JavaScript 引擎基础入门

@hijiangtao

目录

1. 引擎概念梳理
2. 引擎优化基础
3. 引擎优化进阶
4. 其他优化
5. 更多内容
6. Take Away
7. Q&A

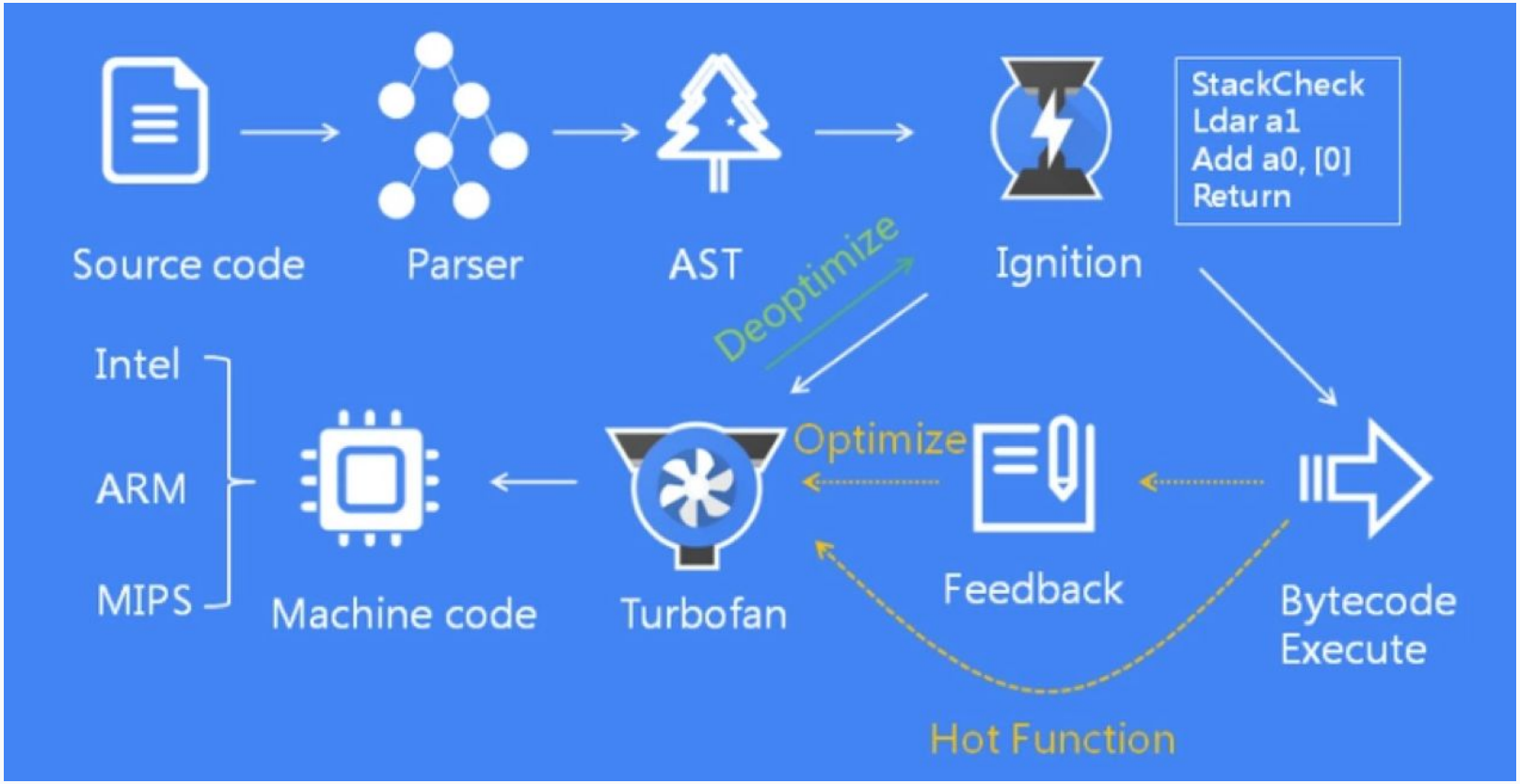
引擎概念梳理

第一部分

- 概述
- Parser
- Interpreter
- Optimizing compiler
- 浏览器间差异对比

引擎概念梳理 - 概述

JavaScript 代码的执行生命周期从源代码开始，首先被转成 AST，再转成字节码/机器码，然后被机器执行。



引擎概念梳理 - Parser

词法分析 (Tokenizing/Lexing) 就是将程序源代码分解成对编程语言来说有意义的代码块，这些代码块被称为词法单元 (Token)。

```
let i = 1;
```

→

```
[
    {
        "type": "Keyword",
        "value": "let"
    },
    {
        "type": "Identifier",
        "value": "i"
    },
    {
        "type": "Punctuator",
        "value": "="
    },
    {
        "type": "Numeric",
        "value": "1"
    },
    {
        "type": "Punctuator",
        "value": ";"
    }
]
```

→

1. 关键字 `let`
2. 标识符 `i`
3. 运算符 `=`
4. 数值 `1`
5. 分号 `;`

如果你想试一试自己的代码会被如何分析，可以用用这个工具 https://esprima.org/demo/parse.html#

引擎概念梳理 - Parser

Parser 即解析器，他的作用是根据 V8 分析出的词法单元生成抽象语法树，即我们所称的 AST。他会做几件事情：

- 分析语法错误：遇到错误的语法会抛出异常；
- 输出 AST：将词法单元流（数组）转换为一个由元素逐级嵌套所组成的抽象语法树；
- 确定词法作用域；

引擎概念梳理 - Parser


```
{
  "type": "Program",
  "start": 0,
  "end": 10,
  "body": [
    {
      "type": "VariableDeclaration",
      "start": 0,
      "end": 10,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 4,
          "end": 9,
          "id": {
            "type": "Identifier",
            "start": 4,
            "end": 5,
            "name": "i"
          },
          "init": {
            "type": "Literal",
            "start": 8,
            "end": 9,
            "value": 1,
            "raw": "1"
          }
        }
      ],
      "kind": "let"
    }
  ],
  "sourceType": "module"
}
```


在线生成 AST 的一个工具，你也可以自己试试 https://astexplorer.net/

引擎概念梳理 - Parser

如果一个函数从未被调用，用 Parser 去解析岂不是浪费资源了？

```
function foo () {
    console.log('I\'m function foo')
}

function bar () {
    console.log('I\'m function bar')
}

foo()
```

引擎概念梳理 - Parser

具体就 V8 而言，有两个解析器用于解析 JavaScript 代码，分别是 Parser 和 Pre-Parser。

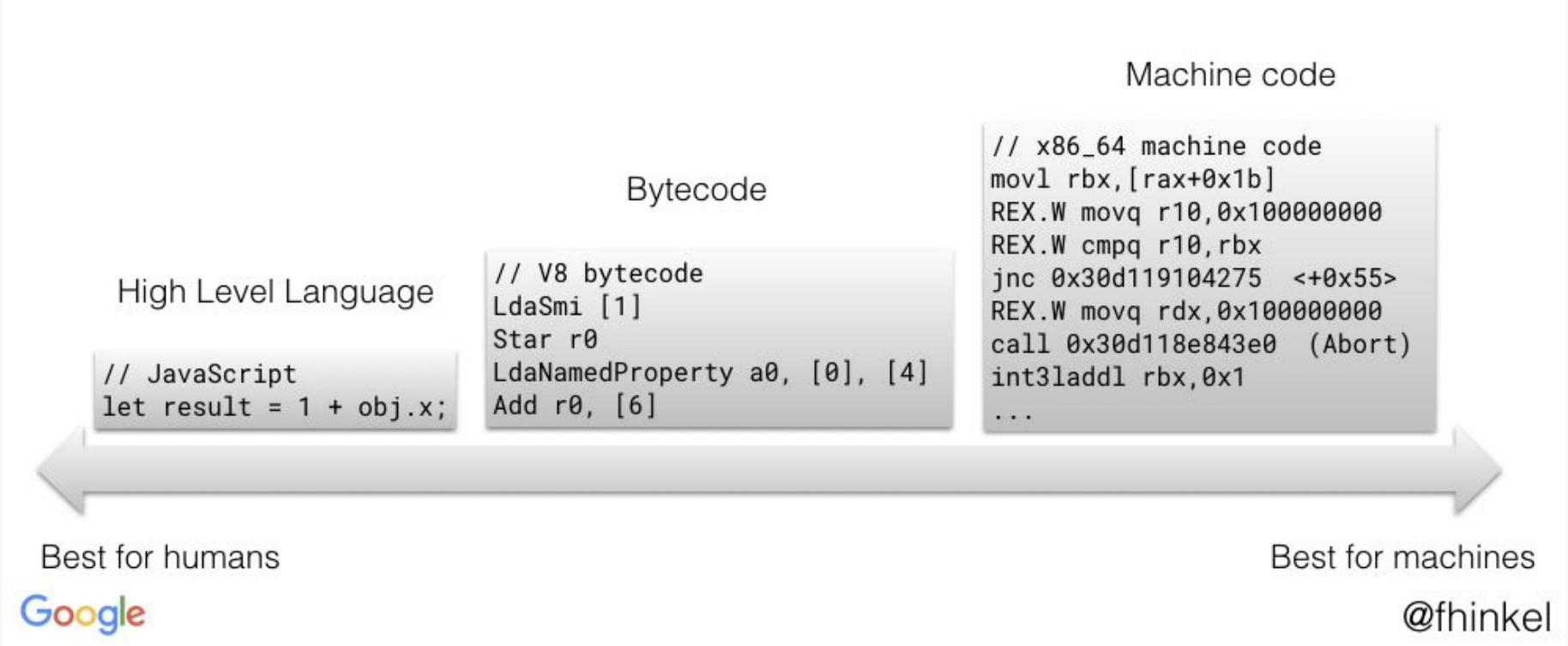
- Parser 又称为 full parser（全量解析）或者 eager parser（饥饿解析），它会解析所有立即执行的代码，包括语法检查、生成 AST、确定词法作用域。
- Pre-Parser 又称为惰性解析，它只解析未被立即执行的代码（如函数），不生成 AST，只确定作用域，以此来提高性能。当预解析后的代码开始执行时，才进行 Parser 解析。

相比 Parser，Pre-Parser 的速度可以快至两倍以及更多。关于 V8 惰性解析的内容，可以移步博客 Lazy JavaScript Parsing in V8 https://www.mattzeunert.com/2017/01/30/lazy-javascript-parsing-in-v8.html 查看更多。

引擎概念梳理 - Interpreter

Interpreter 即解释器，它负责的工作包括：

- 将 AST 转换为字节码
- 逐行解释执行字节码



引擎概念梳理 - Interpreter

你可以字节码看作是小型的构建块(bytecodes as small building blocks)，这些构建块组合在一起构成任何 JavaScript 功能。

通过 Node 命令 node --print-bytecode example.js，你可以得到由你代码生成的字节码结果。

```
...

4599 E> 0x300456f5f28b @  321 : a5 08
ThrowReferenceErrorIfHole [ 8]
        0x300456f5f28d @  323 : 26 f1             Star r10
4626 E> 0x300456f5f28f @  325 : 28 f1 01 32
LdaNamedProperty r10, [ 1], [50]
        0x300456f5f293 @  329 : 26 f1             Star r10
        0x300456f5f295 @  331 : 0b                LdaZero
4633 E> 0x300456f5f296 @  332 : 65 f1 34
TestEqualStrict r10, [ 52]
        0x300456f5f299 @  335 : 4f                LogicalNot
4639 S> 0x300456f5f29a @  336 : a4                Return
Constant pool (size =  19)
Handler Table (size =  0)
```

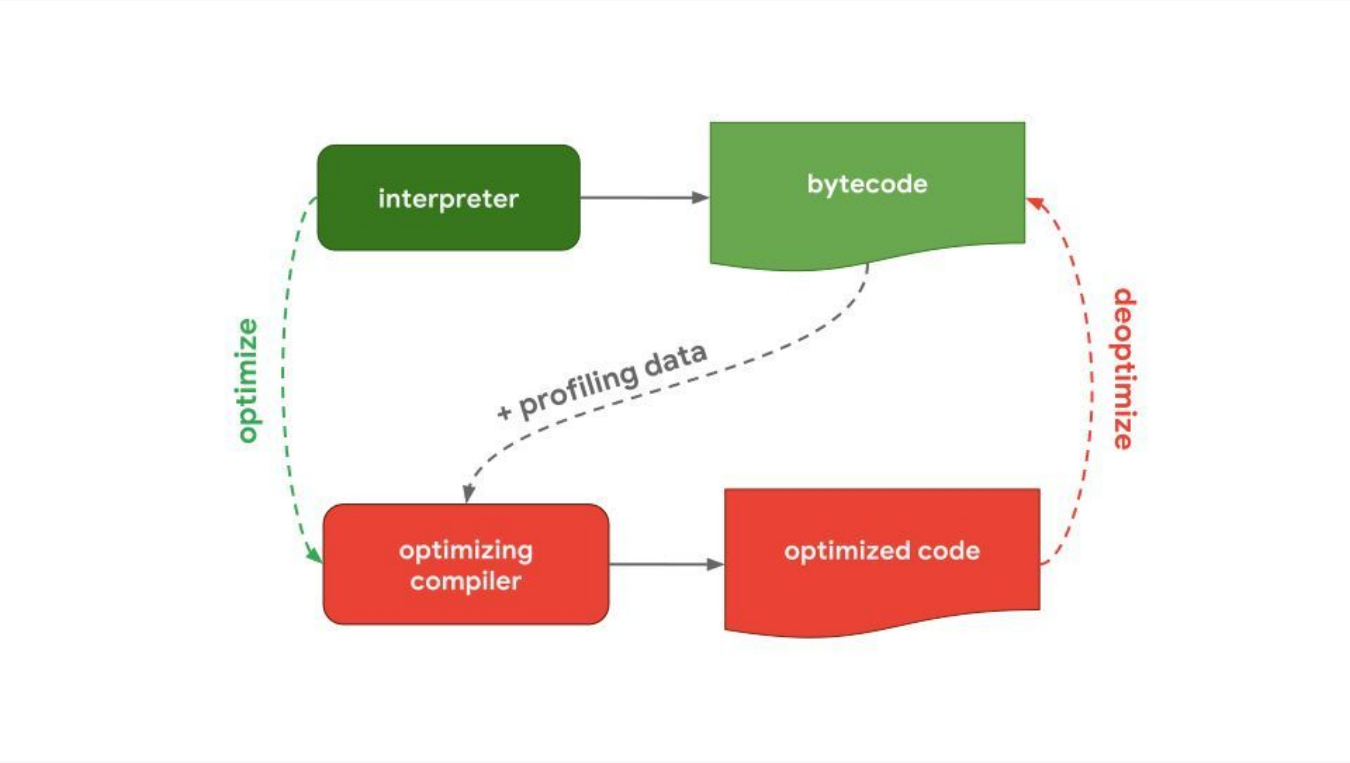
关于字节码你可以在 Understanding V8's Bytecode 这篇文章里了解更多
https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775

引擎概念梳理 - Optimizing compiler

Optimizing compiler，顾名思义即为优化编译器。

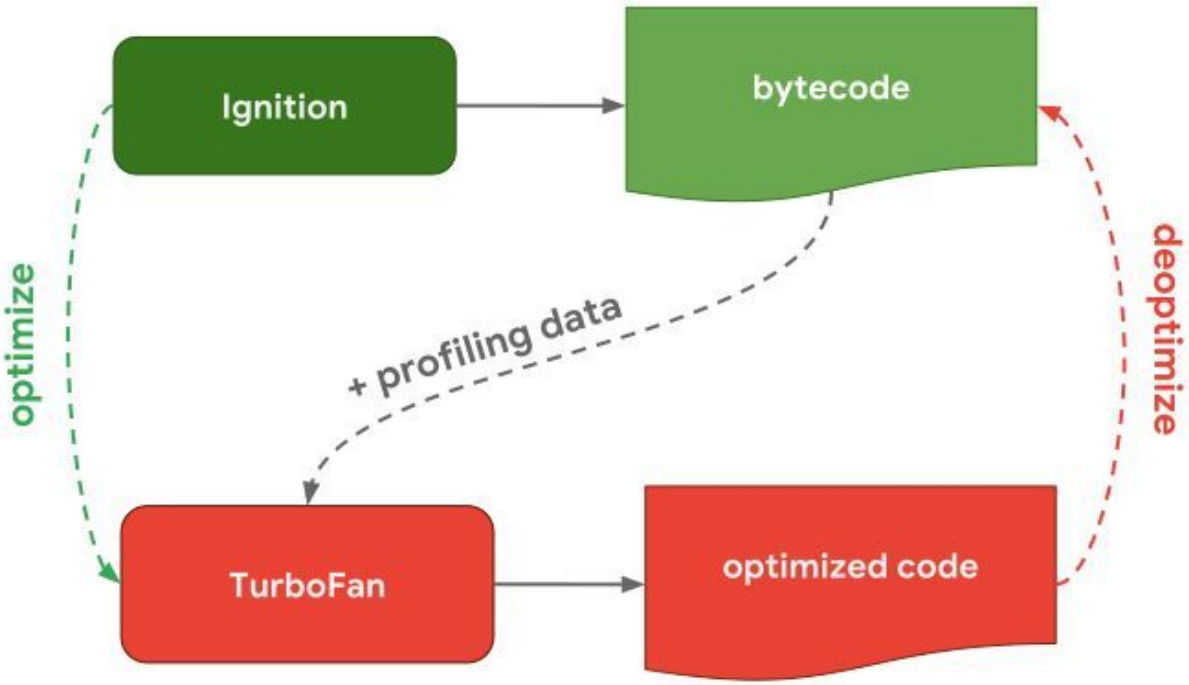
当代码转换为字节码后，程序就可以执行了，为什么还需要优化编译器呢？

所有的优化都是一定生效的么？



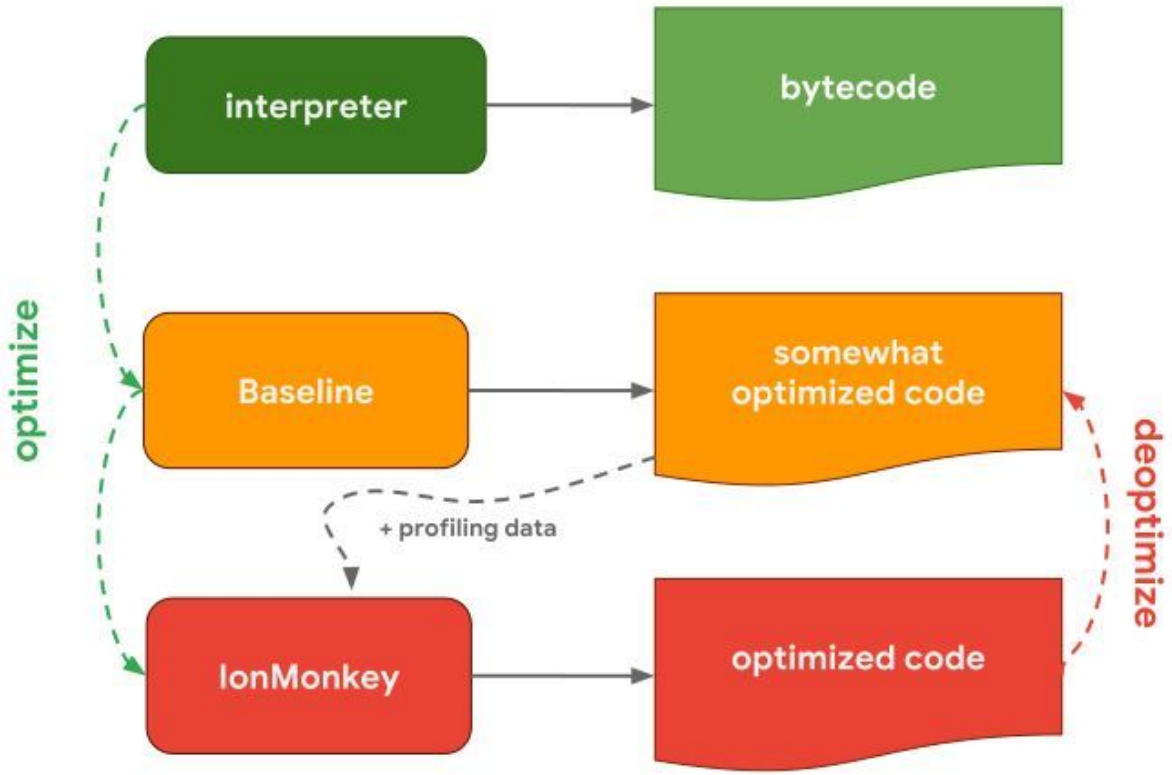
引擎概念梳理 - 浏览器间差异对比

V8 (Chrome)



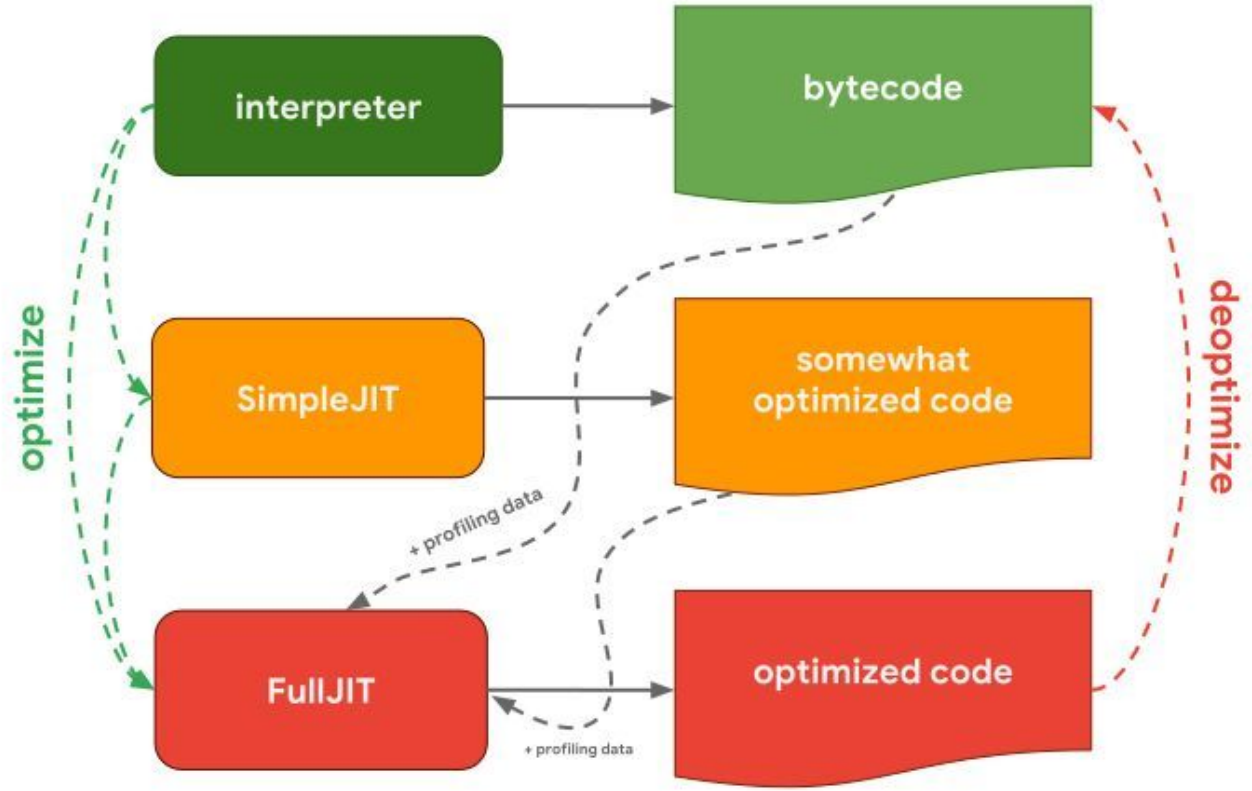
引擎概念梳理 - 浏览器间差异对比

SpiderMonkey (Mozilla)



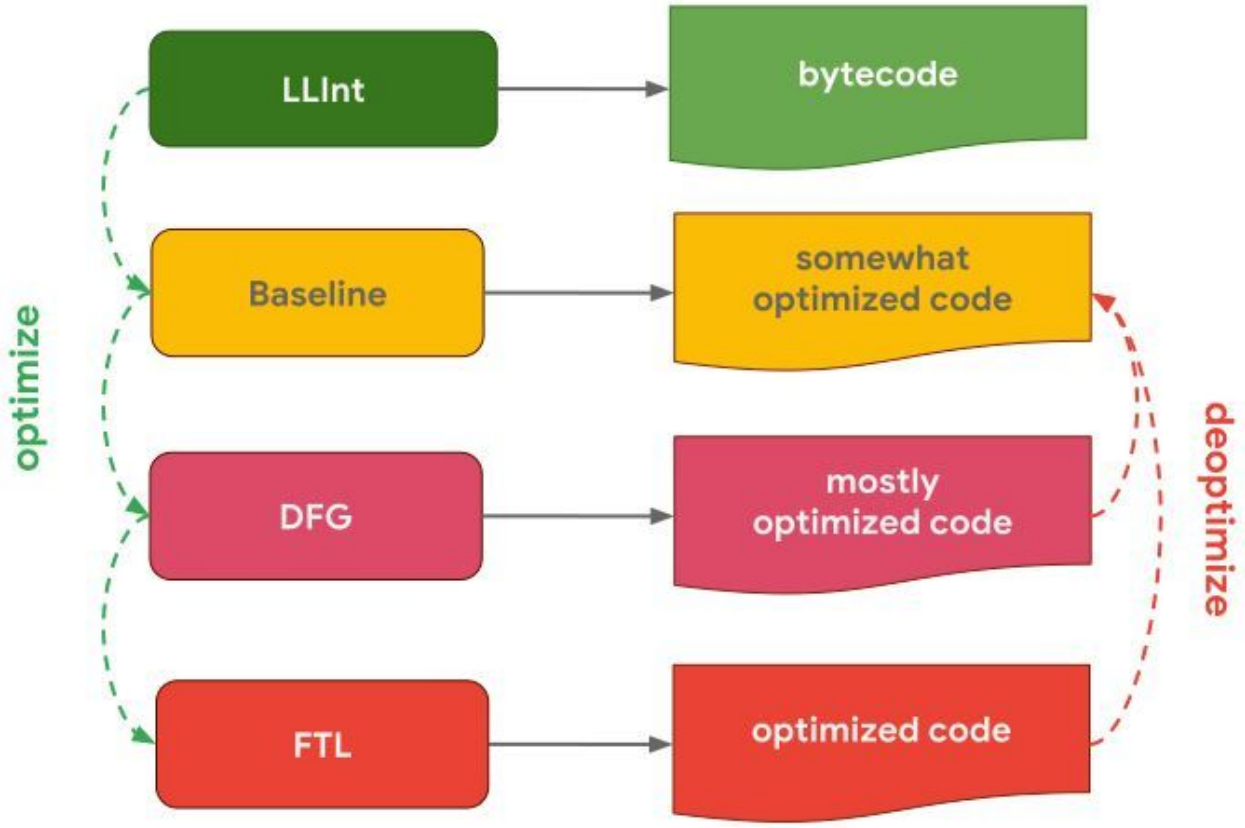
引擎概念梳理 - 浏览器间差异对比

Chakra (Edge)



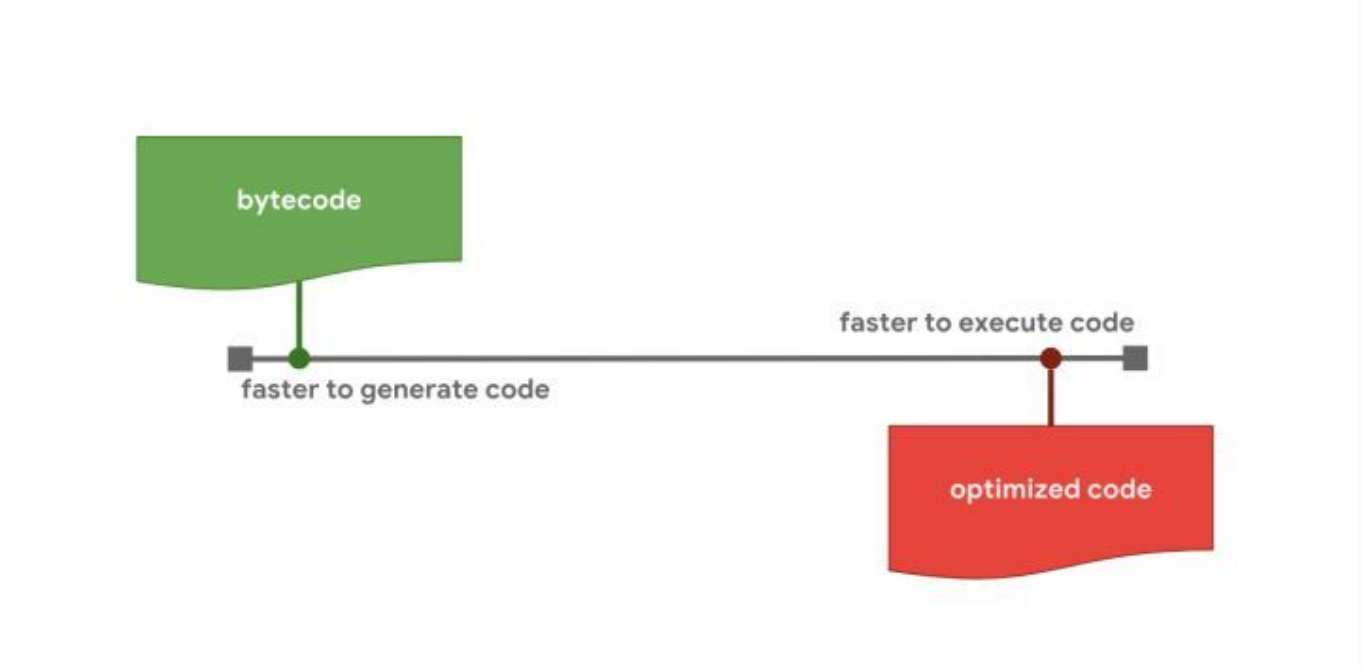
引擎概念梳理 - 浏览器间差异对比

JavaScriptCore (Apple)



引擎概念梳理 - 浏览器间差异对比

为什么有些引擎会拥有更多的优化编译器呢？



引擎优化基础

第二部分

- JavaScript 对象模型
- 属性访问优化
  - Shapes
  - Transition 链与树
  - Inline Caches
- 高效存储数组

JavaScript 对象模型

Q：什么是 JavaScript 对象模型？

Q：JavaScript 引擎是如何实现 JavaScript 对象模型的呢？

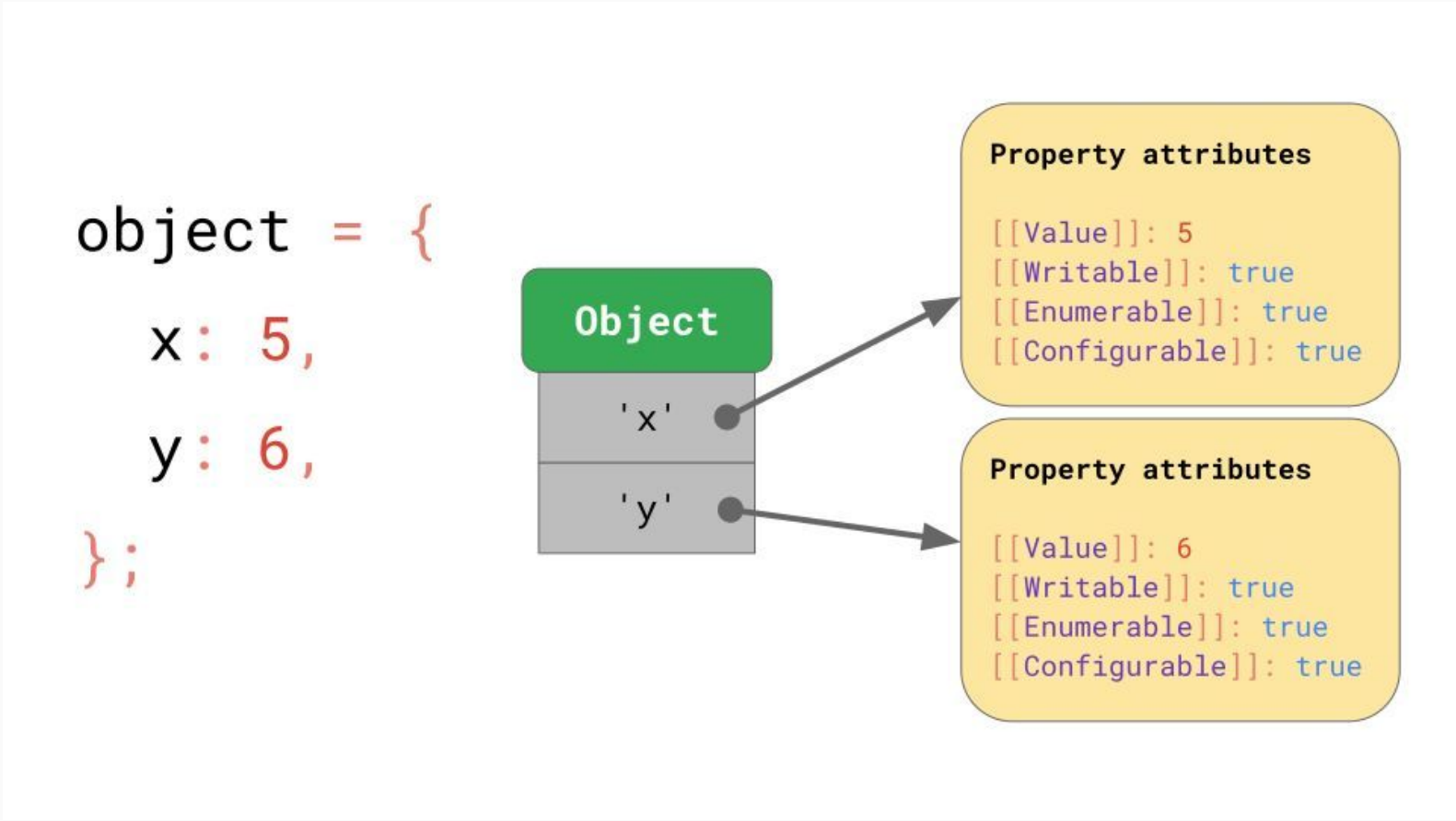
JavaScript 对象模型 - 对象

对于一个 JavaScript 对象来说，除 [[Value]] 外，规范还定义了如下属性：

- [[Writable]] 决定该属性是否可以被重新赋值；
- [[Enumerable]] 决定该属性是否出现在 for-in 循环中；
- [[Configurable]] 决定该属性是否可被删除。

JavaScript 对象模型 - 对象

```
const object = { foo: 42 };
Object.getOwnPropertyDescriptor(object, 'foo');
// → { value: 42, writable: true, enumerable: true, configurable: true }
```



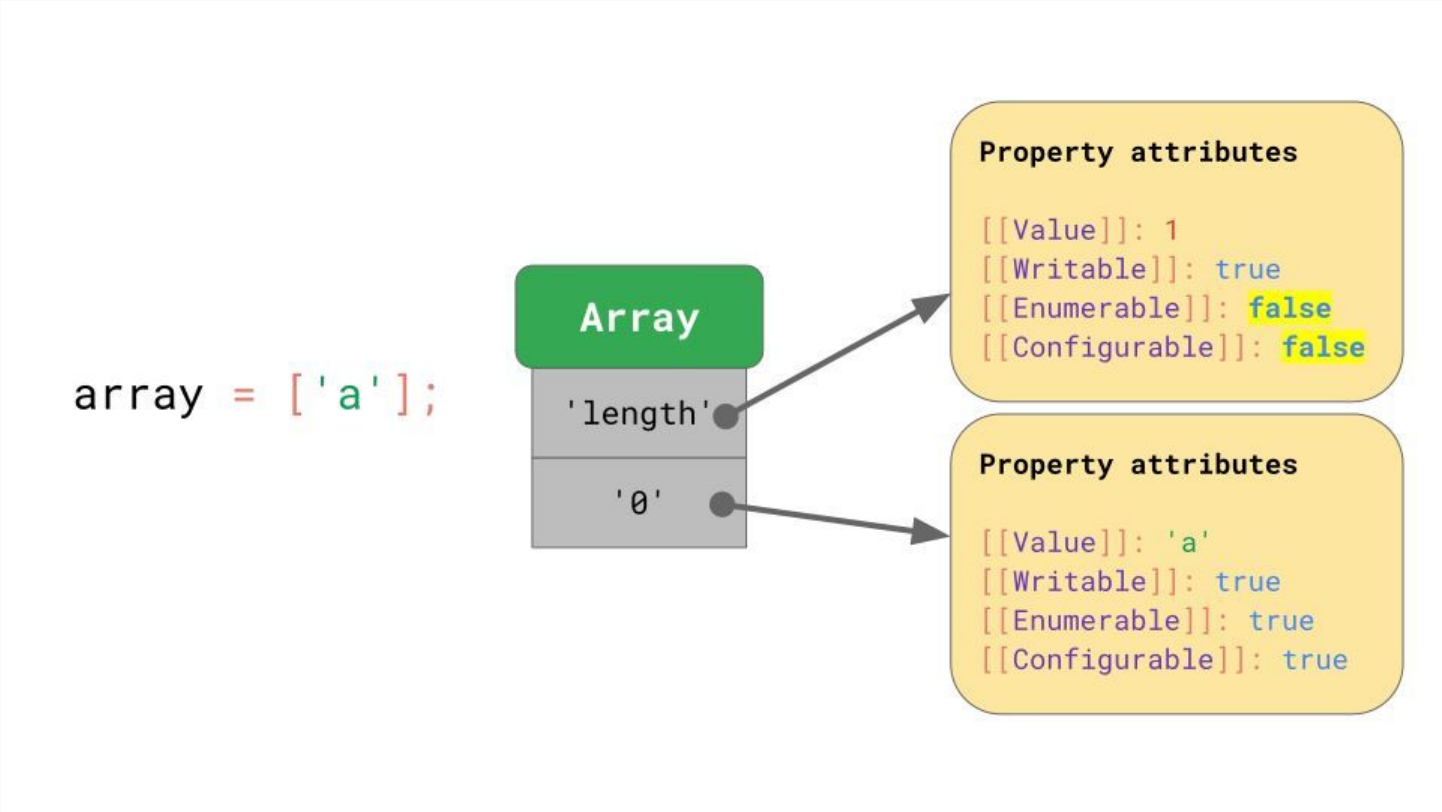
JavaScript 对象模型 - 数组

- 同：特殊的对象
- 异：数组会对数组索引进行特殊的处理。

这里所指的数组索引是 ECMAScript 规范中的一个特殊术语。在 JavaScript 中，数组被限制最多只能拥有2^32-1项。数组索引是指该限制内的任何有效索引，即从0到2^32-2的任何整数。

JavaScript 对象模型 - 数组

```
const array = ['a', 'b'];
array.length; // → 2
array[2] = 'c';
array.length; // → 3
```



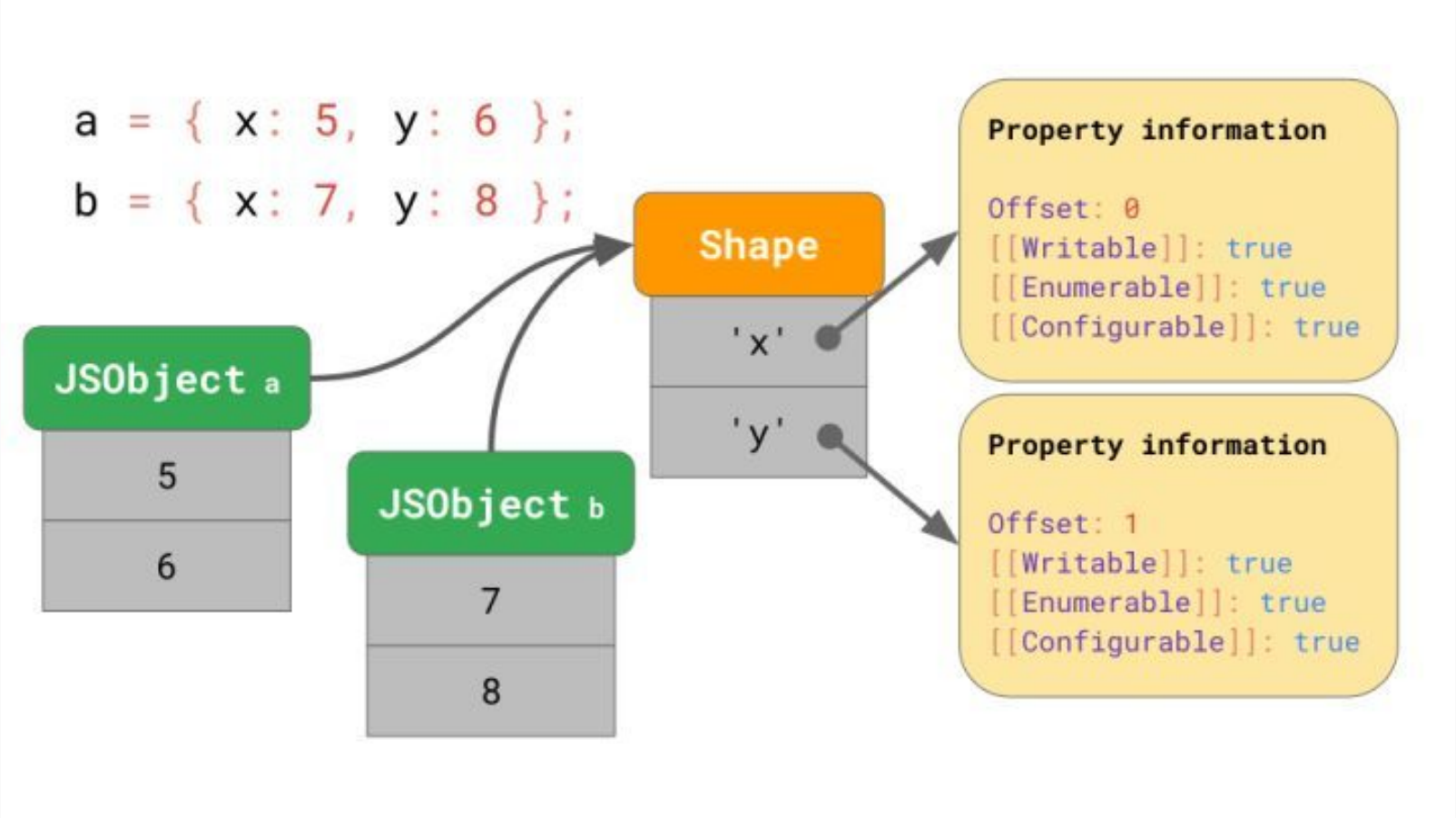
属性访问优化 - Shapes

```
function logX(object) {
    console.log(object.x);
    //          ^^^^^^^^
  }

  const object1 = { x: 5, y: 6 };
  const object2 = { x: 7, y: 8 };

  logX(object1);
  logX(object2);
```

属性访问优化 - Shapes



属性访问优化 - Shapes

所有的 JavaScript 引擎都使用了 Shapes 作为优化，但称呼各有不同：

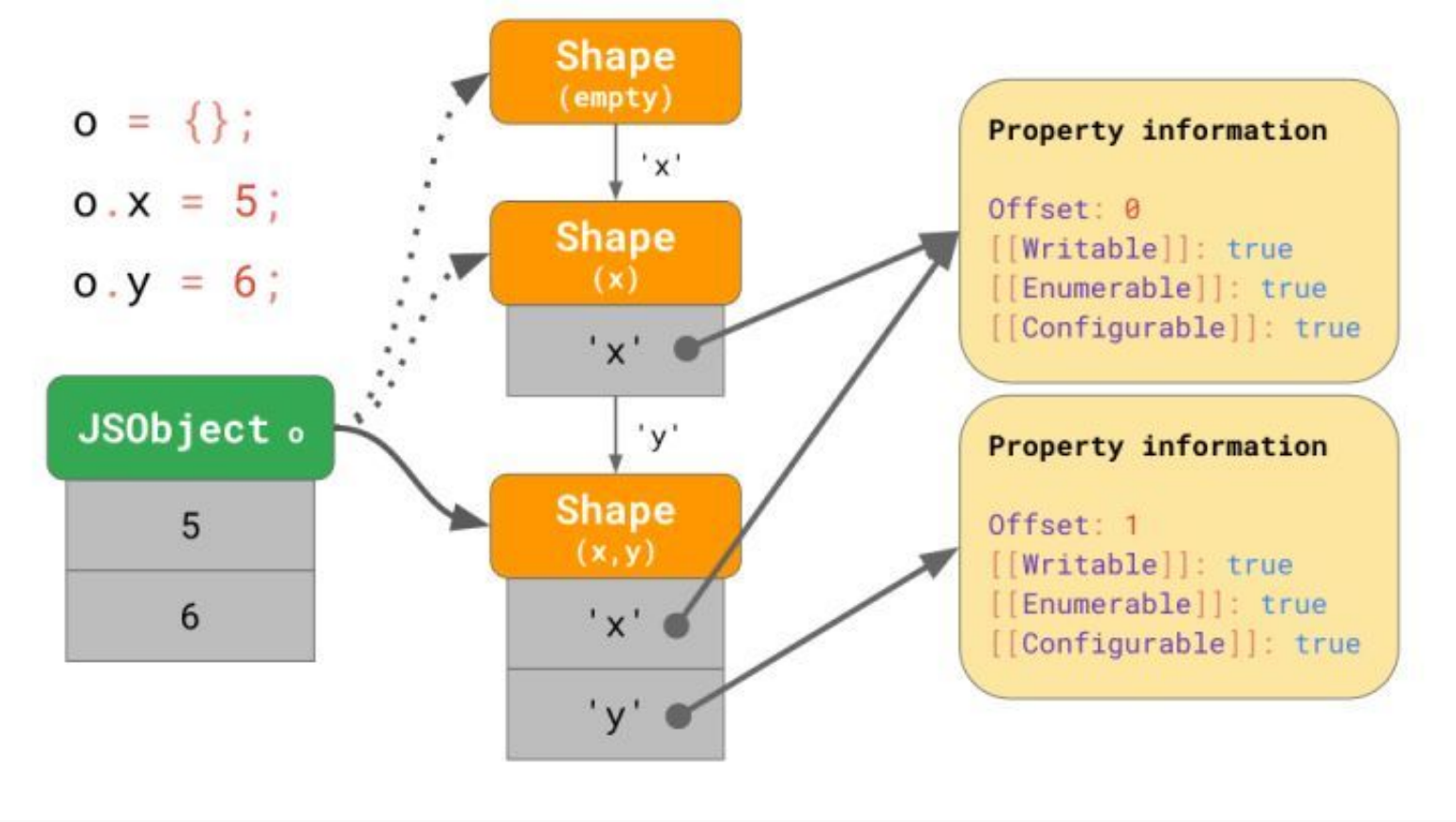
- 学术论文称它们为 Hidden Classes
- V8 将它们称为 Maps
- Chakra 将它们称为 Types
- JavaScriptCore 称它们为 Structures
- SpiderMonkey 称他们为 Shapes

属性访问优化 - Transition 链与树

并不是所有对象的定义都是一蹴而就的

```
const object = {};
object.x = 5;
object.y = 6;
```

属性访问优化 - Transition 链与树

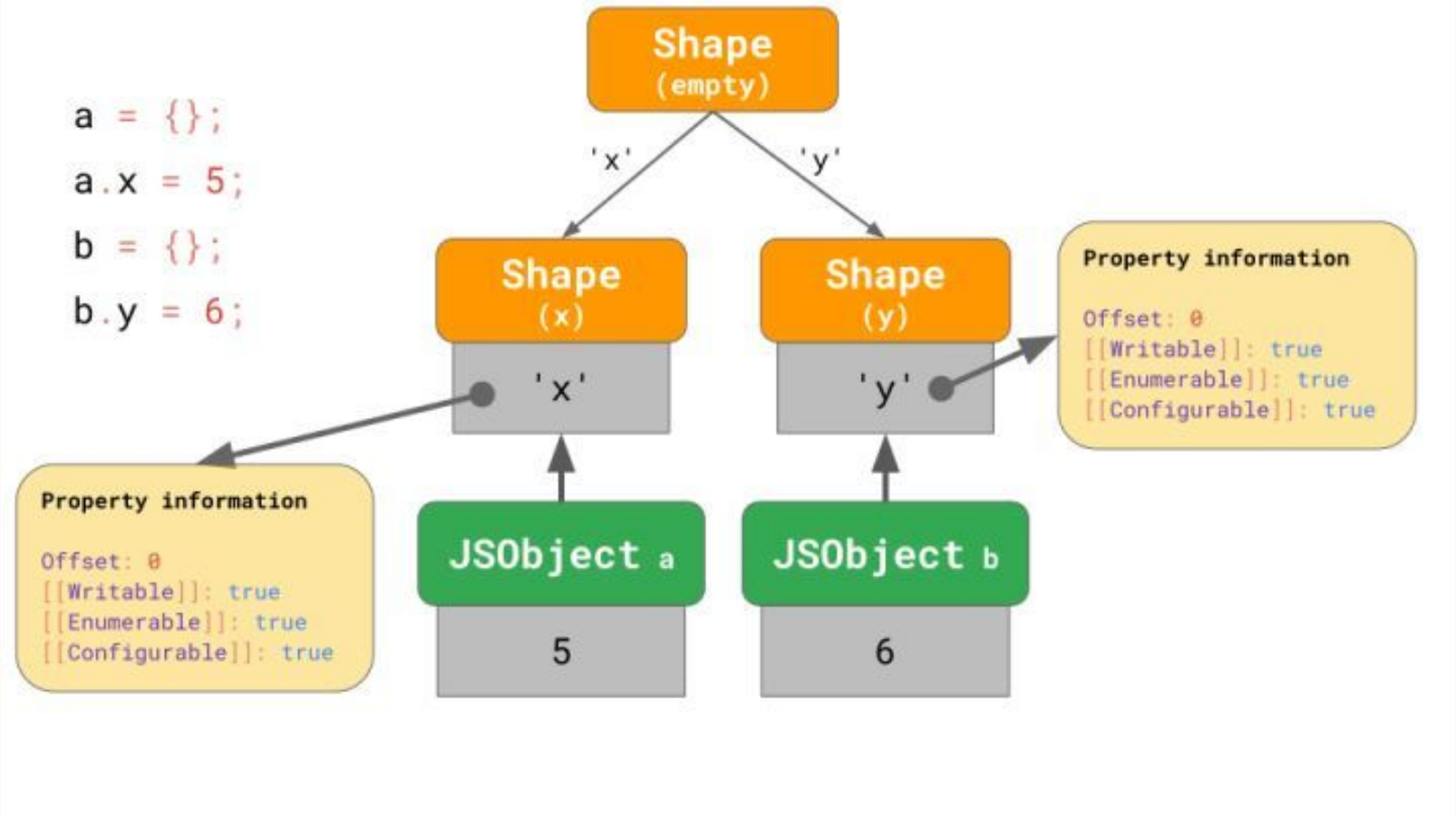


属性访问优化 - Transition 链与树

我们再来看另一个例子

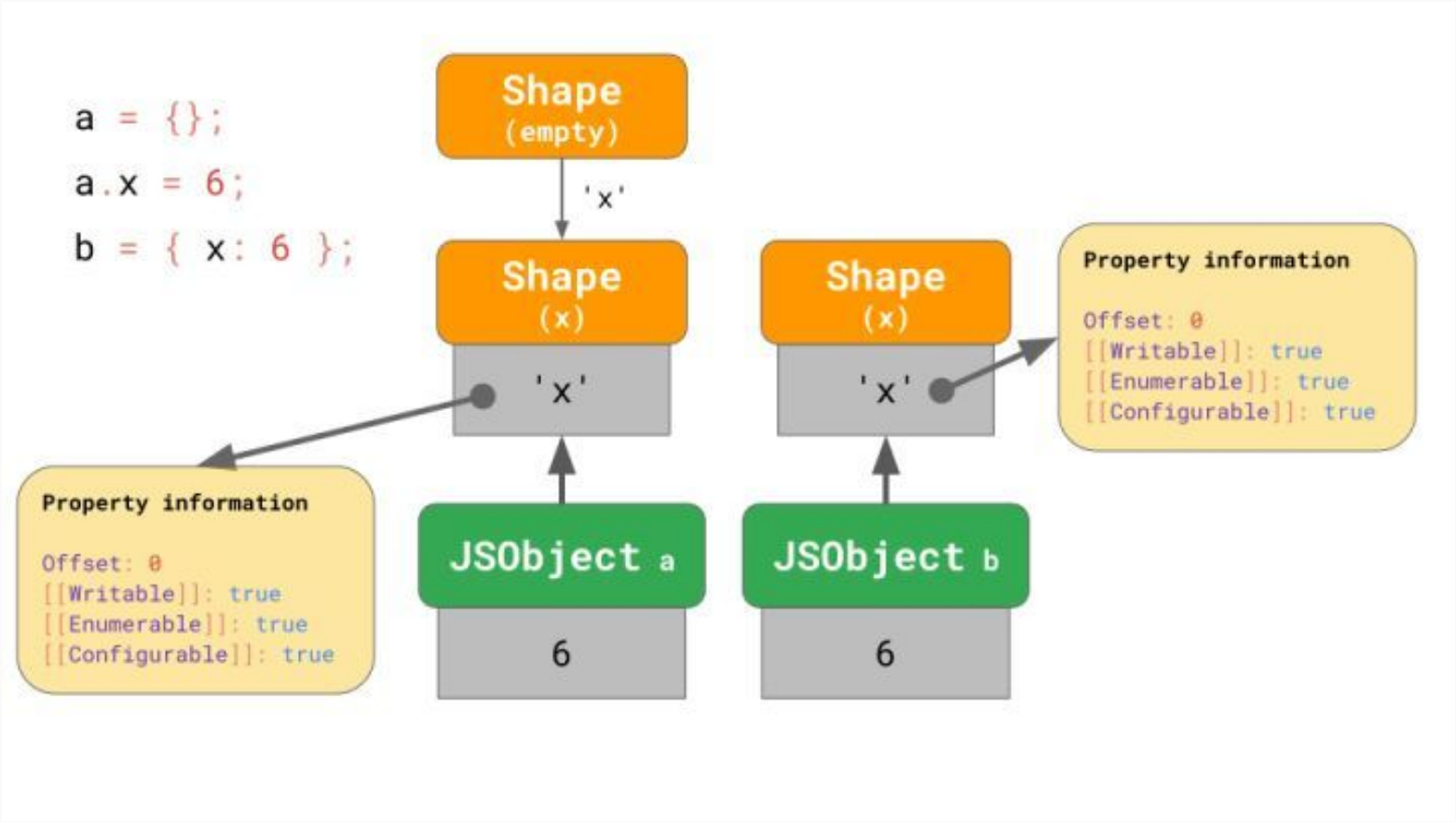
```
const object1 = {};
object1.x = 5;
const object2 = {};
object2.y = 6;
```

属性访问优化 - Transition 链与树



属性访问优化 - Transition 链与树

但并不是所有对象都从空的 Shape 开始构建



Benedikt 的博文 Surprising polymorphism in React applications 讨论了这些微妙之处是如何影响实际性能的。

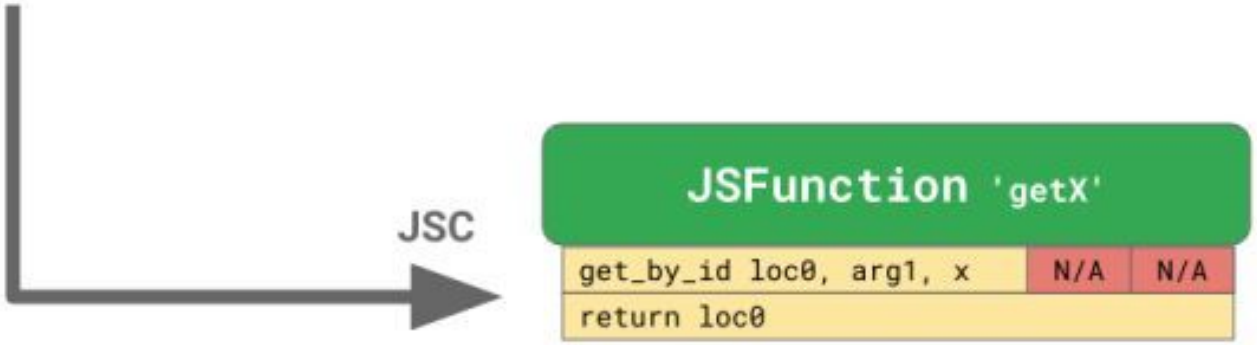
属性访问优化 - Inline Caches

Shapes 背后的主要动机是 Inline Caches 或 ICs 的概念。ICs 是促使 JavaScript 快速运行的关键因素！JavaScript 引擎利用 ICs 来记忆去哪里寻找对象属性的信息，以减少昂贵的查找次数。

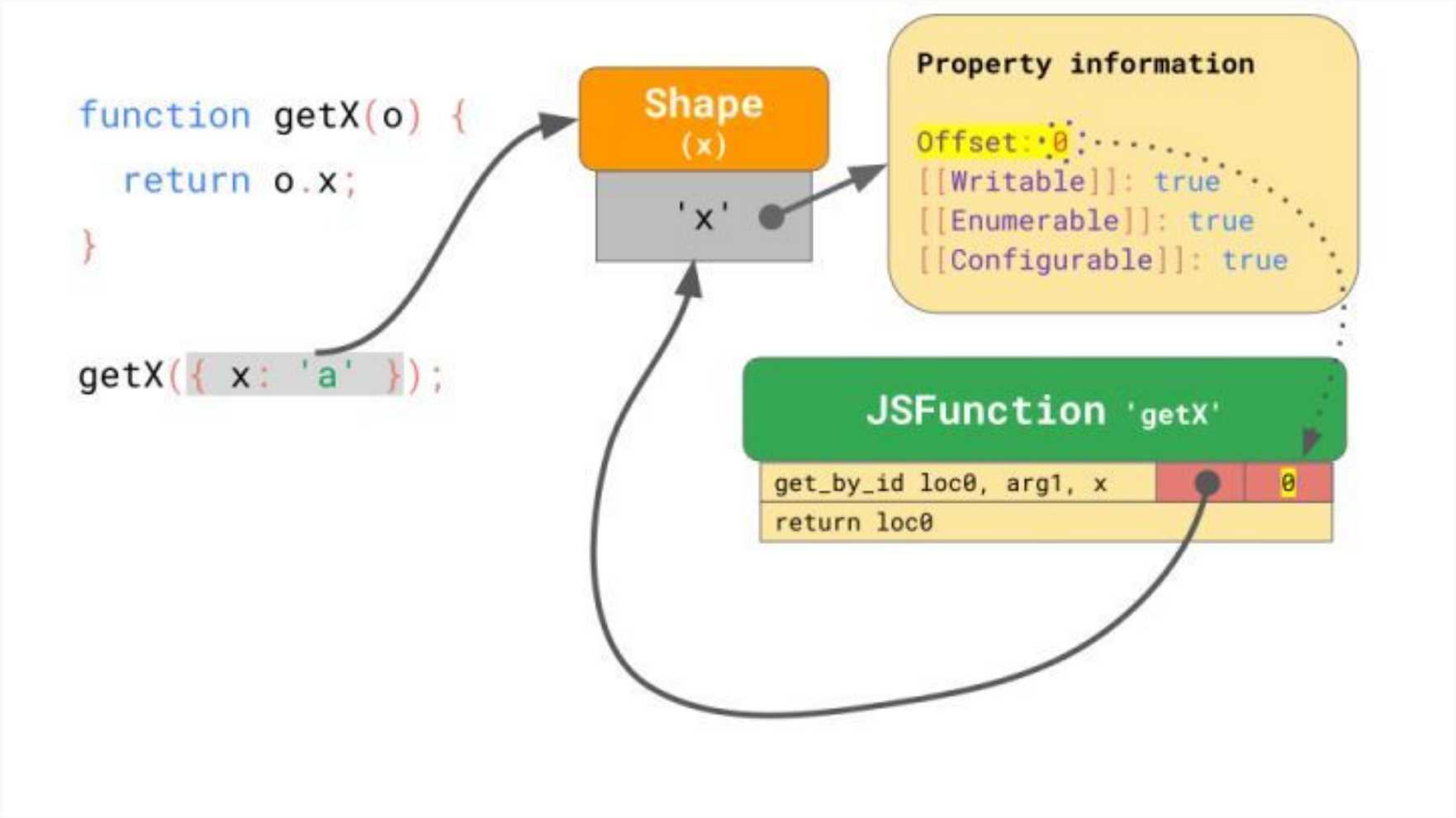
```
function getX(o) {
    return o.x;
}
```

属性访问优化 - Inline Caches

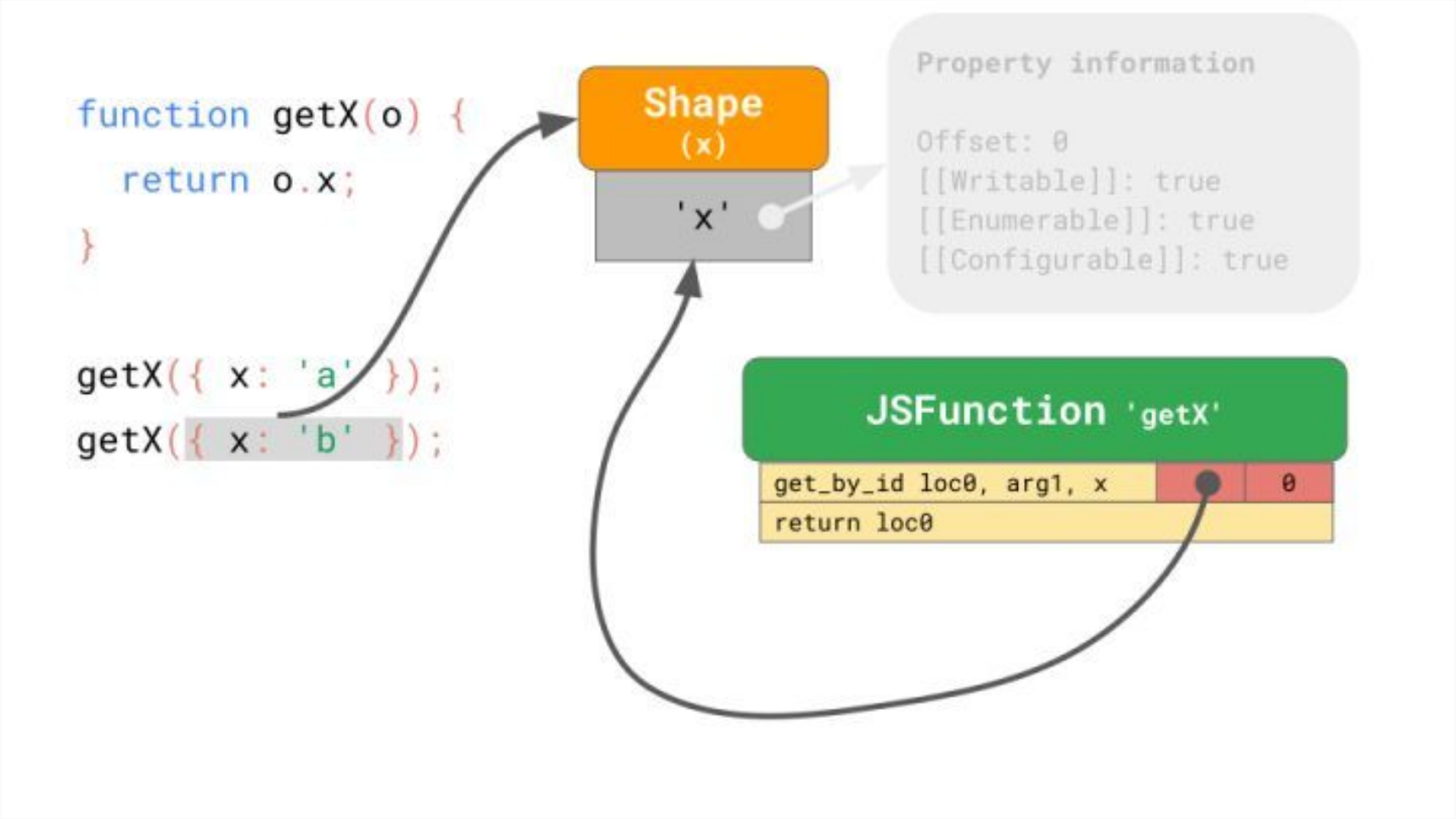
```
function getX(o) {
  return o.x;
}
```



属性访问优化 - Inline Caches



属性访问优化 - Inline Caches

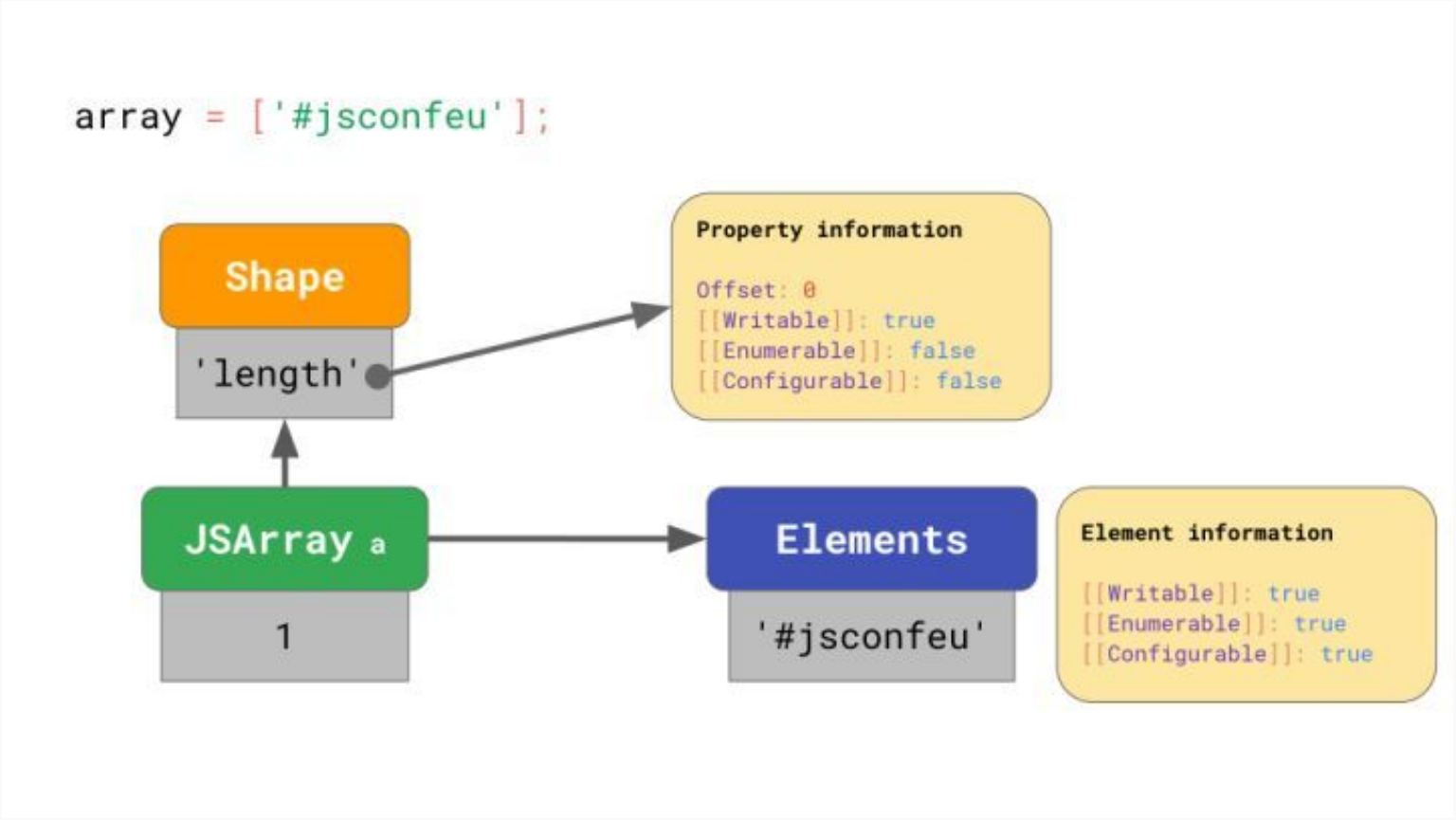


高效存储数组

```
const array = [
    '#jsconfeu',
];
```

高效存储数组

每个数组都有一个单独的 elements backing store，其中包含所有数组索引的属性值。JavaScript 引擎不必为数组元素存储任何属性特性，因为它们通常都是可写的，可枚举的以及可配置的。

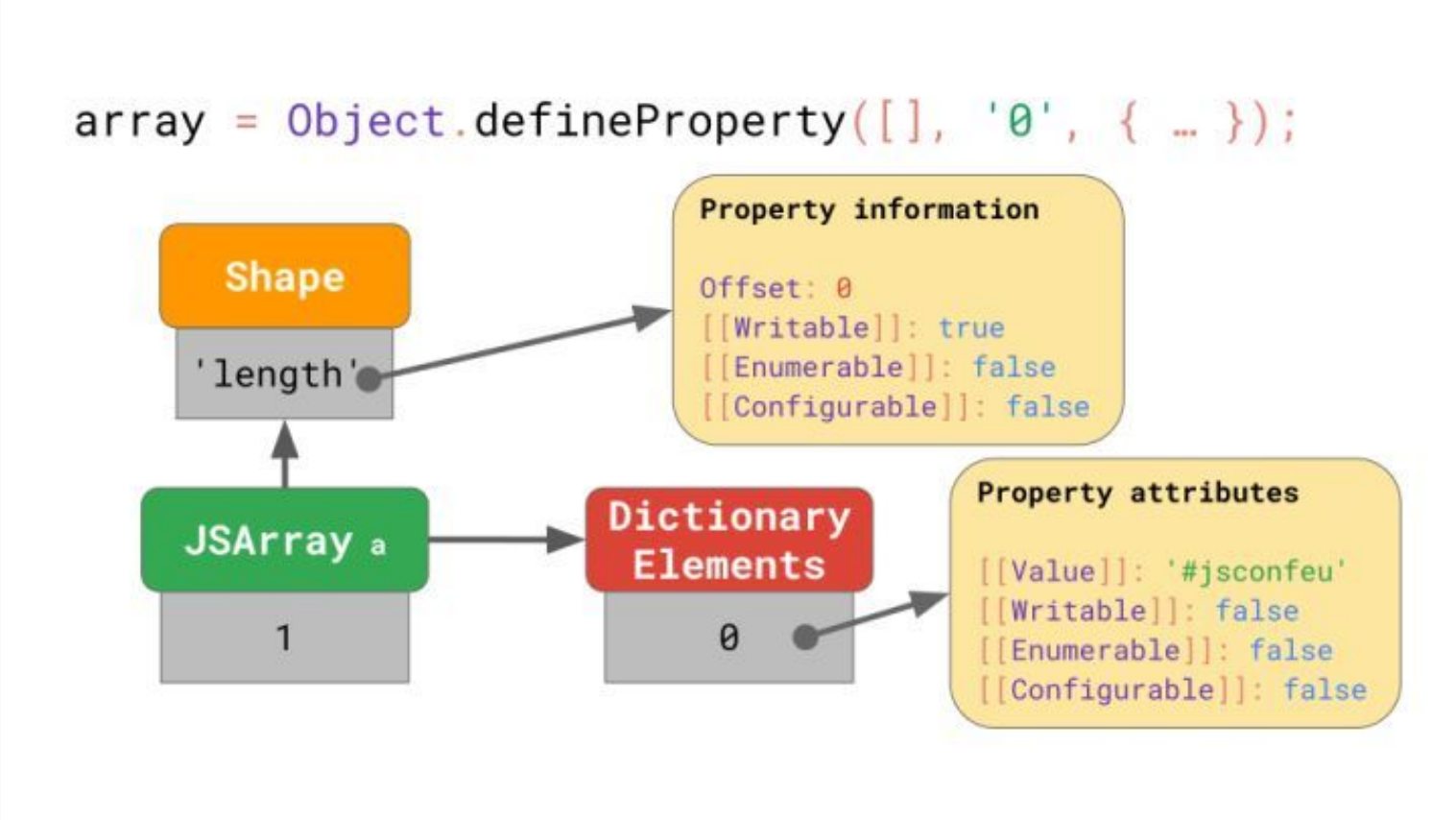


高效存储数组

```
const array = Object.defineProperty(
    [],
    '0',
    {
      value: 'Oh noes!!1',
      writable: false,
      enumerable: false,
      configurable: false,
    }
);
```

高效存储数组

在这种边缘情况下，JavaScript 引擎会将全部的 elements backing store 表示为一个由数组下标映射到属性特性的字典。



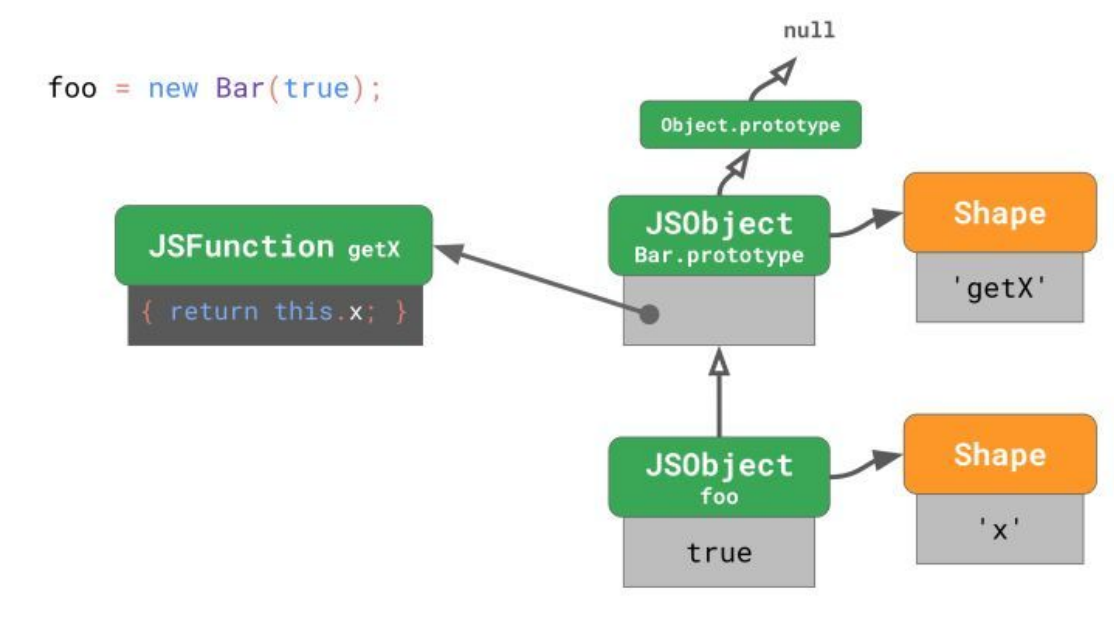
其他优化

第三部分

优化层级与执行效率的取舍

原型属性访问优化

……



更多内容 | 第四部分

- JavaScript 引擎基础：Shapes 和 Inline Caches https://zhuanlan.zhihu.com/p/38202123
- JavaScript 引擎基础：原型优化 https://zhuanlan.zhihu.com/p/42630183
- JavaScript 引擎是如何工作的 https://github.com/jeuino/Blog/issues/6
- Ignition: An Interpreter for V8 https://docs.google.com/presentation/d/1OqjVqRhtwlKeKfvMdX6HaCIu9wpZsrzqpIVIwQSuiXQ/edit#slide=id.g1357e6d1a4_0_58
- What is V8? https://v8.dev/
- How JavaScript works: an overview of the engine, the runtime, and the call stack https://blog.sessionstack.com/how-does-javascript-actually-work-part-1-b0bacc073cf
- Ignition: Jump-starting an Interpreter for V8 https://docs.google.com/presentation/d/1HgDDXBYqCJNasBKBDf9szap1j4q4wnSHhOYpaNy5mHU/edit#slide=id.g1357e6d1a4_0_58
- The Journey of JavaScript: from Downloading Scripts to Execution - Part I https://www.telerik.com/blogs/journey-of-javascript-downloading-scripts-to-execution-part-i
- Lazy JavaScript Parsing in V8 https://www.mattzeunert.com/2017/01/30/lazy-javascript-parsing-in-v8.html
- AST Explorer https://astexplorer.net/
- Parser (AST) https://esprima.org/demo/parse.html#
- 理解 V8 的字节码 https://zhuanlan.zhihu.com/p/28590489
- How JavaScript works: inside the V8 engine + 5 tips on how to write optimized code https://blog.sessionstack.com/how-javascript-works-inside-the-v8-engine-5-tips-on-how-to-write-optimized-code-ac089e62b12e

Take Away | 第五部分

- 可以试试看 JavaScript 引擎在解释、编译和运行代码时都做了些什么
- 针对对象类型，始终以相同的方式初始化对象，以确保它们不会走向不同的 shape 方向
- 针对数组类型，不要混淆数组元素的属性特性（property attributes），以确保可以高效地存储和操作它们
- JavaScript 引擎还有很多其他的优化策略，比如优化层级取舍、原型优化等

Q&A

@hijiangtao