

D 2 T E R M I N A L

Qking: 跨端 JS 引擎的深 度探索与突破

胥加成 (花名: 平逸)

阿里巴巴无线开发专家



目录

01

背景介绍

02

性能优化

03

功能增强

04

质量保障

第十七届

D2 终端技术大会

2022.12.17-12.18

前端 & 客户端

合作伙伴

 阿里云开发者社区
ALIBABA CLOUD DEVELOPER COMMUNITY

 大淘宝技术
TAO TECHNOLOGY


 老司机技术
SWIFT OLDDRIVER



segmentfault 思否



 稀土掘金

 钉钉开发者

 InfoQ 极客传媒

 T Salon

 YOUKU 优酷



扫码进入D2官网

*排名不分先后顺序

01

背景介绍

跨端JS引擎的前世今生

JS引擎的跨端应用场景



JS引擎的跨端应用场景



JS引擎的跨端应用场景



JS引擎的选型

常见方案



WebView

- 资源开销大
- 启动慢
- 使用不方便
- 一致性差



V8 JSC

- 资源开销大
- 启动慢
- 双端一致性差
- 稳定性问题严重
- 无法调试JS

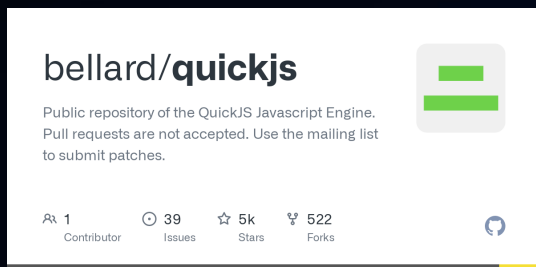


JerryScript DukeTape

- 性能较差
- JS 标准支持不全
- 无法调试 JS

JS引擎的选型

新生代引擎



QuickJS

- 启动性能不够好
- 缺失调试功能
- 堆栈还原无法输出列号
导致 JS minify 文件的
线上报错无法定位



Hermes

- ES 标准支持太低
- 字节码预编译优化和版本绑定，难以进行网络下发
- 缺少线上的 eval 支持

JS引擎的选型

对比

项目	V8 JSC	JerryScript DukeTape	WebView	Hermes	QuickJS
包体积	C	A	A	B	A
内存占用	C	A	C	B	B
启动性能	B	C	C	A+	B-
执行性能	A	C	A	B	B
JS 标准支持	A	C	B	C	A
双端一致性	B	A	C	A	A
稳定性	C	B	C	B	B
使用复杂度	B	A	C	C	A
JS 调试支持	无	无	有	有	无

常见方案面临的问题

前端业务开发

耗费开发时间进行优化

靠console.log进行调试
浪费时间

更大的polyfill产物
更多的适配耗时

发布上线的风险更大

问题

性能问题

JS不可调试

ES标准支持不一致
引擎JS行为不一致

资源占用大
引擎不稳定容易crash

客户端框架开发

耗费开发时间进行优化

更多时间排查问题
端上crash问题恶化

Qking

基于QuickJS的JS引擎

低内存占用

支持内存安全性检查

高ES标准支持

低包大小

支持内存泄露检查

高可用性

毫秒级别启动性能

执行性能提升
20%+

质量

性能

功能

支持Chrome调试

支持JS minify后
行列号还原

Qking

提升交付效率 & 质量

前端业务开发

更好的性能表现
更少时间用在优化上

开发效率和线上问题
排查效率都更高

更小的前端产物
更少的适配用时

更稳定的框架

特性

字节码预编译
优化启动性能

原生Chrome Debug
Protocol调试支持

高ES标准支持
双端行为一致

低资源占用
内存Checker检查

客户端框架开发

更好的性能表现
更少时间用在优化上

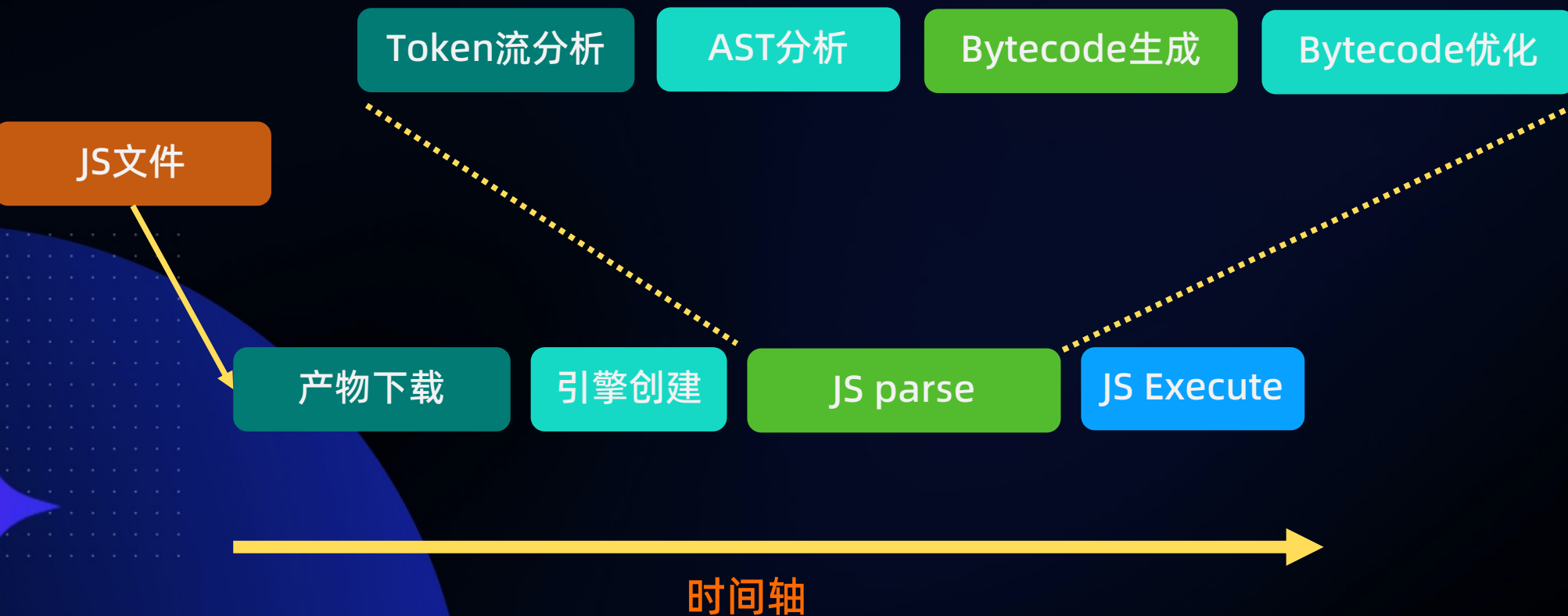
稳定性有保障
问题排查有手段

02

性能优化

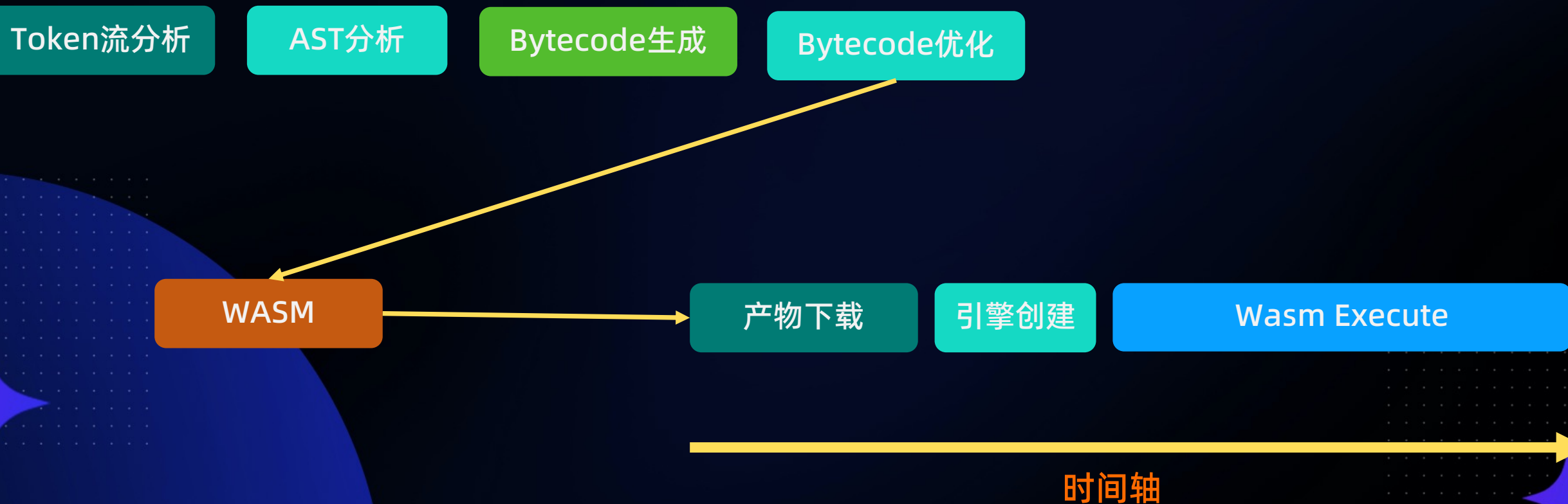
性能的构成

链路长，耗时长



性能的构成

WebAssembly: bytecode生成被前置到离线流程里



性能的构成

1. 线上统计部分复杂业务的 Parse : Execute 耗时达到 3 : 1
2. QuickJS 的 parse 本身相对V8慢

优化手段

- V8: bytecode cache
- Hermes: 预编译bytecode
- QuickJS: 预编译bytecode

缺陷

- Bytecode 无法在引擎版本间兼容, 就很难发布作为CDN的产物
- V8: 产物大小膨胀
- Hermes 阉割掉了在线 eval 的支持
- 没有利用离线parse更充裕的分析时间

JS文件

产物下载

引擎创建

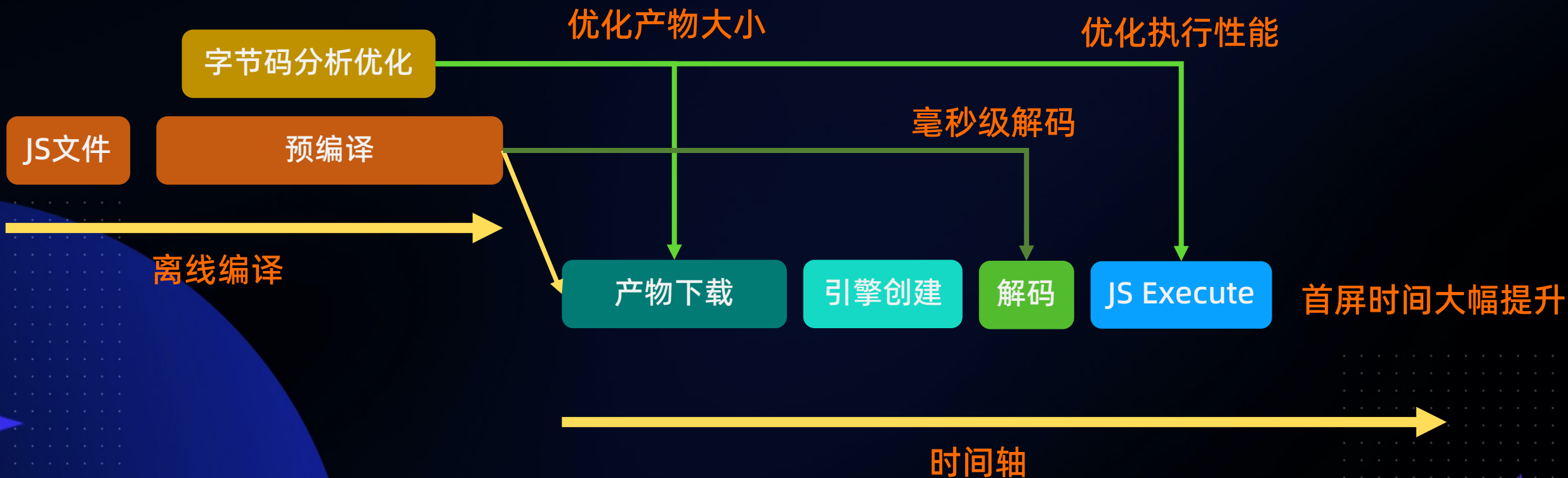
JS parse

JS Execute

时间轴

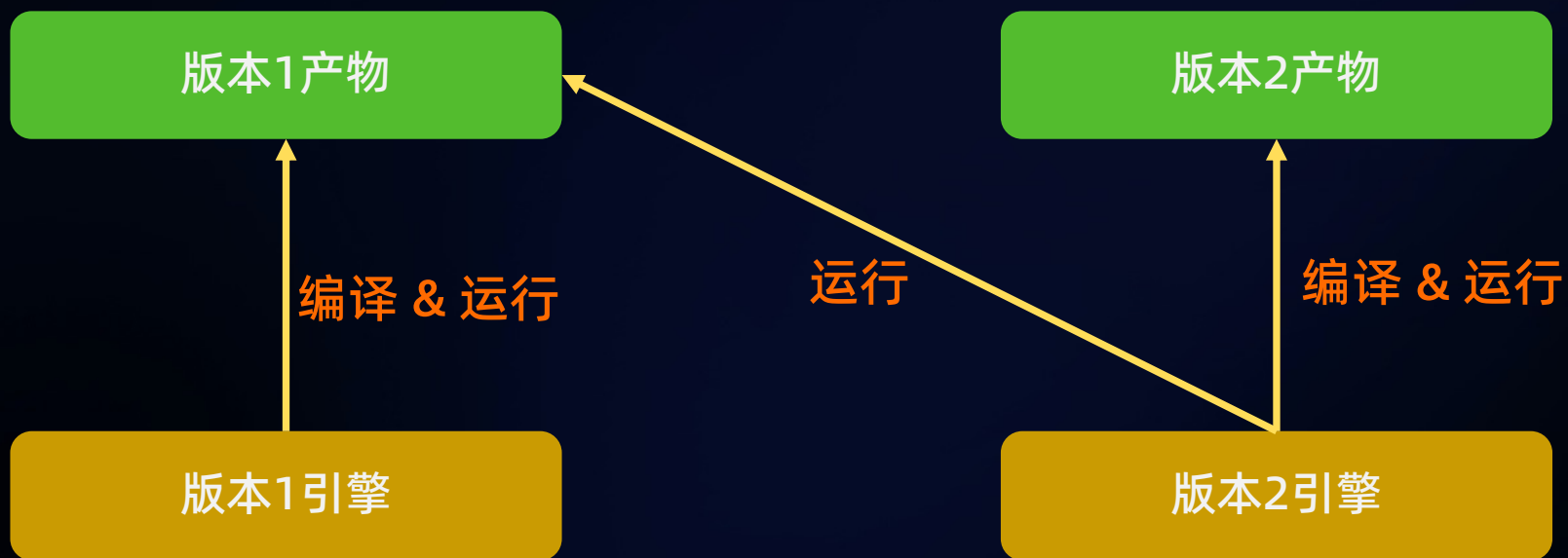
二进制预编译优化

- 实现了版本间字节码兼容
- 保留了原有 QuickJS 的在线 eval 支持
- 利用离线 parse 更充裕的分析时间提升字节码质量



二进制预编译优化

版本兼容的方案



- 字节码变动频率相对很低
- 保持运行时向前兼容只需要保持对老字节码的支持，代码膨胀较少

二进制预编译优化

版本兼容的方案

```
void JS_CallInternal(){
    static const void * const dispatch_table[256] = { ... }

    #define DISPATCH() goto dispatch_table[*pc++]

    v20_op_code_1:
    DISPATCH();

    v20_op_code_2:
    DISPATCH();

    ...
}
```

- Dispatch table切换
- 极低额外开销 1%

```
void JS_CallInternal(){
    static const void * const dispatch_table_v1[256] = { ... }
    static const void * const dispatch_table_v2[256] = { ... }
    void * const current_table =
        bytecode.version ? dispatch_table_v1 : dispatch_table_v2;

    #define DISPATCH() goto current_table[*pc++]

    v20_op_code_1:
    DISPATCH();

    v21_op_code_1:
    DISPATCH();

    ...
}
```


二进制预编译优化

字节码优化的思路

Token Parse

Emit OPCode

窥孔优化1

窥孔优化2

QuickJS原版流程

问题:

- 窥孔优化可优化的 case 不够泛化 (基于栈的字节码数据关系是隐含的)
- 为了保障 parse 速度, 不能做太复杂的分析



优化后

Token Parse

Emit OPCode

窥孔优化1

窥孔优化2

在线Parse不变
保证速度

Token Parse

Emit OPCode

窥孔优化1

Optimizer

窥孔优化2

离线 Parse 增加
Optimizer流程
最大化优化

功能:

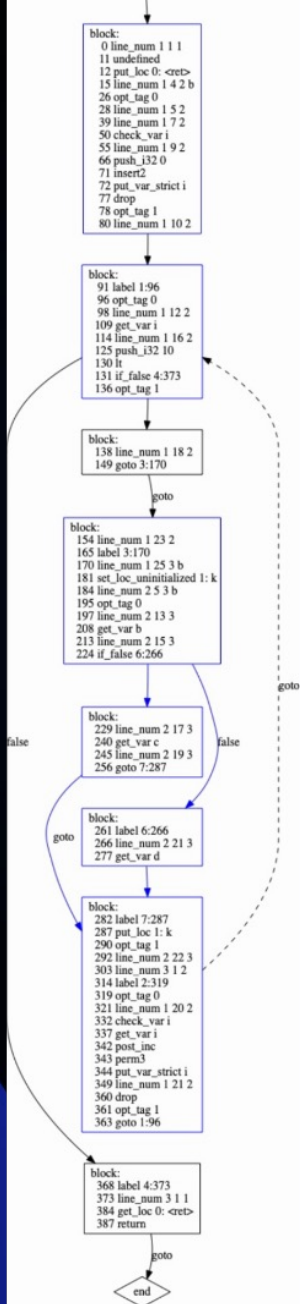
1. 控制流分析

1. 数据溢出分析
2. Deadcode删除

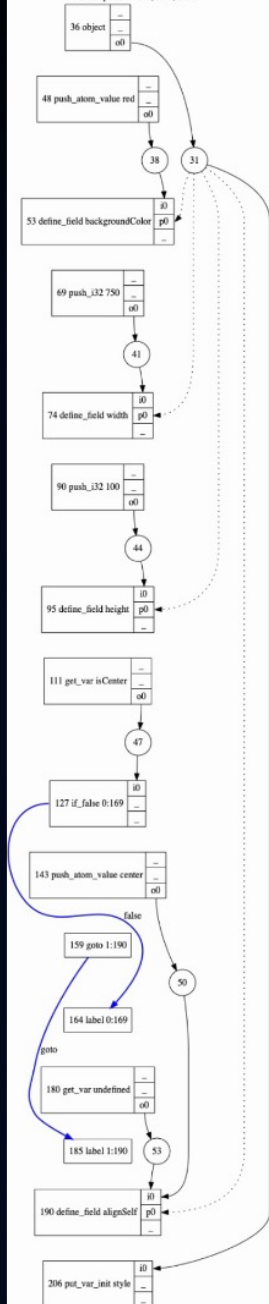
2. 数据流分析

1. 指令合并
2. 指令改写
3. 常量传播

3. 可视化

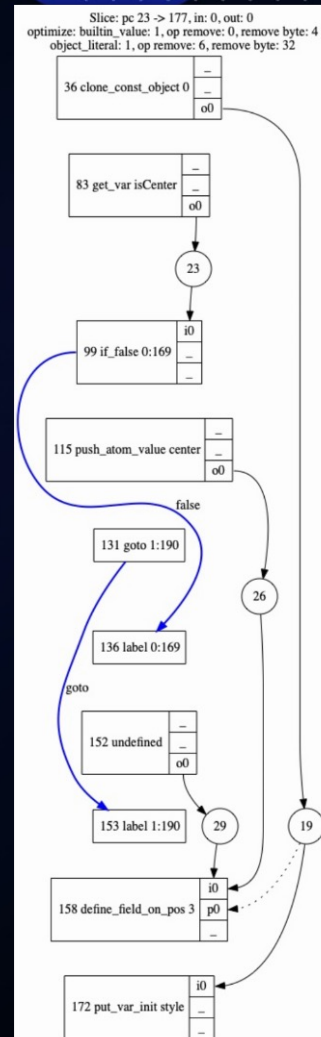


控制流分析



数据流分析

优化后



指令数量减少

二进制预编译优化

得与失

牺牲

锁死JS引擎

对产物发布链路有一定
依赖

获得

首屏性能提升一倍

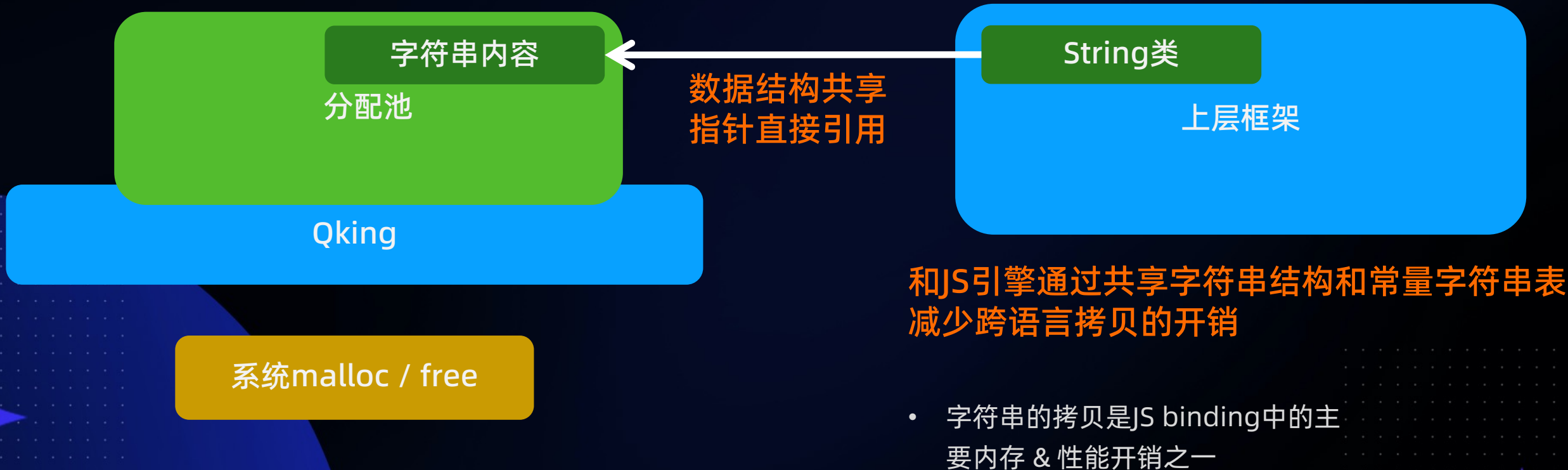
执行性能提升10~20%

额外的功能优势

其他优化手段

内存分配池化 & 字符串Binding优化

利用JS频繁分配释放 & QuickJS 的单线程设计，进行池化分配，减少 malloc / free 次数



性能对比

执行性能对比

测试项目	QuickJS	Qking	Hermes	V8 JIT	V8 NOJIT	JSC JIT	JSC NOJIT
cripto	153.772	126.4	91.904	4.3	160.64	4.7	84
deltablue	11.139	9.95	8.009	0.6	6.5	0.6	8.5
earley-boyer	181.903	146.9	134.051	8.5	77.84	4.7	107.9
navier-strokes	117.27	88.1	90.4	5.4	133.605	6.8	79.68
raytrace	101.018	76.2	62.713	2.9	32.588	1.6	45.1
regex	449.265	382.2	251.02	15.996	34	18.5	244.4
richards2	5.505	5	3.558	0.12	3.9	0.25	3.83
splay	4.951	2.26	2.266	0.31	1.535	0.18	2.34

性能对比

启动性能对比

测试项目	QuickJS	Qking (JS)	Qking (字节码)	Hermes	V8	JSC
引擎启动 + Parse耗时 PC	62	40	2.5	2.6	17	18.2
引擎启动 + Parse耗时 Android低端机	241.6	160.2	8.4	-	-	71.3

- 以一个270KB的JS 业务case为例

性能对比

内存对比

测试项目	QuickJS	Qking
分配次数	242394	138913 (-40%)
分配量	18MB	13MB (-27%)
内存占用量	9.3MB	8.1MB (-13%)

- 以一个线上 JS 业务case为例

03

功能增强

调试功能

传统方案的调试功能实现



V8

- 比较难运行在IOS上
- 全功能的调试器代码较多，包大小大



JSC

- 调试协议非chrome
- IOS上系统提供的JSC不包含debugger

调试功能

传统方案的调试功能实现：虚拟环境



调试功能

传统方案的调试功能实现：替换引擎

上层跨端框架

JS Binding 注入

底层JS引擎(JSC)

线上包

- 维护投入多
- 和真实引擎环境有差异
- 前端开发换包麻烦

上层跨端框架

JS Binding 注入

底层JS引擎(支持调试的引擎)

调试专用包

调试功能

原生调试 vs 虚拟调试

项目	原生调试	虚拟调试/专用包
维护成本	一次性投入，小	持续投入，大
可靠性	非常高	容易出现和真实环境不一致
架构复杂性	低	高
内存dump	和真实场景占用一致	不一致
线上问题现场	可以接入调试	无法接入，现场丢失

正确好用的调试能降低成本，而错误的调试一定会大幅增加成本

调试功能

难点 & 问题

- 需要修改 QuickJS 内部的实现，提供一套Debug用的API出来
- 和CDP协议对齐需要大量的JSON协议处理 & 类型转换
- 断点的实现需要实现 字节码OP offset 和 行列号 之间的双向映射能力，原生 QuickJS 只有行号的还原，没有列号也没有 行列到offset的映射。
- 部分非核心的CDP的功能无法实现，比如代码自动提示 (需要无副作用的执行环境，较难实现)

预编译二进制

行列号映射支持

QuickJS 没有列号映射原因:

1. 影响parse速度
2. 导致产生的bytecode映射mapping过大

Qking的预编译二进制:

1. 离线parse可以不影响线上性能
2. 映射mapping可以抽离单独的文件发布

预编译天生适合解决此问题

预编译二进制

行列号映射支持

QuickJS 仅行号支持

Token Parse

Bytecode Emit

OP_add
OP_linenum 1
OP_get_field

Bytecode Opt1

OP_add
OP_get_field

Bytecode Opt2

bytecode

行号映射

Linenum
信息收集

Qking 在线流程: 和 QuickJS 原版保持一致

Token Parse

Bytecode Emit

Bytecode Opt1

Bytecode Opt2

bytecode

行号映射

预编译二进制

行列号映射支持

QuickJS 仅行号支持

Token Parse

Bytecode Emit

Bytecode Opt1

Bytecode Opt2

bytecode

行号映射

Qking 离线流程

Token Parse

Bytecode Emit

OP_add
OP_linenum 1 2
OP_get_field

Bytecode Opt1

OP_add
OP_get_field

Bytecode Opt2

二进制文件

行列号映射文件

OP_linenum增加列号参数
Parse期间分析并发射列号

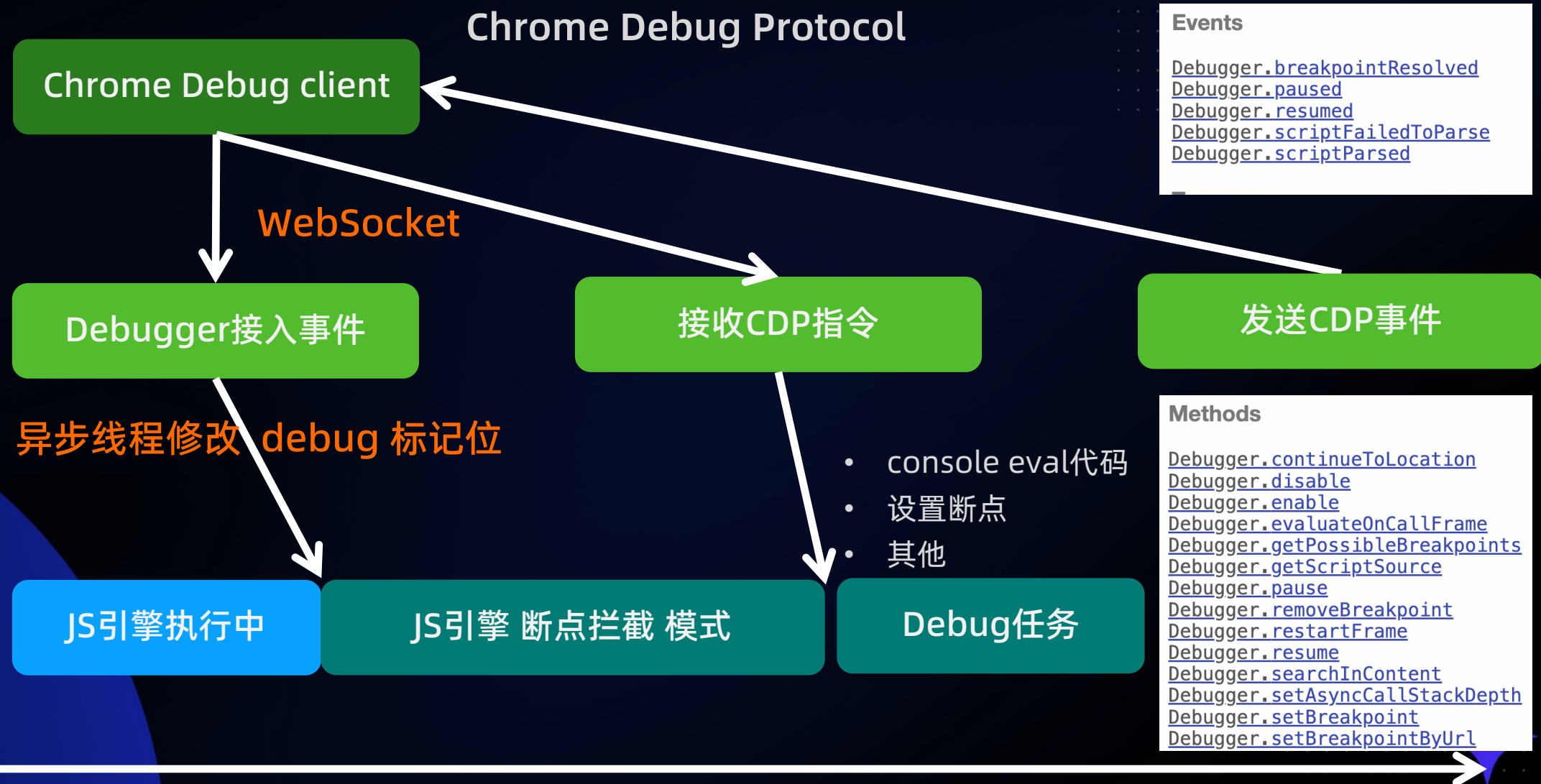
收集成单独的mapping文件
减少产物大小

调试功能

Chrome Debug Protocol

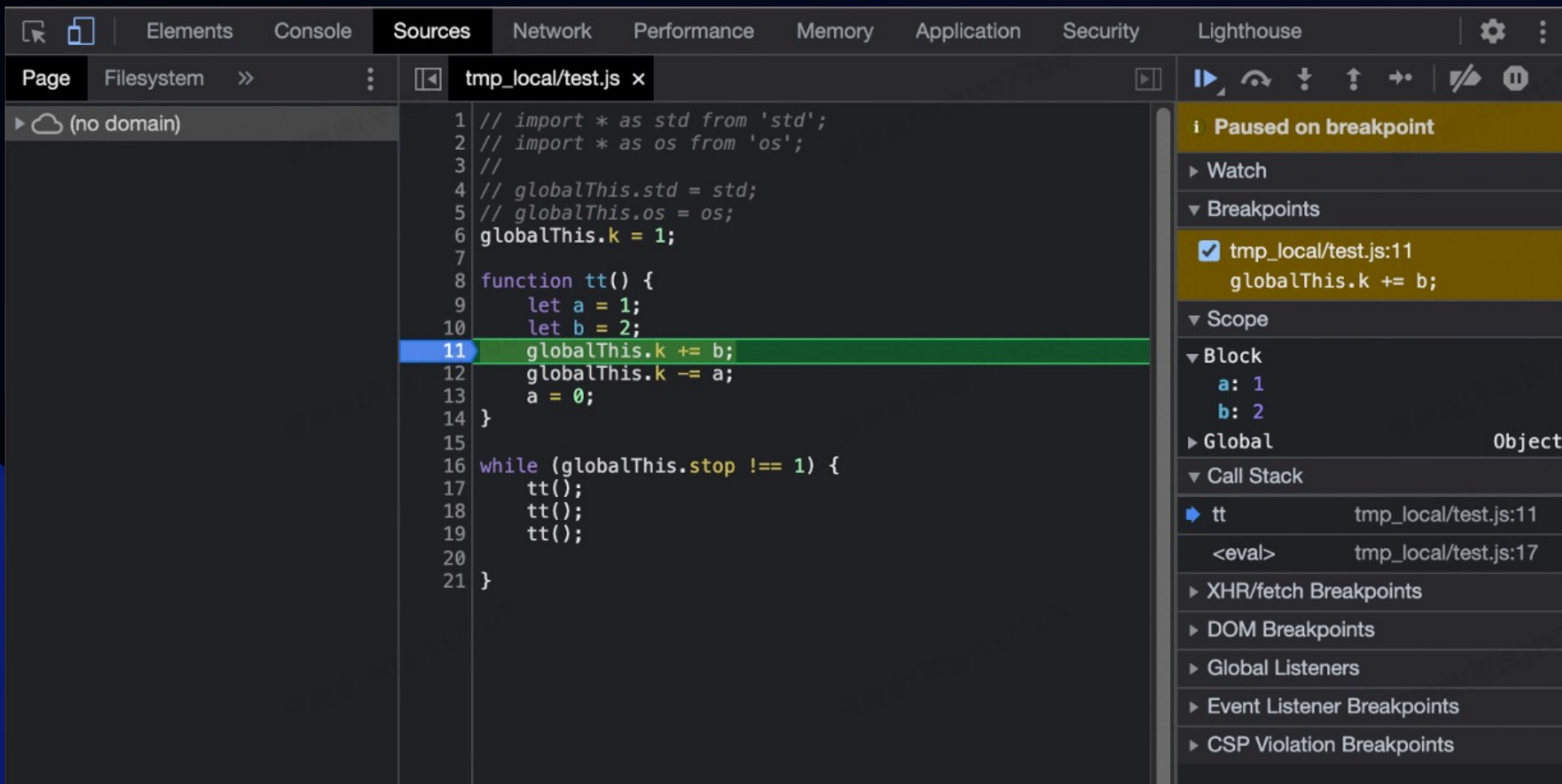
Debugger线程

JS线程



调试功能

效果



预编译二进制

原生 JS 功能上的一些缺陷

问题

JS文件有线上损坏的问题
(由于文件或网络的原因被截断)

QuickJS 没有列号映射
影响 JS minify 文件的报错监控

解决方案

二进制里增加checksum进行自校验

改造离线的parser
在预编译二进制里支持列号映射

预编译二进制

功能	JS文件 (QuickJS)	Hermes 二进制	QuickJS 原版二进制	Qking 预编译二进制
版本间兼容性	无兼容问题	不支持	不支持	支持
搭建场景文件拼接	支持	不支持	不支持	支持
行列号映射	只支持行号	支持	只支持行号	支持
额外 Resource 打包	不支持	不支持	不支持	支持
文件自校验	不支持	支持	不支持	支持

- 预编译二进制的功能集是 原生 JS 的超集

预编译二进制

和QuickJS 原版bytecode对比

QuickJS bytecode

Strings

Object

预编译二进制

Header

Checksum

Strings

Object

Resource Sections

Meta Section

可继续扩展

- 版本号 magic number等
- 自校验功能
- 类似原版bytecode格式，存储function的方法体
- 额外Resource打包
- 携带meta信息

04

质量保障

质量保障

跨端 JS 引擎的常见问题

问题

内存泄露，内存占用高

JS引擎内野指针造成 crash

建议

App不要使用全局的JS引擎，从业务维度多实例隔离。确保业务页面退出可以释放干净。

保障对引擎的API的正确使用，防止泄露/野指针出现

需求

JS 引擎的创建销毁足够轻量

API设计本身不容易出错
使用出错的时候可排查定位

质量保障

跨端 JS 引擎的常见问题

JSC

V8

JerryScript

QuickJS

需求

相对更重

轻量

JS 引擎的创建销毁足够轻量

容易出错

API
设计
不易
出错

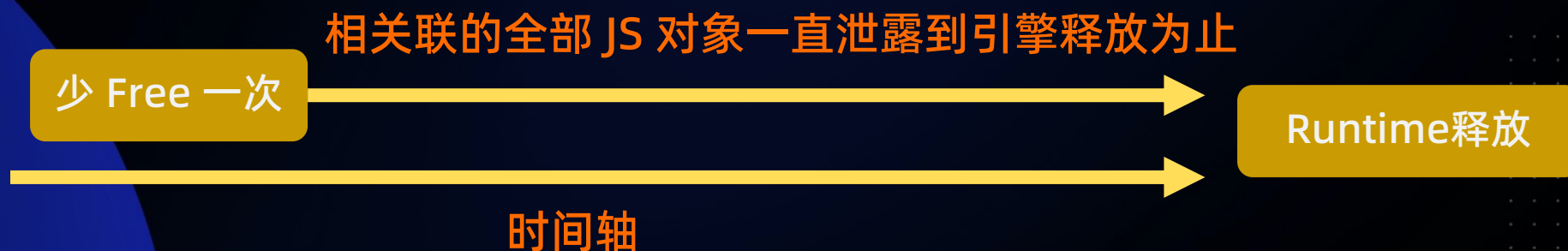
引用计数风格API
非常容易出错

API设计本身不容易出错
使用出错的时候可排查定位

质量保障

跨端 JS 引擎的常见问题

1. 可能立刻crash
2. 可能不crash
3. 可能在很久之后不相关的函数里crash



质量保障

现实情况

墨菲定律：会出错的事总会出错

问题的特点：难以定位，出现具有随机性



令人头痛的线上问题

LeakChecker

防止泄露

操作堆栈记录:
+1 hello.c:32

遍历未释放的对象, 报告泄露根源
Leak at hello.c:32: ref count: 1

JS_NewObject

JS_FreeRuntime

时间轴



效果

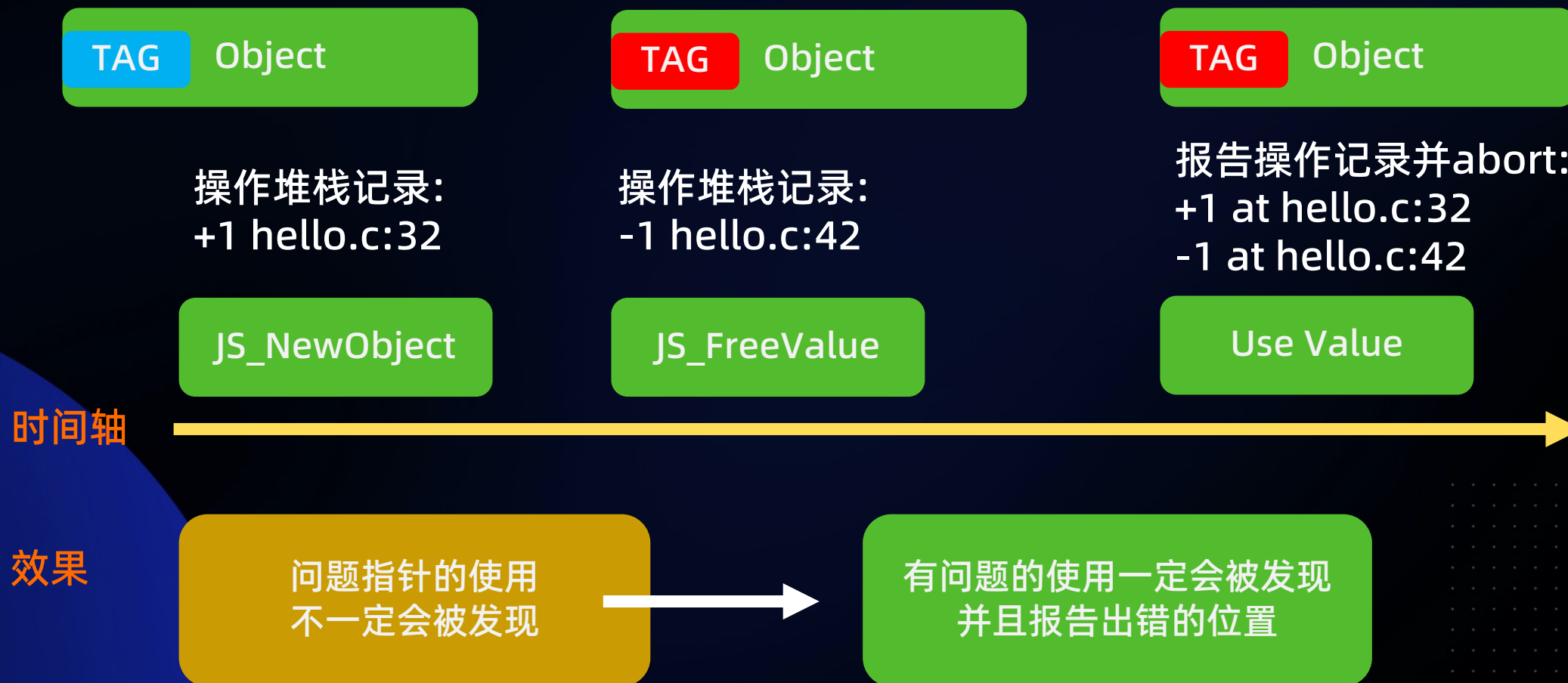
泄露问题只知道发生了,
不知道为什么



泄露问题一定可以
被发现 & 定位

PointerChecker

防止野指针



Checker使用效果

- 发现了大批线上 Binding 代码的错误使用 case
- 保障了 Qking 的上层框架在2w行 DOM binding代码中没有线上问题的产生
- 大幅提高了开发期间写 binding 的开发效率

THANK FOR YOUR WATCH

感谢大家观看



**扫码回复「D2」
获取第十七届 D2 演讲 PDF 材料**

后续也将推送 D2 会后技术文章，敬请关注！！