NODE.JS & V8

@SYSU

About Me

张秋怡

- 12 级软件工程
- 校招进入 alinode (阿里云)
  - https://alinode.aliyun.com
- 目前工作
  - Node.js 管理解决方案（alinode）的开发
  - 内外部客户的性能优化技术支持
- joyeec9h3@gmail.com

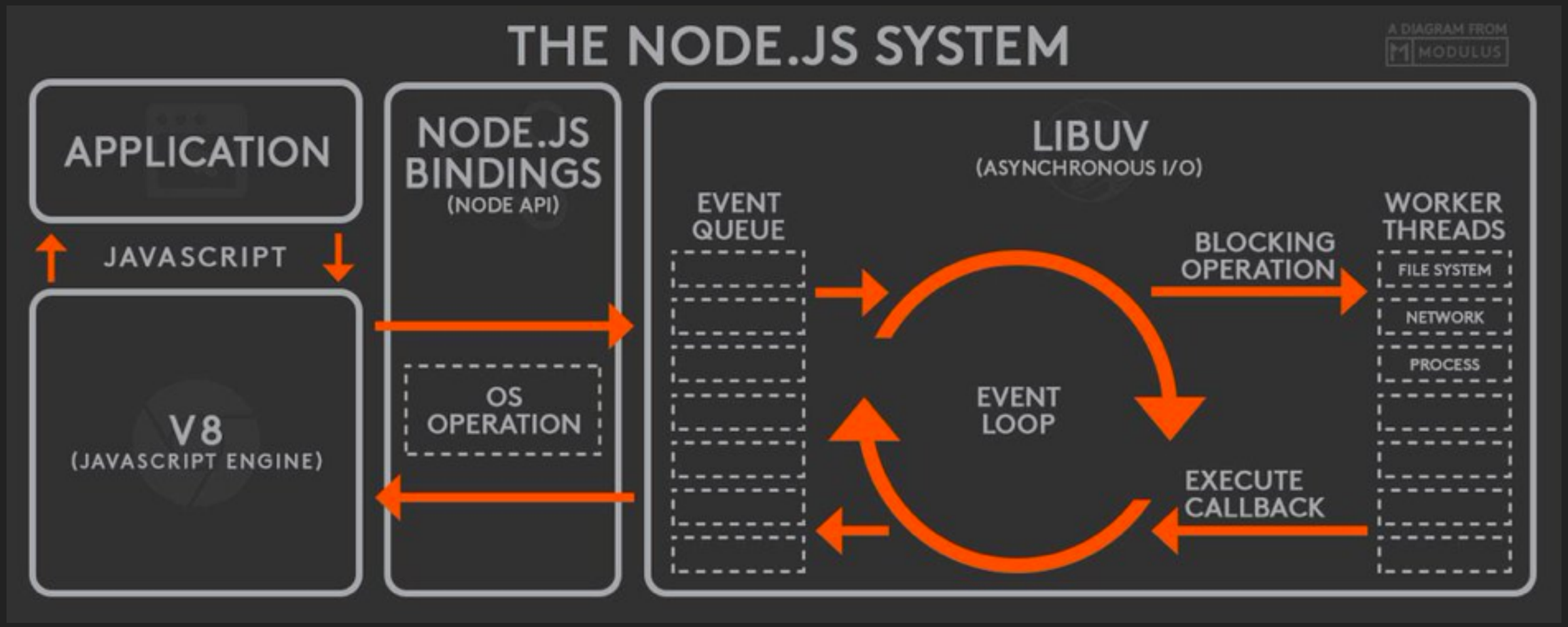
Before We Start

Node.js 是什么?

```
const http = require('http');


const hostname = '127.0.0.1';
const port = 3000;


const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Before We Start

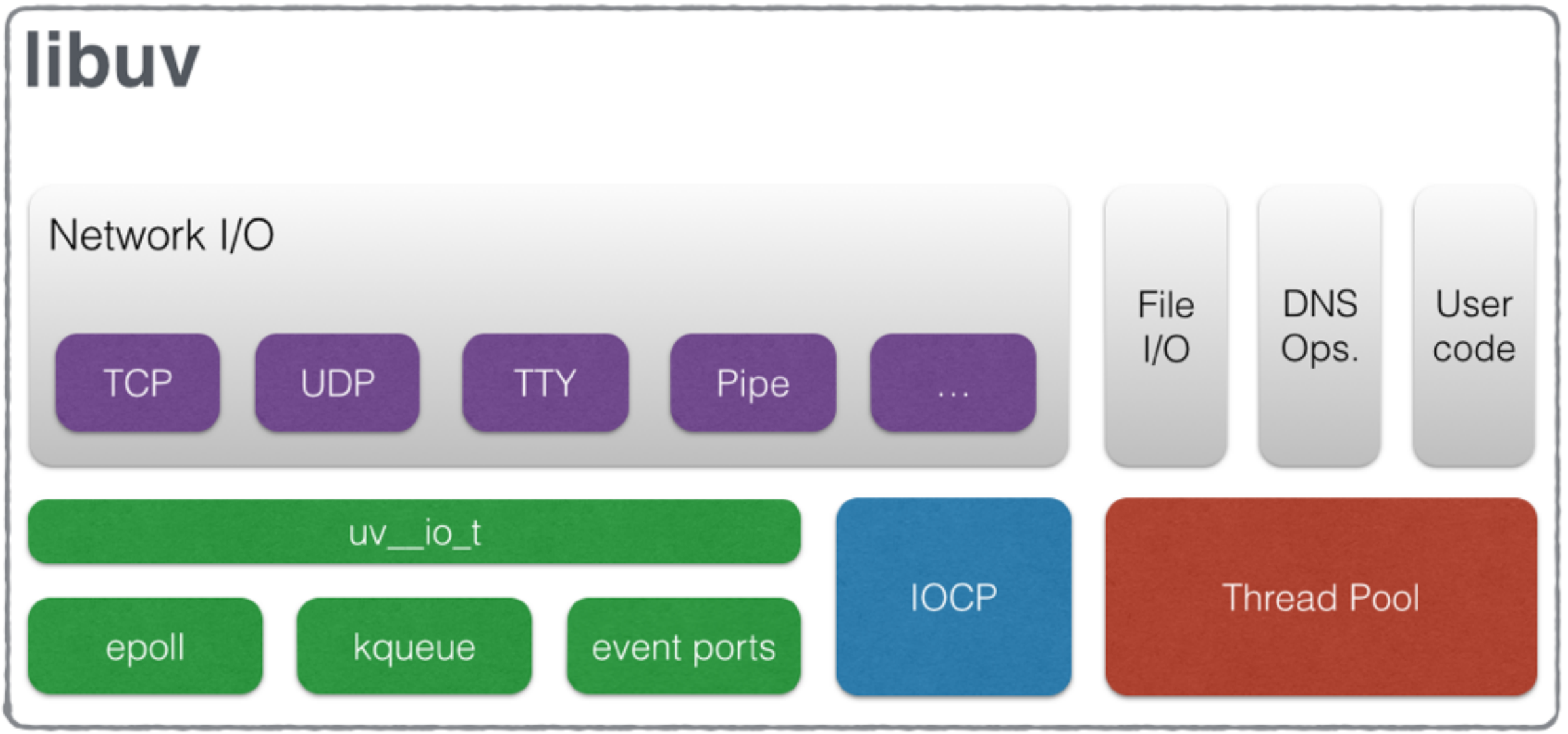
Node.js 是什么?



Credit: BusyRich @ Twitter

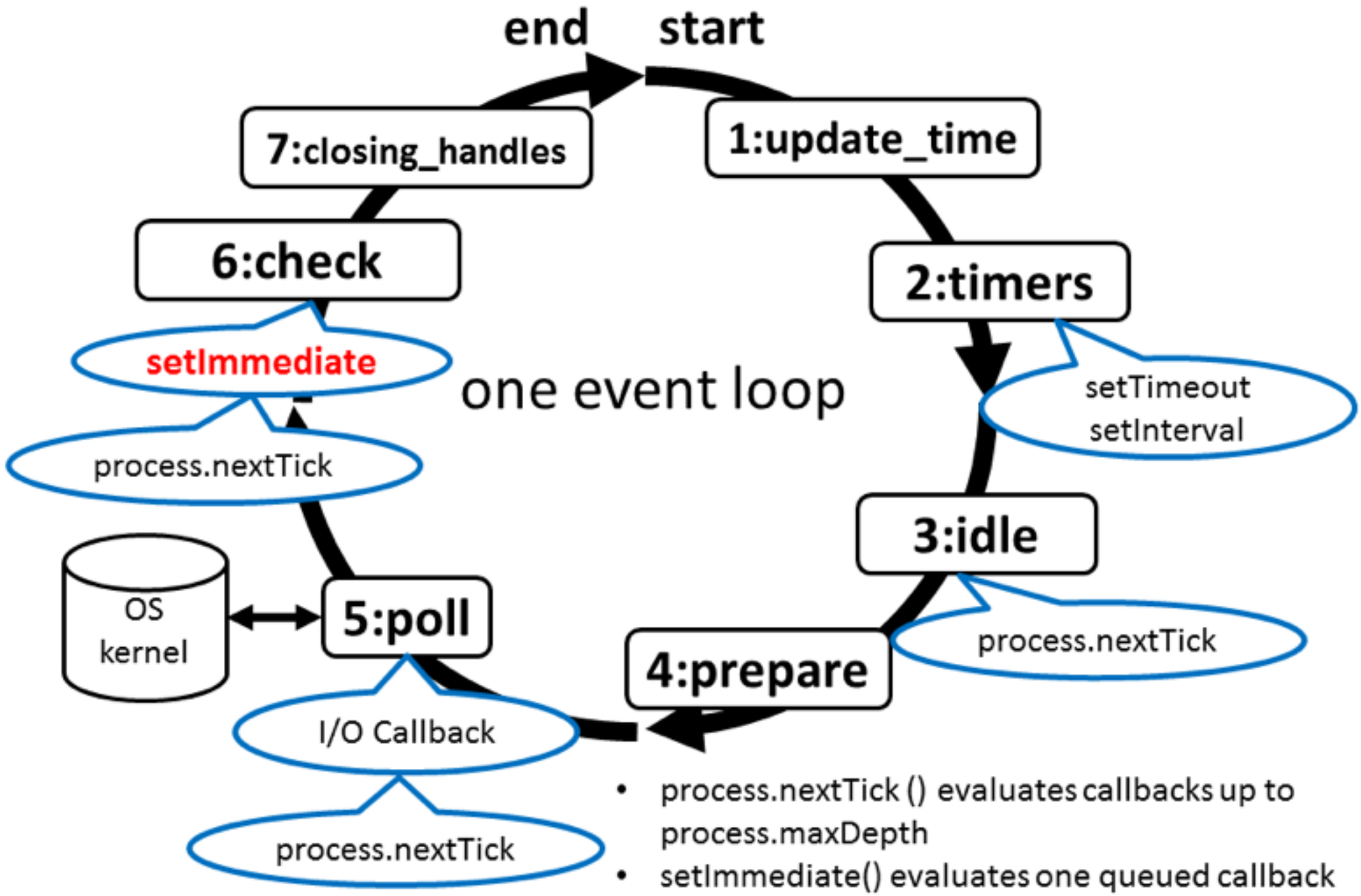
- 核心包括 libuv + cares（DNS）+ OpenSSL(crypto) + V8(执行JavaScript) 等
- 本身主要由 C/C++ 组成，部分是 JavaScript 写的
- 用户在使用 Node.js 时，代码主要是 JavaScript 的，也可以写 C/C++ 的 addon，暴露 binding 和 JavaScript 互相调用

What is libuv?



Credit: libuv documentation

What is libuv?

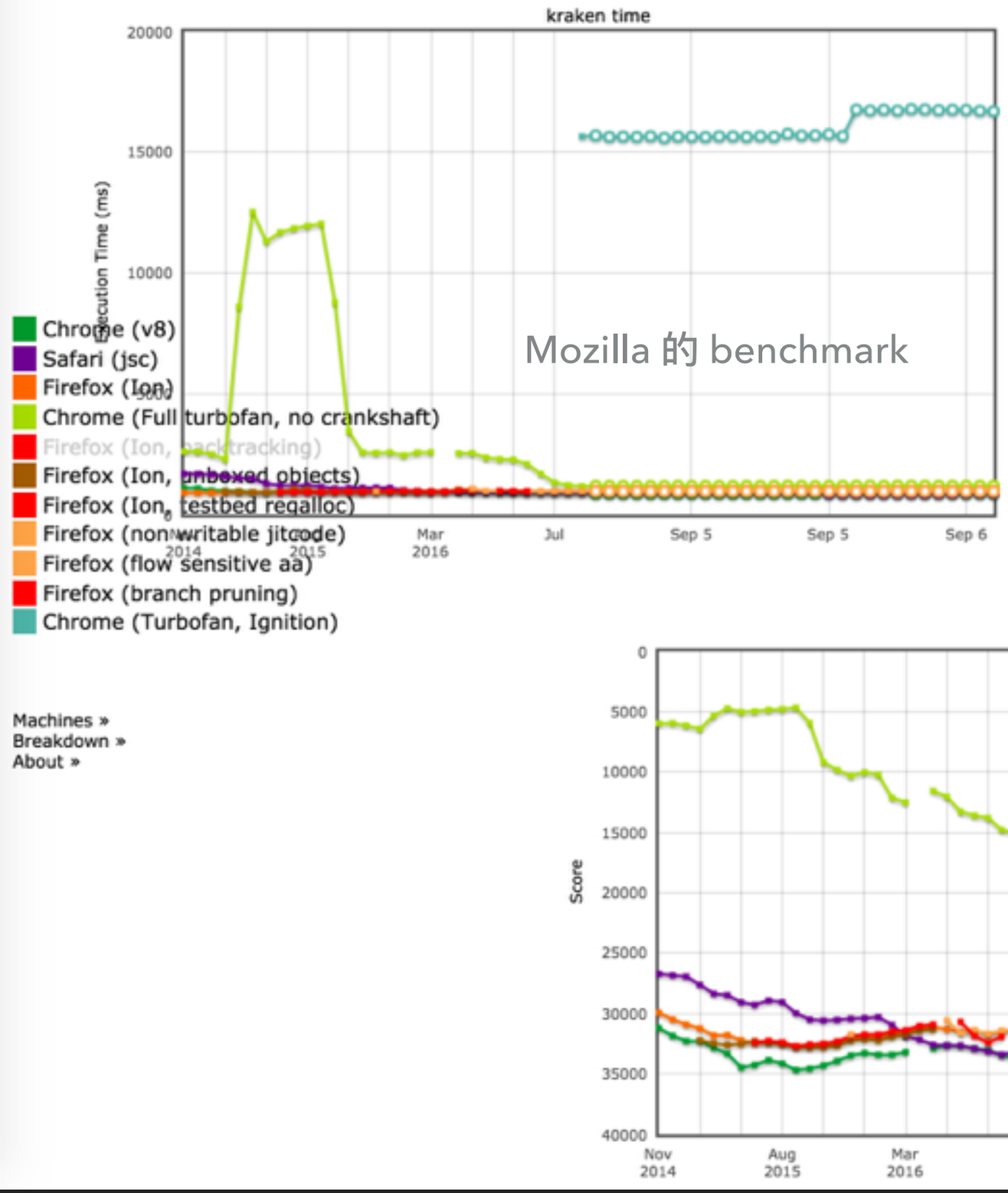


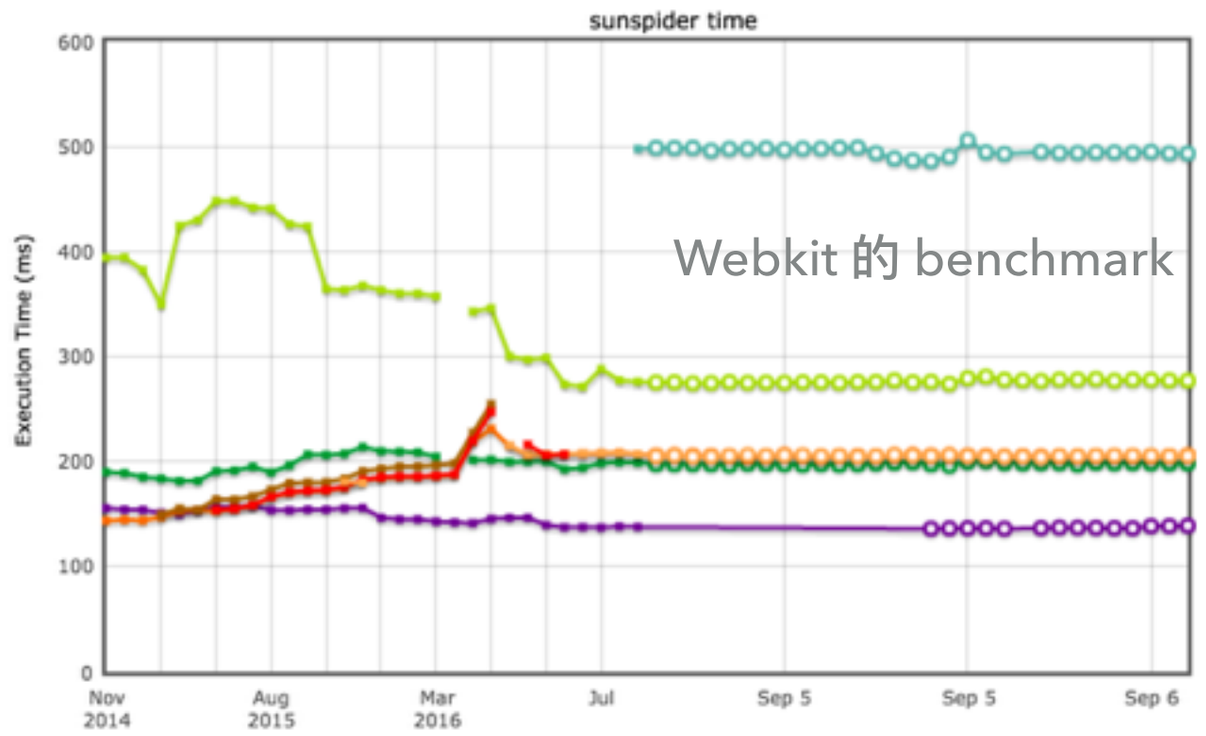
Credit: ?

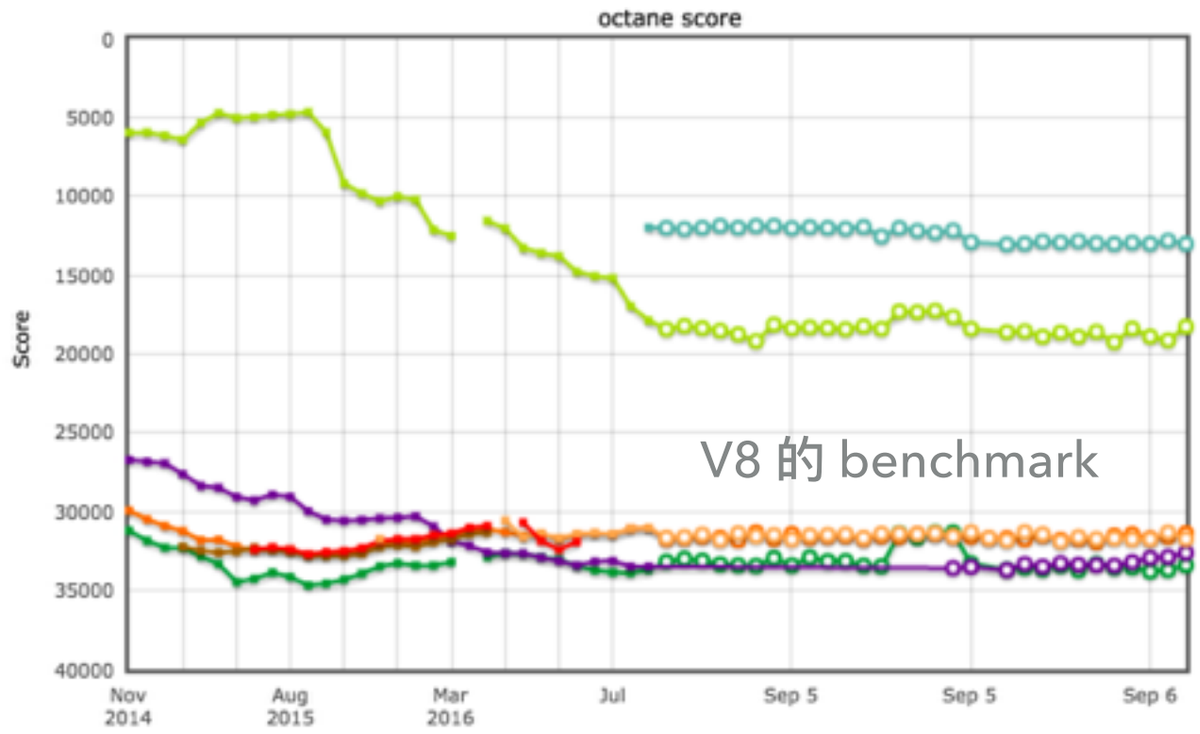
What is V8?

- Chrome 里的 JavaScript 引擎
- Designed by Lars Bak
  - HotSpot JVM 的设计者
- JavaScript 引擎大战的导火索

JavaScript 引擎大战

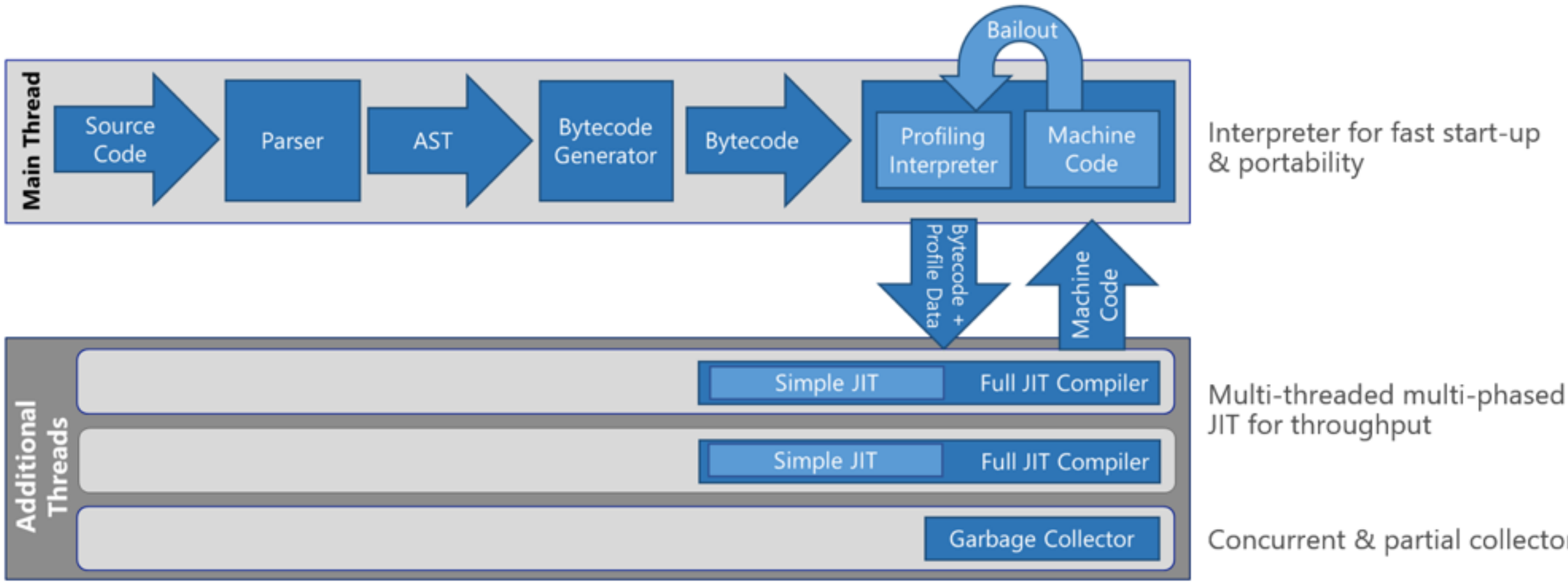






Credit: Are we fast yet?

JavaScript 引擎的架构

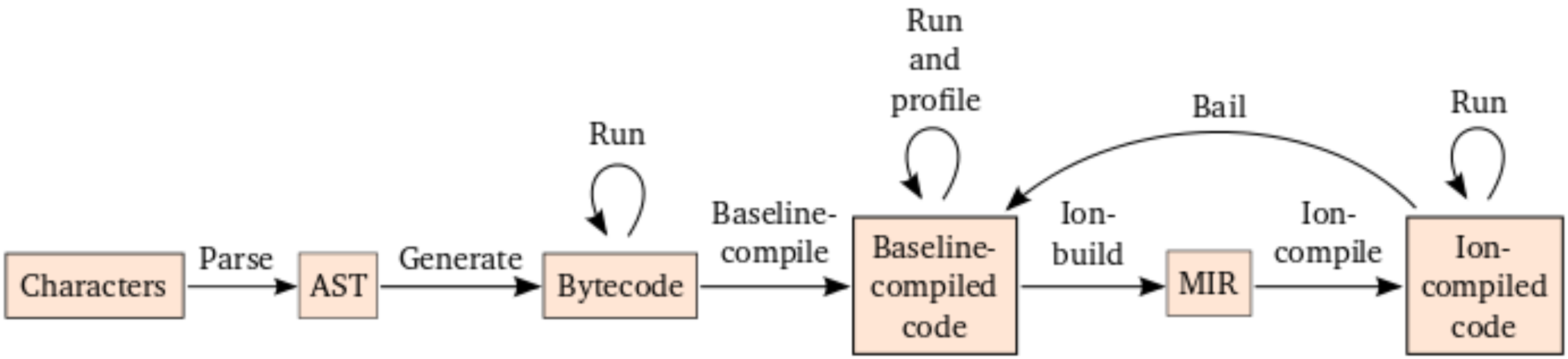


Credit: MS Edge Blog

Chakra

(IE/Edge/WP)

JavaScript 引擎的架构

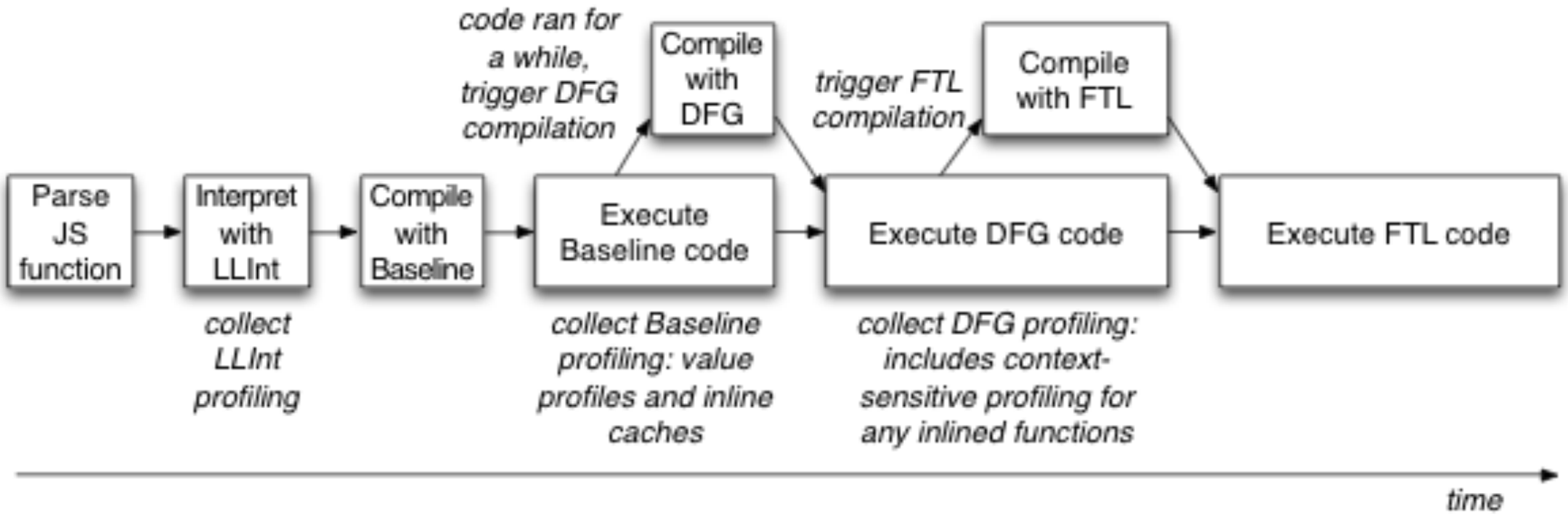


Credit: Luke Wagner's Blog

SpiderMonkey

(FireFox)

JavaScript 引擎的架构

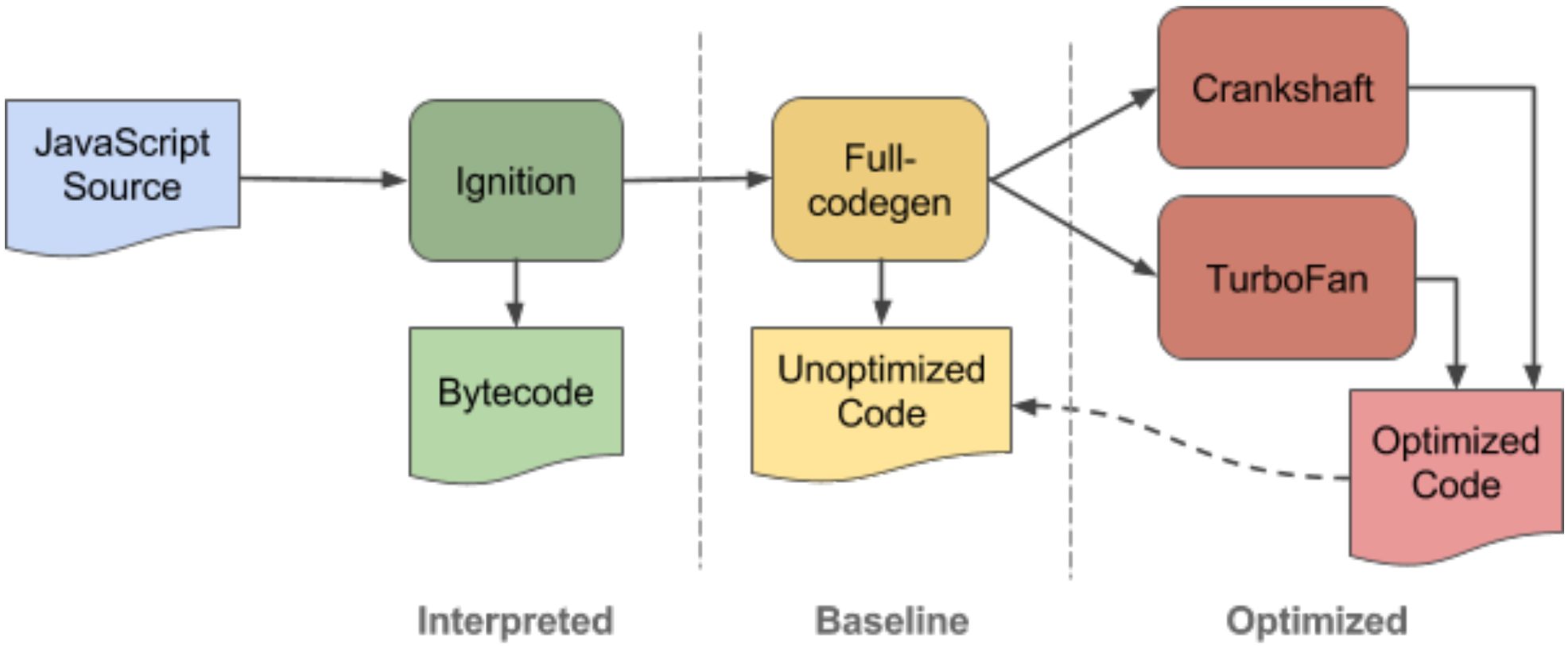


Credit: WebKit's blog

JavaScriptCore

(Safari/iOS WebView)

JavaScript 引擎的架构



Credit: V8's blog

V8

(Chrome/Android WebView)

异曲同工

- 解释器＋多 tier 的 JIT 编译
  - 首先你要将源代码解析成 AST（抽象语法树），字符串 -> 树形数据结构

```
1  // Life, Universe, and Everything
2  var answer = 6 * 7;
3
```

No error

Syntax node location info (start, end):

- [ ] Index-based range
- [ ] Line and column-based

- [ ] Attach comments

Syntax | **Tree** | Tokens

Expand All | Collapse All

- Program body [1]
  - VariableDeclaration
    - declarations [1]
      - VariableDeclarator
        - id
          - Identifier
            - name: answer
        - init
          - BinaryExpression
            - operator: *
            - left
              - Literal
                - value: 6
                - raw: 6
            - right
              - Literal
                - value: 7
                - raw: 7
    - kind: var

http://esprima.org/demo/parse.html

解释器?

- 遍历 AST，编译到 bytecode
- 可以想象成一个大大的 while loop，对 bytecode 做 switch 并解释执行
- CPython，Lua，第一个 JavaScript 引擎……大部分的动态语言的第一个实现

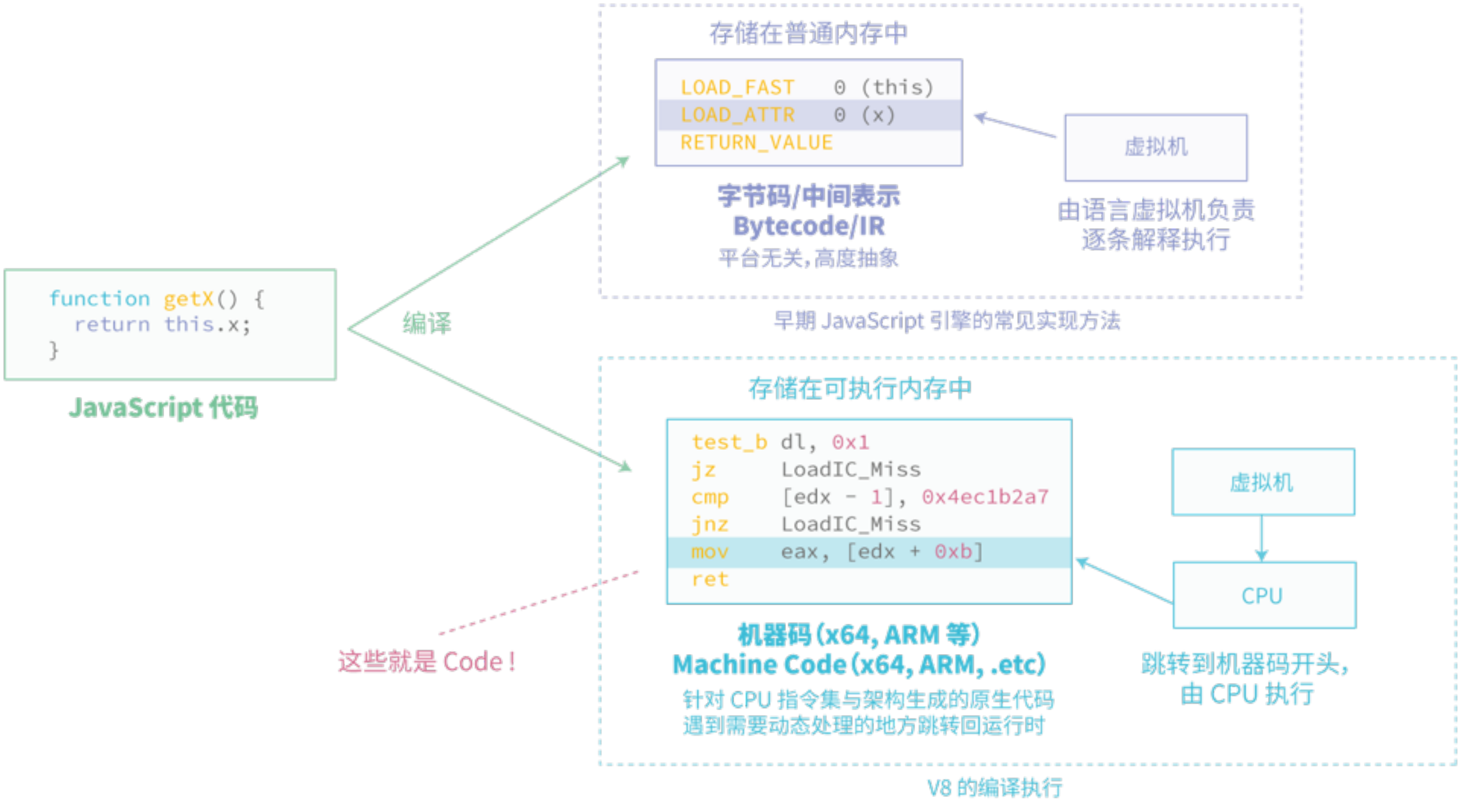
```
while(true) {
  bytecode = fetch_code();
  switch (bytecode.opcode):
    case 'ADD':
      add(bytecode.operand);
      break;
    case 'LOAD':
      load(bytecode.operand);
    ...
}
```

JIT 编译?

- 加载到可执行内存（类似 shellcode）
  - 为什么 iOS 上的 Chrome 不能跑 V8?
- 耗费内存和运算时间
  - 不做优化，直接翻译（V8 的 baseline JIT ~ stack machine）
  - 放在解释器后面，先直接解释跑起来，再编译部分热点到机器码
- 多 tier 的 JIT
  - 有的不优化，编译速度快，出来的代码执行效率一般
  - 有的进行优化，编译速度慢，出来的代码效率高
    - 优化力度不一样，编译时间不一样
    - 对的写法 + 对的优化 = 比 C++ 还快的 JavaScript
    - 优化 JIT 和 AOT 比也是要流氓……
- 各种各样的非系统级语言
  - Java, C#…

JIT 编译

- 遍历 AST，（生成 IR 后再）生成机器码，也可以在 bytecode 上做
- bytecode（一堆数据，0/1） / AST -> 机器码（一堆数据，0/1）



JIT 编译

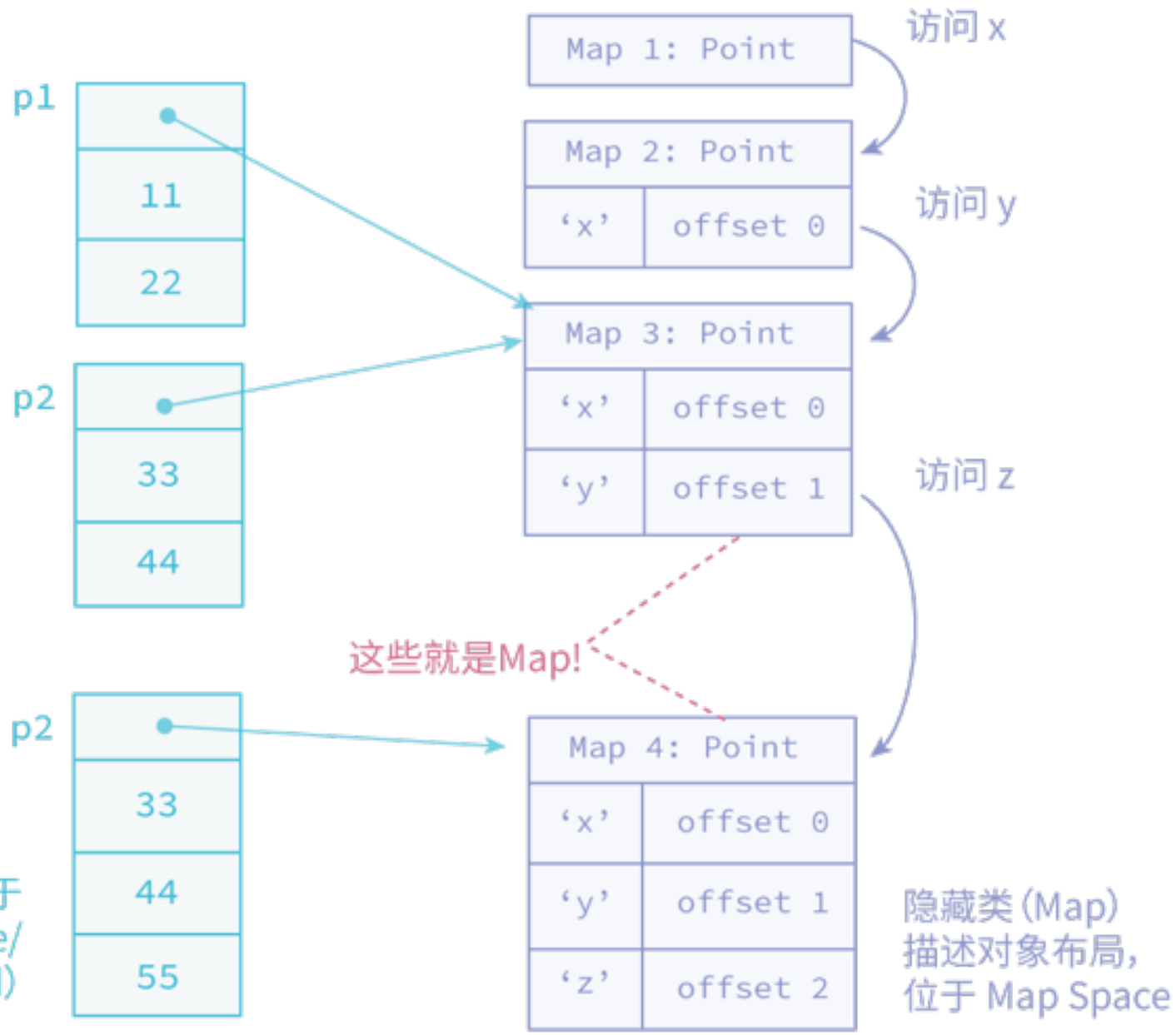
- V8 最近才加上解释器 Ignition，并且只在内存<500M的Android上开启
- 不能说 JavaScript 是解释型语言，因为你电脑上的 V8 只有编译器
- 动态特性（eval, .etc）逃回 runtime 处理
- Baseline JIT + Optimizing JIT
  - 热点启用
  - 默认 Crankshaft
  - Ignition & WebAssembly 使用 TurboFan（WIP）
- 想要快，让代码落到 Crankshaft 进行优化
  - 写得跟 C 一样朴实的 JavaScript

```
// Flags for Crankshaft.
DEFINE_BOOL(crankshaft, true, "use crankshaft")
```

```
// Flags for TurboFan.
DEFINE_BOOL(turbo, false, "enable TurboFan compiler")
```

V8 Hidden Classes

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p1 = new Point(11, 22);

var p2 = new Point(33, 44);

p2.z = 55;
```



Inline Cache

- 弄几个快速的特化版本.....

```
function divideSomeNumbersBy2 (lhsArray,
  for (var i = 0, l = lhsArray.length; i
    resultArray[i] = divideBy2(lhsArray[
  }
}
```

```
function divideByNumber (lhs, rhs) {
  return lhs / rhs;
}
function divideBy2 (lhs) {
  return lhs >> 1;
}
function divideBy4 (lhs) {
  return lhs >> 2;
}
```

```
function divideSomeNumbersBy4 (lhsArray, resultArray) {
  for (var i = 0, l = lhsArray.length; i < l; i++) {
    resultArray[i] = divideBy4(lhsArray[i]);
  }
}

function divideSomeNumbersByUnknown (lhsArray, divisor, resultArray) {
  for (var i = 0, l = lhsArray.length; i < l; i++) {
    resultArray[i] = divideByNumber(lhsArray[i], divisor);
  }
}
```

Credit: JSIL's wiki

Inline Cache

- 运行的时候跳一跳，万一中了呢?
  - 只要代码写的朴实，经常中

```
function divideSomeNumbers (lhsArray, divisor, resultArray) {
  if (lhsArray.length !== resultArray.length)
    throw new Error("Arrays must be the same size");

  // Inline cache
  if (divisor === 2) {
    return divideSomeNumbersBy2(lhsArray, resultArray);
  } else if (divisor === 4)
    return divideSomeNumbersBy4(lhsArray, resultArray);
  } else {
    // Cache miss! A JIT would likely record the miss here, and consider
    //  updating the cache. It'd notice eventually if most trips through
    //  the cache are misses, or if the cache has too many entries.
    // In these cases the IC might be removed entirely for performance.

    return divideSomeNumbersByUnknown(lhsArray, divisor, resultArray);
  }
}
```

Credit: JSIL's wiki

Inline Cache

- 上面的是 JSIL 的实现例子
- V8 里 Inline Cache everything!
  - 比如，最频繁的，对象属性的访问

```
ClassicObject.prototype.getX = function () {
  return this.x; // (1)
};
```

```
mov eax, [ebp+0x8] ;; load this from the stack
mov edx, eax       ;; receiver in edx
mov ecx, "x"       ;; property name in ecx
call LoadIC_Initialize  ;; invoke IC stub
```

Credit: Vyacheslav Egorov

Inline Cache

- 上面的是 JSIL 的实现例子
- V8 里 Inline Cache everything!
  - 比如，最频繁的，对象属性的访问

```
mov eax, [ebp+0x8] ;; load this from the stack
mov edx, eax       ;; receiver in edx
mov ecx, "x"       ;; property name in ecx
call LoadIC_Initialize ;; invoke IC stub
```

替换 stub！

Credit: Vyacheslav Egorov

```
test_b edx, 0x1  ;; check that receiver is an object not a smi (SMall Integer)
jz miss          ;; otherwise fallthrough to miss
cmp [edx-1], 0x2bb0ece1  ;; check hidden class of the object
jnz miss                 ;; otherwise fallthrough to miss
mov eax, [edx+0xb]  ;; inline cache hit, load field by fixed offset and return
ret
miss:
jmp LoadIC_Miss  ;; jump to runtime to handle inline cache miss.
```

Polymorphic Inline Cache

- 看上去很美，那我们要怎么知道会有哪些情况出现呢?

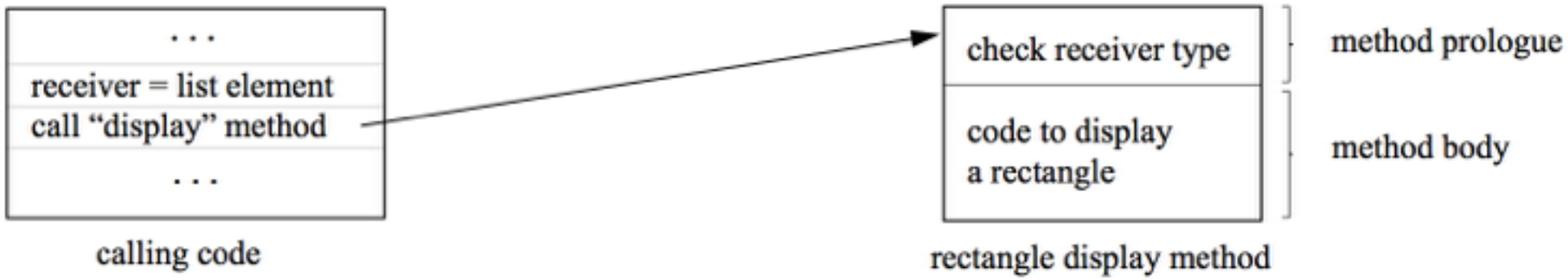


Figure 2. Inline cache after first send

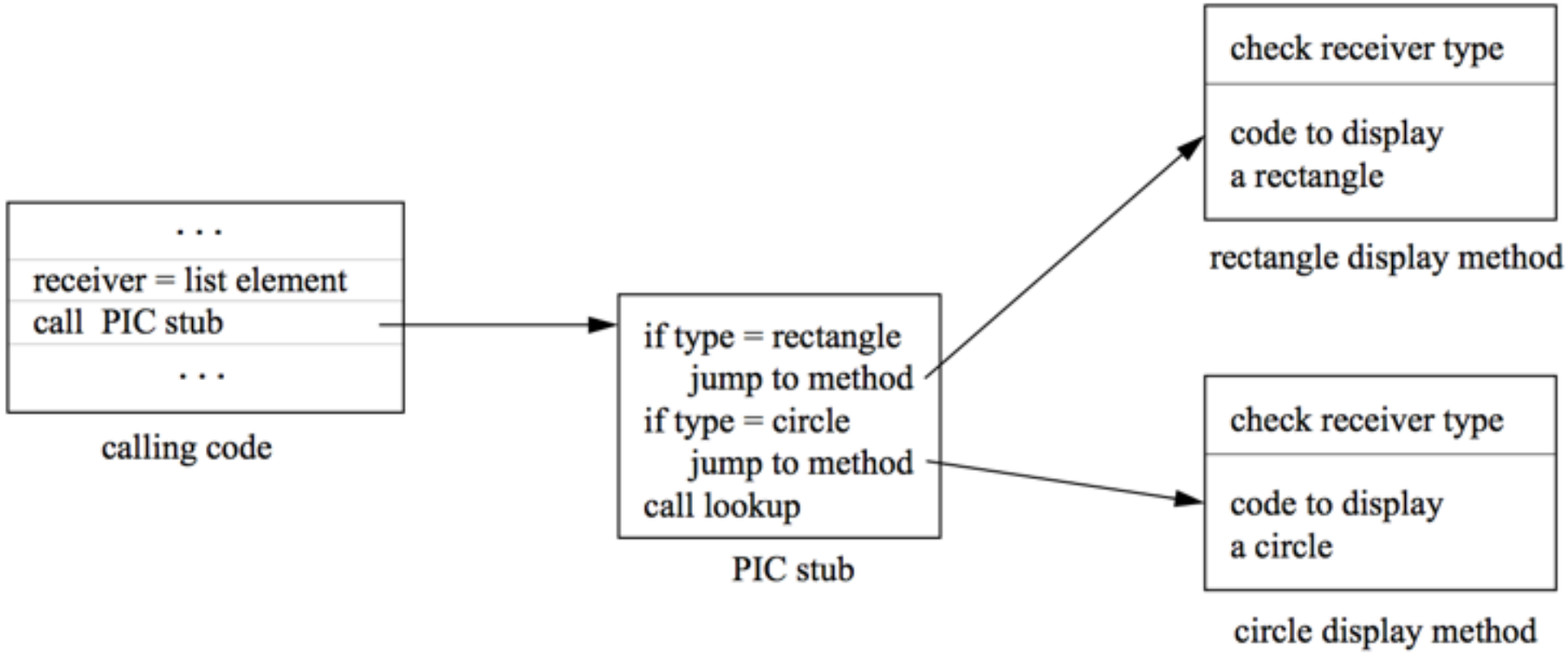
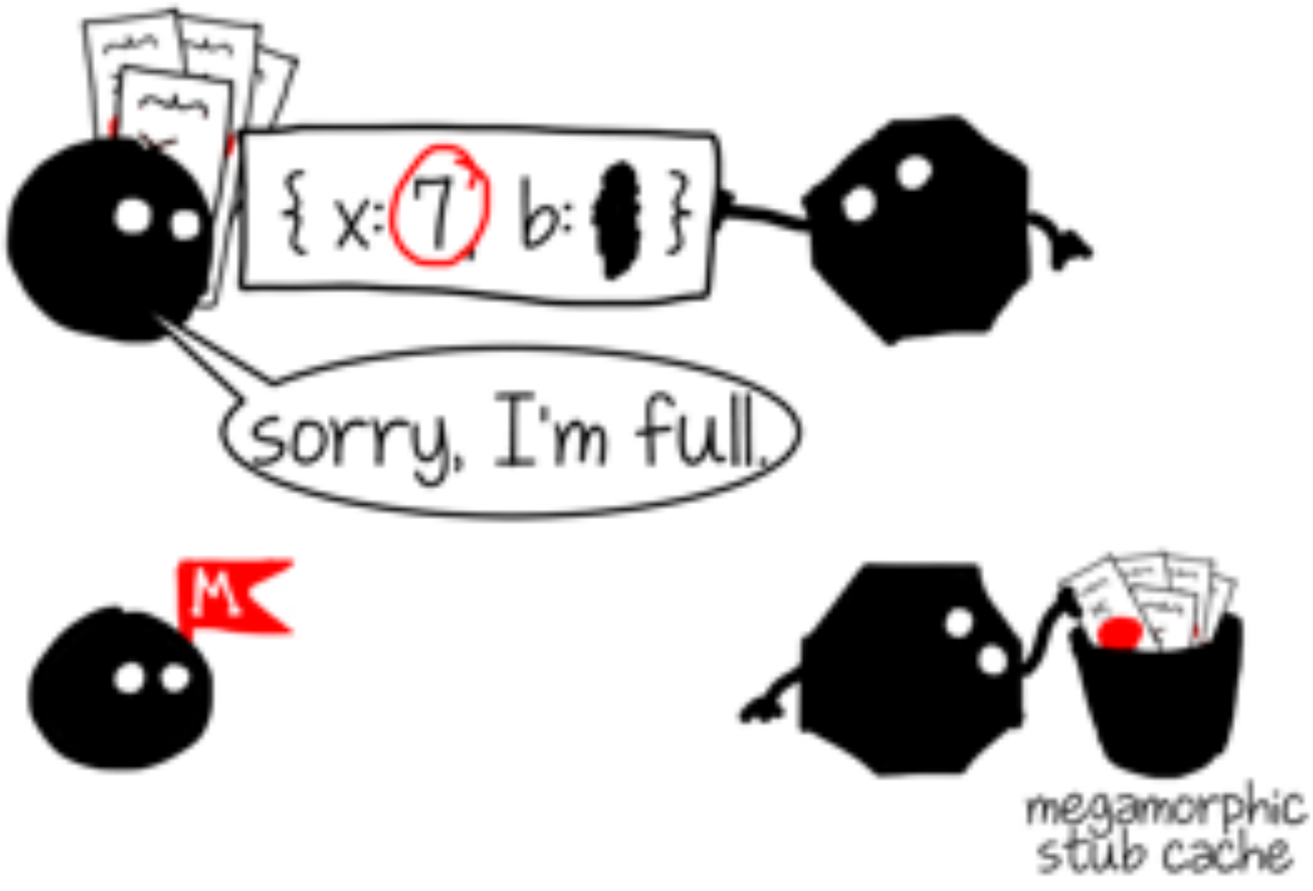


Figure 3. Polymorphic inline cache

Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches

Megamorphic Inline Cache

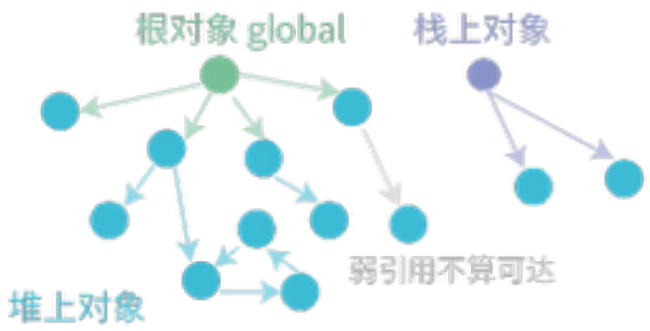
- 太多种例外情况出现了，怎么办?
  - 放到一个固定大小的 hash table
  - 放不下了就丢掉老的呗



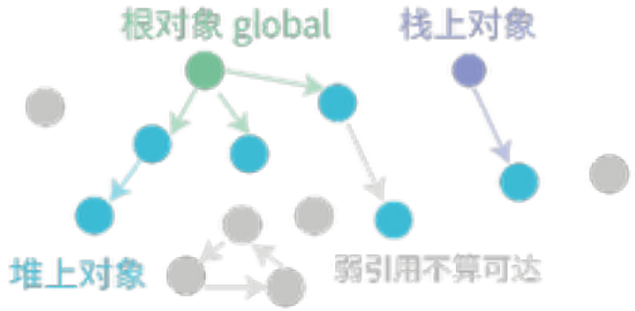
Credit: Vyacheslav Egorov

V8 Garbage Collection

初始状态



运行一段时间后

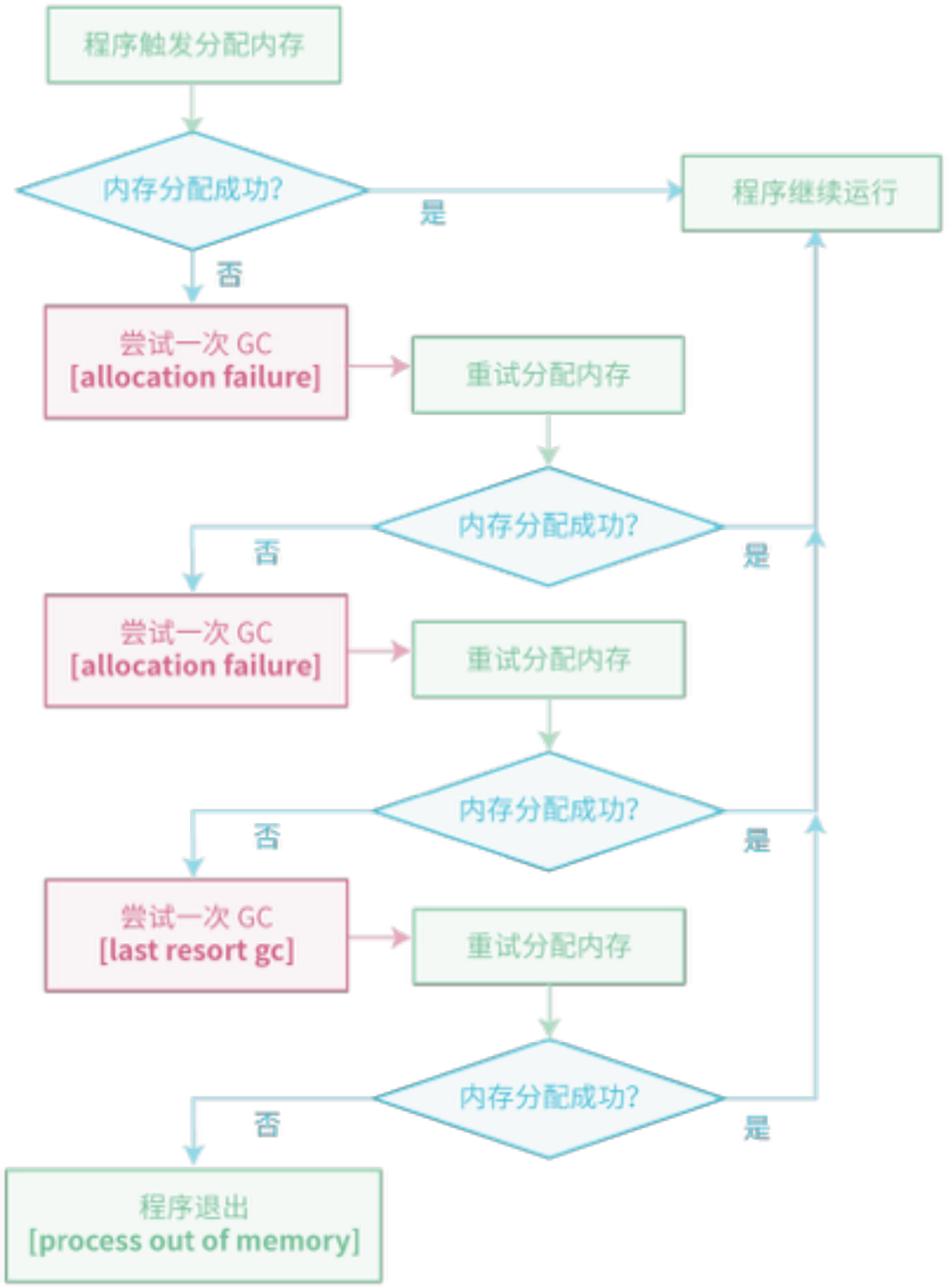


清空不可达对象


```
function foo() {
  var bar = global.bar = {
    a: 1
  };

  bar.x = { c: 3 };

  var baz = { b: 2 };
  ...
}
```

```
bar.x = null;
delete f;

var d = new SomeClass(10);
```

需要分配新对象，但是内存不够用了

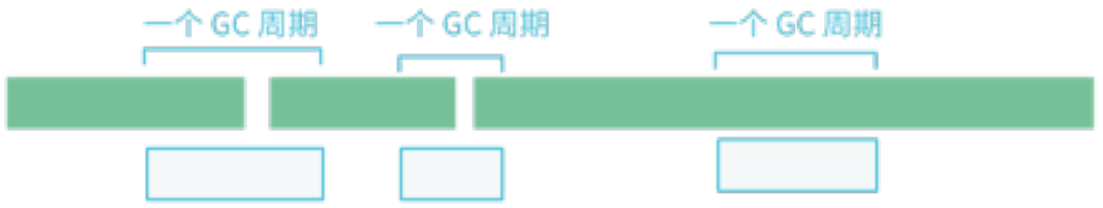


V8 Garbage Collection

GC 暂停的类型

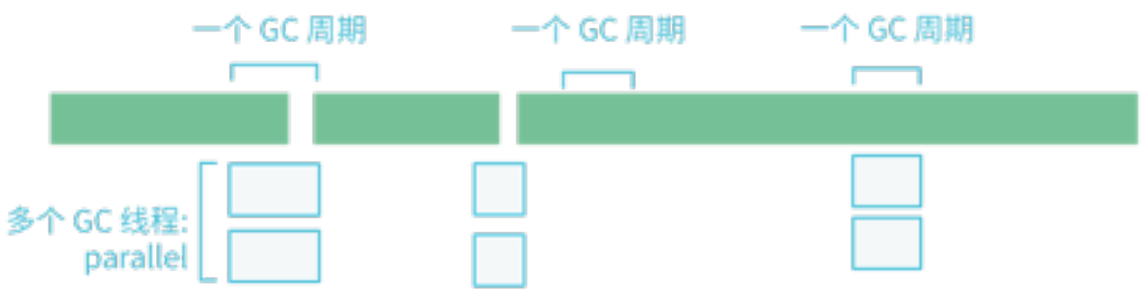
stop-the-world

并发式 (concurrent)



增量式 (incremental)

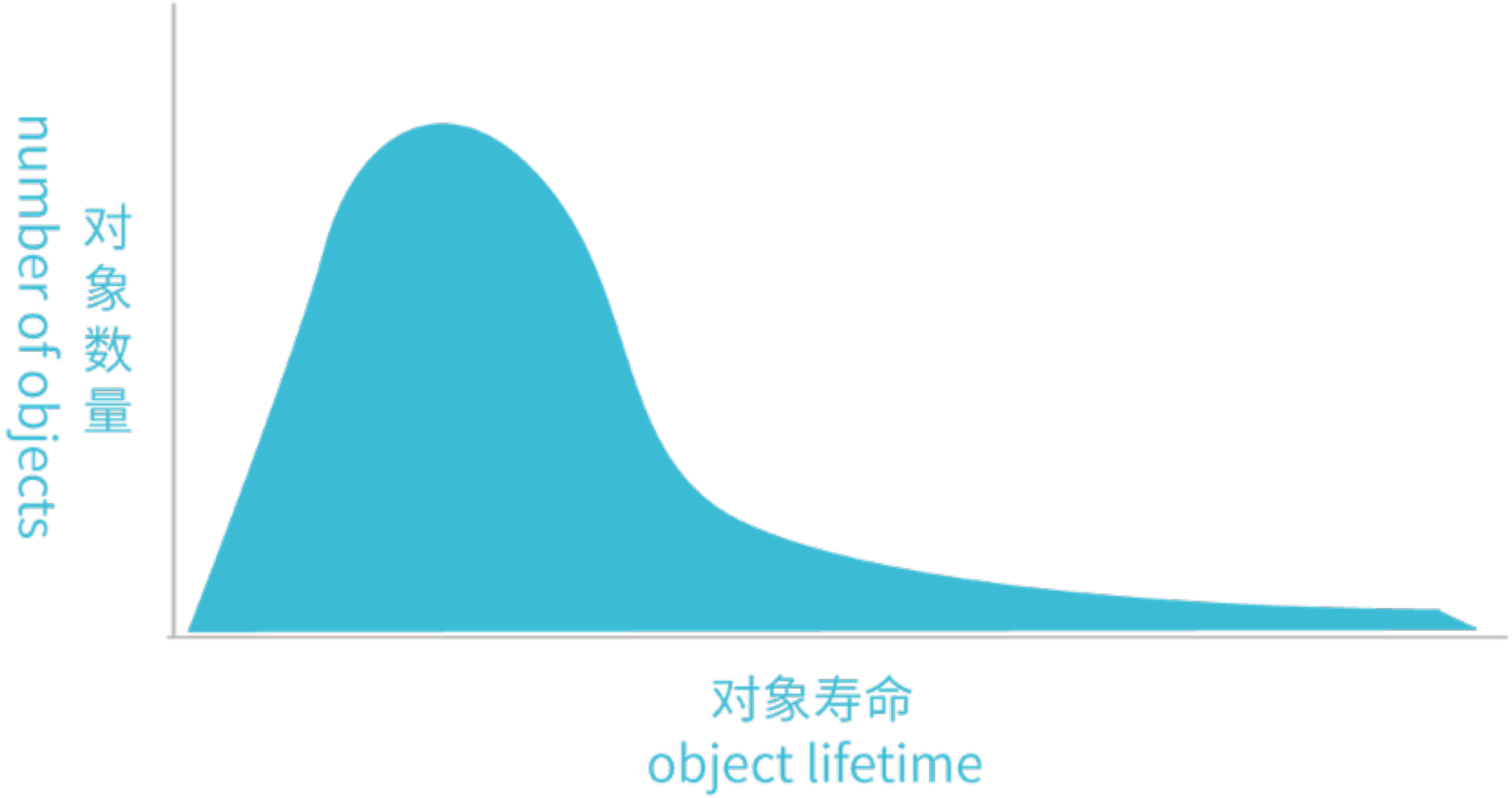
并行式(parallel, 假设也并发 conccurent)



V8 对 GC 的不同阶段采用不同的策略, 是上述几种类型的混合体

V8 Garbage Collection

弱分代假设



V8 Garbage Collection

小整数 (SMI) 结构

32 位系统, 字长 4 字节

整数	31 位有符号整数 (31-bit signed integer)	**0**

(无法用 31 位表示的整数将被封装为对象)

指针	地址	0	**1**

(因为 4 字节对齐的地址一定能被 4 整除, 倒数第二位是 0)

最后一位是 tag,
0 表示整数,
1 表示指针

64 位系统, 字长 8 字节

整数	32 位有号整数 (32-bit signed integer)	31 个 0	**0**

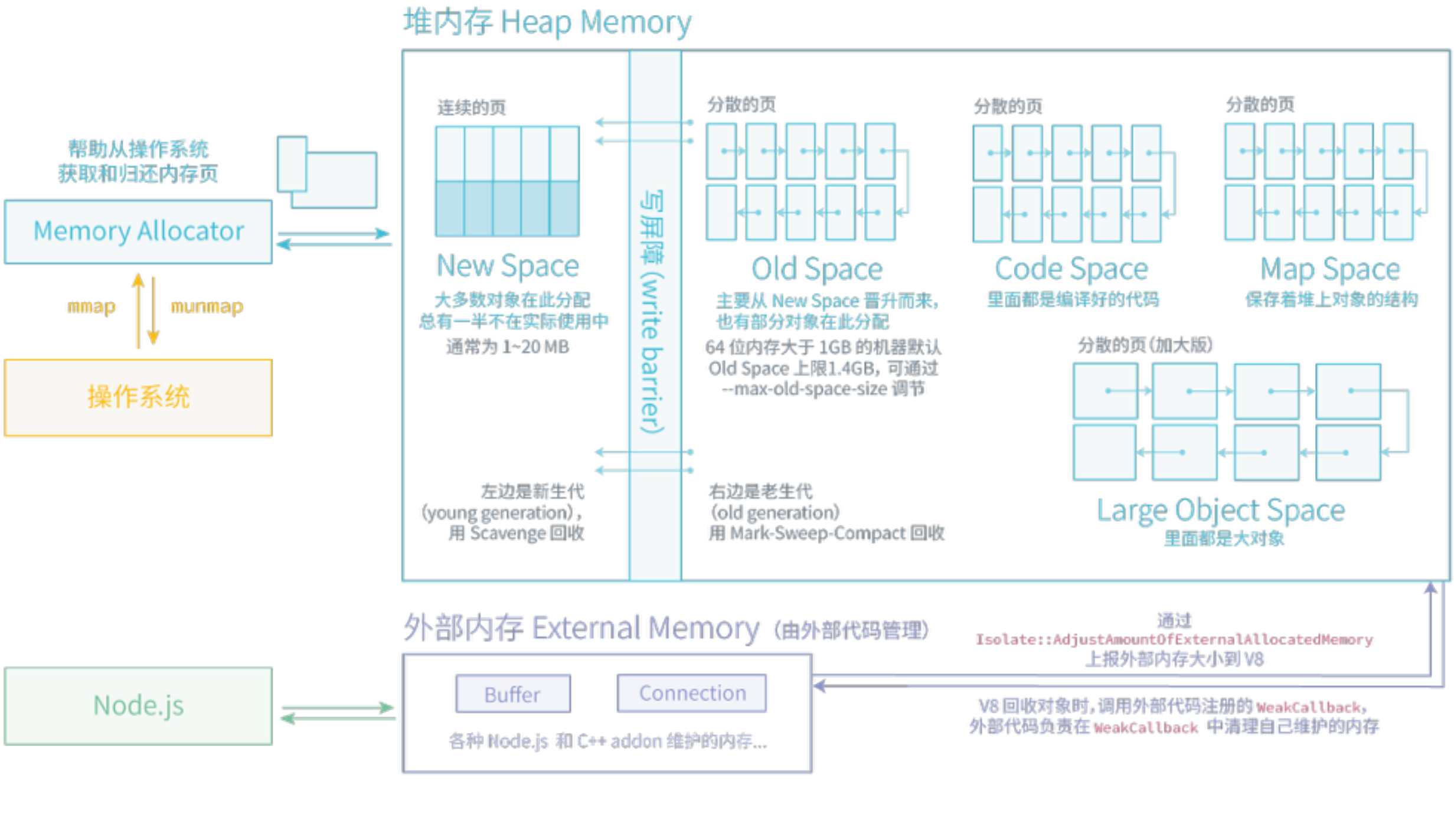
(无法用 32 位表示的整数将被封装为对象)

指针	地址	0	0	**1**

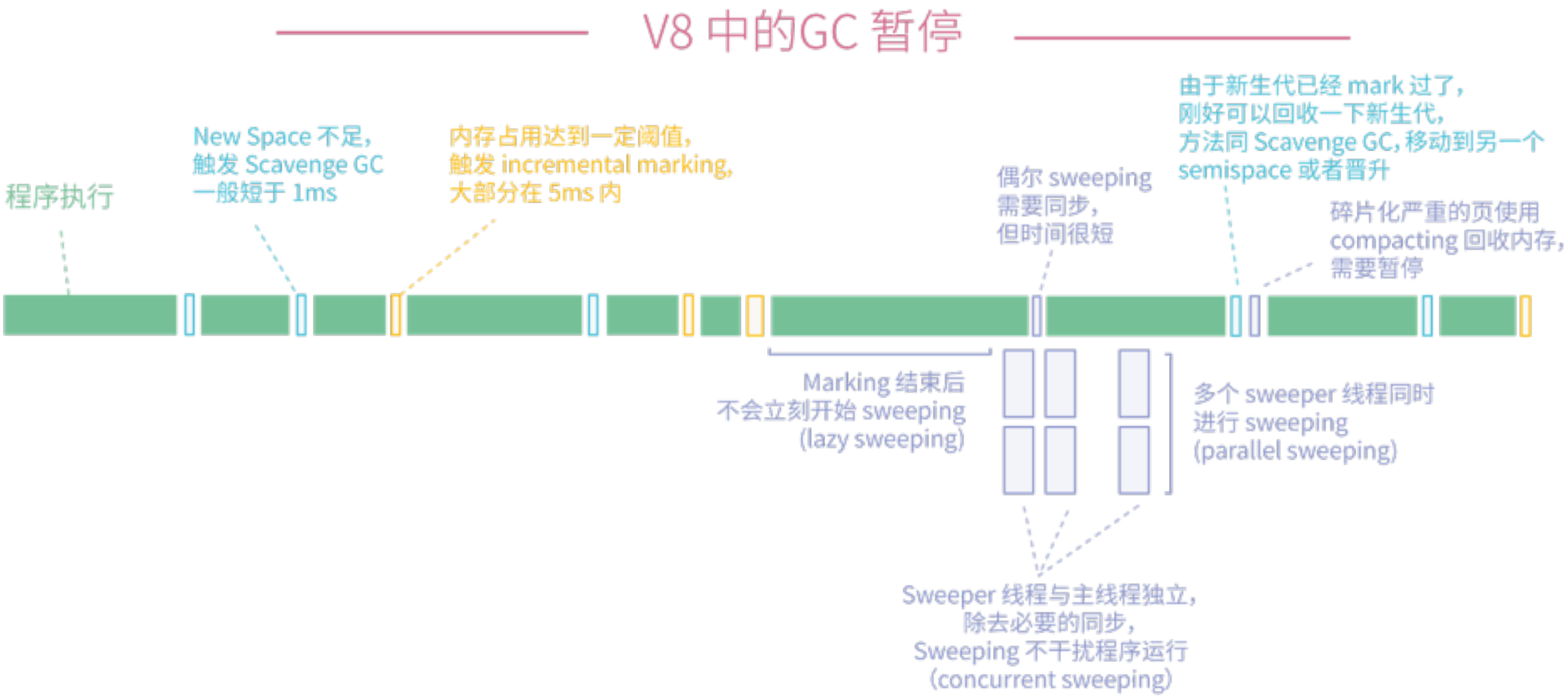
(因为 8 字节对齐的地址一定能被 8 整除, 倒数第二三位是 0)

V8 Garbage Collection

V8 堆内外内存分布

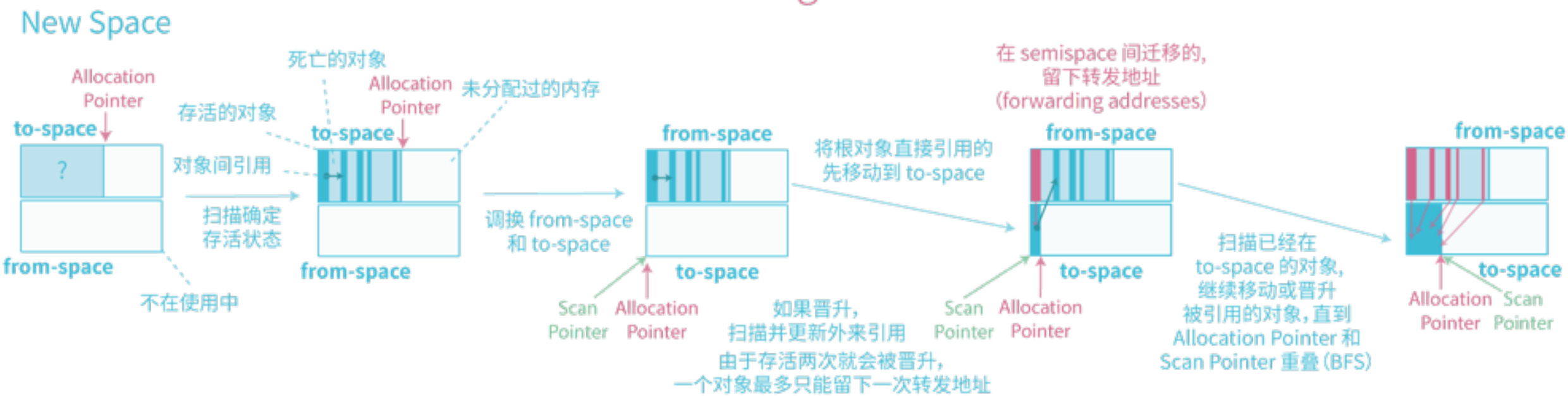


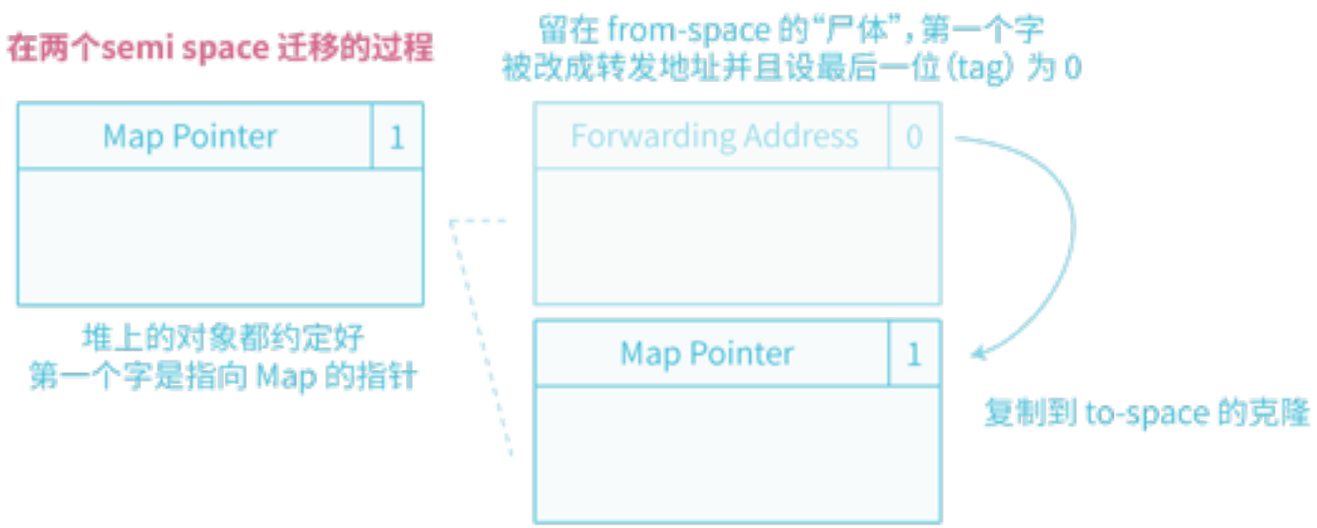
V8 Garbage Collection



V8 Garbage Collection

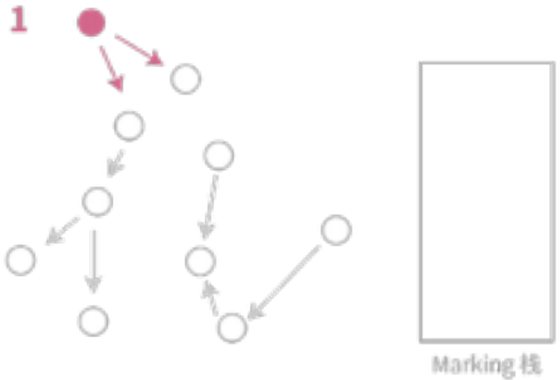
Scavenge GC



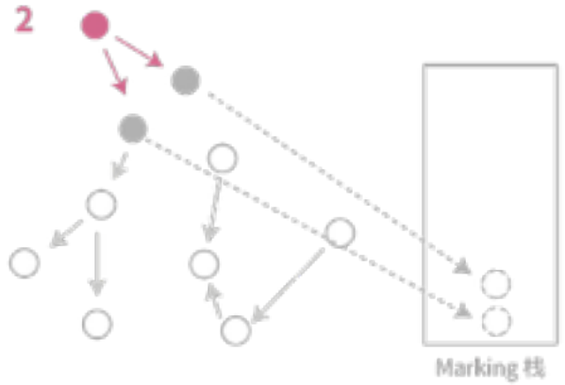


V8 Garbage Collection

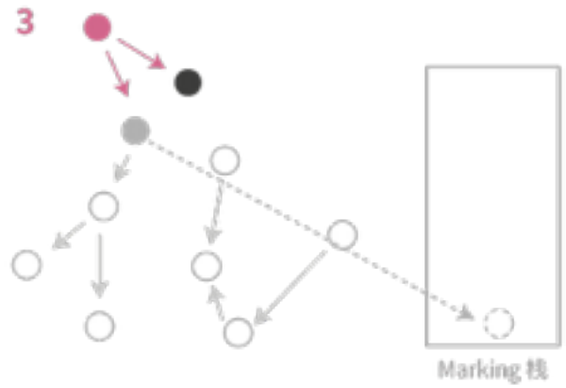
Tricolor Marking



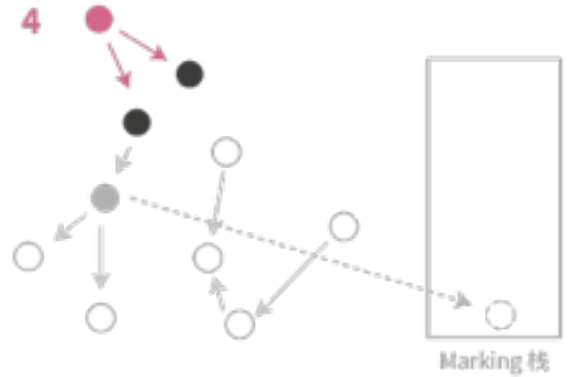
初始状态，所有非根对象都是白色



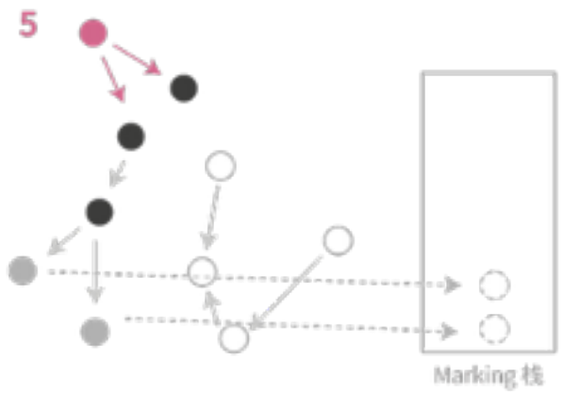
将根对象引用的对象标记为灰色，push 进栈



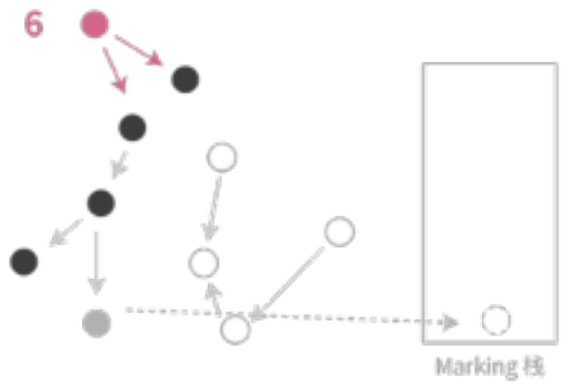
pop 一个对象出来，标记为黑，将它引用的
对象标记为灰色并 push 进栈



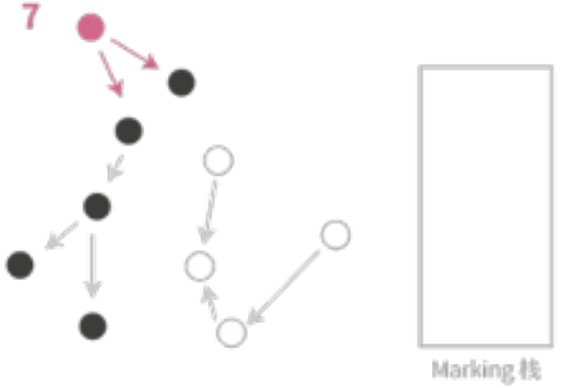
pop 一个对象出来，标记为黑，将它引用的
对象标记为灰色并 push 进栈



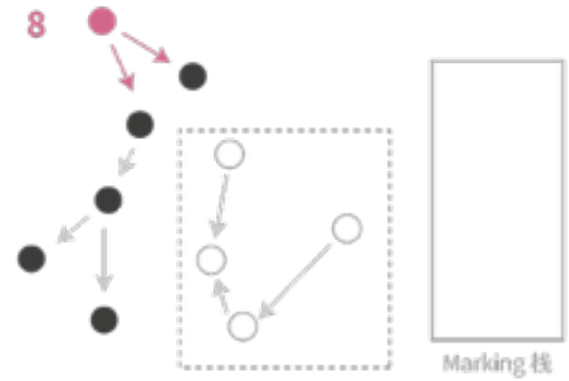
pop 一个对象出来，标记为黑，将它引用的
对象标记为灰色并 push 进栈



pop 一个对象出来，标记为黑，将它引用的
对象标记为灰色并 push 进栈



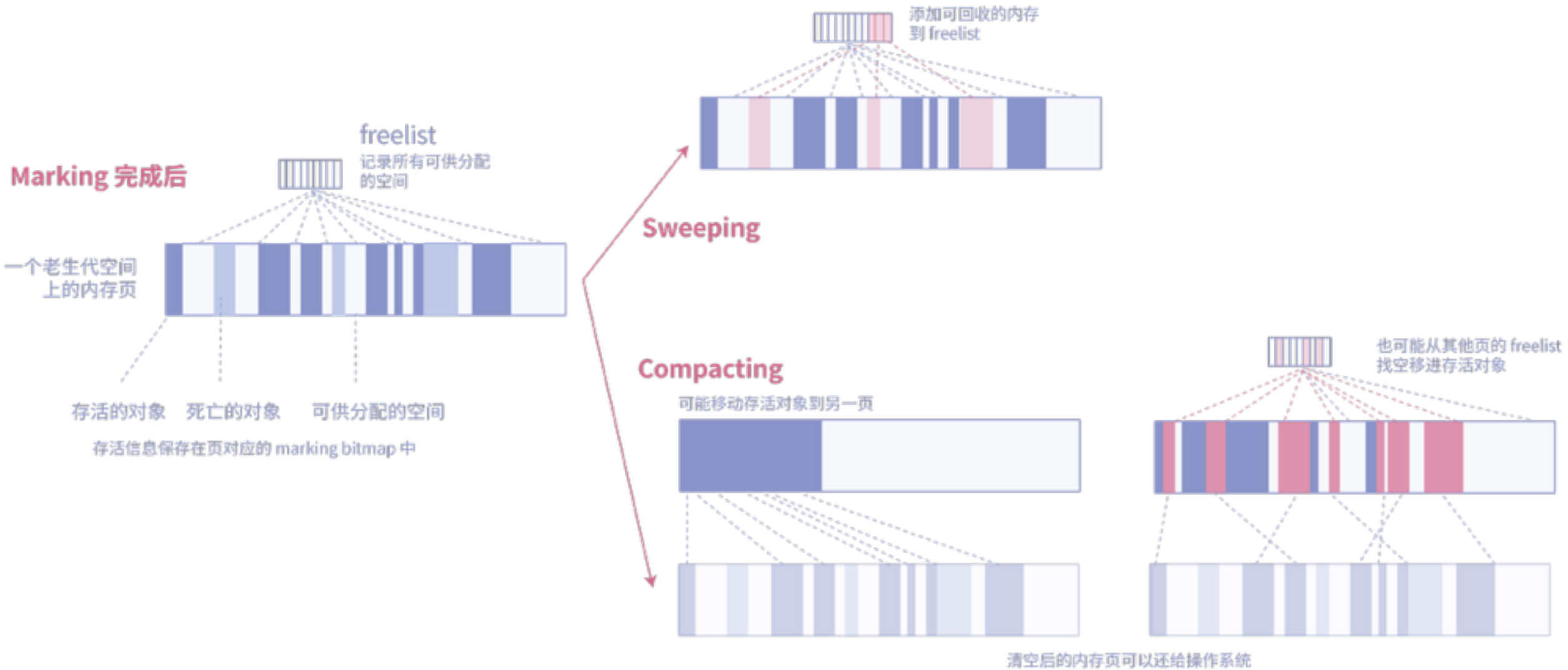
pop 一个对象出来，标记为黑，将它引用的
对象标记为灰色并 push 进栈



当栈清空的时候，剩下依然标记为白色的
对象就可以回收了

V8 Garbage Collection

Sweeping 与 Compacting



相关课程

- 操作系统：理解 libuv 和 V8 底层，文件系统，编译链接，进程线程, .etc
  - 课程没有讲 coroutine?
  - 建议做 PINTOS，有的老师会当作业
- 编译原理：理解 V8
  - 课程没有讲 JIT 编译和 GC？（可能后端一带而过）
    - 推荐选法师，推荐看斯坦福 Alex Aiken 的视频
      - 你知道 FJL 给 12 计应出的期末试卷就是斯坦福的试卷改改数据么?
- 计算机体系结构 / 计算机组成原理：汇编，理解 V8 底层优化
  - 建议阅读 CSAPP / CAAQA
    - CAAQA 好像是体系结构的课本?

相关课程

- 计算机网络：libuv，Node.js 底层
  - 建议课后自己写一个可以 CGI 的 HTTP Server 玩
    - http://tinyhttpd.sourceforge.net/ 代码风格诡异，凑合看看
    - HTTP 大部分就是个解析字符串的活，还特别暴力
    - 然后你就知道 Node.js 依赖的 http_parser 是干嘛的了
    - 进阶：用 epoll/IOCP 之类配合 Socket（UDP/TCP）和文件系统改进你的 server，然后你就知道 libuv 是干嘛的了
- Web 安全：crypto，OpenSSL
  - 把老蔡布置的所有作业都独立做掉，你就入门了
  - 上完你就知道 Node.js 的 crypto API 是干嘛用的，依赖的 OpenSSL 是干啥的了
  - 其实利用 buffer overflow 攻击写的 shell code 和 JIT 编译异曲同工

Q&A