
Architect
SACC
2022 中国系统架构师大会
SYSTEM ARCHITECT CONFERENCE CHINA 2022
激发架构性能 点亮业务活力
云上会议 网络直播 | 2022年10月27-29日
IT168.com
ChinaUnix.net
ITPUB

高性能 WebF 渲染

关于我

董天成

https://github.com/andycall

- WebF TSC 主席 & 核心开发者
- 3年前端，3年客户端开发经验
- 2016 - 2019 百度
- 2019 - 2022 淘宝

目录

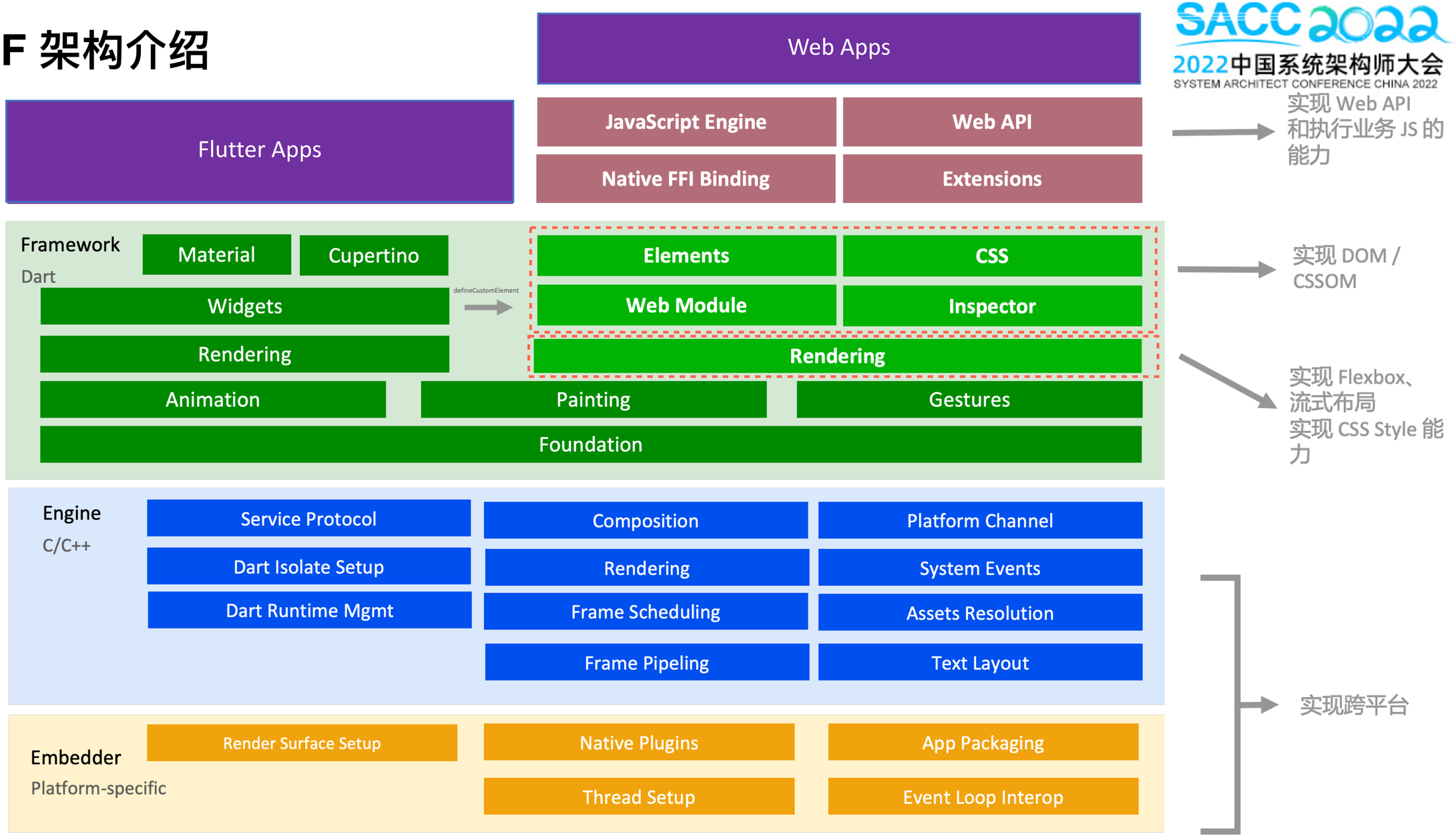
- WebF 项目介绍
- 如何评估一个渲染引擎的性能?
- 一个应用从打开到运行的五个阶段
  - 深入分析 Parse 阶段的工作细节
  - 深入分析 Eval 阶段的工作细节
  - 深入分析 Render 阶段的工作细节
- Q & A

WebF 是什么?

https://github.com/openwebf

- WebF 是一款基于 W3C 标准的高性能渲染引擎
- WebF 渲染内容 1:1 复刻了 DOM，CSS 能力，同时底层基于 Flutter 进行渲染，通过 Flutter 自绘的特性，保证多端一致性。

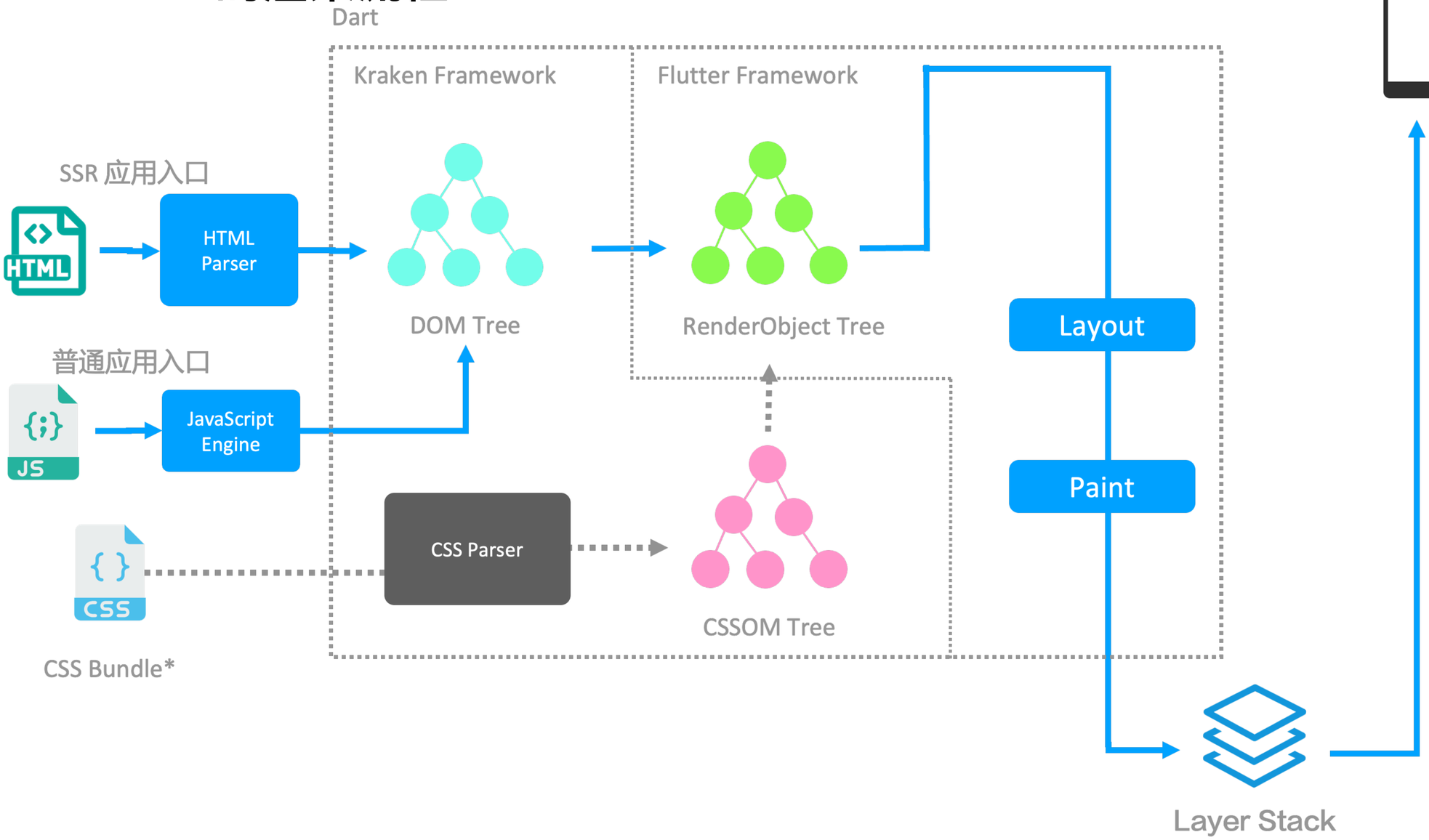
WebF 架构介绍



Features

- ✓ Web 标准&前端生态
- ✓ 动态性
- ✓ 跨端一致性
- ✓ Flutter 原生性能&体验
- ✓ 灵活的调试工具
- ✓ Flutter 生态

WebF 的渲染流程



如何评估一个渲染引擎的性能

加载

首屏启动时间

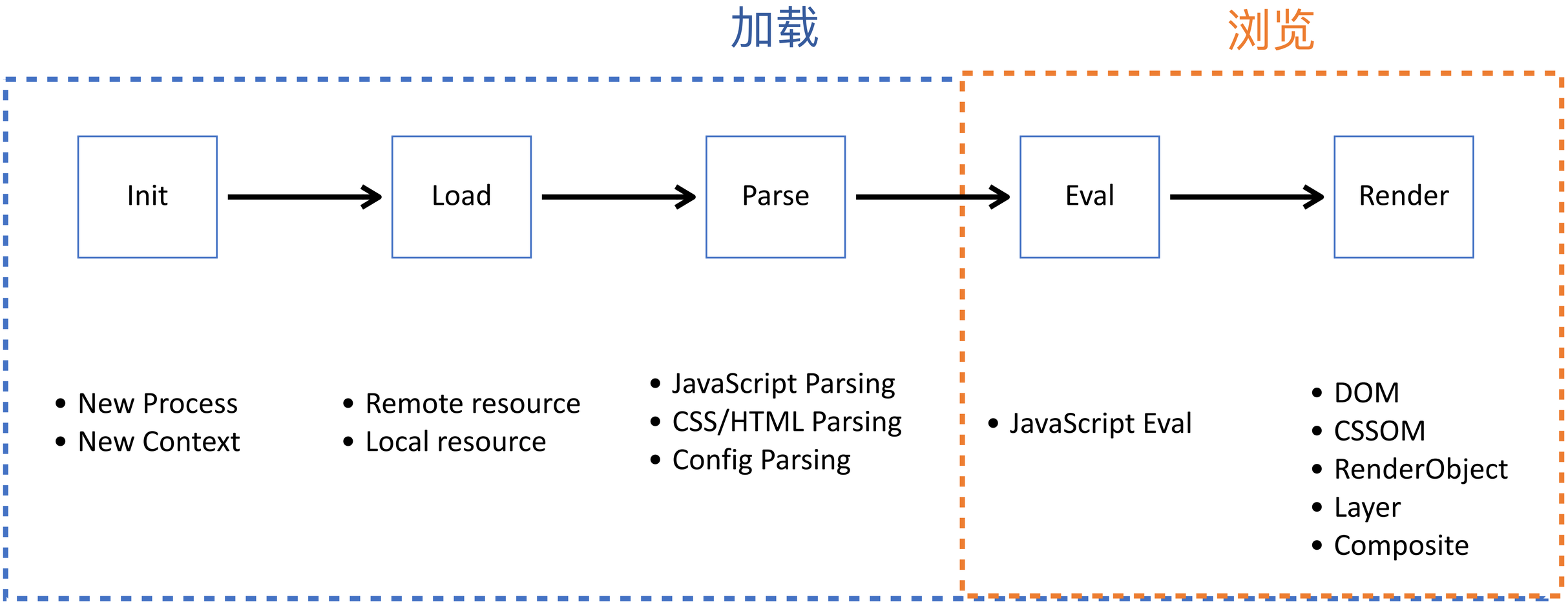
+

浏览

帧率

内存占用

一个页面从打开到运行的五个阶段



How Parse Works

JavaScript

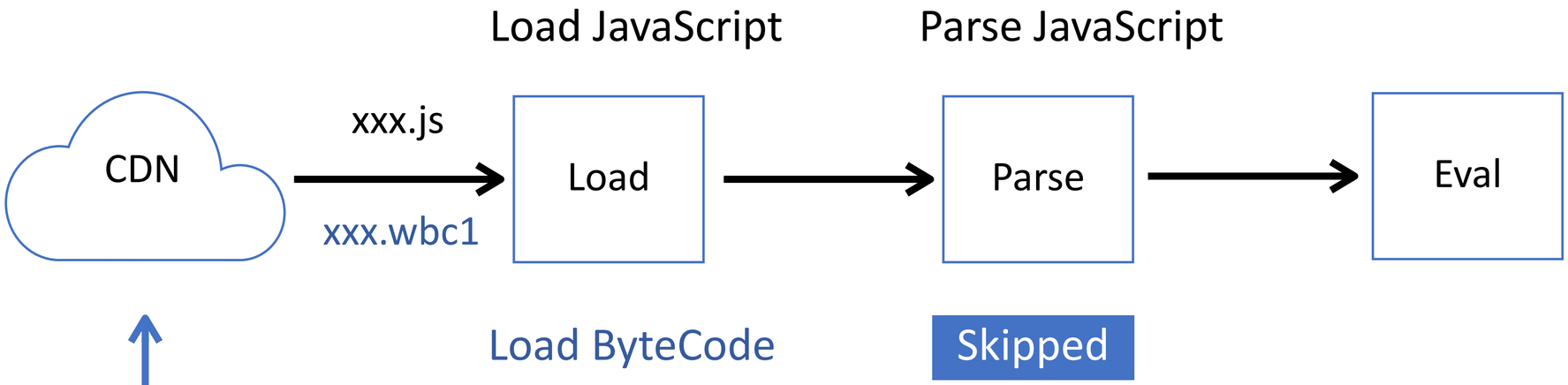
HTML/CSS

Bytecode (QuickJS)

Data Structure (HTML,CSS)

Other format…

如何优化 Parse 阶段的耗时?



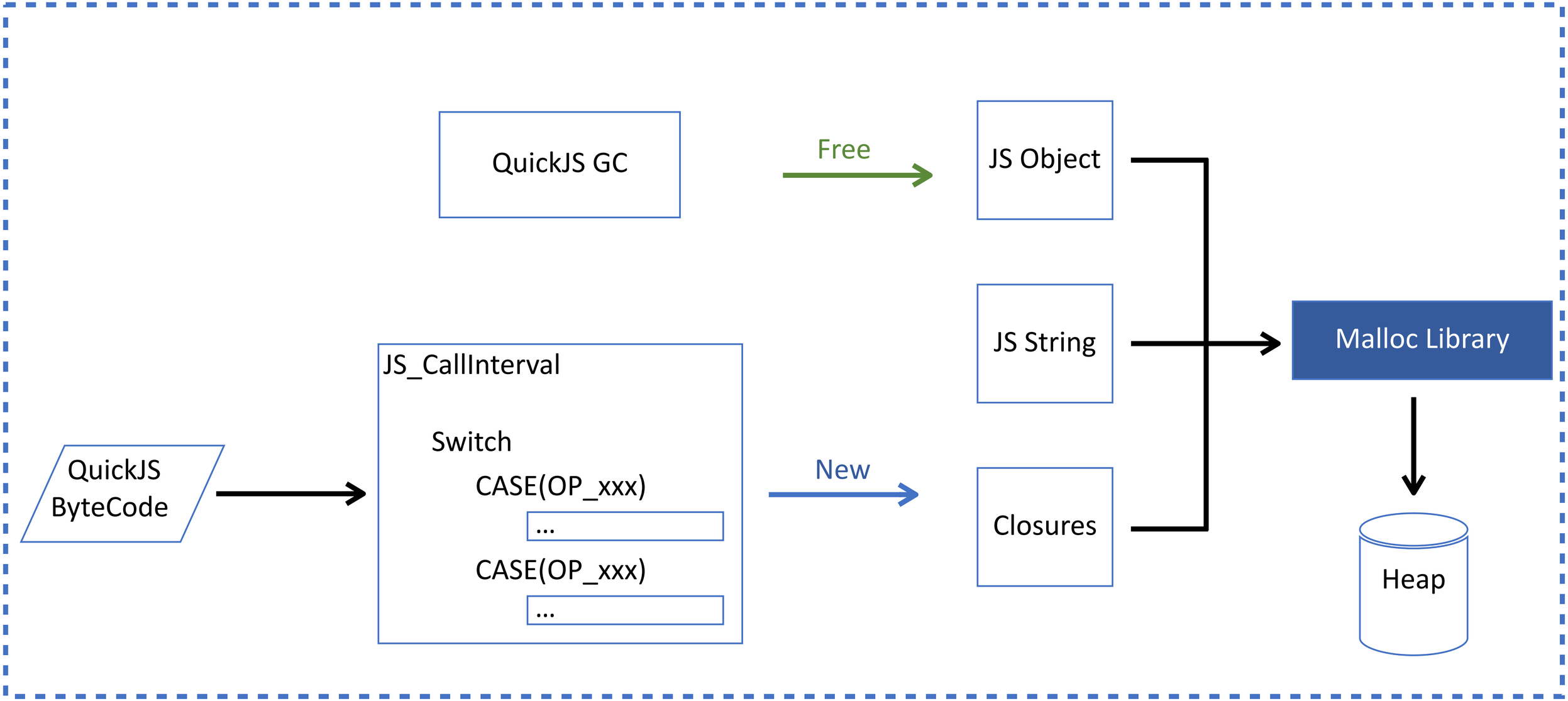

```
var a = 1 + 1;
```

JavaScript Code

```
op_push_1 1
op_push_1 1
op_add
```

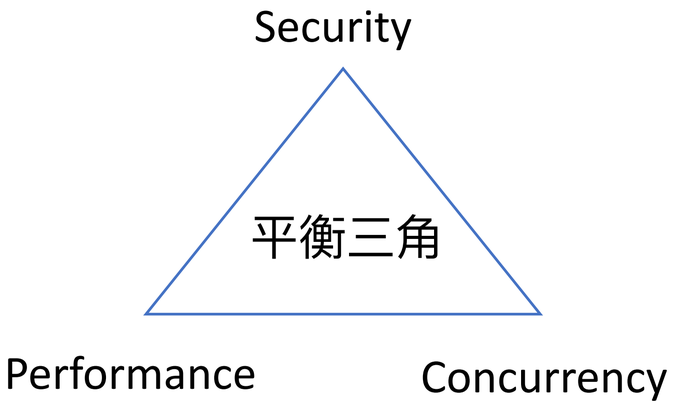
QuickJS ByteCode

How QuickJS Eval Works



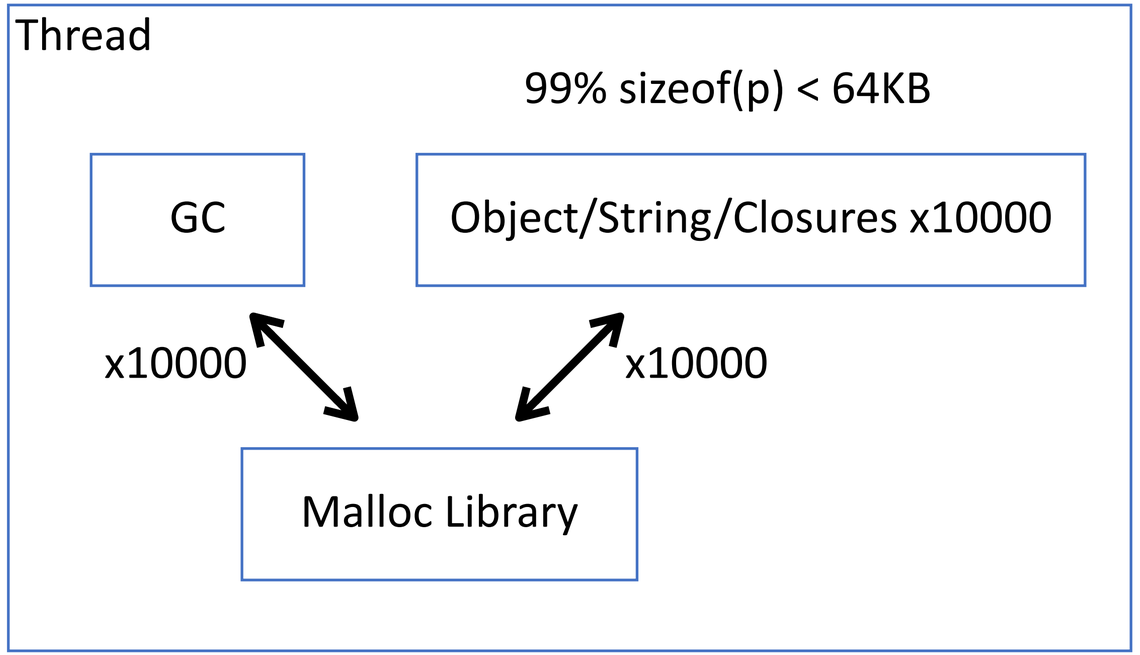
选用正确的 Malloc Library 有助于提升性能

软件设计没有银弹



- 系统内置的 mallc/free 包含线程锁
- 大量的小对象会导致堆碎片化

QuickJS 的 malloc 场景



- 大量而频繁的 malloc/free 操作
- 都是小对象内存分配
- 单个线程，没有多线程场景

使用 mimalloc 是一个不错的选择

https://github.com/microsoft/mimalloc

微软专门为 Koka and Lean 语言而设计的 Malloc Library

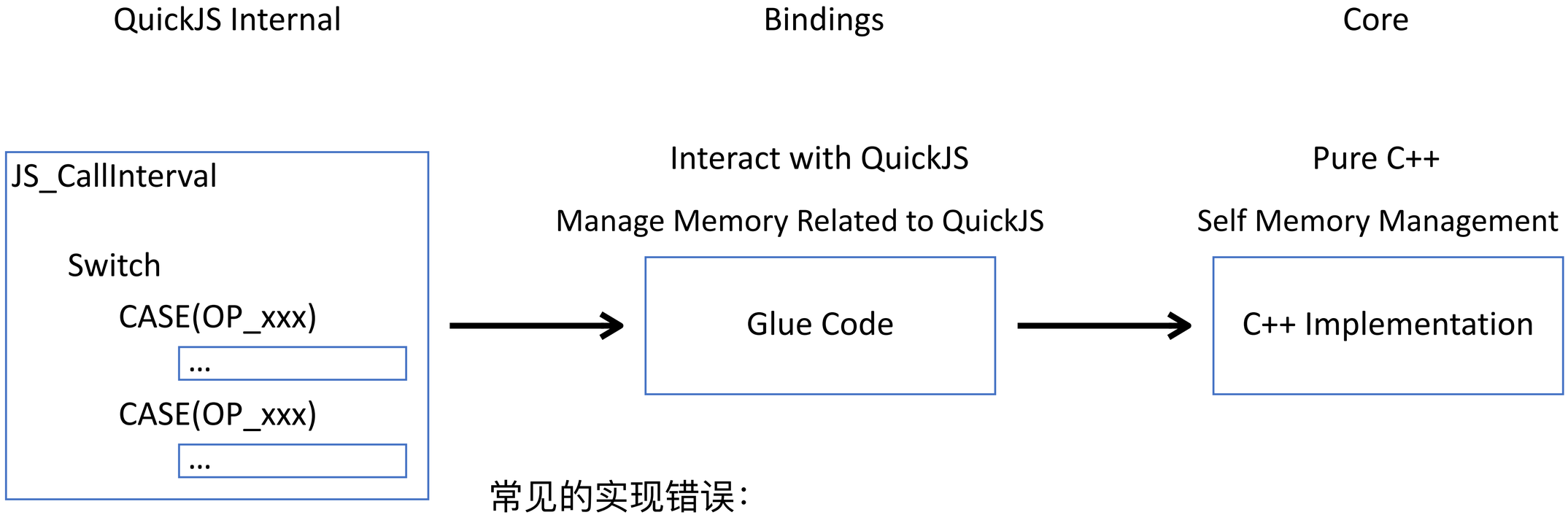
QuickJS 的 malloc 场景

- 大量而频繁的 malloc/free 操作 ✓
- 都是小对象内存分配 ✓
- 单个线程，没有多线程场景 ✓

设计理念：

- 每个内存段内都有多个小的链表
  - 优势：减少小对象导致的内存碎片
- 每个线程都有独立的内存段和链表
  - 优势：单个线程内的内存操作无需考虑锁带来的性能损耗

JS Context 中有大量 C++ 实现的 Binding API



常见的实现错误：

Interact with QuickJS

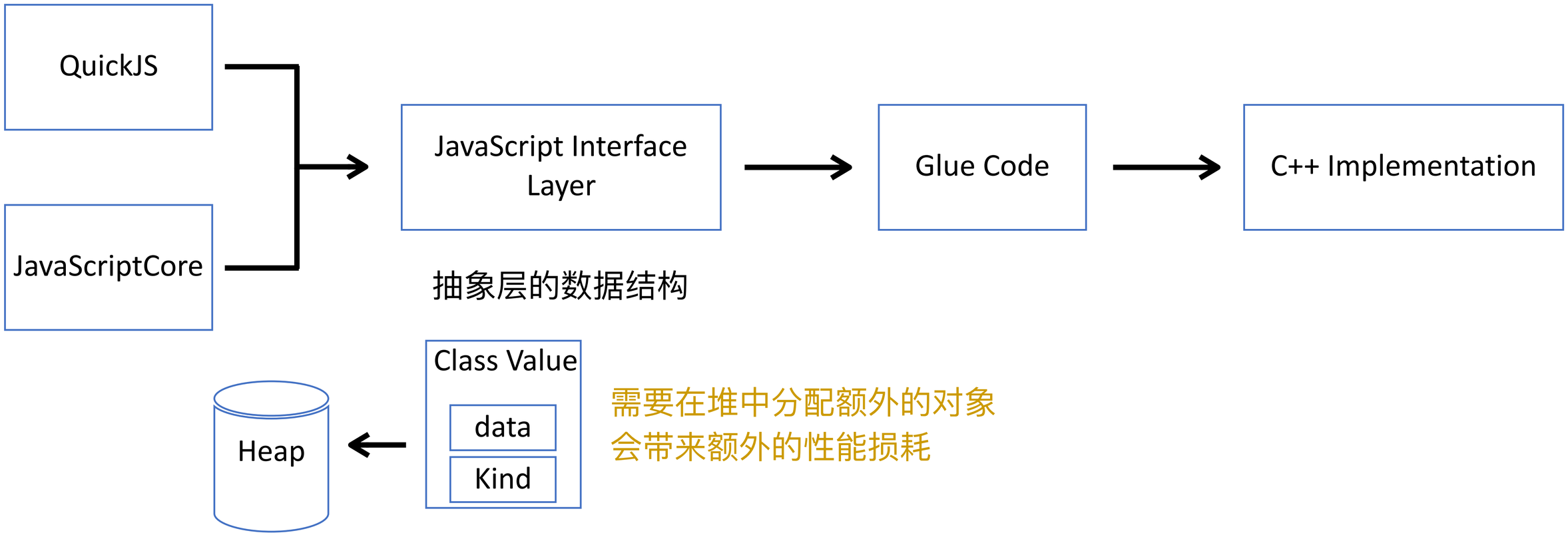
Use JavaScript Engine API In C++ ≈≈ Use JavaScript Directly

Interact with QuickJS

Binding 通常会有跨引擎的需求

为了节省支持多个引擎所需要多次实现 Glue Code 的时间

常见的做法是加一个抽象层，然后上层代码和抽象层进行交互



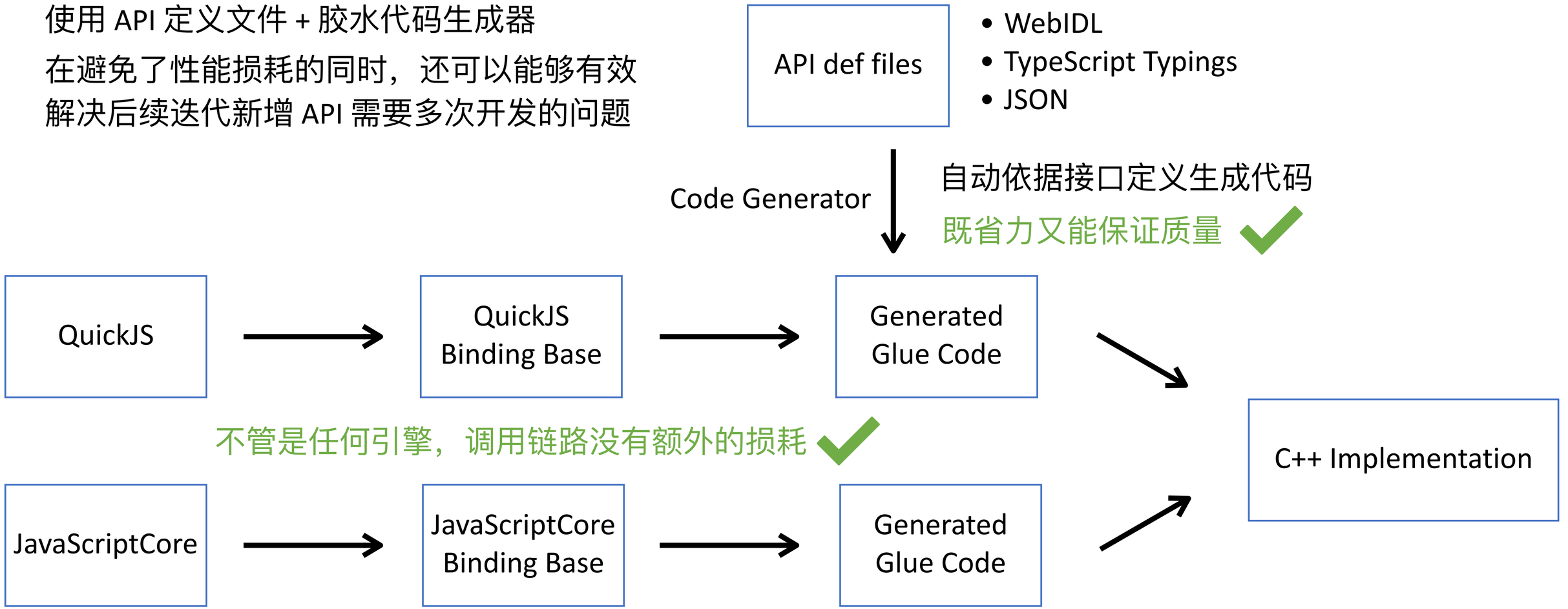
另辟蹊径 —— 代码生成器的妙用

使用 API 定义文件 + 胶水代码生成器

在避免了性能损耗的同时，还可以能够有效解决后续迭代新增 API 需要多次开发的问题

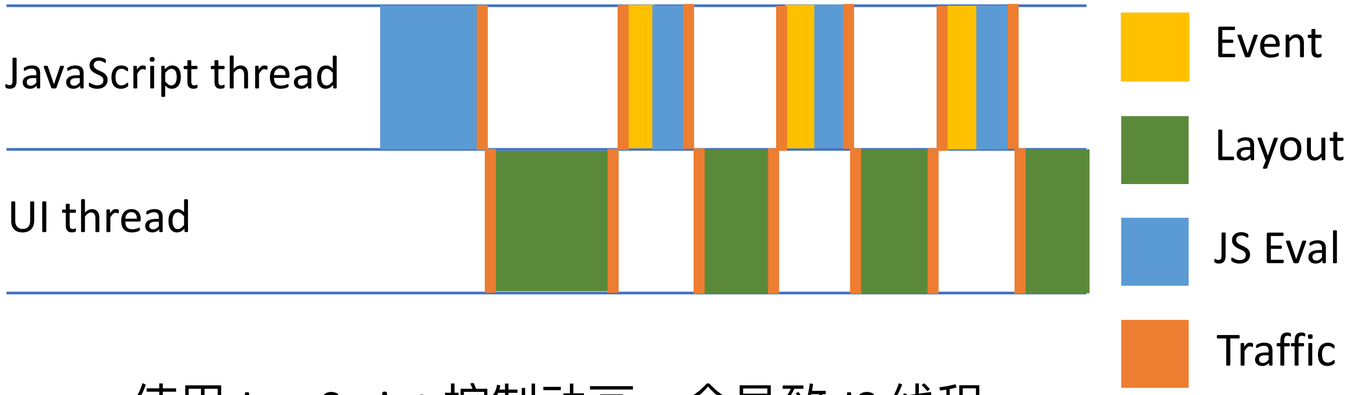
任意支持描述接口的语言

- WebIDL
- TypeScript Typings
- JSON

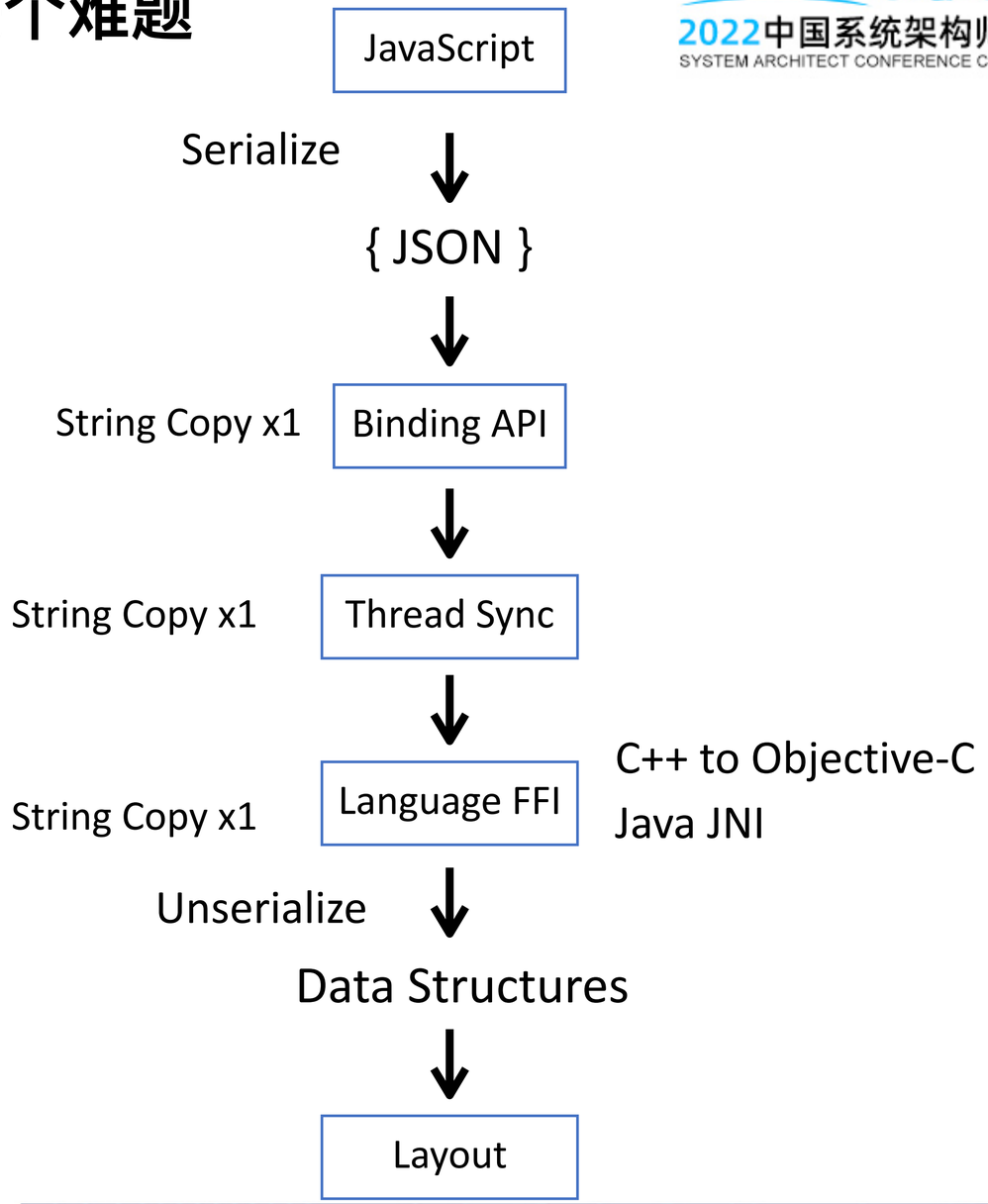


在跨端引擎上使用 JS 自定义动画的性能是个难题

很多跨端方案会把 JS Engine 作为一个独立线程



- 使用 JavaScript 控制动画，会导致 JS 线程和 UI 线程频繁通信
- JSON 并不是一种高效的通信格式
- 一份指令至少复制三次才能传递到 Layout



JSON 并不是一种高性能的通信格式

JSON 的 parser 要做的事情很多

对于渲染引擎来说，传输的渲染指令并不需要特别复杂的数据结构，因此使用 JSON 只会大材小用

```
{"data": ["value1", "value2", "value3"]}
```

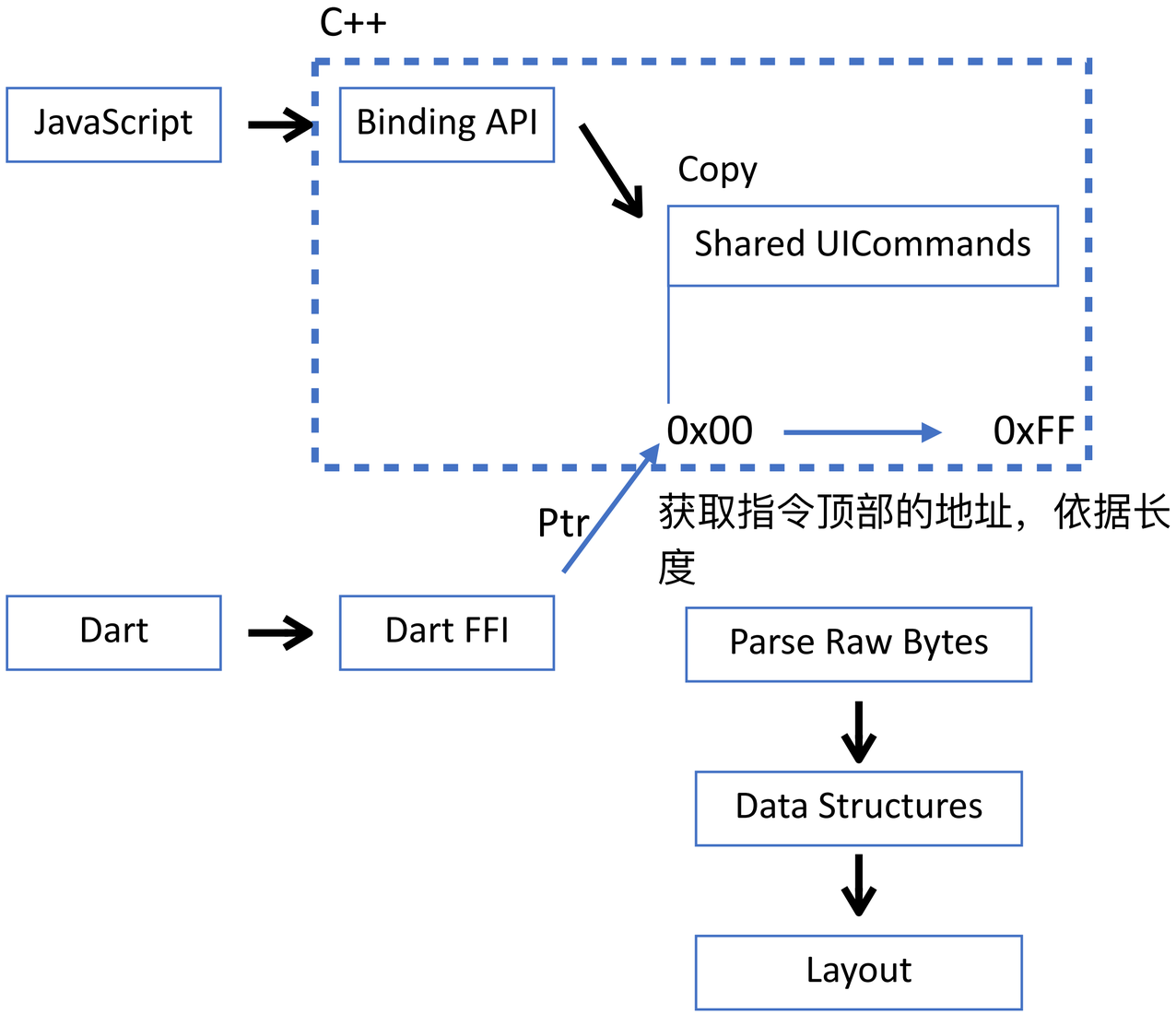
- 语法检查
- 字符串编码
- 类型识别

设计一个更加简单的通信数据格式

UICommand

- 无需语法检查，直接解析
- 二进制格式，无需编解码
- 类型预定义，不需要额外判断

利用指针寻址，尽可能减少数据复制



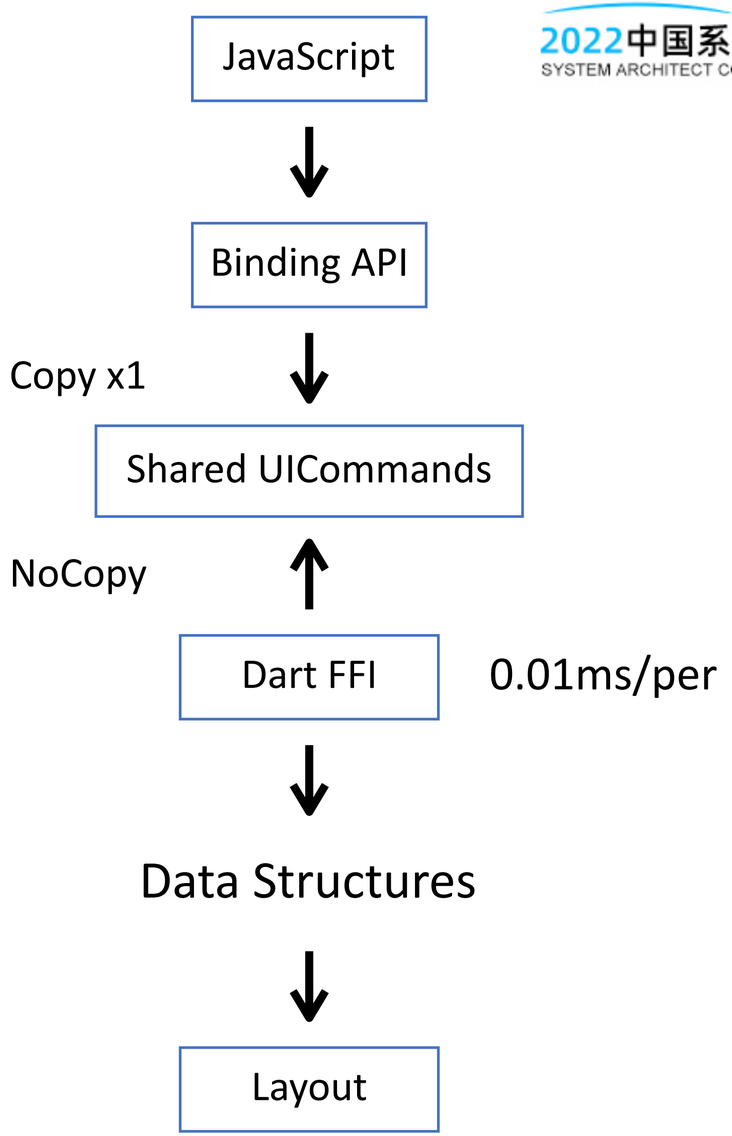
数据只需复制一次即可到达 Layout

同一个线程是高性能通信的前提

和浏览器的线程模型保持一致

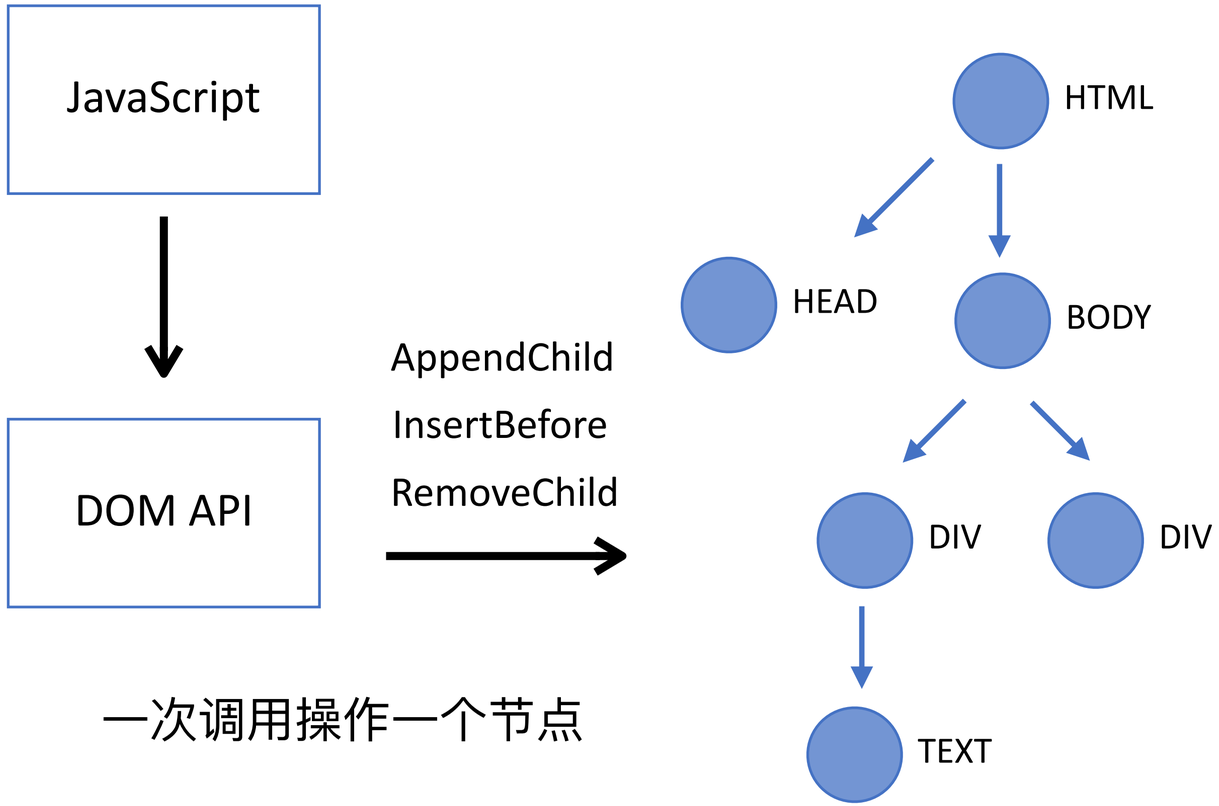
无线程等待

定制的数据格式提升解析效率



如何才能更快的生成 DOM 树

前端应用中的 DOM 树通过 JavaScript API 来生成



一次调用操作一个节点

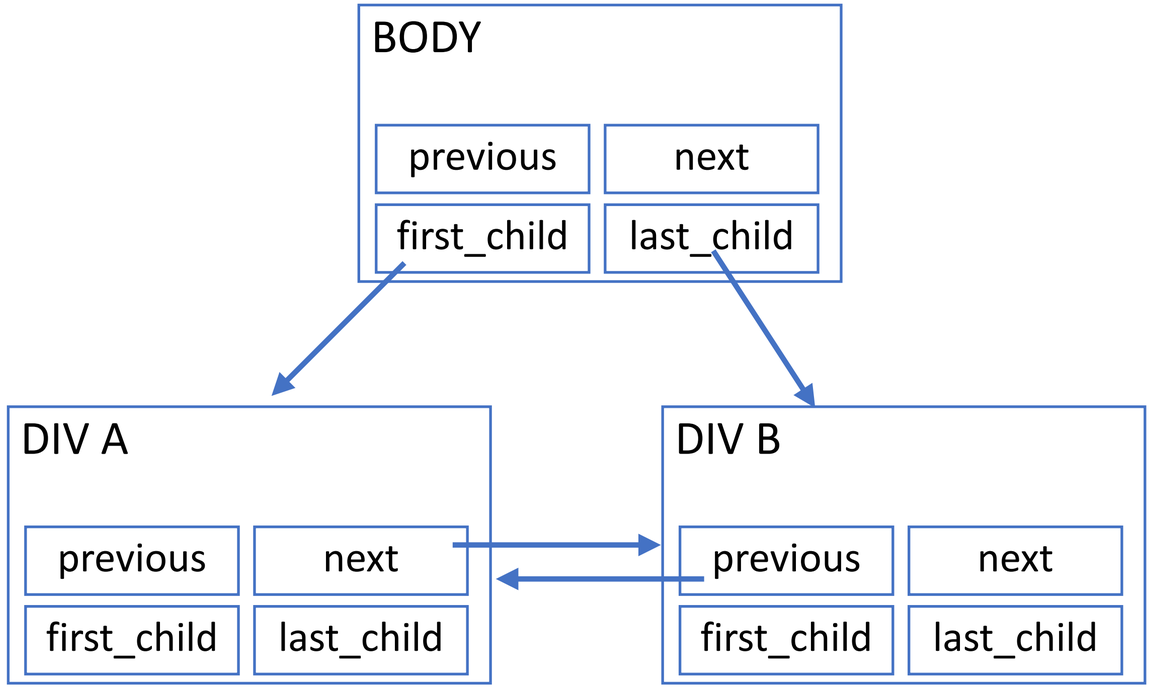
1000 个DOM 节点，DOM API 的调用次数 >= 1000

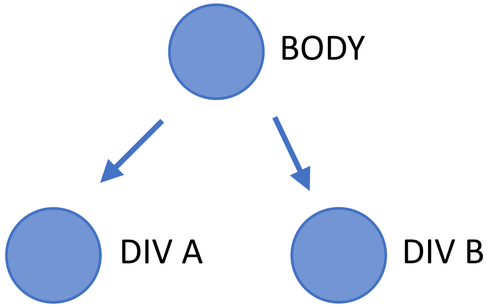
必须要确保 API 的时间复杂度为

$$O(1) <= T(n) <= O(log(n))$$

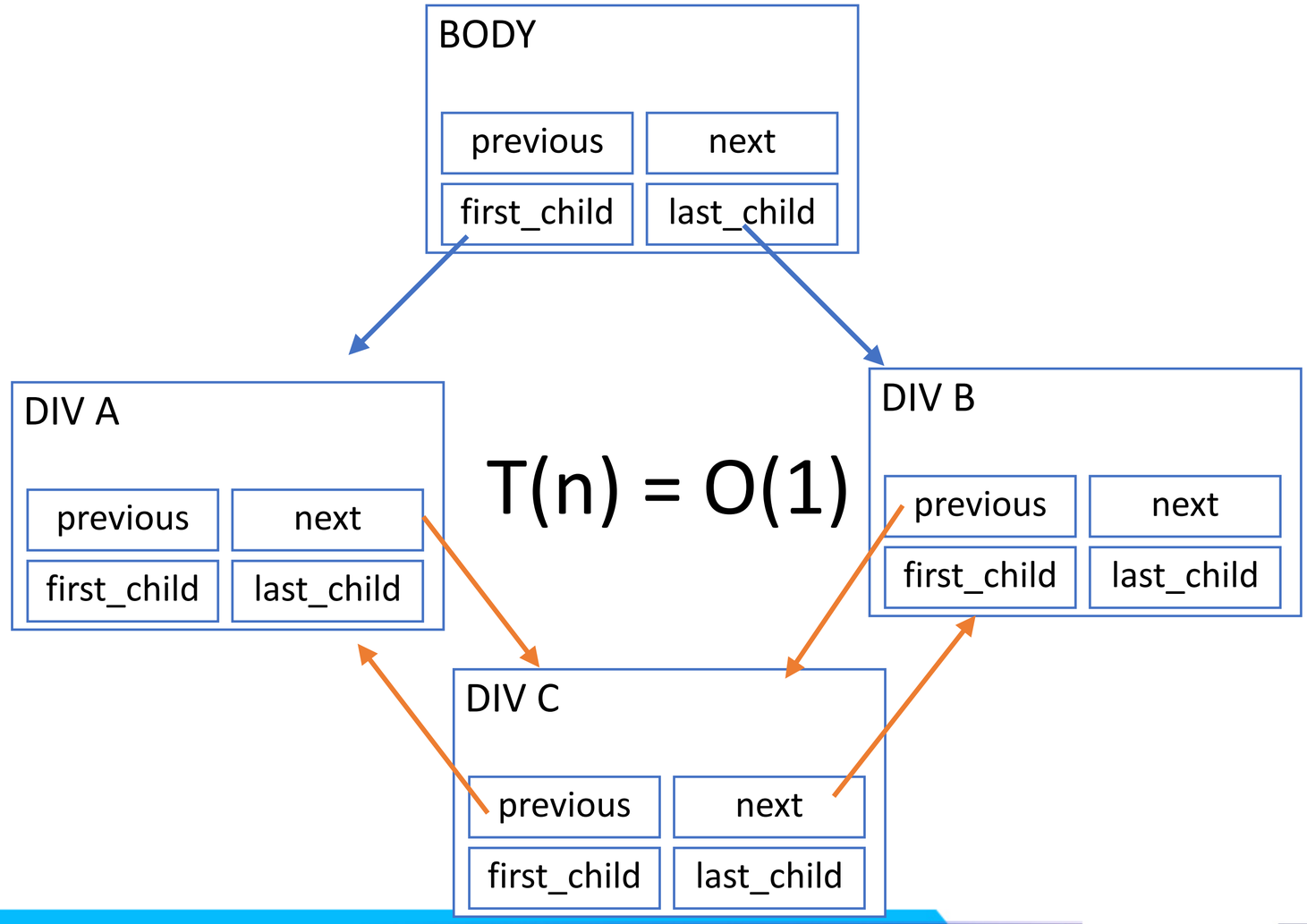
才能确保 DOM 树可以高效生成

使用双向链表来实现 DOM 树

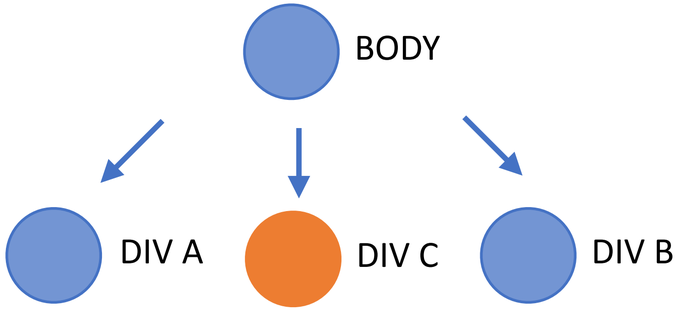




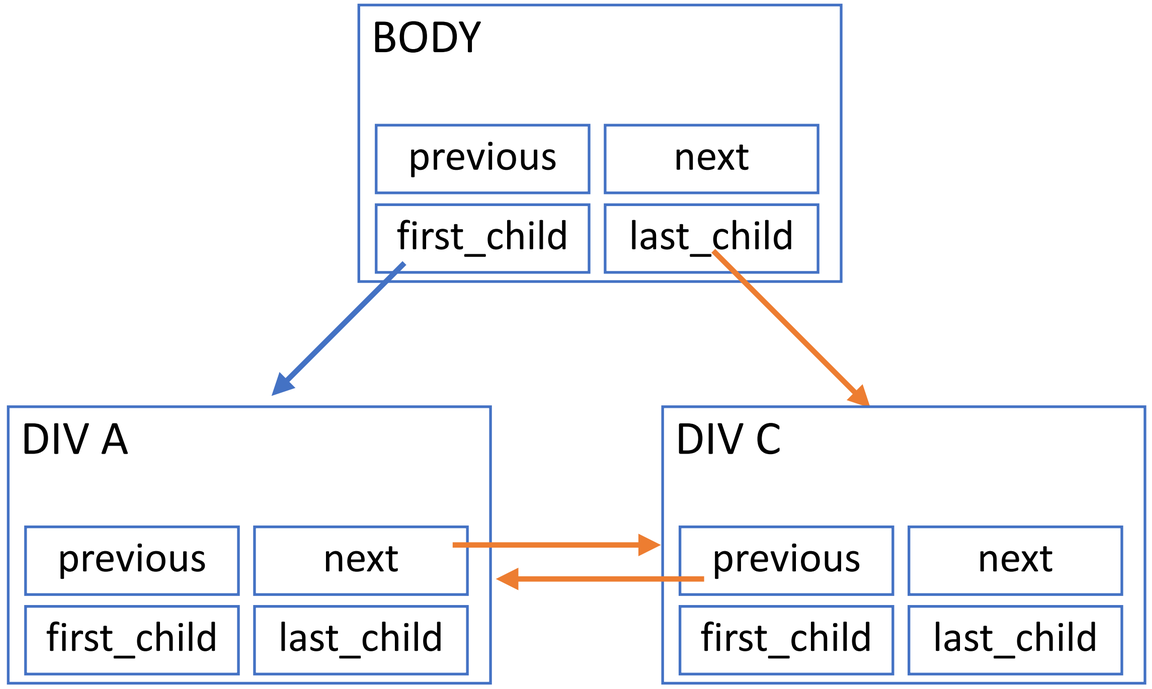
使用双向链表来实现 DOM 树



Body.insertBefore(C, B)

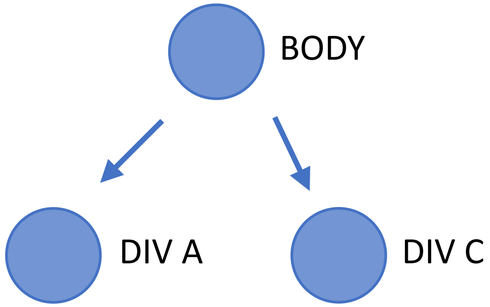


使用双向链表来实现 DOM 树



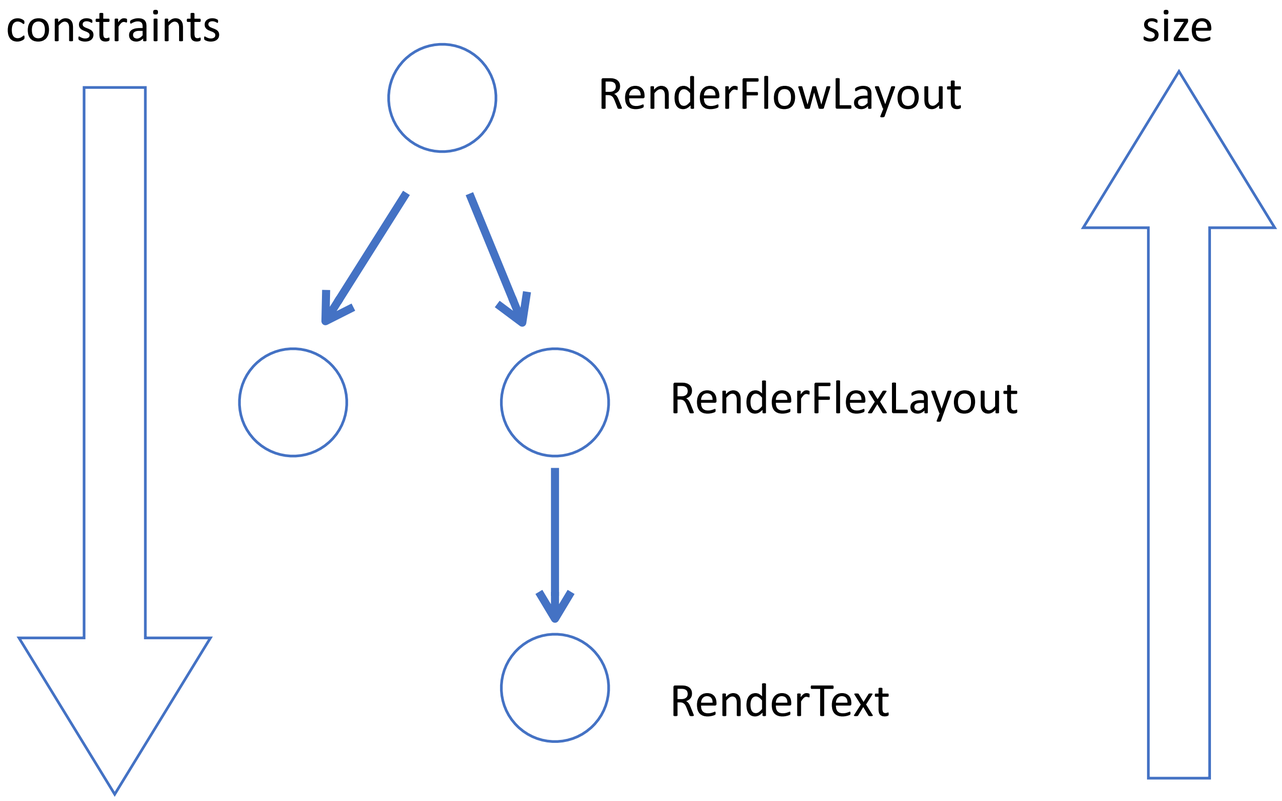
$T(n) = O(1)$

Body.removeChild(B)



Flutter Layout 的工作过程

Layout 的主要目的是确定每个 RenderObject 的尺寸和位置

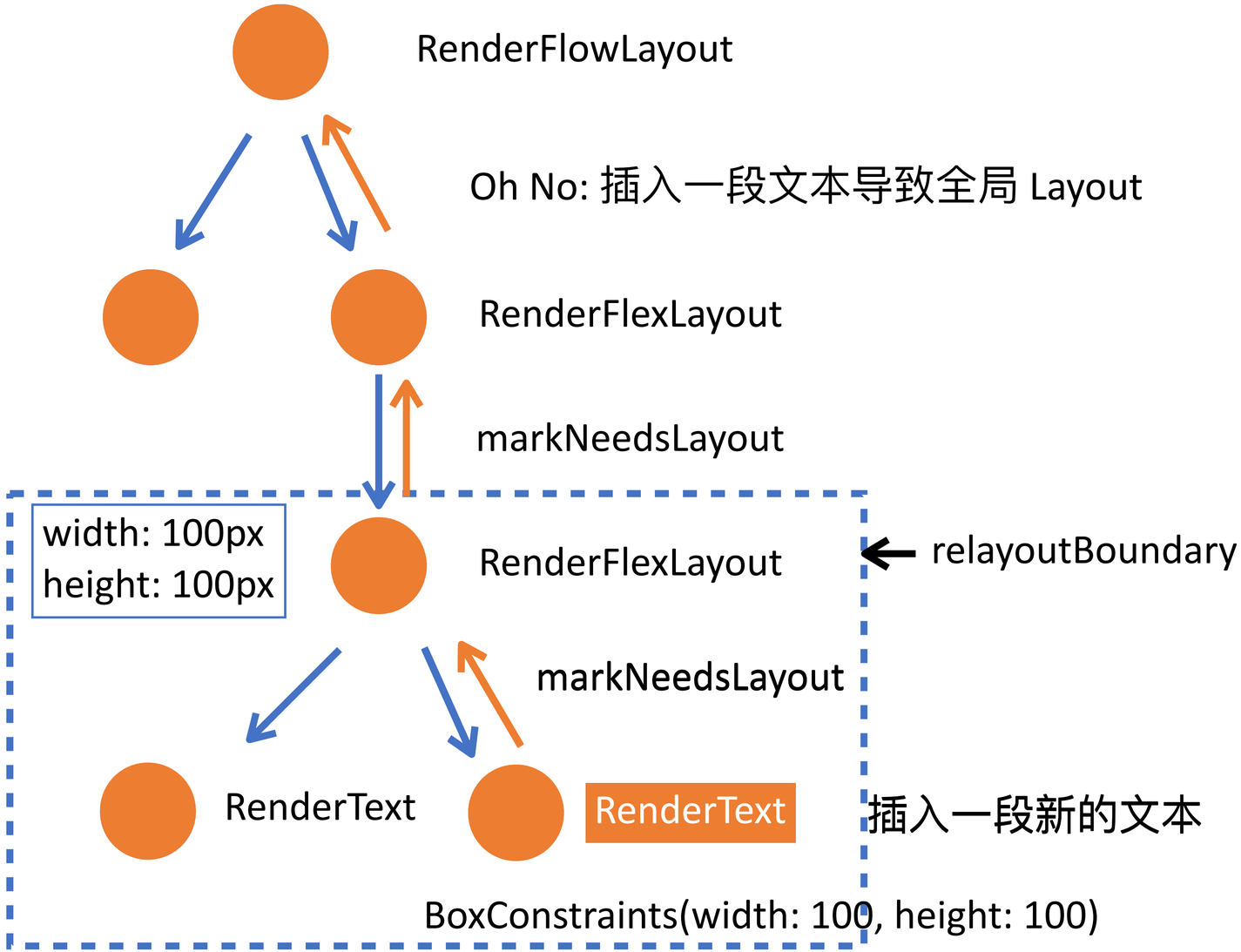


1. Parent 通过自身的尺寸条件（通过 style 决定），计算出约束（constraints），然后传递给 Child 进行布局。

2. Parent 根据所有 child 的 size 以及自身的布局规则来确认自身的 size（Flex 布局还需要根据 child 的尺寸进行二次调整，因此还需要重复步骤 1）。

3. 确认 Parent.Parent 的尺寸，持续递归直至整棵树布局完毕。

Flutter Layout 更新优化策略



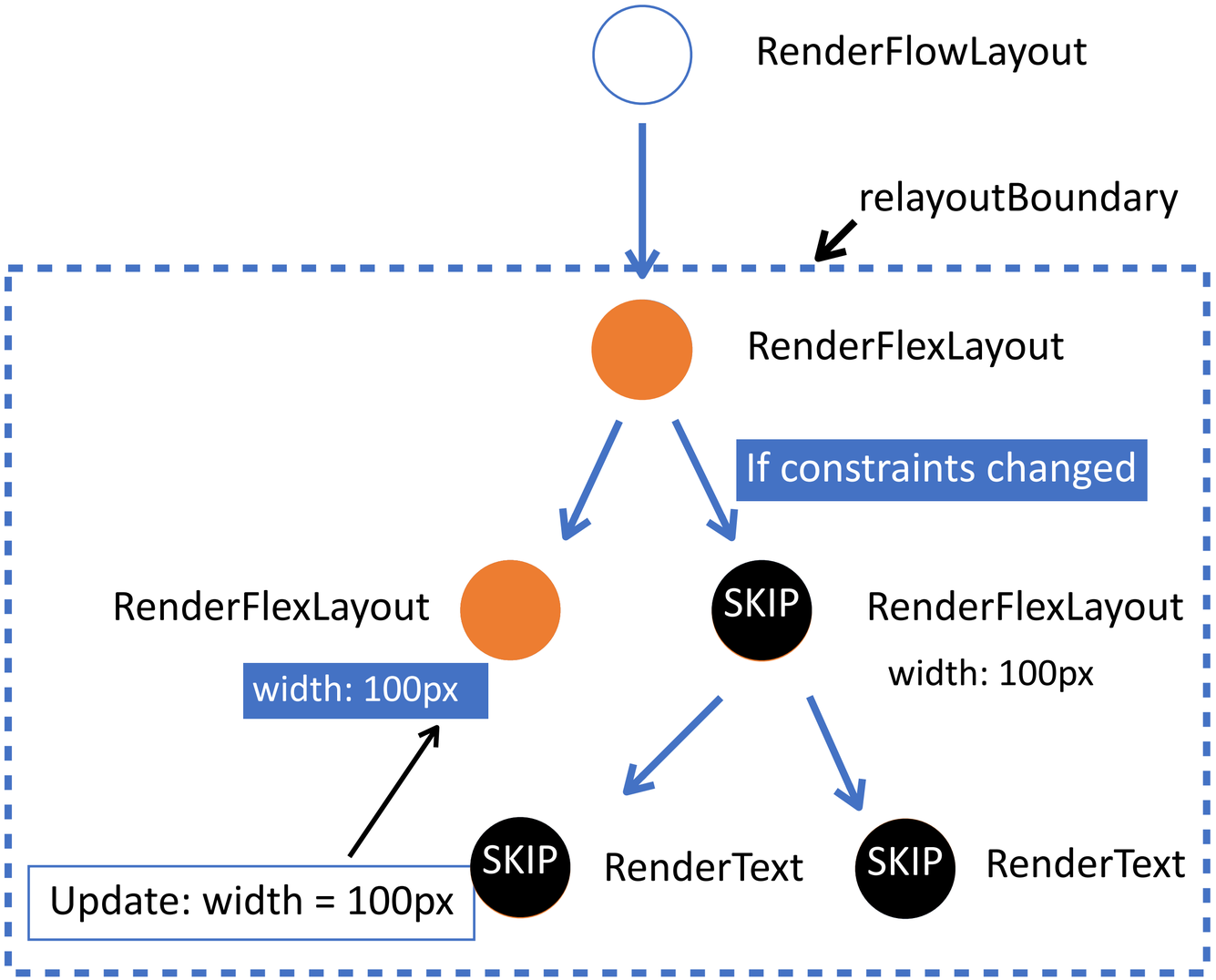
为了避免这种情况，Flutter 引入了 relayoutBoundary 优化策略。
通过避免向父级进行 markNeedsLayout，以免影响全局

relayoutBoundary 的生成条件有哪些?

1. 给元素设置固定尺寸
2. 通过 CSS `contain: layout` 来设置

WebF Layout 更新策略

在 Flutter 的基础之上，WebF 还有一些额外的优化策略



当同级的尺寸发生变化，如何避免相邻元素进行 Layout？

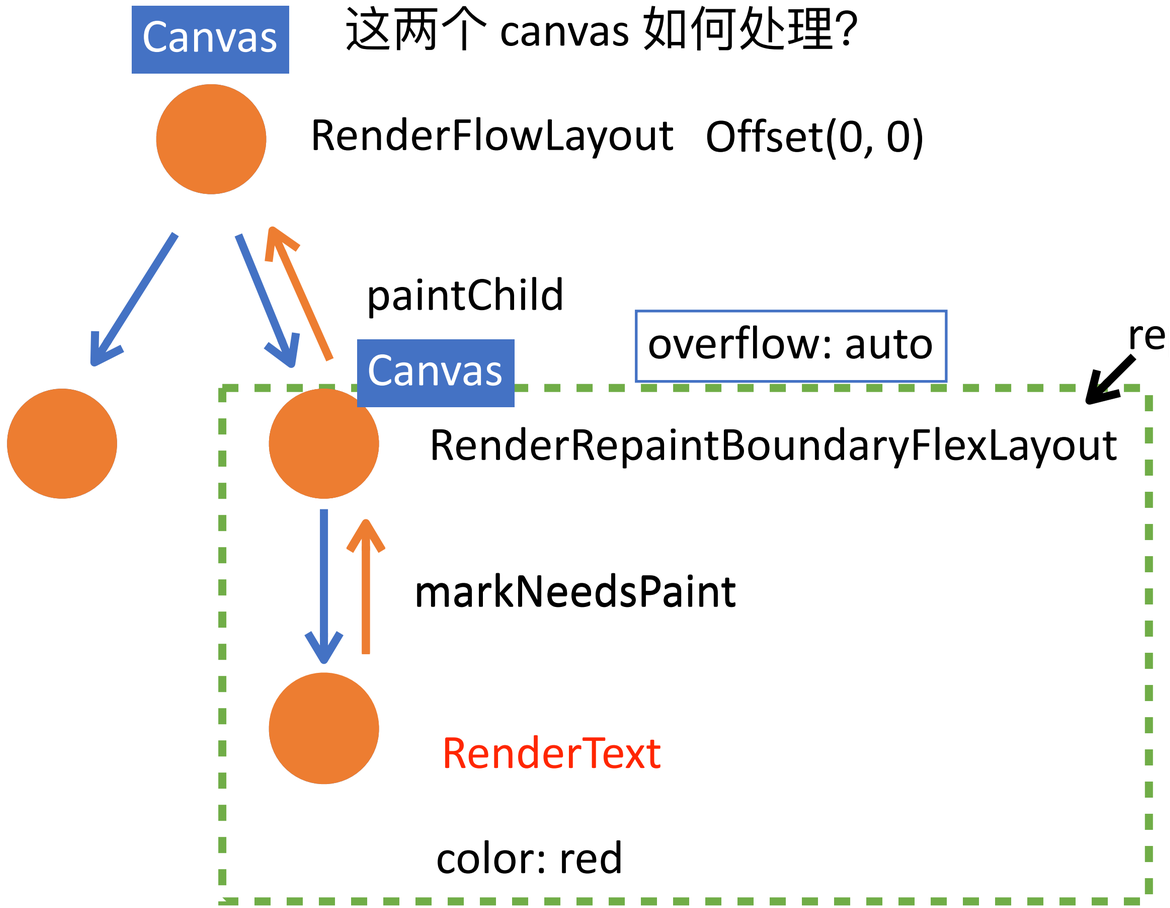
二次 layout 时，通过判断推导出的 constrains 是否变化来判断子级元素的尺寸是否会发生变化，进而取消后续无用的计算。

constraints 的推导条件有哪些?

1. 依据盒模型中定义的尺寸进行推导
2. 通过 CSS `contain: size` 来设置

Flutter Paint 工作过程和优化策略

先创建一个 canvas 画布，然后从顶部递归依据 Layout 生成的坐标为基准，再调用 canvas API 进行绘制

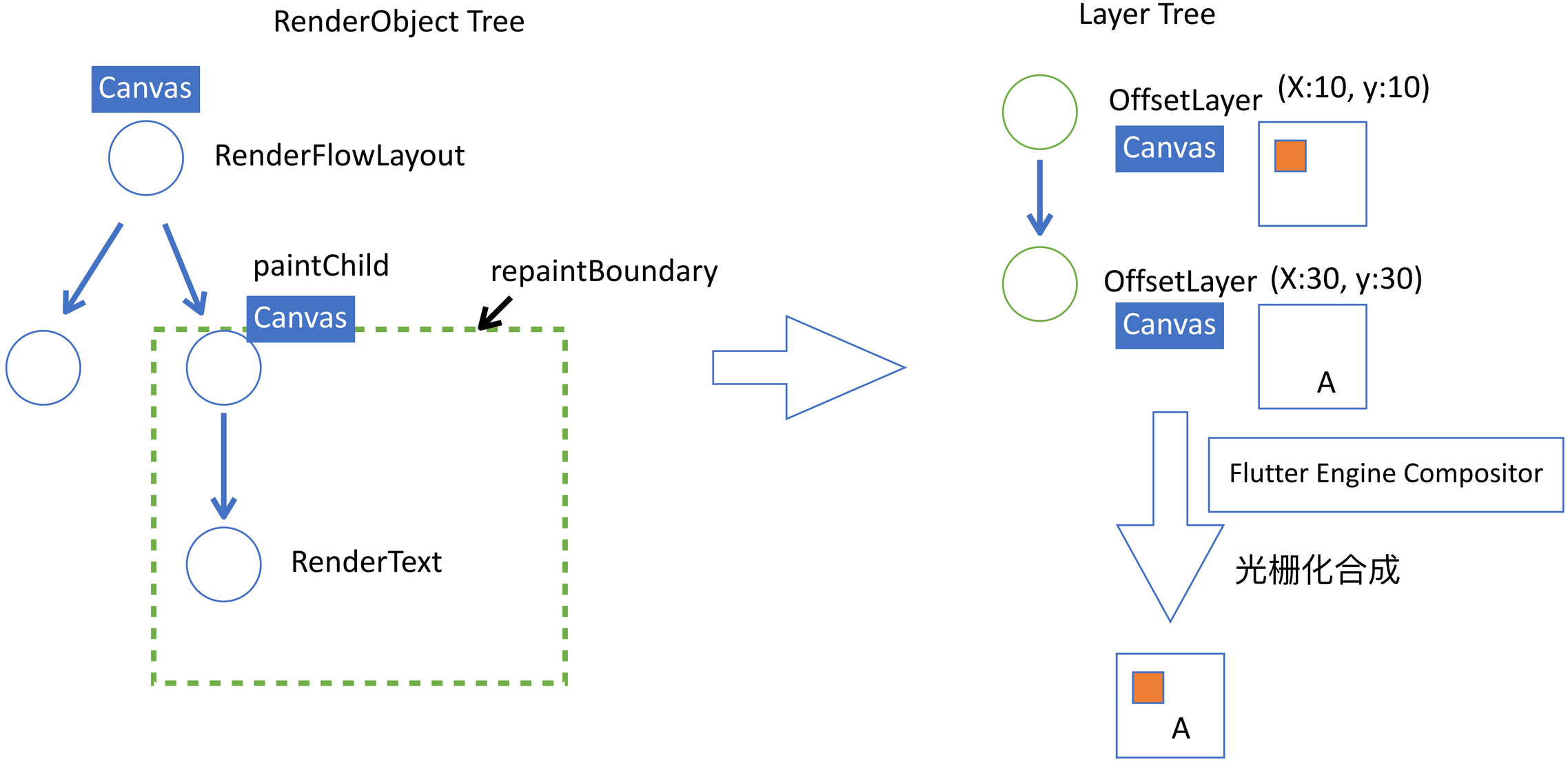


为了避免这种情况，Flutter 引入了 repaintBoundary 优化策略。
通过避免向父级进行 markNeedsPaint，以免影响全局
同时还可以防止外层向内调用 paintChild

repaintBoundary 的生成条件有哪些?

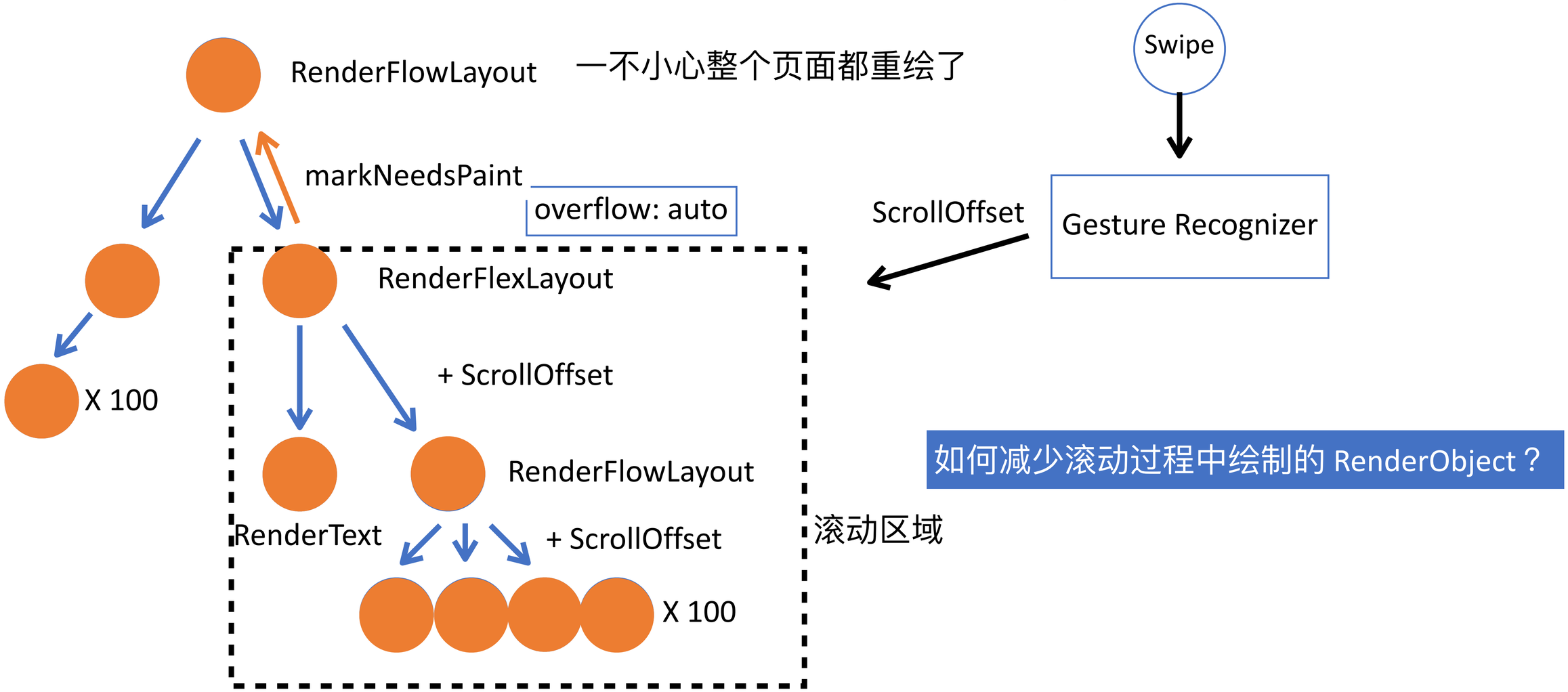
1. overflow: auto
2. display: silver
3. transform
4. position: fixed

Flutter 中的渲染图层 (Layers)



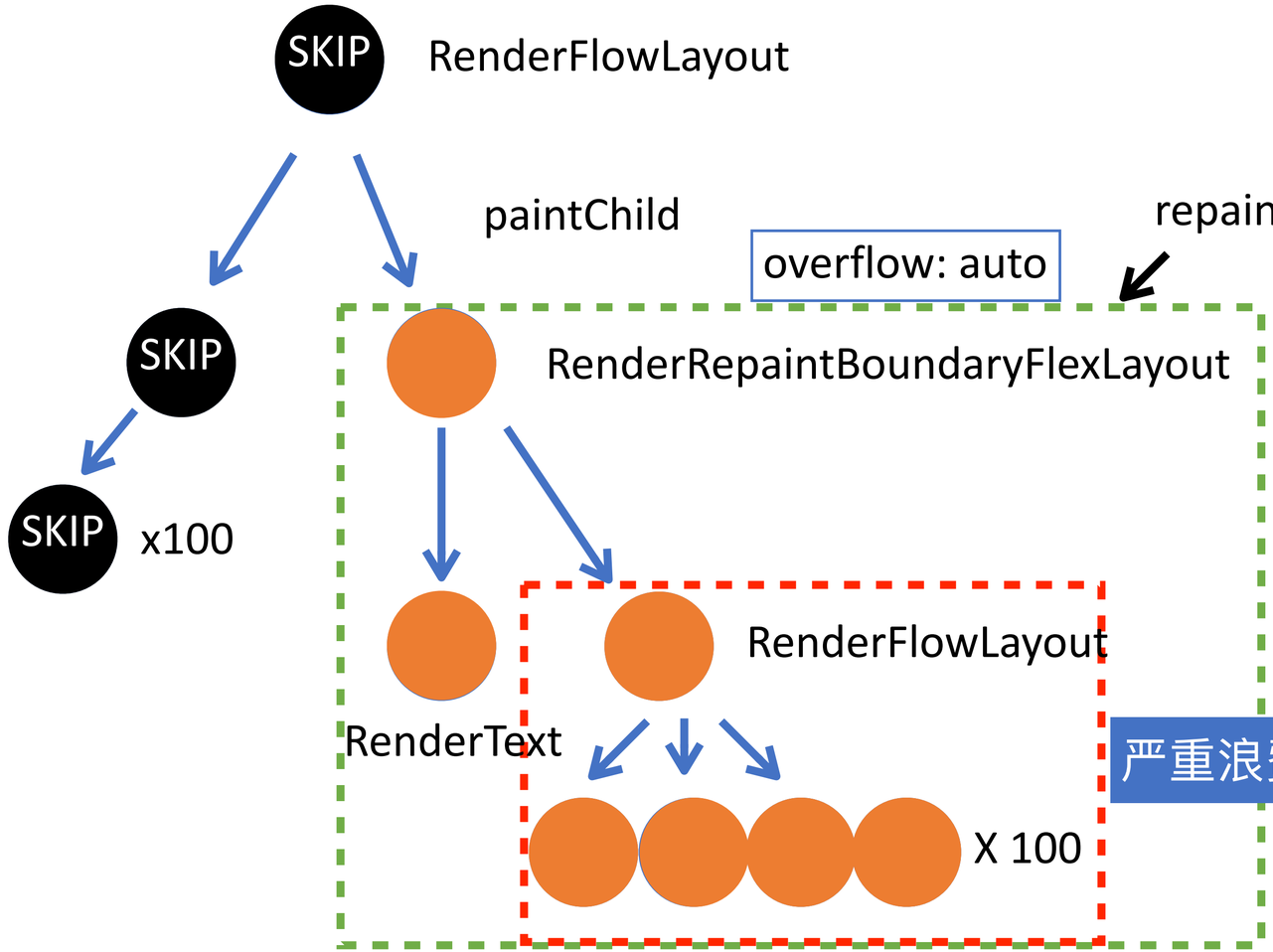
页面滚动的基本原理

滚动的基本原理是通过监听手势的偏移量，来移动一组 RenderObject 在页面中的位置



如何减少滚动过程中绘制的 RenderObject？

如何实现高性能滚动



添加一层 repaintBoundary 只能省掉外层的绘制

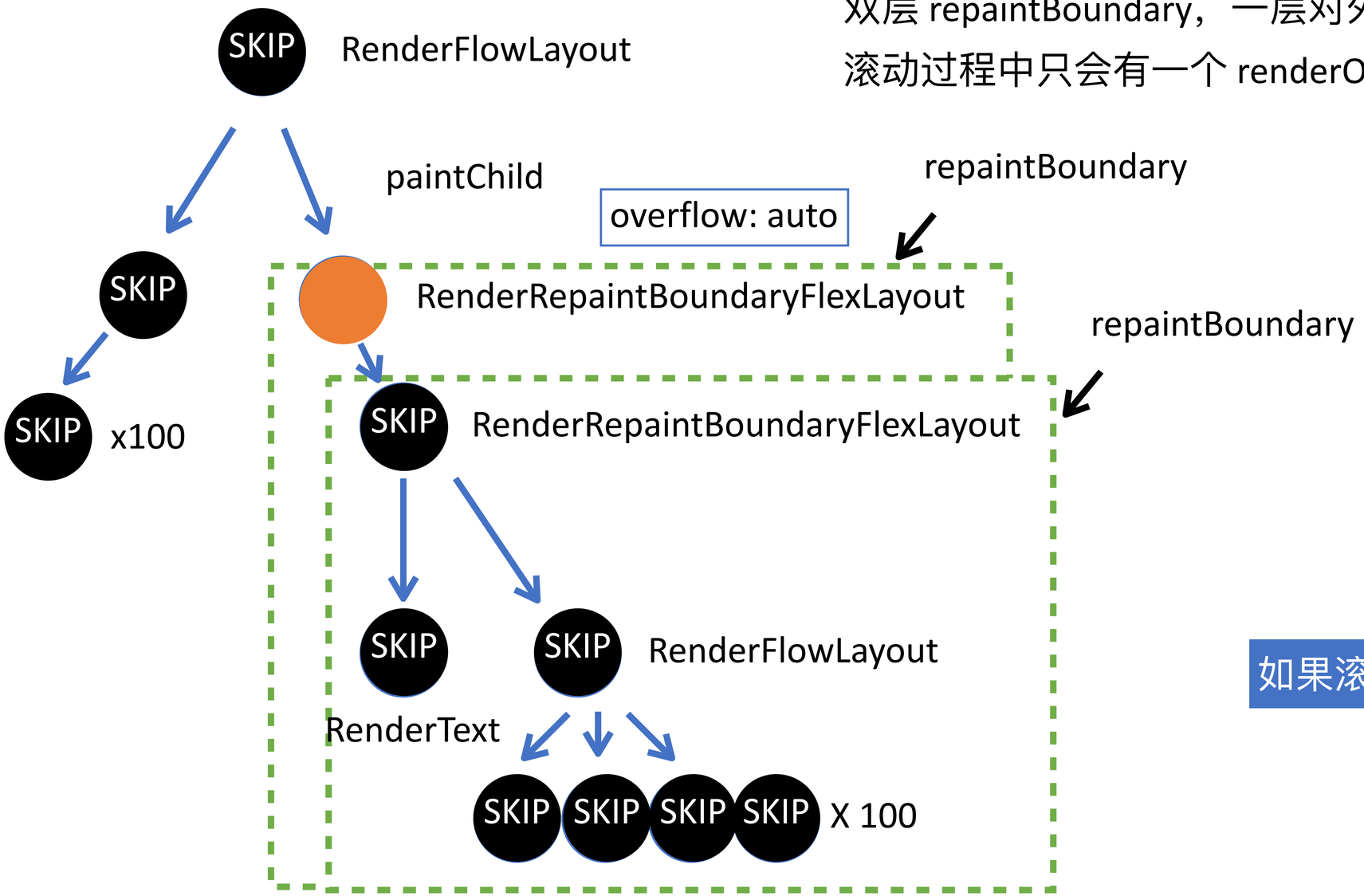
内层的 RenderObject 如何避免调用?

严重浪费计算资源

如何实现高性能滚动

双层 repaintBoundary，一层对外，一层对内

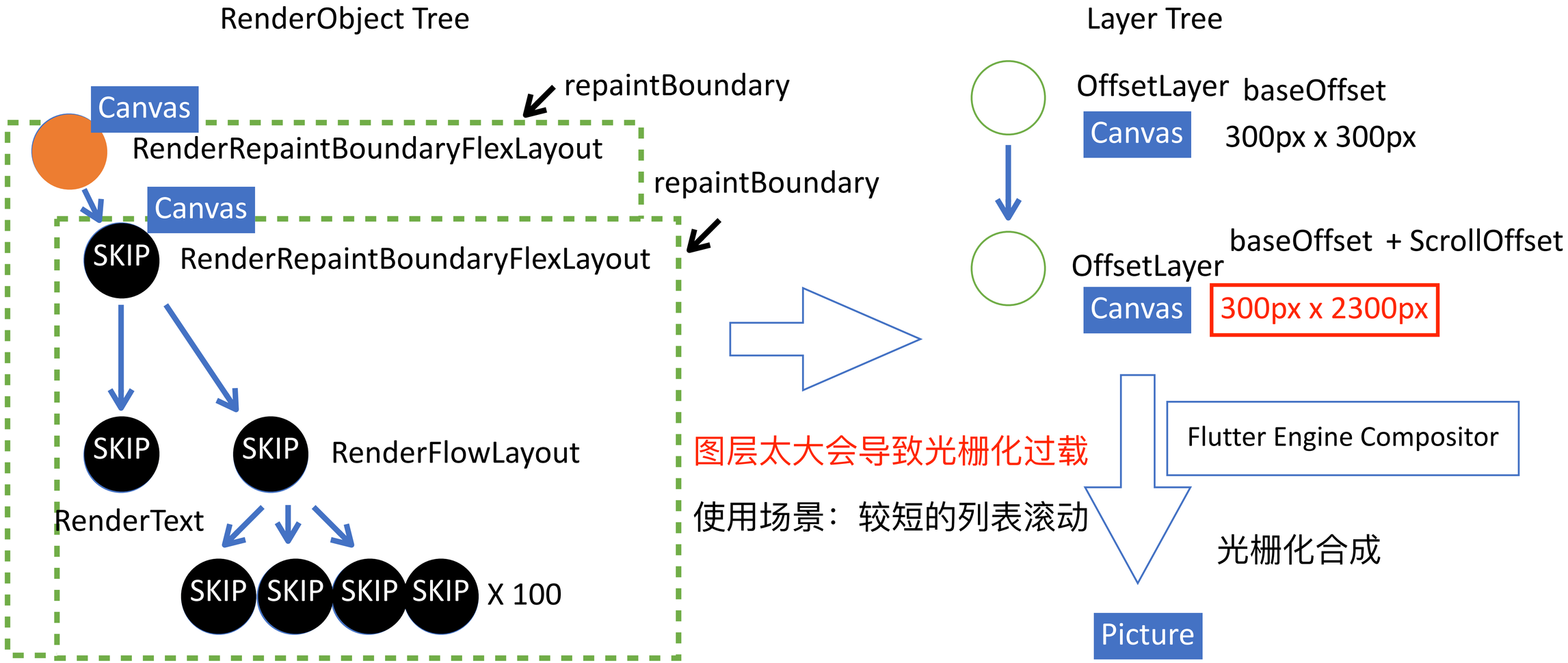
滚动过程中只会有一个 renderObject 在重绘



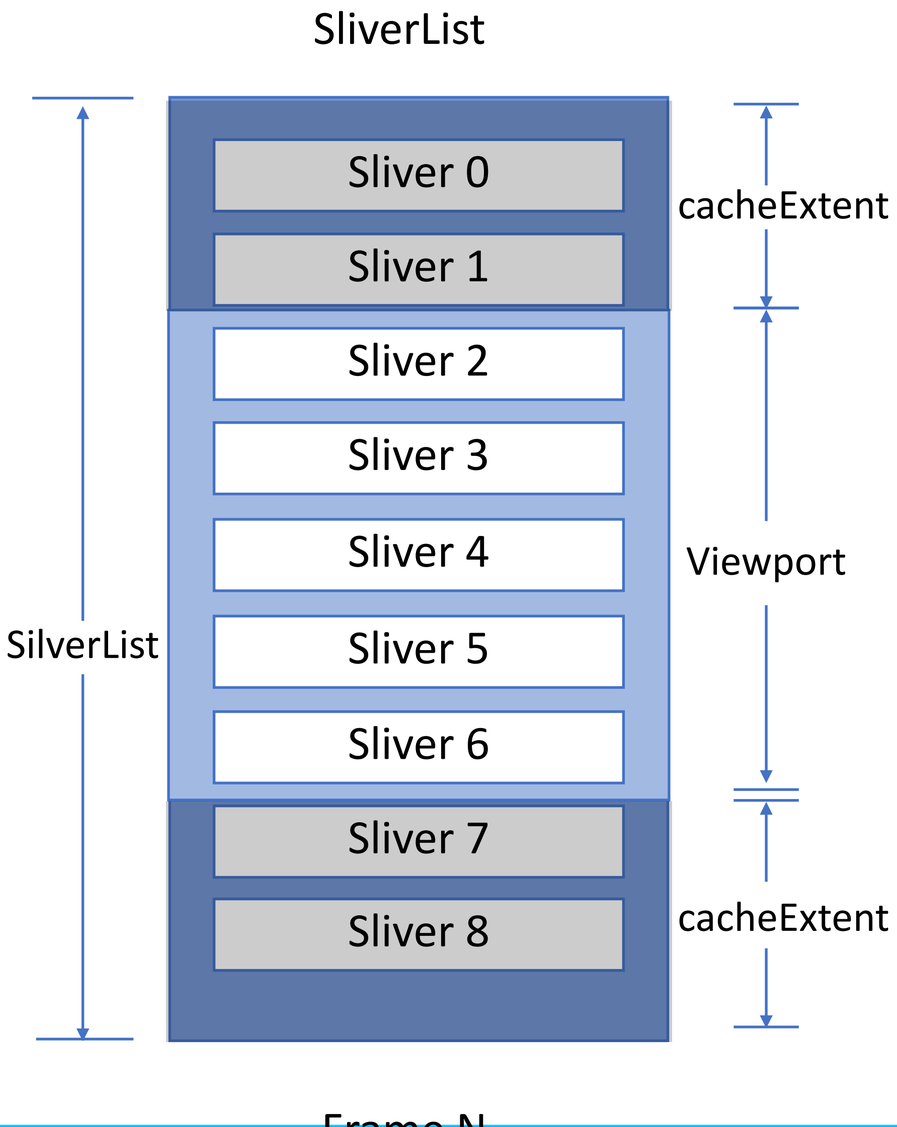
如果滚动的页面特别长呢?

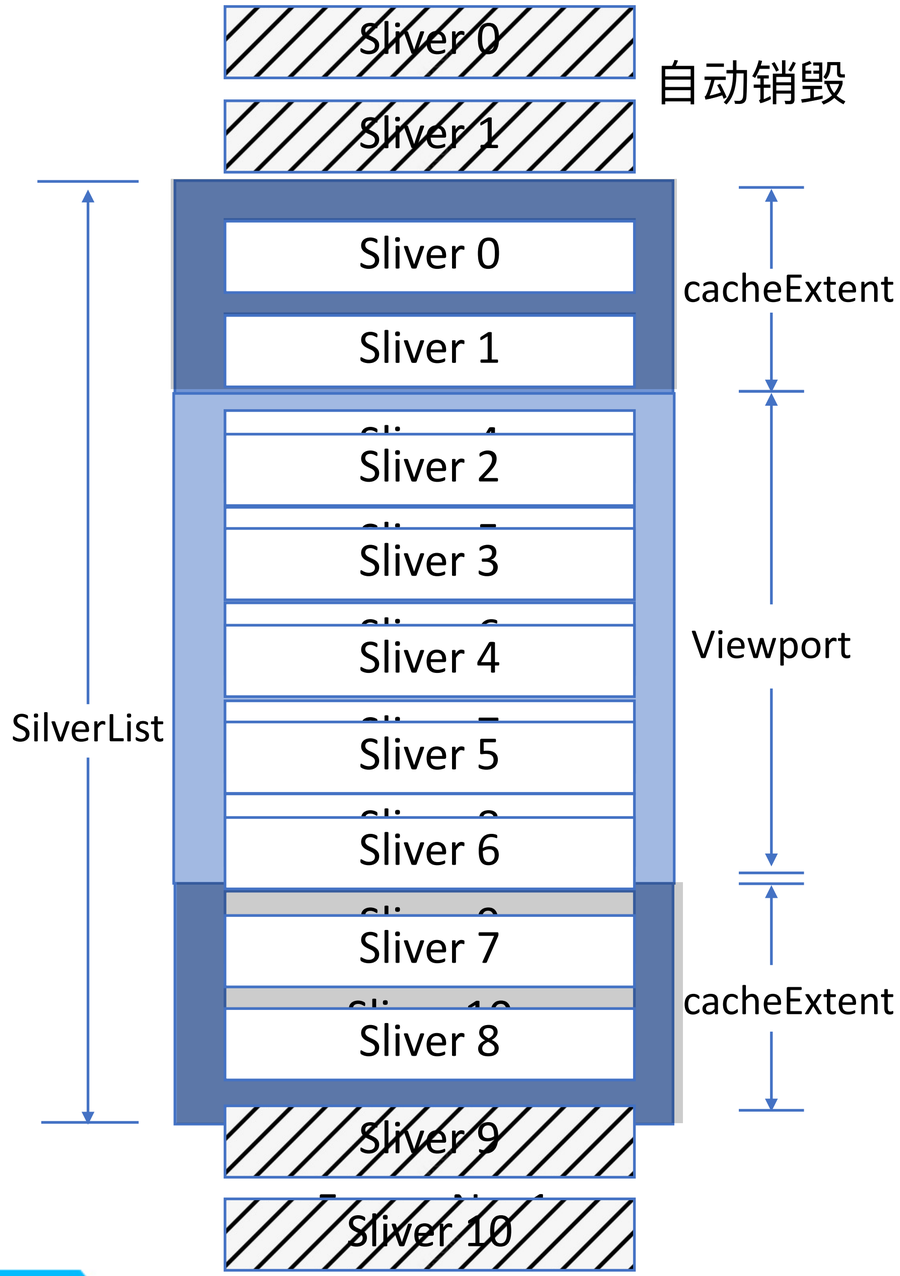
双 repaintBoundary 滚动的瓶颈

双 repaintBoundary 的基本原理是通过创建两个 Canvas，通过将滚动偏移传递给合成器来实现滚动



Flutter 长列表优化策略



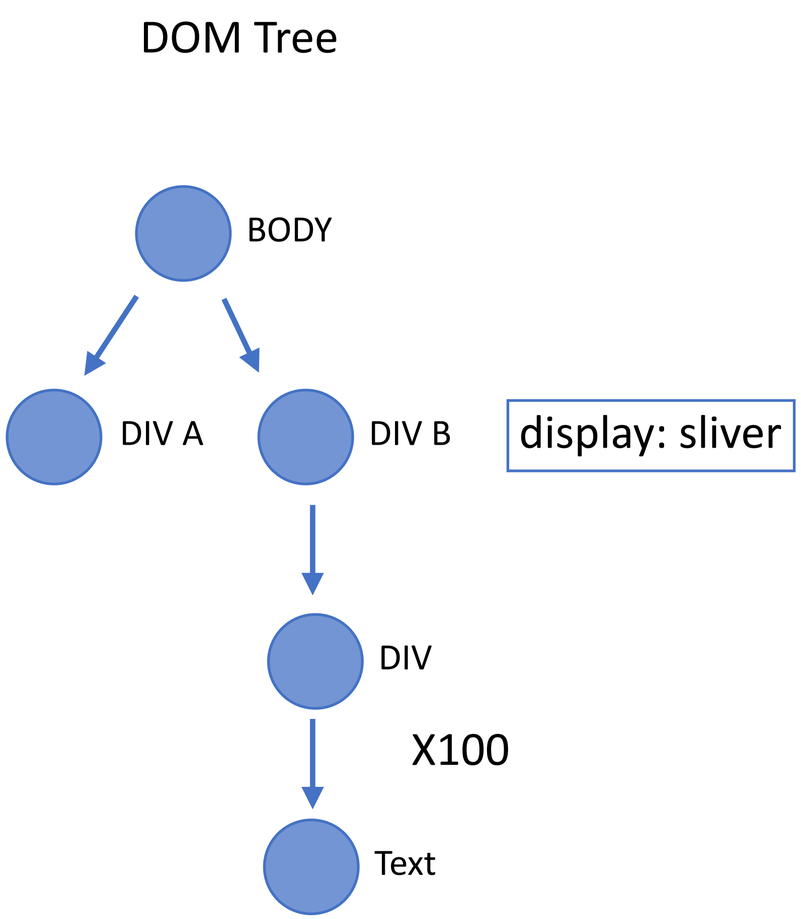


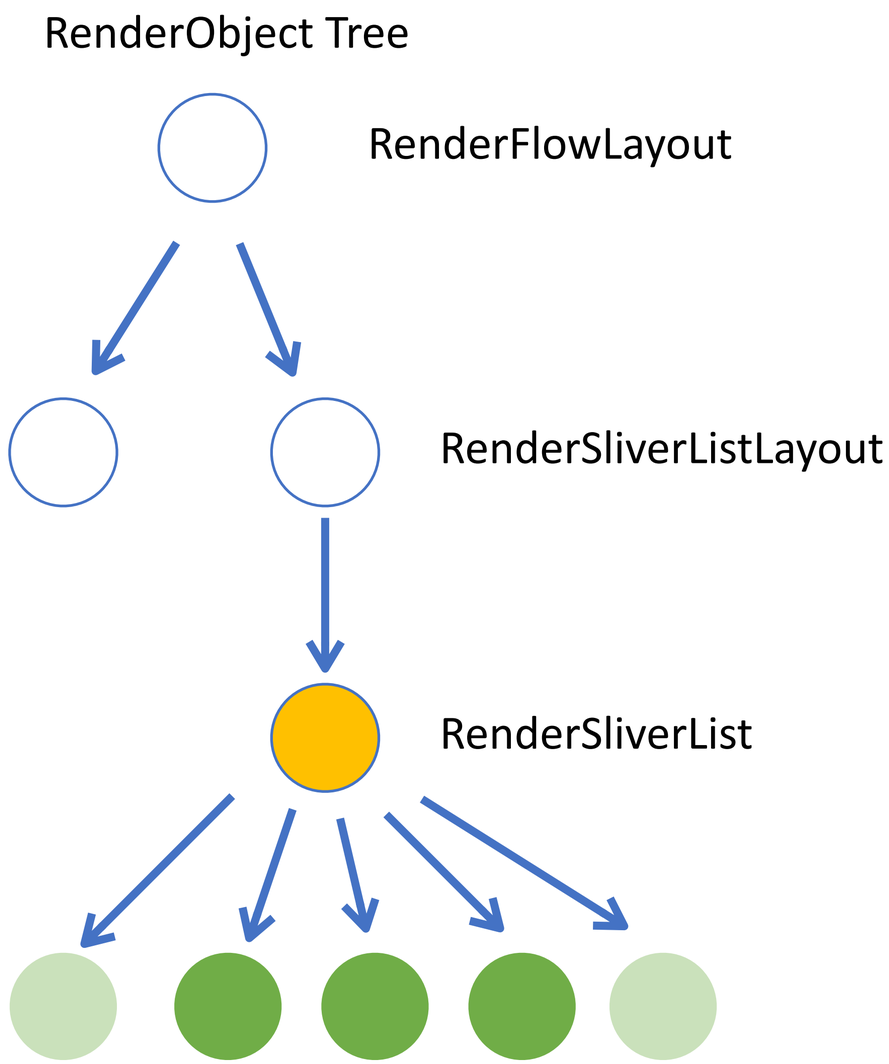
屏幕

Viewport.slivers

采用懒加载 Sliver 元素，只有出现在屏幕内的元素才会执行渲染

通过 CSS 来使用 Flutter SliverList





Q & A

THANKS