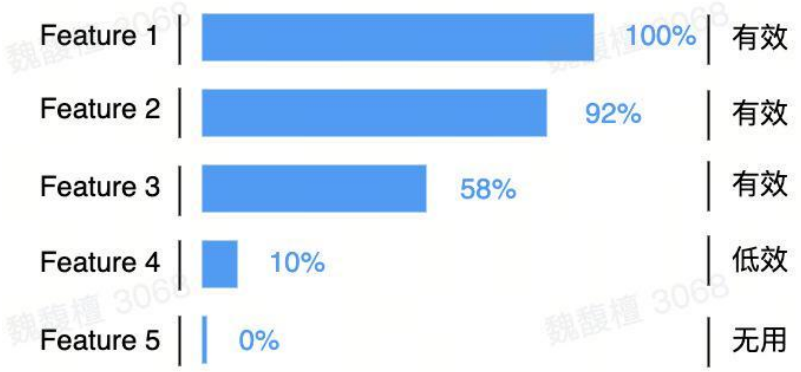

Bitmap
日千万级上报
聚合
线上代码使用热图
高维分析
业务线聚合
aar聚合
MR聚合
Activity、Fragment聚合
低频业务
低频库
低频需求
低频View
生成
业务报告
协助
业务优化

03_ 需求有效率

上线需求 + 代码覆盖率 = ？

历史需求新增类的线上覆盖比例

- 已上线需求，新增类列表固定
- 代码覆盖率可持续监控类使用情况



需求有效率

- 可追溯
- 持续监控

⑩ 业务线下需求40+

⑩ 额外消费无用Class 3000+

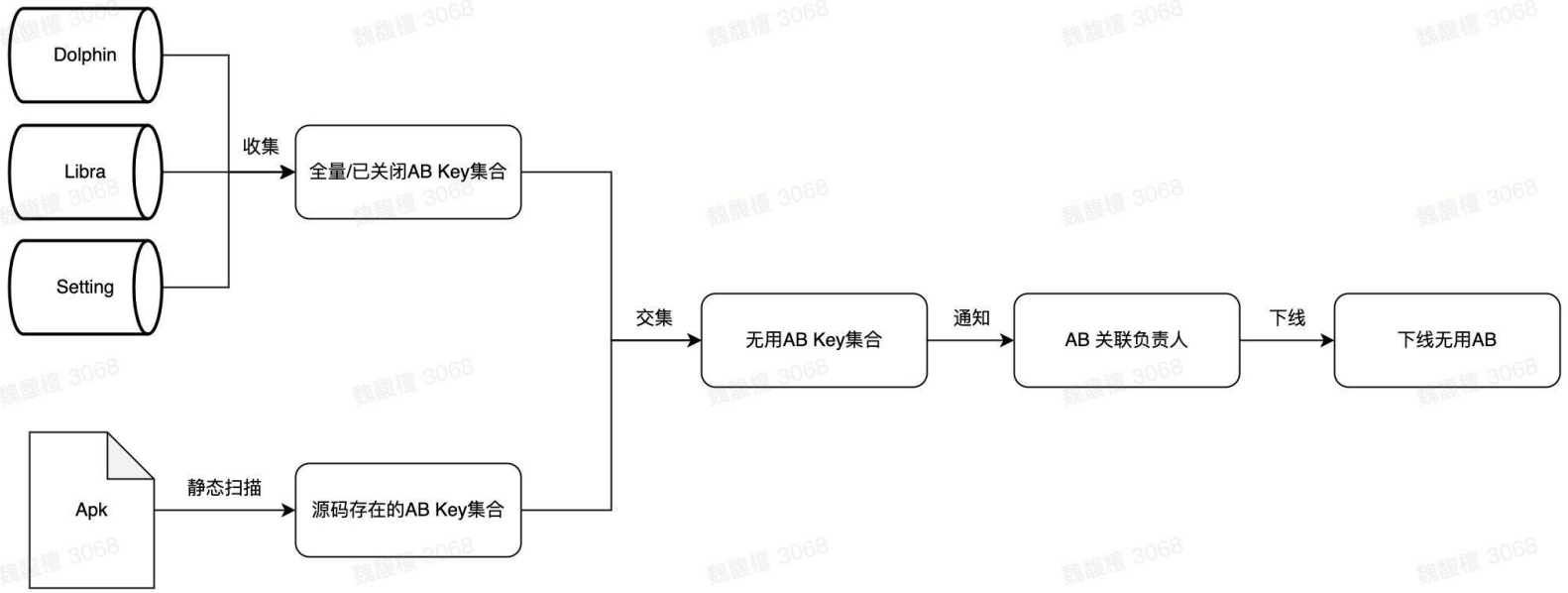
⑩ 累计收益2M+

03_无用AB消费

问题背景：

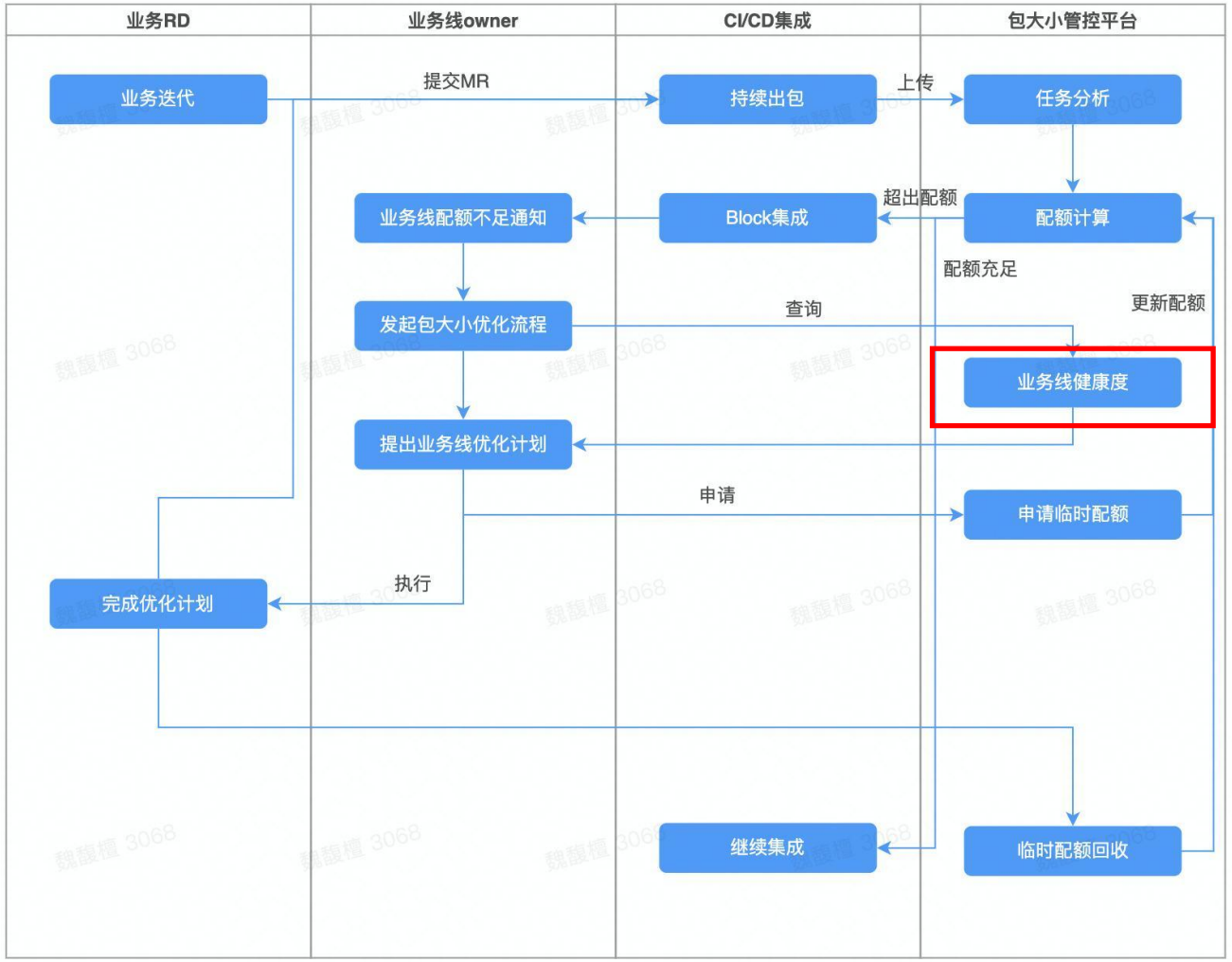
AB实验在全量或者关闭后，源码层面并没有删除，占据代码空间

定期提醒，推进下线



通过代码的静态分析和AB平台数据综合计算

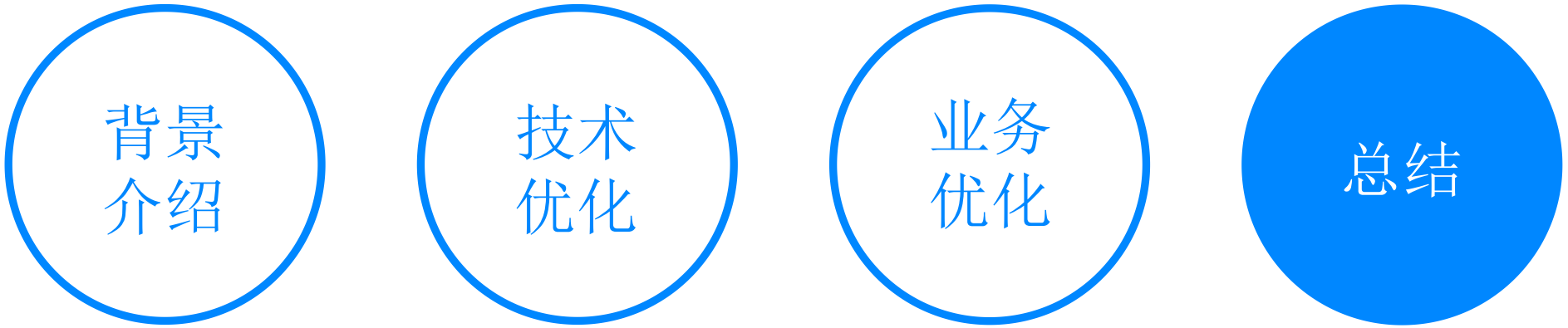
03_业务管控



包含代码覆盖率数据，要求消费：

- 无用库
- 无用Activity/Fragment
- 无用需求
- 无用AB

通过管控，促进消费，实现闭环


背景
介绍
技术
优化
业务
优化
总结

04_总结

- 优化思路
  - 线下技术优化
    - 移除无用代码(ProGuard Keep优化)
    - 从指令、引用、常量池、格式等优化Dex大小
  - 线上业务优化
    - 代码覆盖率监控
    - 无用AB实验下线
    - 与业务管控结合
- 优化工具
  - ByteX
  - Redex

预告：

⑩ 如何极致的优化图片大小？

⑩ 如何实现业务无感知的图片上云？

⑩ 如何监控资源在线上的使用情况？

下期抖音Android资源优化实践与你共享！

04_致谢

感谢**西瓜基础技术团队张祖桥、AppInfa团队冯瑞**同学对本次演讲内容的大力支持

ByteTech

THANKS.