



抖音性能防劣化平台极致演绎之路

曹冬平



曹冬平

2015 年开始致力于 android 开发，先后就职于百度和爱奇艺，期间专注于基础性能与稳定性优化。在 2022 年 1 月加入字节跳动抖音基础技术团队，负责抖音性能防劣化平台的相关工作。



01 背景介绍

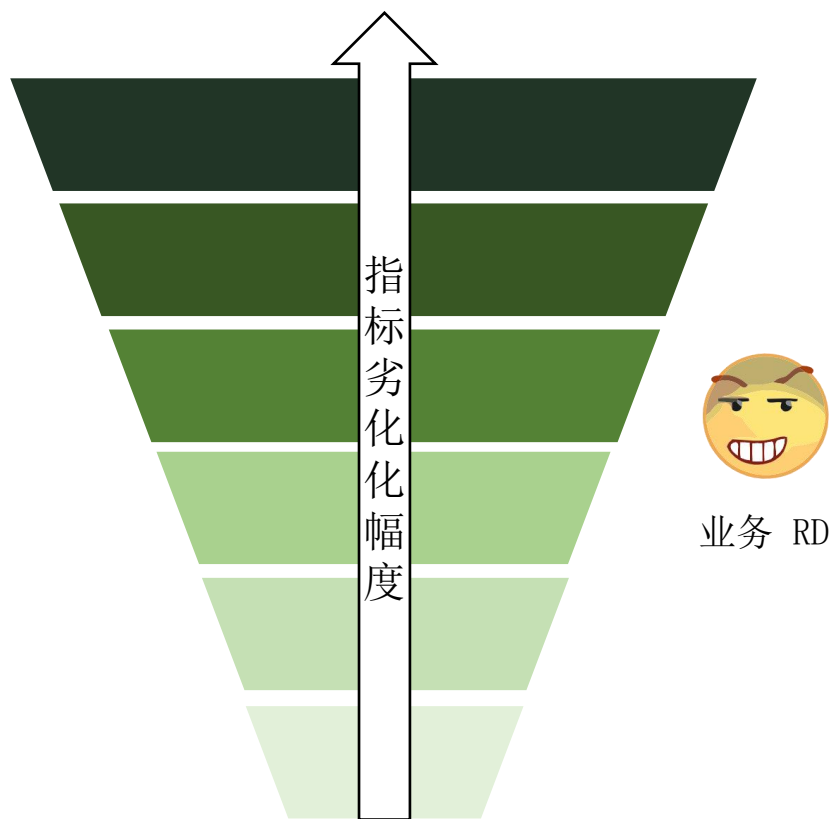
02 体系建设

03 落地效果

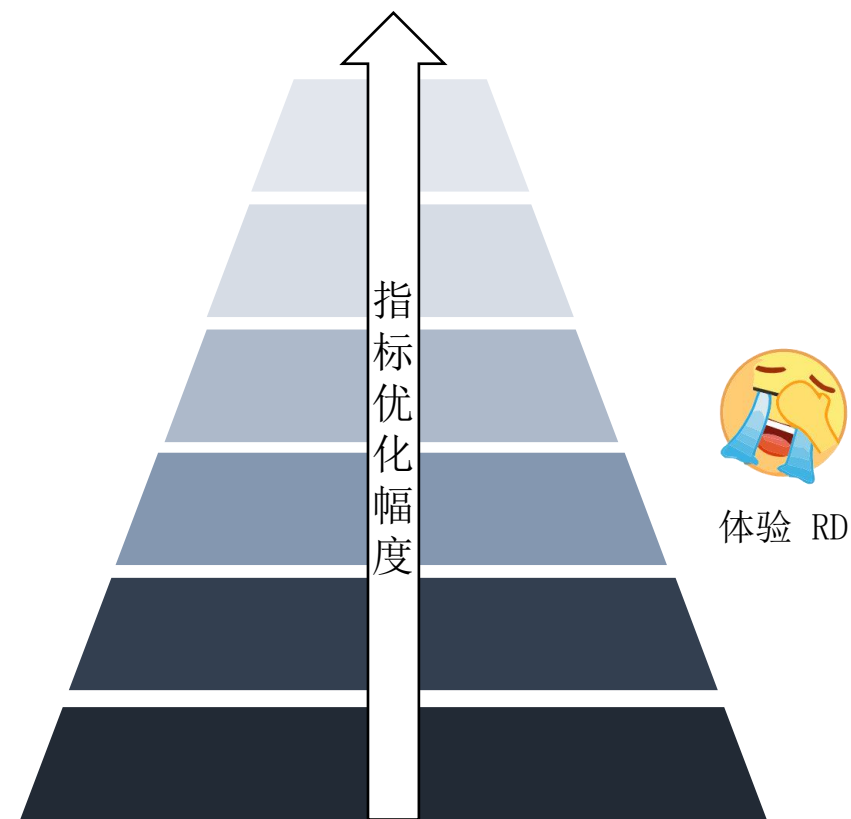
04 展望未来

背景介绍

为什么需要性能防劣化



每天合入需求约 **1000** 个!

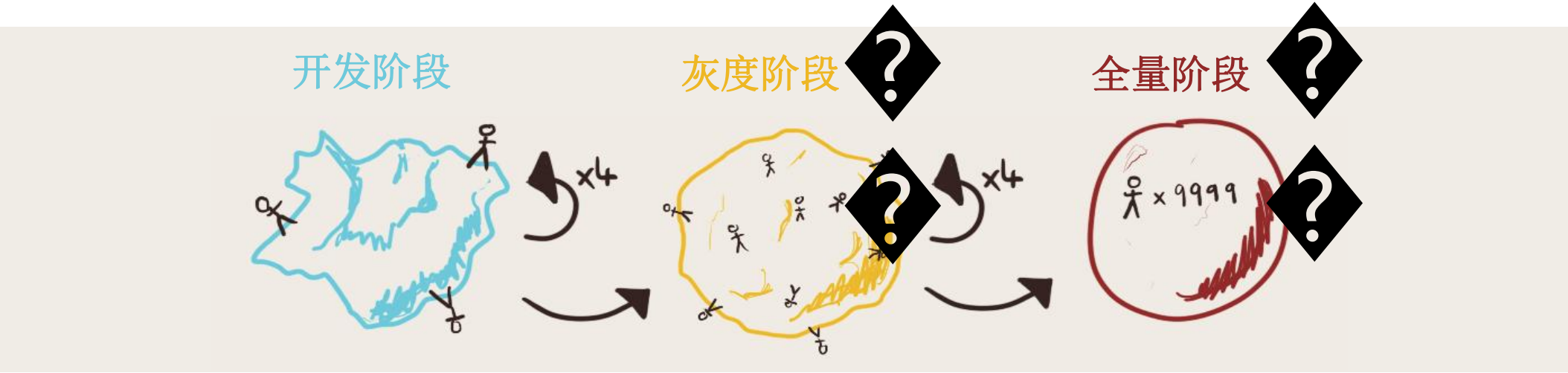


双月启动劣化个数 **30+**，累计约
150ms
劣化排查 **8** 人日

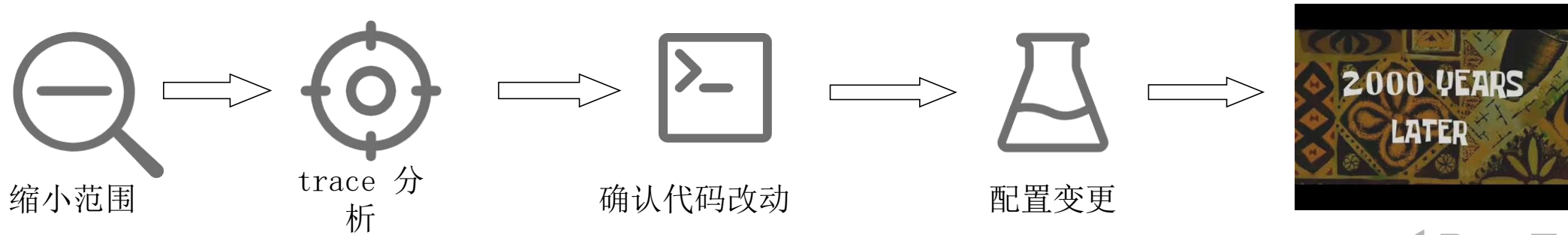
我们亟需建设性能防劣化平台

防劣化现状

1. 发现劣化：大多数情况下是在灰度/全量阶段报警发现问题



2. 归因劣化：基本通过线下手动归因



现状问题




劣化发现
时间晚

留给问题修复的**时间少**



劣化归因
效率低

纯人工归因且**周期长**



劣化归因
手段少

疑难劣化问题**定位难**

我们的思路



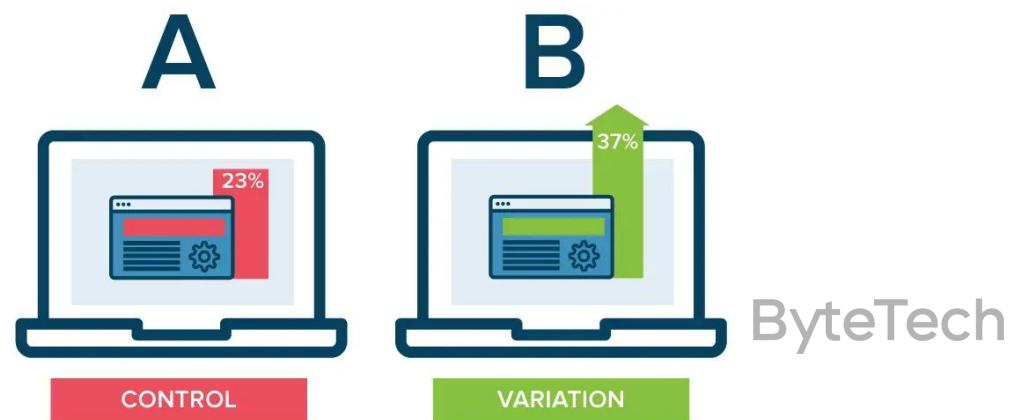
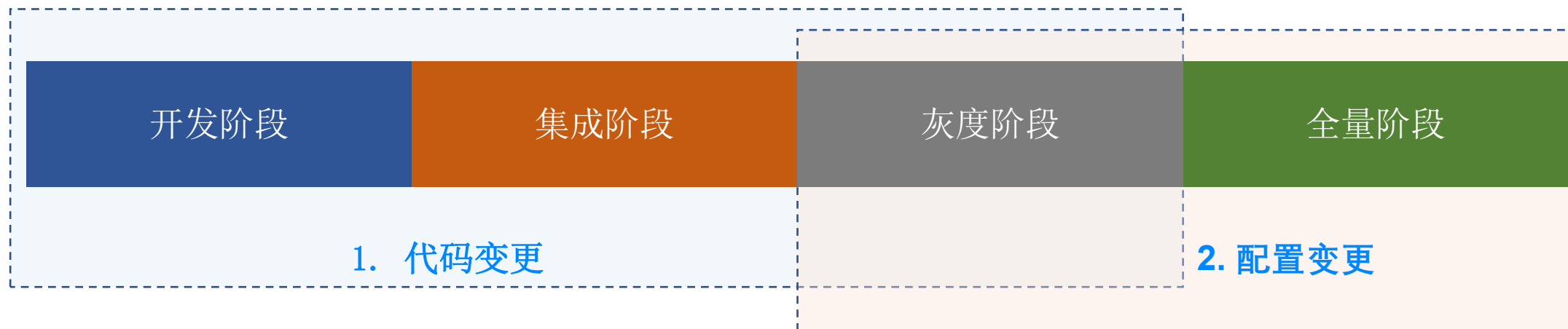
更前置
发现
劣化问题

自动化
归因
劣化问题

更全面
定位
劣化能力

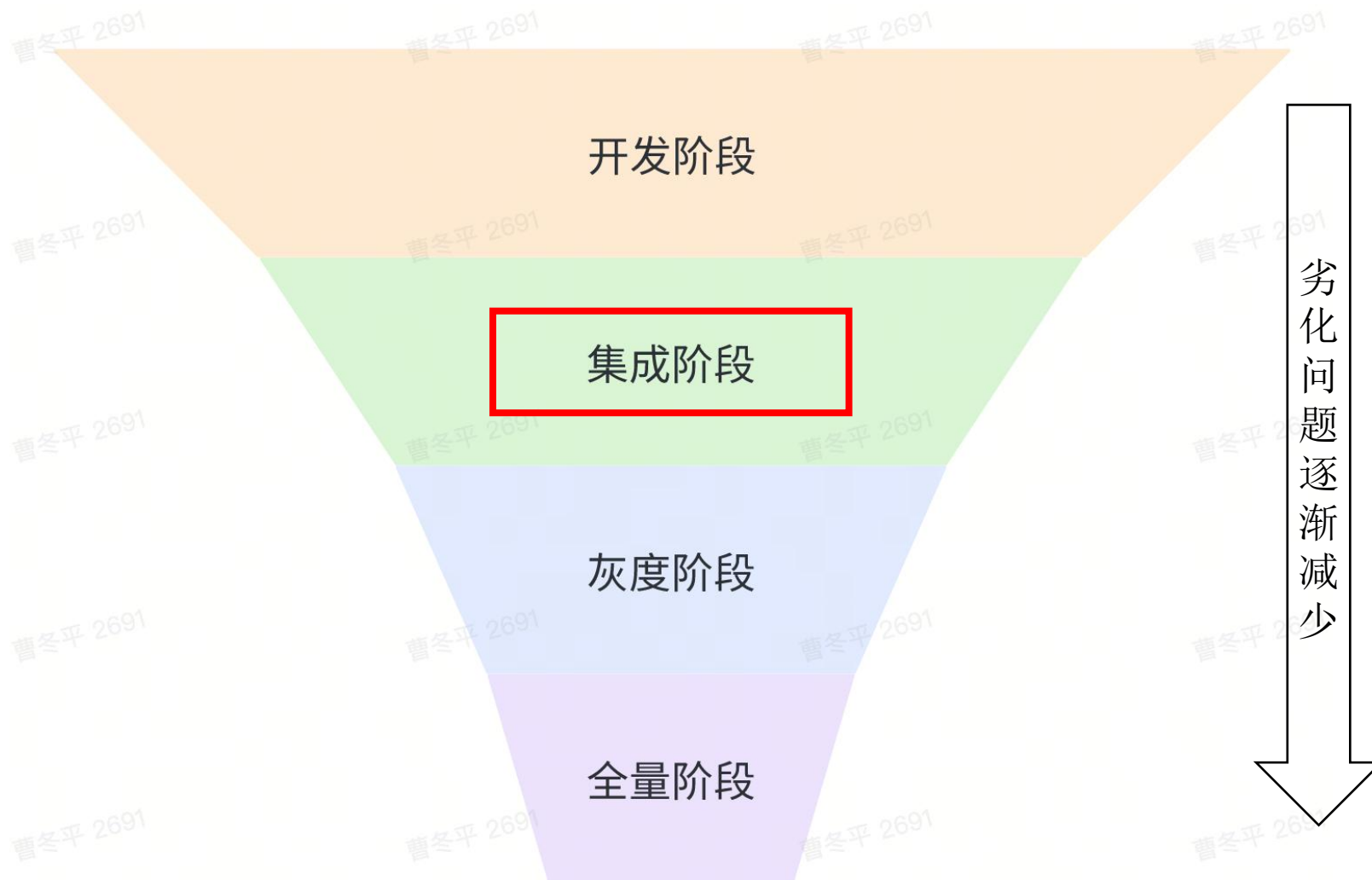
体系建设

开发链路与劣化来源



如何建设对应的防劣化能力

全链路防劣化



性能防劣化测试

集成阶段：利用自动化对比测试自动归因劣化问题

传统思路

指标自动化对比测试

指标
对比



劣化了吗?
是!

指标
对比



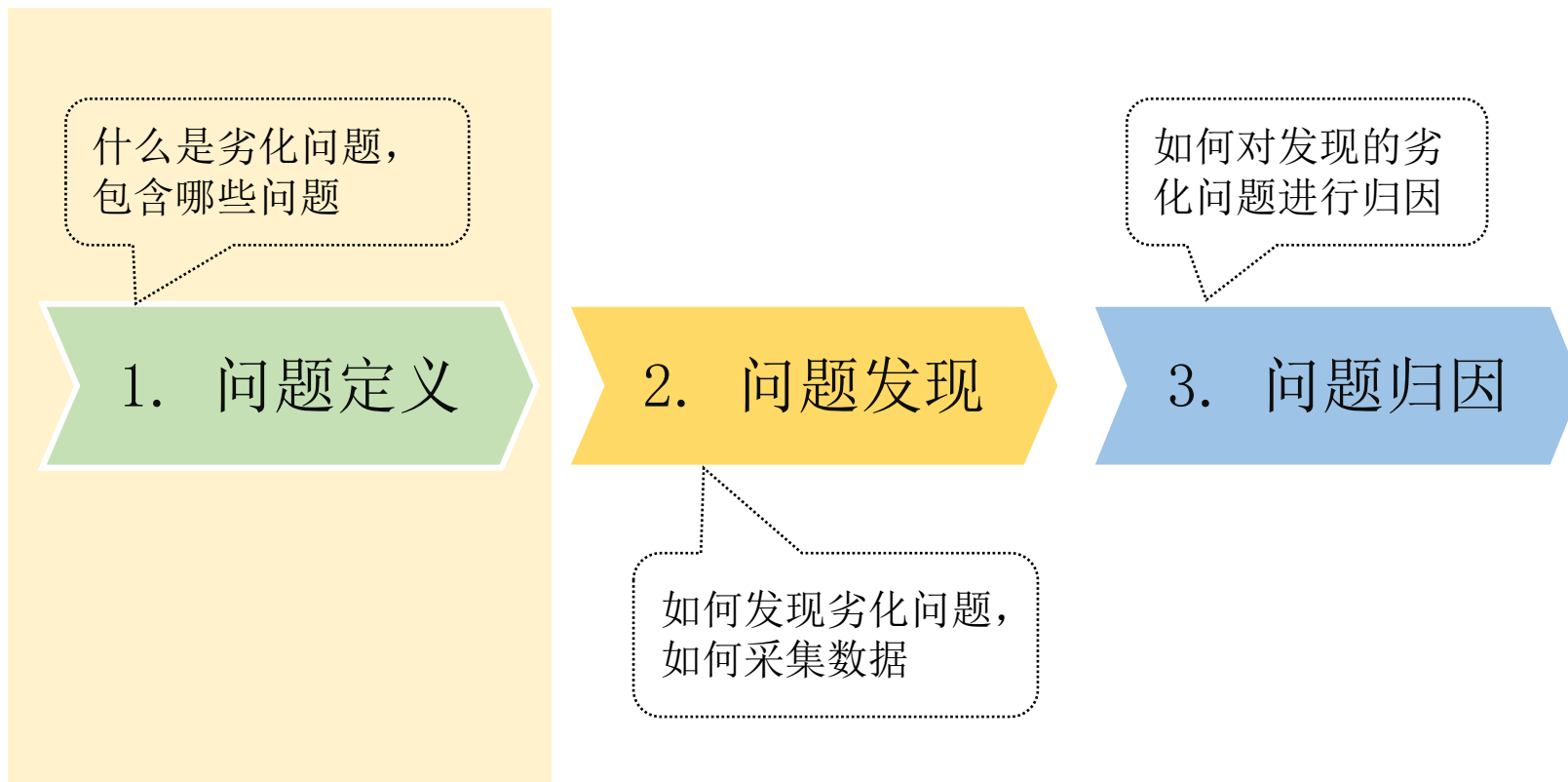
没有劣化吗?
不一定!

传统思路的问题

1. 准确率低，指标波动大容易误报；
2. 逃逸率高，指标优化大于劣化则有可能逃逸；
3. 精度低，无法发现小幅劣化；
4. 归因效率低，依赖手动归因 ；

我们的思路

我们的思路



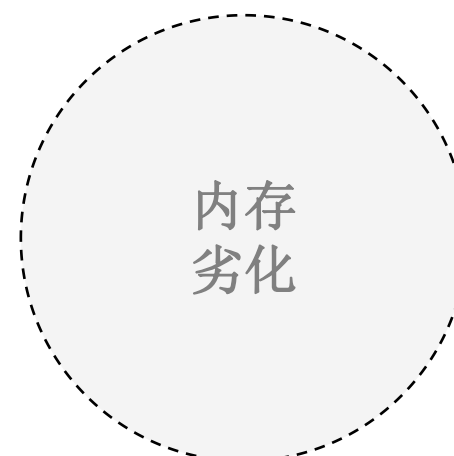
问题定义-劣化维度



主线程慢函数影响指标



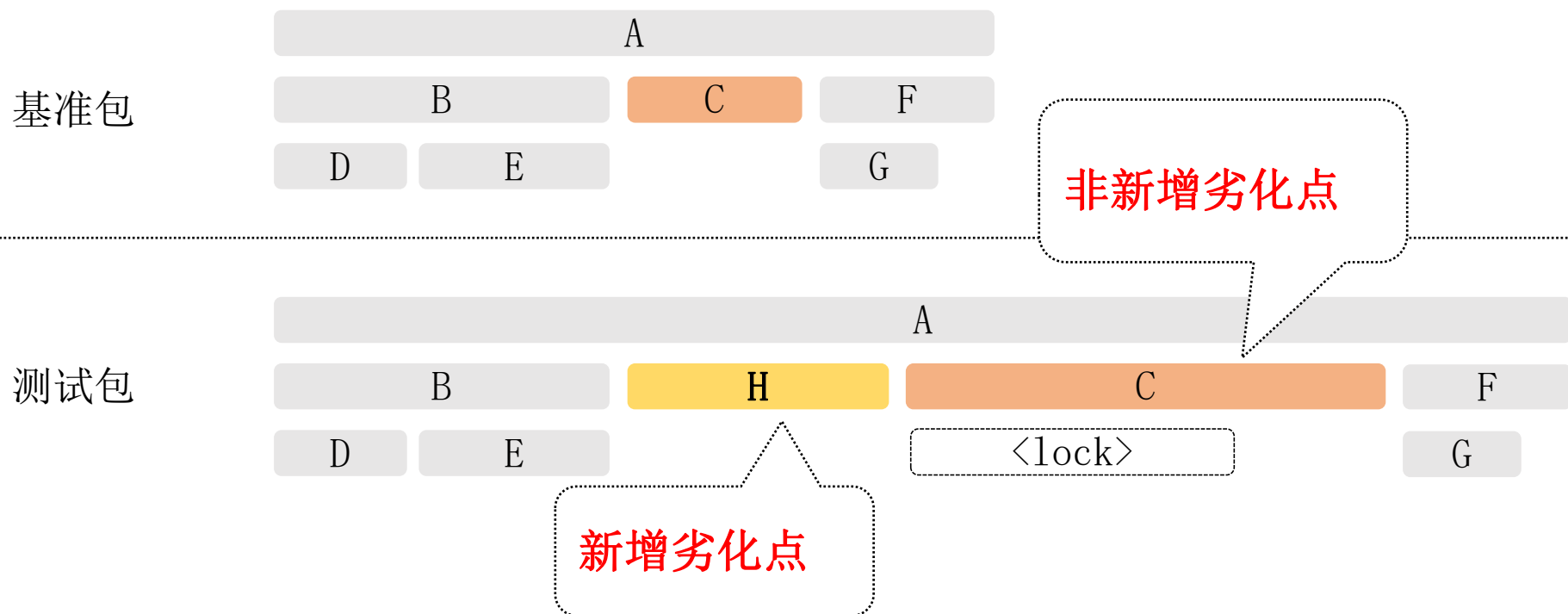
后台线程抢占资源影响指标



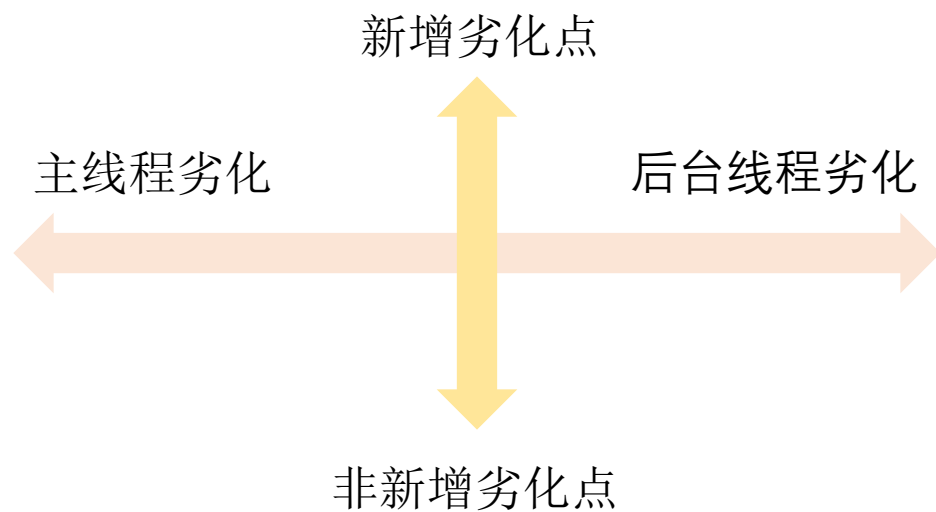
内存不足频繁 GC

未实施，暂不介绍

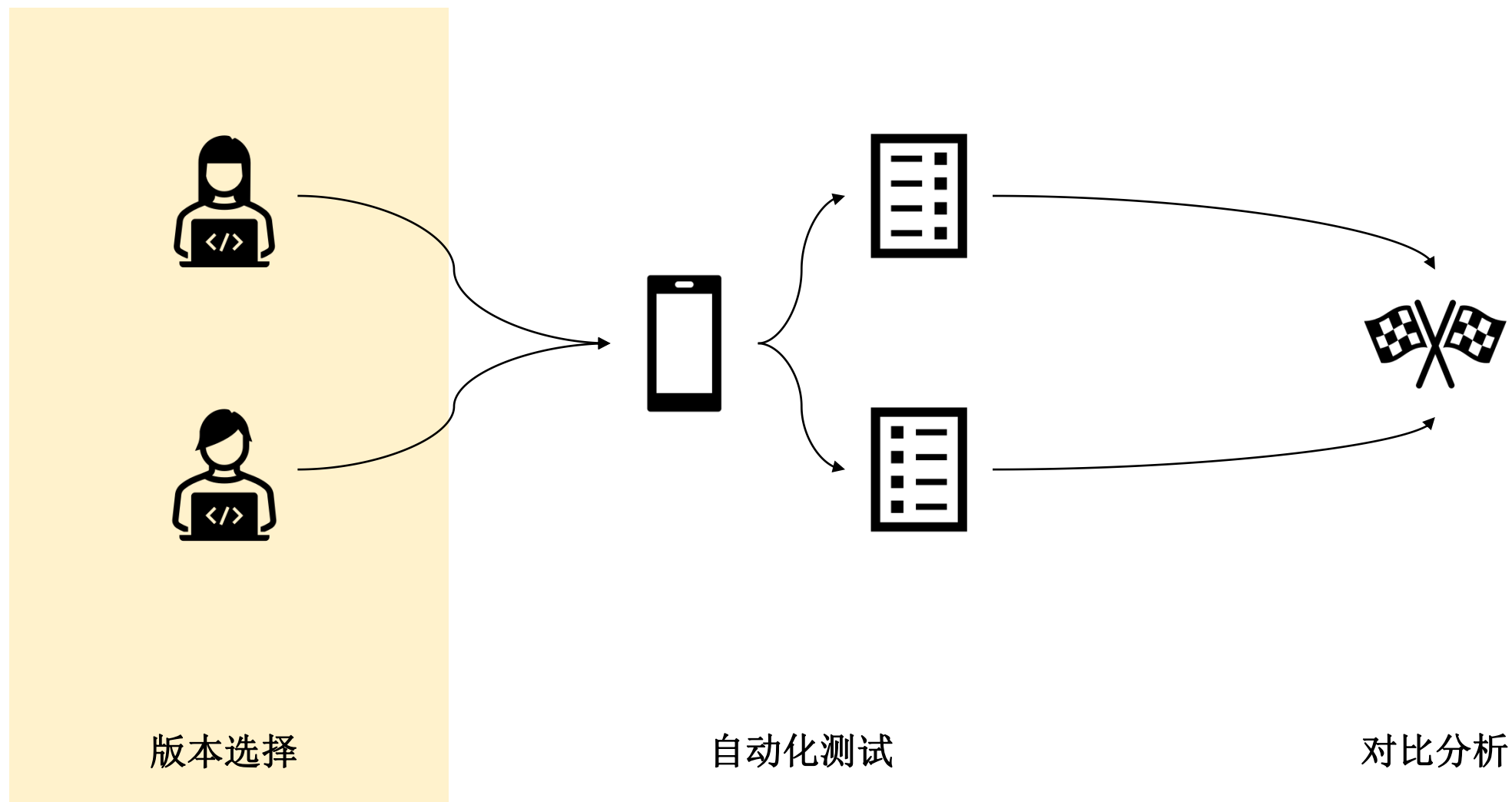
问题定义-劣化类型



围绕主线程和后台线程两个维度，以及新增与非新增劣化两个类型来展开
问题发现



问题发现方案



版本选择

1

事件触发

测试包：事件发生时的代码

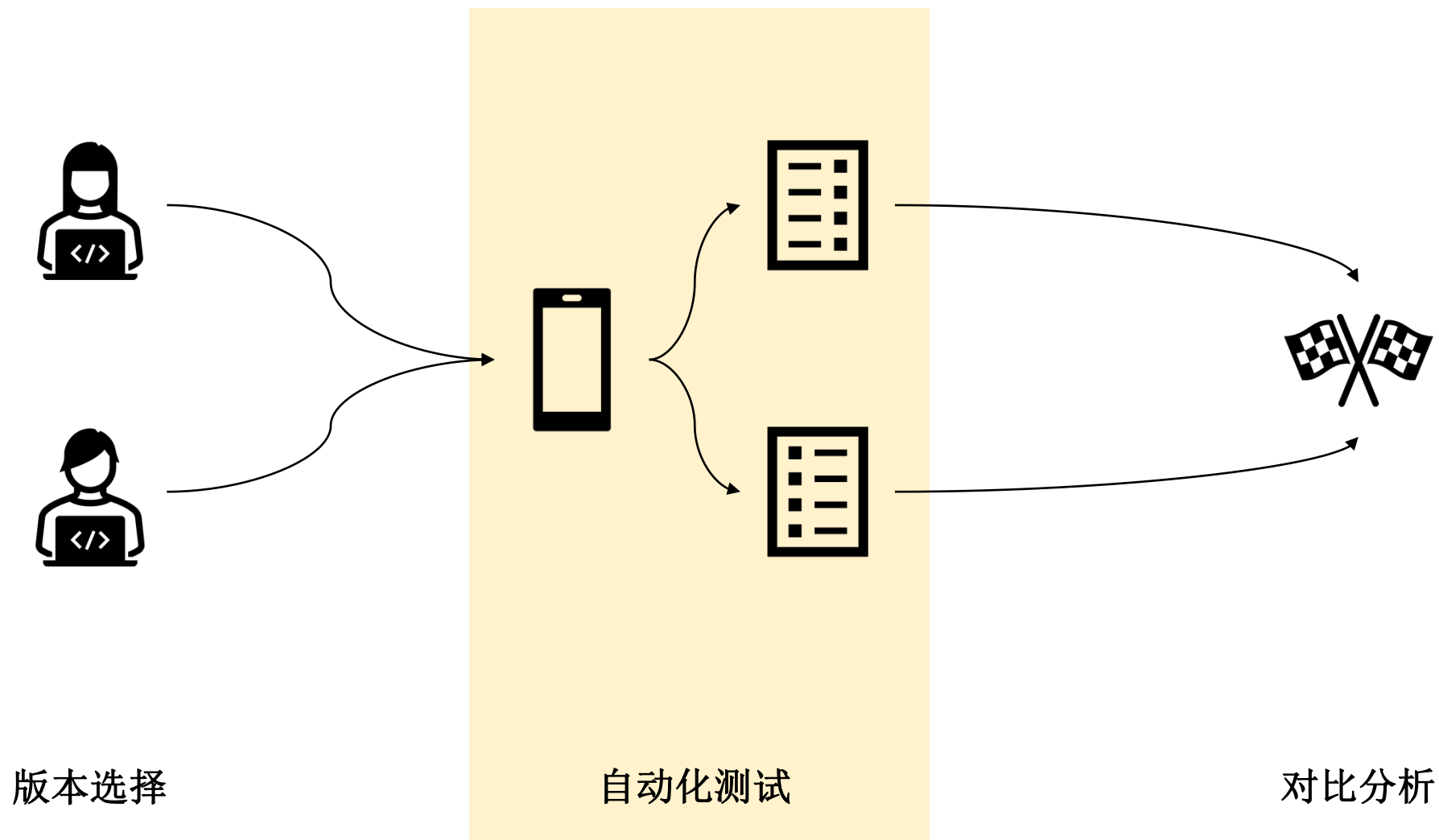
基准包：上个版本最新代码

2

手动触发

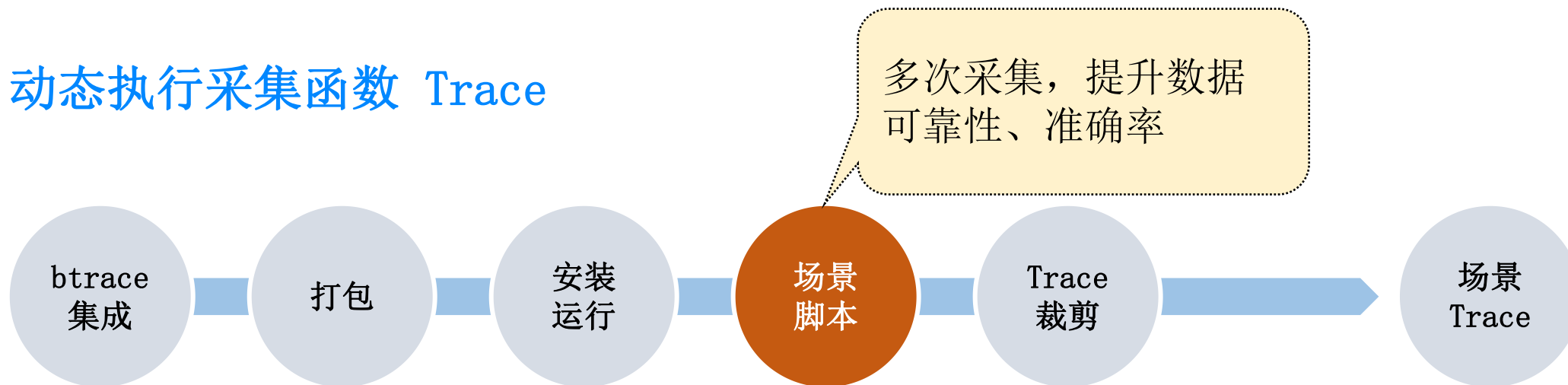
自定义基准包与测试包

自动化测试



自动化测试

动态执行采集函数 Trace



基准包产物



测试包产物

自动化测试遇到的问题

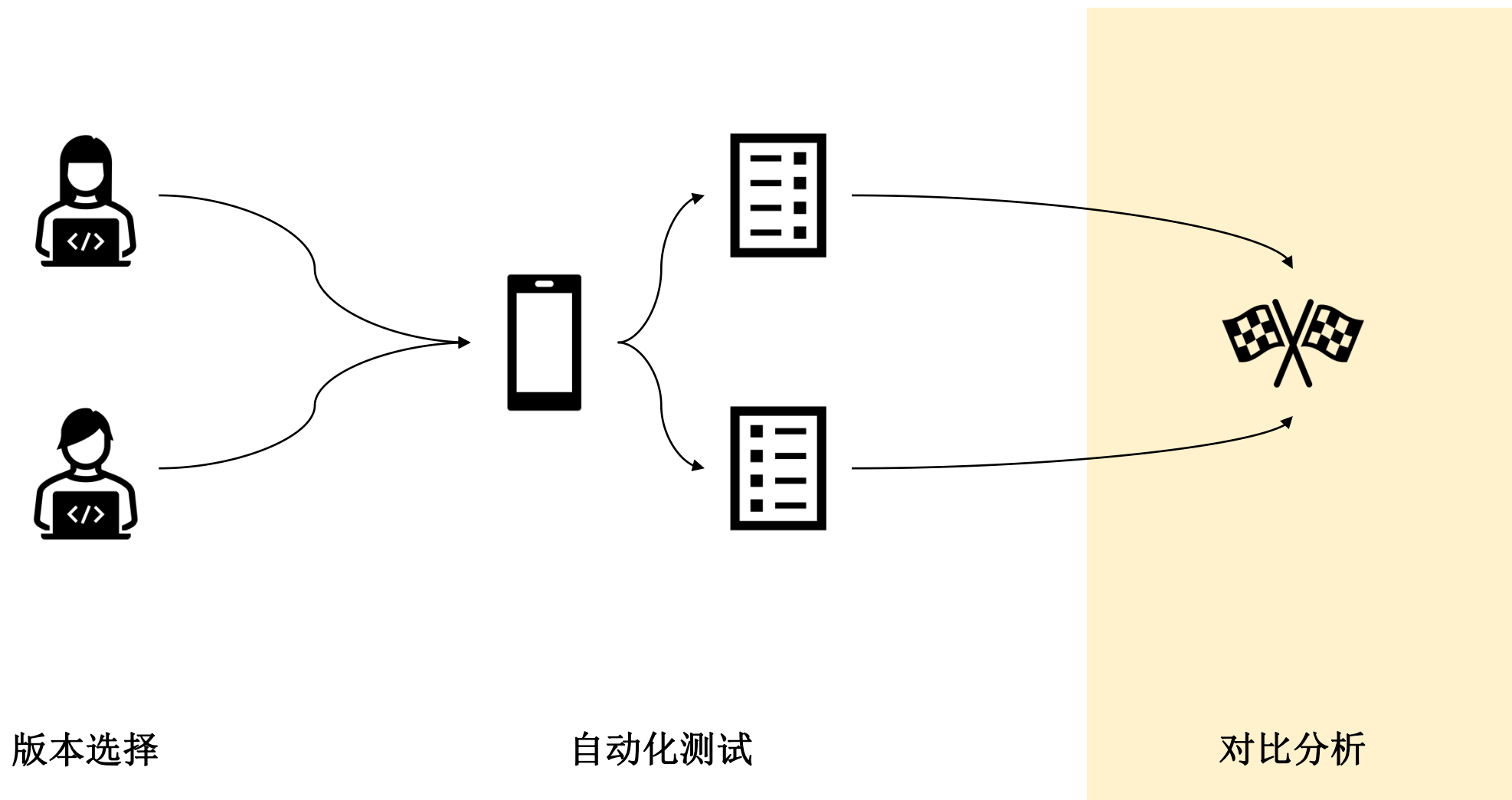
1. 耗时波动大，CPU time 可达 150ms 波动
2. 分支条件不固定，劣化误报较多，准确率 5%

测试环境优化



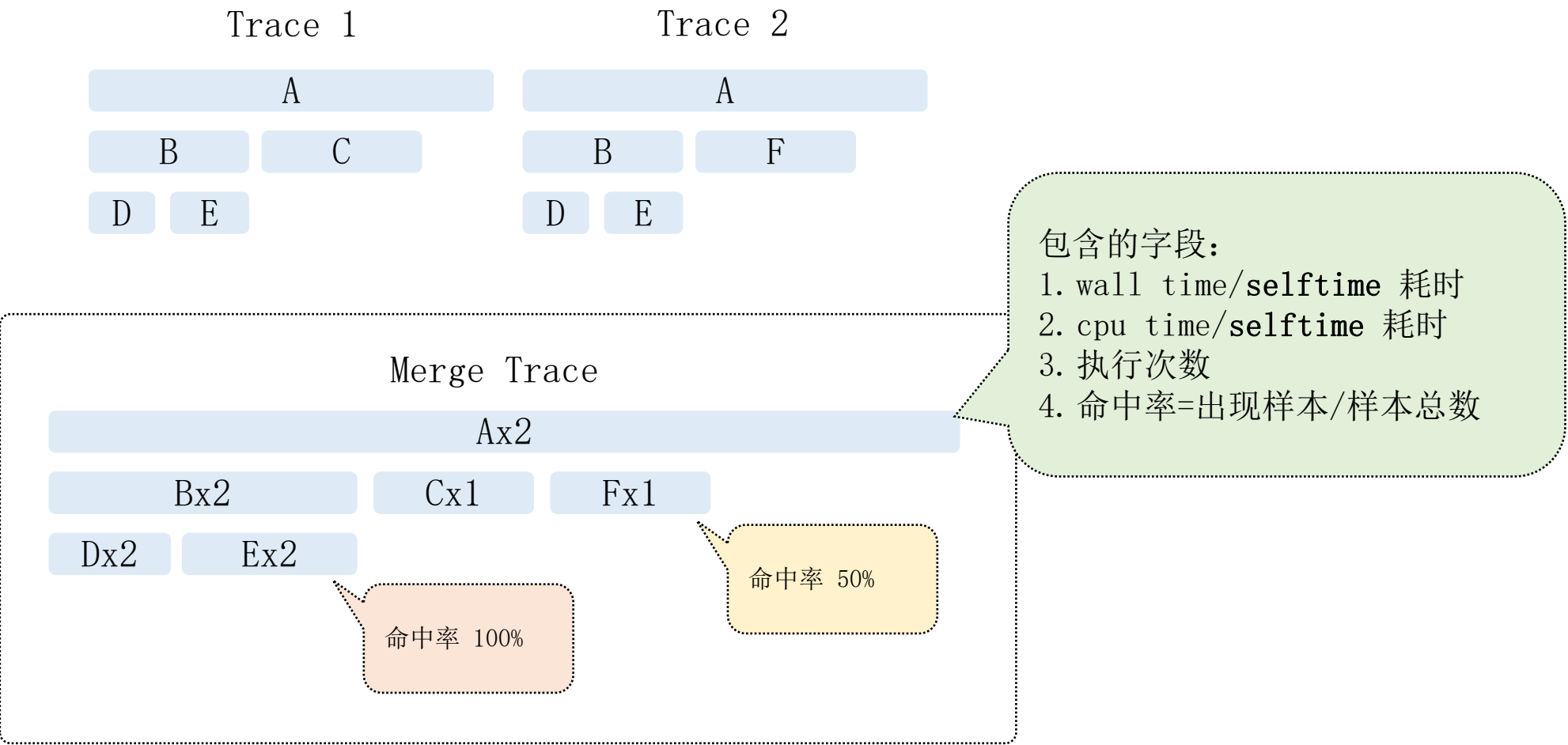
CPU time 波动低至 30ms

对比分析



Trace 聚合

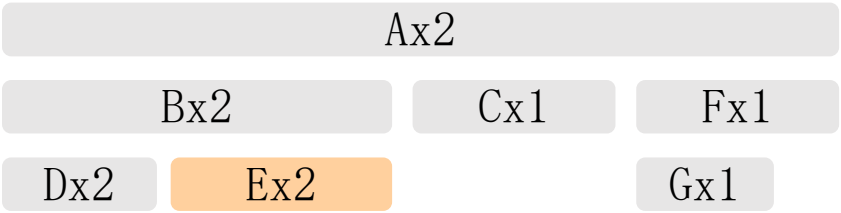
主线程聚合



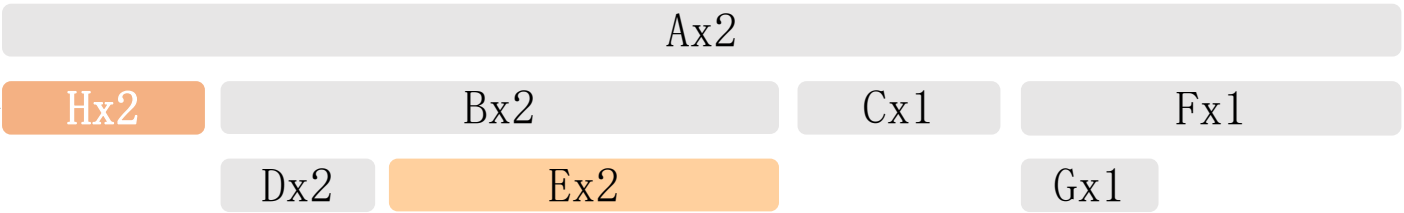
Trace Diff

主线程对比

基准包



测试包



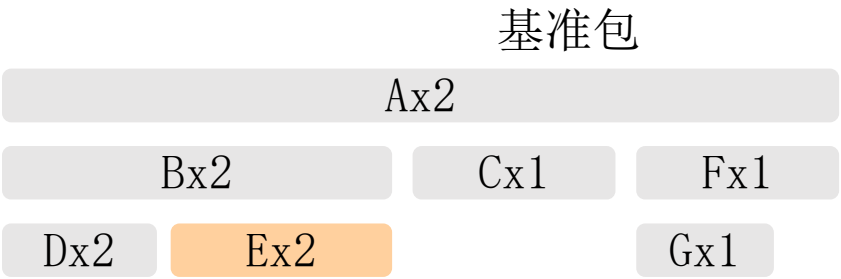
新增劣化点

- 1. 新增函数 walltime 对比
- 2. 精度可达 5ms

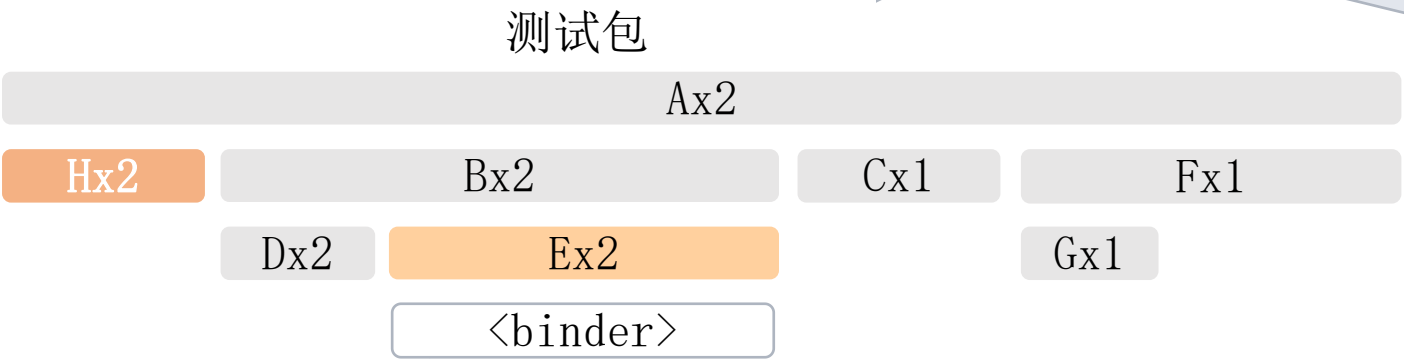
非新增劣化点

- 1. 非新增函数, wall selftime 对比
- 2. 初期精度 50ms

精细化归因



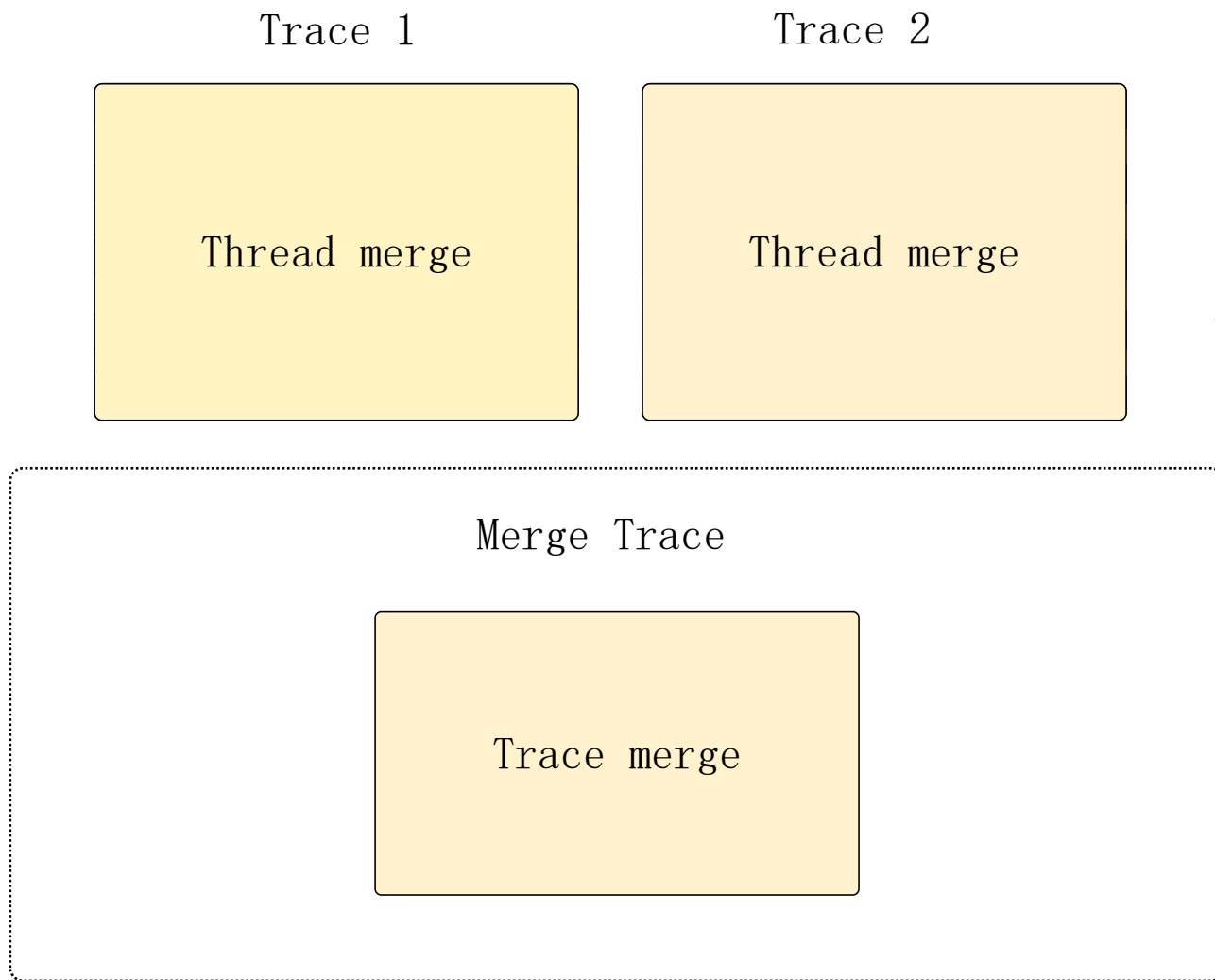
btrace2.0 新增 8 项精细化数据
敬请期待



1. 添加 lock/binder 等精细化耗时数据，归因劣化原因
2. 提升精度低至 10ms

每双月有 1-2 个后台线程劣化逃逸

后台线程聚合

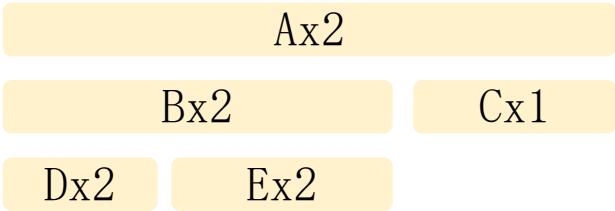


- Trace 内部线程合并
- Trace 之间合并
- 与主线程合并算一致

后台线程对比

- ✓ 更关注 CPU time
- ✓ 任务视角

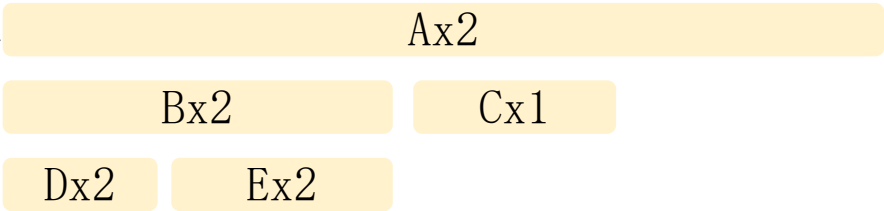
基准包



新增劣化点

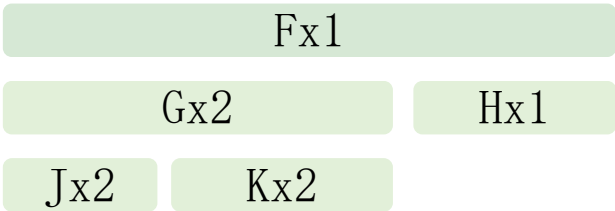
- 1. 新增任务, cputime 对比
- 2. 精度可达 50ms

测试包



非新增劣化点

- 1. 非新增任务, cpu selftime 对比
- 2. 精度可达 100ms

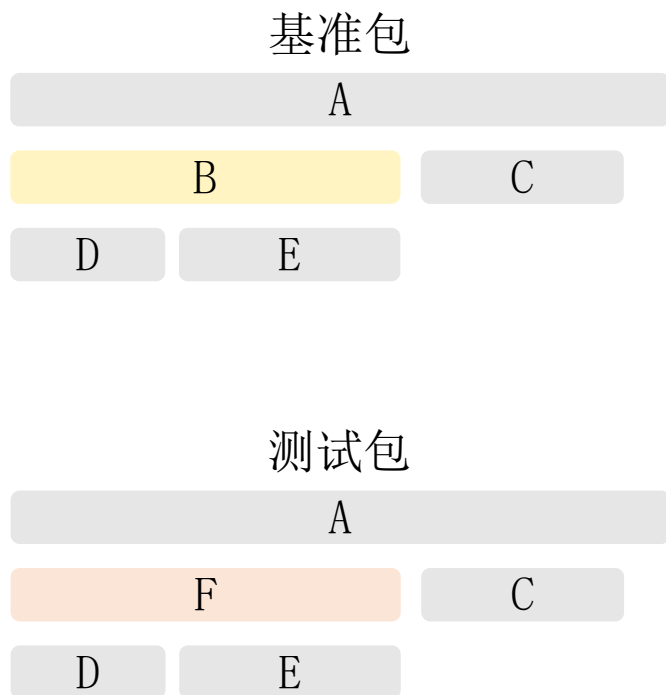


准确率 40% 徘徊，大部分误报重构导致
如何排除重构导致的误报？

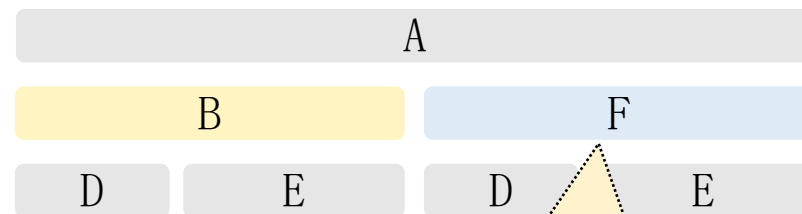
重构关键问题

1. 方法重命名
2. 方法封装
3. 方法迁移

方法重命名识别



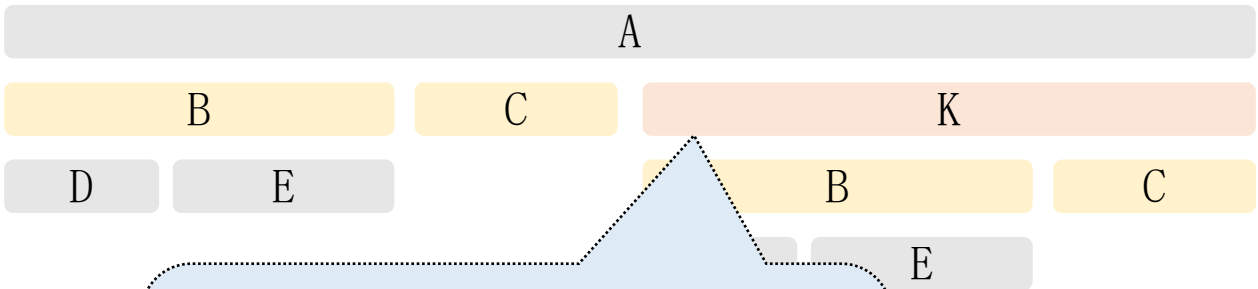
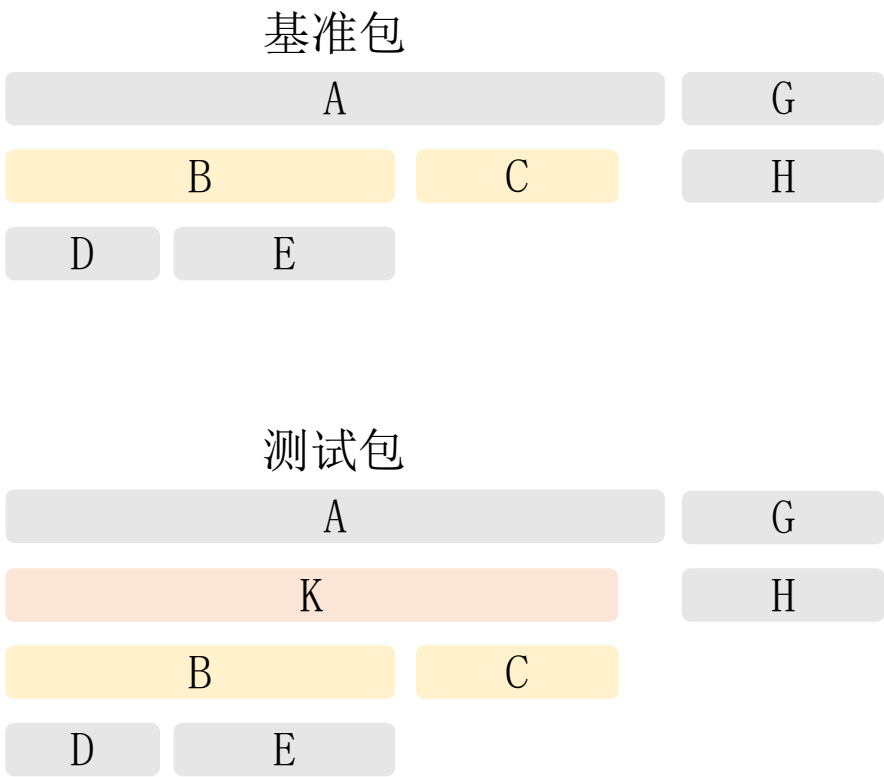
✓ 构建劣化点caller与callee火焰图



1. F 的 caller存在删除节点 B, 并且 callee 相同;
2. F 相比 B 耗时未劣化;
3. 判定为方法重命名;

方法封装识别

✓ 构建劣化点caller与callee火焰图



1. K 的 callee 与 A 原来的 callee 相同;
2. K 耗时与 callee 耗时之和基本相同;
3. 判定为方法封装;

落地效果

召回率

有效劣化数

有效劣化数 + 逃逸劣化数

80%

准确率

有效劣化数

有效劣化数 + 无效劣化数

65%

劣化
精度

低至 5ms

人效
提升

节省 8人日/月

约 50% 的劣化不能在当前版本修复
会影响到线上版本

能不能更前置的发现劣化问题？

MR 性能防劣化

开发阶段：提前拦截劣化 MR 代码合入

传统思路

针对每个提交的 MR 执行动态**指标对比**任务，拦截劣化 MR

1. 资源浪费，需要大量的硬件设备支撑；
2. 逃逸风险，指标测试波动大，劣化发现精度低；
3. 无法归因，只能发现劣化，手动归因劣化；

是否每个 MR 都需要分析？

场景无关的改动不会影响指标

```
public class SearchActivity {  
    public void onCreate() {  
        methodB();  
        methodC();  
        readFile();  
    }  
}
```

场景下代码改动不一定会影响指标

```
public class MainActivity {  
    public void onCreate() {  
        methodB();  
        methodC();  
        Log.i("hello world");  
    }  
}
```


能否通过静态分析判断 MR 是否导致指标劣化?

静态分析 MR 改动是否会影响指标

1. 获取 MR 变更方法列表



Git diff 文本对比

2. 采集指标路径方法列表



自动化测试采集

3. 判断变更代码影响指标



Class diff 指令对比



静态可达性分析

```
public void onCreate() {  
    methodB();  
    methodC();  
    methodA();  
}
```

```
...  
onCreate();  
methodB();  
methodB();  
...
```

静态分析 MR 改动是否会导致指标劣化

```
public void onCreate()  
{  
    methodB();  
    methodC();  
    methodA();  
}
```

传递性

```
private void methodA()  
{  
    methodE();  
    File.copy();  
}
```

IO 耗时特征

1. 方法耗时具有特征性

耗时类型	指令名	说明
锁	monitorenter、monitorexit、ACC_SYNCHRONIZED	
循环	goto	
特殊方法调用	InvokeVirtual、InvokeSpecial、InvokeStatic、InvokeInterface、InvokeDynamic	调用方法存在 native、IO 相关、线程相关等方法调用

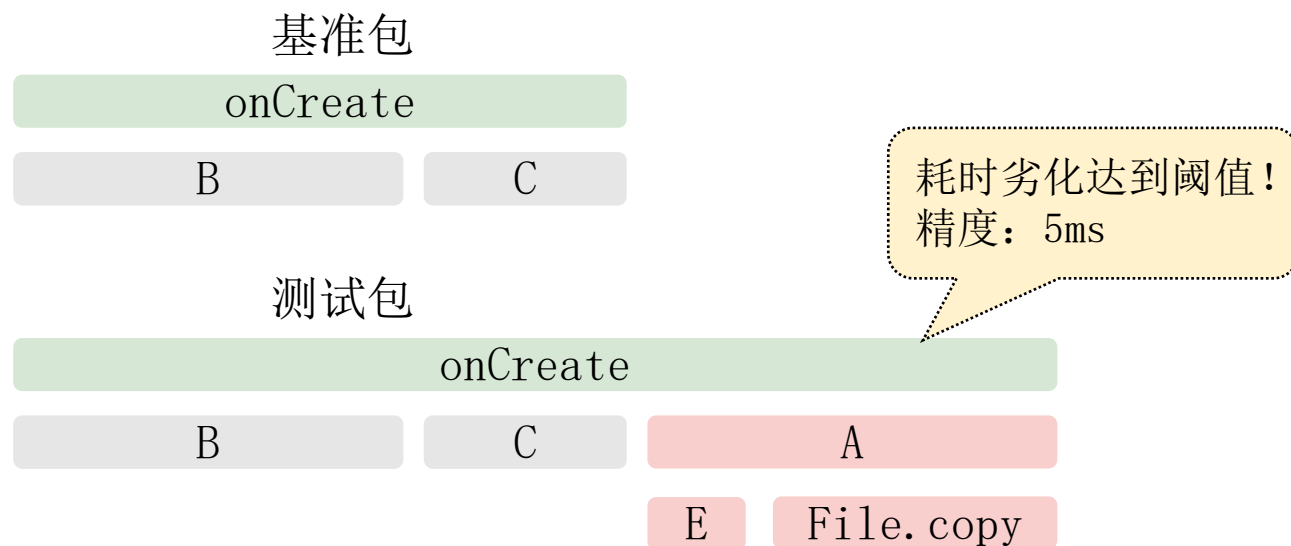
2. 方法耗时具有传递性

约 14% MR 会影响指标，并且命中耗时规则

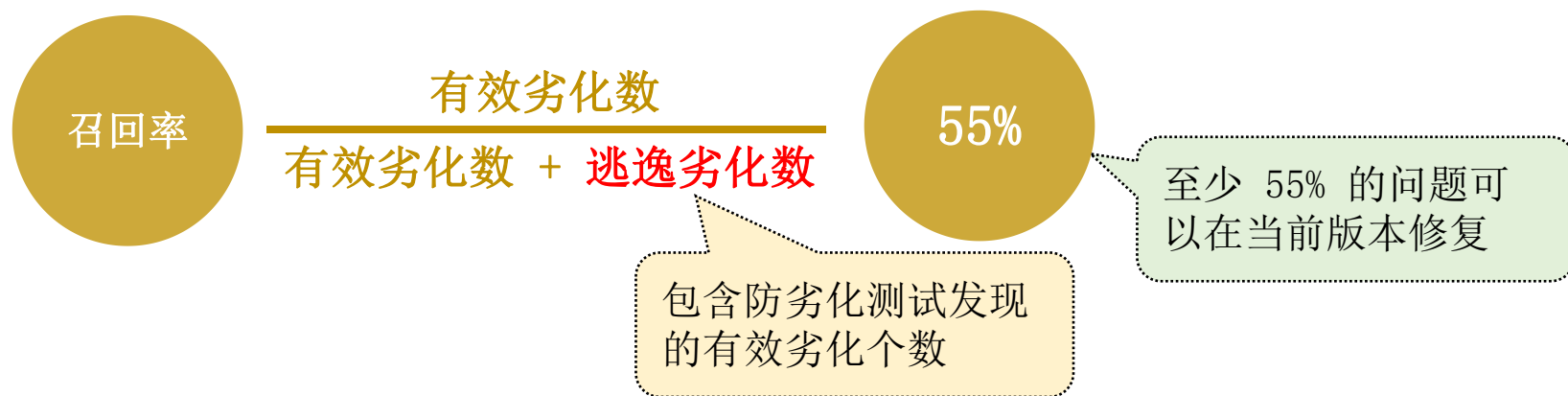
难以消费

结合性能防劣化测试

动态测试分析实际劣化



落地效果



1. 线上配置变更，线下无法发现
2. 线下防劣化本身能力不完善可能导致逃逸
3. 每月线上 2 个劣化报警，平均劣化排查 4 人日

能否自动定位线上指标劣化的原因？

线上指标劣化归因

发布阶段：利用线上大数据发现指标劣化并完成归因

传统思路

通过灰度、线上指标对比发现劣化，线下手动归因；

1. 逃逸风险，当优化大于劣化时，可能逃逸；
2. 无法归因，只能发现劣化，手动归因劣化；

我们的思路

指标报警

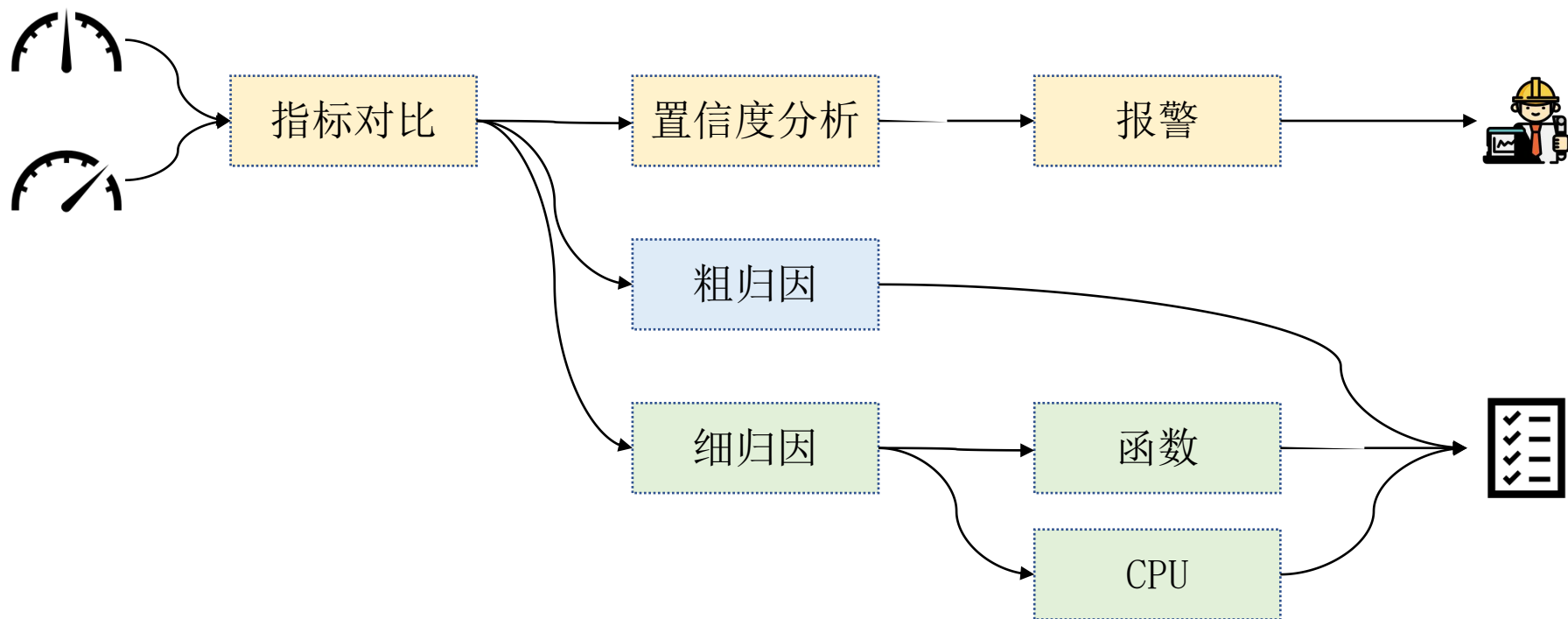
指标报警：劣化
提前发现与粗归
因手段

自动归因

自动归因：指标
是否劣化，都要
自动归因。

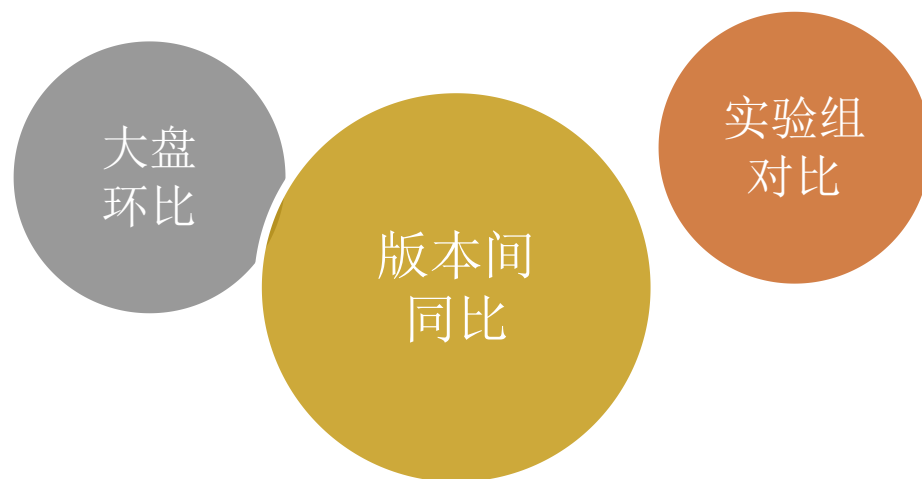
整体方案

工作流程图

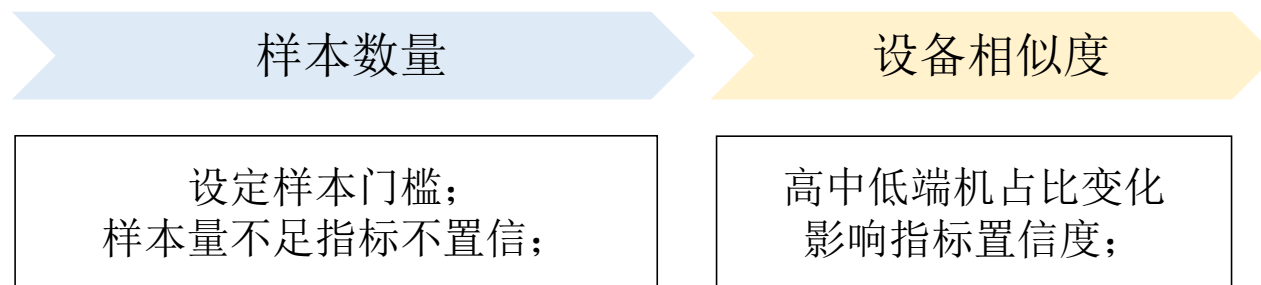


劣化报警

1. 指标对比



2. 报警置信度



满足指标劣化与置信度分析后，即可拉群报警

粗归因

缩小劣化排查范围

多维度数据

系统、品牌、机型（高中低）等
节假日日活提升，低端机占比提高

子阶段数据

将指标拆解为若干个子阶段

AB 实验

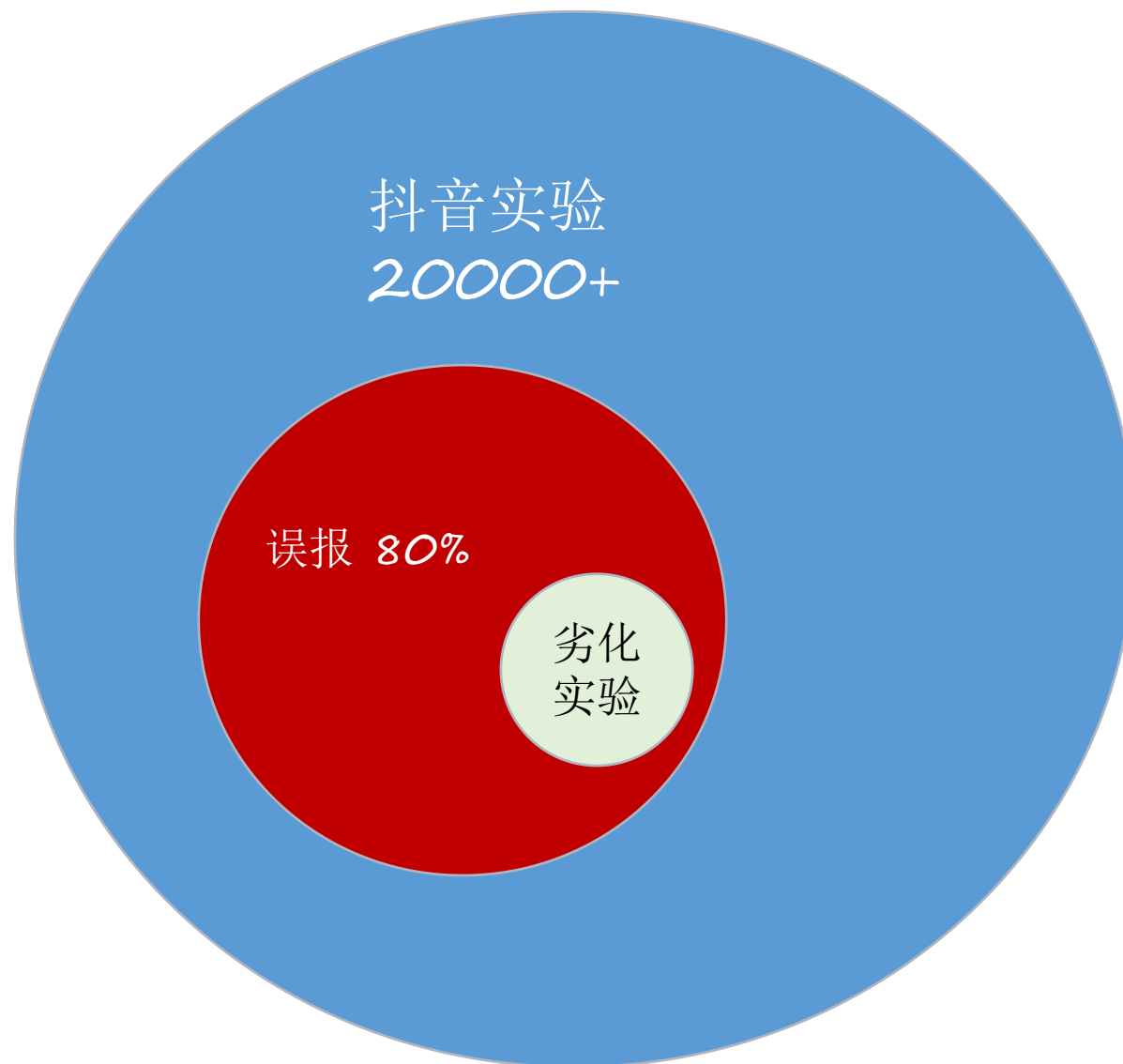
发现因 AB 放量导致的劣化问题

自定义
归因数据

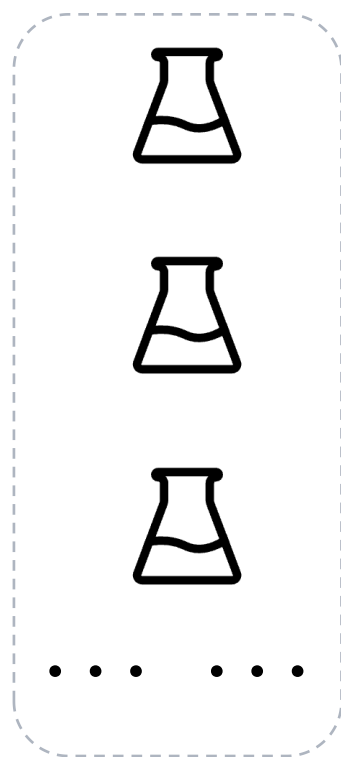
比如缓存占比

重点介绍 AB 实验粗归因

粗归因-AB 实验



粗归因-AB 实验



实验圈存



实验初筛



置信度分析



报警

粗归因-AB 实验

召回率

有效劣化实验

有效劣化实验 + 逃逸劣化实验

90%

准确率

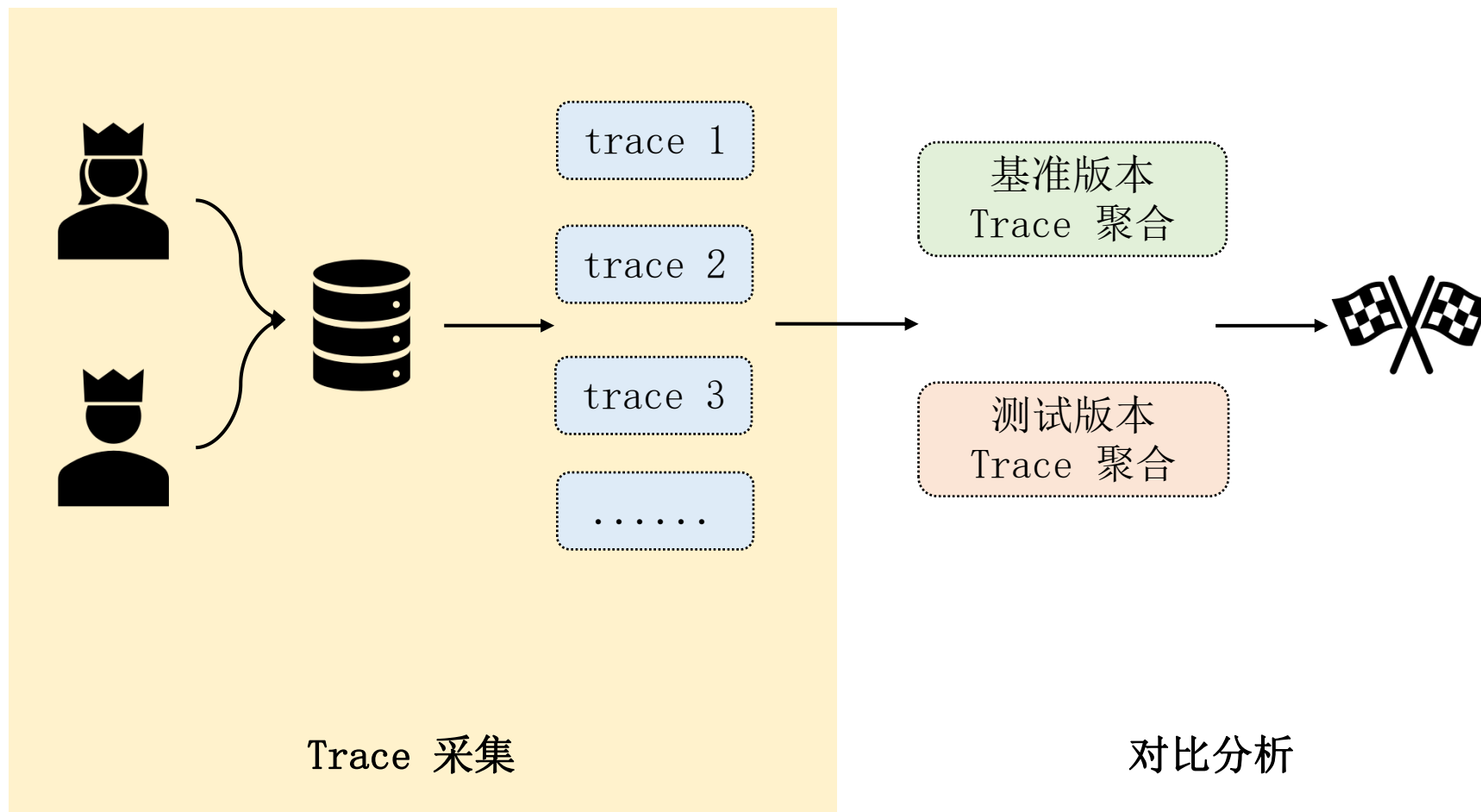
有效劣化实验

有效劣化实验 + 无效劣化实验

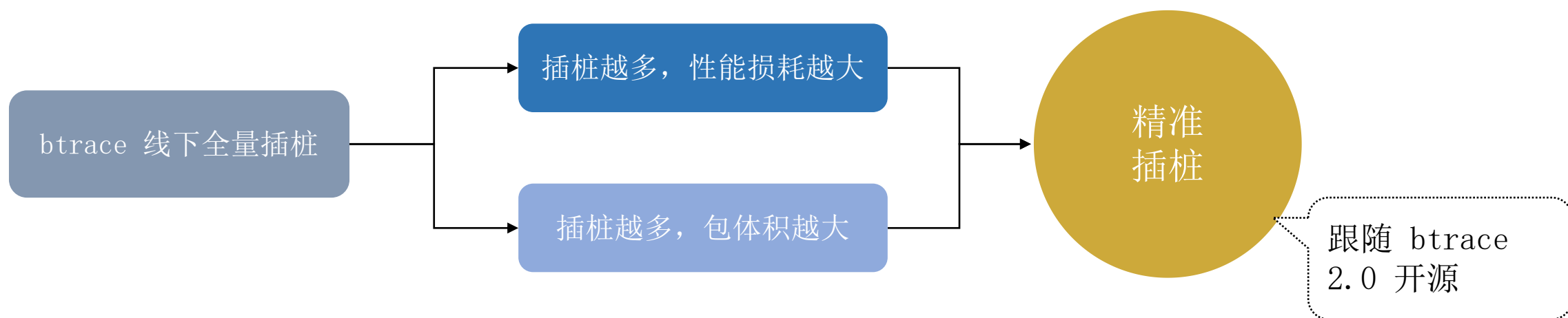
80%

能否更细粒度的发现劣化问题？

函数归因



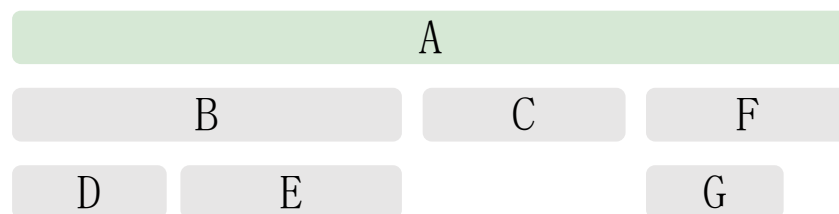
Trace 采集



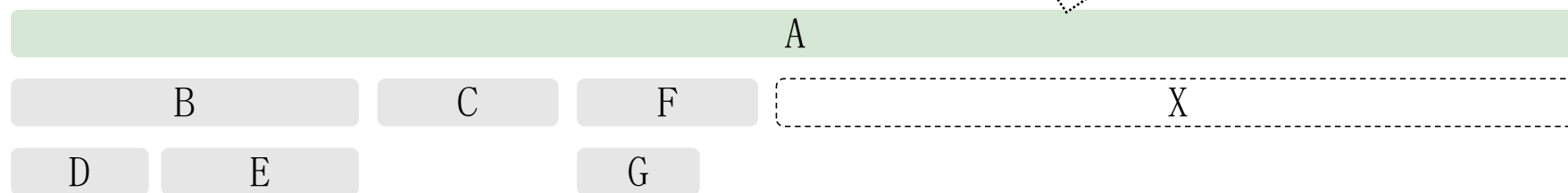
只对可疑耗时代码与上下文插桩

精准插桩问题

问题 1: 插桩数量变少, 劣化难以归因



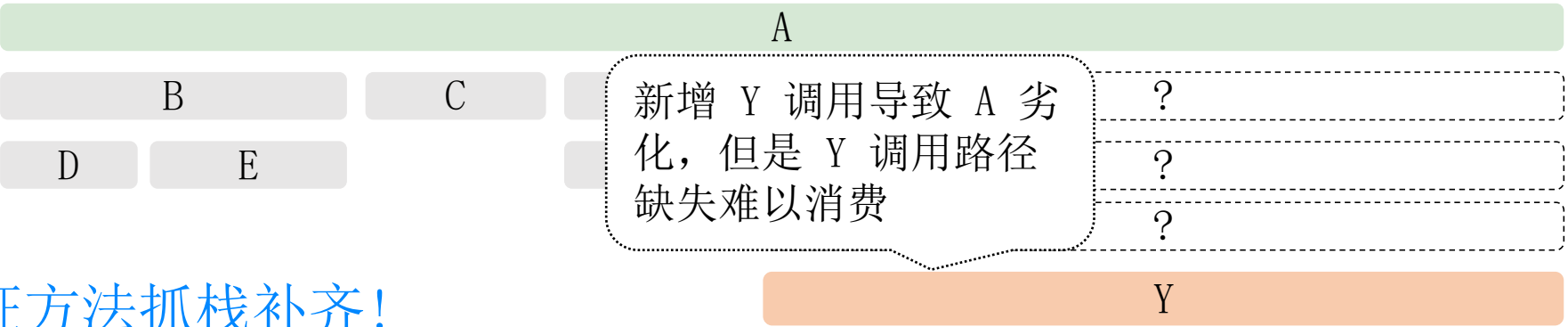
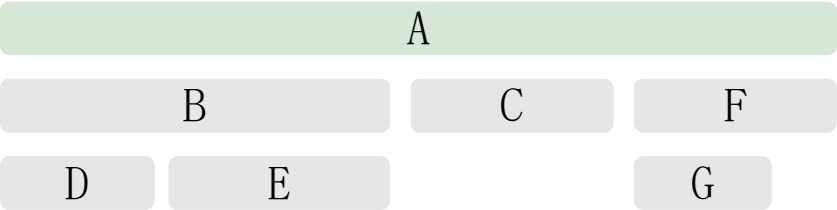
新增 X 方法调用导致 A 方法劣化;
如果 X 方法未插桩, A 劣化的原因难以归因;



版本变更代码精准插桩!

精准插桩问题

问题 2：插桩数量变少，劣化难以消费



关键耗时特征方法抓栈补齐！

线上如何对比分析？

对比分析



聚合

TraceDiff

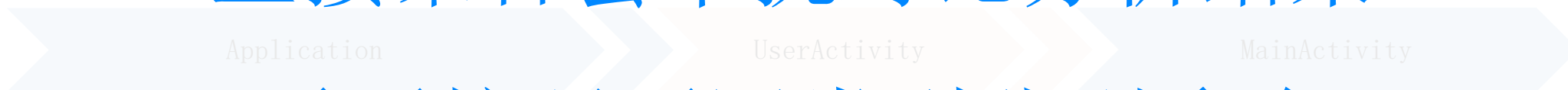
聚合算法与线下一致
但是有些特定问题需要处理

非正常启动路径

正常启动路径:



外部拉起直接聚合会干扰对比分析结果



需要按照不同类型分别聚合

Push 路径:



TraceDiff 思路与线下一致

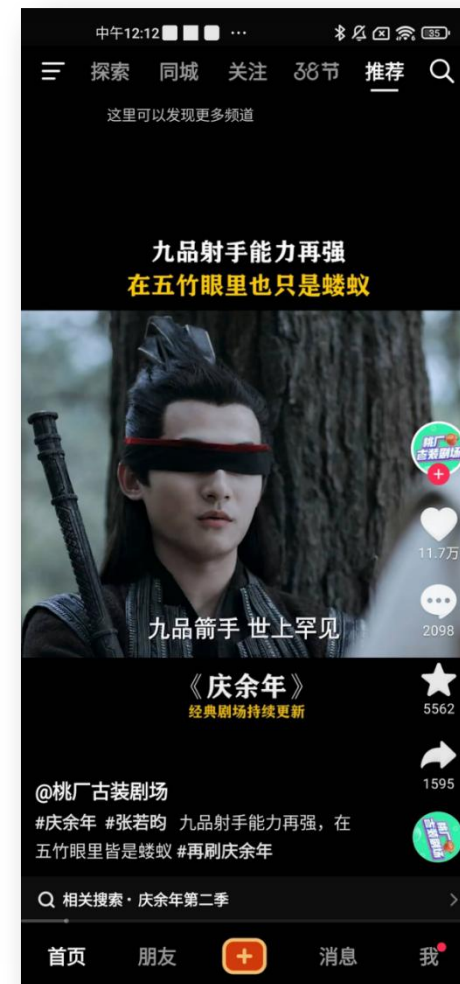
落地案例

函数归因

1. MallHomeTab:onTabCreated 平均执行次数 0.41→0.90
2. HomeBottomSkinManager:loadSkin 平均执行次数 0 → 0.59

归因结论

妇女节电商活动 & 两会底 tab 换肤



落地效果

召回率
提升

80% → 90+%

归因
成功率

完成归因报警个数
总有效劣化报警个数

80%

归因
效率

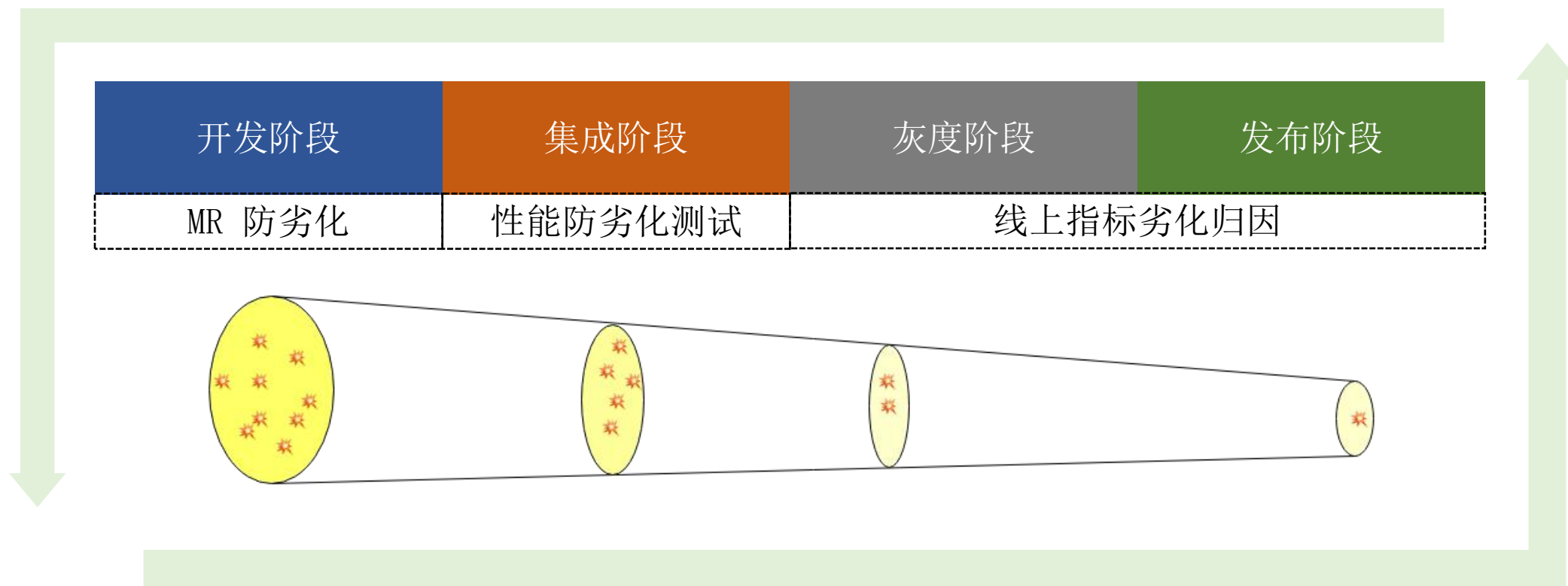
节省 4 人日/月

全链路防劣化回顾

02_全链路正循环

漏斗模型：全链路防劣化能力，经过不同阶段的防劣化筛选，劣化问题越来越少

正循环： 右边逃逸反哺左边归因能力，劣化点不断左移



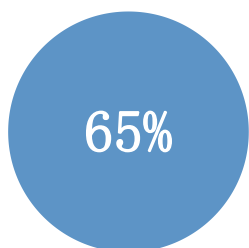
落地效果

整体效果

全链路防劣化

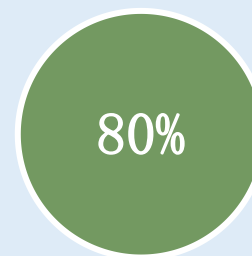


召回率



准确率

线上指标劣化归因

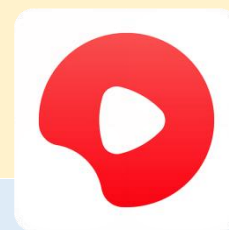
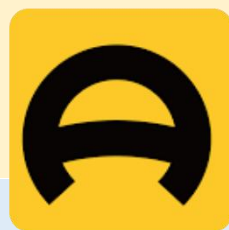


成功率

应用与业务

已接入应用

抖音、抖级、Resso、剪映、懂车帝、西瓜视频等 12 个应用



已接入业务

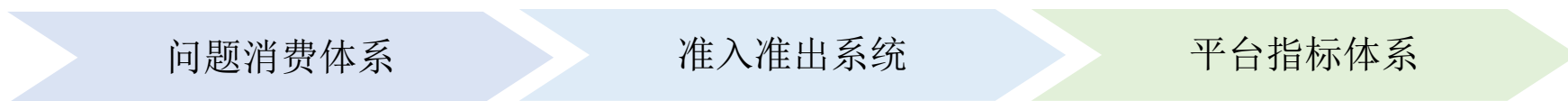
基线、电商、综搜、生活服务、私域、直播等 10 条业务线

展望未来

展望未来



1. 持续优化完善全链路防劣化能力，提升准召率；
2. 覆盖内存、功耗、流量、存储等全场景防劣化；



1. 健全防劣化配套的问题消费、指标统计、劣化准出体系；



THANKS.