[译] JavaScript 引擎基础：Shapes 和 Inline Caches

前言：本文也可以被称做 "JavaScript Engines: The Good Parts™"，其来自 Mathias 和 Benedikt 在 JSConf EU 2018 上为本文主题演讲所起的题目，更多 JSconf EU 2018 上有趣的主题分享可以参考这个答案 (https://www.zhihu.com/question/279637889/answer/408989776)。

本文就所有 JavaScript 引擎中常见的一些关键基础内容进行了介绍——这不仅仅局限于 V8 引擎。作为一名 JavaScript 开发者，深入了解 JavaScript 引擎是如何工作的将有助于你了解自己所写代码的性能特征。关于本文，全文共由五个部分组成：

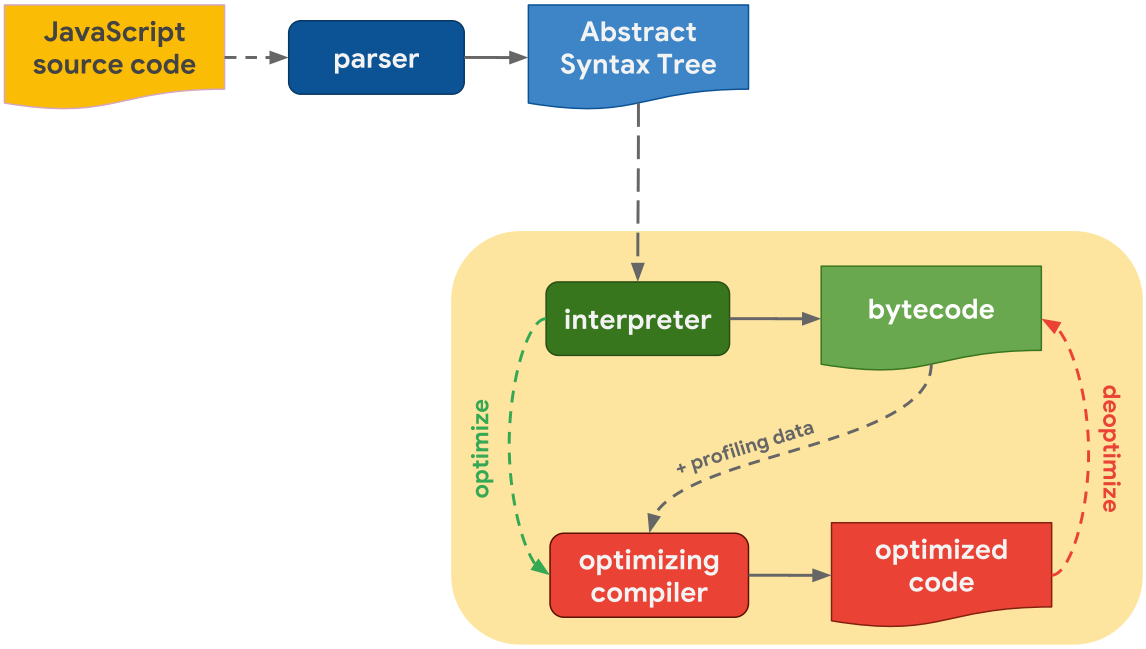
1. **JavaScript 引擎工作流程**：介绍 JavaScript 引擎的处理流水线，这一部分会涉及到解释器 / 编译器的内容，且会分点介绍不同引擎间的差别与共同点；
2. **JavaScript 对象模型**；
3. **属性访问的优化**：通过 Shapes、Transistion 链与树、ICs 等概念的穿插介绍引擎是如何优化获取对象属性的；
4. **高效存储数组**；
5. **Take-aways**：对全文内容做了一个小结，并给了两点建议。

原文 JavaScript engine fundamentals: Shapes and Inline Caches (https://mathiasbynens.be/notes/shapes-ics)，作者 @Benedikt (https://twitter.com/bmeurer) 和 @Mathias (https://twitter.com/mathias)，译者 hijiangtao (https://github.com/hijiangtao)，你也可以在知乎专栏查看本文 (https://zhuanlan.zhihu.com/p/38202123)。以下开始正文。

> *如果你倾向看视频演讲，请移步 YouTube (https://www.youtube.com/embed/5nmpokoRaZI) 查看更多。*

1. JavaScript 引擎工作流程

这一切都得从你所写的 JavaScript 代码开始说起。JavaScript 引擎在解析源码后将其转换为抽象语法树（AST）。基于 AST，解释器便可以开始工作并产生字节码。非常棒！此时引擎正在执行 JavaScript 代码。



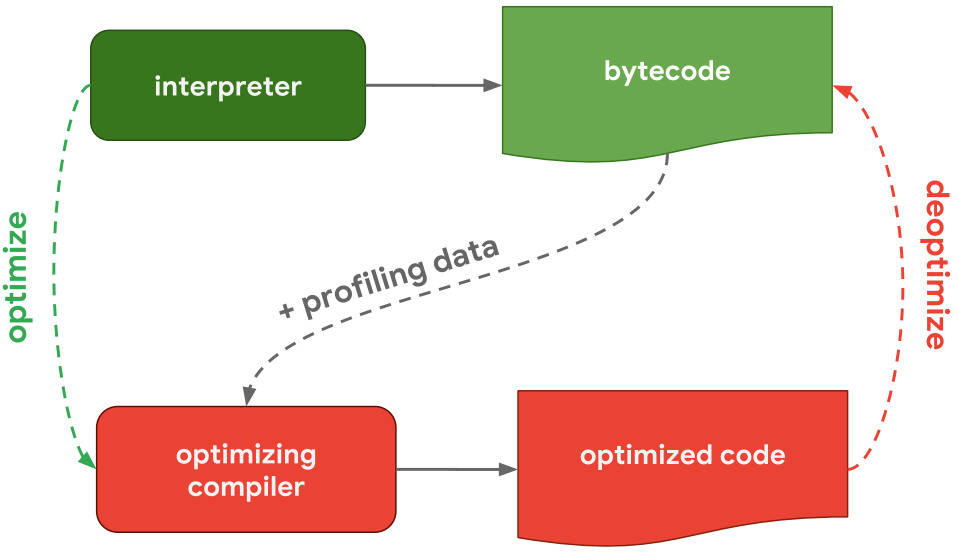
为了使它执行得更快，可以将字节码与分析数据（profiling data）一起发给优化编译器。优化编译器根据已有的分析数据做出特定假设，然后生成高度优化的机器码。

如果在某点上一个假设被证明是不正确的，那么优化编译器会**去优化**并回退至解释器部分。

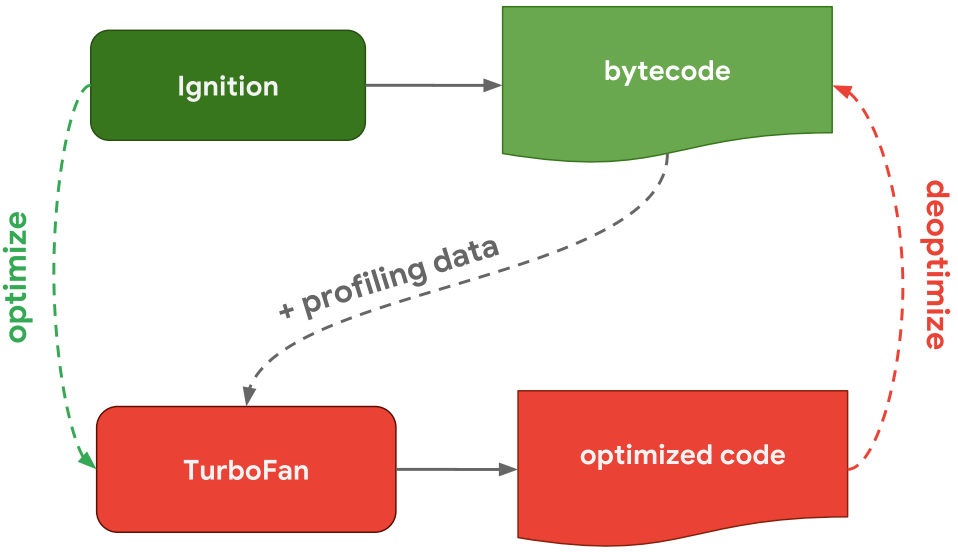
1.1 JavaScript 引擎中的解释器/编译器流程

现在，让我们关注实际执行 JavaScript 代码的这部分流程，即代码被解释和优化的地方，并讨论其在主要的 JavaScript 引擎之间存在的一些差异。

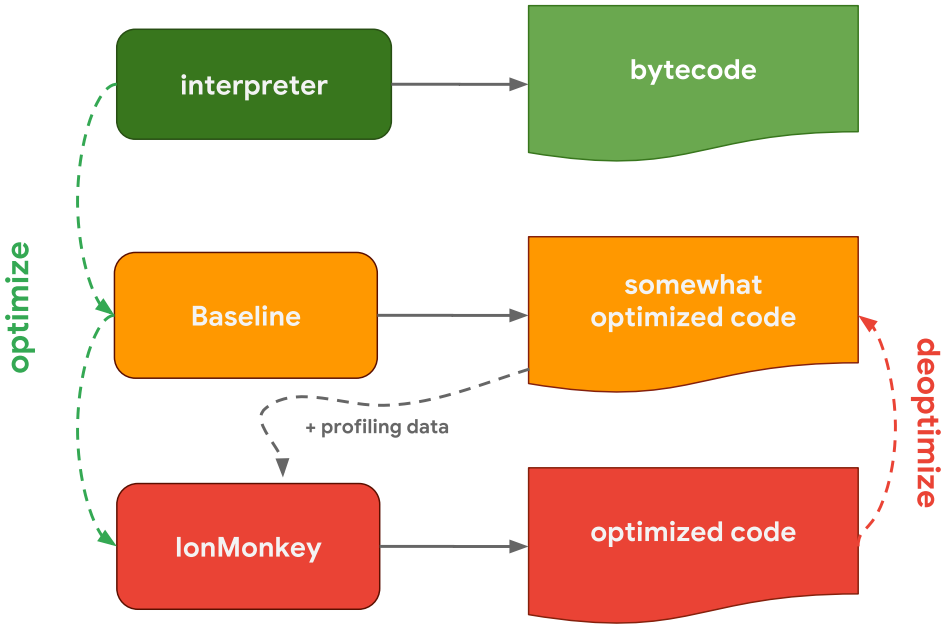
一般来说，（所有 JavaScript 引擎）都有一个包含解释器和优化编译器的处理流程。其中，解释器可以快速生成未优化的字节码，而优化编译器会需要更长的时间，以便最终生成高度优化的机器码。



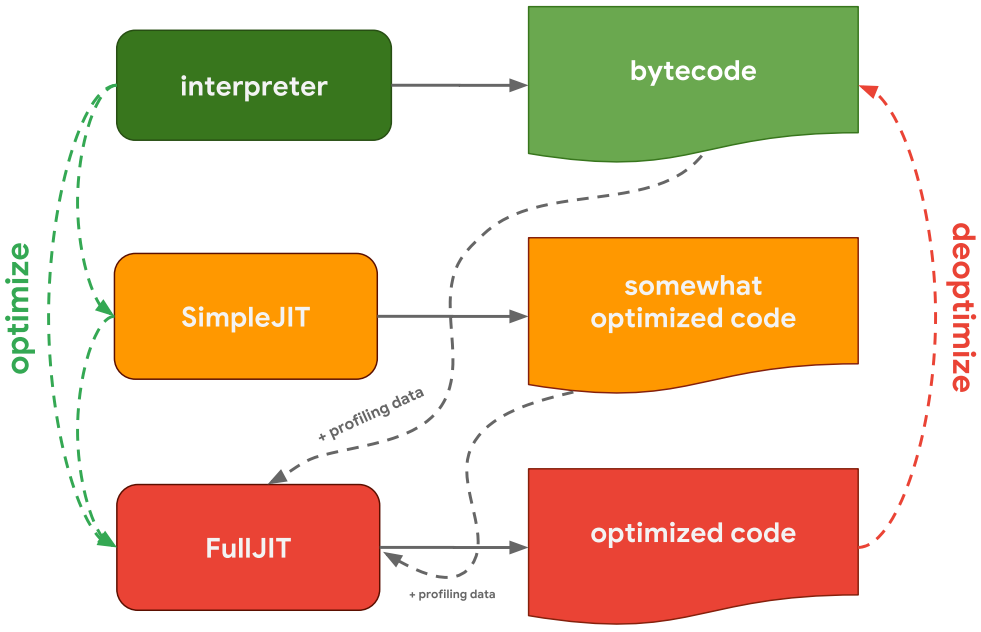
这个通用流程几乎与在 Chrome 和 Node.js 中使用的 V8 引擎工作流程一致：



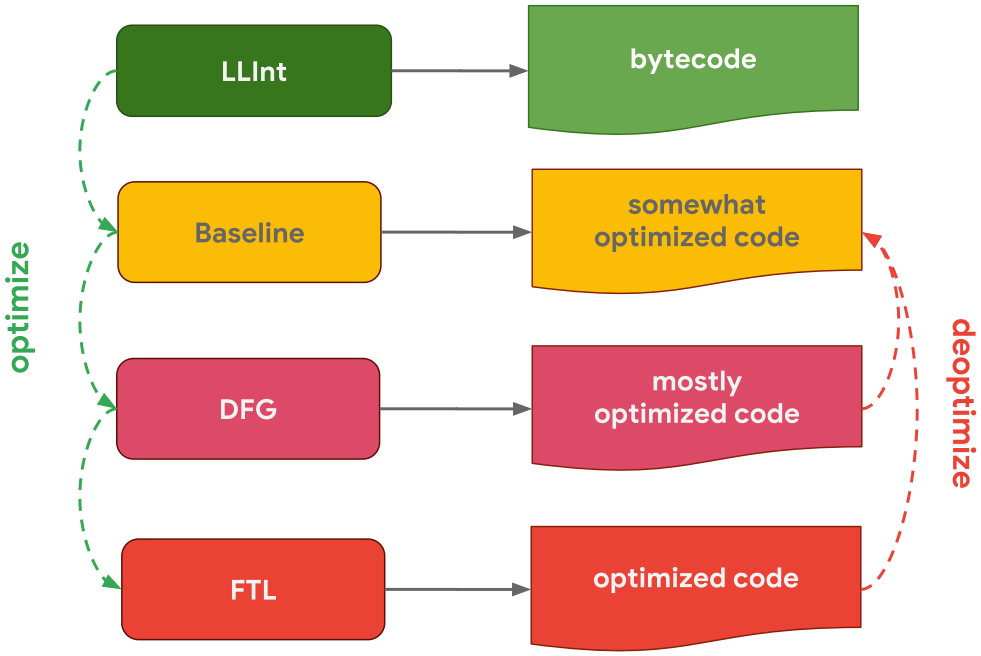
V8 中的解释器被称作 Ignition，它负责生成并执行字节码。当它运行字节码时会收集分析数据，而它之后可以被用于加快（代码）执行的速度。当一个函数变得 *hot*，例如它经常被调用，生成的字节码和分析数据则会被传给 TurboFan——我们的优化编译器，它会依据分析数据生成高度优化的机器码。



SpiderMonkey，在 Firefox 和 SpiderNode (https://github.com/mozilla/spidernode) 中使用的 Mozilla 的 JavaScript 引擎，则有一些不同的地方。它们有两个优化编译器。解释器将代码解释给 Baseline 编译器，该编译器可以生成部分优化的代码。 结合运行代码时收集的分析数据，IonMonkey 编译器可以生成高度优化的代码。 如果尝试优化失败，IonMonkey 将回退到 Baseline 阶段的代码。



Chakra，用于 Edge 和 Node-ChakraCore (https://github.com/nodejs/node-chakracore) 两个项目的微软 JavaScript 引擎，也有类似两个优化编译器的设置。解释器将代码优化成 SimpleJIT——其中 JIT 代表 Just-In-Time 编译器——它可以生成部分优化的代码。 结合分析数据，FullJIT 可以生成更深入优化的代码。



JavaScriptCore（缩写为JSC），Apple 的 JavaScript 引擎，被用于 Safari 和 React Native 两个项目中，它通过三种不同的优化编译器使效果达到极致。低级解释器 LLInt将代码解释后传递给 Baseline 编译器，而（经过 Baseline 编译器）优化后的代码便传给了 DFG 编译器，（在 DFG 编译器处理后）结果最终传给了 FTL 编译器进行处理。

为什么有些引擎会拥有更多的优化编译器呢？这完全是一些折衷的取舍。解释器可以快速生成字节码，但字节码通常不够高效。另一方面，优化编译器处理需要更长的时间，但最终会生成更高效的机器码。到底是快速获取可执行的代码（解释器），还是花费更多时间但最终以最佳性能运行代码（优化编译器），这其中包含一个平衡点。一些引擎选择添加具有不同耗时/效率特性的多个优化编译器，以更高的复杂性为代价来对这些折衷点进行更细粒度的控制。

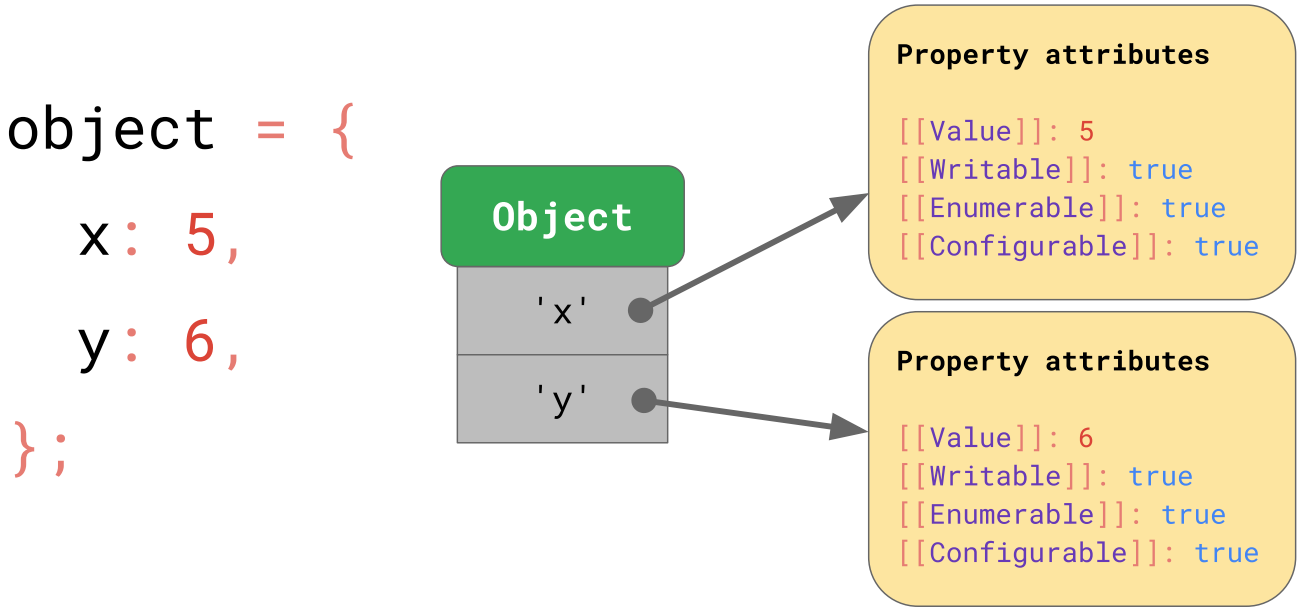
我们刚刚强调了每个 JavaScript 引擎中解释器和优化编译器流程中的主要区别。除了这些差异之外，**所有 JavaScript 引擎都有相同的架构**：那就是拥有一个解析器和某种解释器/编译器流程。

2. JavaScript 对象模型

通过关注一些方面的具体实现，让我们来看看 JavaScript 引擎间还有哪些共同之处。

例如，JavaScript 引擎是如何实现 JavaScript 对象模型的，以及他们使用了哪些技巧来加快获取 JavaScript 对象属性的速度？事实证明，所有主要引擎在这一点上的实现都很相似。

ECMAScript 规范基本上将所有对象定义为由字符串键值映射到 *property 属性 (https://tc39.github.io/ecma262/#sec-property-attributes)* 的字典。



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。

`[[双方括号]]` 的符号表示看上去有些特别，但这正是规范定义不能直接暴露给 JavaScript 的属性的表示方法。在 JavaScript 中你仍然可以通过 `Object.getOwnPropertyDescriptor` API 获得指定对象的属性值：

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

JavaScript 就是这个定义对象的，那么数组呢？

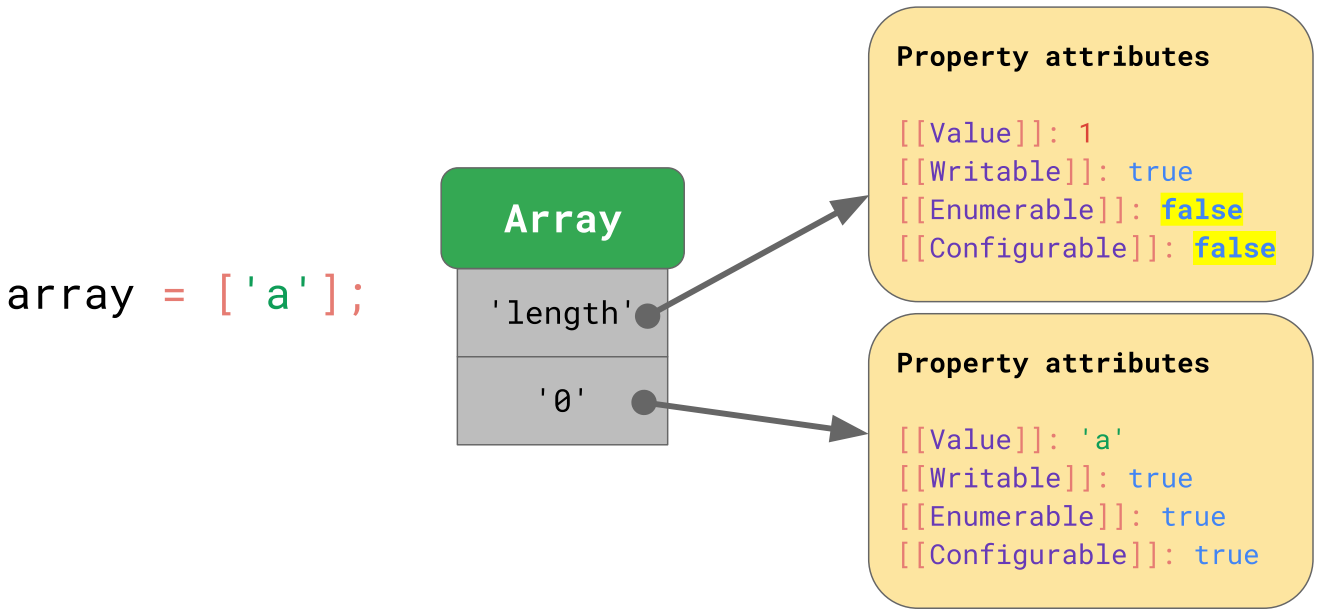
你可以将数组想象成一组特殊的对象。两者的一个区别便是数组会对数组索引进行特殊的处理。这里所指的数组索引是ECMAScript 规范中的一个特殊术语。在 JavaScript 中，数组被限制最多只能拥有2^{32}-1项。数组索引是指该限制内的任何有效索引，即从0到2^{32}-2的任何整数。

另一个区别是数组还有一个充满魔力的 `length` 属性。

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

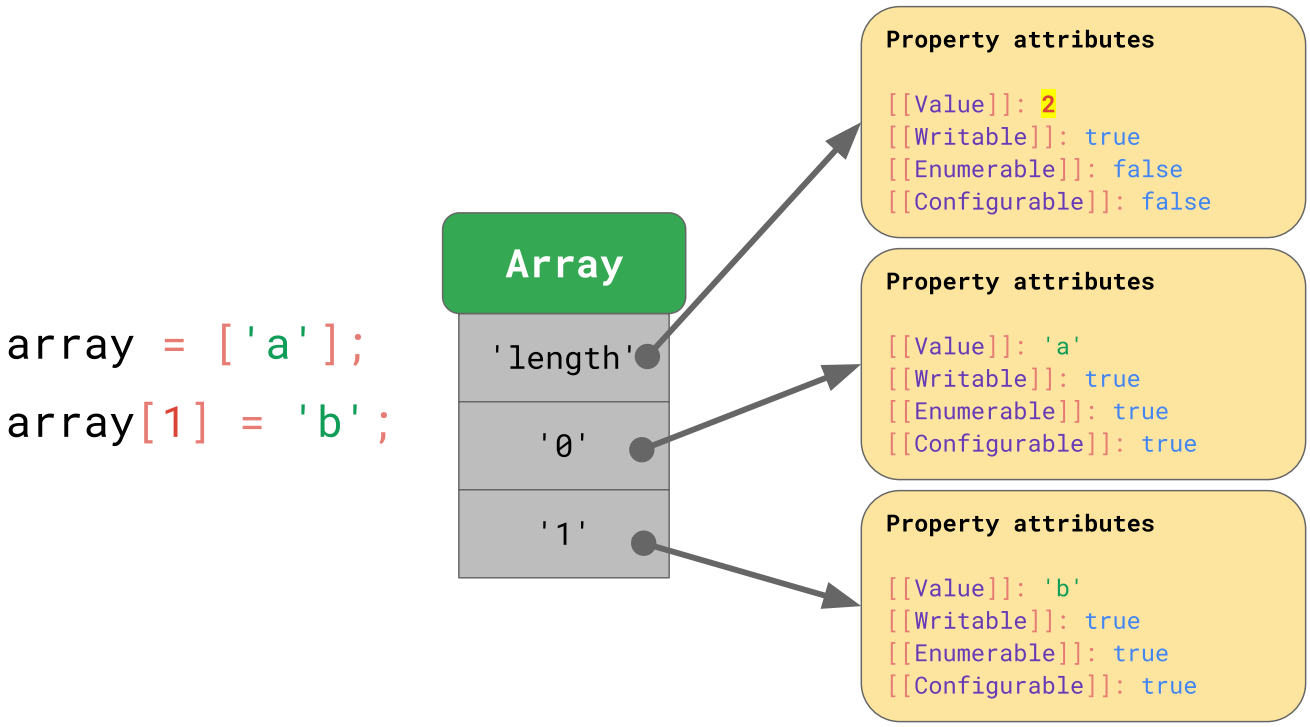
在这个例子中， `array` 在生成时长度单位为2。接着我们向索引为 `2` 的位置分配了另一个元素， `length` 属性便自动更新。

JavaScript 在定义数组的方式上和对象类似。例如，包括数组索引的所有键值都明确地表示为字符串。 数组中的第一个元素存储在键值为 '0' 的位置下。



`'length'` 属性恰好是另一个不可枚举且不可配置的属性。

一个元素一旦被添加到数组中，JavaScript 便会自动更新 `'length'` 属性的 `[[Value]]` 属性值。



一般来说，数组的行为与对象也非常相似。

3. 属性访问的优化

让我们深入了解下 JavaScript 引擎是如何有效地应对对象相关操作的。

观察 JavaScript 程序，访问属性是最常见的一个操作。使得 JavaScript 引擎能够快速获取属性便至关重要。

```
const object = {
        foo: 'bar',
        baz: 'qux',
};

// Here, we're accessing the property `foo` on `object`:
doSomething(object.foo);
//          ^^^^^^^^^^
```

3.1 Shapes

在 JavaScript 程序中，多个对象具有相同的键值属性是非常常见的。这些对象都具有相同的形状。

```
const object1 = { x: 1, y: 2 };
const object2 = { x: 3, y: 4 };
// `object1` and `object2` have the same shape.
```

访问具有相同形状对象的相同属性也很常见：

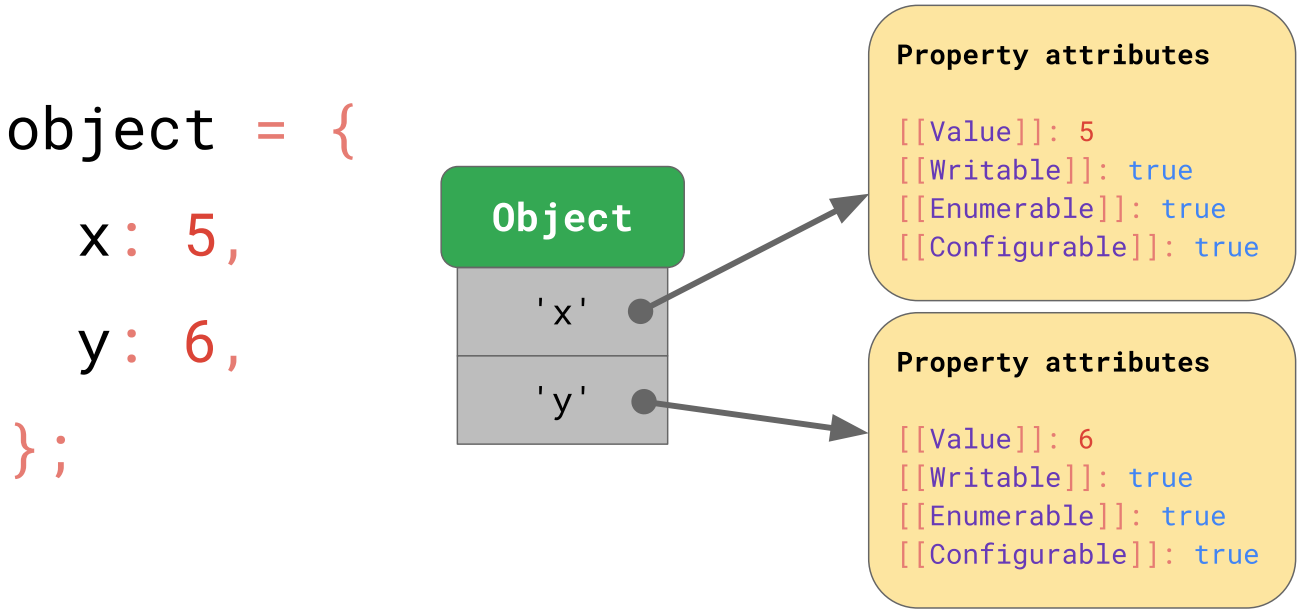
```
function logX(object) {
        console.log(object.x);
        //          ^^^^^^^^
}

const object1 = { x: 1, y: 2 };
const object2 = { x: 3, y: 4 };

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

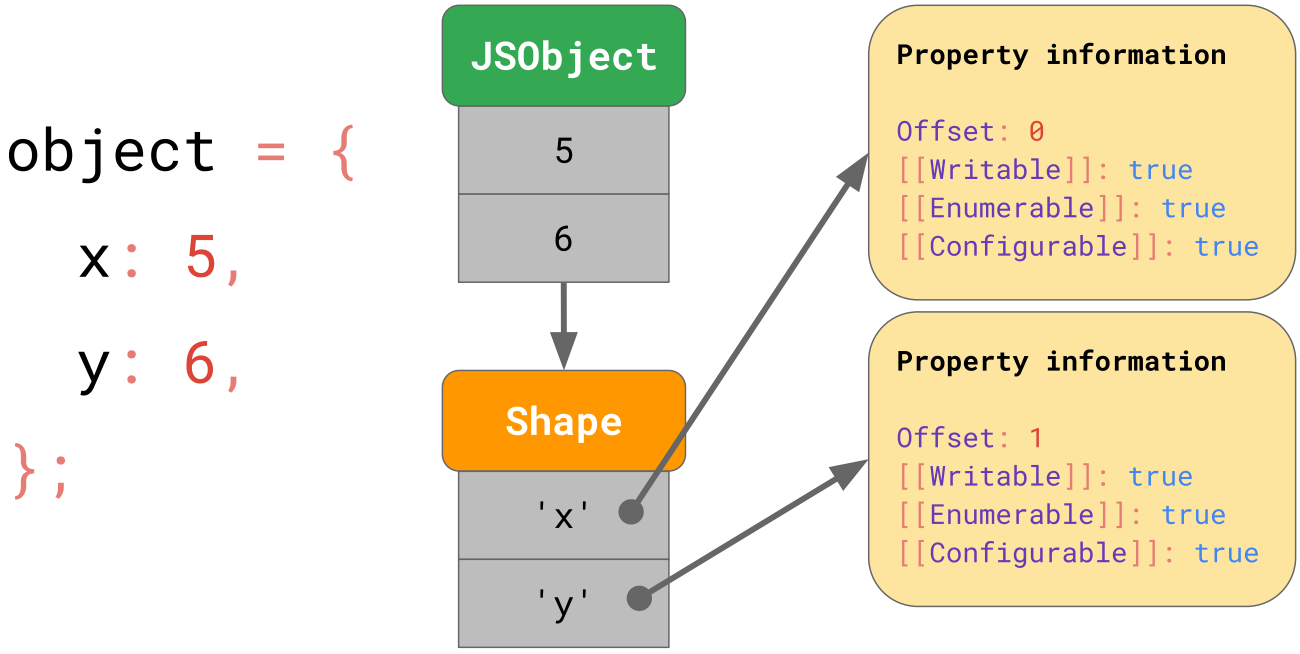
考虑到这一点，JavaScript 引擎可以根据对象的形状来优化对象的属性获取。它是这么实现的。

假设我们有一个具有属性 `x` 和 `y` 的对象，它使用我们前面讨论过的字典数据结构：它包含用字符串表示的键值，而它们指向各自的属性值。

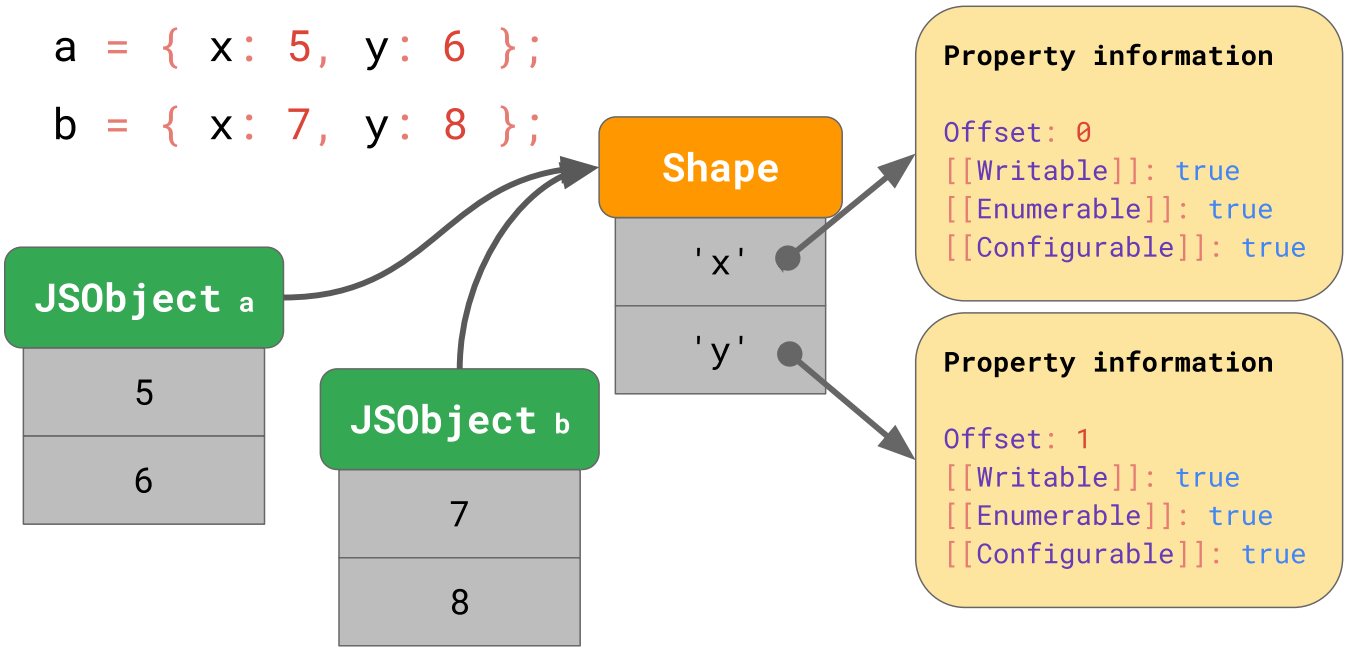


如果你访问某个属性，例如 `object.y`，JavaScript 引擎会在 `JSObject` 中查找键值 `'y'`，然后加载相应的属性值，最后返回 `[[Value]]`。

但这些属性值在内存中是如何存储的呢？我们是否应该将它们存储为 `JSObject` 的一部分？假设我们稍后会遇到更多同形状的对象，那么在 `JSObject` 自身存储包含属性名和属性值的完整字典便是很浪费（空间）的，因为对具有相同形状的所有对象我们都重复了一遍属性名称。 它太冗余且引入了不必要的内存使用。 作为优化，引擎将对象的 `Shape` 分开存储。



`Shape` 包含除 `[[Value]]` 之外的所有属性名和其余特性。相反， `Shape` 包含 `JSObject` 内部值的偏移量，以便 JavaScript 引擎知道去哪查找具体值。每个具有相同形状的 `JSObject` 都指向这个 `Shape` 实例。 现在每个 `JSObject` 只需要存储对这个对象来说唯一的那些值。



当我们有多个对象时，优势变得清晰可见。无论有多少个对象，只要它们具有相同的形状，我们只需要将它们的形状与键值属性信息存储一次！

所有的 JavaScript 引擎都使用了形状作为优化，但称呼各有不同：

- 学术论文称它们为 *Hidden Classes*（容易与 JavaScript 中的类概念混淆）
- V8 将它们称为 *Maps*（容易与 JavaScript 中的 `Map` 概念混淆）
- Chakra 将它们称为 *Types*（容易与 JavaScript 中的动态类型和关键字 `typeof` 混淆）
- JavaScriptCore 称它们为 *Structures*
- SpiderMonkey 称他们为 *Shapes*

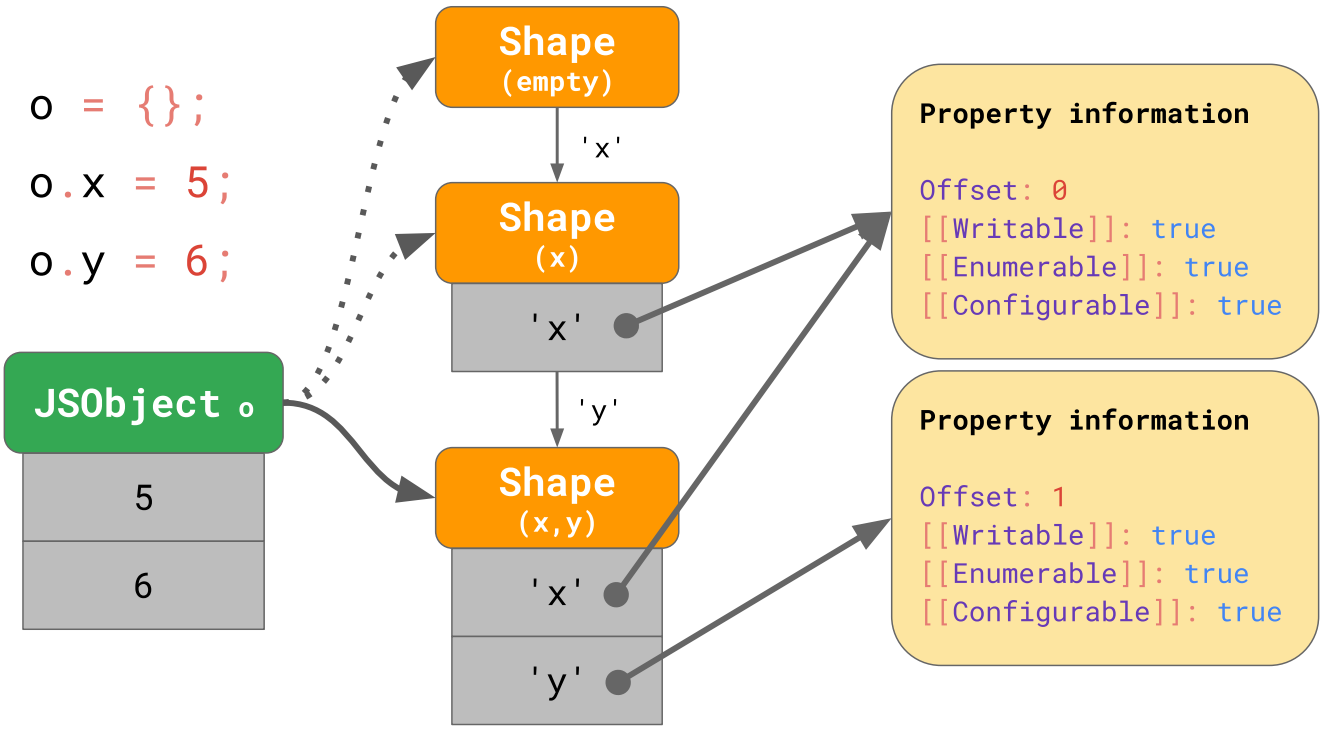
本文中，我们会继续称它为 *shapes*。

3.2 Transition 链与树

如果你有一个具有特定形状的对象，但你又向它添加了一个属性，此时会发生什么？ JavaScript 引擎是如何找到这个新形状的?

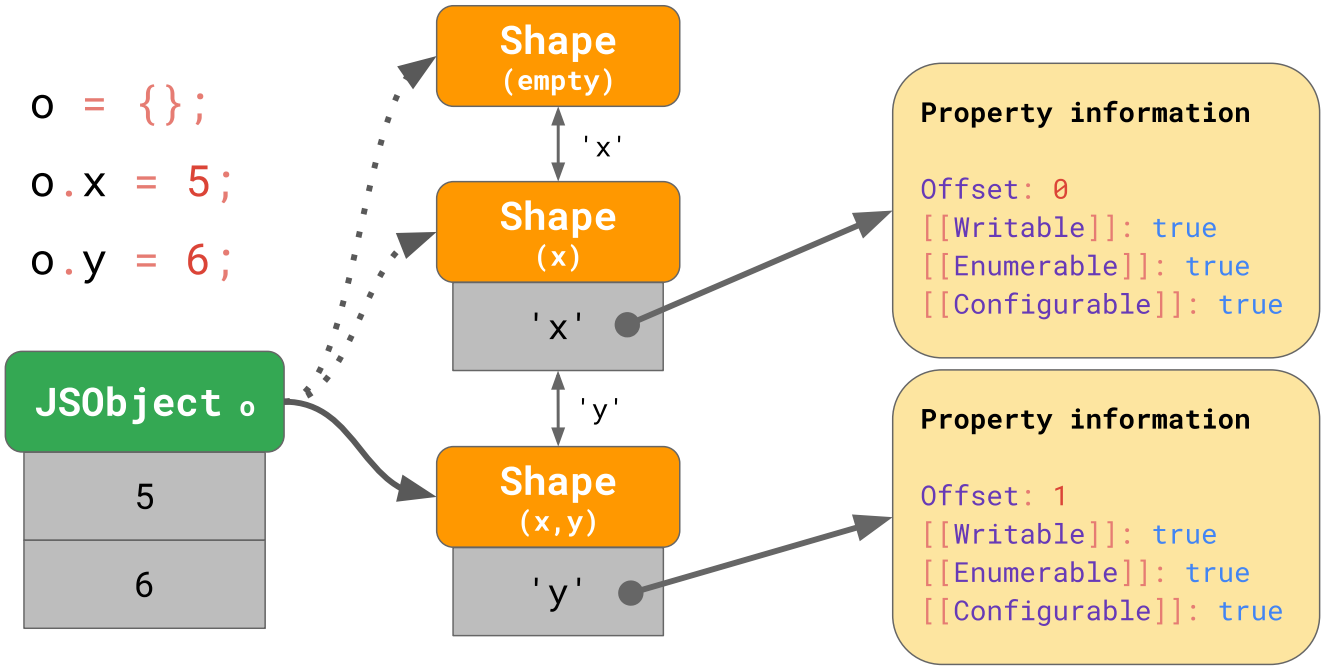
```
const object = {};
object.x = 5;
object.y = 6;
```

在 JavaScript 引擎中，shapes 的表现形式被称作 *transition 链*。以下展示一个示例：



该对象在初始化时没有任何属性，因此它指向一个空的 shape。下一个语句为该对象添加值为 `5` 的属性 “x”，所以 JavaScript 引擎转向一个包含属性 “x” 的 Shape，并向 `JSObject` 的第一个偏移量为0处添加了一个值 `5`。 接下来一个语句添加了一个属性 `'y'`，引擎便转向另一个包含 `'x'` 和 `'y'` 的 Shape，并将值 `6` 附加到 `JSObject` （位于偏移量 `1` 处）。

我们甚至不需要为每个 Shape 存储完整的属性表。相反，每个 Shape 只需要知道它引入的新属性。 例如在此例中，我们不必在最后一个 Shape 中存储关于 `'x'` 的信息，因为它可以在更早的链上被找到。要做到这一点，每一个 Shape 都会与其之前的 Shape 相连：

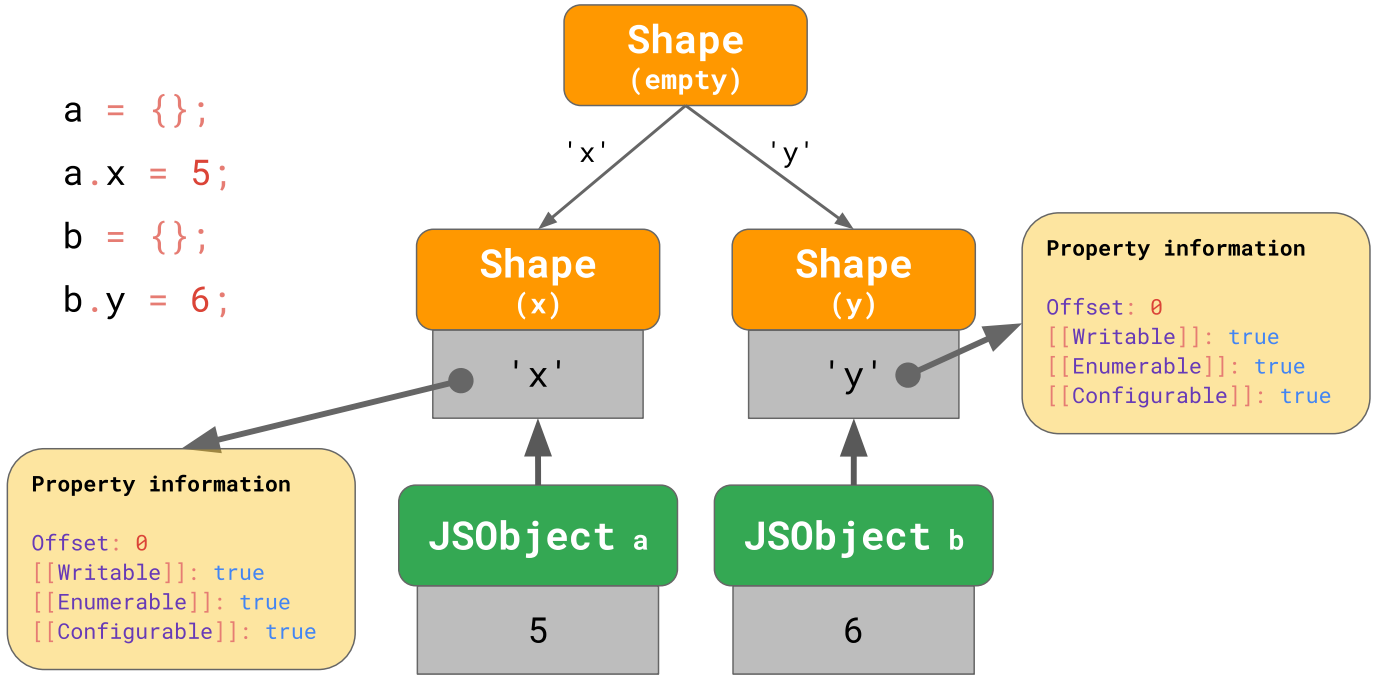


如果你在 JavaScript 代码中写到了 `o.x`，则 JavaScript 引擎会沿着 transition 链去查找属性 “x”，直到找到引入属性 “x” 的 Shape。

但是，如果不能只创建一个 transition 链呢？例如，如果你有两个空对象，并且你为每个对象都添加了一个不同的属性？

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

在这种情况下我们便必须进行分支操作，此时我们最终会得到一个 *transition 树* 而不是 transition 链：



在这里，我们创建一个空对象 `a`，然后为它添加一个属性 `'x'`。 我们最终得到一个包含单个值的 `JSObject`，以及两个 Shapes：空 Shape 和仅包含属性 `x` 的 Shape。

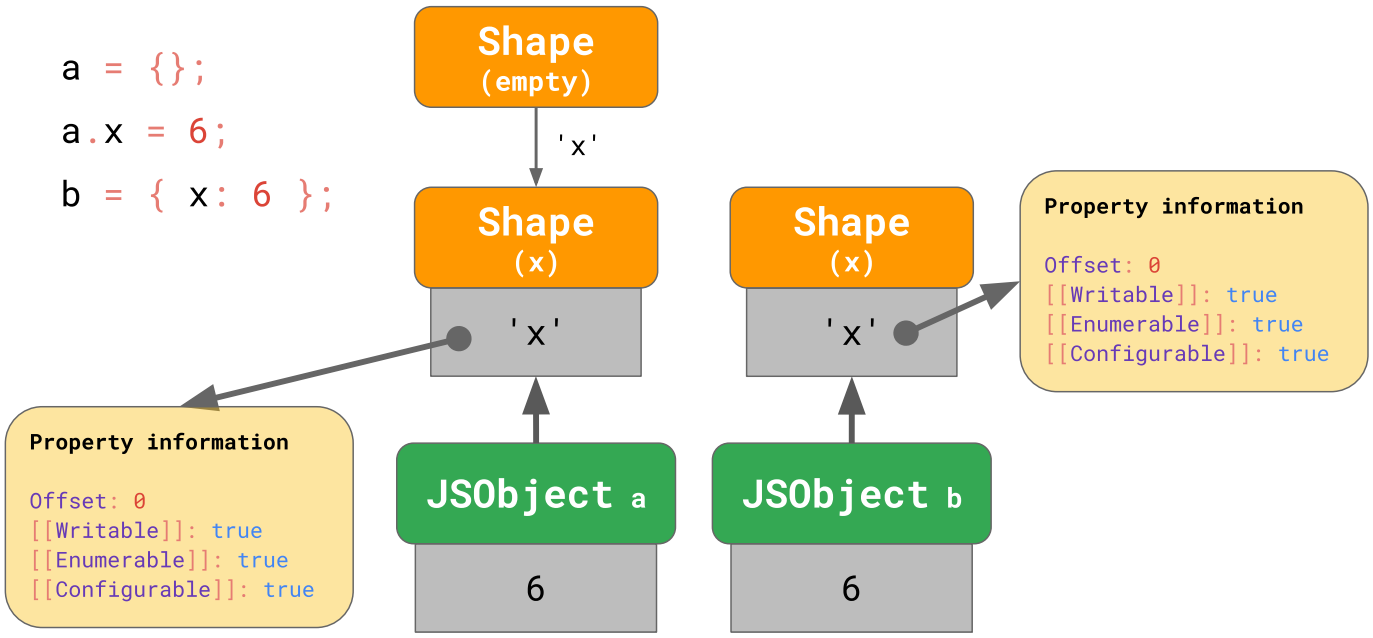
第二个例子也是从一个空对象 `b` 开始的，但之后被添加了一个不同的属性 `'y'`。我们最终形成两个 shape 链，总共是三个 shape。

这是否意味着我们总是需要从空 shape 开始呢？ 并不是。引擎对已包含属性的对象字面量会应用一些优化。比方说，我们要么从空对象字面量开始添加 `x` 属性，要么有一个已经包含属性 `x` 的对象字面量：

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

在第一个例子中，我们从空 shape 开始，然后转向包含 `x` 的 shape，这正如我们我们之前所见。

在 `object2` 一例中，直接生成具有属性 `x` 的对象是有意义的，而不是从空对象开始然后进行 transition 连接。



包含属性 `'x'` 的对象字面量从包含 `'x'` 的 shape 开始，可以有效地跳过空的 shape。V8 和 SpiderMonkey （至少）正是这么做的。这种优化缩短了 transition 链，并使得从字面量构造对象更加高效。

Benedikt 的博文 surprising polymorphism in React applications (https://medium.com/@bmeurer/surprising-polymorphism-in-react-applications-63015b50abc) 讨论了这些微妙之处是如何影响实际性能的。

3.3 Inline Caches (ICs)

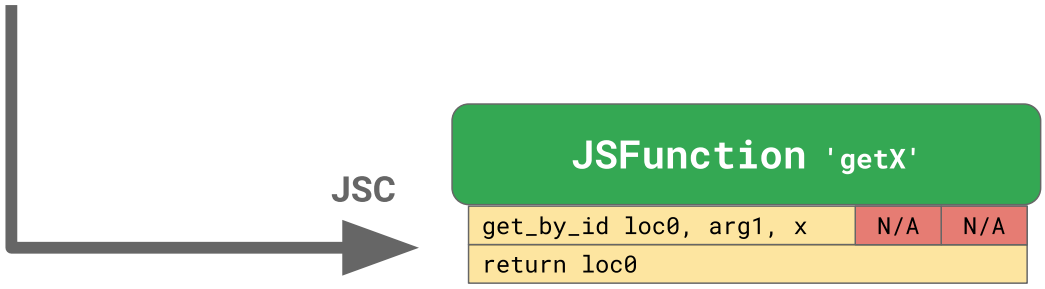
Shapes 背后的主要动机是 Inline Caches 或 ICs 的概念。ICs 是促使 JavaScript 快速运行的关键因素！JavaScript 引擎利用 ICs 来记忆去哪里寻找对象属性的信息，以减少昂贵的查找次数。

这里有一个函数 `getX`，它接受一个对象并从中取出属性 `x` 的值：

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

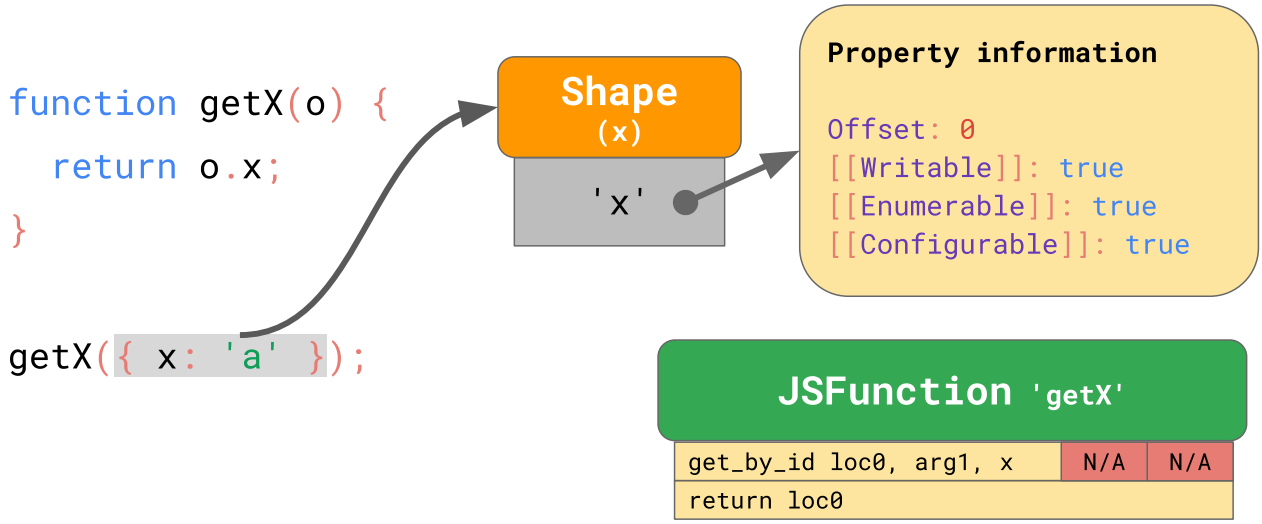
如果我们在 JSC 中执行这个函数，它会生成如下字节码：

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

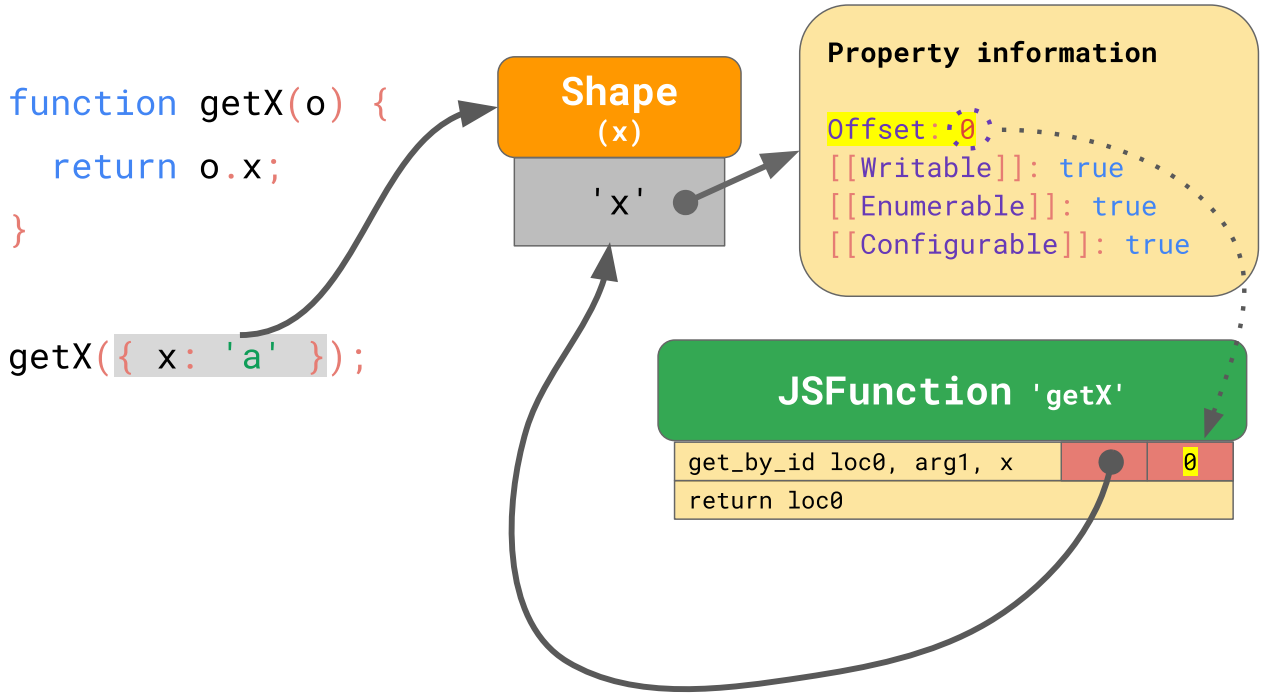


指令一 `get_by_id` 从第一个参数（`arg1`）中加载属性 `'x'` 值并将其存储到地址 `loc0` 中。 第二条指令返回我们存储到 `loc0` 中的内容。

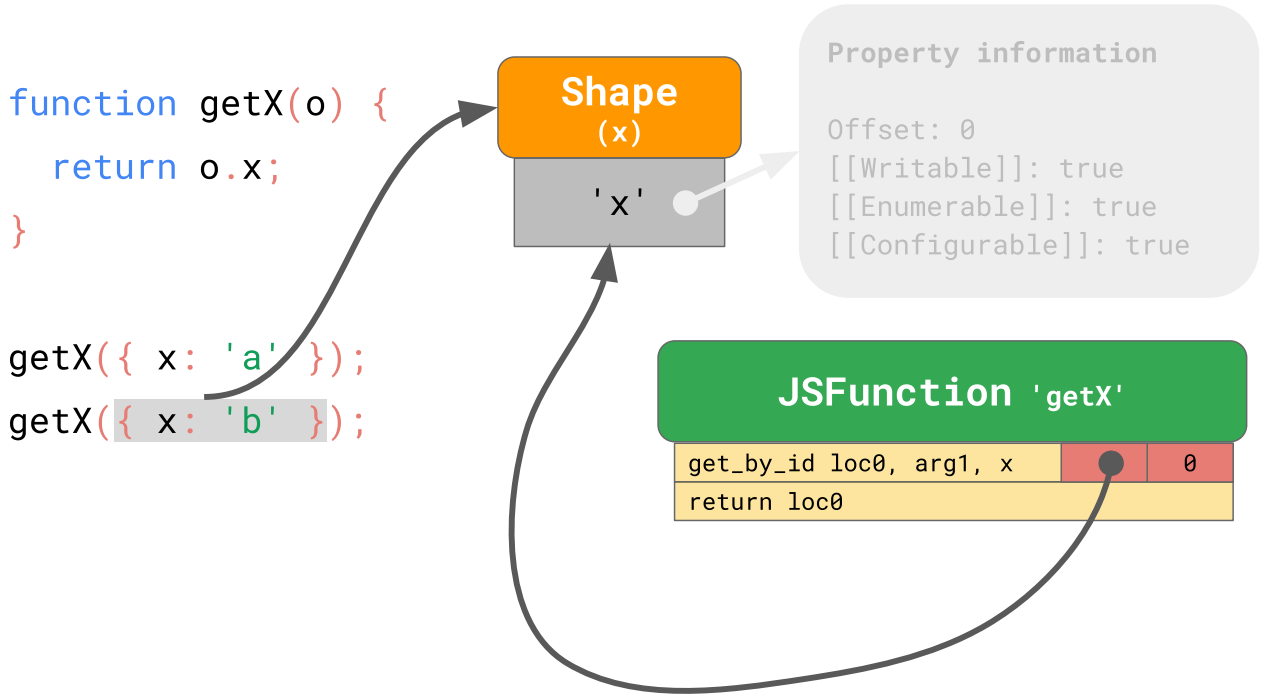
JSC 还在 `get_by_id` 指令中嵌入了 Inline Cache，它由两个未初始化的插槽组成。



现在让我们假设我们用对象 `{x: 'a'}` 调用 `getX` 函数。正如我们所知，这个对象有一个包含属性 `'x'` 的 Shape，该 Shape 存储了属性 `x` 的偏移量和其他特性。当你第一次执行该函数时，`get_by_id` 指令将查找属性 `'x'`，然后发现其值存储在偏移量 `0` 处。



嵌入到 `get_by_id` 指令中的 IC 存储该属性的 shape 和偏移量：



对于后续运行，IC 只需要对比 shape，如果它与以前相同，只需从记忆的偏移量处加载该属性值。具体来说，如果 JavaScript 引擎看到一个对象的 shape 之前被 IC 记录过，它则不再需要接触属性信息——而是完全可以跳过昂贵的属性信息查找（过程）。这比每次查找属性要快得多。

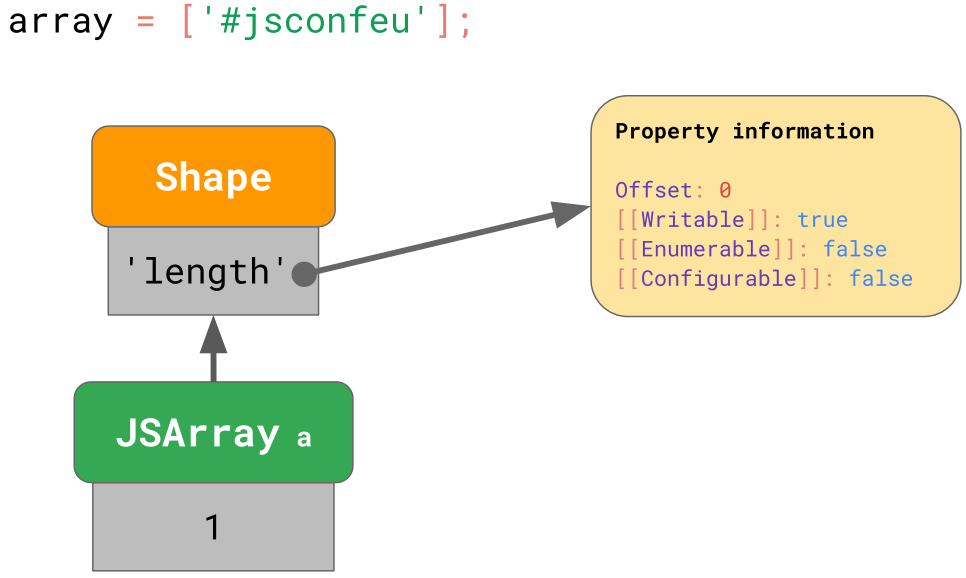
4. 高效存储数组

对于数组来说，存储属性诸如*数组索引*等是非常常见的。这些属性的值被称为数组元素。存储每个数组中的每个数组元素的属性特性（property attributes）将是一种很浪费的存储方式。相反，由于数组索引默认属性是可写的、可枚举的并且可以配置的，JavaScript 引擎利用这一点，将数组元素与其他命名属性分开存储。

考虑这个数组：

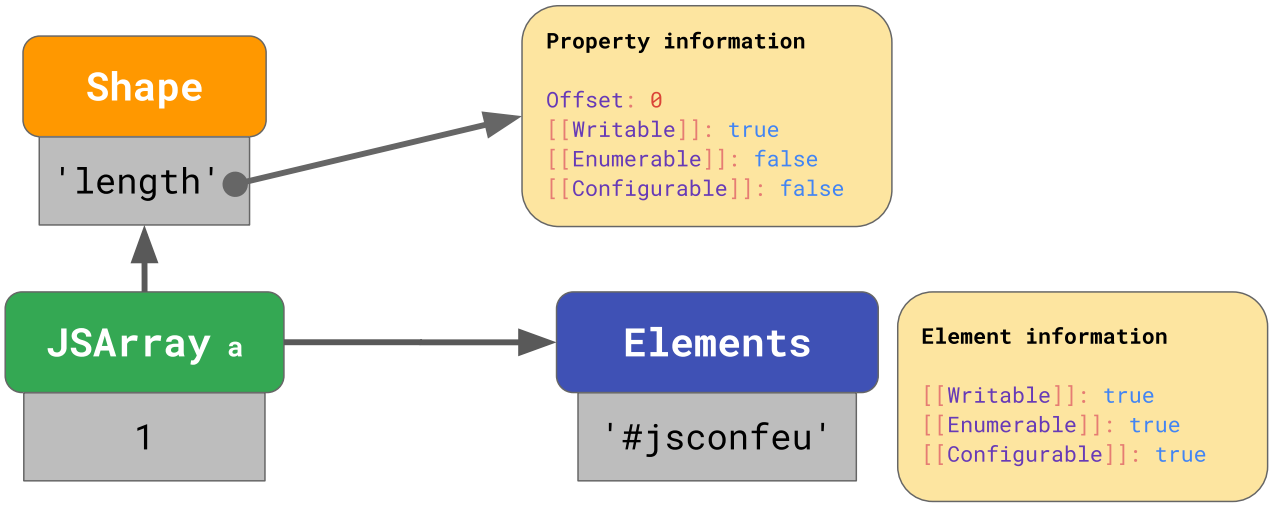
```
const array = [
        '#jsconfeu',
];
```

引擎存储了数组长度（`1`），并指向包含 `offset` 和 `'length'` 特性属性的 Shape。



这与我们之前见过的类似……但数组值存储在哪里呢？

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



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。

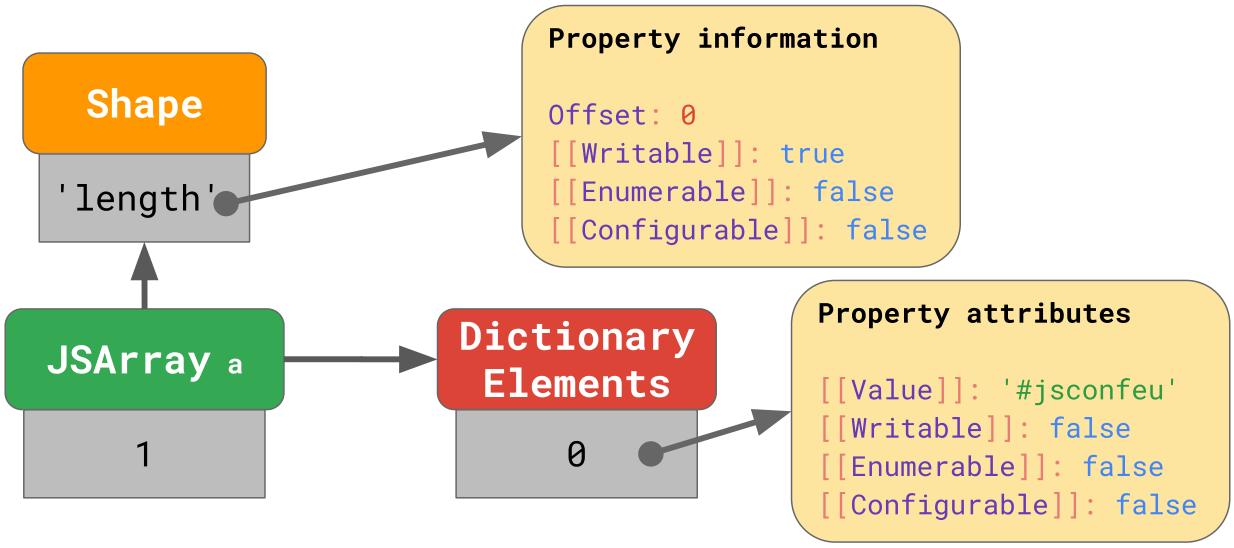
那么如果不是通常的情况呢？如果更改了数组元素的属性，该怎么办？

```
// Please don't ever do this!
const array = Object.defineProperty(
        [],
        '0',
        {
                value: 'Oh noes!!1',
                writable: false,
                enumerable: false,
                configurable: false,
        }
);
```

上面的代码片段定义了一个名为 `'0'` 的属性（这恰好是一个数组索引），但其特性（ `value` ）被设置为了一个非默认值。

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。

```
array = Object.defineProperty([], '0', { … });
```



即使只有一个数组元素具有非默认属性，整个数组的 backing store 处理也会进入这种缓慢而低效的模式。 **避免在数组索引上使用 `Object.defineProperty`!** （我不知道为什么你会想这样做。这看上去似乎是一个奇怪的且毫无价值的事情。）

5. Take-aways

我们已经学习了 JavaScript 引擎是如何存储对象和数组的，以及 Shapes 和 IC 是如何优化针对它们的常见操作的。基于这些知识，我们确定了一些有助于提升性能的实用 JavaScript 编码技巧：

- 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。
- 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。

（完）

Tags: 2018 Cache JavaScript Shapes 前端 引擎

Categories: Document

Updated: June 17, 2018

LEAVE A COMMENT

0 (https://github.com/hijiangtao/hijiangtao.github.io.comments/issues/15) comments

Anonymous

Leave a comment

Markdown is supported (https://guides.github.com/features/mastering-markdown/)

Preview

Login with GitHub

Be the first person to leave a comment!