DevFest

北京 / Beijing

2026

驯服 AI：Compose 代码的质量防线

霍丙乾 bennyhuo
Google 开发者专家

Google Developer Groups

SPEAKER · 2026 DEVFEST BEIJING

讲者简介

霍丙乾 bennyhuo

Google 开发者专家 / Kotlin 布道师 / Gemini 爱好者

技术身份与社区贡献

- Google Developer Expert (Android)
- 腾讯开源项目 KuiklyBase-Kotlin 和 ovCompose 跨平台框架核心贡献者

技术专著

- 《深入理解 Kotlin 协程》作者
- 《深入实践 Kotlin 元编程》作者
- 长期关注编译器技术、Kotlin 与 Compose 工程质量



Google Developer Groups

SPEAKER · TALKS HISTORY

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	GDG Android 11 Meetup	Kotlin 协程那些事儿
2020.10	GDG DevFest	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	GDG DevFest	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？
2023.11	GDG DevFest	使用 Kotlin 元编程技术提升开发效率
2023.12	机械工业出版社	我们对 Java 有哪些常见的误区？
2024.2	GDG 社区说	TensorFlow Lite 在小猿口算中的应用与实践
2024.6	GDG IO Extend / KotlinConf	Compose 是如何实现跨平台的？
2024.12	GDG DevFest	Kotlin Native：跨平台开发的新选择
2025.11	GDG DevFest	那些令人意想不到的Compose 的陷阱
2026.6	社区技术分享	腾讯视频 Compose 动态化实践

Google Developer Groups

PART 0 · INDUSTRY DATA (2026)

代码生成提速，验证与审查全面过载

Google Android Bench 2.0 ↗

~28%

真实工程任务通过率

- **单点任务 (Bench 1.0)**：~91%
- **长链路真实工程**：最高 ~28%
- **瓶颈**：跨组件架构与状态迁移

Faros AI 2026 ↗

+441%

PR Review 耗时激增

- **PR 平均体积**：+51%
- **单 PR 缺陷数**：+54%
- **跳过审查合入**：+31%

Sonar Developer Survey ↗

53%

表面正确，运行不可靠

- **AI 生成代码占比**：42%
- **隐性技术债加剧**：88%
- **首要痛点**：编译通过但行为异常

42% 代码由 AI 生成，人工 Review 已过载，Compose 的质量防线靠什么守住？

隐患藏在哪？（1/2）

✓ 编译器 + AndroidX Lint + compose-rules 全部放行

```
@Composable
fun PromoBannerCard(
  title: String,
  progress: State<Float>,
  modifier: Modifier = Modifier,
) {
  // 为做边界裁剪与“即将抢光”判断，先解包出当前 进度
  val cur = progress.value.coerceIn(0f, 1f)
  val endingSoon = cur >= 0.9f
  Column(modifier.padding(16.dp)) {
    PromoHeader(title = title, highlight = endingSoon)
    CouponSummaryRow()
    // 顺手复用 cur 传入 () -> Float，延迟读取彻底失效
    ProgressTrack(progress = { cur })
  }
}
```

为什么开发者与 AI 都会这样写？

- **业务动机合理**：先裁剪进度，再判断 endingSoon
- **顺手复用变量**：直接把 { cur } 传给子组件
- **表面写法合规**：含 modifier，签名 () -> Float

运行时隐患：提前解包 导致延迟读取失效

60fps 动画 进度每帧更新 → 顶层读 .value 父作用域失效 → **卡片每帧重组 180 次 / 3s**

- **阈值未收敛**：endingSoon 仅变 1 次却每帧重算
- **延迟读取失效**：{ cur } 仅捕获普通值，绘制优化落空

静态 Lint 只认 () -> Float 类型签名，看不穿“先在组合阶段读取状态、再包进闭包”的数据流向。

隐患藏在哪？（2/2）

✓ 编译器、官方 Lint 与参数签名检查全部放行

```
@Composable
fun PromoCouponRow(
  items: List<PromoItem>, modifier: Modifier = Modifier,
) {
  // 为展示序号角标选用 itemsIndexed，顺手拿 i 当 key
  LazyRow(modifier.fillMaxWidth()) {
    itemsIndexed(items, key = { i, _ -> i }) { i, it ->
      // 滑出视口后需跨回收保留的状态用 rememberSaveable
      var reminded by rememberSaveable {
        mutableStateOf(false)
      }
      CouponChip("#${i + 1} ${it.name}", reminded) {
        reminded = !reminded
      }
    }
  }
}
```

删除中间项 B 后，子组合状态错位图解

① 删除前：用户在 key = 1（键盘 B）开启「★ 已设提醒」

Slot [key = 0]	Slot [key = 1]	Slot [key = 2]
#1 耳机 A	#2 键盘 B ✕	#3 鼠标 C
○ 未设提醒	★ 已设提醒	○ 未设提醒

▼ 删除「键盘 B」→ 鼠标 C 左移到 i = 1 位置

② 删除后：key = 1 原地复用，C 错误继承 B 的提醒状态

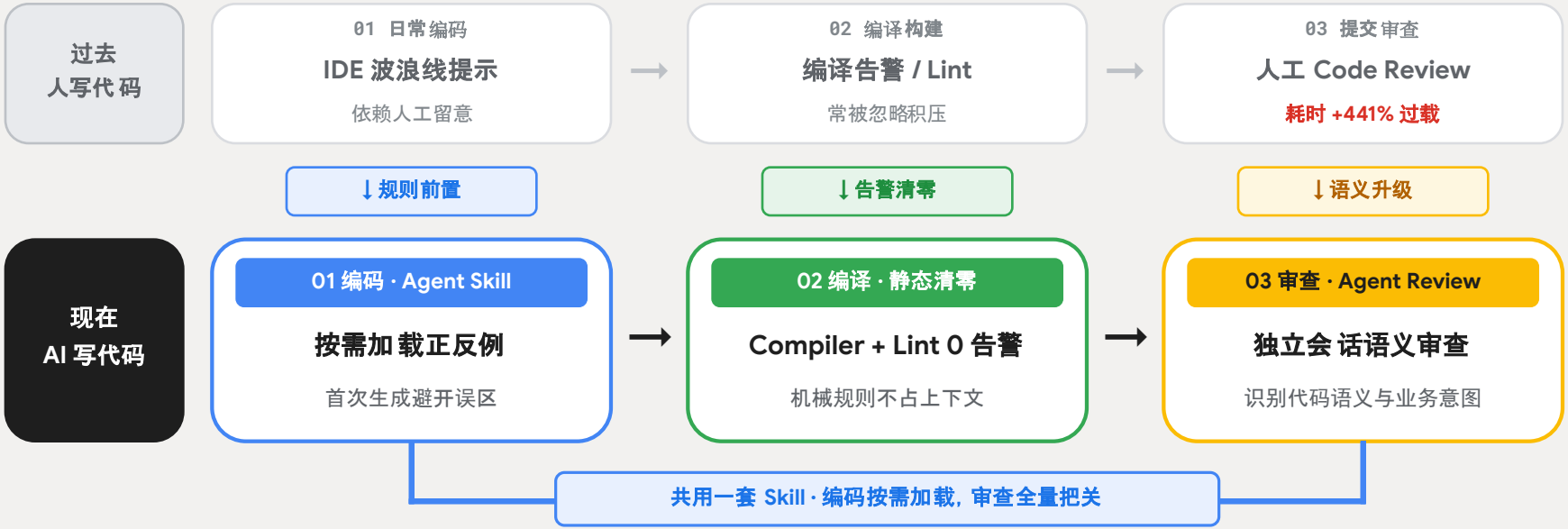
Slot [key = 0]	key = 1 原地复用	key = 2 已释放
#1 耳机 A	#2 鼠标 C	C 原有的状态随末尾项释放
○ 未设提醒	★ 已设提醒（原 B）	

✓ 正解：key = { _, it -> it.id }，状态随数据项迁移

传统静态工具难以判断哪个字段代表稳定业务主键；Agent 能结合上下文识别用列表下标充当 key。

PART 0 · ARCHITECTURE SHIFT

质量防线升级：从 Compiler Lint 到 Agent Lint



静态工具守住语法下限，Agent Skill 守住语义上限。

Google Developer Groups

PART 0 · RESPONSIBILITY BOUNDARY

职责边界：机械语法归 Linter，代码语义归 Agent

Compiler-based Lint

基于局部 AST 与类型签名 · 零上下文开销

AndroidX Lint	状态漏写 remember 装箱 · 组合阶段启动协程
compose-rules	Modifier 透传 / 复用 禁止下传 ViewModel
ComposeGuard	Skippable 检查 稳定性退化对比

✓ 机械规则 100% 静态清零

静态全绿放行

静态漏检
语义隐患

Agent-based Lint

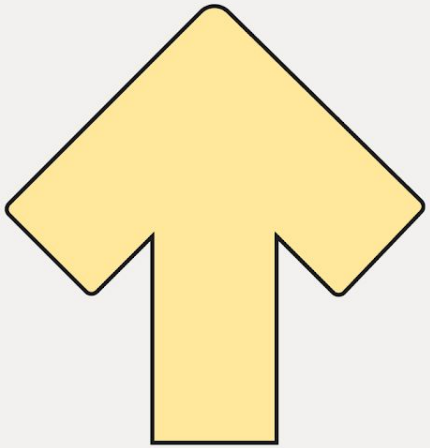
理解数据频率、状态流向与业务身份 · Skill 驱动

读取阶段与频率	提前读取高频状态 延迟读取失效，引发父级重组
收敛比与状态流	判断阈值状态是否收敛 评估 derivedStateOf 收益
业务身份与键值	区分列表下标与业务 ID 避免删除或重排后状态错位

✓ 编码阶段规范引导 · 审查阶段语义把关

静态 Linter 能拦的绝不塞进 Skill；上下文窗口只留给静态工具测不出的语义规则。

01



静态检查的盲区

Google
Developer
Groups

延迟状态读取失效：读取位置决定重组范围

✕ Incorrect · 组合阶段提前读取再封装闭包

```
@Composable
fun CompositionTrack(progress: State<Float>) {
  // 组合阶段提前读取，父 RestartGroup 已注册快照依赖
  val value = progress.value
  ProgressTrack(progress = { value })
}

@Composable
fun ProgressTrack(progress: () -> Float) {
  Canvas(Modifier.fillMaxWidth().height(26.dp)) {
    val start = Offset(0f, size.height / 2f)
    drawLine(color = barColor, start = start,
      end = Offset(size.width * progress(), start.y),
      strokeWidth = 20f, cap = StrokeCap.Round)
  }
}
```

组合阶段已读取 progress.value，父作用域每帧失效

✓ Correct · 将状态读取推迟到绘制阶段

```
@Composable
fun DeferredTrack(progress: State<Float>) {
  // 传入读取闭包，组合阶段不读状态
  ProgressTrack(progress = { progress.value })
}

@Composable
fun ProgressTrack(progress: () -> Float) {
  Canvas(Modifier.fillMaxWidth().height(26.dp)) {
    val start = Offset(0f, size.height / 2f)
    drawLine(color = barColor, start = start,
      end = Offset(size.width * progress(), start.y),
      strokeWidth = 20f, cap = StrokeCap.Round)
  }
}
```

Canvas 绘制阶段才调用 progress()，父作用域零重组

函数类型参数本身不带来优化，真正调用 progress() 读取状态的阶段才决定快照依赖位置。

SECTION 01 · DEMO #1 (DRAW READ)

实测对比：签名同为 () -> Float，父级重组截然不同

视觉效果一致，底层执行路径完全不同

两条轨道均平滑走完 3 秒动画，肉眼看不出任何差别

每帧 +1
父级重组次数

✕ 提前读取 · CompositionTrack

组合 · 每帧重跑 → 绘制 · Canvas

父作用域每帧失效，持续触发无效重组

0 新增
保持初始 1 次

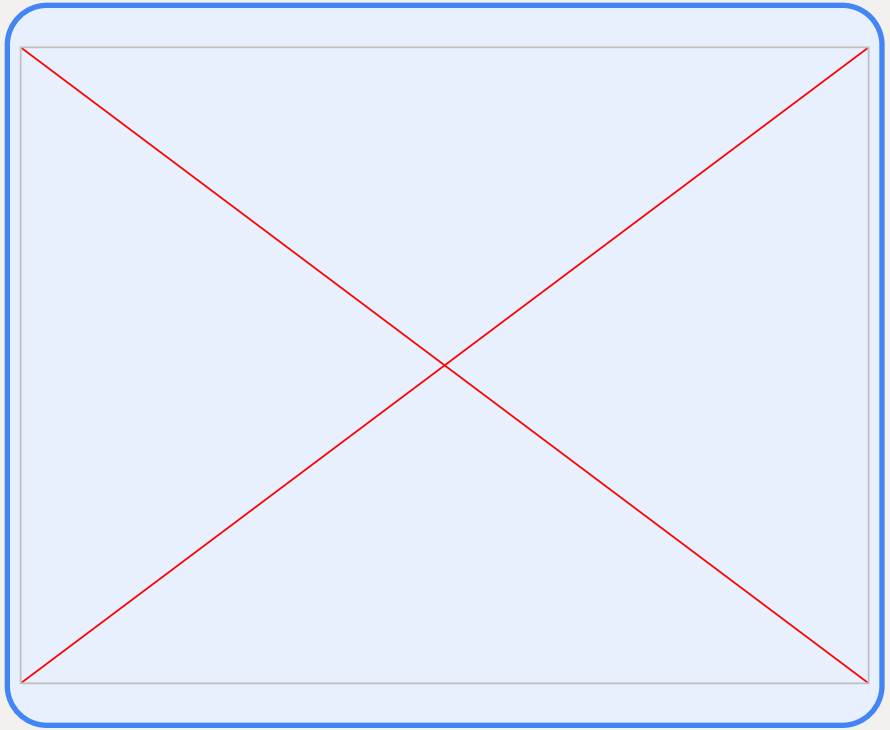
✓ 延迟读取 · DeferredTrack

组合 · 跳过 → 仅绘制 · Canvas

跳过父作用域重组，直接进入绘制阶段

同类高频状态场景（应避免在父组件组合阶段读取）

音视频进度　手势拖拽偏移　AI 流式文本



Google Developer Groups

derivedStateOf：频率收敛边界与外部参数 key

✕ Incorrect · 同频冗余包装 / 漏传外部参数 key

```
// 1. 同频 1:1 拼接，输入每变结果必变，徒增快照开销
@Composable
fun NameBar(first: State<String>, last: State<String>) {
  val full by remember {
    derivedStateOf { "${first.value} ${last.value}" }
  }
  Text(full)
}

// 2. threshold 为普通参数，变更后仍沿用旧阈值
@Composable
fun Tracker(offset: State<Int>, threshold: Int) {
  val crossed by remember {
    derivedStateOf { offset.value >= threshold }
  }
}
```

同频映射只会增加开销；普通参数漏传 key 导致阈值过期

✓ Correct · 高频转低频收敛 + 外部参数写入 key

```
// 1. 同频 1:1 派生直接计算，无需套用 derivedStateOf
@Composable
fun NameBar(first: State<String>, last: State<String>) {
  Text("${first.value} ${last.value}")
}

// 2. 多对一降频才用 derivedStateOf，且普通参数写进 key
@Composable
fun Tracker(offset: State<Int>, threshold: Int) {
  val crossed by remember(threshold) {
    derivedStateOf { offset.value >= threshold }
  }
}
```

同频状态直接读取，多对一收敛才用；普通参数写入 key

高频输入收敛为低频结果才用 derivedStateOf，且块内捕获的非 State 参数必须写入 remember(key)。

状态所有权：消除反向同步与元素原地修改

✕ Incorrect · Effect 反向同步 / 原地改列表元素属性

```
// 1. 冗余状态 + LaunchedEffect 反向同步，多闪一帧旧值
@Composable
fun GreetingBad(first: String, last: String) {
  var full by remember { mutableStateOf("") }
  LaunchedEffect(first, last) { full = "$first $last" }
  Text(full)
}

// 2. SnapshotStateList 不感知元素内部普通 var 修改
class Todo(val id: Int, var done: Boolean = false)
val todos = remember { mutableStateListOf(Todo(1)) }
Button(onClick = {
  todos[0].done = true // 列表结构未变，界面不刷新
}) { Text("完成") }
```

反向同步多出一帧旧值闪烁；直接改元素 var 无法触发重组

✓ Correct · 单向直接计算 / 不可变元素拷贝替换

```
// 1. 派生值按单向数据流直接计算，无需冗余状态与协程
@Composable
fun GreetingGood(first: String, last: String) {
  val full = "$first $last"
  Text(full)
}

// 2. 不可变 data class + 索引替换，精准触发快照失效
data class Todo(val id: Int, val done: Boolean = false)
val todos = remember { mutableStateListOf(Todo(1)) }
Button(onClick = {
  todos[0] = todos[0].copy(done = true)
}) { Text("完成") }
```

单向直接计算零延迟；不可变元素替换触发 Snapshot 更新

禁止建冗余状态再用 Effect 反向同步；SnapshotStateList 须搭配不可变元素替换更新。

组合身份：绑定稳定业务 ID，禁用下标充当 key

✕ Incorrect · 普通循环漏传 key / 列表下标充当 key

```
// 1. 普通 for 循环未加 key，删除 B 后 C 继承 B 的状态
for (item in items) {
  RememberedNoteRow(item)
}

// 2. 用列表下标充当 key，仅满足签名检查，
//    语义等同按位置复用，删除或重排后状态错位
LazyColumn {
  itemsIndexed(items, key = { index, _ -> index }) {
    _, item -> SaveableNoteRow(item)
  }
}
```

调用位置不是业务身份；用 index 充当 key 等同按位置复用

✓ Correct · 动态列表统一绑定稳定唯一的业务 ID

```
// 1. 普通循环同样使用 key(item.id) 锚定组合身份
for (item in items) {
  key(item.id) {
    RememberedNoteRow(item)
  }
}

// 2. Lazy 列表提供同级唯一业务 key + 异构 contentType
LazyColumn {
  items(
    items = items,
    key = { "${it.type}-${it.id}" },
    contentType = { it.type },
  ) { item -> SaveableNoteRow(item) }
}
```

局部状态随数据项迁移，多类型条目按 contentType 复用

普通 for 循环同样需要 key()；Lazy 列表的 key 必须同级唯一且跨重组稳定，禁用 index 与随机 UUID。

SECTION 01 · DEMO #2 (KEY REMOVE)

实测对比：删除中间项后，位置身份与 业务 key 的差异

运行时状态归属对比

1. 初始状 态(每行 remember 局部 UI 状态)

仅 B 勾选加星、展开抽屉、★ ×3(重要优惠)
A、C 均保持未勾选、折叠、×1

2. 删除 B 后 · 普通 for 循环(无 key)

C 前移到位置 1，按位置复用原来的 Slot
C 错误继承了 B 的勾选、展开、★ ×3 与「重要优惠」

3. 删除 B 后 · key(item.id) 修正版

按业务 ID 匹配身份，B 对应的 Slot 随之销毁
C 正常保持未勾选、折叠、×1 与「普通商品」



Google Developer Groups

对象身份与旧回调：一次性绑定宿主的闭包过期

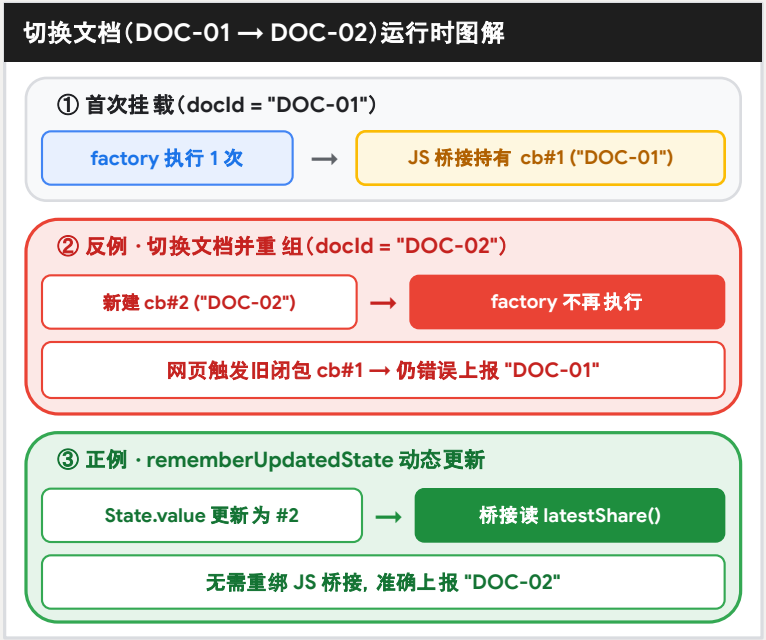
✕ Incorrect · 重组新建闭包，但 JS 桥接仅注册一次

```
@Composable
fun WebBox(docId: String, onShare: (String) -> Unit) {
  val cb = { onShare(docId) } // 重组新建，但宿主未替换
  AndroidView(factory = { ctx ->
    WebView(ctx).apply {
      addJavascriptInterface(JsBridge { cb() }, "app")
    }
  })
}
```

✓ Correct · 保持桥接稳定，内部读取最新回调

```
val latestShare by rememberUpdatedState { onShare(docId) }
AndroidView(factory = { ctx ->
  WebView(ctx).apply { // 桥接只注册一次，触发时读最新引用
    addJavascriptInterface(JsBridge { latestShare() }, "app")
  }
})
```

切换文档（DOC-01 → DOC-02）运行时图解



重组新建闭包不会自动替换一次性宿主的旧绑定，应保持桥接稳定并用 rememberUpdatedState 读取最新回调。

共享资源清理竞态：转场期间误释放与 BindingToken

✕ Incorrect · 仅比对 videoId，无法区分新旧持有者

```
// Feed 与详情页共享播放器，转场期间续播《伟大的长征》
DisposableEffect(playerPool, videoId, surface) {
  playerPool.attach(videoId, surface)
  onDispose {
    if (playerPool.currentVideoId == videoId)
      playerPool.release() // 误将详情页的《伟大的长征》释放
  }
}
```

✓ Correct · 每次绑定颁发 BindingToken，校验后释放

```
// token = BindingToken(videoId, pageId, generation)
DisposableEffect(playerPool, videoId, pageId, surface) {
  val token = playerPool.attach(videoId, pageId, surface)
  onDispose {
    playerPool.releaseIfCurrent(token) // 凭证匹配才释放
  }
}
```

Feed 跳详情页无缝续播 · 转场时序图解

T0 Feed 页挂载并播放《伟大的长征》
当前持有者：Feed · Token#1 ("伟大的长征", gen=1)

↓

T1 详情页先 attach("伟大的长征") 接管播放
新持有者：Detail · Token#2 ("伟大的长征", gen=2)

↓

T2 · 动画结束，Feed 页延迟触发 onDispose

✕ 仅比对 videoId：current == "伟大的长征" 依然成立
→ release() 被错误执行，详情页画面瞬间黑屏

✓ 校验 BindingToken：Token#2 != Token#1
→ 旧清理安全跳过，详情页继续流畅播放

审查共享资源释放不能只看有没有写 onDispose 或比对资源 ID，必须校验清理方是否仍持有本次 BindingToken。

取消过期异步等待：collectLatest 与挂起后复查

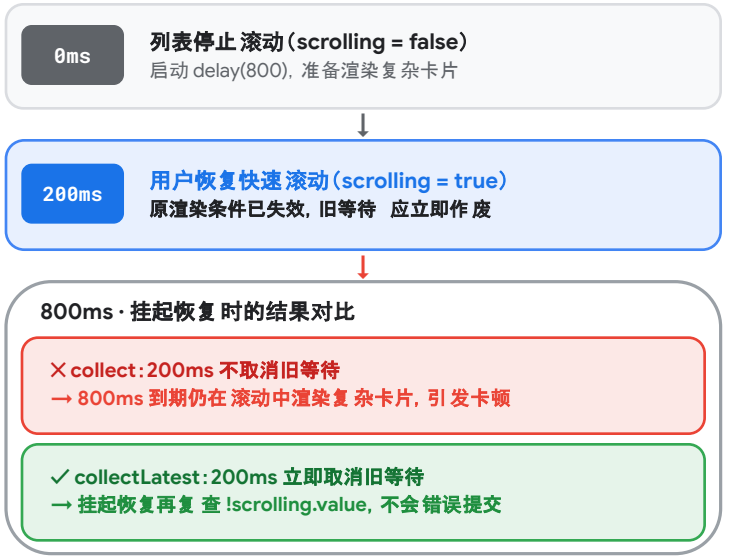
✕ Incorrect · collect 顺序消费，恢复滚动后仍触发过期渲染

```
// 场景：列表停止滚动后，延迟 800ms 渲染复杂卡片
LaunchedEffect(scrolling) {
  snapshotFlow { scrolling.value }.collect { busy ->
    if (busy) return@collect
    delay(800) // 200ms 恢复滚动不会取消等待
    commitHeavyCard() // 滚动中仍触发过期渲染
  }
}
```

✓ Correct · collectLatest 立即取消旧等待并复查状态

```
LaunchedEffect(scrolling) {
  snapshotFlow { scrolling.value }.collectLatest { busy ->
    if (busy) return@collectLatest
    delay(800) // 新状态到来立即取消旧等待
    if (!scrolling.value) commitHeavyCard() // 恢复后复查
  }
}
```

停止滚动后延迟渲染 · 并发取消时序图解



可取消的 UI 异步等待应用 collectLatest 取消旧任务并在恢复后复查状态；必须完成的业务操作应下沉至 ViewModel。

Popup 跨窗口定位:就地挂 载与窗口防越界

✕Incorrect · 脱离锚点手动监听坐标并回写状 态

```
// 脱离锚点挂载，且未按窗口与 弹窗尺寸限制 边界
var rect by remember { mutableStateOf(IntRect.Zero) }
Box(
  Modifier.onGloballyPositioned {
    // 局部坐标错位；改窗口坐 标回写则滚动每帧重组
    rect = it.boundsInParent().roundToIntRect()
  }
) { HelpIcon(onClick = { showTip = true }) }

if (showTip) {
  Popup(remember(rect) { RawPos(rect) }) { TipCard() }
}
```

局部坐 标错位;若改用窗口坐 标回写则滚动每帧重组

✓Correct · 就地挂 载复用窗口坐 标并限制 边界

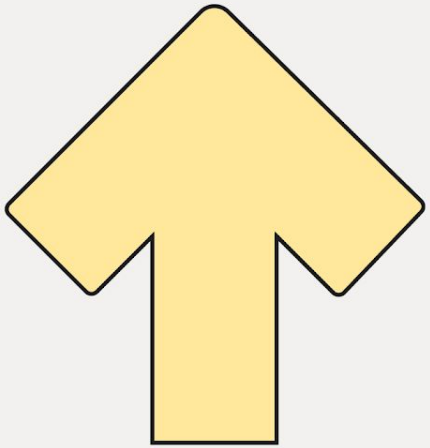
```
Box { // 1. 就地挂进锚点内，复用系 统窗口坐标
  HelpIcon(onClick = { showTip = true })
  if (showTip) Popup(SafeTipPos) { TipCard() }
}

object SafeTipPos : PopupPositionProvider {
  override fun calculatePosition(
    a: IntRect, w: IntSize,
    d: LayoutDirection, p: IntSize,
  ) = IntOffset( // 2. 结合窗口与弹窗尺寸防越界
    a.left.coerceIn(0, max(0, w.width - p.width)),
    a.bottom.coerceIn(0, max(0, w.height - p.height)),
  )
}
```

仅展开时在布局 阶段追踪，零状 态回写且防越界

Popup 应就地挂在 锚点容器内复用原生窗口坐 标 anchorBounds，避免手 动测量局部坐 标错位或窗口坐 标回写重 组。

02



用 Agent Skill 把关

Google
Developer
Groups

SECTION 02 · PROGRESSIVE DISCLOSURE

三级渐进式披露：SKILL.md + references/ + AGENTS.md

compose-guardrails/ 目录结构

```
compose-guardrails/
├── SKILL.md                    # ~100 行 · 索引与场景路由表
├── references/                 # ~1,300 行 · 6 大模块正反例
│   ├── 01-state-and-collections.md
│   ├── 02-identity-and-bridges.md
│   ├── 03-lifecycle-and-resources.md
│   ├── 04-phases-and-recomposition.md
│   ├── 05-tasks-scroll-and-popups.md
│   └── 06-component-api-and-lazy.md
├── scripts/build_agents_md.py
└── AGENTS.md                   # ~1,300 行 · 全量审查合并版
```

Level 1 · YAML 元数据 (~60 Tokens)

仅含名称与触发描述，非 Compose 任务零负担

Level 2 · SKILL.md 路由入口 (~100 行)

编译清零基线 + 反模式速查表 + 场景路由表

Level 3 · references/*.md (按需精读)

写列表只读 02+06，写动画只读 04，注意力高度集中

Full Review · AGENTS.md (~1,300 行)

独立 Code Review 会话一次性加载，避免遗漏子模块

编码阶段按需加载 ~400 行相关规则；独立审查一次性加载 ~1,300 行完整手册（占上下文窗口 <10%）。

SECTION 02 · RULE ANATOMY

面向 Agent 改写规则：边界清晰、不教条的卡片 结构

单条规则核心 5 要素

- **触发信号**：指明代码中可识别的结构特征
- **行为边界**：明确推荐做法与常 见误用写法
- **底层原理**：说明 Snapshot、Slot 与三阶段原理
- **最小对照**：Incorrect 反例 + Correct 正例
- **例外边界**：说明不适用 场景，防止过度优化

节选自 references/04-phases-and-recomposition.md

```
### 禁止提前读取状态导致延迟读取失效
* **影响等级**：CRITICAL
* **触发信号**：参数使用 () -> T，但父组件先写
  val v = state.value 再传递 { v }
* **底层原理**：Snapshot 按实际读取阶段注册依赖；
  函数签名本身不带来优化，调用阶段才决定依赖位置。
* **反例**：val v = progress.value; Track({ v })
* **正例**：Track({ progress.value }) // Canvas 内读取
* **例外边界**：决定分支显隐或文本内容时不可下沉至
  Drawing 阶段，应改用独立叶子组件缩小重组作用域。
```

反例帮助 AI 识别隐蔽误用，例外边界防止机械套用引发负优化。

SECTION 02 · RULE-GUIDED GENERATION

编码阶段规范引导：挂载 compose-guardrails Skill 前后对比

✕ 未挂载 Skill 直接生成（编译通过）

```
@Composable
fun ChatProgressScreen(
  initialTitle: String, msgs: List<ChatMsg>,
  progress: State<Float>,
) {
  // 漏传 key：上游 initialTitle 变更后标题不更新
  var title by remember { mutableStateOf(initialTitle) }
  // 组合阶段提前读高频进度，导致父组件与进度条每帧重组
  val p = progress.value
  Column {
    TopBar(title) { title = it }
    ProgressTrack({ p })
    LazyColumn { itemsIndexed(msgs, { i, _ -> i }) { ... } }
  }
}
```

同时存在状态不同步、提前读取状态与下标充当 key 三类隐患

✓ 挂载 compose-guardrails Skill 后生成

```
@Composable
fun ChatProgressScreen(
  title: String, onTitleChange: (String) -> Unit,
  msgs: List<ChatMsg>, progressProvider: () -> Float,
  modifier: Modifier = Modifier,
) {
  Column(modifier) {
    TopBar(title, onTitleChange)
    ProgressTrack(progress = progressProvider) // 绘制时读取
    LazyColumn {
      items(msgs, key = { it.id }, contentType = { "msg" }) {
        MessageRow(it)
      }
    }
  }
}
```

状态提升 + 延迟读取透传 + 稳定业务 key 与 contentType

编码前按场景路由加载正反例，让第一版代码直接符合工程规范。

SECTION 02 · DEMO #3 (AGENT REVIEW)

独立审查阶段：Agent Review 检出静态漏检缺陷

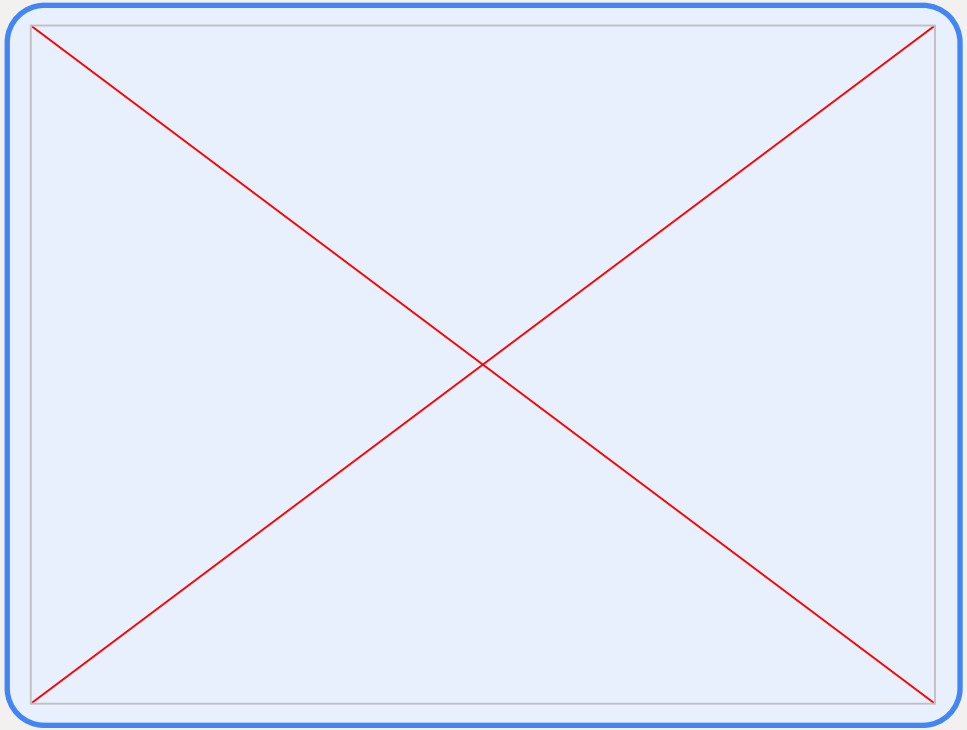
独立审查会话输出摘要

审查文件：PromoTrackCard.kt（编译 0 Error，lintDebug 0 Error）

[CRITICAL] L100 · 共享资源清理缺少持有者凭证校验
问题：DisposableEffect 漏传 holder/ownerId，且 onDispose 仅比对 resourceId，转场接管时易误释放新持有者的资源
修复：key 补齐（holder，resourceId，ownerId），保存 bind() 返回的 BindingToken，清理时调用 clearIfCurrent(token)

[CRITICAL] L112 · 组合阶段提前解包导致延迟读取失效
问题：提前读取 progress.value 再包进 { current }，使整张卡片订阅高频进度，每帧触发无效重组
修复：删去组合阶段解包，直接传 progressProvider = { progress.value }，将状态读取推迟到 Canvas 绘制阶段

[HIGH] L113 · 高频进度转低频阈值未做 derivedStateOf 收敛
问题：直接按进度计算 endingSoon 阈值，未跨过 0.9f 时每次进度更新仍会触发卡片重组
修复：改用 remember(progress) { derivedStateOf { progress.value.coerceIn(0f, 1f) >= 0.9f } } 过滤无效重组



Google Developer Groups

SUMMARY · KEY TAKEAWAYS

全景总结：AI 时代的 Compose 代码质量防线

1 次范式升级

- **过去**：Compiler-based Lint
  检查局部 AST 与类型签名
- **现在**：Agent-based Lint
  理解频率、身份、所有权与宿主边界

3 道协同防线

- **编码阶段**：按需加载正反例
- **编译阶段**：静态告警清零
- **审查阶段**：独立语义审查

可扩展的语义模块

- 状态所有权与可观察集合
- 组合身份、外部数据源与会话
- BindingToken 与四态生命周期
- 读取作用域与延迟状态读取
- 异步取消、滚动控制与 Popup
- 组件分层、Modifier 与 Insets

将运行机制与生产经验沉淀为可执行的 Agent Skill，才能真正守住 AI 时代的工程质量上限。



Google Developer Groups



编译器 Lint 守住语法下限，
Agent Skill 提升语义上限。

Thanks!
Q & A