Kotlin

NTCompose

QQ 团队基于 KMP + Compose 的原生渲染跨端方案

分享人：林锦涛

Developed by Tencent QQ Team

目录

1. NTCompose 简介

2. NTCompose 设计

3. NTCompose 鸿蒙适配高性能方案

目录

1. NTCompose 简介

2. NTCompose 设计

3. NTCompose 鸿蒙适配高性能方案

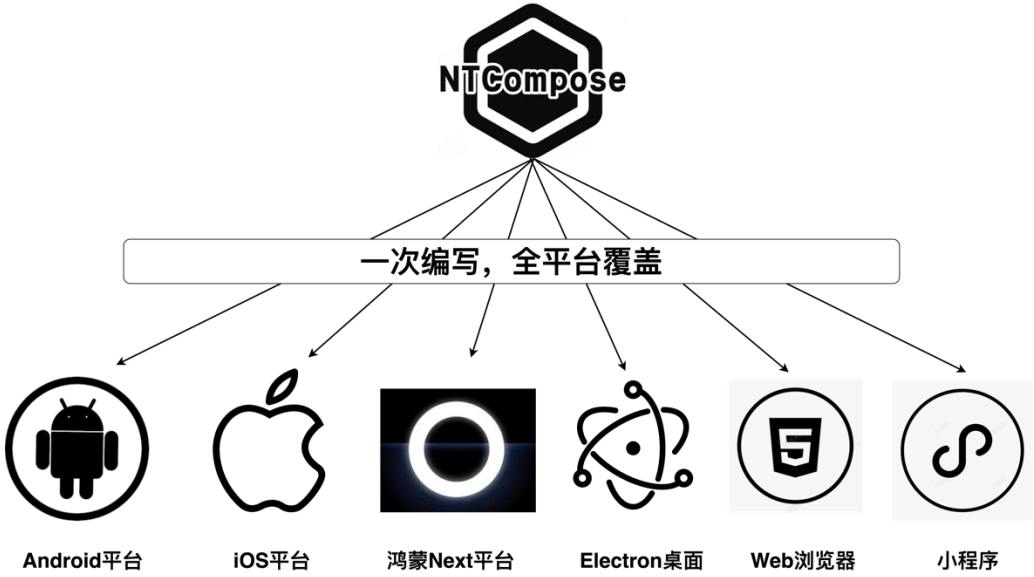
NTCompose: QQ团队基于 KMP + Compose 的原生渲染跨端方案

- 背景: 在多元化设备的背景下, 尤其是鸿蒙 Next 的出现, 急需一种无缝适配多平台&保持原生性能的跨端方案
- NTCompose: QQ团队基于KMP, Compose开发范式构建的原生渲染跨端开发框架

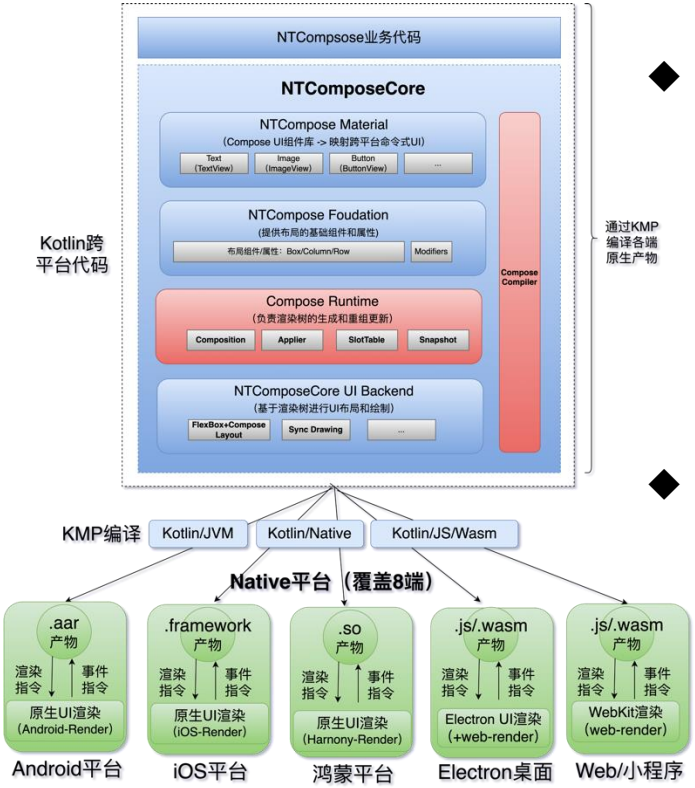
  优势1: 纯原生体验 -> 原生的 执行性能 & 原生的 UI控件渲染

  优势2: 标准 Compose 开发范式 -> 复用 Compose 语法开发 & Compose 业界生态

  优势3: 跨全平台 -> 一码跨 6 端 (Android/iOS/Harmony/PC/Web/小程序)

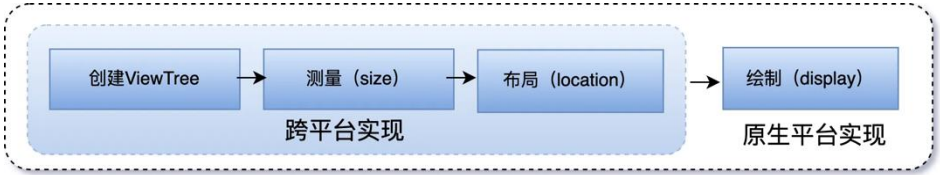


NTCompose跨端核心架构: KMP + Compose Runtime 驱动原生 UI



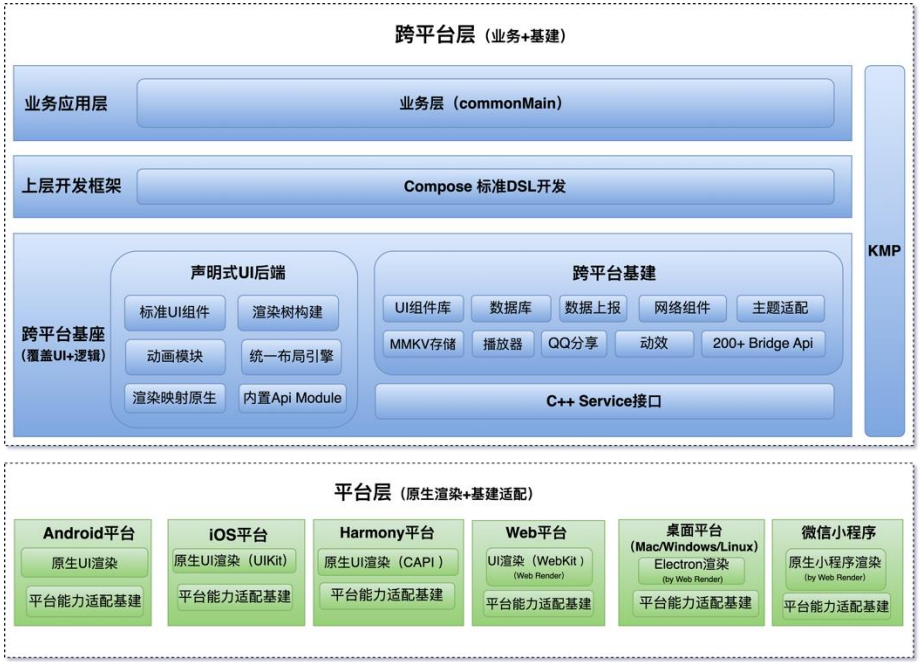
- NTCompose分层设计: 跨平台侧 NTComposeCore + 平台侧 Render
  1. NTComposeCore：架构上对齐 Jetpack Compose分层设计，由 Compose Runtime 驱动 NTComposeCore UI Backend
  2. 平台Render：跨平台渲染指令(Create, Insert, Remove, Update)实现层, 底层映射到平台 UI 控件

- 最大化原生UI一致性: 渲染流程提升至跨平台, 平台侧提供UI渲染原子能力
  1. BuildTree -> Measure -> Layout 提升至跨平台侧
  2. 绘制阶段由平台侧提供原子化的 UI 渲染能力



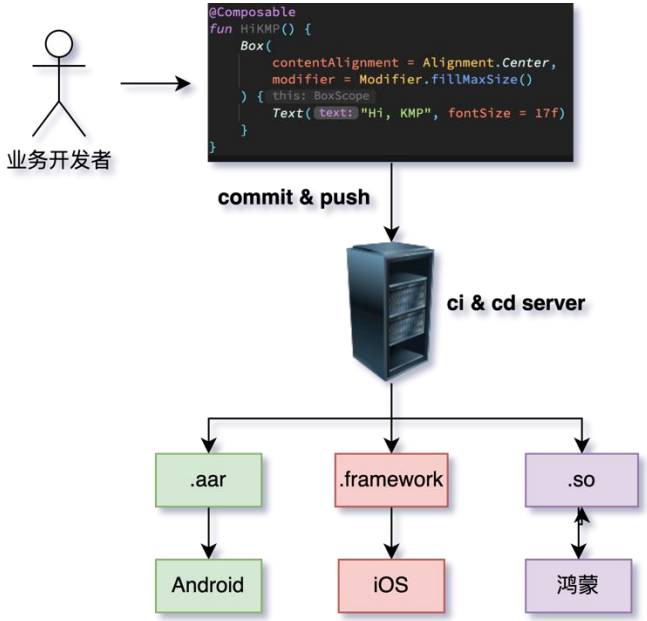
基于 NTCompose 的一码多端应用架构

- 隔离操作系统的完全跨平台 (逻辑 + UI)开发环境
  1. UI：提供了 NTCompose 声明式开发框架
  2. 逻辑：提供了丰富的对标系统能力的 API



基于NTCompose的整体跨端架构

- 一码多端同构流程
  1. 业务开发聚焦commonMain逻辑编写
  2. CI&CD 同构多端原生产物
  3. 同构产物集成到宿主 app



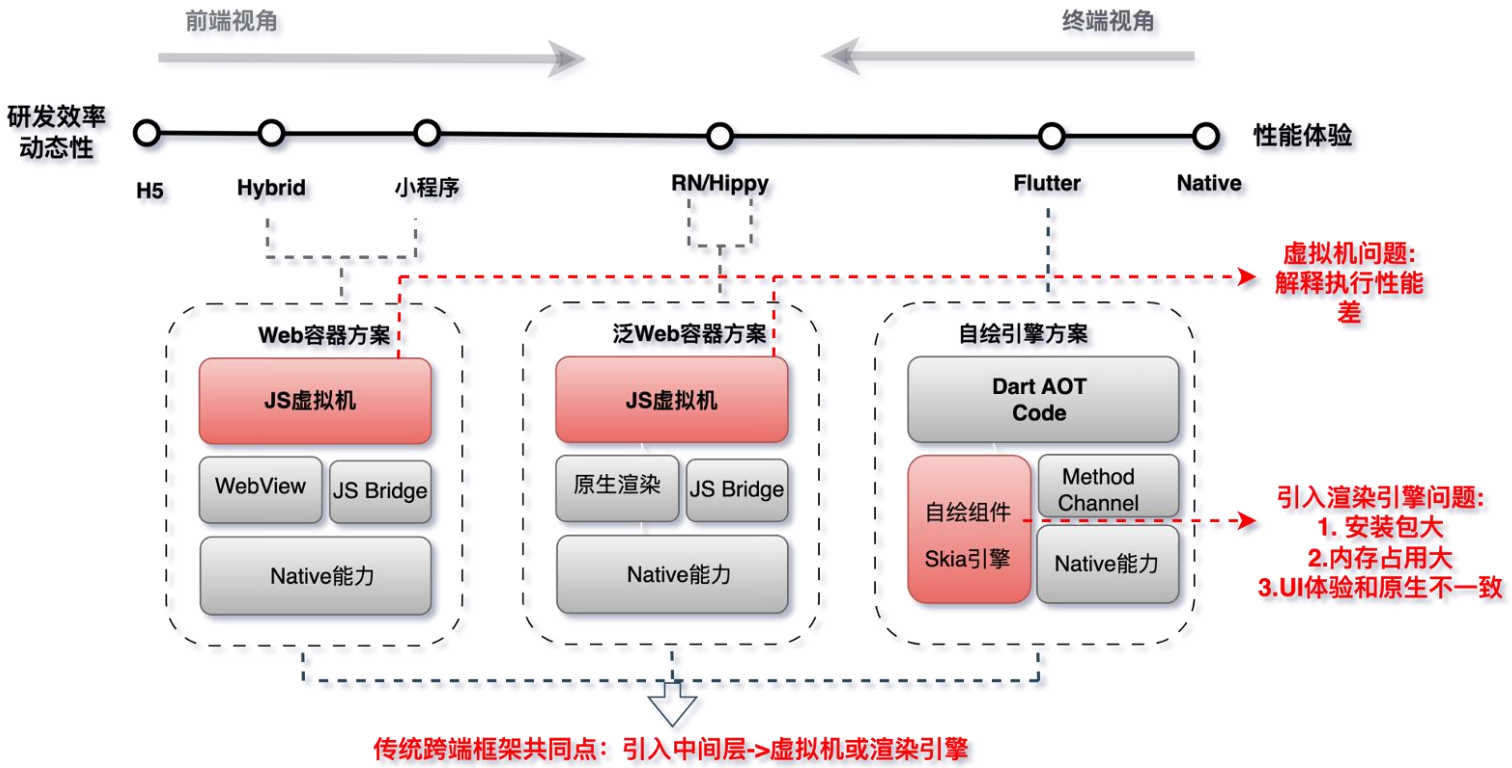
目录

1. NTCompose 简介

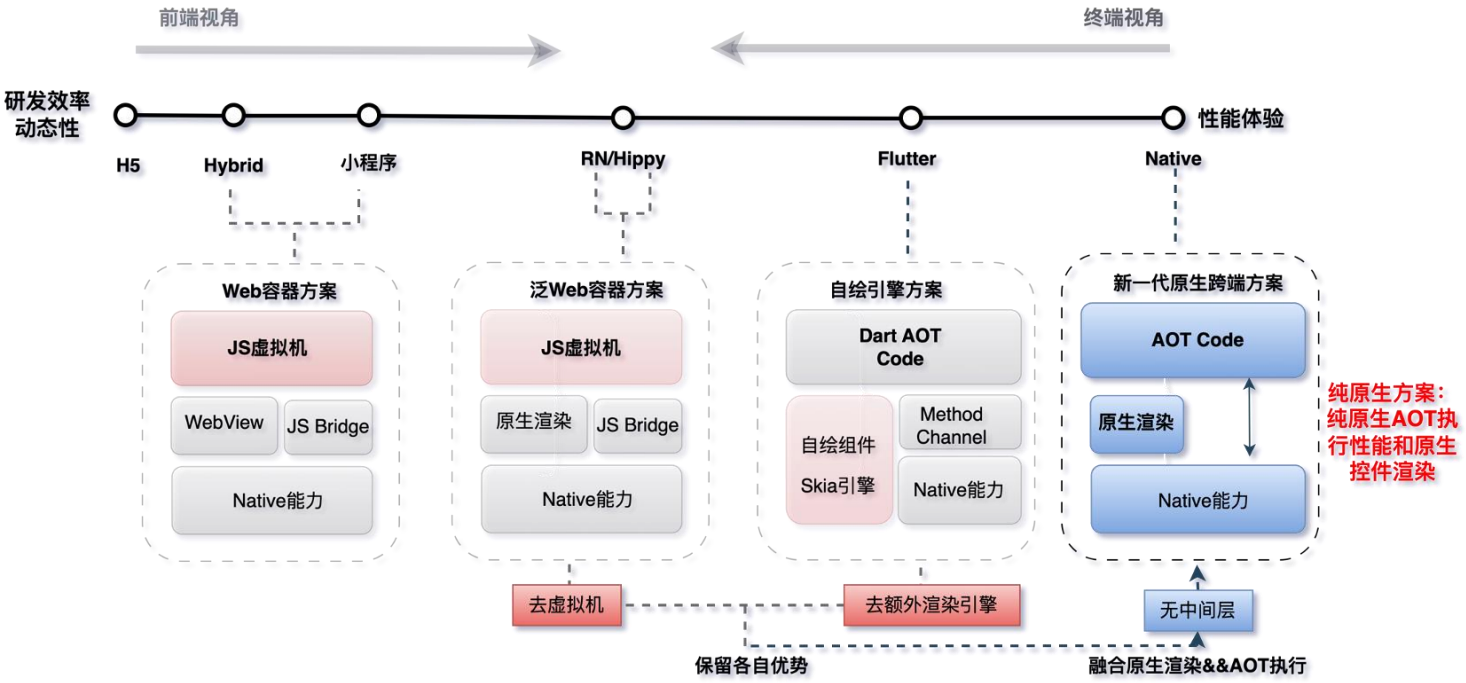
2. NTCompose 设计

3. NTCompose 鸿蒙适配高性能方案

移动端跨端框架演进历史

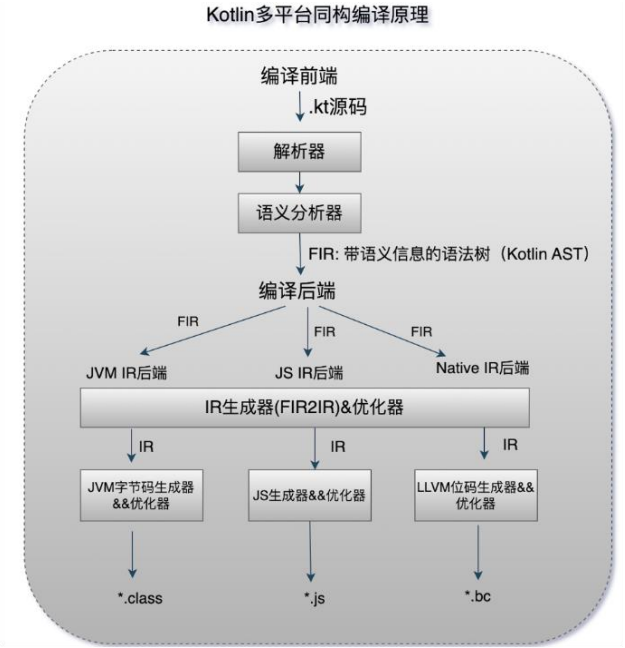


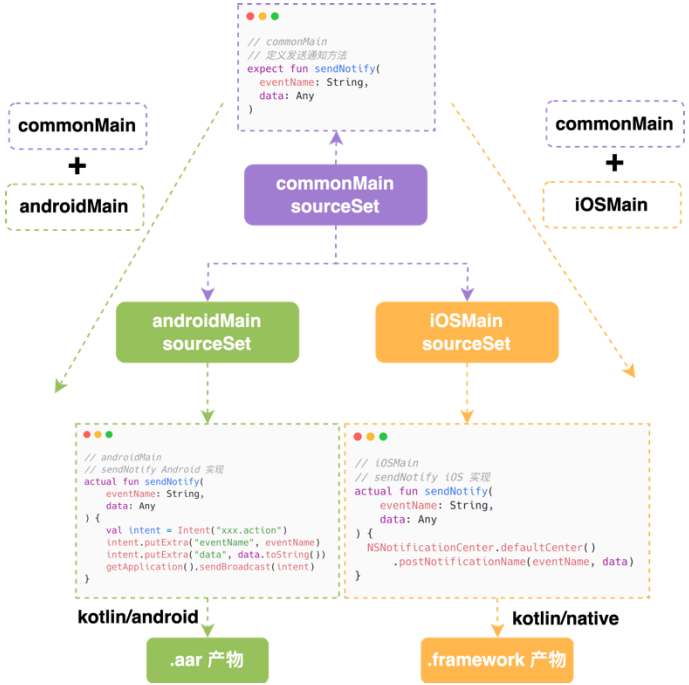
QQ NTCompose: 重新定义跨端，去掉中间层，打造纯原生方案



KMP 逻辑跨端：Kotlin 如何跨平台？

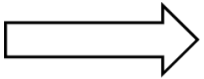
- KMP原理 ：通过 Kotlin 多平台编译器，编译为各个平台的原生执行产物
- 优势 1：原生 AOT 级别执行性能 -> 编译流派 & 无虚拟机
- 优势 2：与原生平台交互零成本 -> 通过 expect / actual 机制与平台 API 交互，避免胶水层
- 优势 3：语言高度同源 -> 统一技术栈

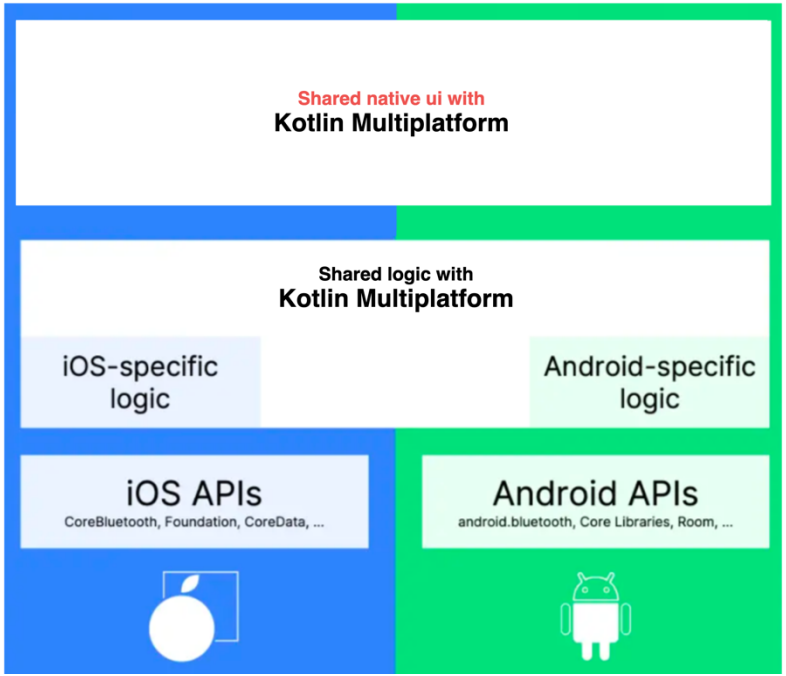




原生 UI 跨端：KMP 不支持原生 UI 跨端，如何破局？

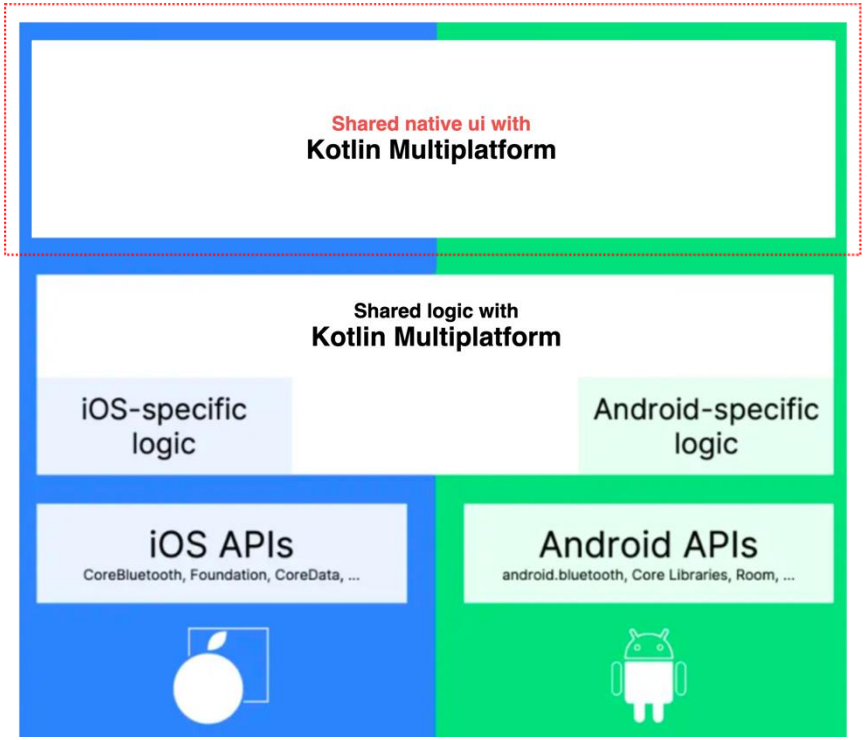
- 在 KMP 中，原生 UI 不支持跨端，需要各端编写
- 原生 UI 如何像逻辑跨端一样，在 KMP 侧跨端



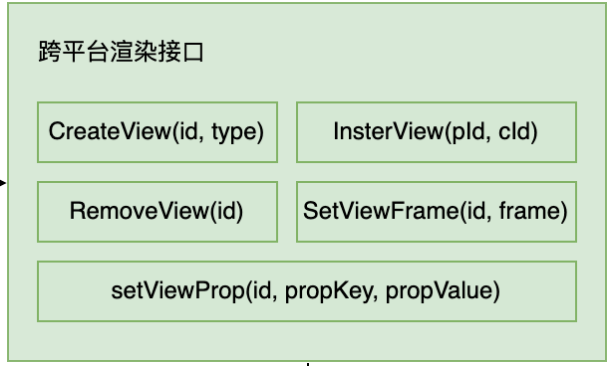


原生 UI 跨端解决方案：KMP 侧 抽象 UI 接口

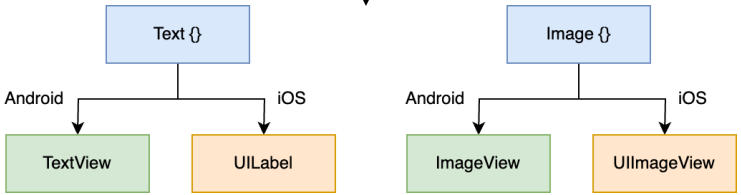
◆ 核心思路：KMP 侧 (commonMain) 抽象操作 UI 的接口，平台侧映射原生 UI 控件



像操作逻辑一样, 操作 UI

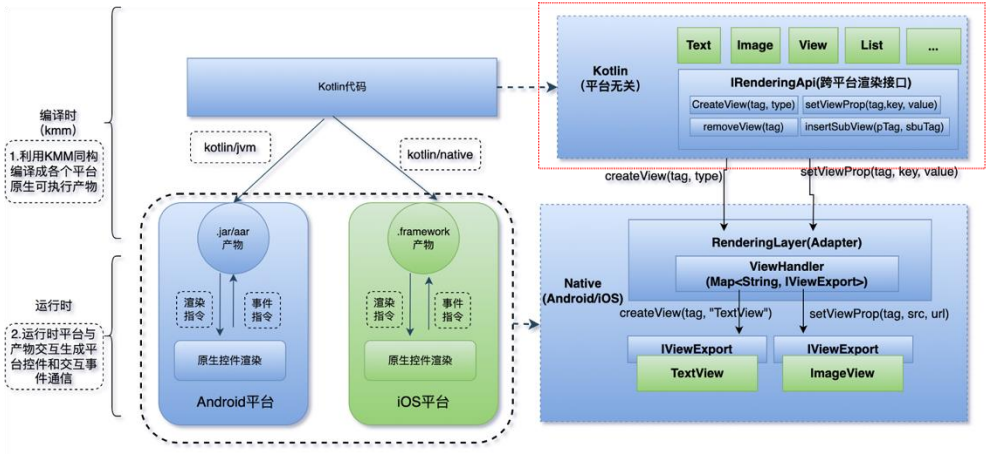


底层映射到原生控件



原生 UI 跨端架构：跨平台命令式 UI 组件

- 抽象跨平台 UI 接口：Create, Insert, Remove, Update
- 跨平台命令式 UI 组件：commonMain 侧构建跨平台命令式 UI 组件



原生跨端 UI架构

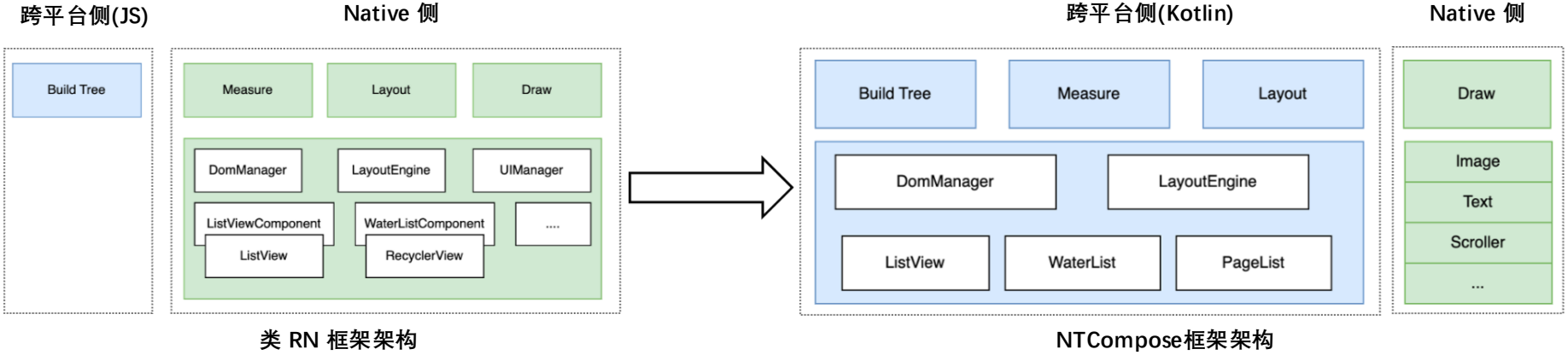
```
// 1.创建容器 View
val viewGroup = ViewGroup()
// 2.创建文本控件
val text = Text()
text.setText("Hi, KMP")
text.sizeToFit()
text.setFrame(0, 0, text.width, text.height)
// 3.添加到父View
viewGroup.addView(text)
```

基于跨平台命令式UI 组件构建页面

跨平台原生UI高一致性方案: 原生UI原子化&渲染流程提升到跨平台

- 类RN框架一致性差原因：重 Native 实现
  1. UI 渲染流程 & UI 组件封装都在 Native
  2. 各端重复实现重逻辑的 UI 组件，容易导致不一致性

- 原生 UI 一致性优化：UI 能力原子化
  1. UI 渲染流程 & UI 组件封装提升到Kotlin跨平台层
  2. 各Native 侧只保留轻逻辑&原子化的 UI 能力。
     由跨平台侧驱动 UI 逻辑



类 RN 框架架构

NTCompose框架架构

Compose开发范式: 跨平台原生 UI + ComposeDSL，加速 UI 开发

- 跨平台命令式 UI 开发范式：开发效率不高
- 业界 UI 开发范式潮流：声明式 UI(Flutter/Compose/SwiftUI)
- 复用标准Compose 开发范式：跨平台命令式 UI + ComposeDSL

“

使用 Compose 构建

相同功能的速度

比使用 Views 大约快 30%”

Rodrigo Represa
Android engineer at adidas

跨平台命令式 UI

```
// 1.创建容器 View
val viewGroup = ViewGroup()
// 2.创建文本控件
val text = Text()
text.setText("Hi, KMP")
text.sizeToFit()
text.setFrame(0, 0, text.width, text.height)
// 3.添加到父View
viewGroup.addView(text)
```

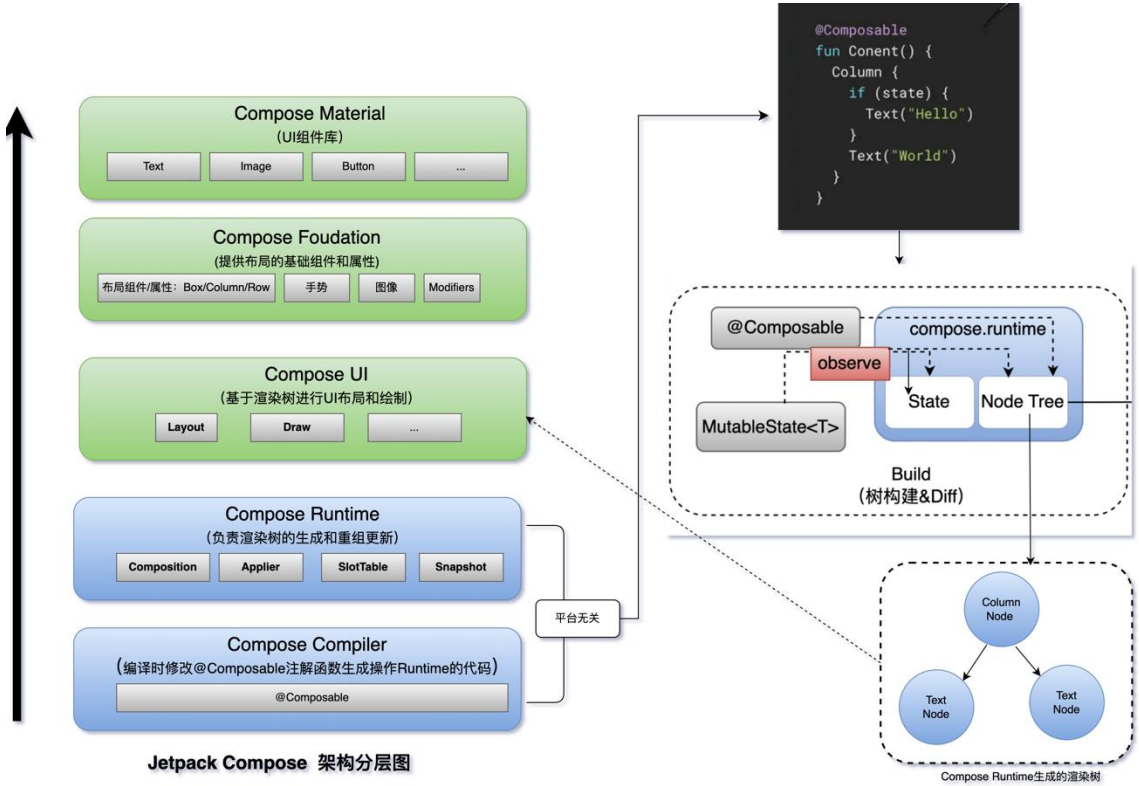
跨平台命令式 UI 结合 标准 Compose DSL

```
@Composable
fun HiKMP() {
    Box(
        contentAlignment = Alignment.Center,
        modifier = Modifier.fillMaxSize()
    ) {
        Text("Hi, KMP")
    }
}
```

Compose开发范式, 提高UI 开发效率

Jetpack Compose架构分层: Compose Runtime是声明式UI 构建基础

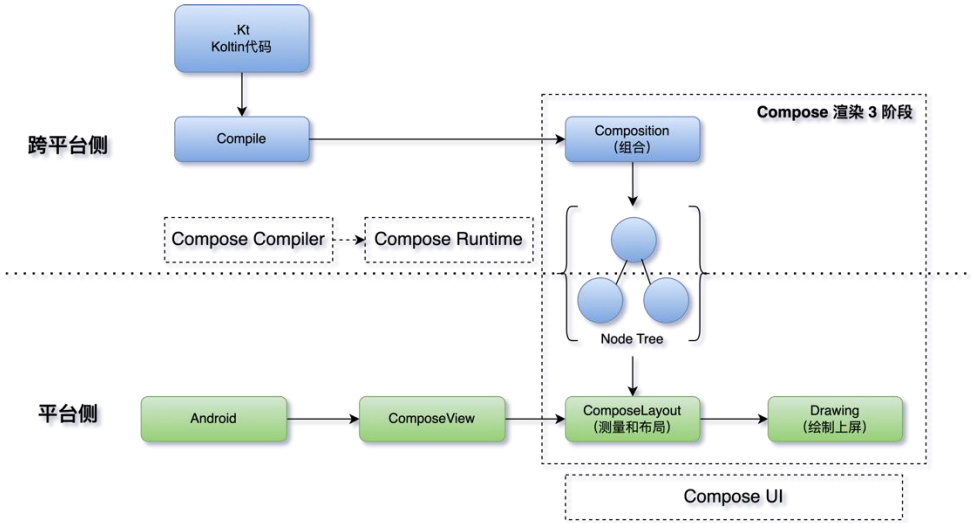
- Compose Runtime：
  1. 提供声明式 UI 语法
  2. 创建 & 维护 NodeTree
  3. 平台无关

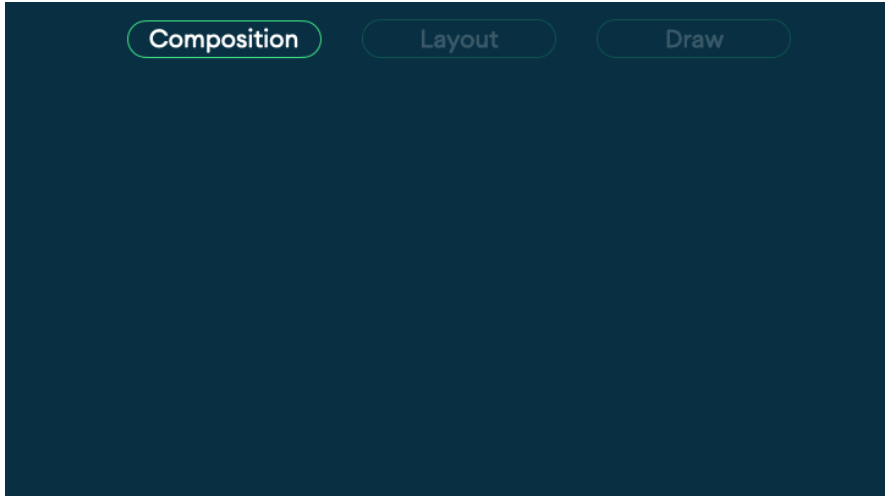


Jetpack Compose 架构分层图

Jetpack Compose渲染流程分析: 基于Compose Runtime可生成平台无关的渲染树

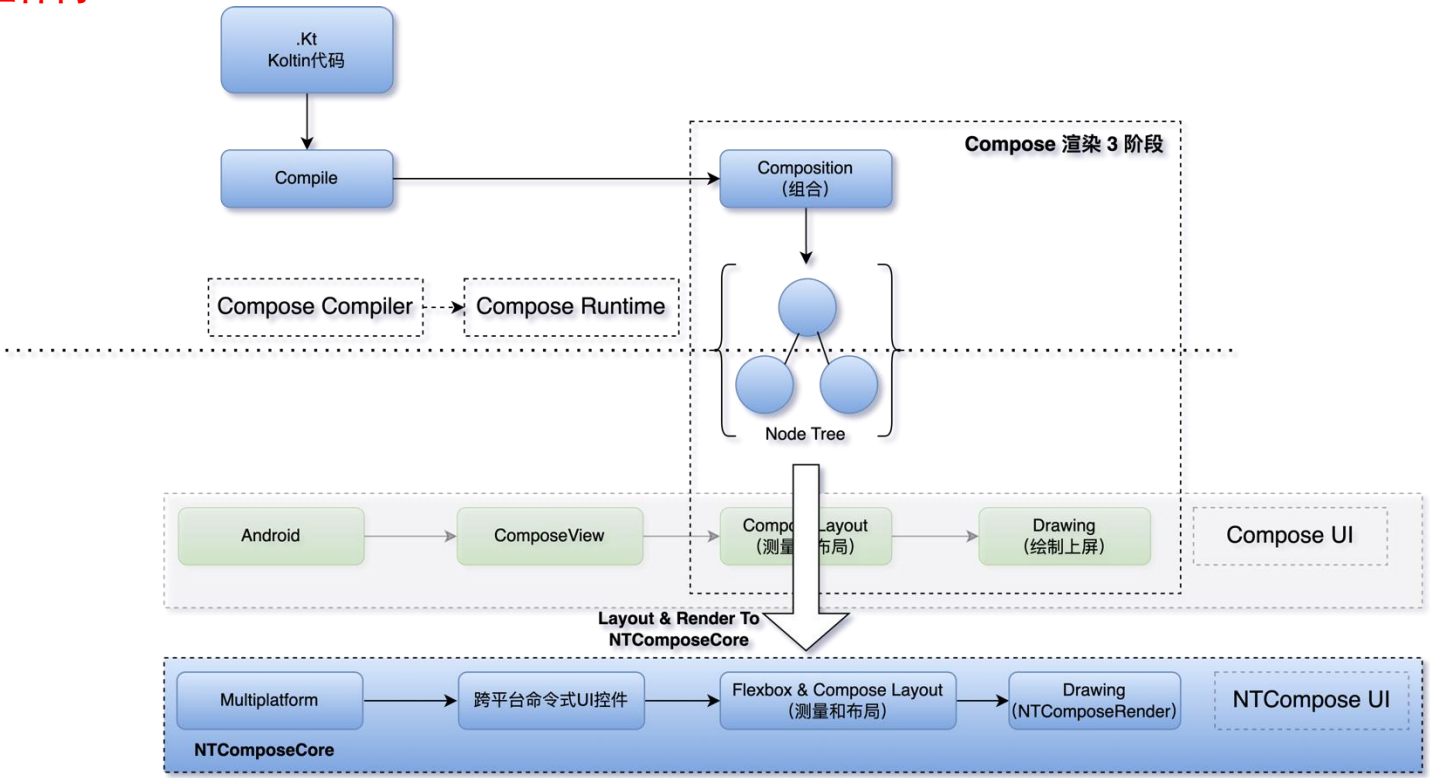
◆ 通过Compose Runtime驱动下产生渲染3 个阶段：Composition -> Layout -> Drawing





NTCompose原生方案之渲染：基于 Compose渲染树映射原生 UI

- 利用平台无关的 Compose Node Tree 实现各个平台原生渲染的最短路径: Compose 渲染树**映射跨平台命令式UI控件树**



NTCompose原生方案实现: 基于Compose Runtime构建跨平台命令式 UI树和更新

- 自定义 Applier，对接跨平台命令式 UI View，实现 ViewTree 增删操作

- 基于 ComposeNode 实现跨平台命令式 UI View 创建和更新

```
实现Compose的Applier节点增删操作
class ViewApplier(private val view: ComposeRenderView)
    : AbstractApplier<ComposeRenderView>(view) {
    override fun insertBottomUp(index: Int, instance: ComposeRenderView) {
        //  current.insert(index, instance)
    }
    override fun insertTopDown(index: Int, instance: ComposeRenderView) {
        current.insert(index, instance)
    }

    override fun move(from: Int, to: Int, count: Int) {
        current.move(from, to, count)
    }
    override fun onClear() {
        current.onClear()
    }
    override fun remove(index: Int, count: Int) {
        current.remove(index, count)
    }
}
```

```
@Composable
fun Image(
    painter: Painter? = null,
    ref: RefFunc<ImageView>? = null,
    contentDescription: String? = null,
    modifier: Modifier = Modifier,
    alignment: Alignment? = null,
    /* 图片缩放适应模式(默认等比放大撑满) */
    contentScale: ContentScale = ContentScale.Crop,
    /* 透明度 */
) {
    ComposeNode<ComposeRenderView, ViewApplier>(
        factory = {
            ImageView().toComposeRenderView()
        },
        update = { this: Updater<ComposeRenderView>
            set(modifier) { this: ComposeRenderView
                this.modifier = it
            }
        }
    )
```

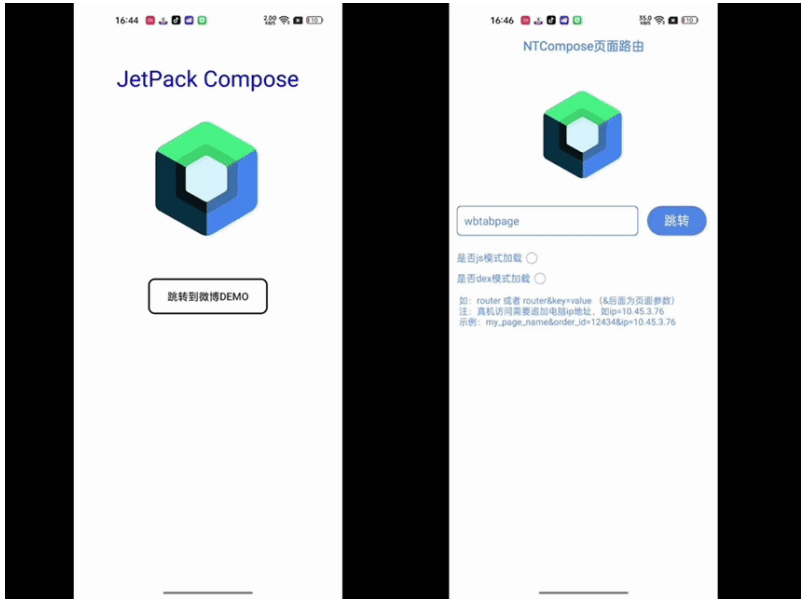
```
* 设置组件的背景颜色
*/
un Modifier.backgroundColor(color: Color): Modifier {
    return setPropWithViewCallback( propKey: "backgroundColor") {attr, event ->
        attr.backgroundColor(color)
    }
```

通过Compose Runtime提供的
Applier&ComposeNode
驱动了渲染树的生成和更新

NTCompose原生方案性能对比

◆ 基于复杂图文 Feed 流页面进行性能测试

框架	首次-启动耗时	非首次-启动耗时	FPS
Jetpack Compose	255ms	150ms	55-59
NT Compose	160ms	127ms	59-60



目录

1. NTCompose 简介

2. NTCompose 设计

3. NTCompose 鸿蒙适配高性能方案

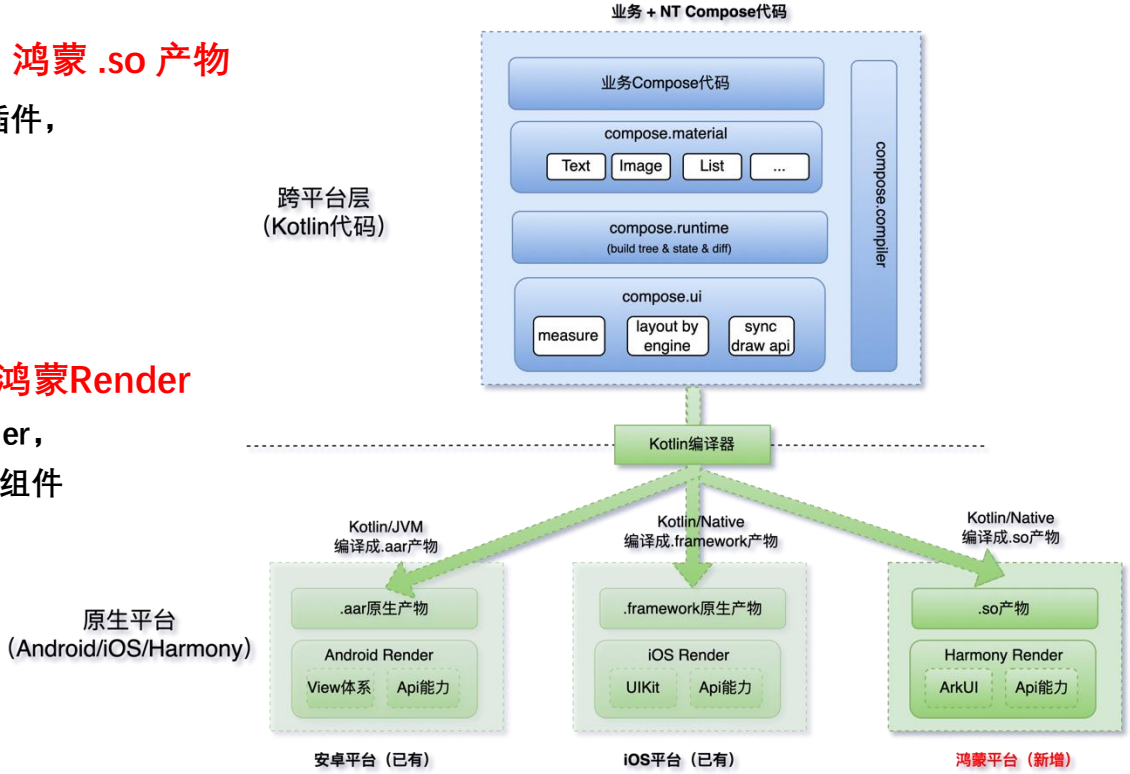
NTCompose适配鸿蒙平台核心思路

- 基跨平台代码 -> 生成目标平台产物：鸿蒙 .so 产物

  1. 新增 ohMain, 修改kotlon/native编译器插件，
     生成鸿蒙平台.so二进制产物

- 平台渲染 -> 映射鸿蒙 ArkUI 渲染：鸿蒙Render

  1. 对齐已有 Android Render或者 iOS Render，
     实现统一跨平台渲染协议映射平台 ArkUI组件



鸿蒙平台UI开发范式

◆ 基鸿蒙平台开发 UI 的两种范式：

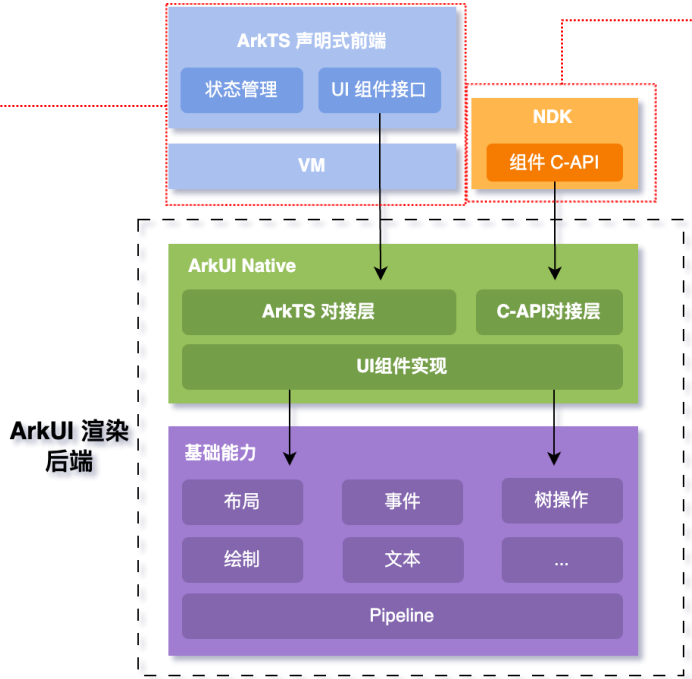
1. 声明式 UI 开发范式：使用 ArkTS 声明式开发范式 构建 UI
2. 命令式 UI 开发范式：使用 NDK 侧 capi 命令式接口 构建 UI

```
auto parent = arkUINativeNodeApi->createNode(ARKUI_NODE_STACK);
auto child = arkUINativeNodeApi->createNode(ARKUI_NODE_STACK);
arkUINativeNodeApi->addChild(parent, child);
arkUINativeNodeApi->removeChild(parent, child);
```

NDK 侧 capi 命令式接口构建 UI

```
Row({ space: 35 }) {
  Text('space: 35').fontSize(15)
  Row().width('10%').height(150)
  Row().width('10%').height(150)
  Row().width('10%').height(150)
}.width('90%')
```

ArkTS 声明式构建 UI



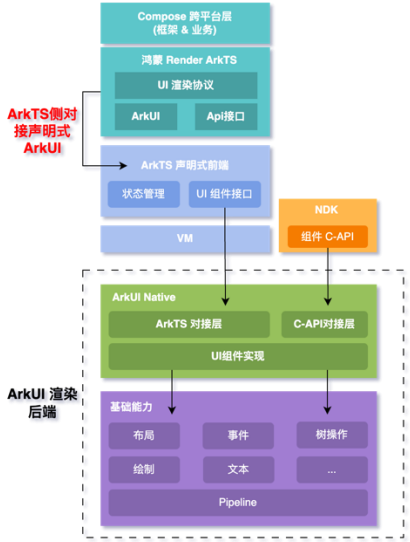
NTCompose 鸿蒙Render适配选型

◆ 跨平台命令式 UI 映射 ArkTS声明式 UI

1. 优点：实现简单，稳定
2. 缺点：性能不佳，存在 ArkTS声明式 UI 更新diff开销

◆ 跨平台命令式 UI 映射capi命令式 UI 接口

1. 优点：高性能，不存在额外开销
2. 缺点：实现难度大

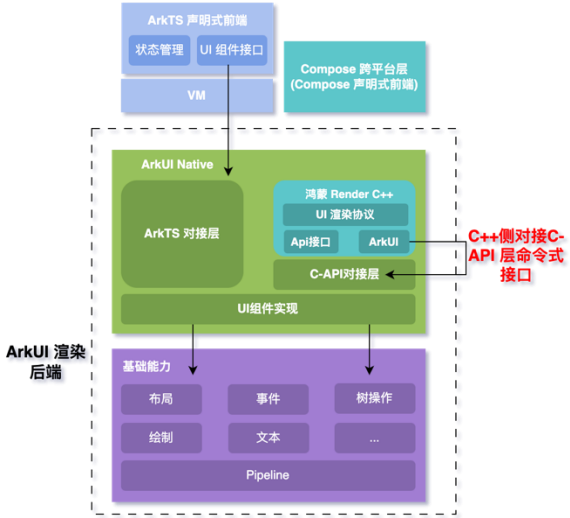


跨平台命令式 UI 组件

```
// 1.创建容器 View
val viewGroup = ViewGroup()
// 2.创建文本控件
val text = Text()
text.setText("Hi, KMP")
text.sizeToFit()
text.setFrame(0, 0, text.width, text.height)
// 3.添加到父View
viewGroup.addView(text)
```

映射声明式 UI 组件

映射命令式 UI 组件




```
Stack() {
  Text("Hi, KMP")
    .fontColor(Color.Black)
}
```

ArkTS声明式 UI 组件

```
auto stack = nodeApi->createNode(ARKUI_NODE_STACK);
auto text = nodeApi->createNode(ARKUI_NODE_TEXT);
nodeApi->addChild(stack, text);
```

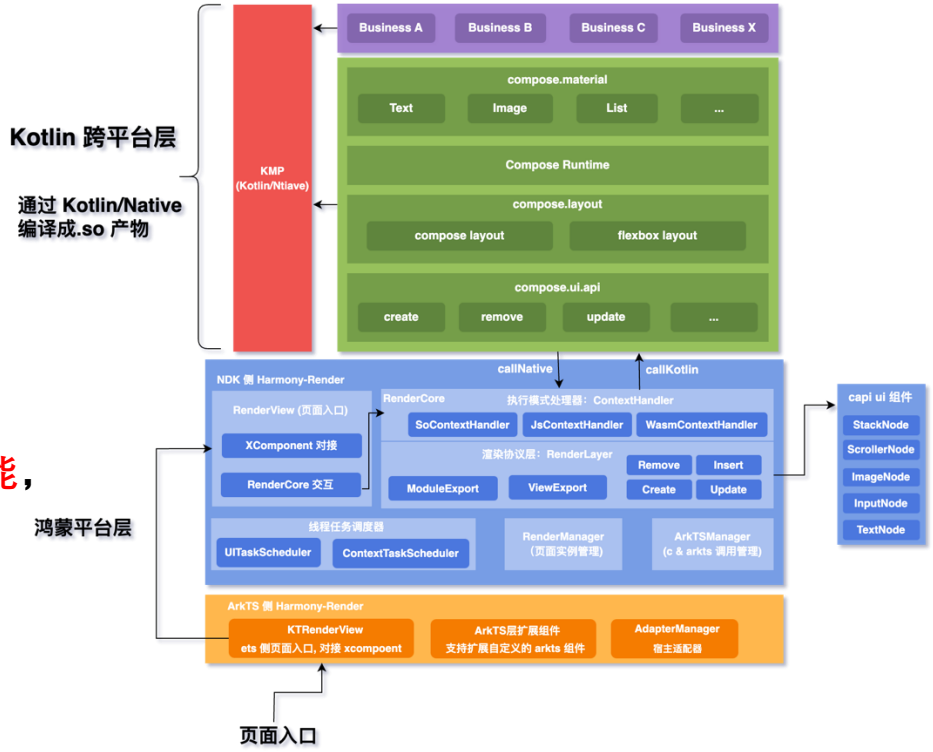
ndk 侧 capi 命令式 UI 组件

NTCompose 高性能适配鸿蒙架构

- 执行产物 AOT 性能：使用Kotlin/Native编译器将业务代码编译成 .so 产物

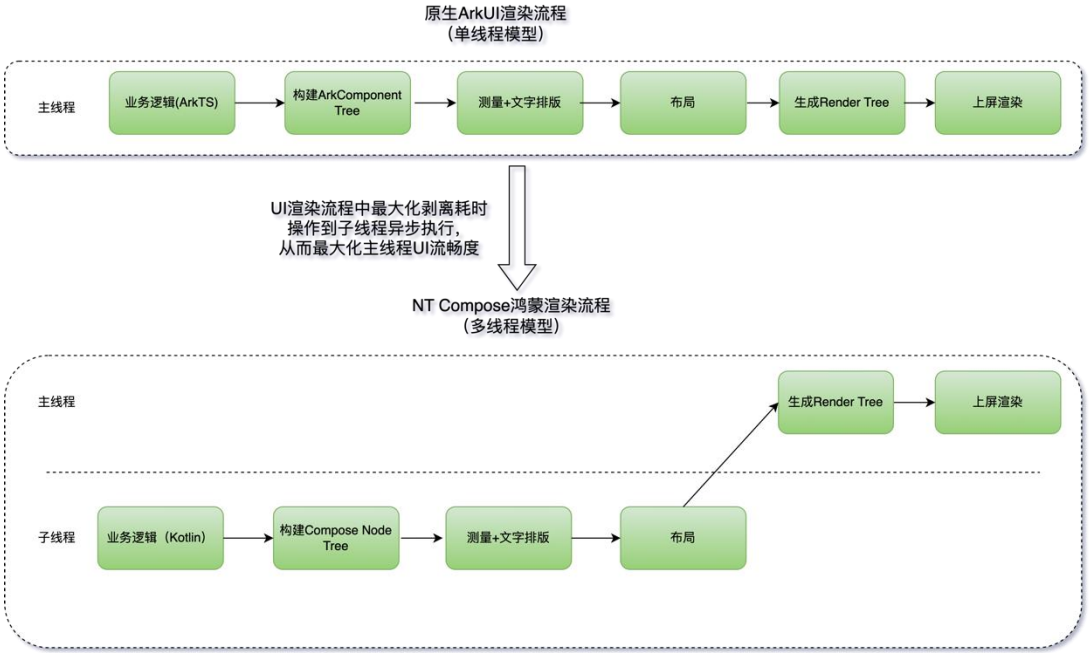
- 原生渲染性能：跨平台命令式 UI 接口对接鸿蒙 ndk侧 capi 命令式UI组件

- 纯C++实现鸿蒙 Render：Render 本身 AOT 执行性能，并且无跨语言损耗对接 capi UI组件



NTCompose鸿蒙Render多线程UI渲染流程：奠定超原生流畅度基础

- 执行设计多线程渲染流程机制：最大化剥离主线程耗时流程到子线程异步执行，从而最大化保证主线程UI流畅度
- 纯C++实现鸿蒙Render：摆脱ArkTS多线程内存隔离限制带来的性能损耗，充分利用C++多线程内存共享优势



NTCompose鸿蒙Render：使用 capi 命令式接口实现渲染协议

◆ 执行基于鸿蒙 capi 命令式接口, 构建 Render渲染协议

```
// 创建 View
GetNodeApi()->createNode(ARKUI_NODE_STACK);
// 删除 View
GetNodeApi()->removeChild(parent_node_, GetNode());
// 插入 View
GetNodeApi()->insertChildAt(node, sub_node, index);
// 更新 View
GetNodeApi()->setAttribute(handle, NODE_MARGIN, &item);
```

capi 命令式 UI 组件接口

基于 capi 接口实现

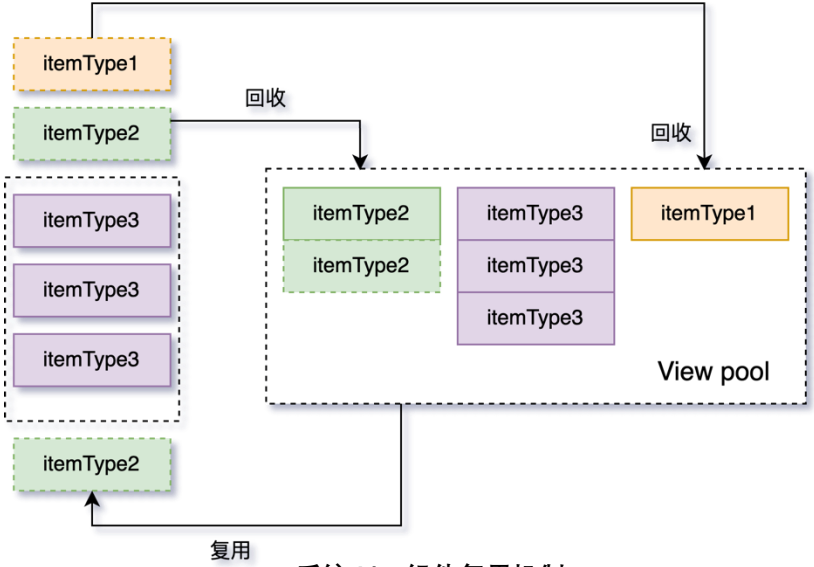
```
class KTRenderLayerHandler {
  // 创建View指令
  void CreateRenderView(
    int tag,
    const std::string &view_name
  )

  // 删除View指令
  void RemoveRenderView(int tag)

  // 插入View指令
  void InsertSubRenderView(
    int parent_tag,
    int child_tag,
    int index
  )

  // 设置View属性
  void SetProp(
    int tag,
    const std::string &prop_key,
    const KTAnyValue &prop_value
  )
}
```

render 渲染协议类

NTCompose鸿蒙Render：渲染协议 View 复用机制

◆ 系统List复用机制存在的问题

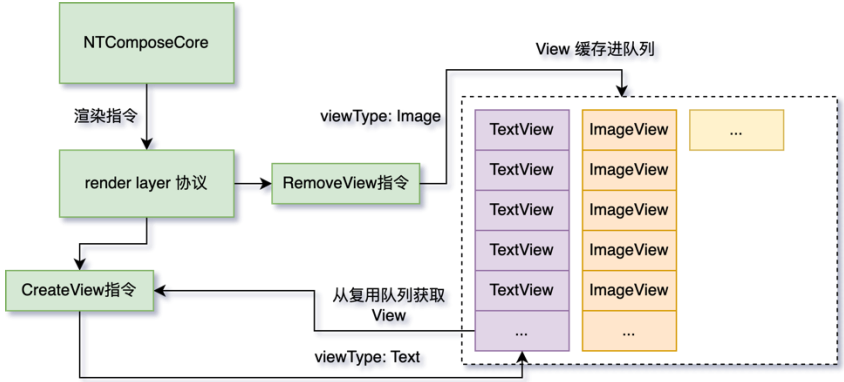
1. 复用粒度为 item 级别，多 itemType 场景容易卡顿
2. 复用范围仅List，无法覆盖页面级别



系统 List 组件复用机制

◆ 鸿蒙 Render 渲染协议 View 复用机制

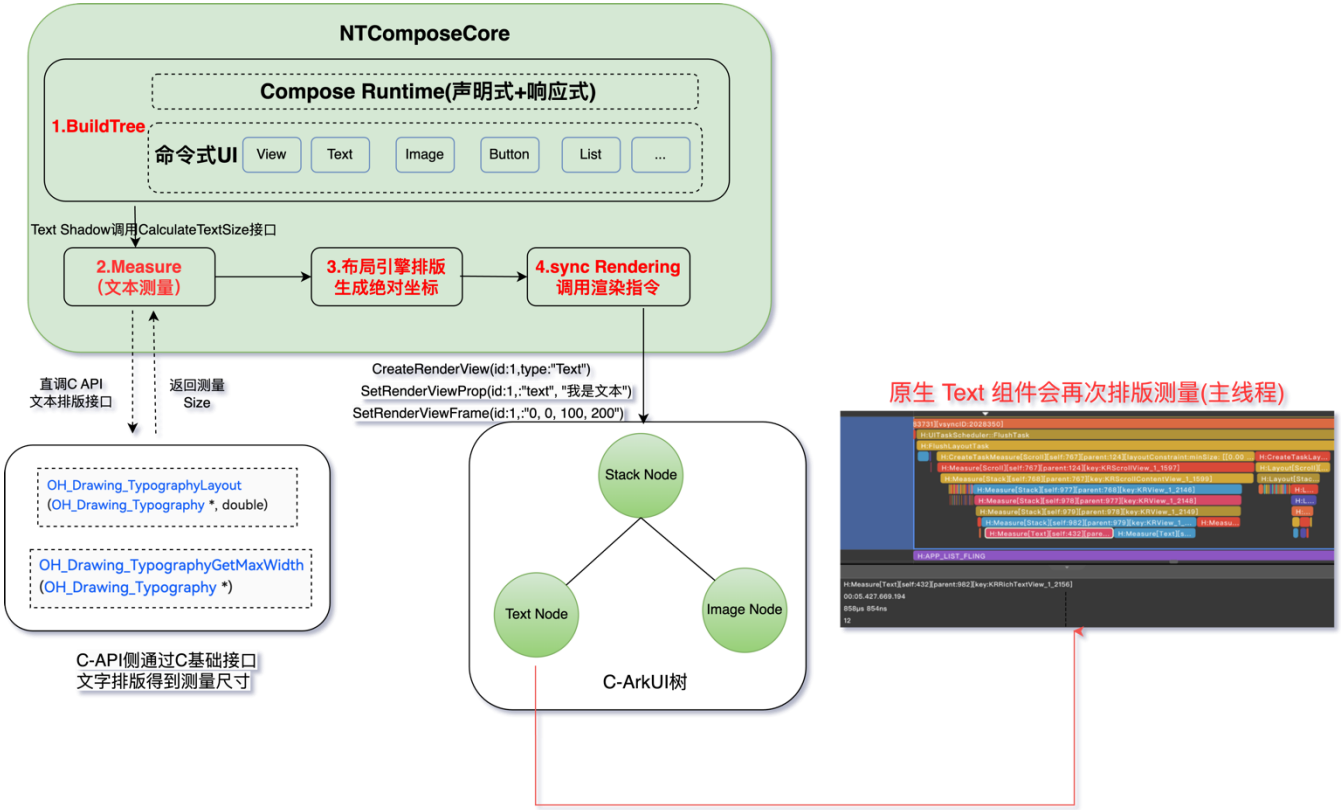
1. 更细粒度复用：基于 View 粒度复用，多 itemType 场景无卡顿
2. 更广的复用范围：回收和复用是基于页面级别



鸿蒙 Render 渲染协议 View 复用机制

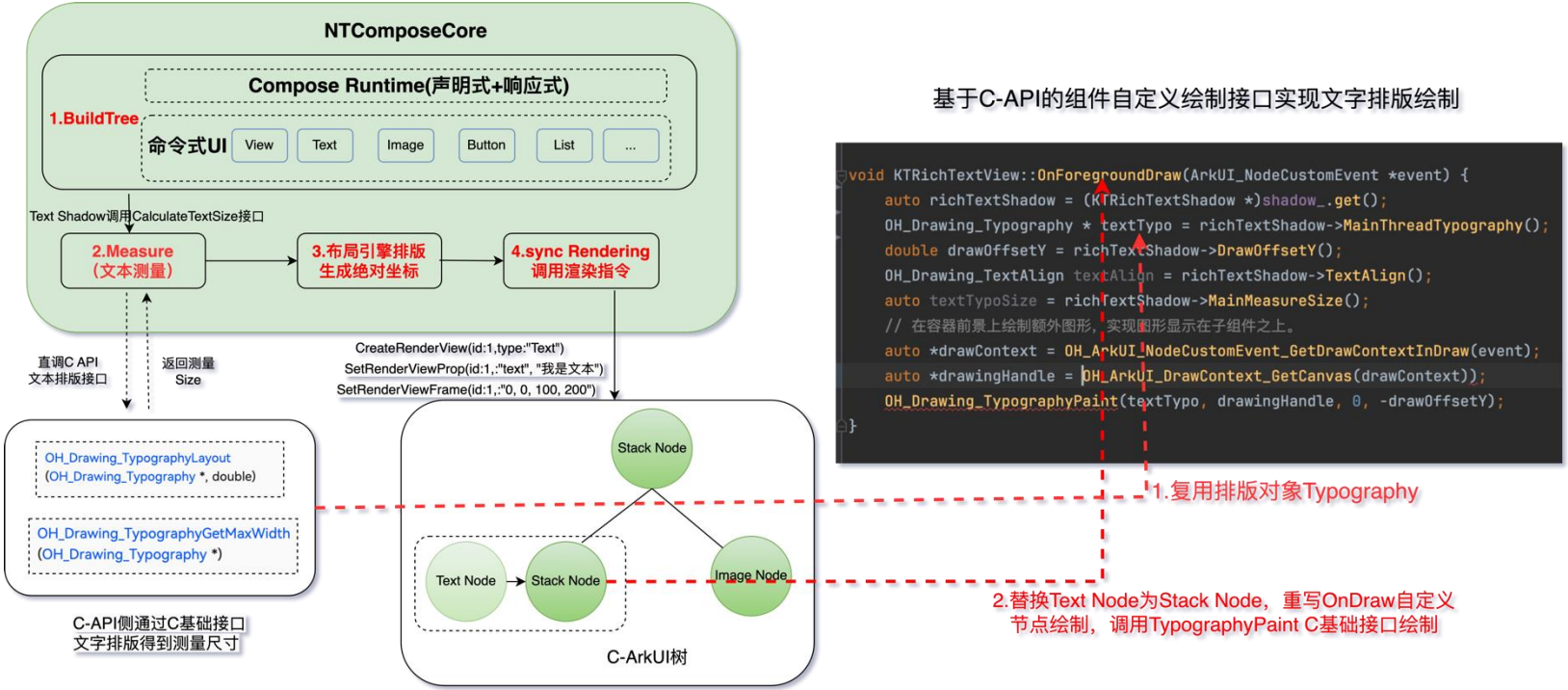
NTCompose鸿蒙Render：文本测量+排版存在二次排版问题

◆ 跨端渲染流程：Build -> Measure -> Layout -> Rendering 来看重复排版问题



NTCompose鸿蒙Render文字排版优化：文字排版对象复用优化

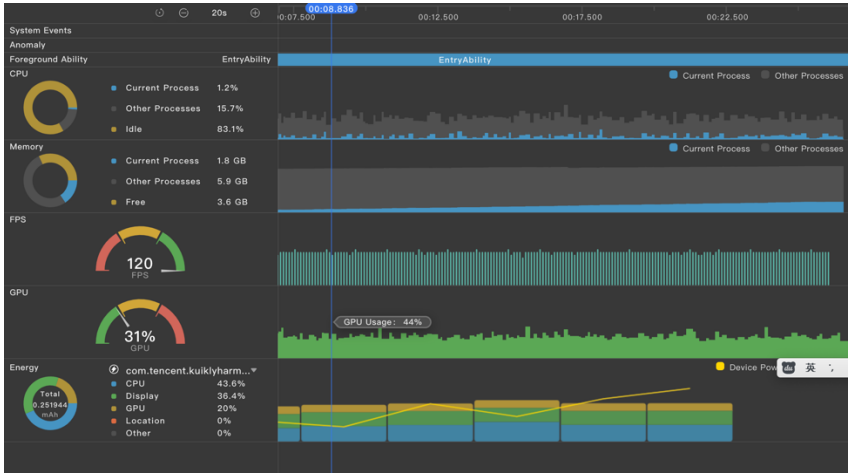
◆ 优化：复用排版结果对象+自定义组件实现排版对象绘制：仅子线程一次排版，主线程不再排版



NTCompose鸿蒙性能数据

◆ **性能数据：**

性能测试项	首屏耗时	FPS流畅度（平均）	内存(初始)	CPU
类RN方案	250+ms	110帧	130mb	10%
NT-Compose	53ms	120帧	114mb	6.5%



高性能适配方案：
1. FPS满帧
2. 比类 RN跨端框架快 5 倍

Thanks! Have a Nice Kotlin

问答环节

Kotlin 中文开发者大会