V8：一些关键组件介绍

陆亚涵
yahan@iscas.ac.cn
2023.9.9

大纲

1. V8简介
2. JS编译
    a. Ignition
    b. Sparkplug
    c. Turbofan
3. wasm编译
    a. baseline
    b. Turbofan
4. Garbage Collector 垃圾回收
5. 其他优化技术
    a. pointer compression
    b. Inline Cache

1 V8简介

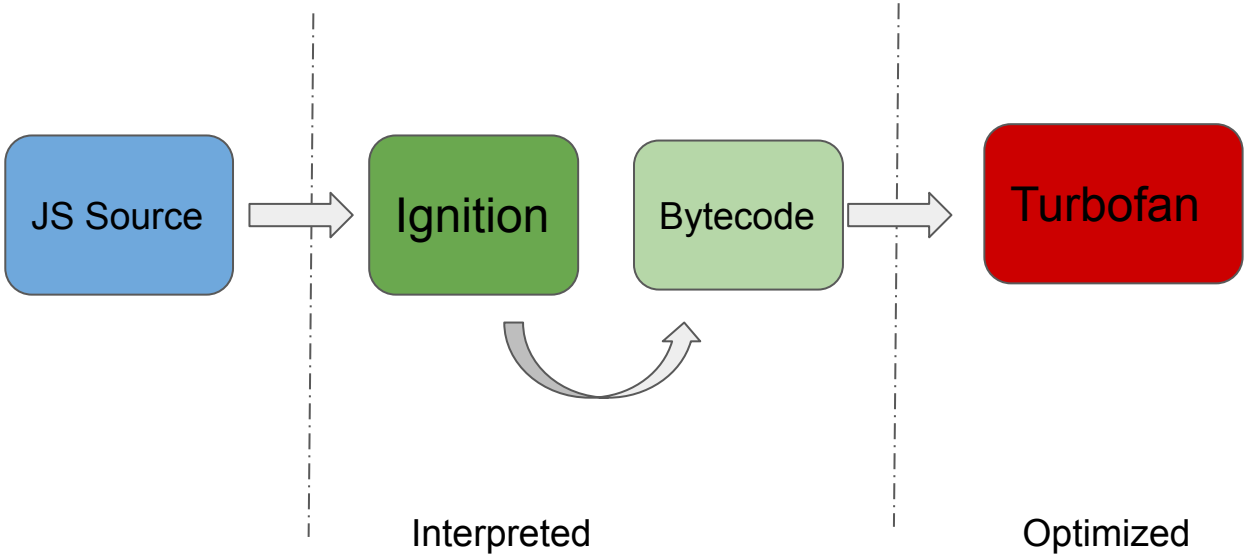
V8是一种JavaScript虚拟机,内嵌于Chrome浏览器当中，主要功能是帮助网页开发者实现动态网页、网页渲染等功能。除了浏览器中，V8也被内置于node.js,用于服务器程序的开发。

V8作为一种动态语言的虚拟机，它具备AOT编译、JIT编译、垃圾回收等主要功能。除此之外，还加入指针压缩、SMI、InlineCache等优化技术，用来改善浏览器内存占用、网页浏览速度等。

大纲

1. V8简介
2. JS编译
    a. Ignition
    b. Sparkplug
    c. Turbofan
3. wasm编译
    a. baseline
    b. Turbofan
4. Garbage Collector 垃圾回收
5. 其他优化技术
    a. pointer compression
    b. Inline Cache

2. V8中的JS编译
2.1 Pipeline

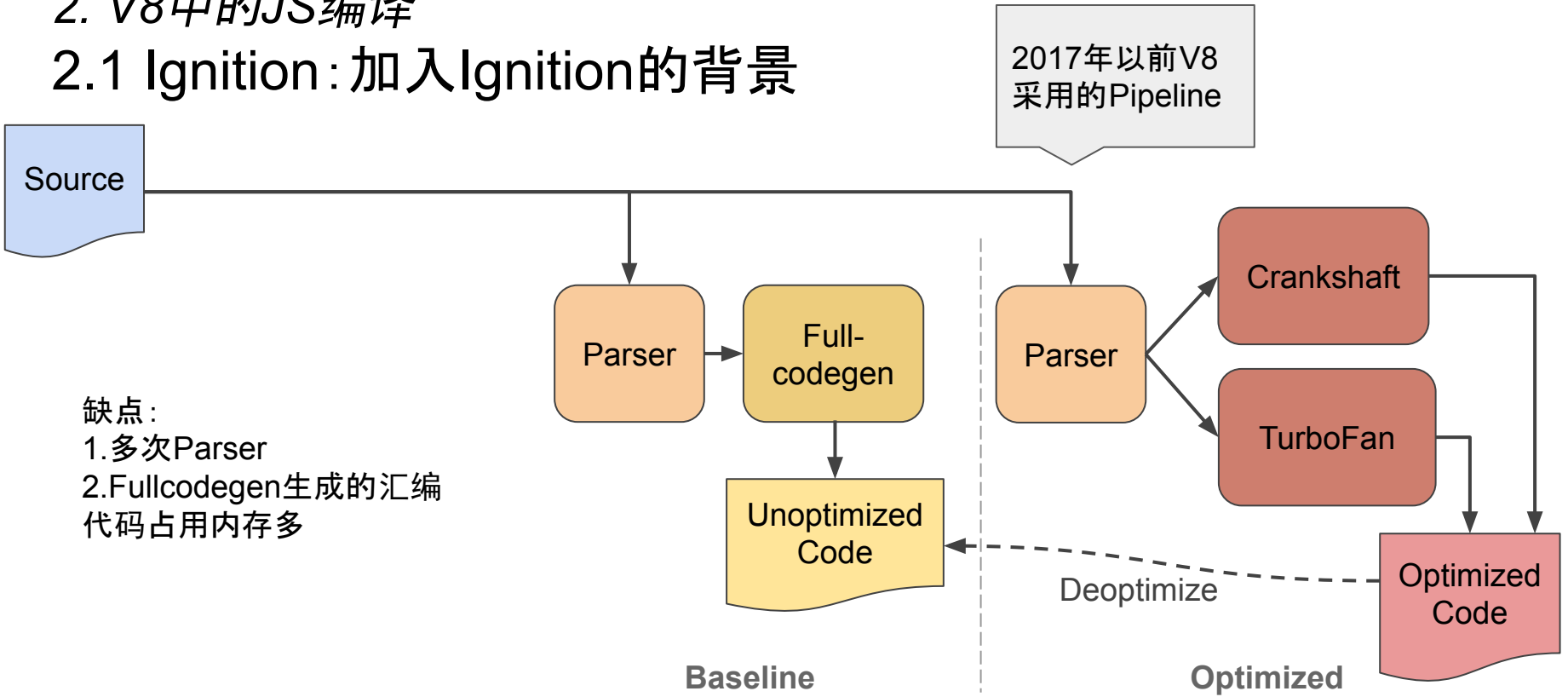


优点

- 减少内存使用
  - *代码被编译到字节码而不是汇编代码*
- 减少 parsing 的次数
  - Bytecode十分简洁
- 减少compiler pipeline复杂度

2. V8中的JS编译

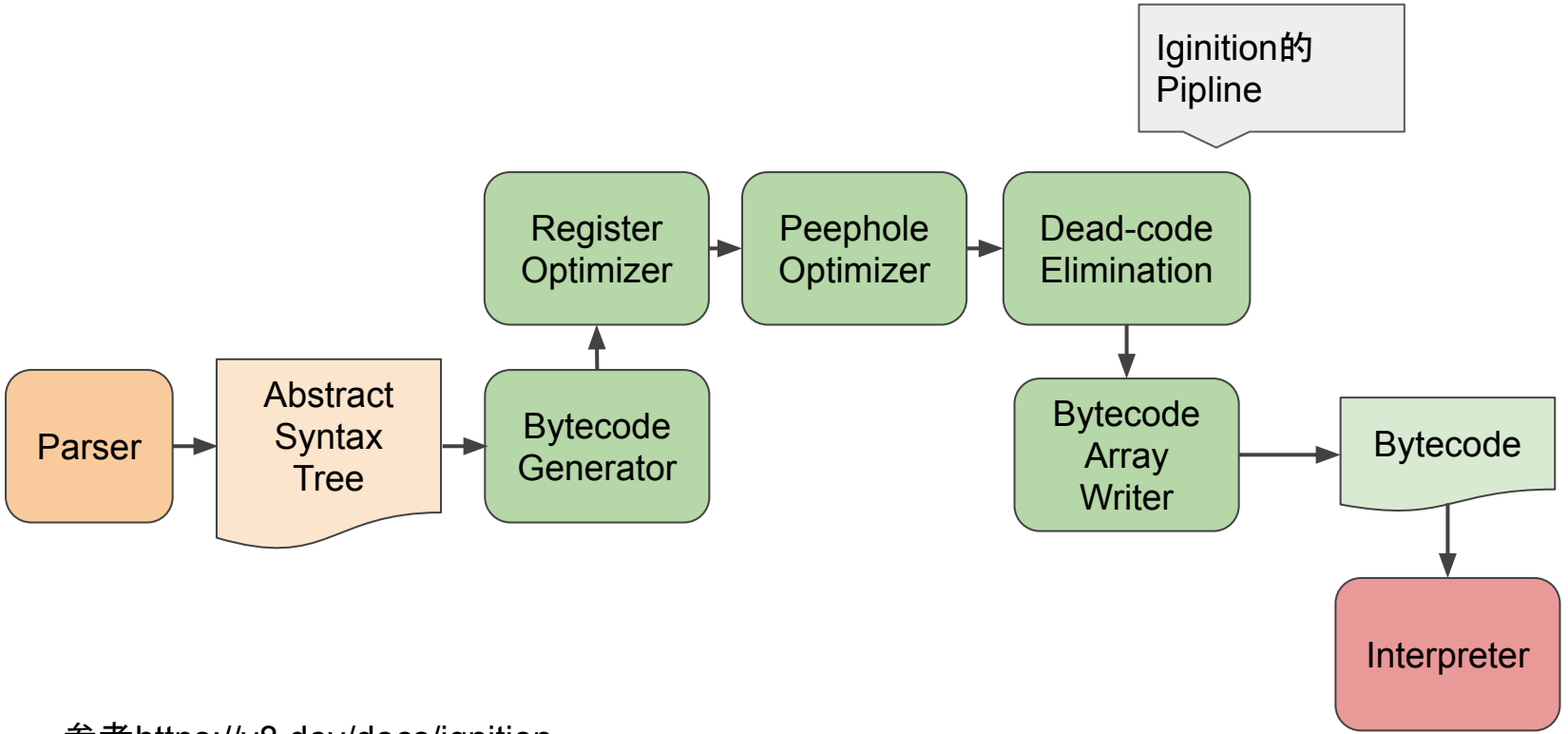
2.1 Ignition：加入Ignition的背景



缺点：
1.多次Parser
2.Fullcodegen生成的汇编
代码占用内存多

参考https://v8.dev/docs/ignition

2. V8中的JS编译 2.1 Ignition：从JS Source code到bytecode



参考https://v8.dev/docs/ignition

大纲

1. V8简介
2. JS编译
    a. Ignition
    b. Sparkplug
    c. Turbofan
3. wasm编译
    a. baseline
    b. Turbofan
4. Garbage Collector 垃圾回收
5. 其他优化技术
    a. pointer compression
    b. Inline Cache

2. V8中的JS编译 2.2 Sparkplug：重新引入Baseline

Ignition无法优化的步骤：1.Bytecode解析；2. Bytecode调度

1. 解析Bytecode需要时间，且无法被Ignition优化掉
2. 解释器阻止了CPU本身的优化技术：两个bytecode之间CPU不能进行分支预测和预取

参考https://v8.dev/blog/sparkplug

2. V8中的JS编译 2.2 Sparkplug：重新引入Baseline

The new compiler pipeline

2021年重新引入Baseline编译器。

设计目标：编译速度快
特点：

1. 无优化直接生成汇编代码
2. 调用堆栈兼容Igniton
3. 大部分Sparkplug的汇编代码就是调用Builtins函数，这些函数跟Ignition所用的函数一样。
4. Sparkplug所做的工作仅仅施救call Builtins 和生成汇编形式的控制流

V8团队认为：CPU本身就是一种解释器，以此看来，Sparkplug就是一种把Ignition bytecode翻译到CPU bytecode的翻译器，将代码从模拟器运行编程native运行。

参考https://v8.dev/blog/sparkplug

大纲

1. V8简介
2. JS编译
    a. Ignition
    b. Sparkplug
    c. Turbofan
3. wasm编译
    a. baseline
    b. Turbofan
4. Garbage Collector 垃圾回收
5. 其他优化技术
    a. pointer compression
    b. Inline Cache

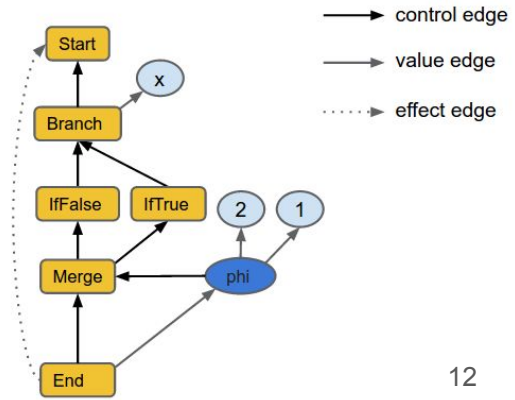
2. V8中的JS编译 2.2 Turbofan

2017年跟Ignition一起引入的新的优化编译器，主要特点：

1. 采用Sea of Nodes IR
   可以 more aggressive optimizations than CrankShaft through a number of advanced techniques

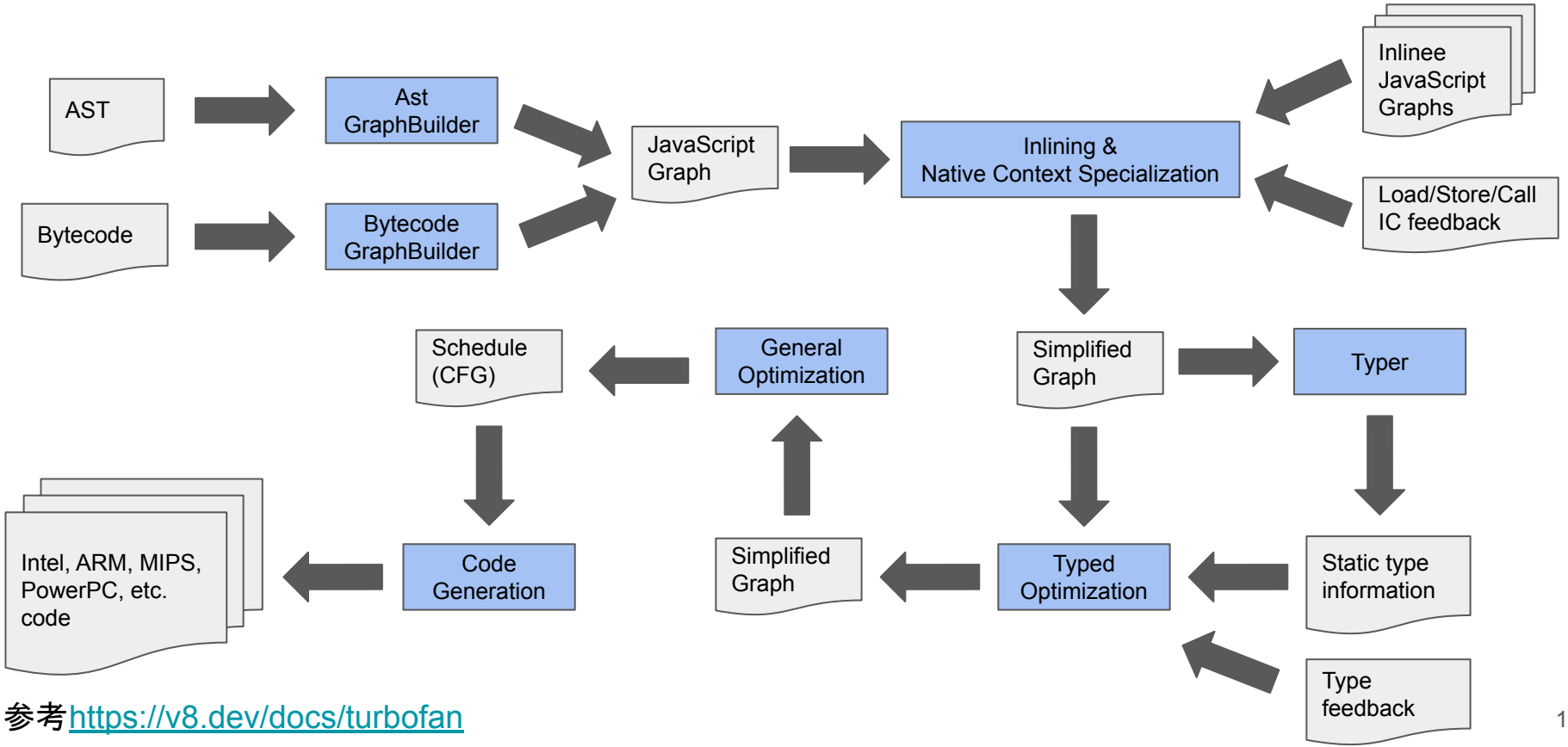
2. 层级架构，便于移植到不同平台
   a. SON IR: js operator/simplified operator/common operator/machine operator
   b. 后端：arch opcode

function (x) { return x ? 1 : 2; }



参考https://v8.dev/docs/turbofan

2. V8中的JS编译 2.2 Turbofan：一种优化编译器



参考https://v8.dev/docs/turbofan

大纲

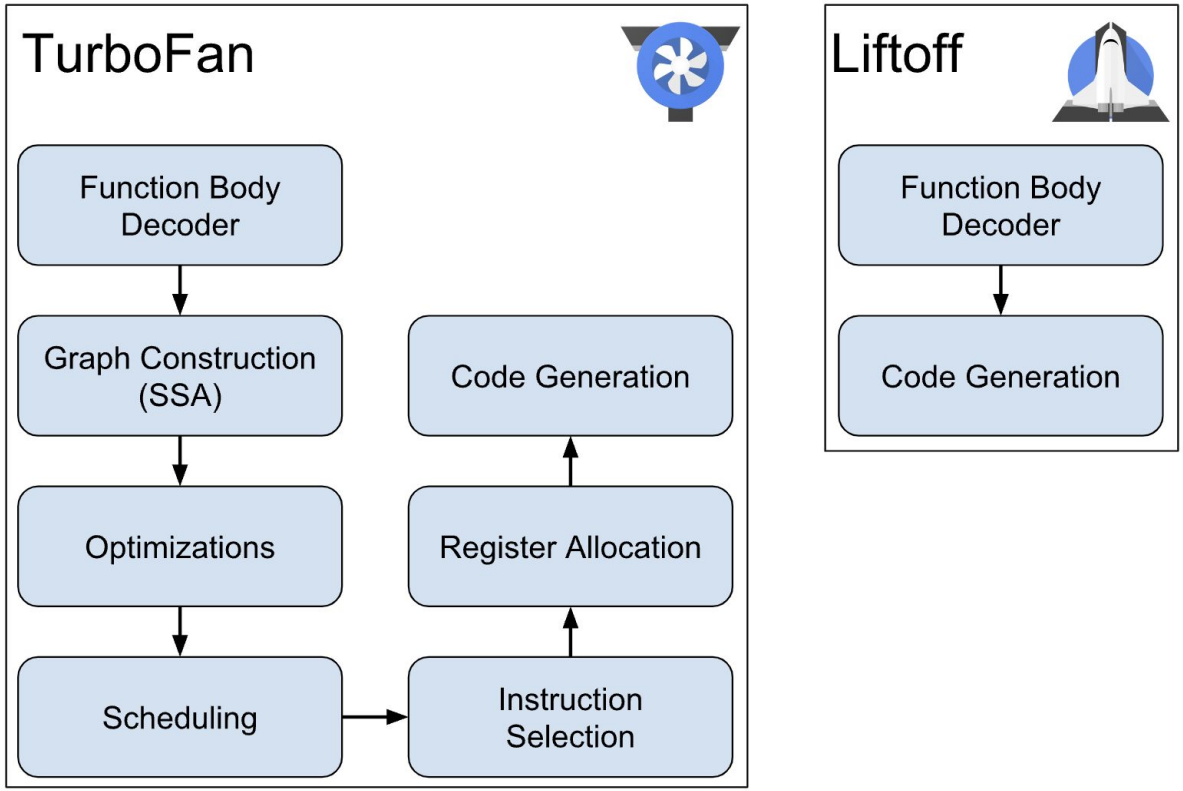
1. V8简介
2. JS编译
    a. Ignition
    b. Sparkplug
    c. Turbofan
3. wasm编译
    a. baseline
    b. Turbofan
4. Garbage Collector 垃圾回收
5. 其他优化技术
    a. pointer compression
    b. Inline Cache

3 WASM 编译

WASM 是一种低级编程语言，被设计来提供比JavaScript更快速的编译及执行。WebAssembly将让开发者能运用自己熟悉的编程语言编译，再由引擎在浏览器内执行。

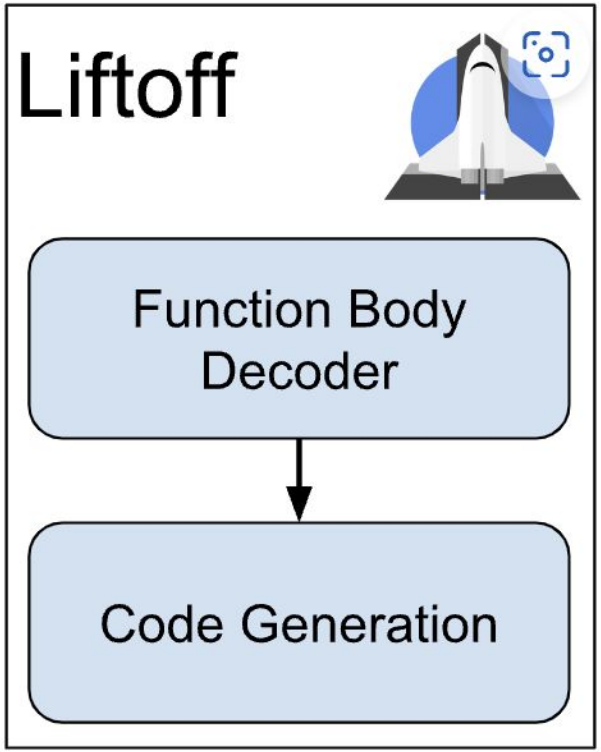
3 WASM 编译:编译架构

2021年加入



参考 https://v8.dev/blog/liftoff

3 WASM 编译:不同编译器的特点

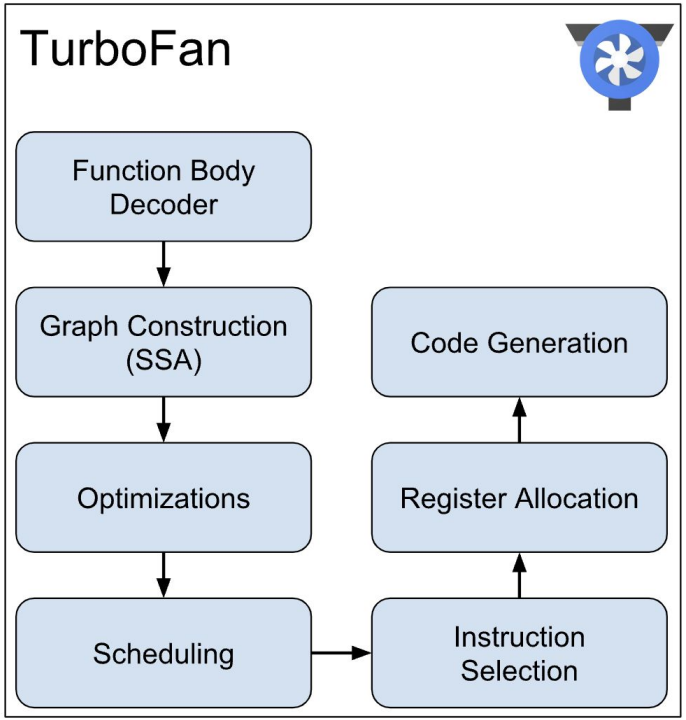


特点:

1. 一次编译
2. 不做优化，直接生成汇编

参考 https://v8.dev/blog/liftoff

3 WASM 编译:不同编译器的特点



特点:

1. 只编译热点代码
2. 多种复杂优化

参考 https://v8.dev/blog/liftoff

3 WASM 编译: 两种编译器取舍策略

WASM权衡两种编译器所采用策略: ***eager tier-up***

liftoff编译完成后,就可以运行WASM, 同时Turoban编译WASM代码

原因:

1. WASM的变量是静态的, 不像JS是动态类型
2. WASM已经试过WASM编译器优化后的二进制代码, V8团队认为已经可以被预测性的运行且速度很快, 不能接受暂停执行WASM然后去编译
3. V8团队尝试为WASM构建了一个解释器, 但解释器执行比liftoff慢20倍。

考虑以上原因, V8团队认为wasm的汇编代码必须被存储, 那就存储最紧凑、执行最快的代码。

参考 https://v8.dev/blog/liftoff

大纲

1. V8简介
2. JS编译
    a. Ignition
    b. Sparkplug
    c. Turbofan
3. wasm编译
    a. baseline
    b. Turbofan
4. Garbage Collector 垃圾回收
5. 其他优化技术
    a. pointer compression
    b. Inline Cache

4. V8垃圾回收器---Orinoco

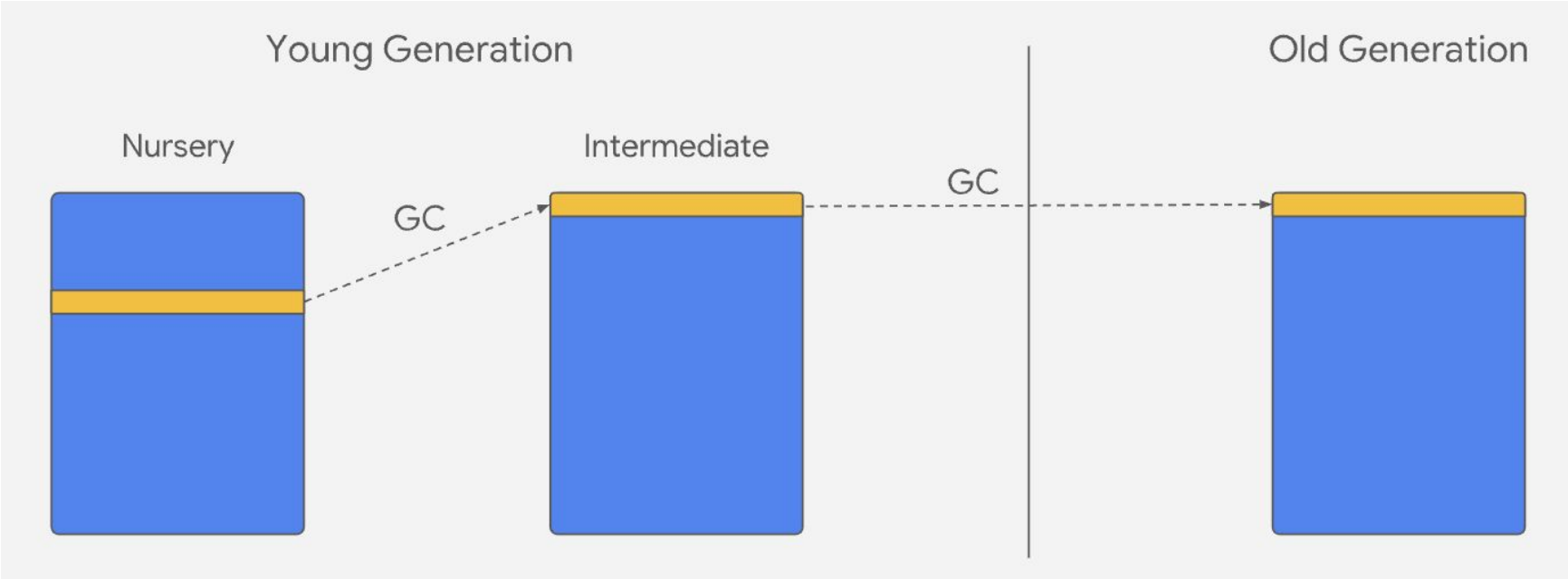
垃圾回收的主要任务:

1. 识别存活对象和死亡对象
2. 回收并重新利用死亡对象所有的内存空间
3. 压缩内存和内存碎片整理(可选)

参考 https://v8.dev/blog/trash-talk

4. V8垃圾回收器---Orinoco

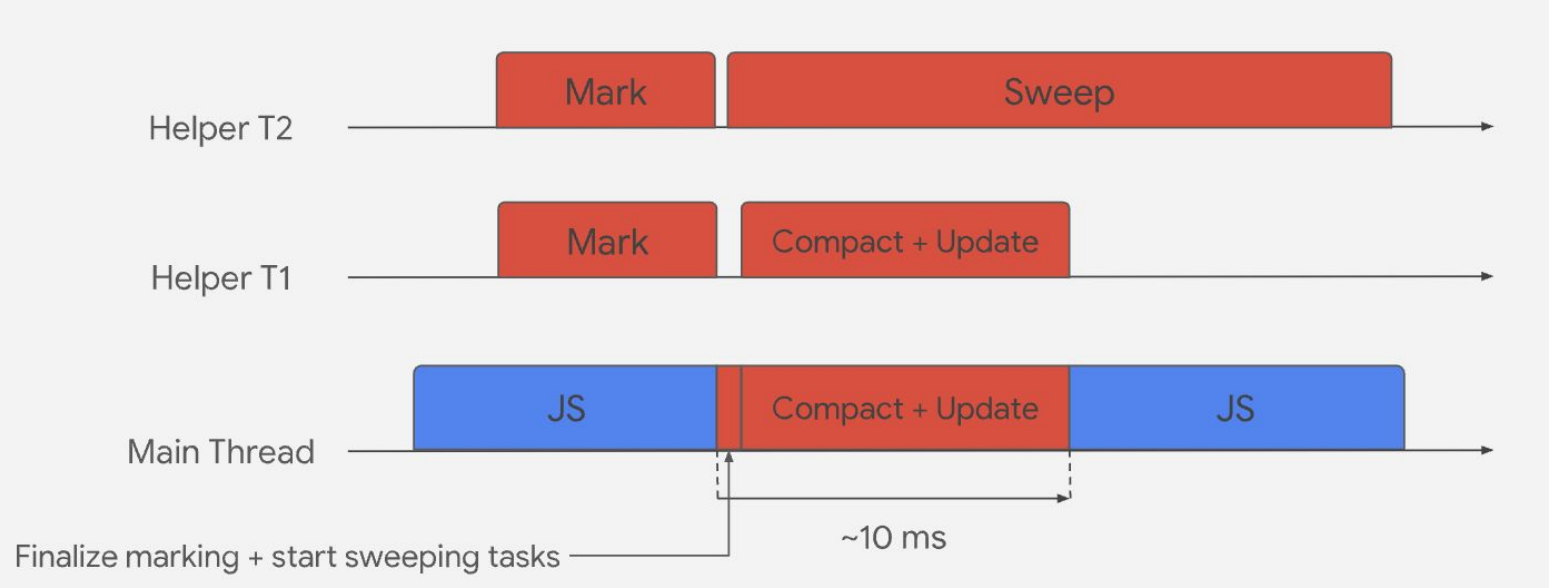
堆内存空间分代管理



参考 https://v8.dev/blog/trash-talk

4. V8垃圾回收器---Orinoco

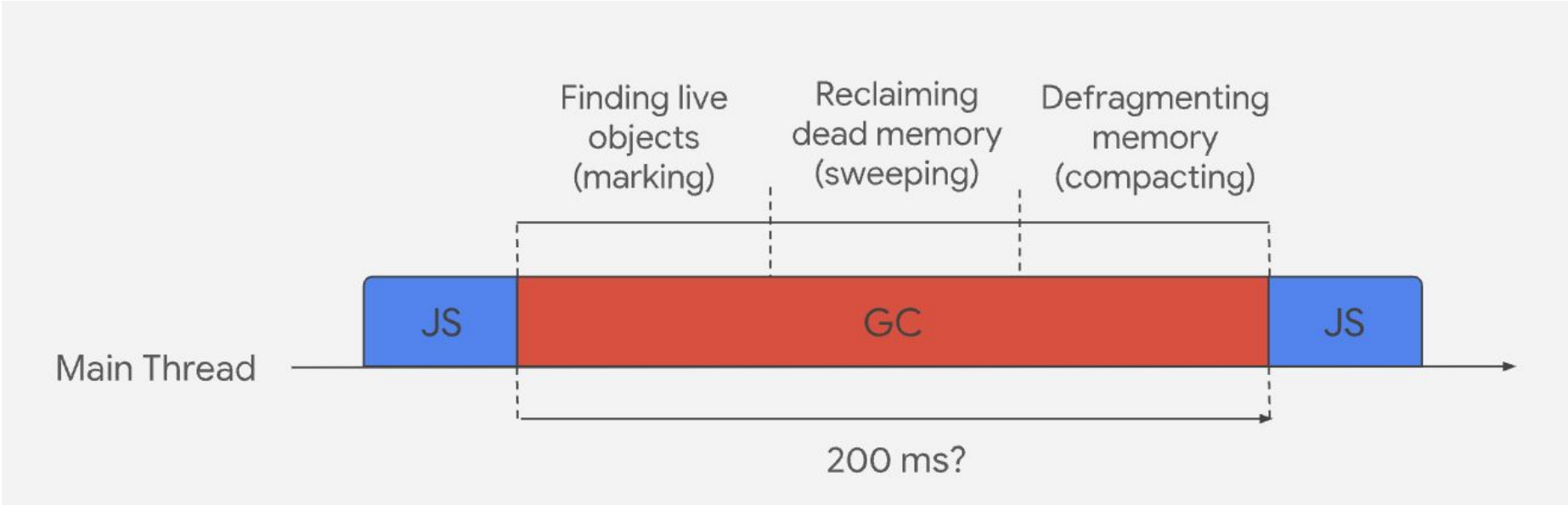
Orinoco 采用并行进行垃圾回收



参考 https://v8.dev/blog/trash-talk

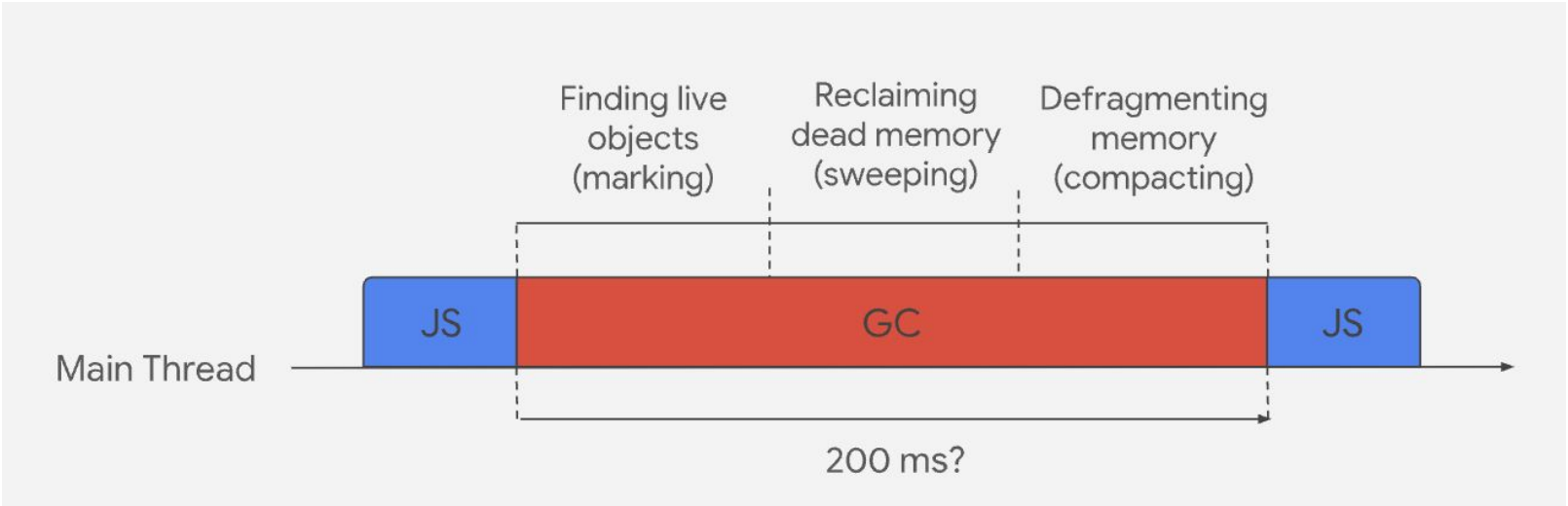
4. V8垃圾回收器---Orinoco

单线程垃圾回收过程



参考 https://v8.dev/blog/trash-talk

4. V8垃圾回收器---Orinoco



主线程被GC占用，会造成浏览器渲染延迟、页面效果差

参考 https://v8.dev/blog/trash-talk

大纲

1. V8简介
2. JS编译
    a. Ignition
    b. Sparkplug
    c. Turbofan
3. wasm编译
    a. baseline
    b. Turbofan
4. Garbage Collector 垃圾回收
5. 其他优化技术
    a. Pointer Tag 与 Pointer Compression
    b. Inline Cache

4.1 其他优化技术 Pointer Tag

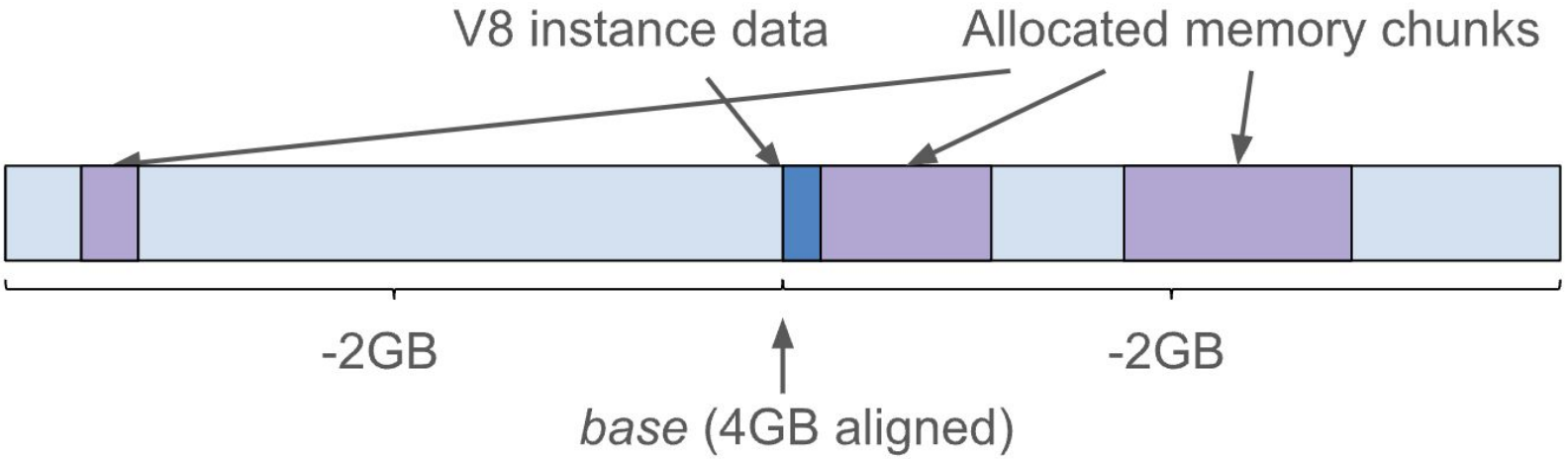
```
                          |----- 32 bits -----|
Pointer:                  |_____address_____w1|
Smi:                      |___int31_value____0|
```

```
              |----- 32 bits -----|----- 32 bits -----|
Pointer:      |_________________address_______________w1|
Smi:          |____int32_value____|00000000000000000000|
```

参考https://v8.dev/blog/pointer-compression

4.1 其他优化技术 Pointer Compression

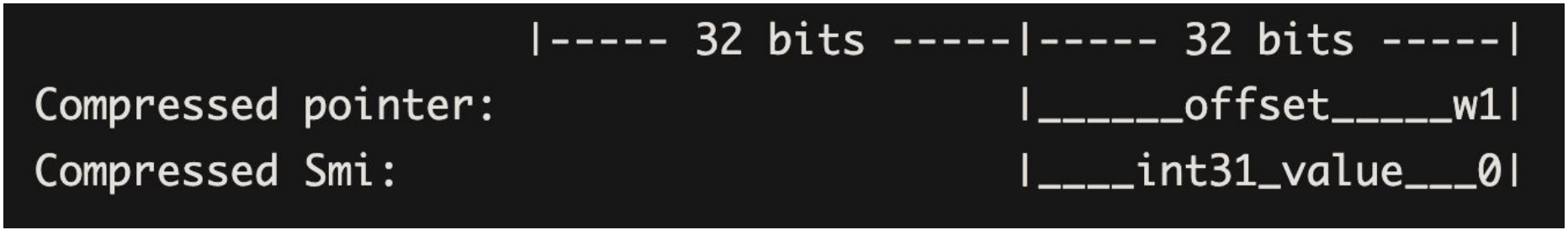
2. base into mid of 4GB



参考https://v8.dev/blog/pointer-compression

4.1 其他优化技术 Pointer Compression

分配后:



参考https://v8.dev/blog/pointer-compression

4. 1 其他优化技术 Pointer Compression

2. base into mid of 4GB

```
int32_t compressed_tagged;

// Common code for both pointer and Smi cases
int64_t uncompressed_tagged = int64_t(compressed_tagged);
if (uncompressed_tagged & 1) {
  // pointer case
  uncompressed_tagged += base;
}
```

解压缩过程变得简单

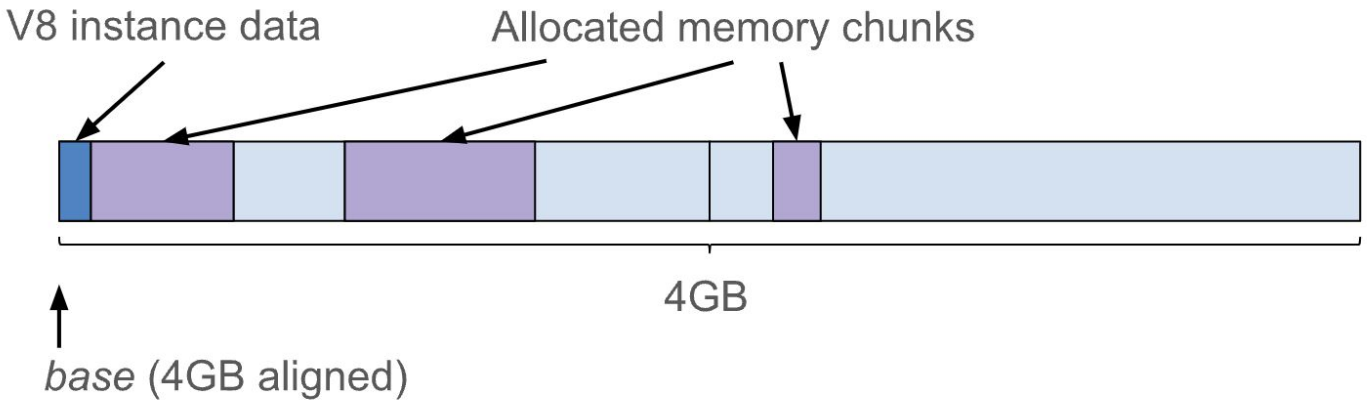
参考https://v8.dev/blog/pointer-compression

4.1 其他优化技术 Pointer Compression

64Bit 架构中 一个指针占用用了8字节空间

V8提出将可分配的内存空间控制在4GB内

1. base into beginning



参考https://v8.dev/blog/pointer-compression

4. 1 其他优化技术 Pointer Compression

分配后:

```
uint32_t compressed_tagged;

uint64_t uncompressed_tagged;
if (compressed_tagged & 1) {
  // pointer case
  uncompressed_tagged = base + uint64_t(compressed_tagged);
} else {
  // Smi case
  uncompressed_tagged = int64_t(compressed_tagged);
}
```

参考https://v8.dev/blog/pointer-compression

大纲

1. V8简介
2. JS编译
    a. Ignition
    b. Sparkplug
    c. Turbofan
3. wasm编译
    a. baseline
    b. Turbofan
4. Garbage Collector 垃圾回收
5. 其他优化技术
    a. Pointer Tag 与 Pointer Compression
    b. Inline Cache

4.2 其他优化技术 Inline Cache

隐藏类

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

  var obj = new
  Point(1,2);
```

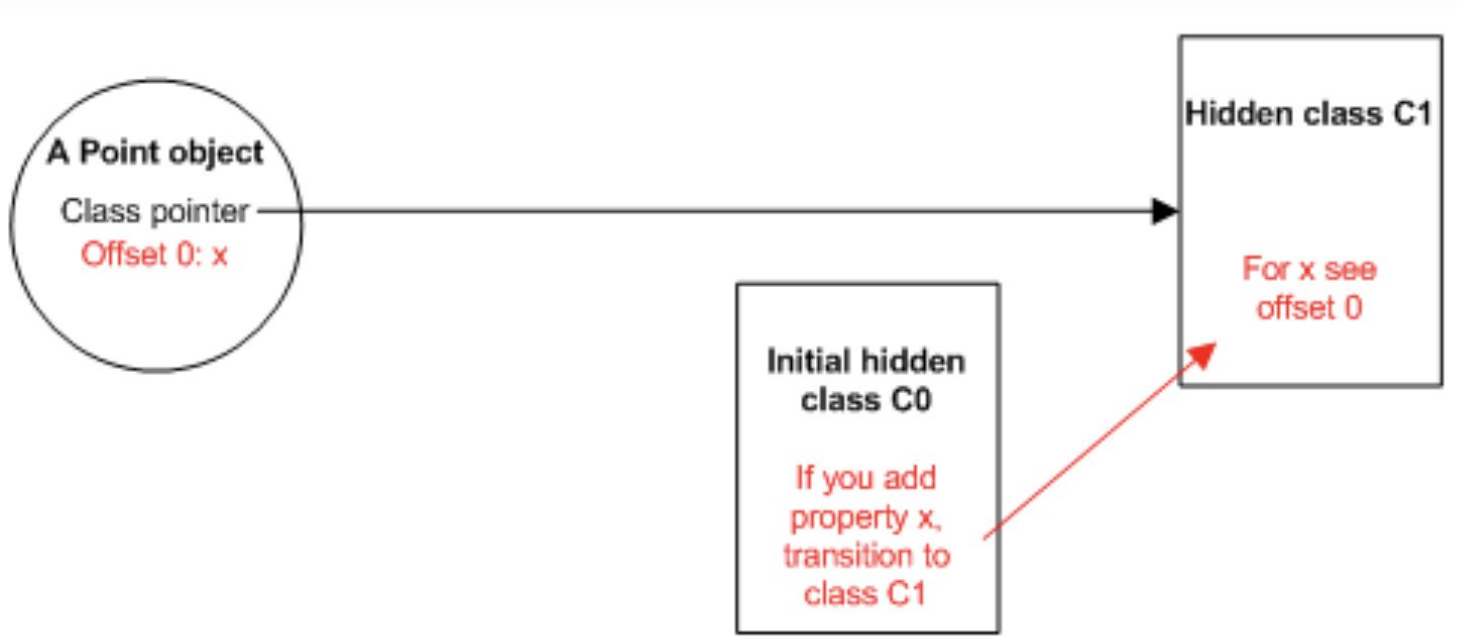
参考 https://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html

4.2 其他优化技术 Inline Cache

隐藏类

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

  var obj = new
  Point(1,2);
```



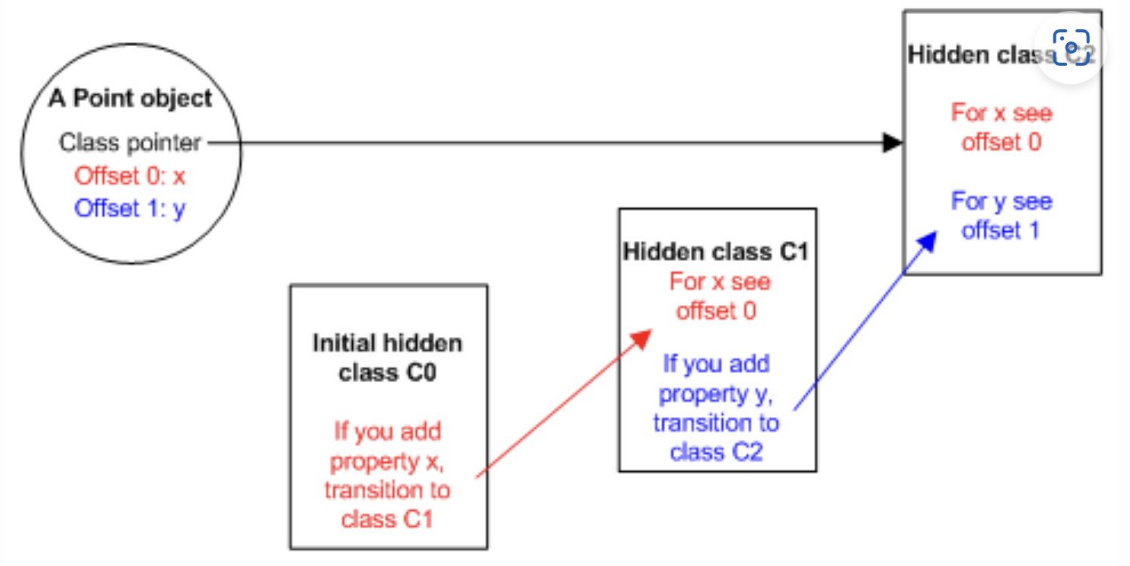
参考 https://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html

4.2 其他优化技术 Inline Cache

隐藏类

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

  var obj = new
  Point(1,2);
```



参考 https://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html

4.2 其他优化技术 Inline Cache

基于以上隐藏类，v8为了快速获取对象的属性，采用了IC

原理：

1. 第一次执行时直接将对应属性的地址记录在Cache中，并在后续被编译器直接采用并生成来生成汇编
2. 第二次执行时，V8的JIT代码将会判断对象的隐藏类是否被改变，如果没有被改变V8将直接利用之前的信息执行代码
3. 如果对象被改变，V8将会执行de-optimize过程，来重新获取offset

参考 https://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html

总结

1. 介绍了V8三种编译器（Interpreter、Baseline、Turbofan）及其特点
2. 介绍了V8所用的垃圾回收技术
3. 介绍了V8的Pointer Compression 和 IC 技术

谢谢！