你好，我是李兵。

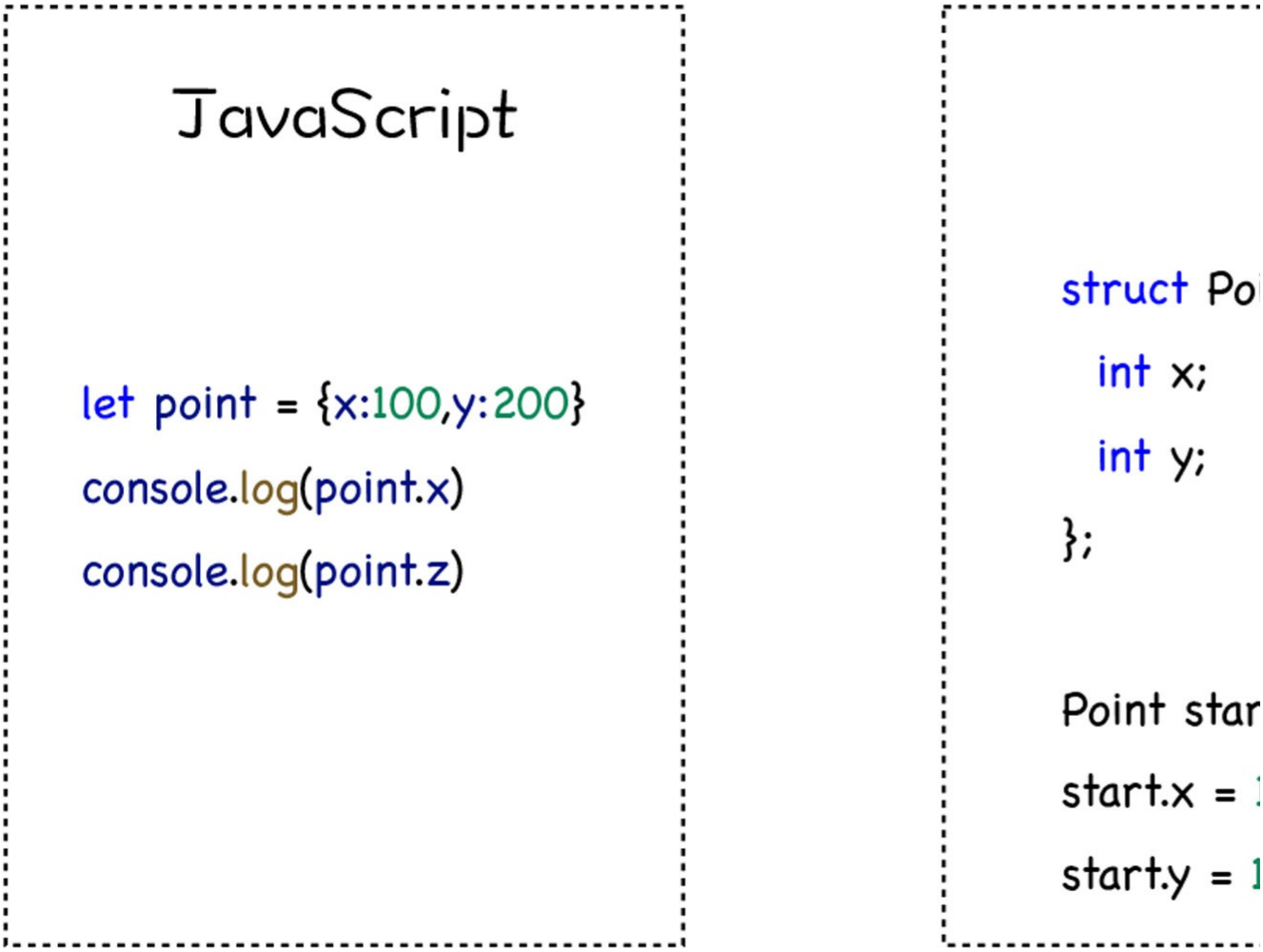
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：



那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

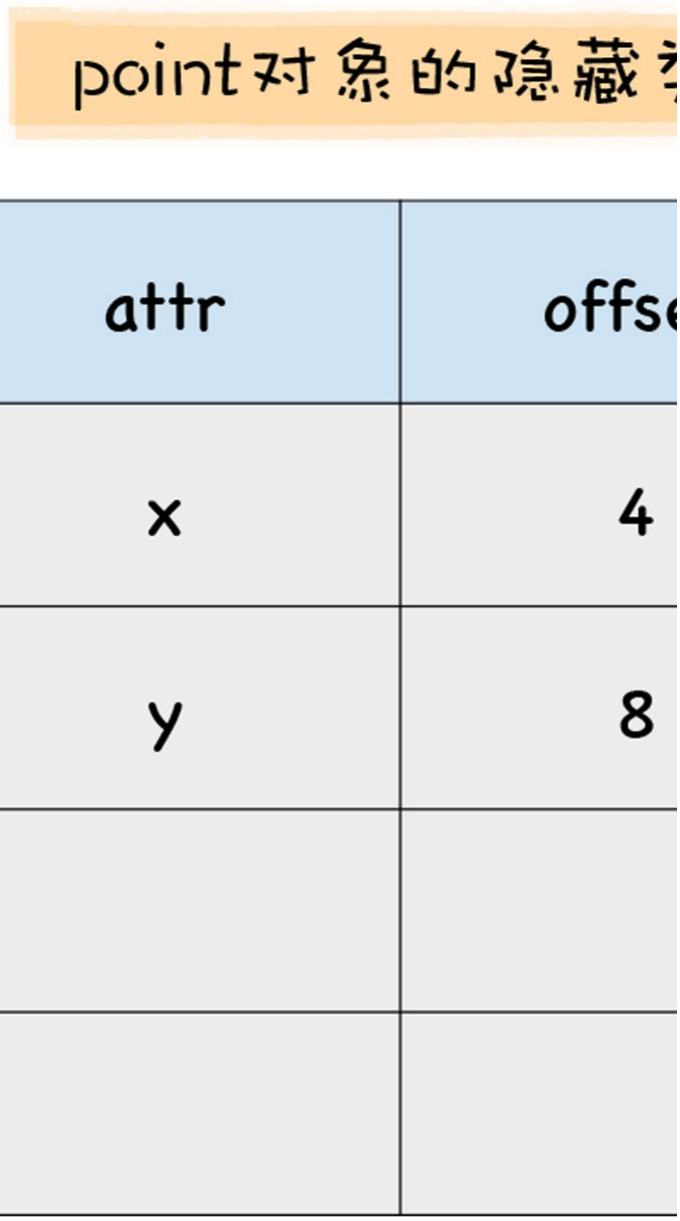
有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

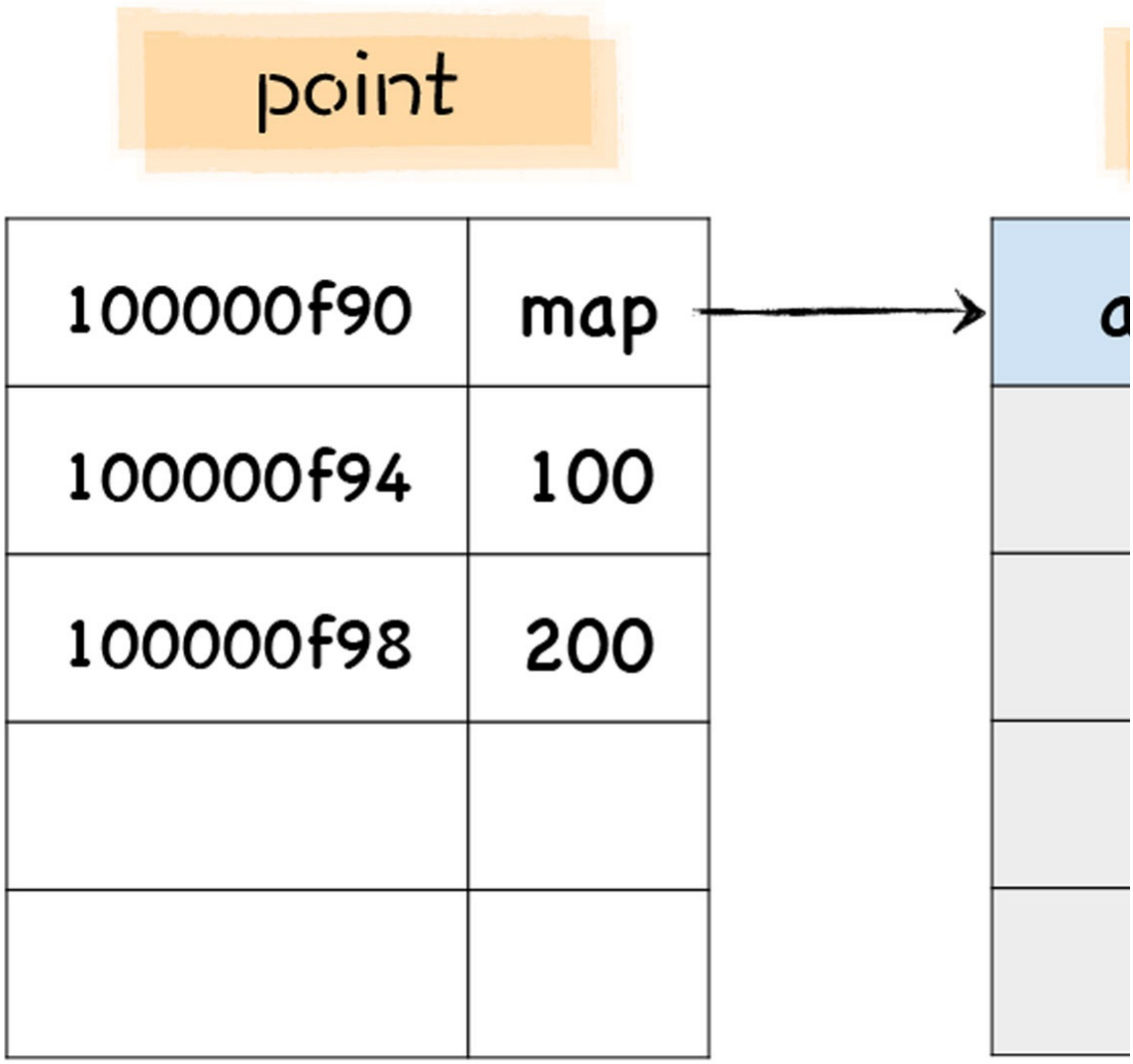
```
let point = {x:100,y:200}
```

当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。



注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

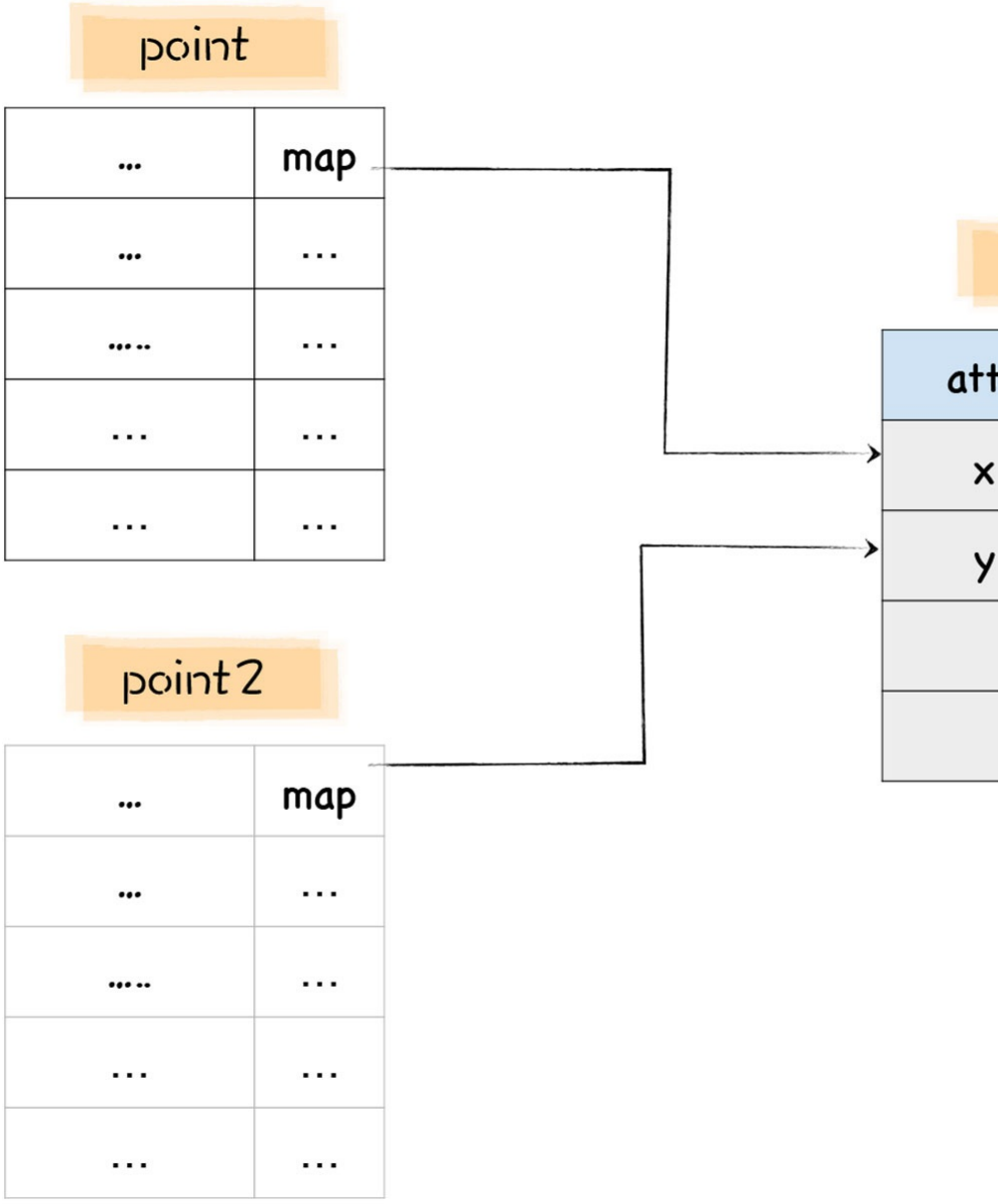
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

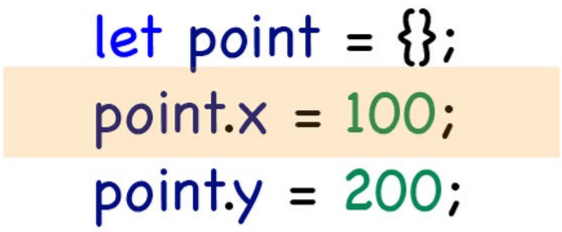
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

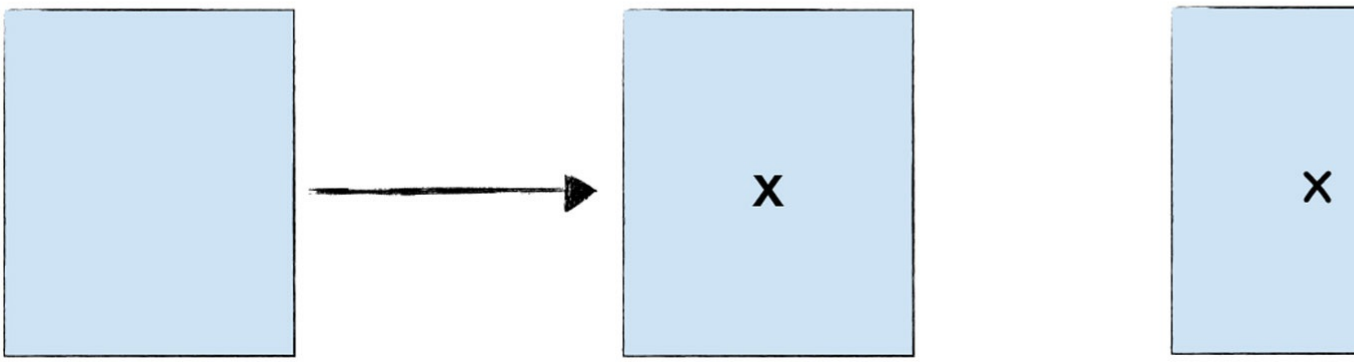
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

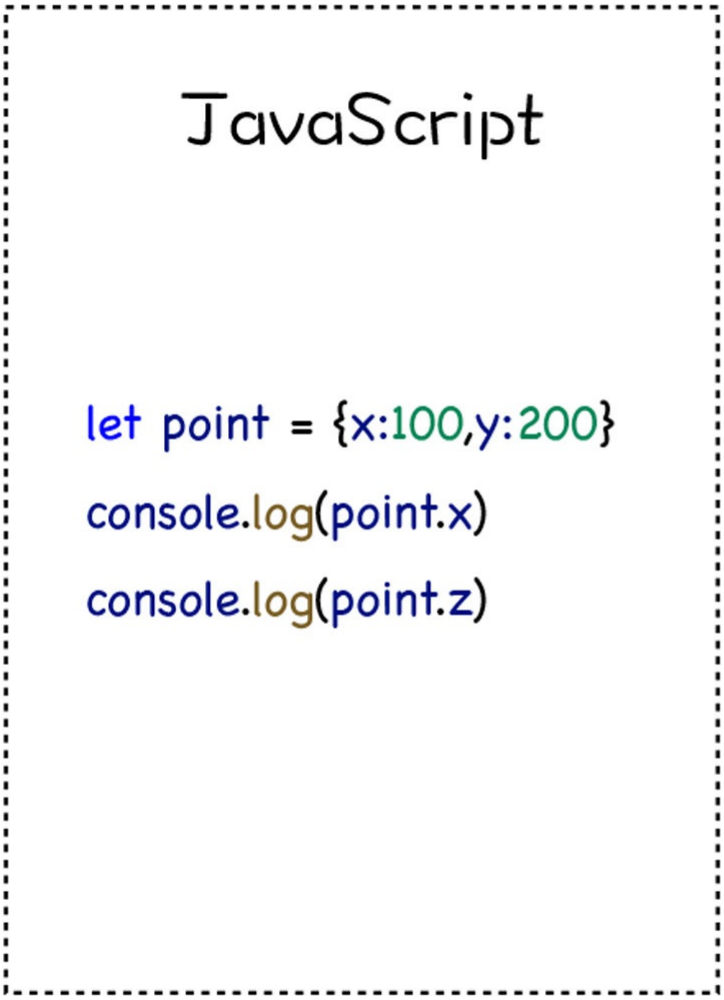
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

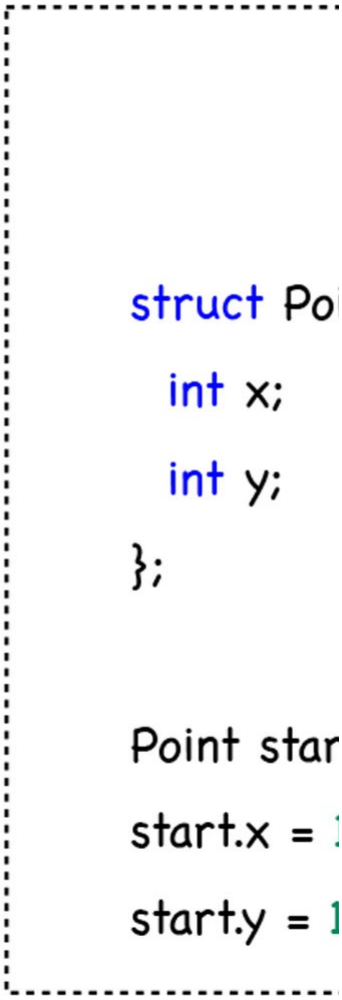
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

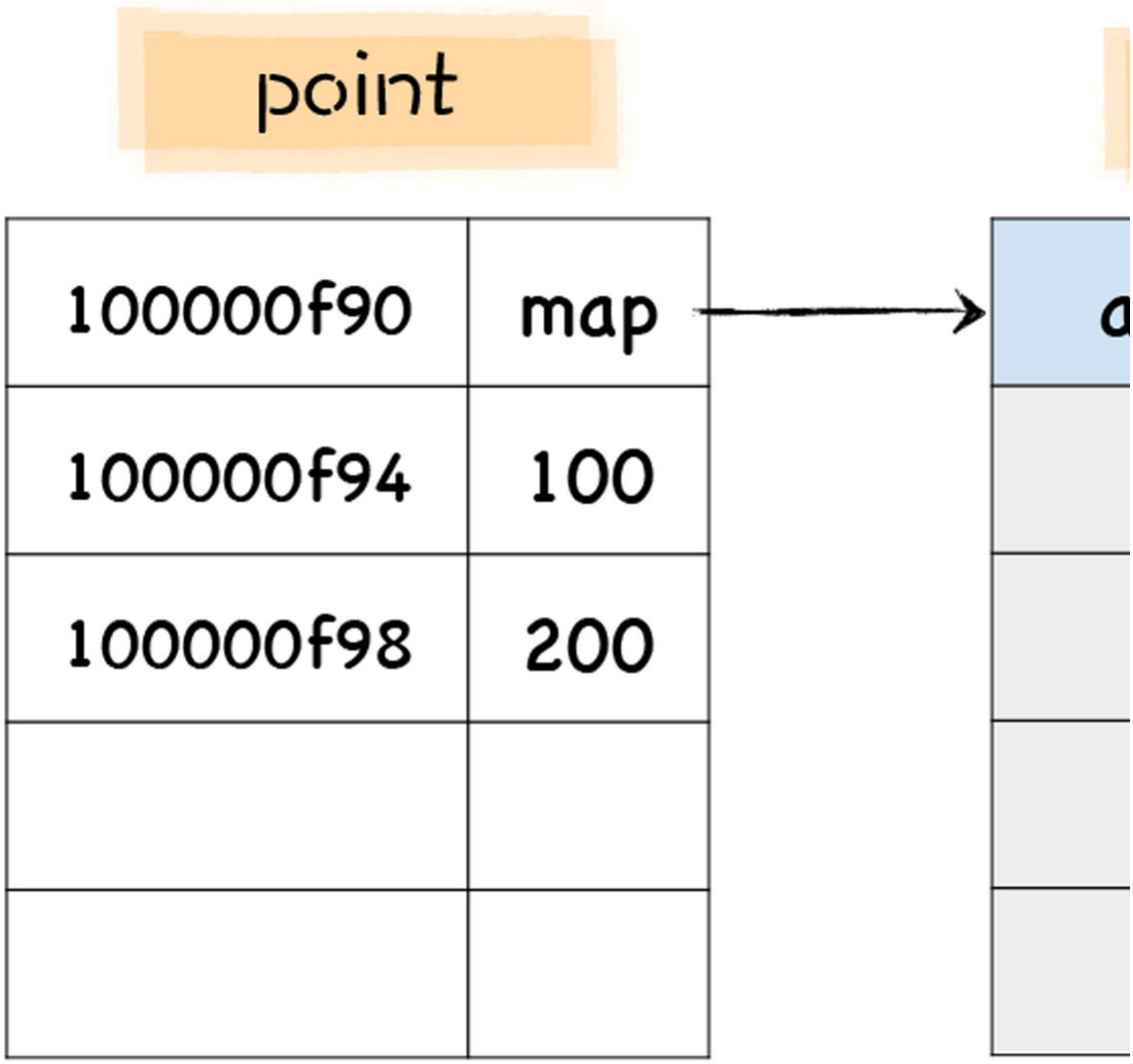
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

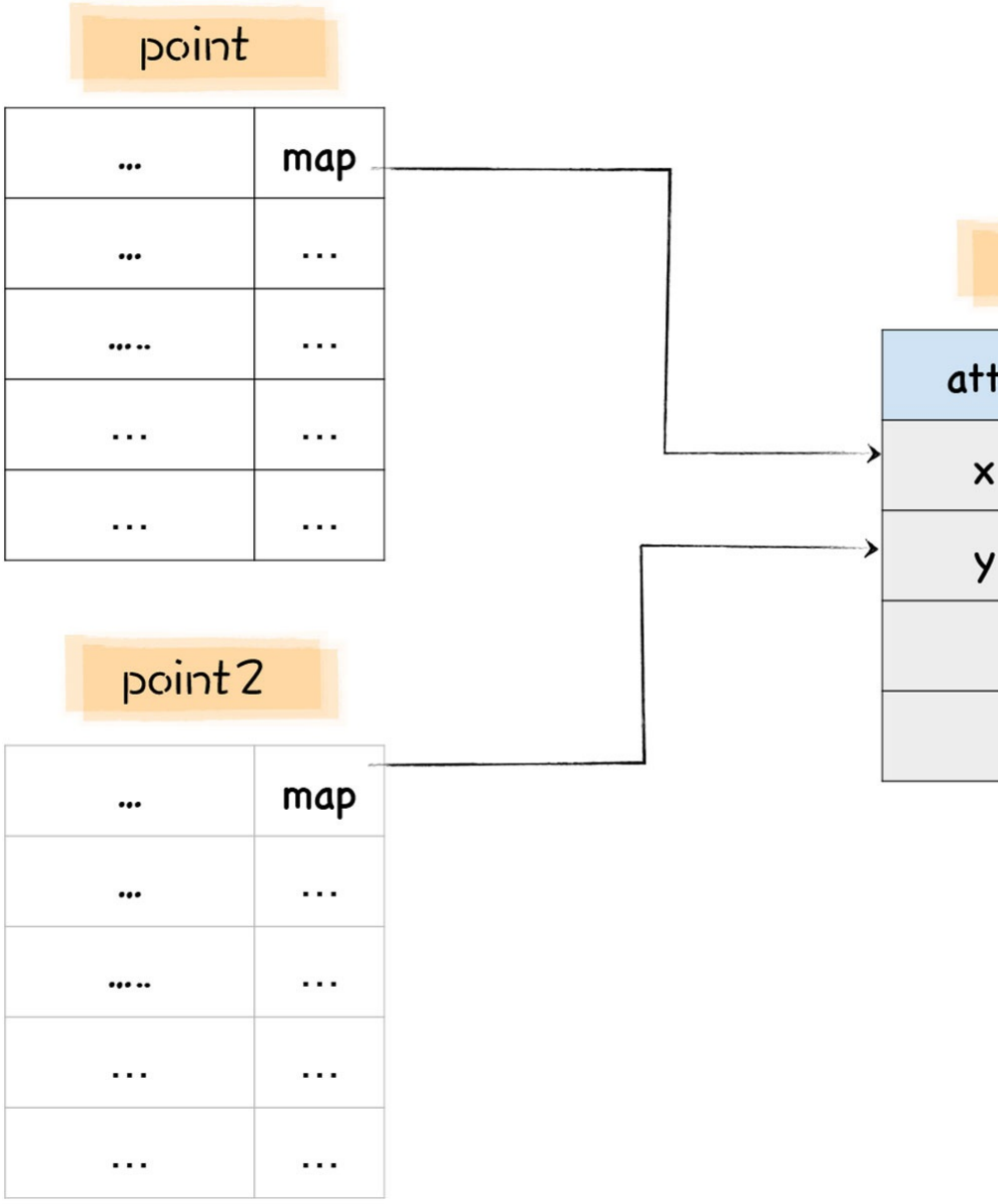
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

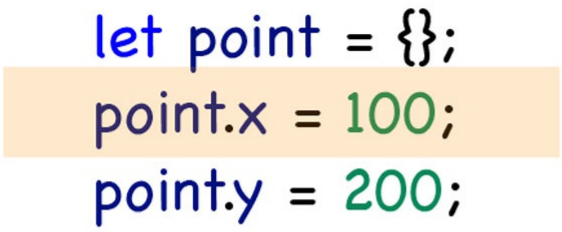
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

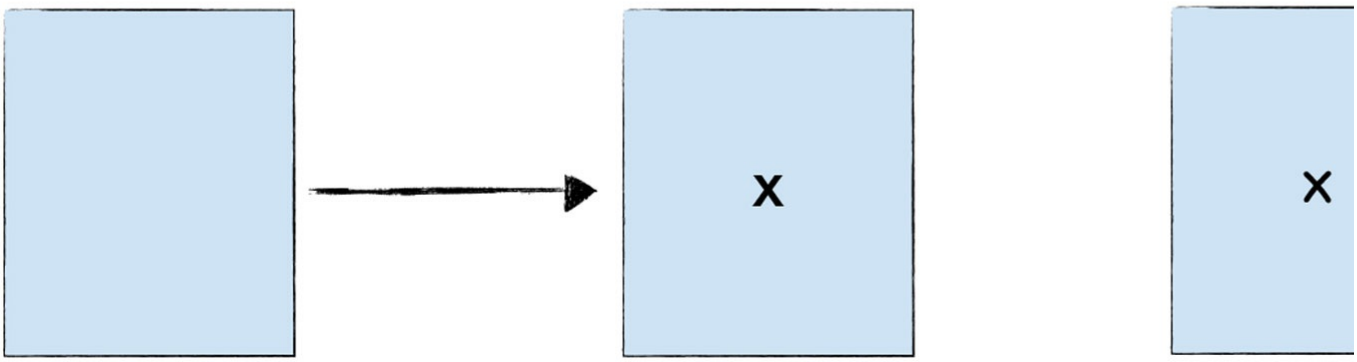
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

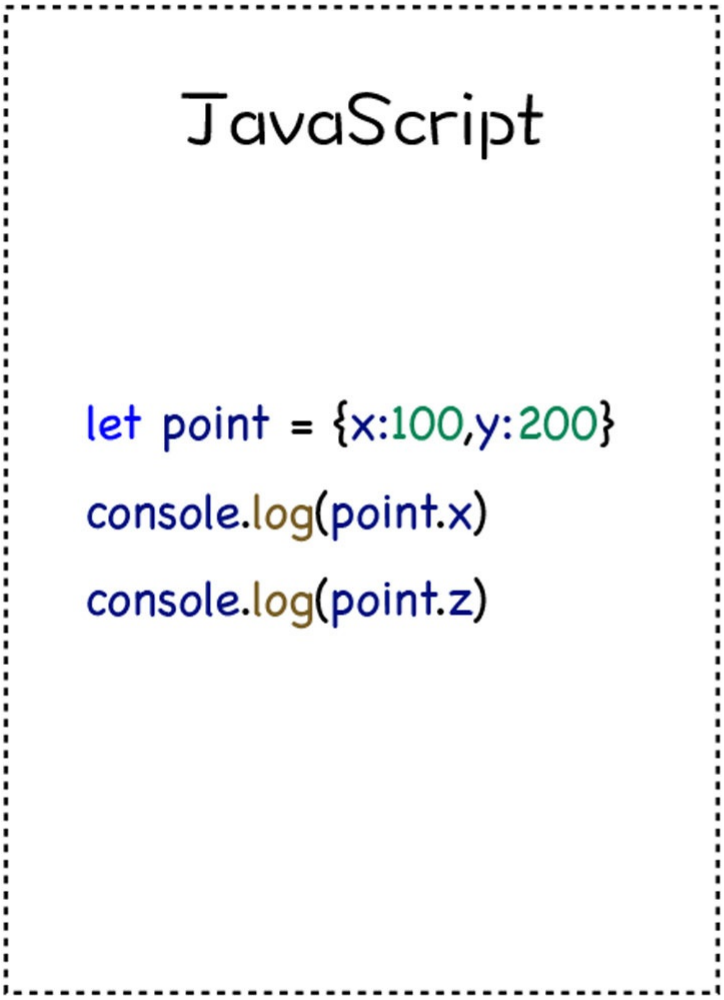
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

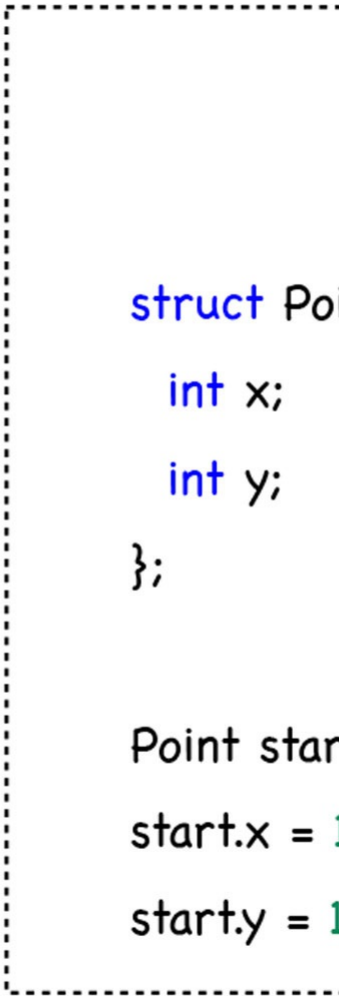
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

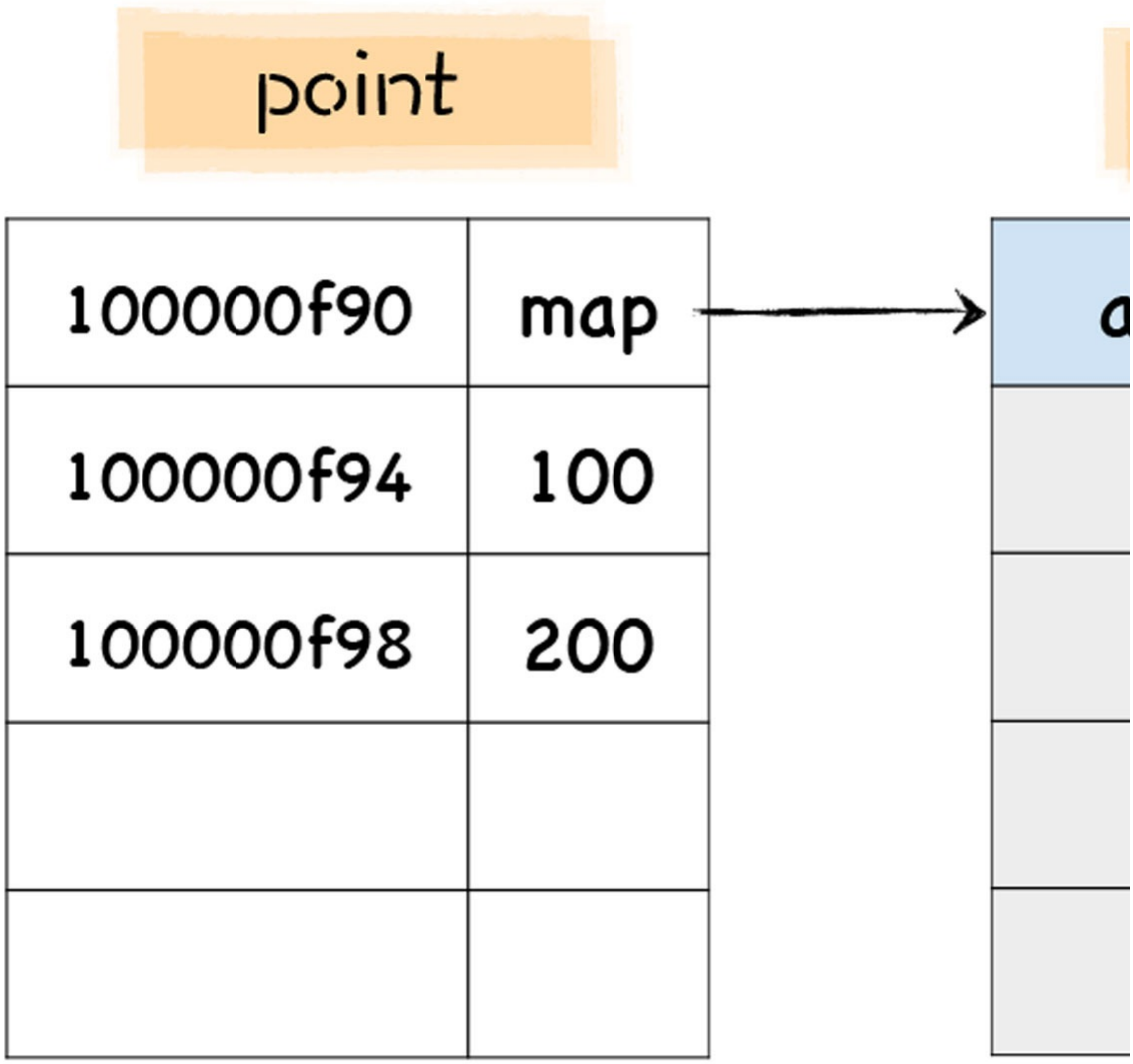
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

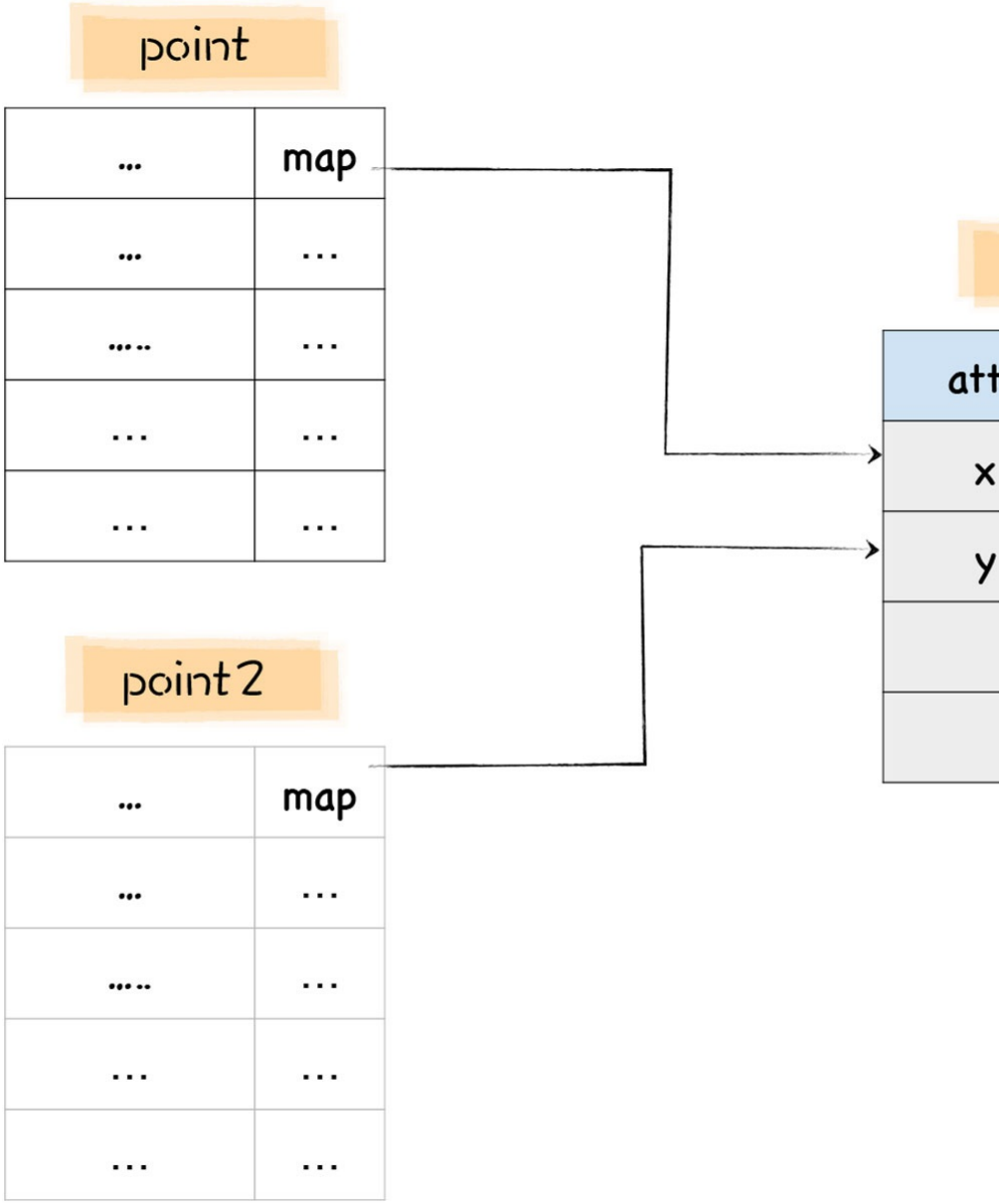
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

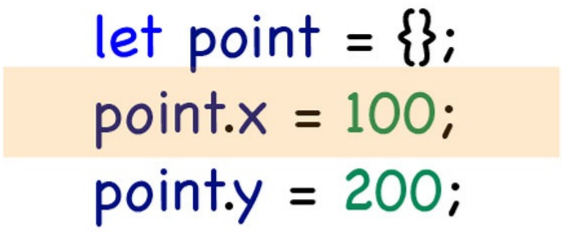
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

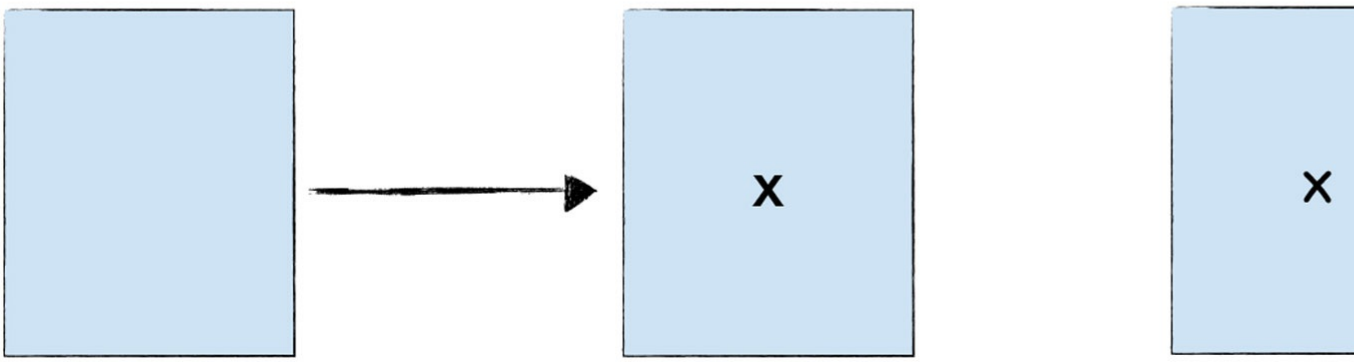
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

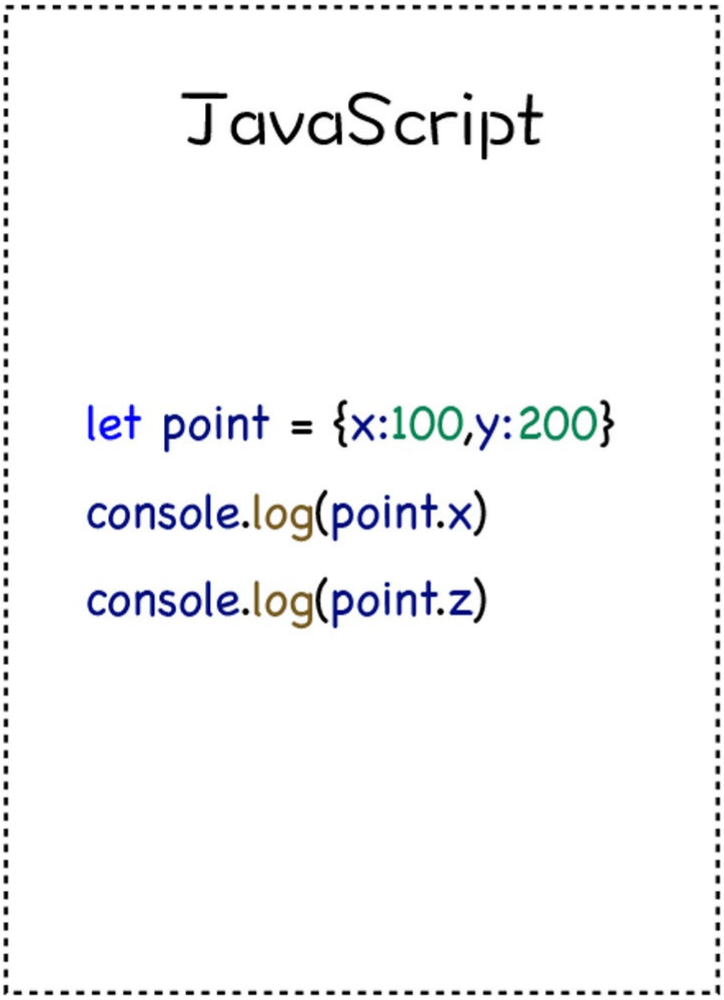
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

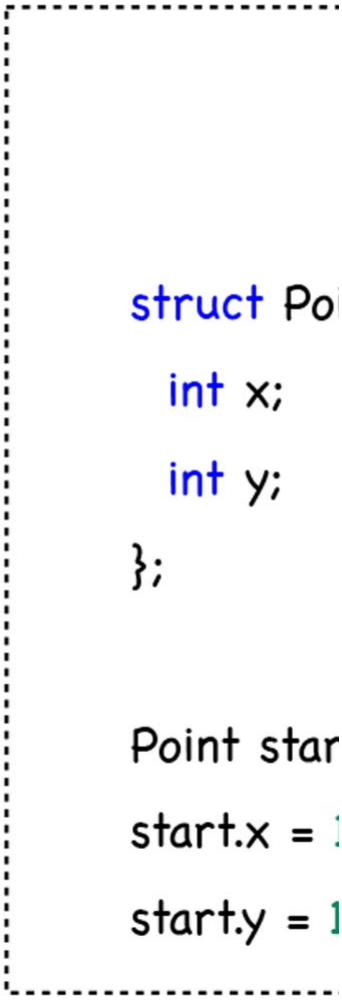
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

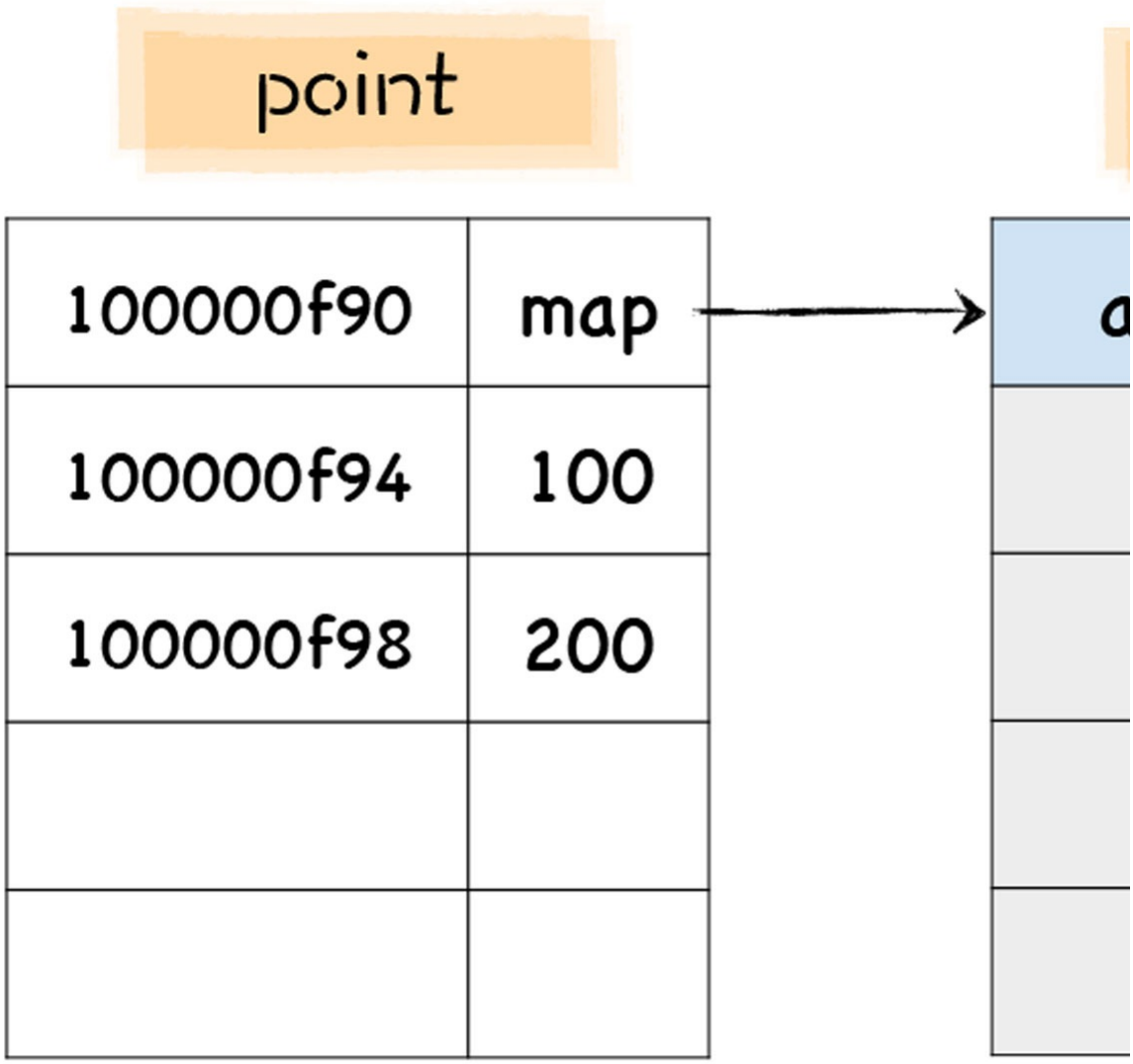
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

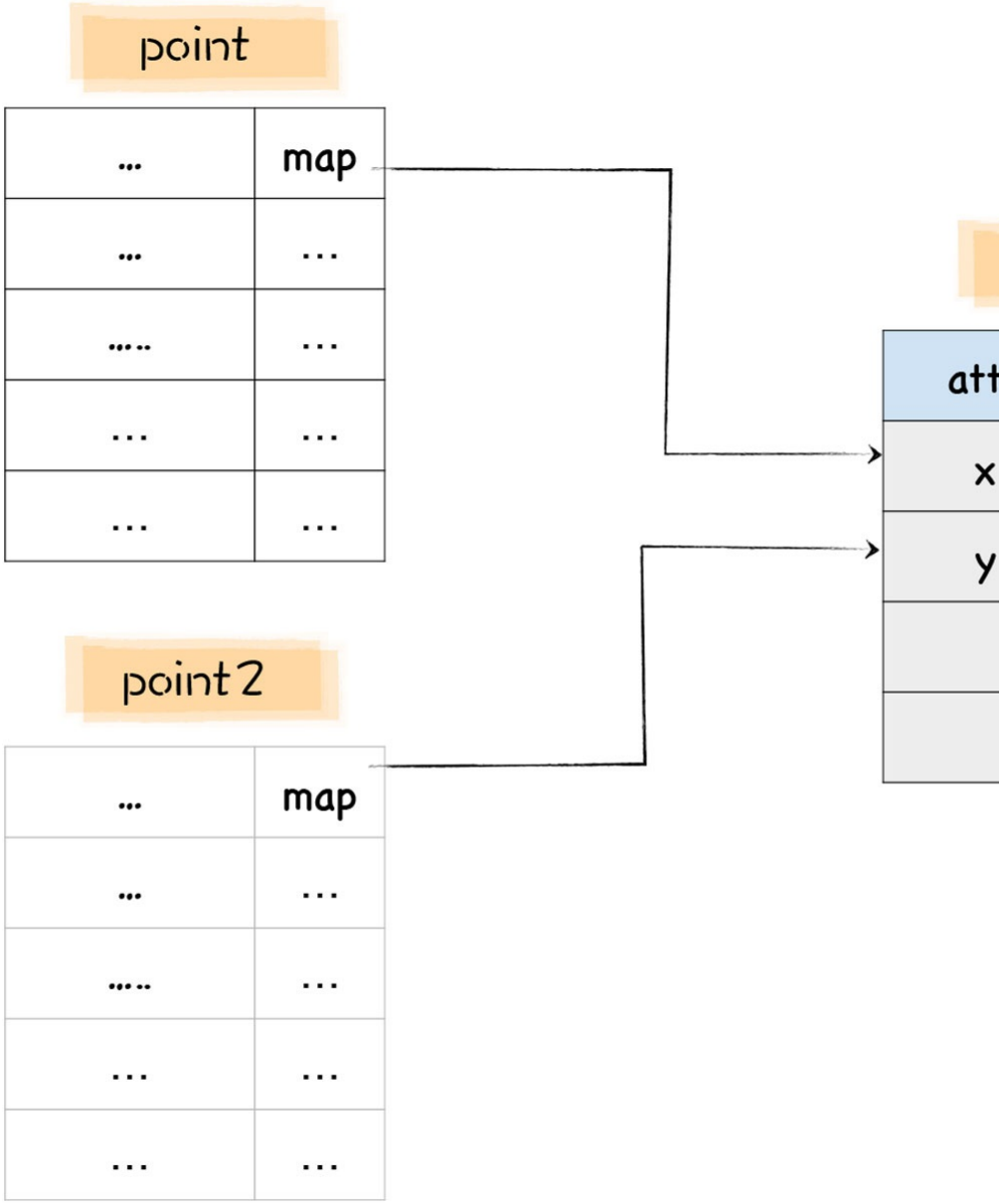
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

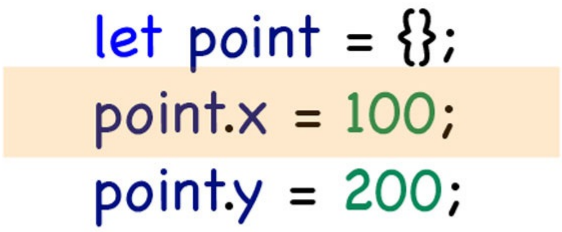
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

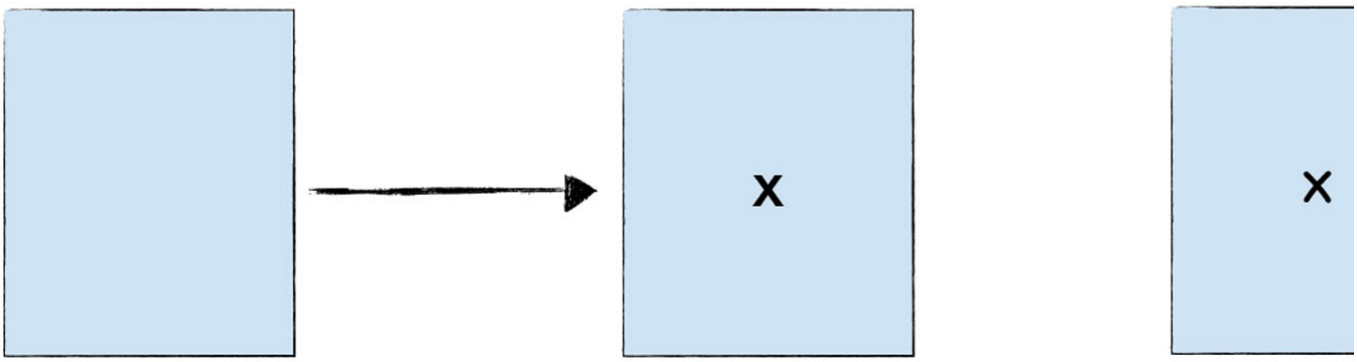
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

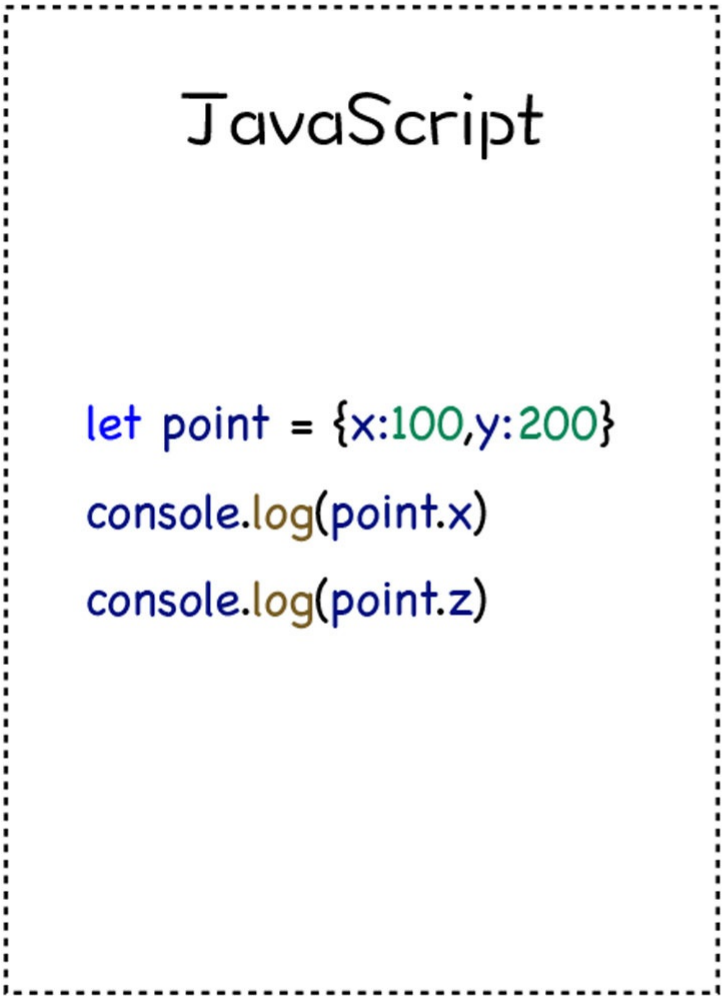
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

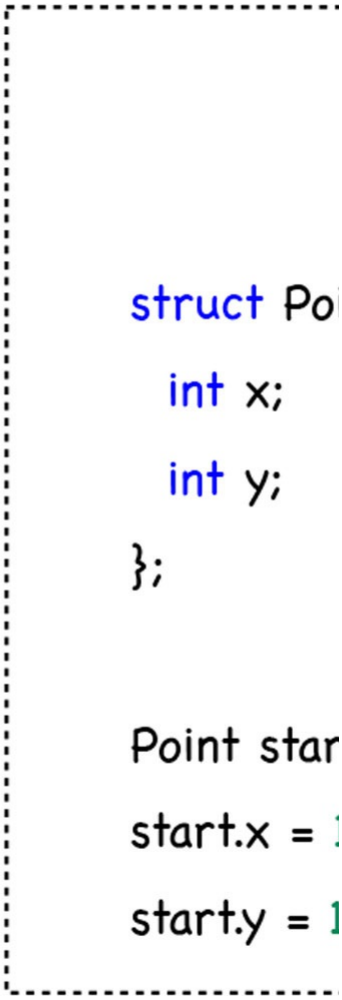
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

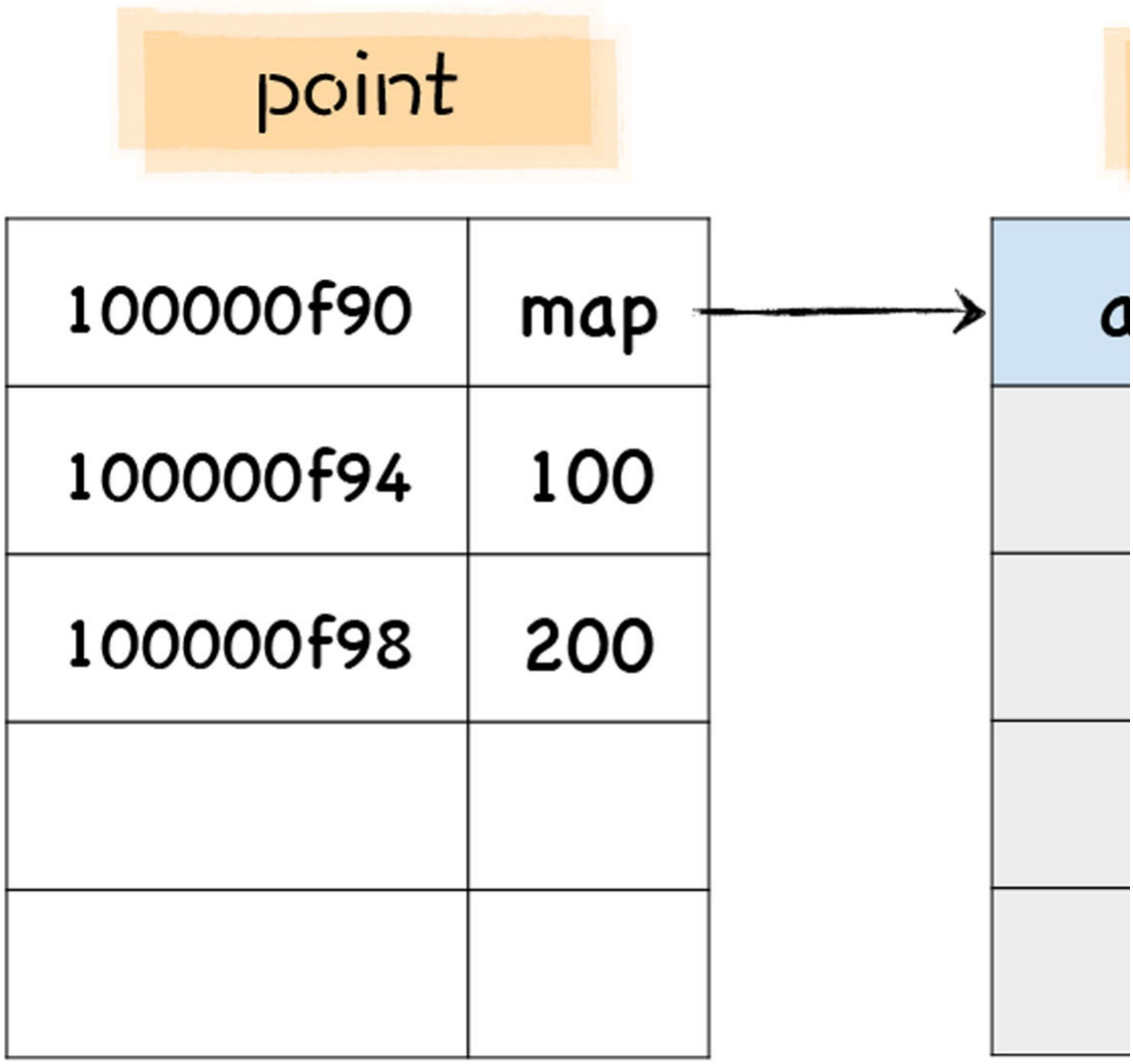
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

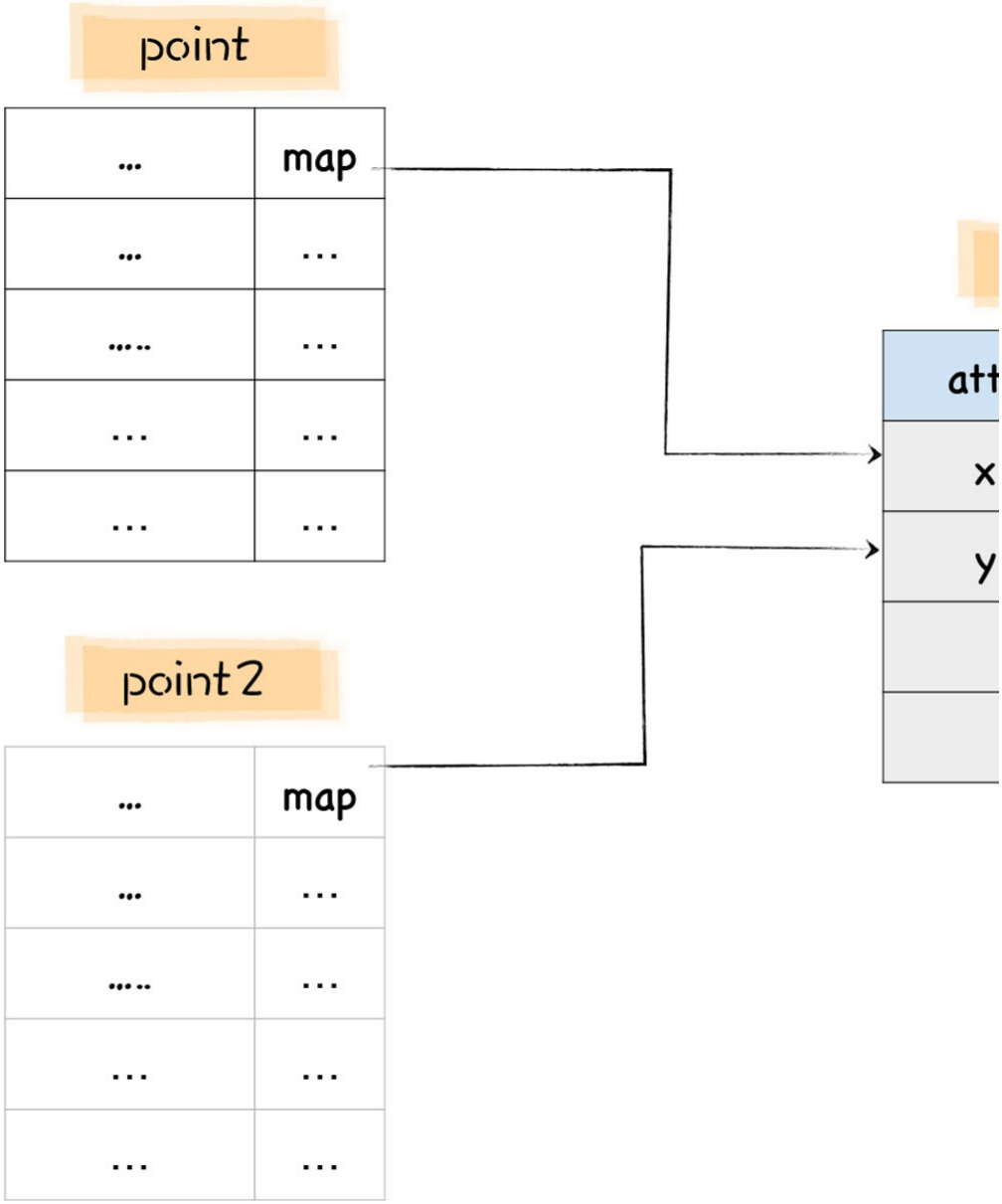
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

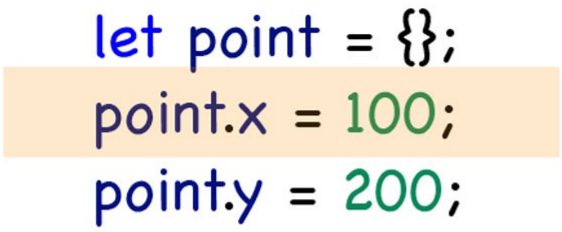
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

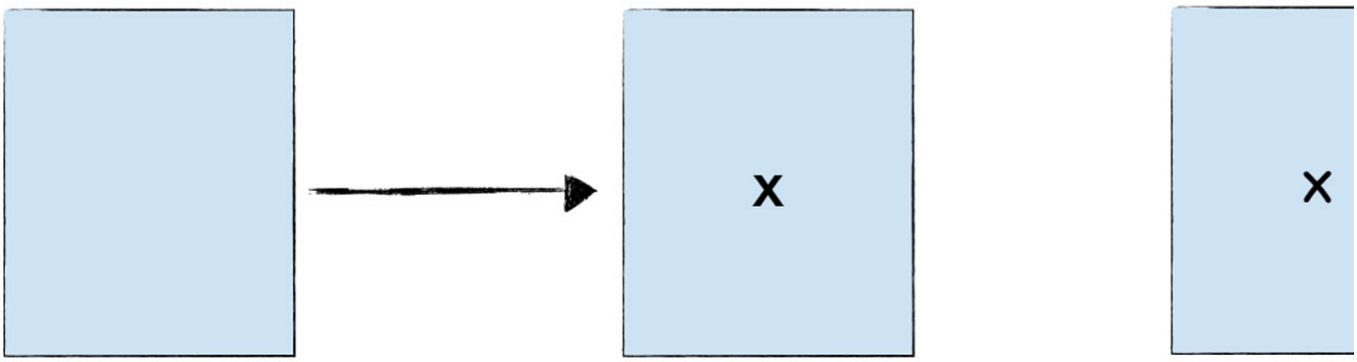
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

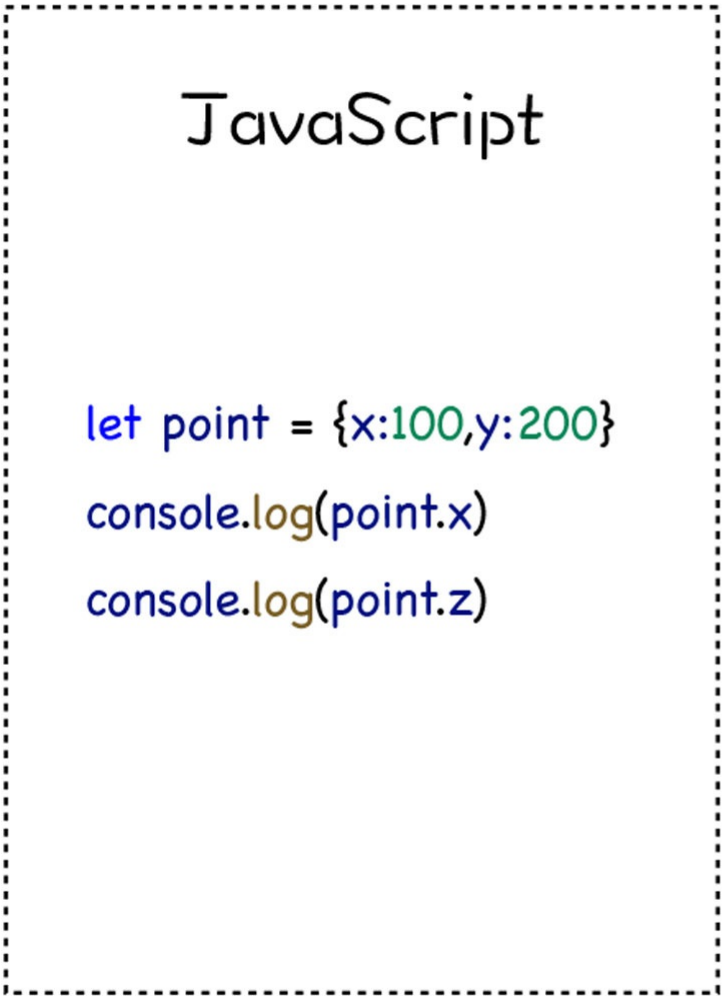
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

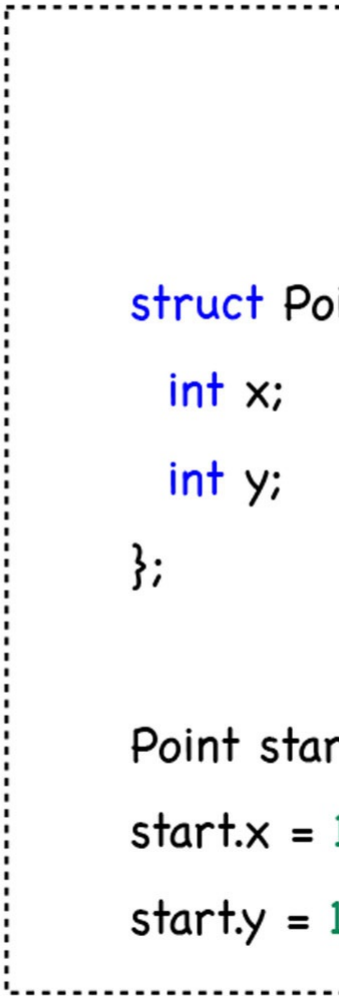
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

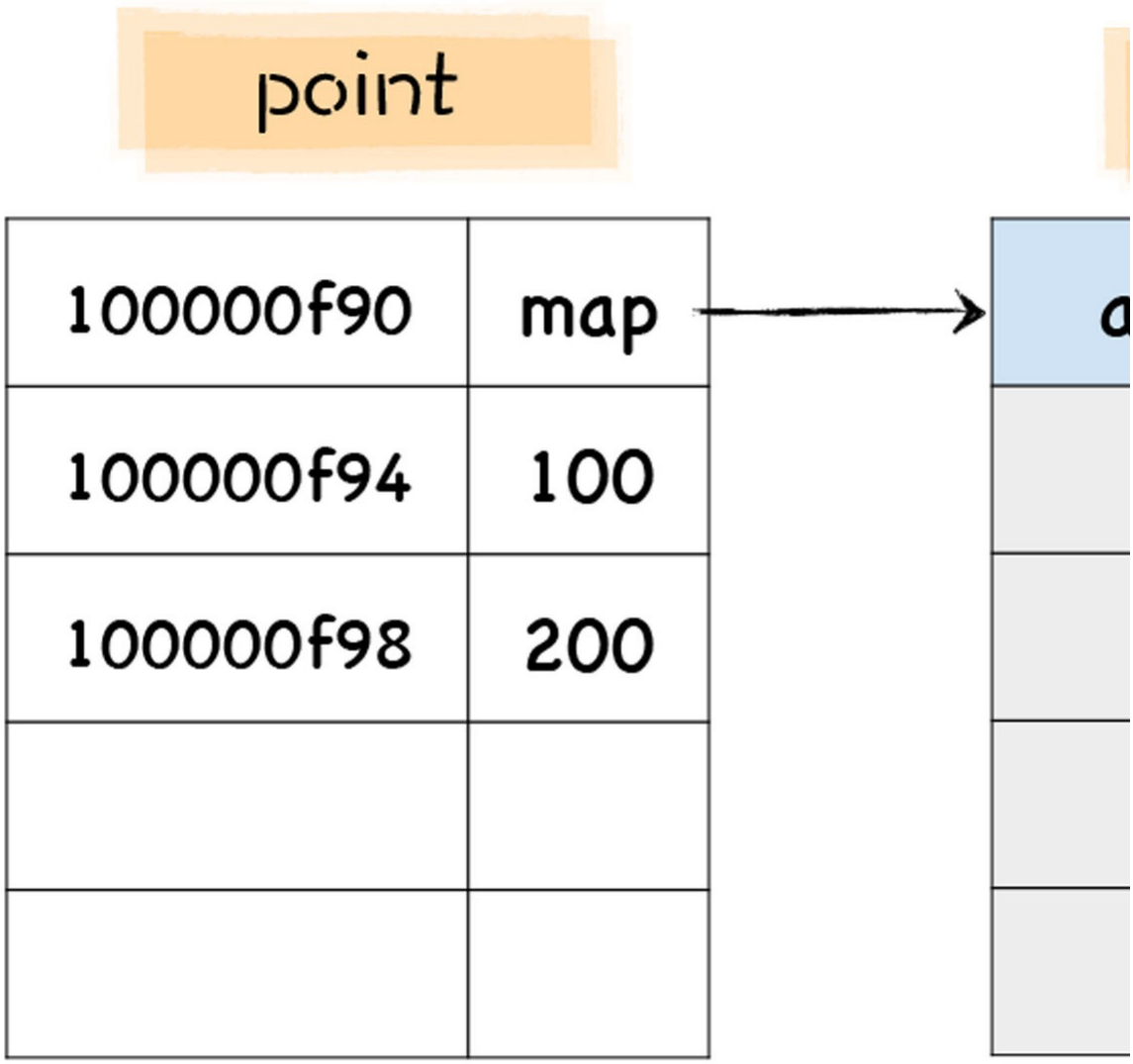
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

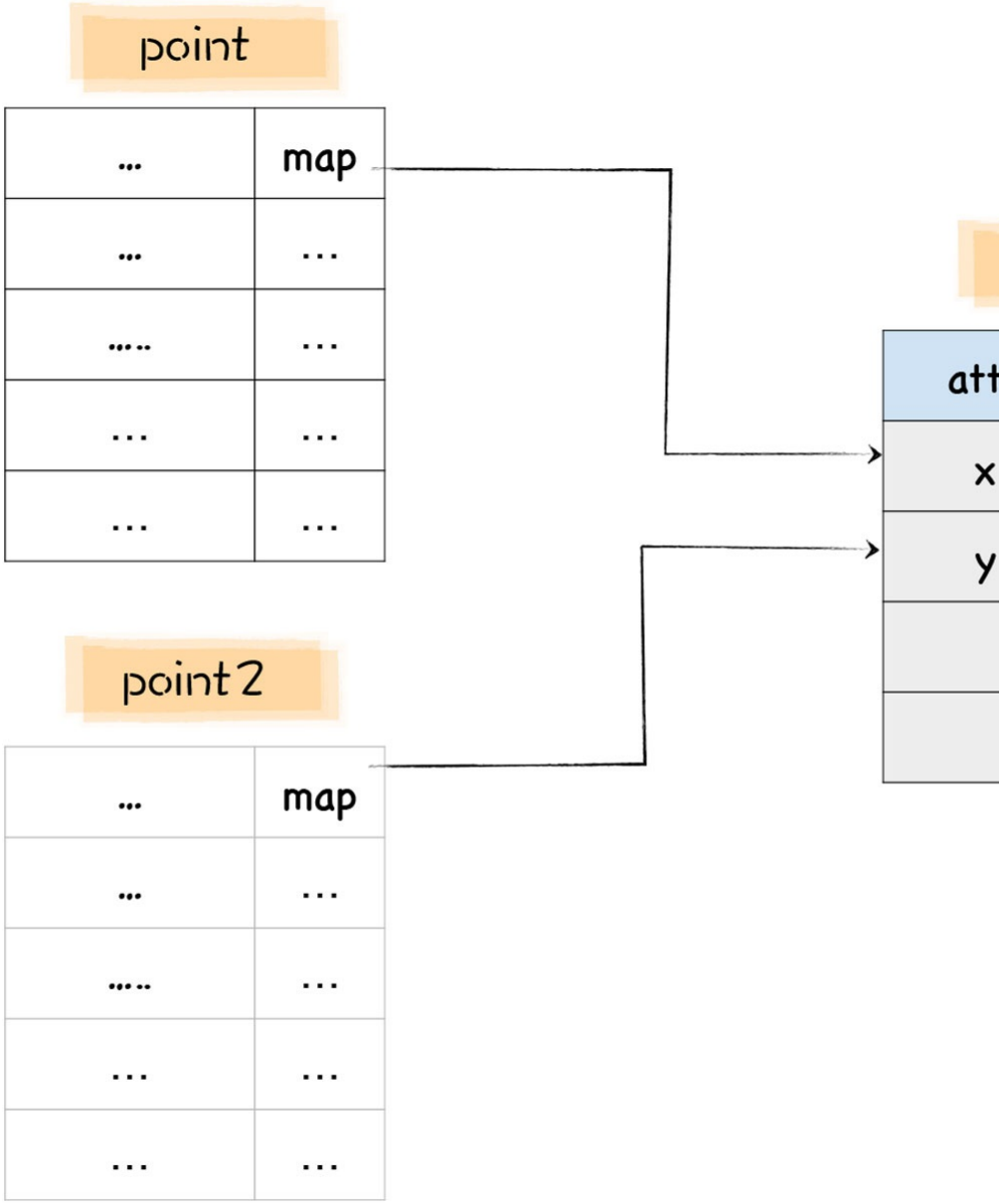
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

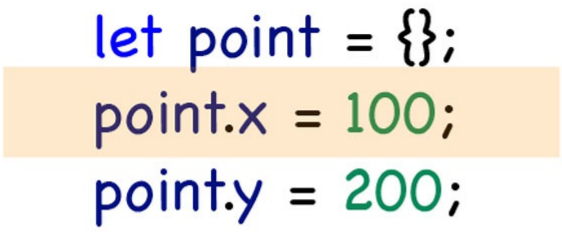
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

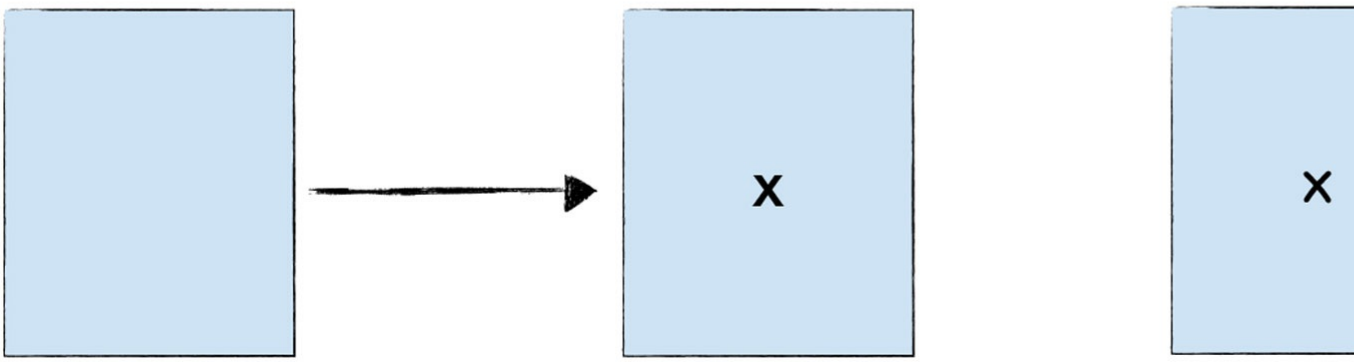
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

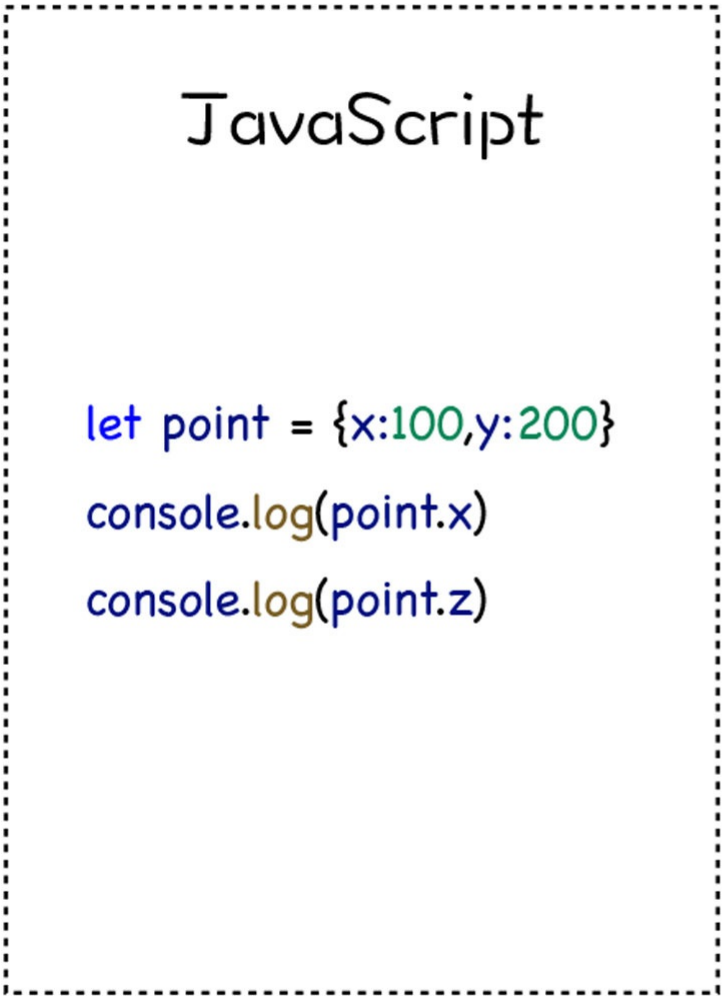
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

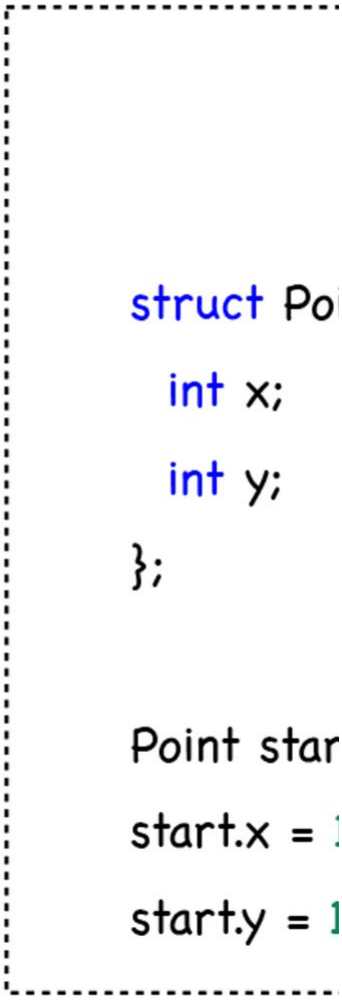
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

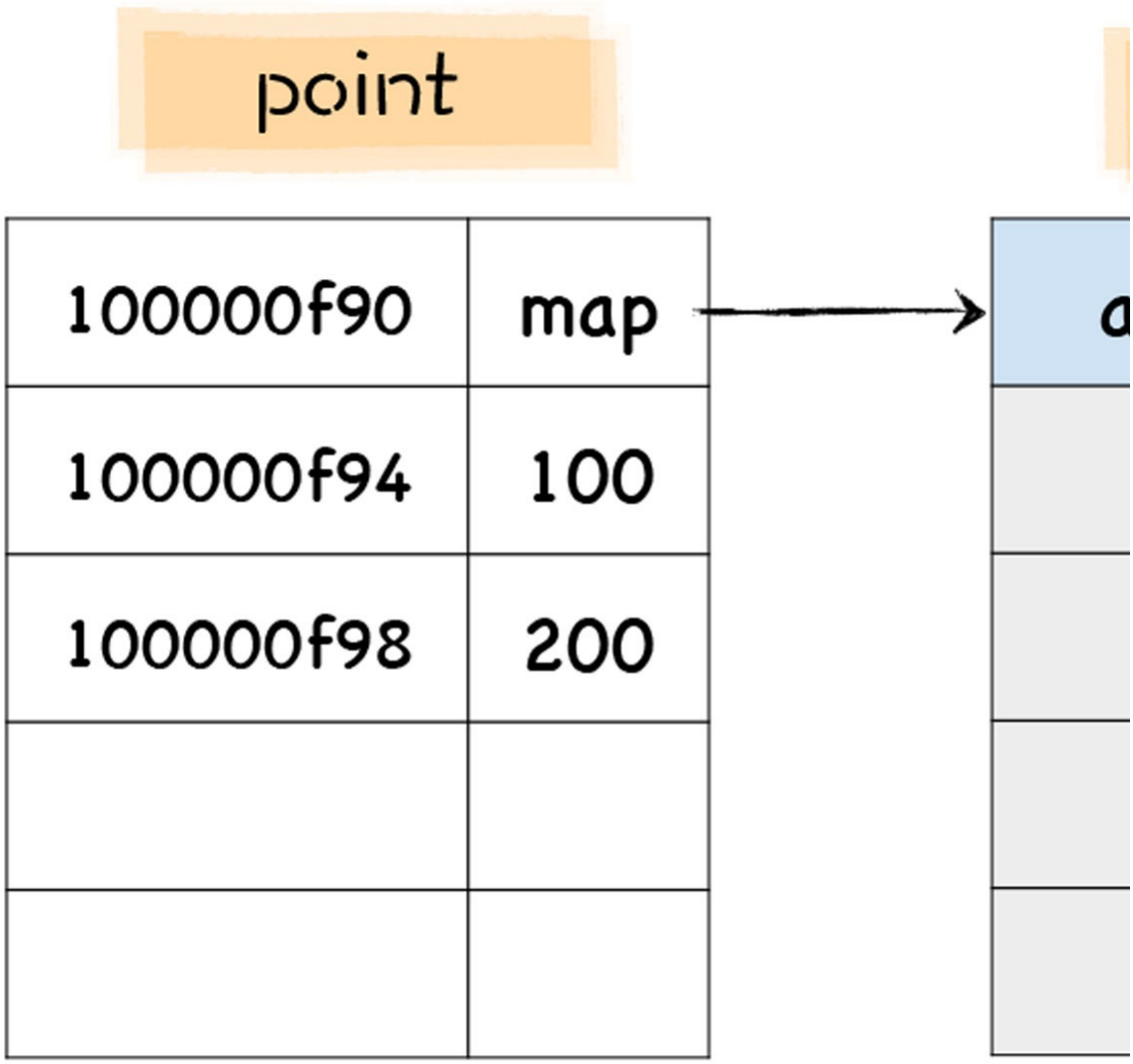
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

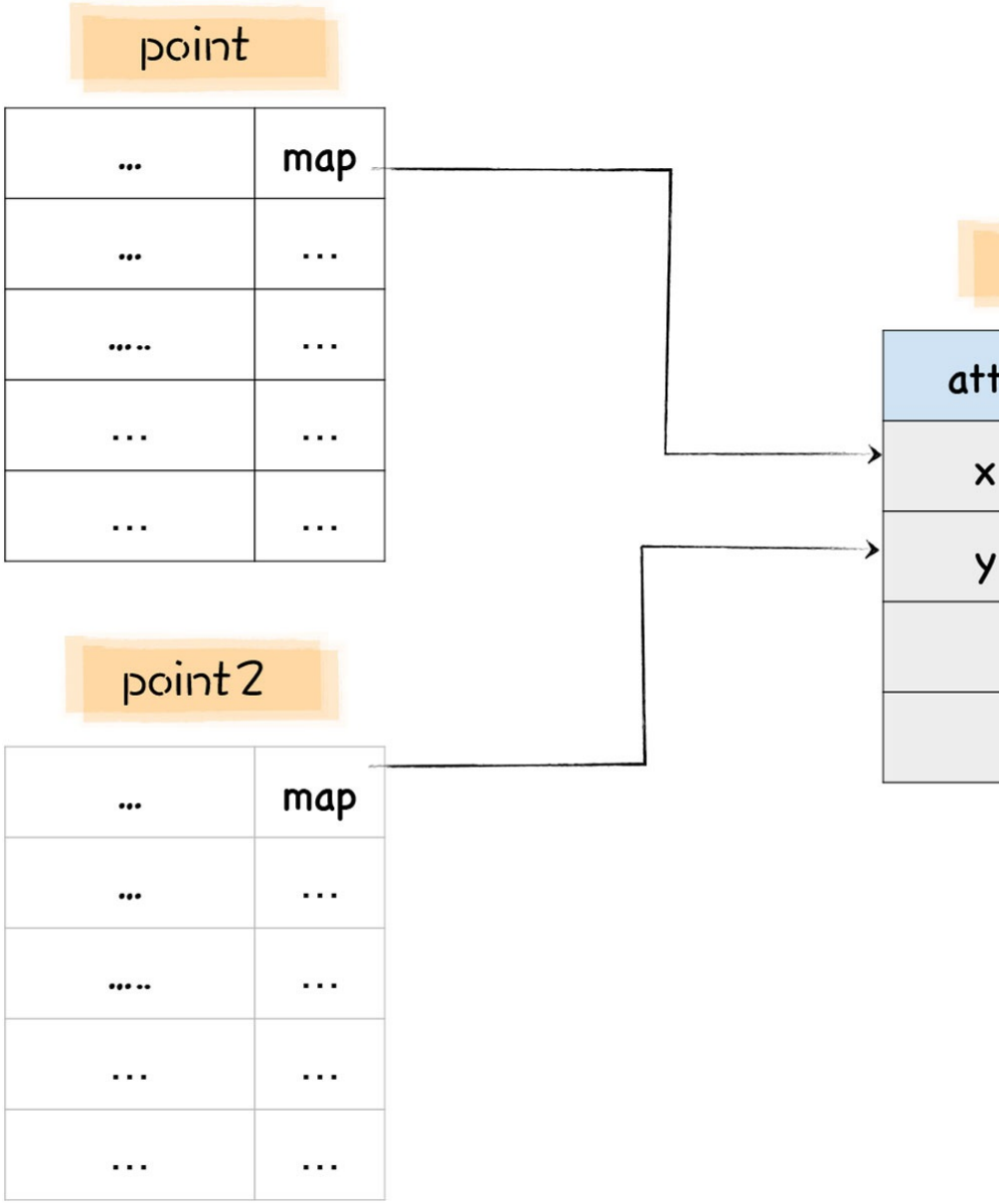
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

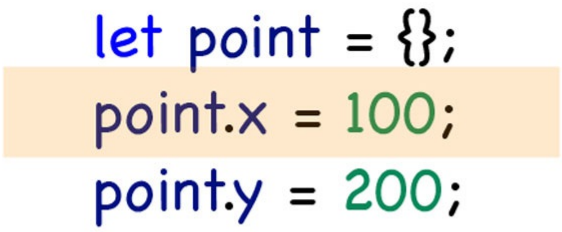
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

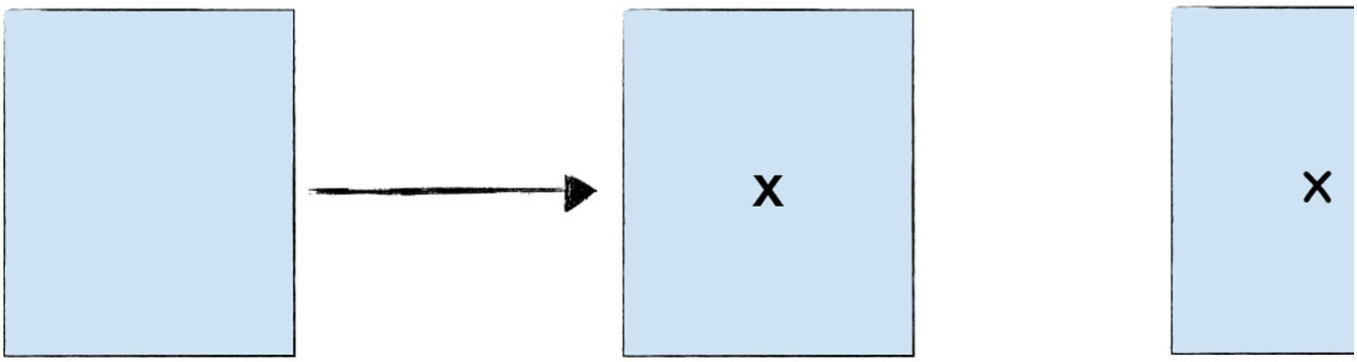
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

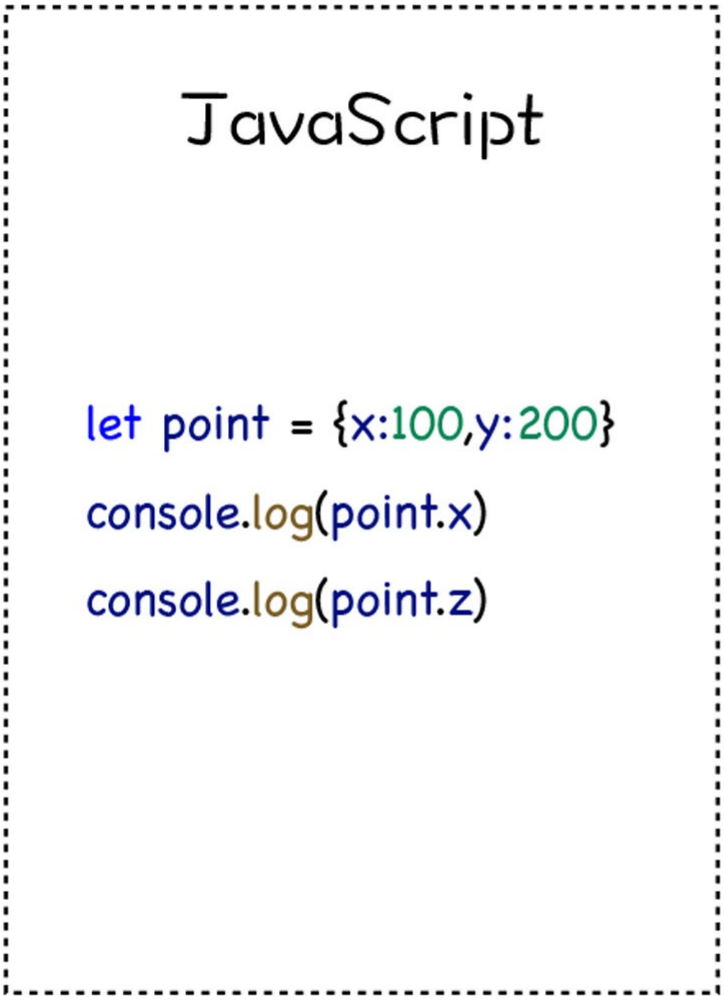
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

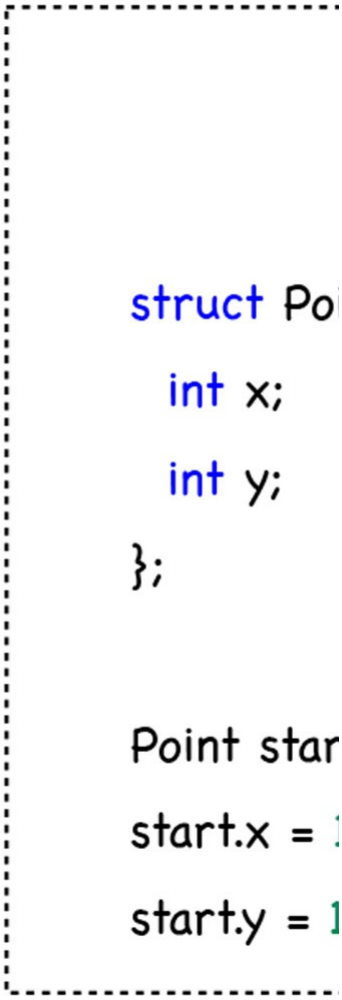
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

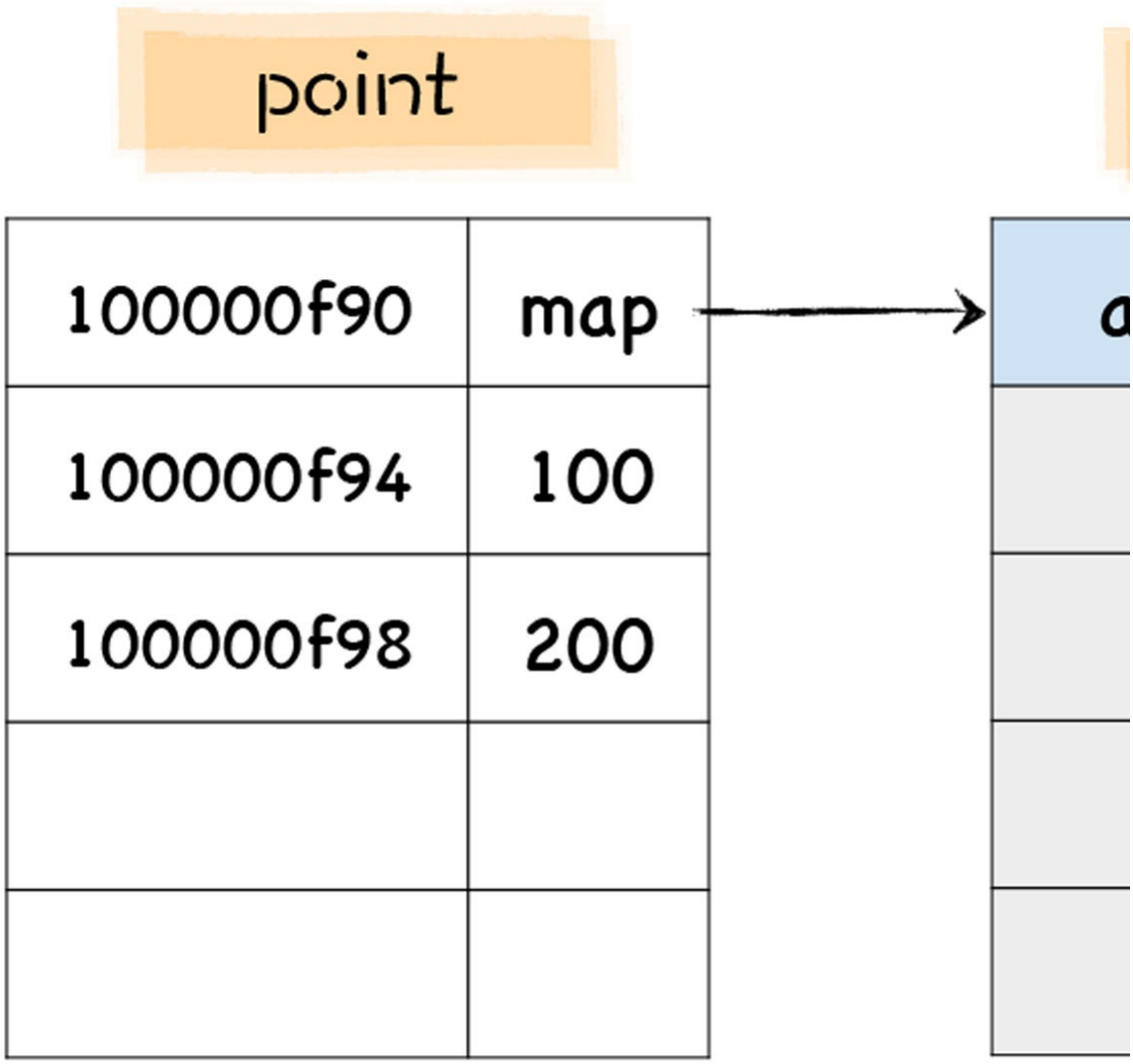
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

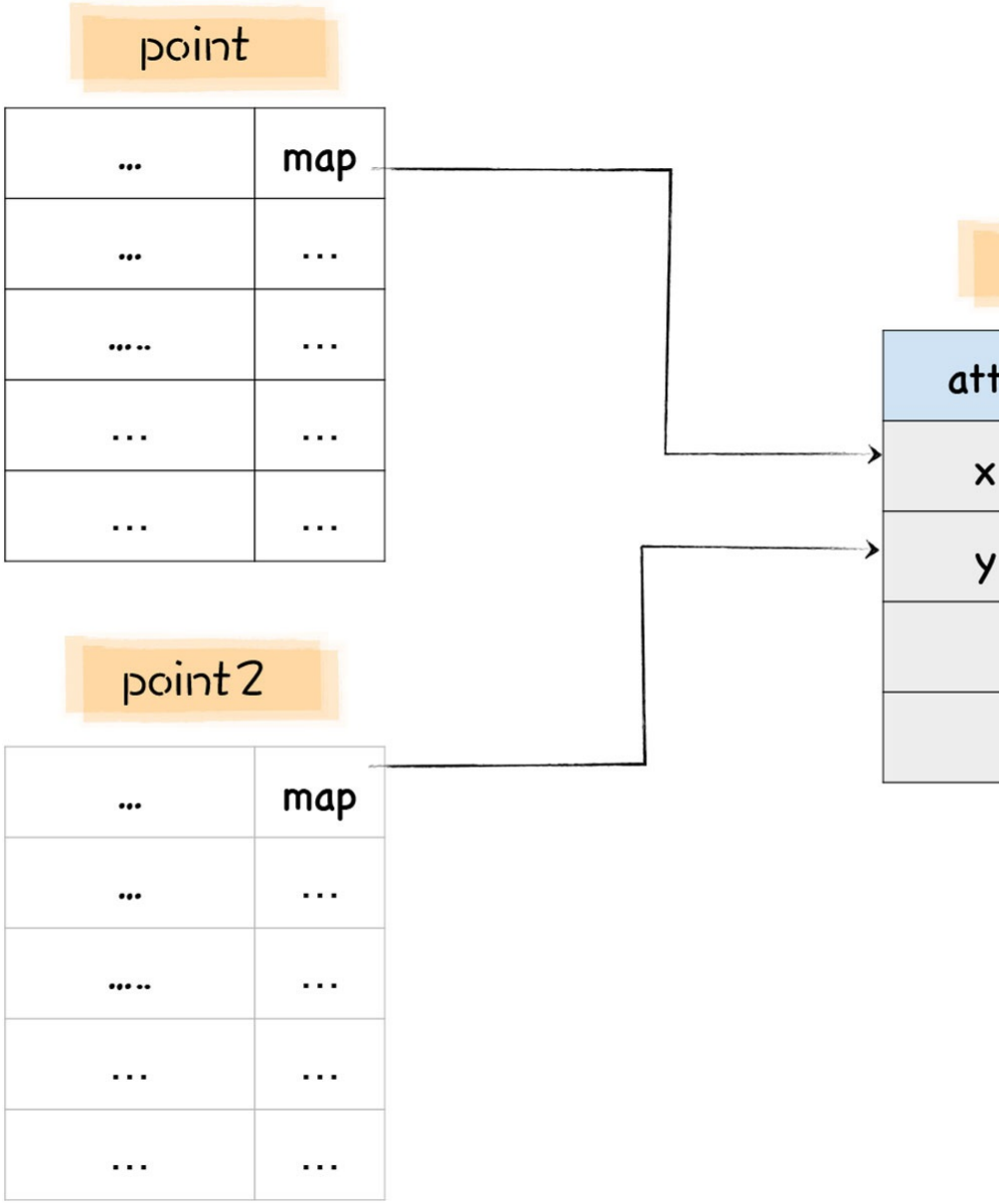
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

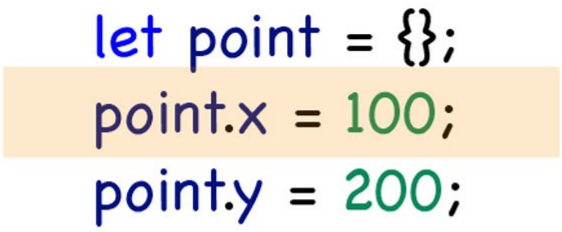
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

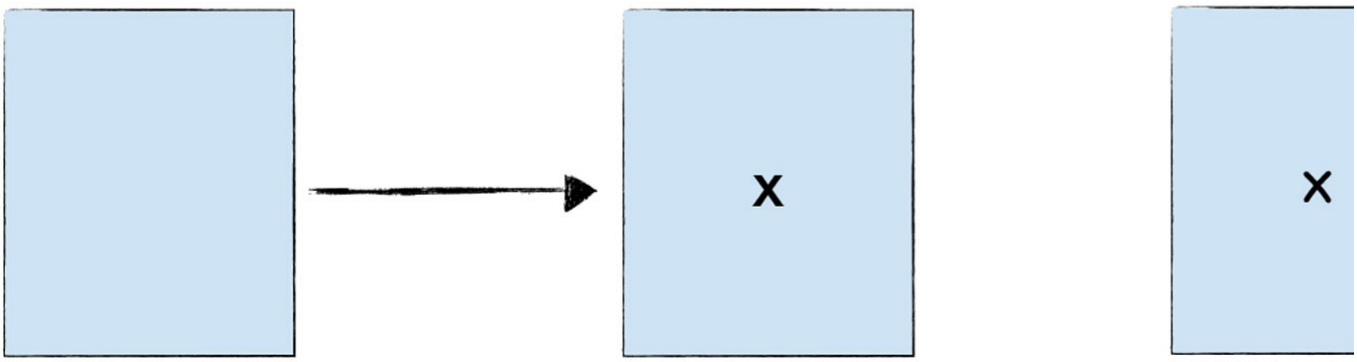
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

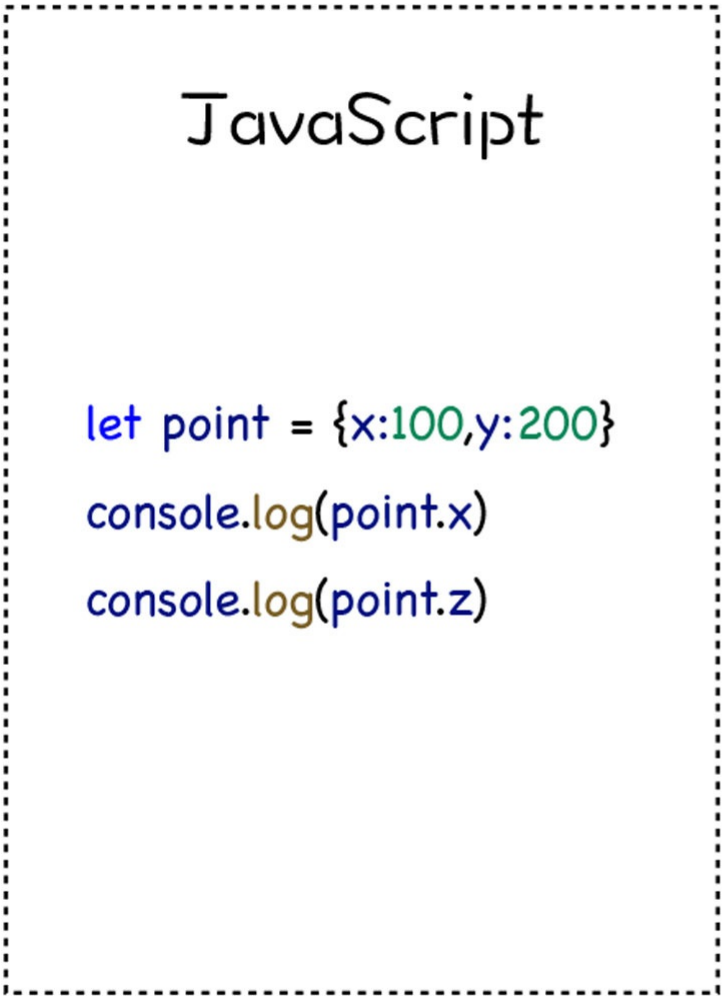
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

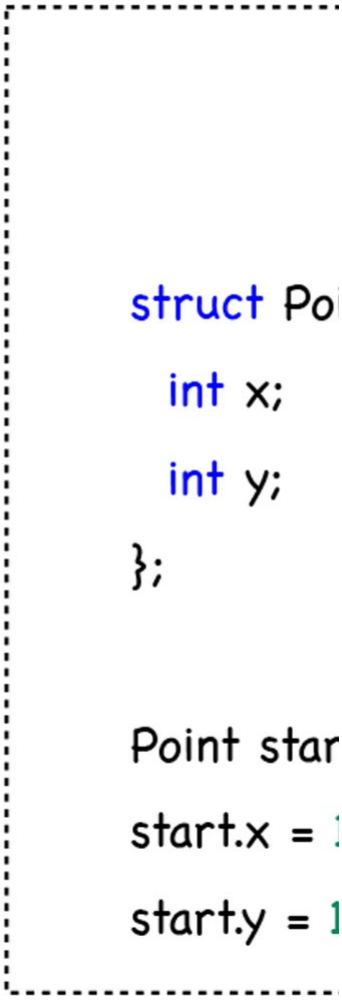
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

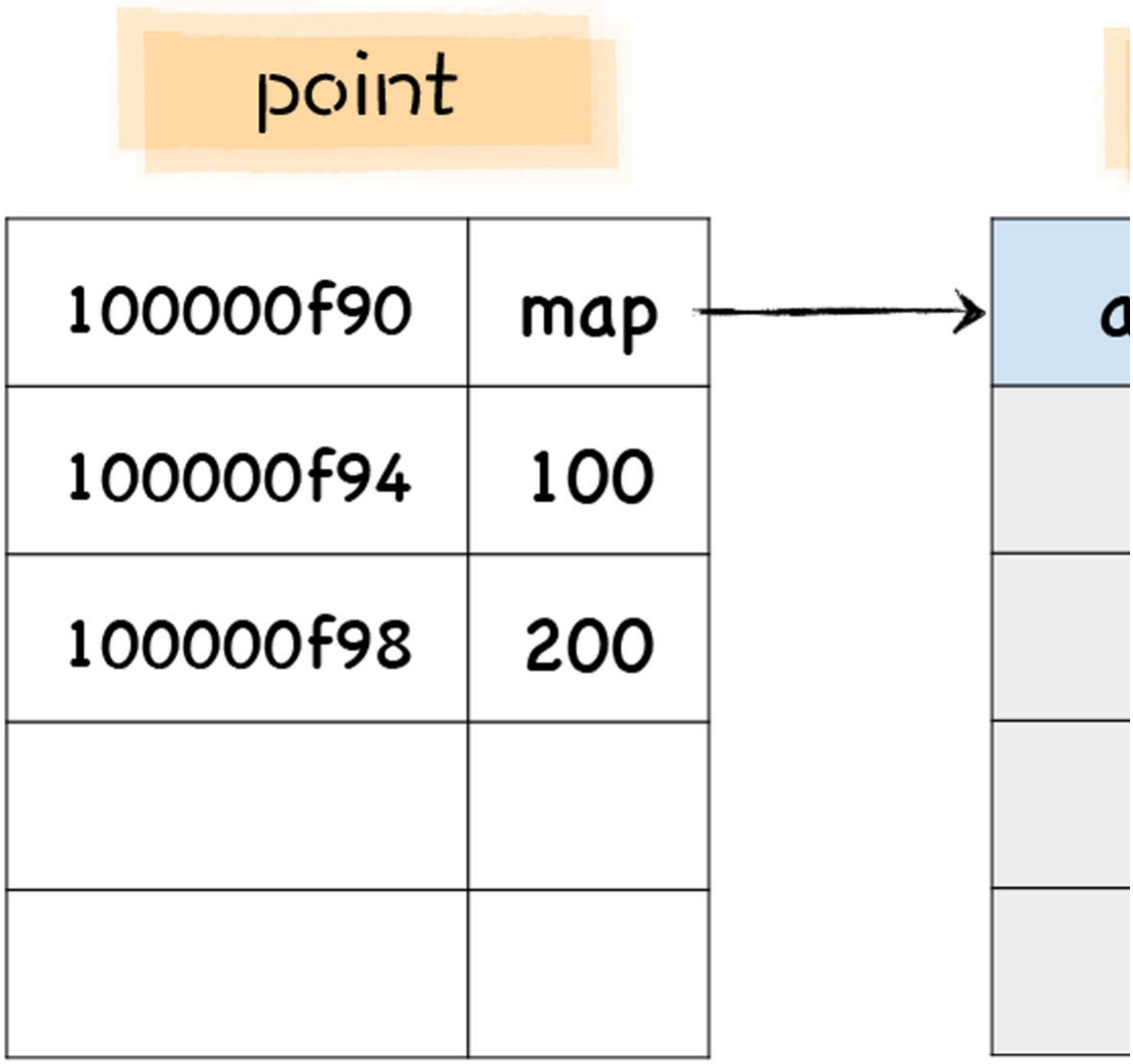
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

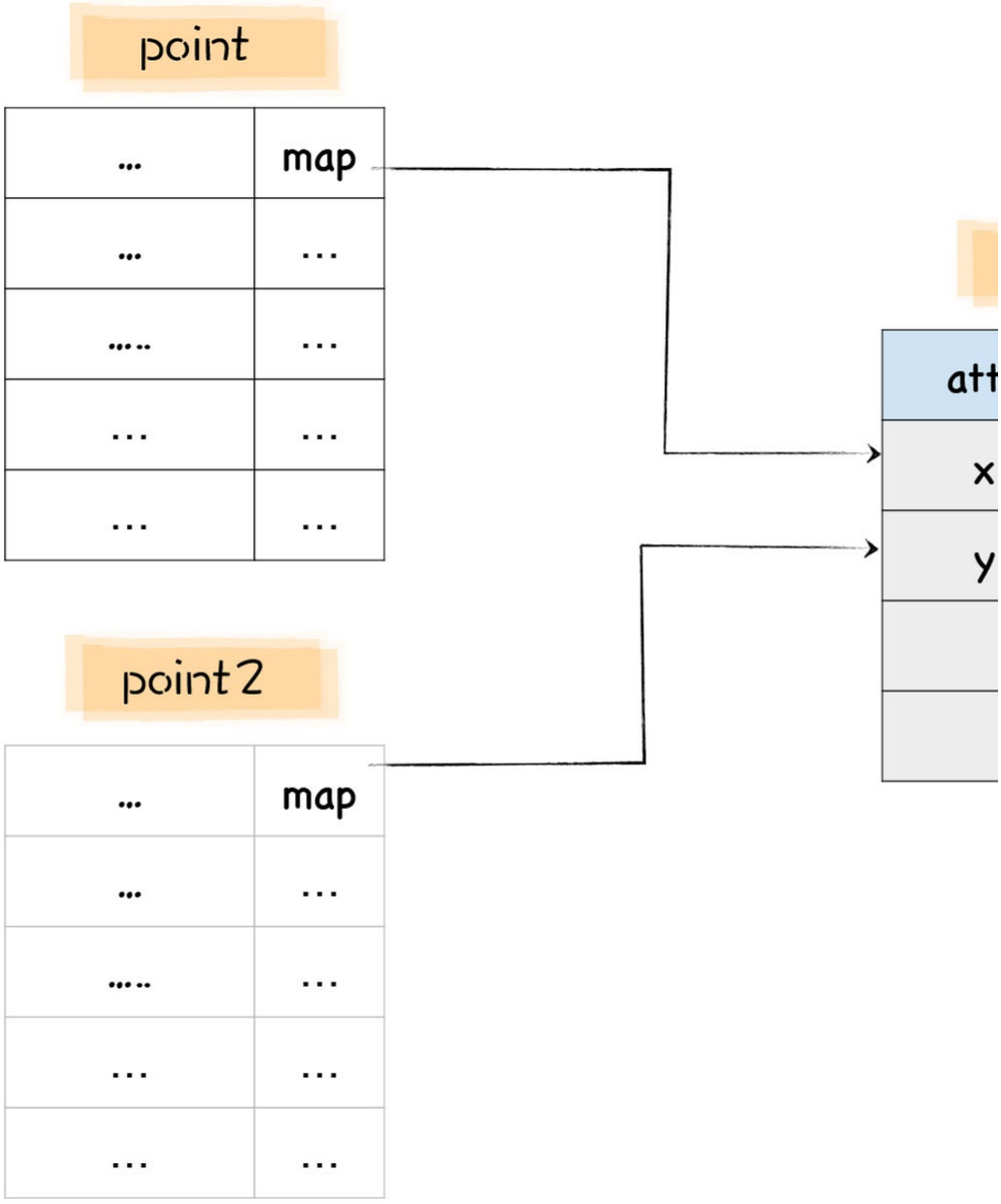
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

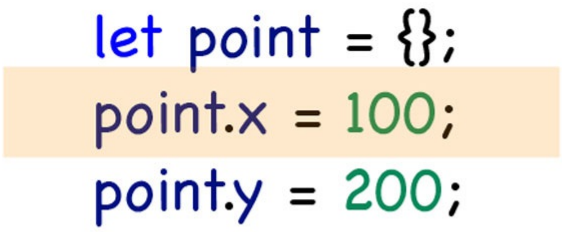
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

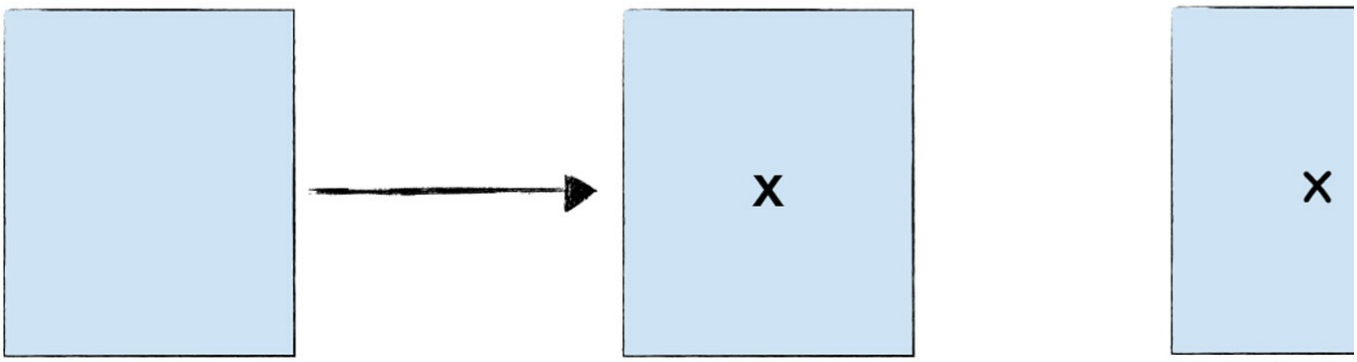
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

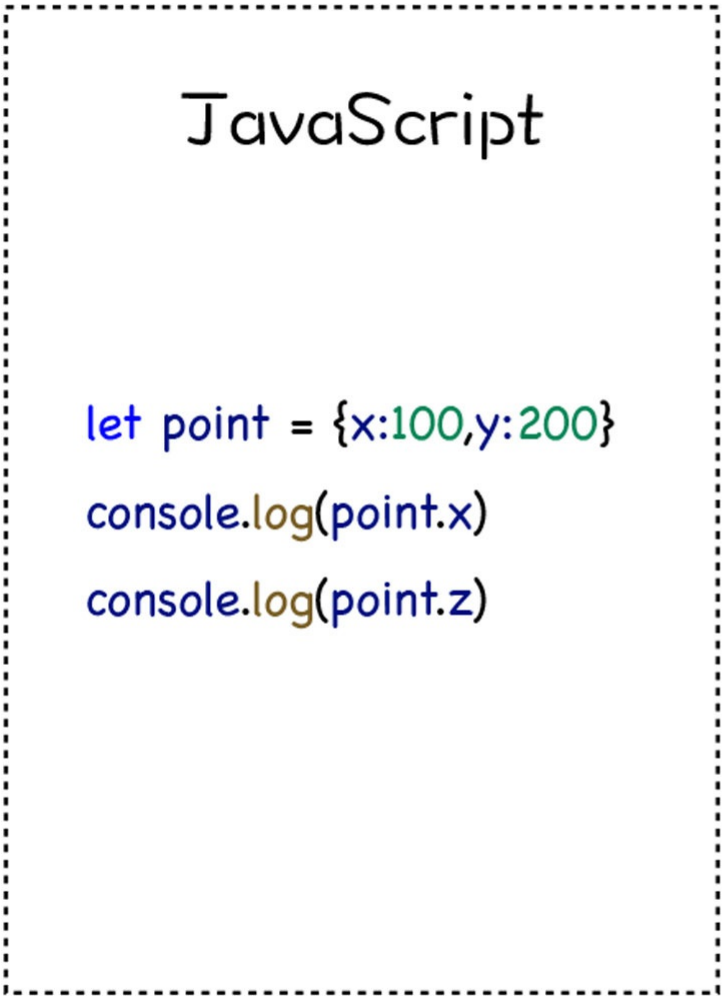
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

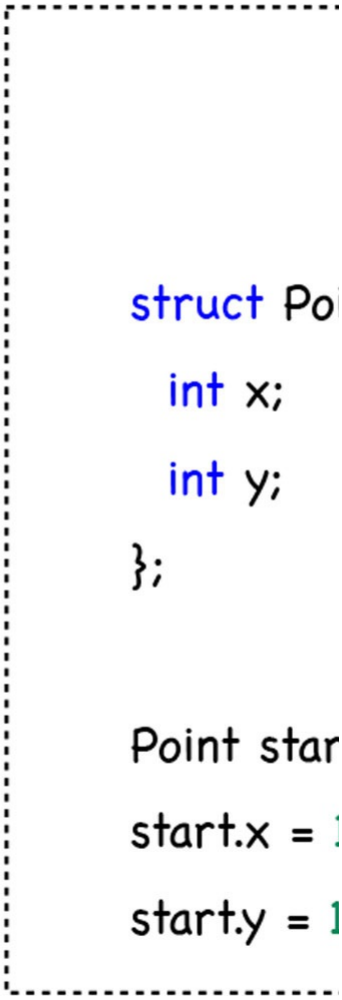
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

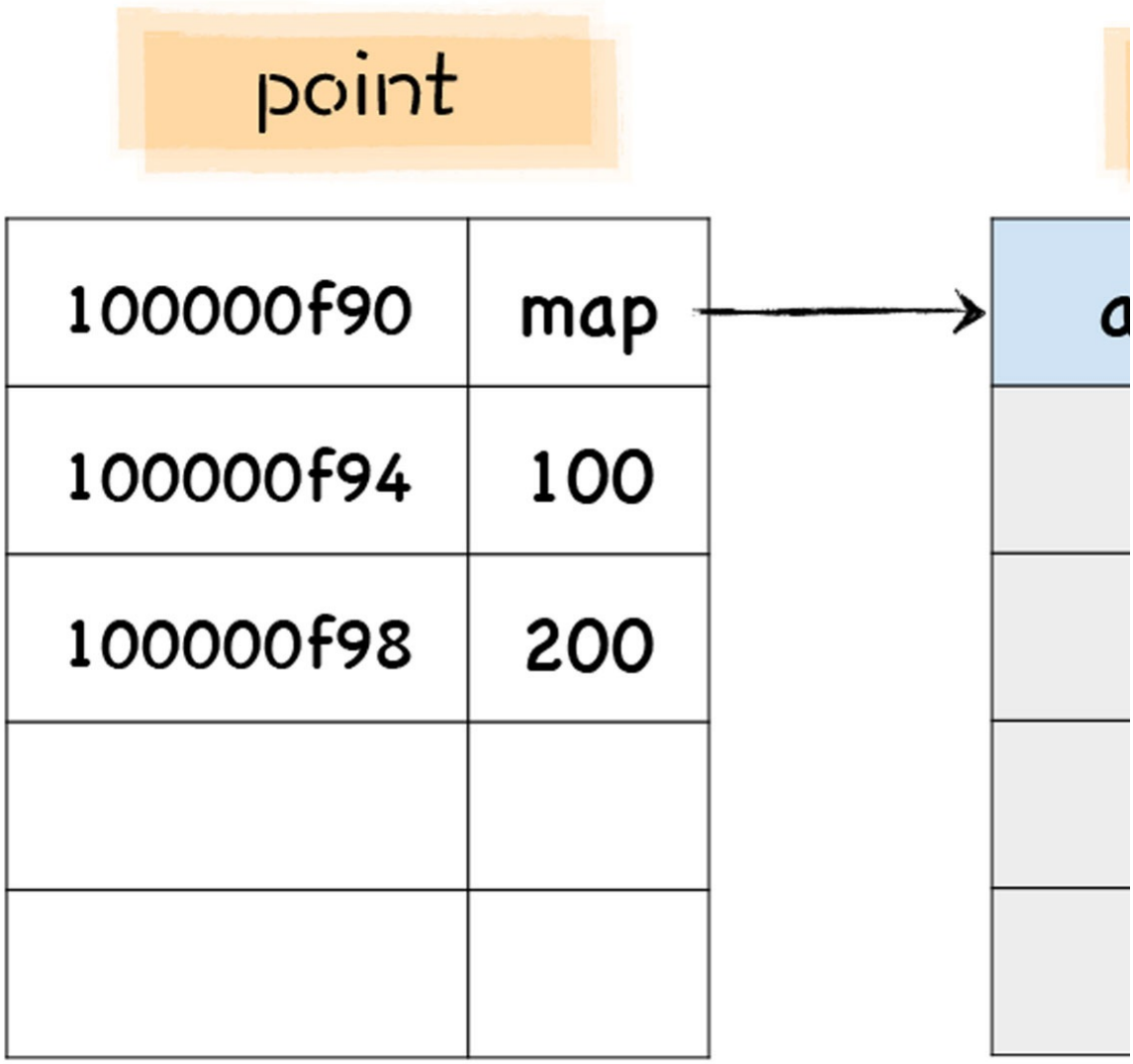
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

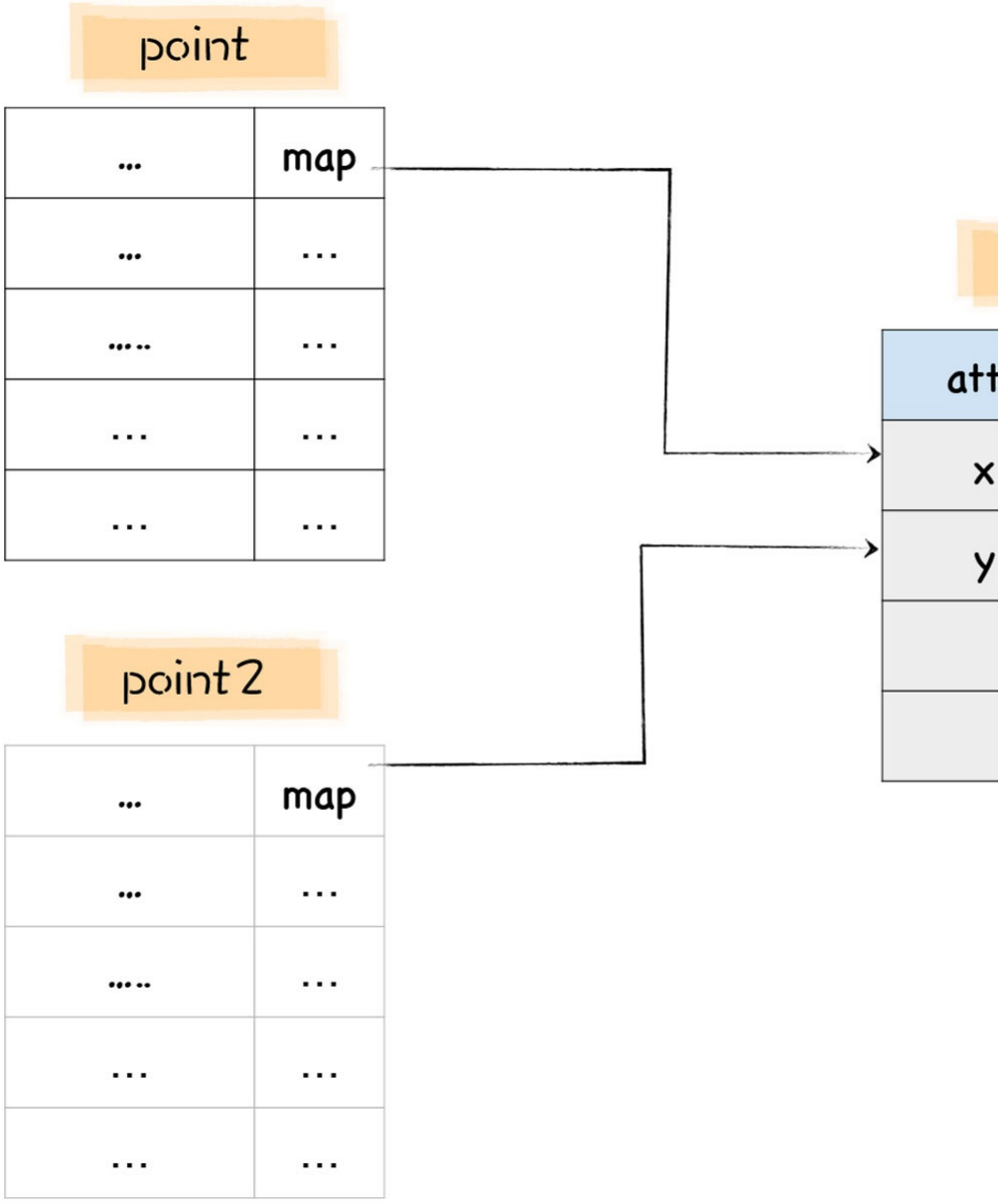
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

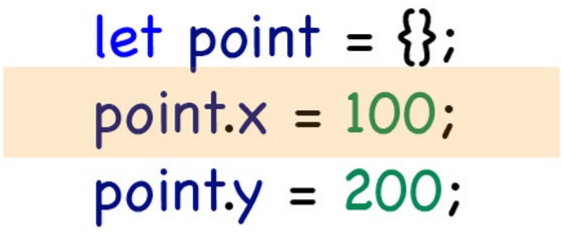
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

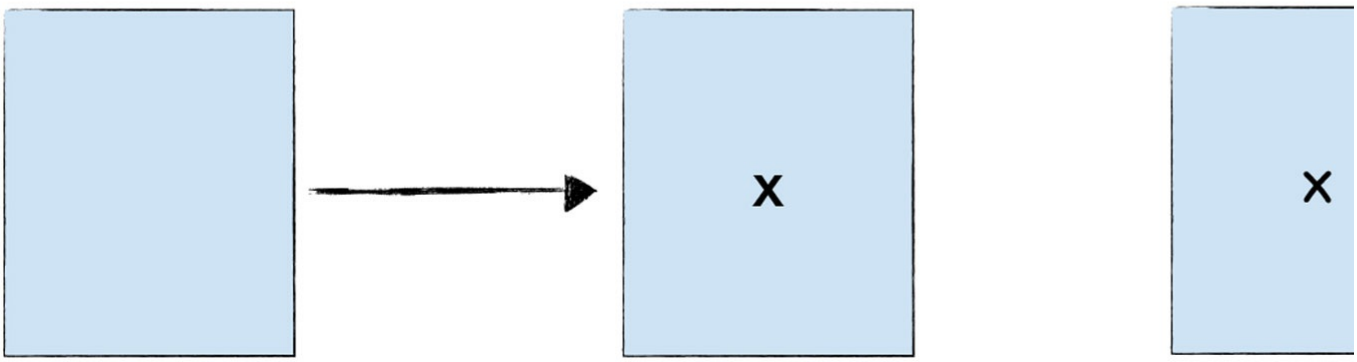
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

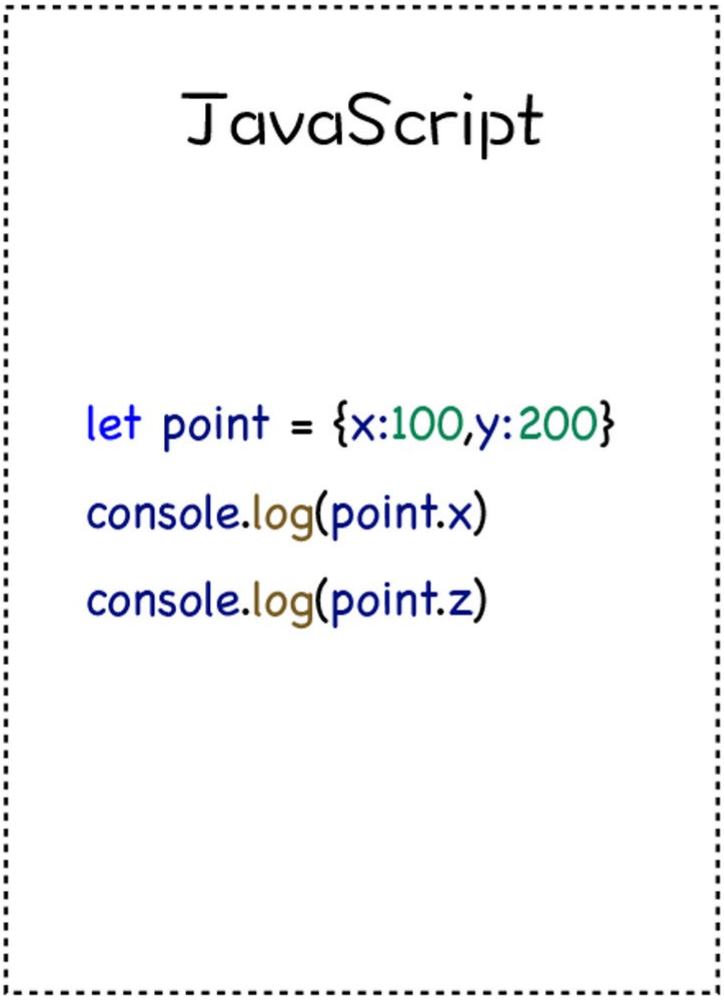
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

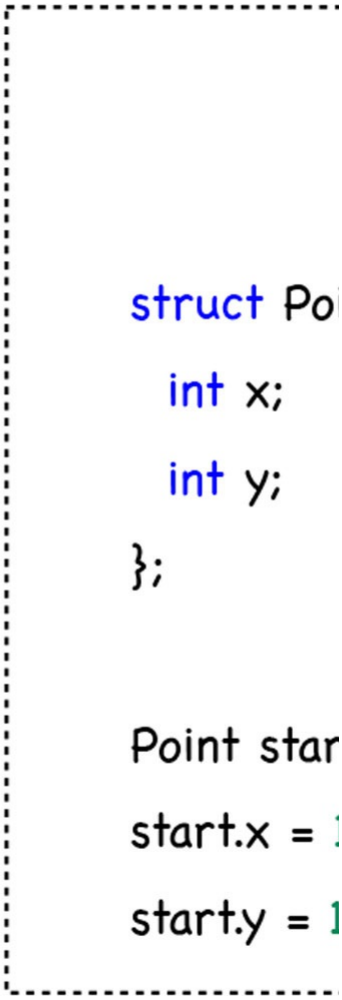
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

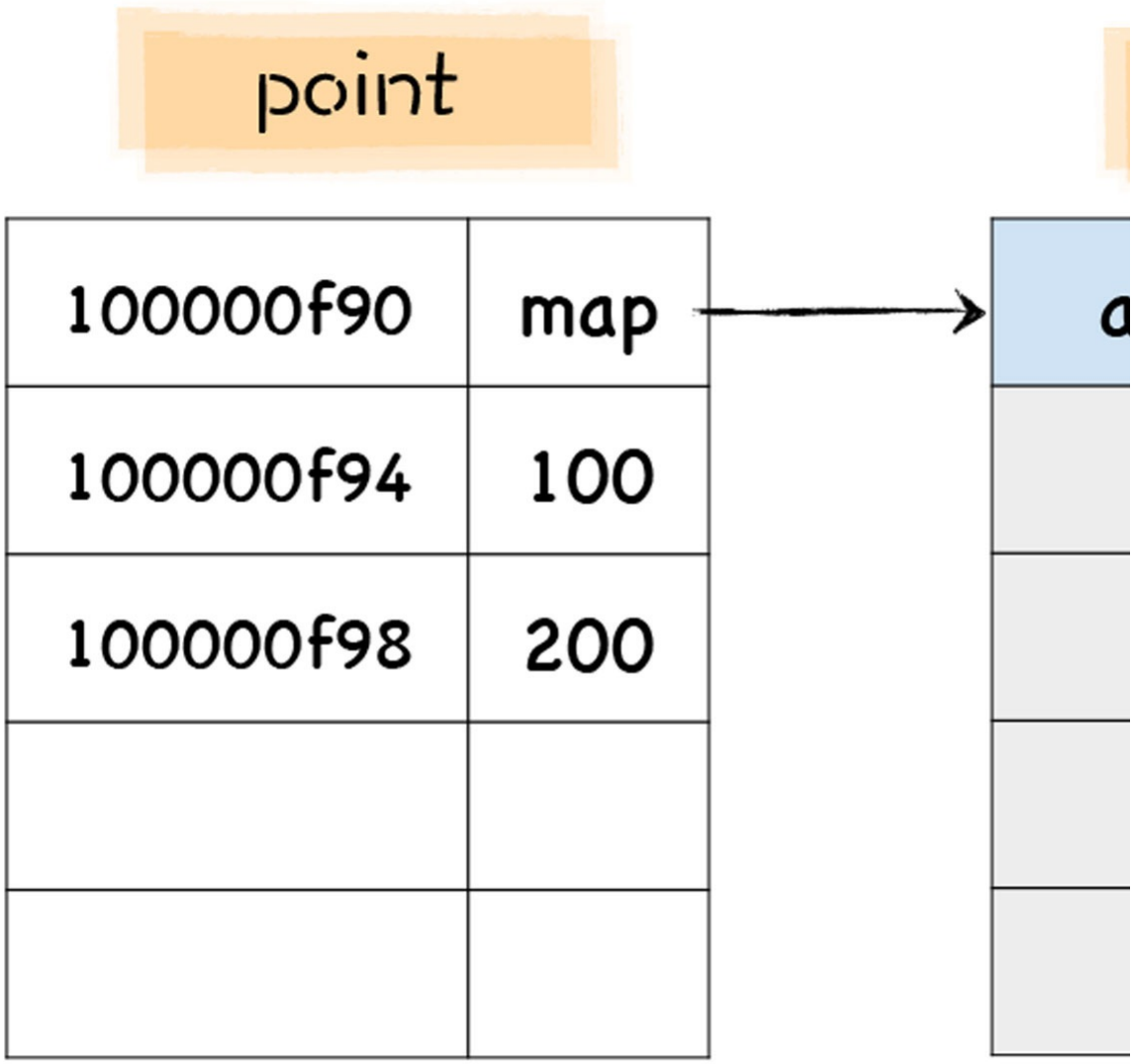
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

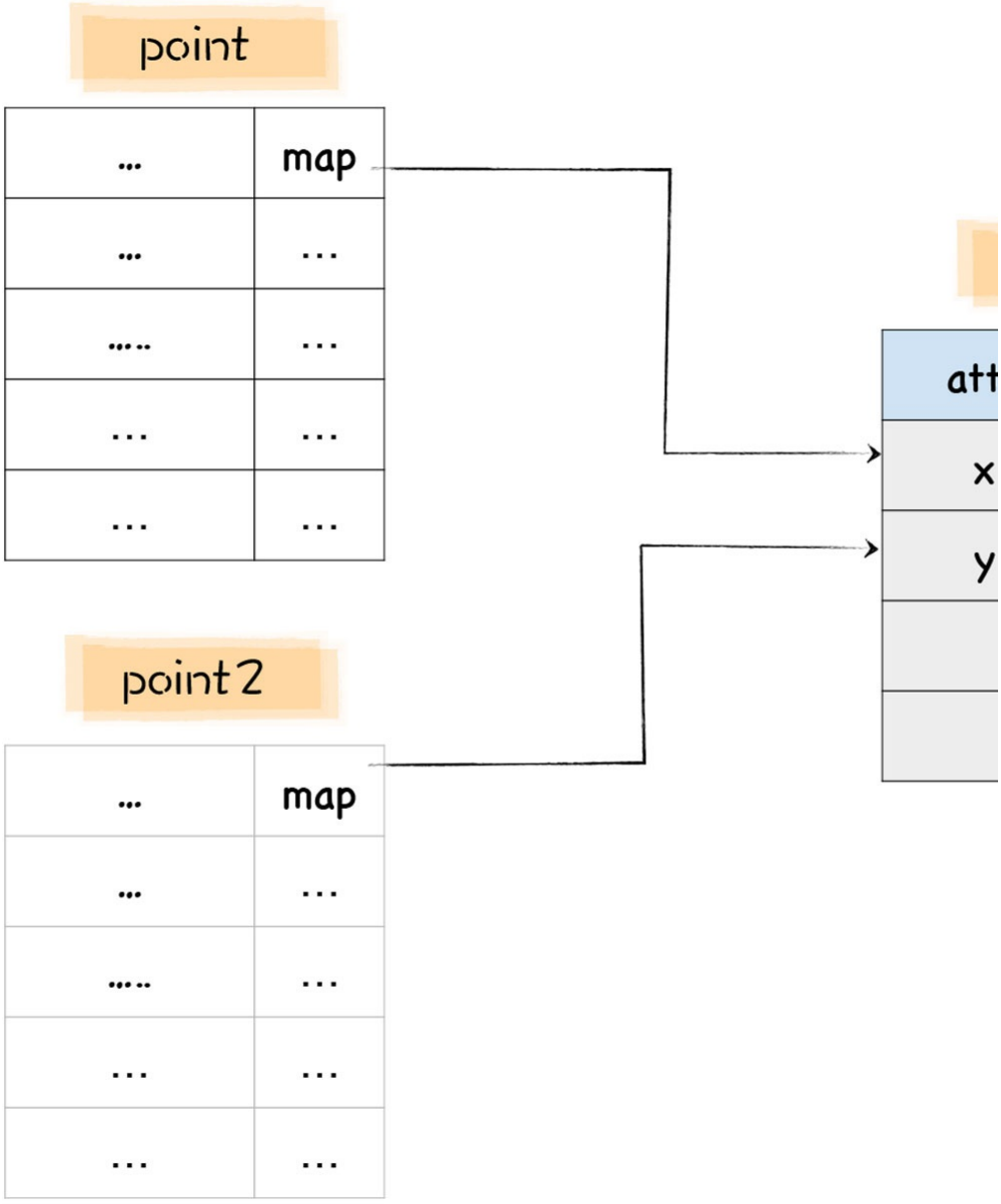
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

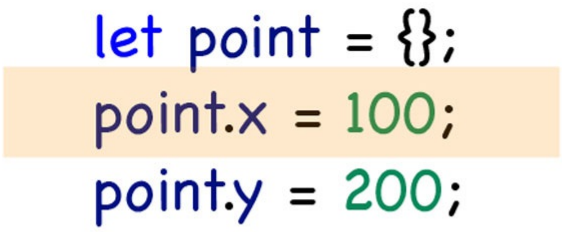
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

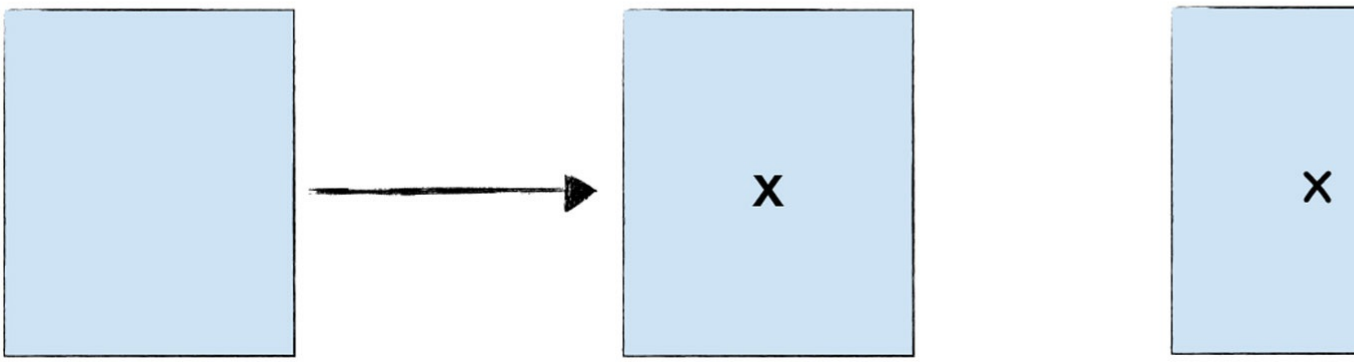
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

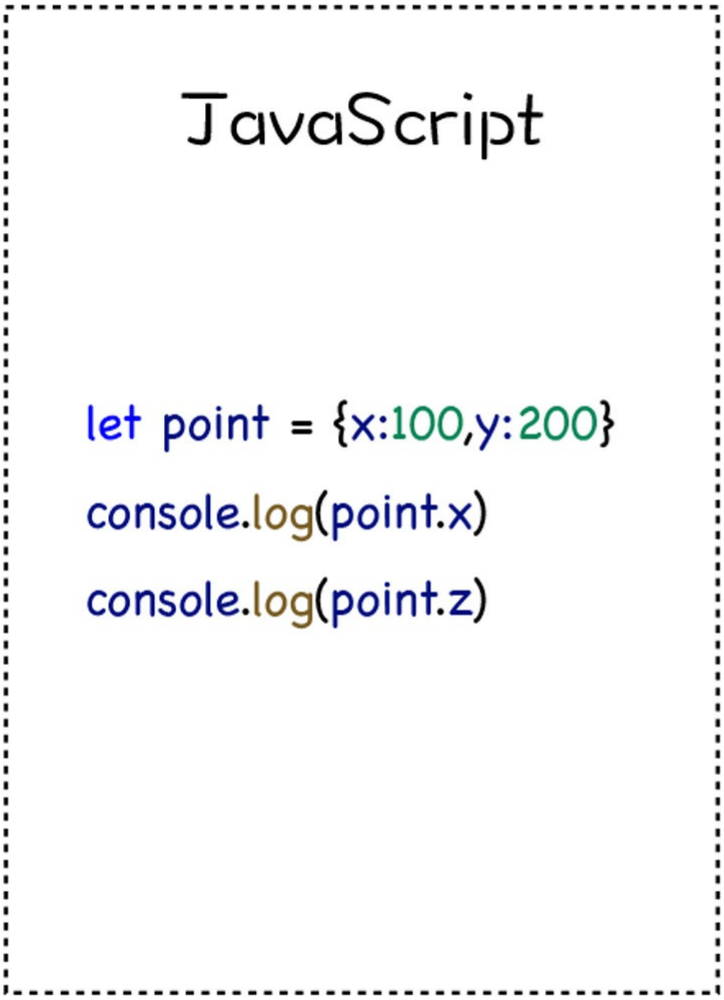
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

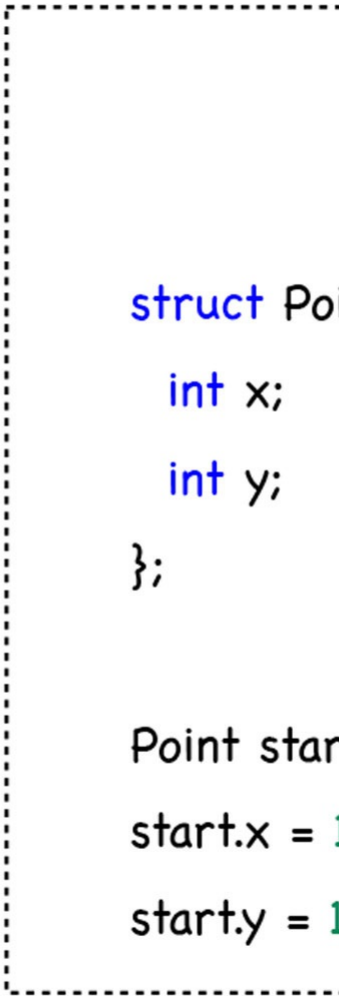
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

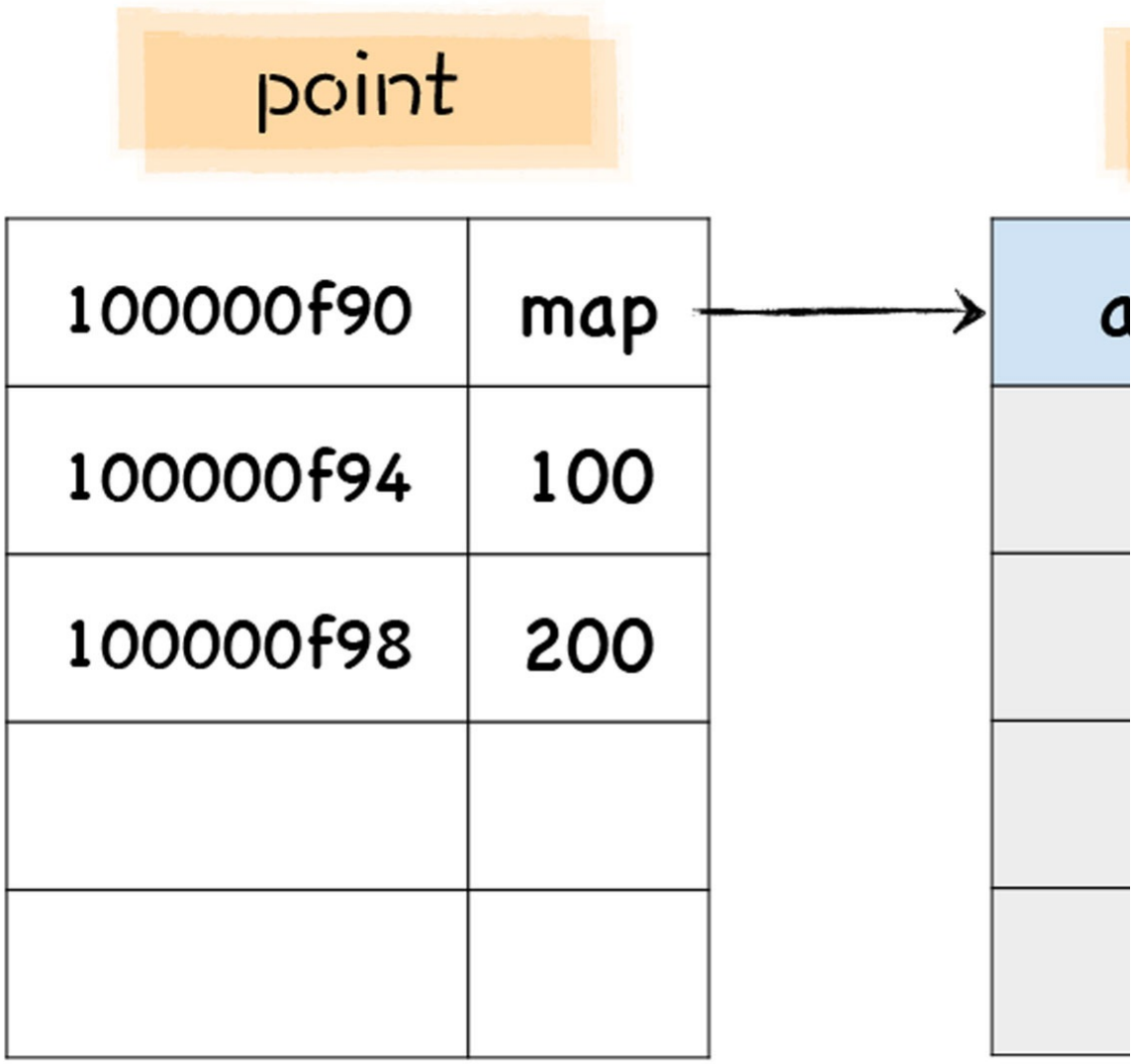
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

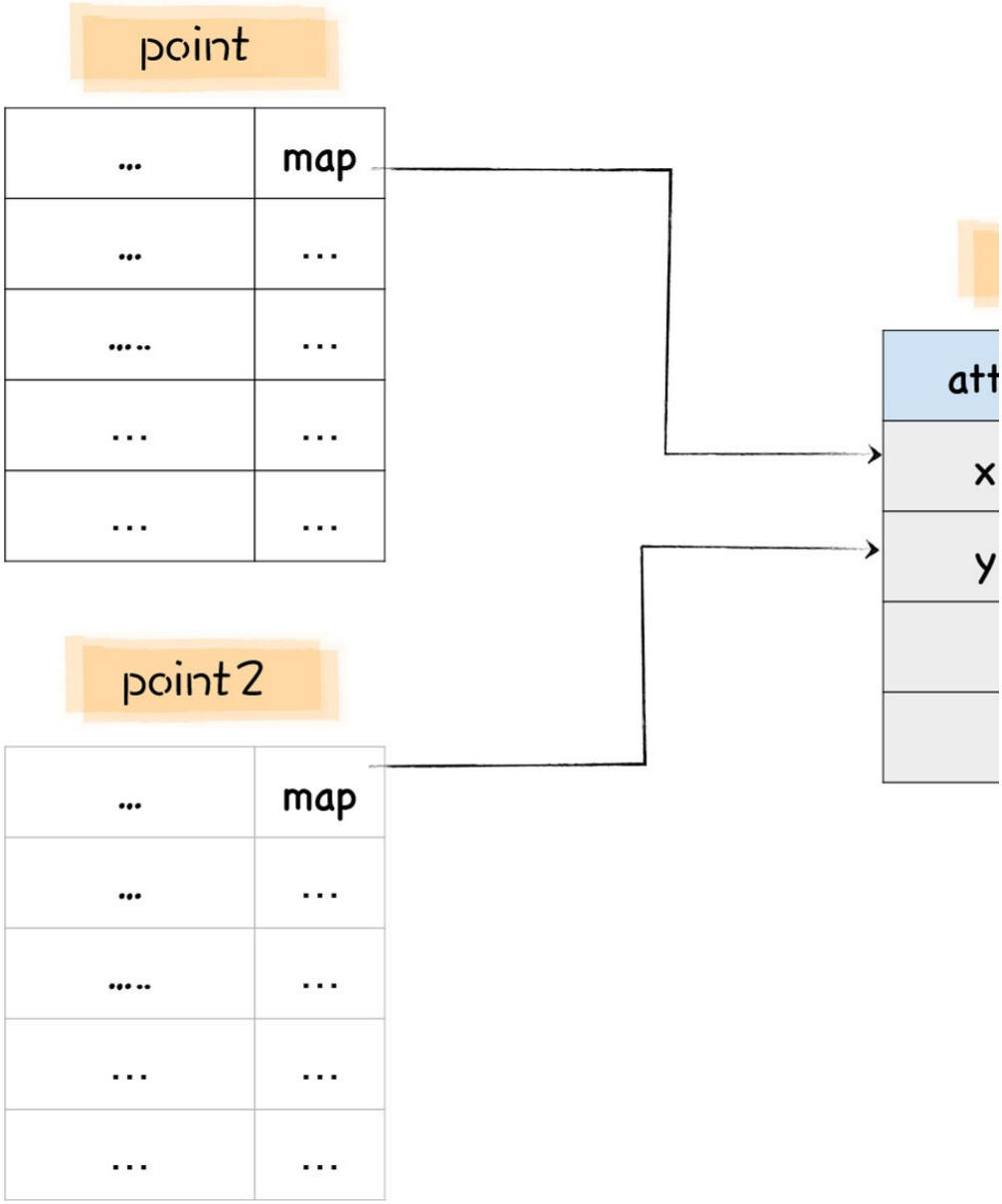
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

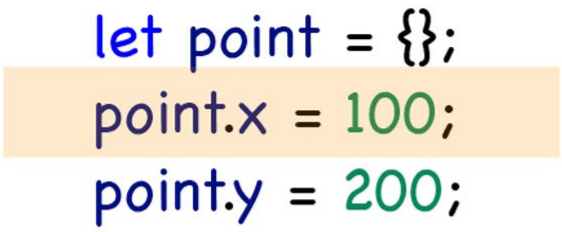
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

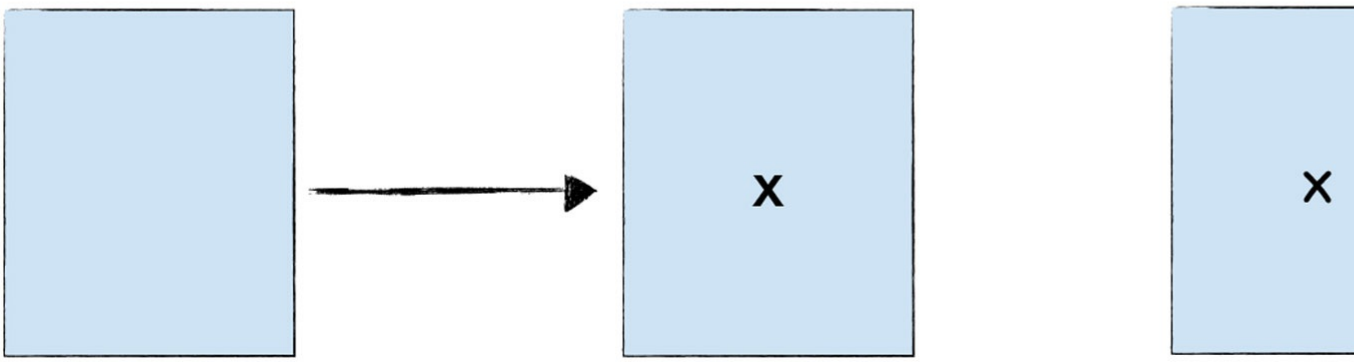
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

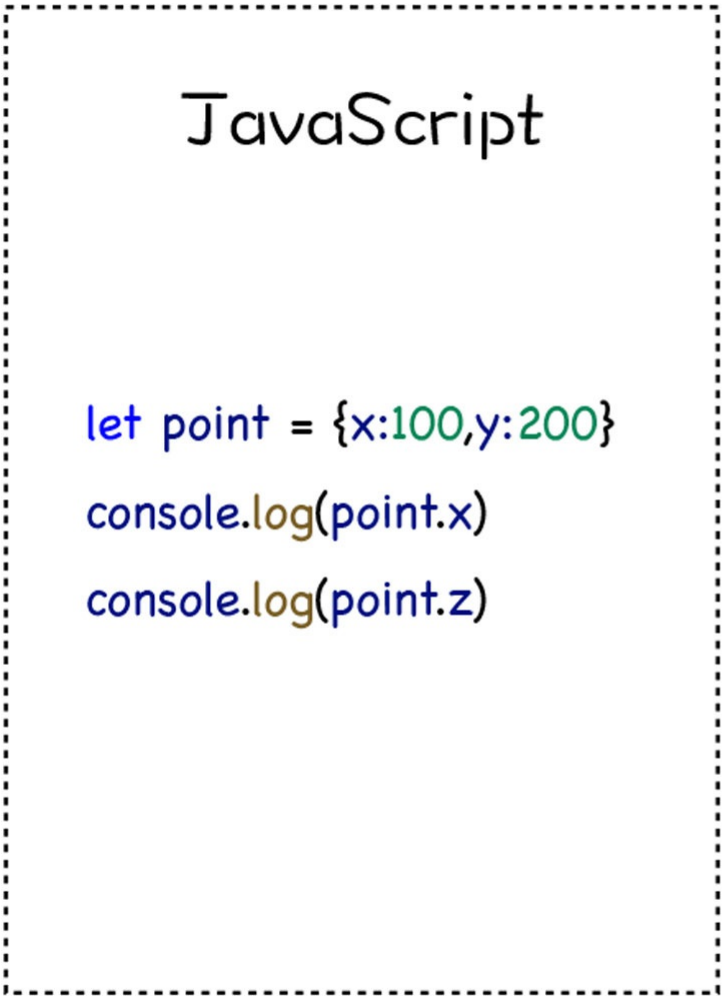
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

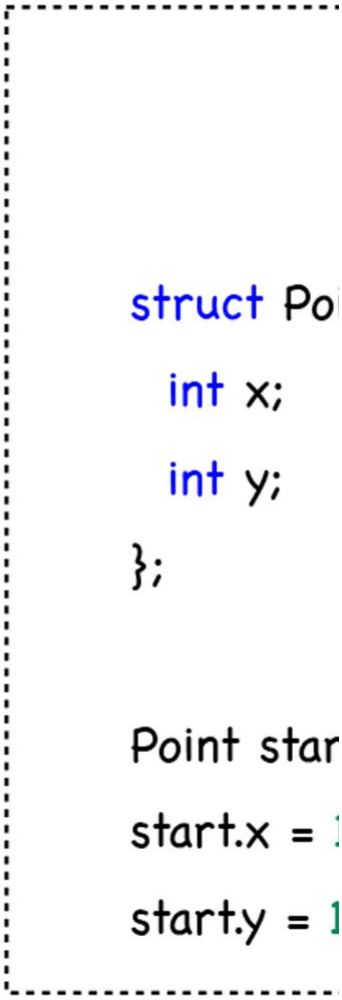
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

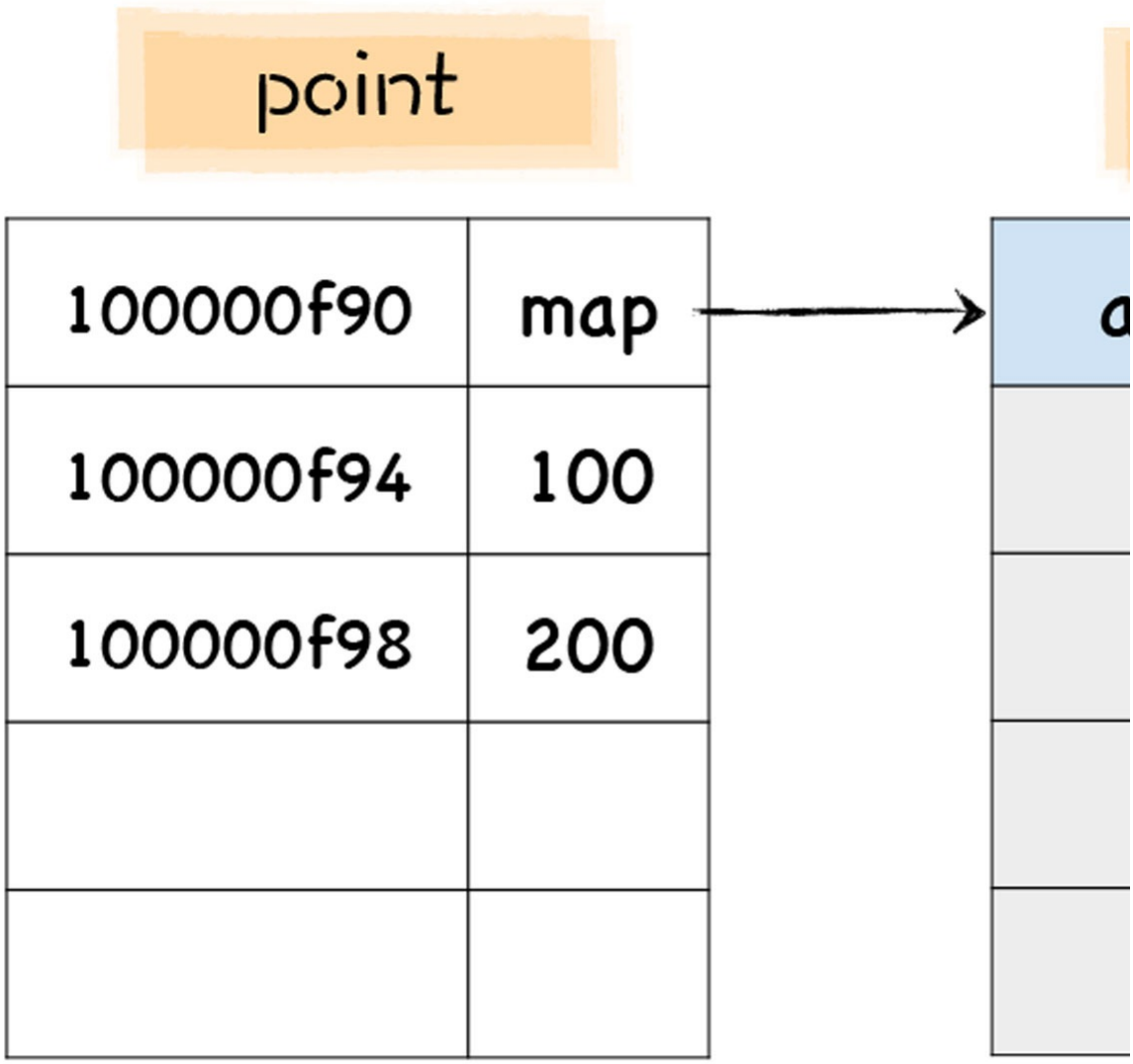
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

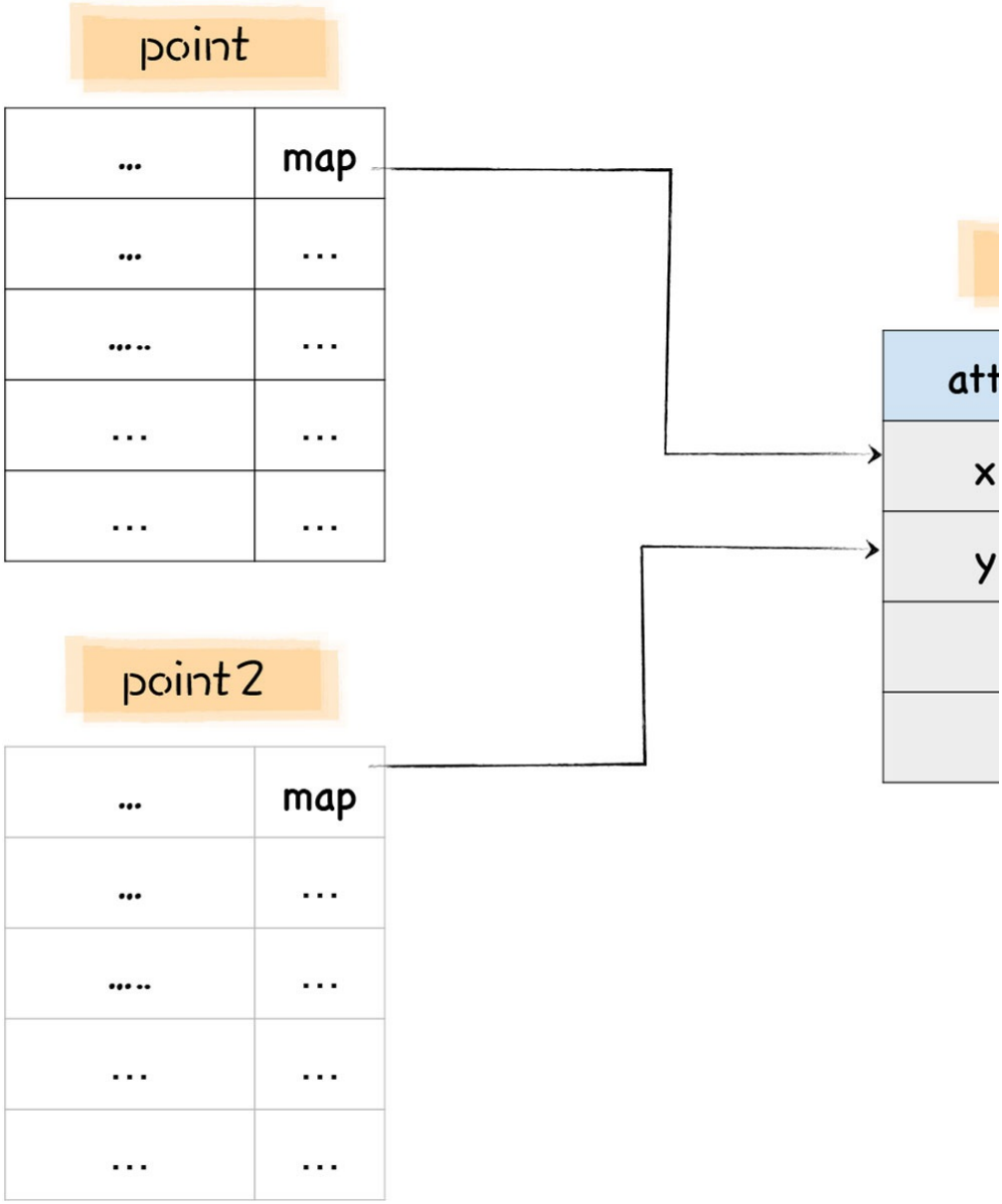
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

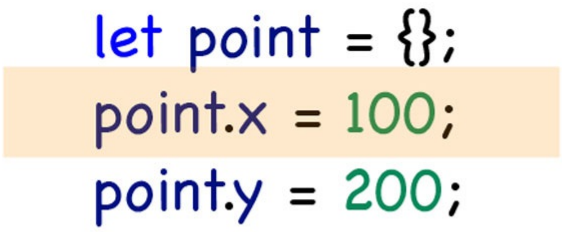
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

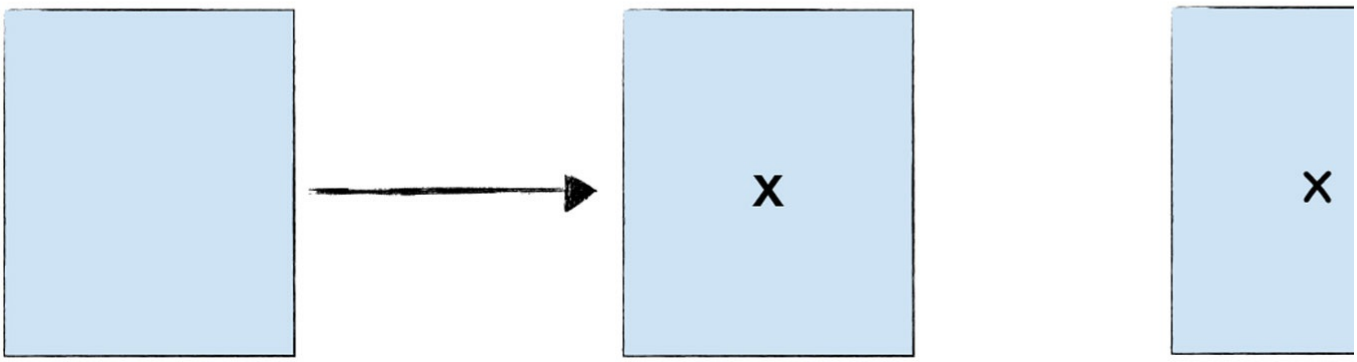
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

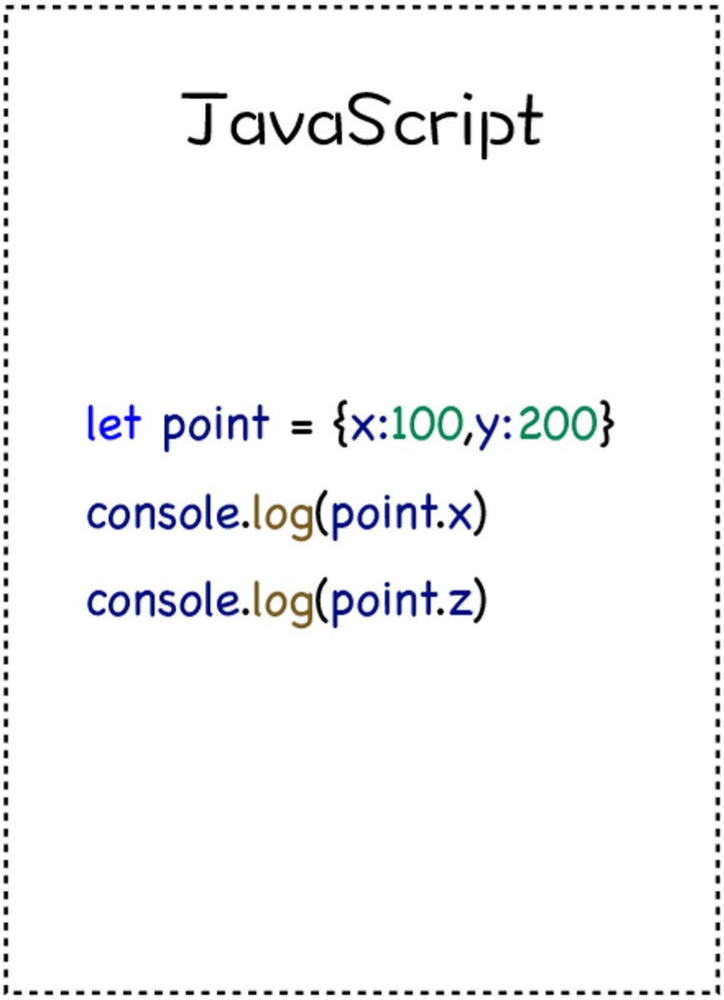
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

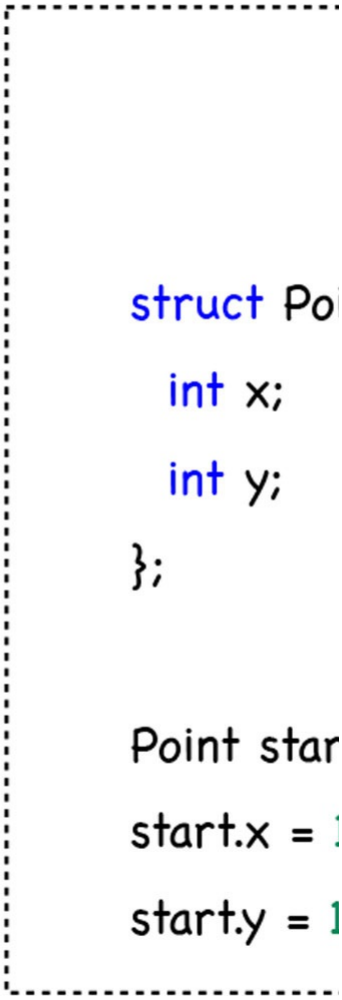
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

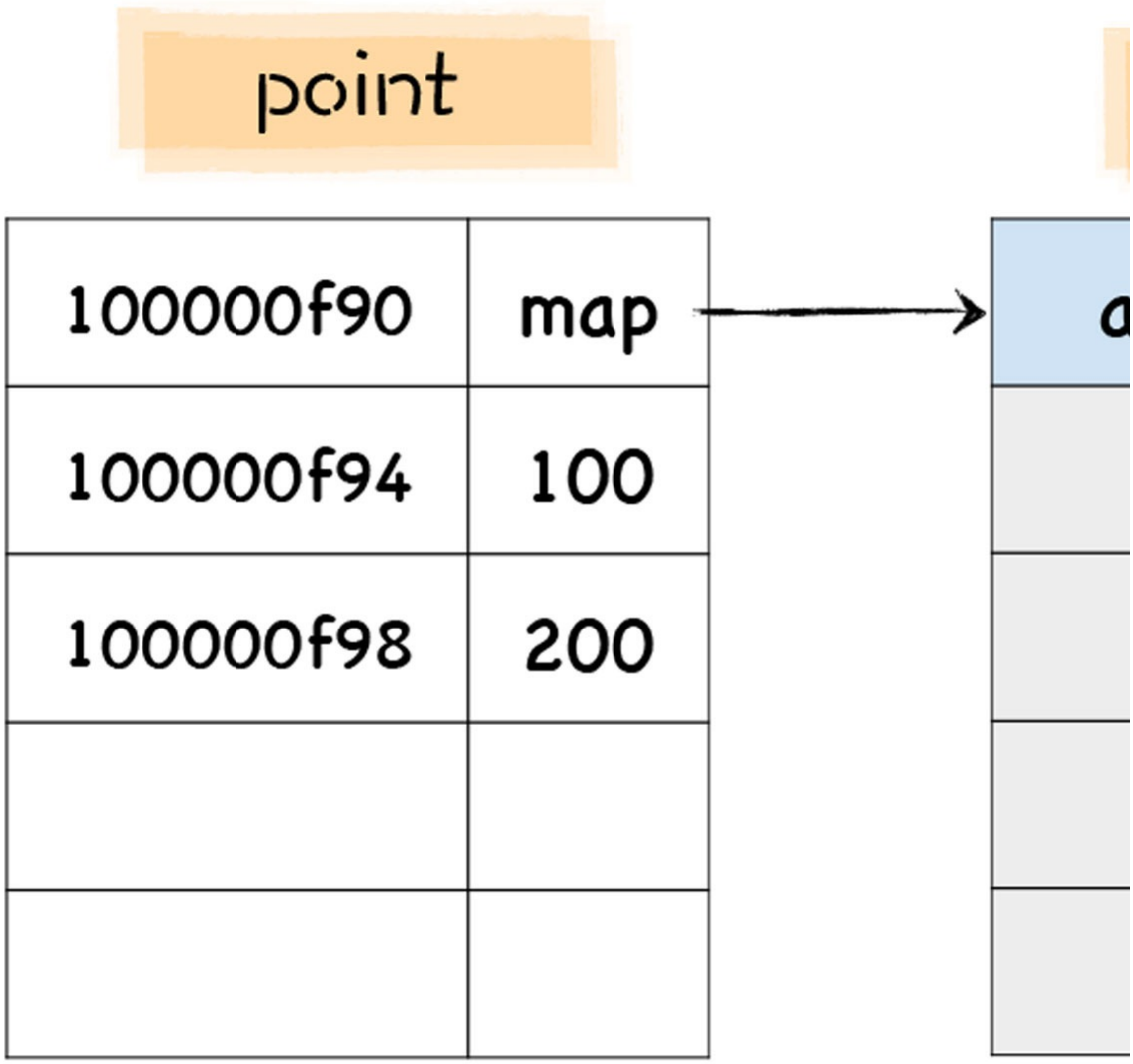
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

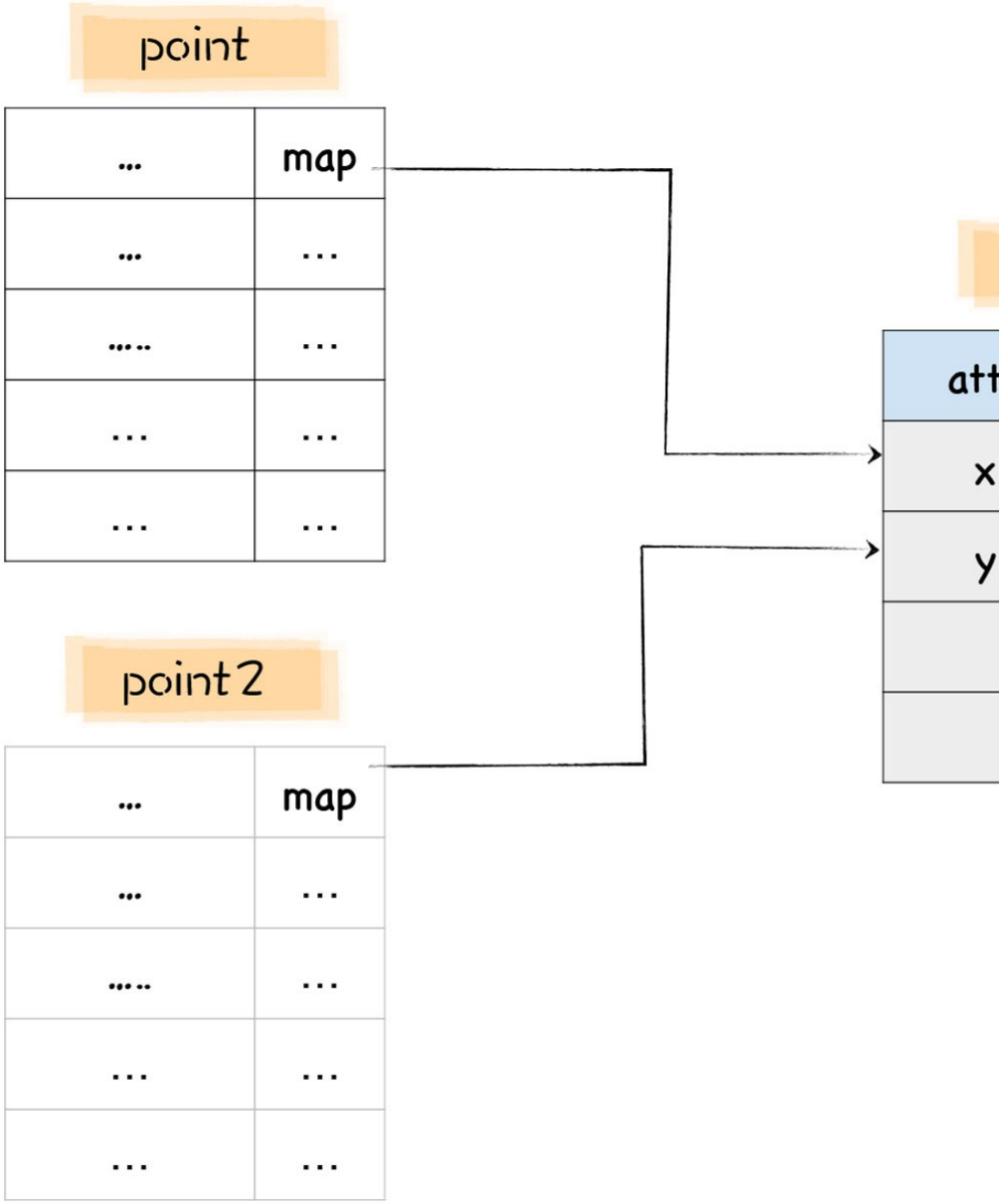
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

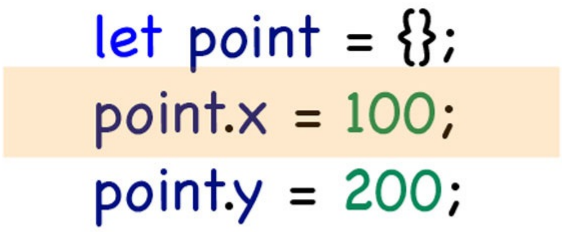
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

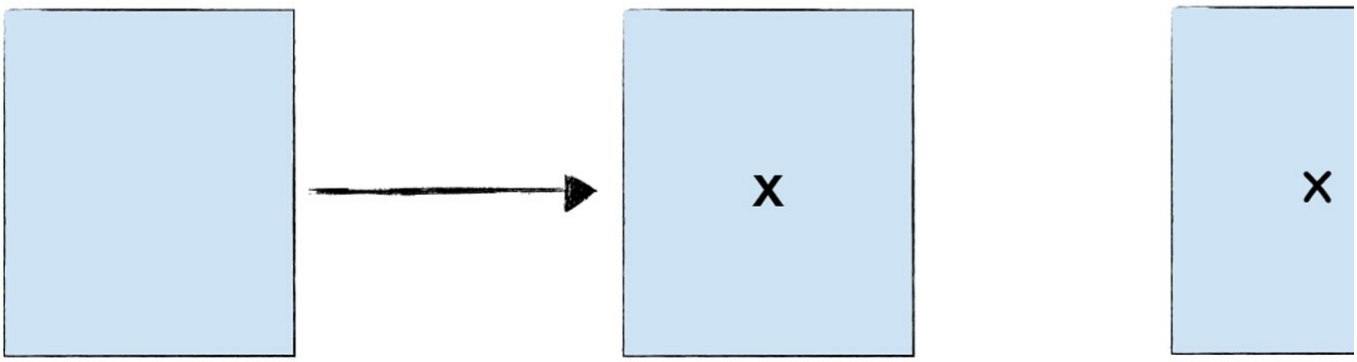
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

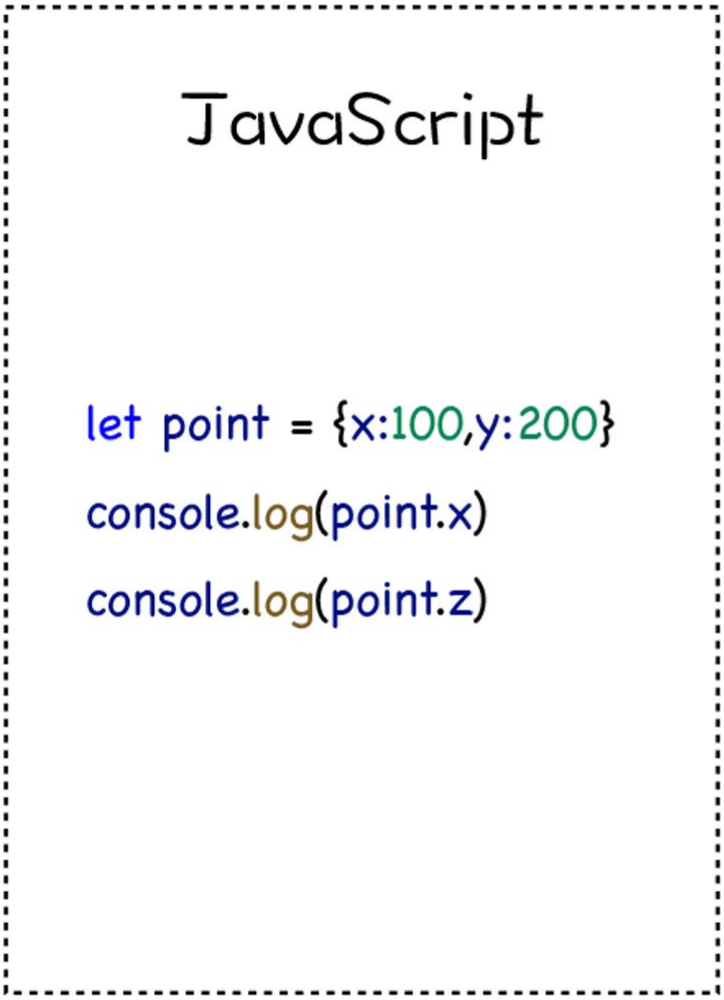
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

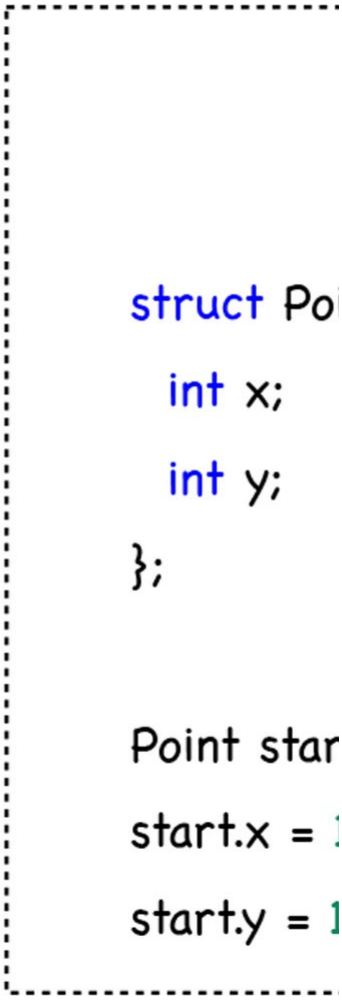
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

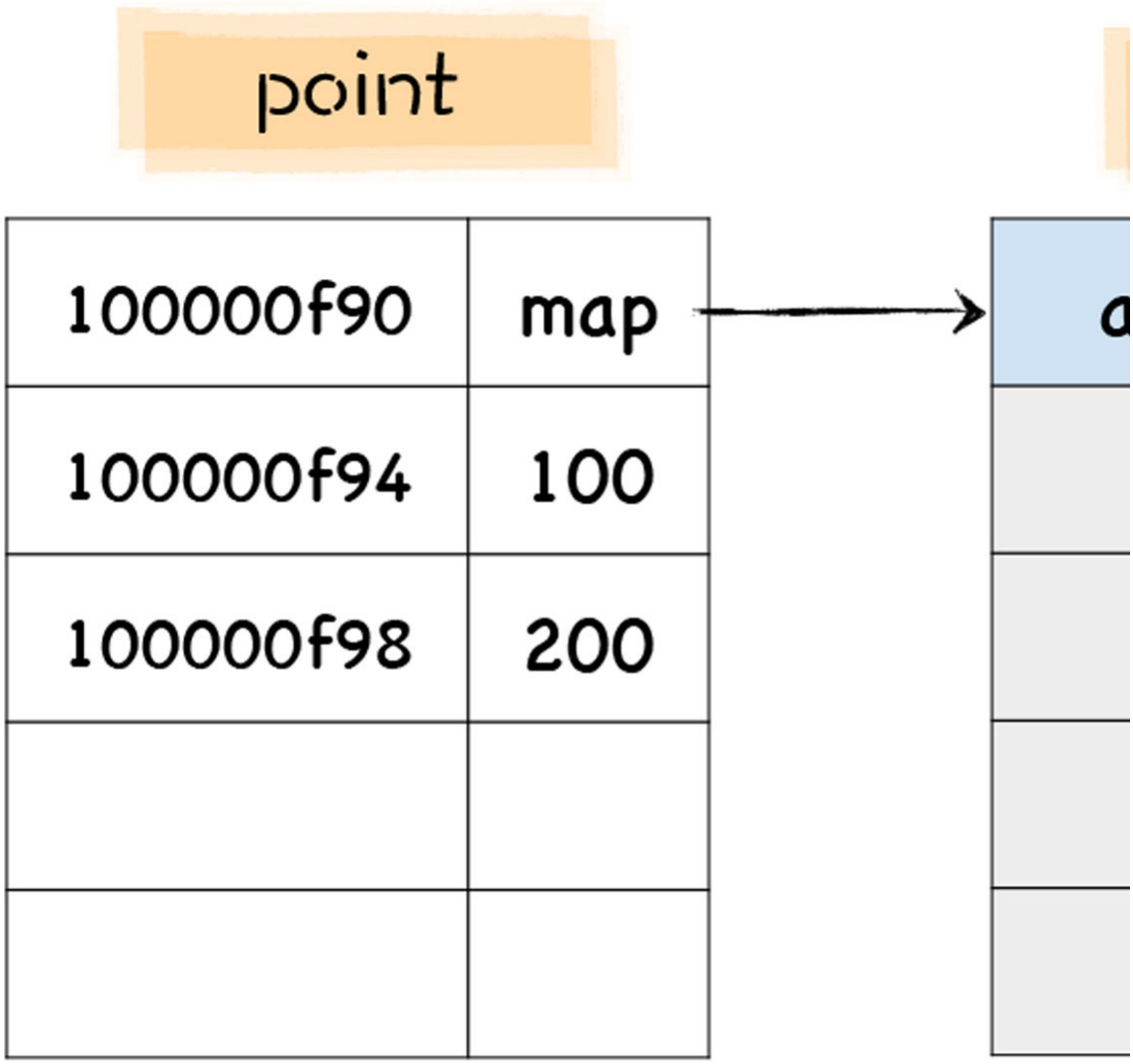
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

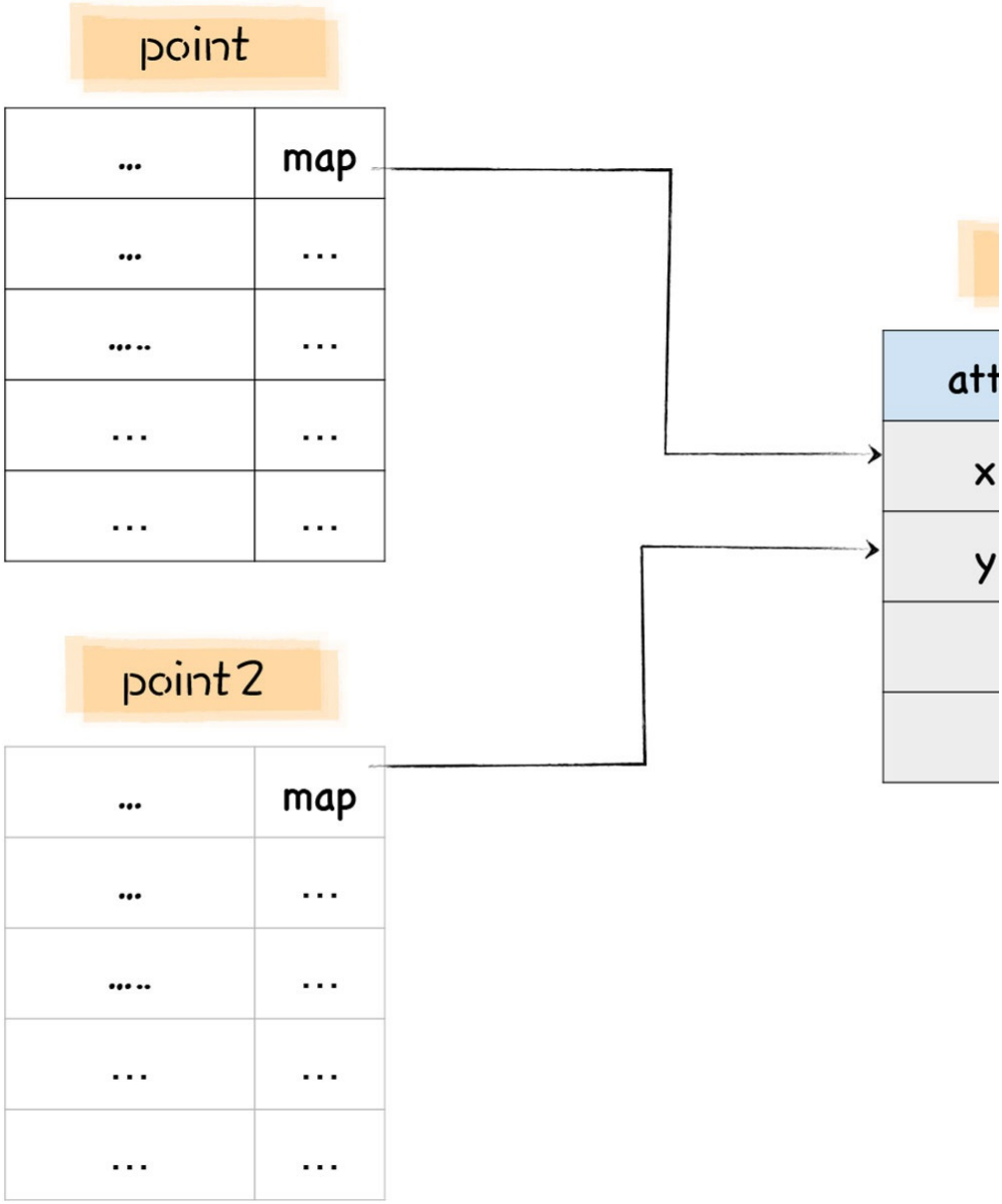
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

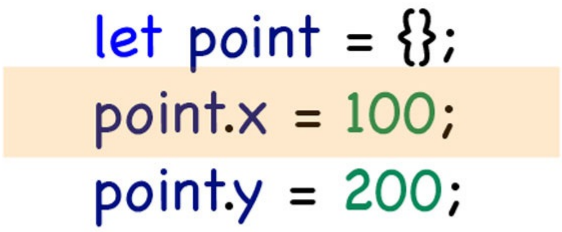
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

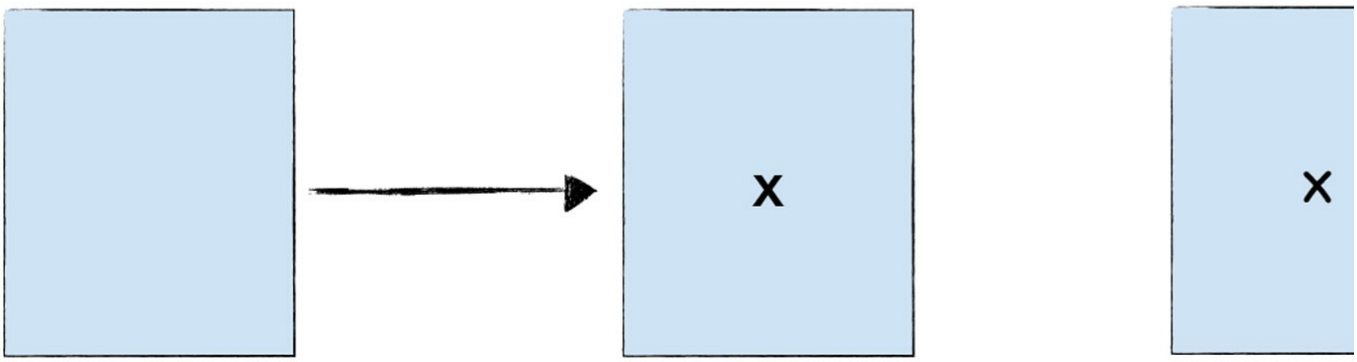
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

你好，我是李兵。

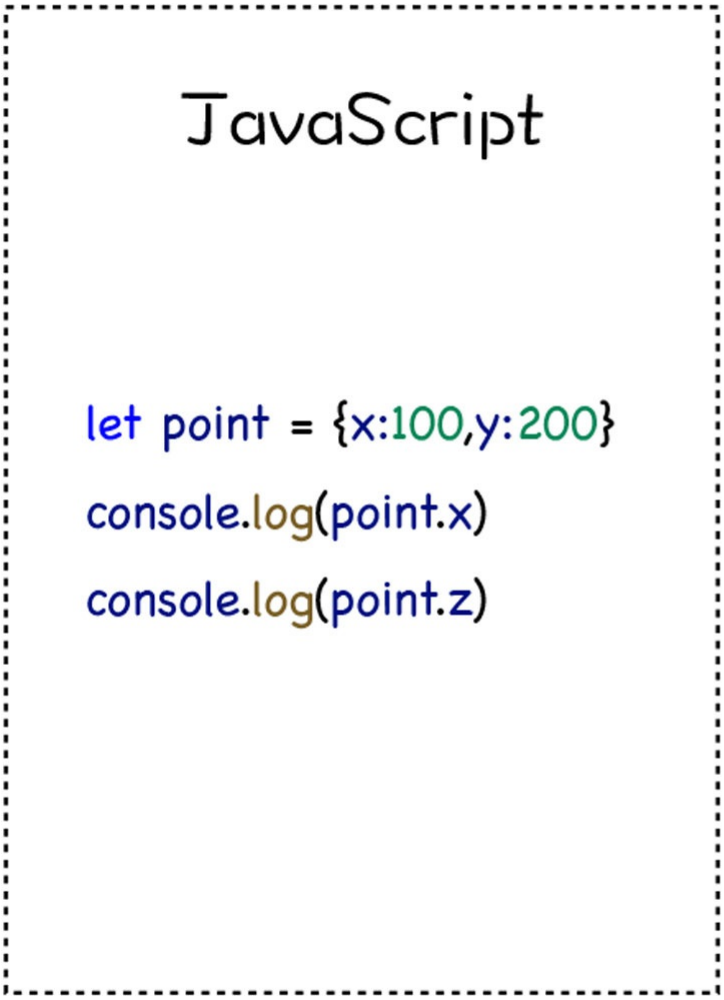
我们知道JavaScript是一门动态语言，其执行效率要低于静态语言，V8为了提升JavaScript的执行速度，借鉴了很多静态语言的特性，比如实现了JIT机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

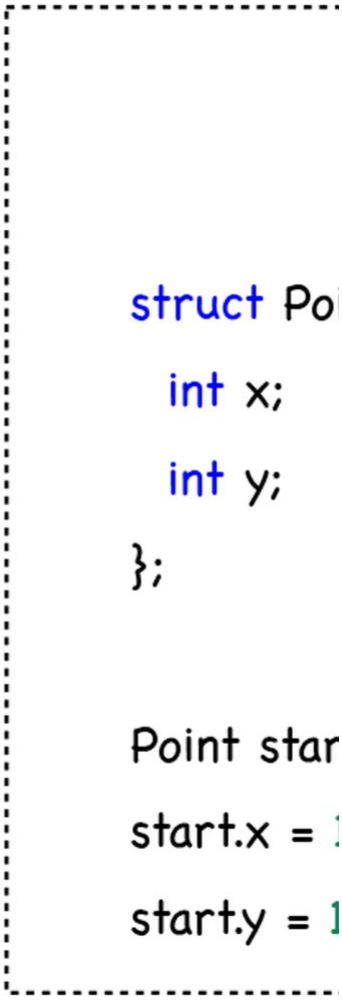
今天我们来重点分析下V8中的隐藏类，看看它是怎么提升访问对象属性值速度的。

为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的JavaScript，另外一段静态语言的C++的源码，具体源码你可以参看下图：





那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript在运行时，对象的属性是可以被修改的，所以当V8使用了一个对象时，比如使用了 start.x的时候，它并不知道该对象中是否有x，也不知道x相对于对象的偏移量是多少，也可以说V8并不知道该对象的具体的形状。

那么，当在JavaScript中要查询对象start中的x属性时，V8会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课程中的内容）。

这种动态查询对象属性的方式和C++这种静态语言不同，C++在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如Point结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，Point的形状是无法被改变的。

那么在C++中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在C++中使用start.x的时候，编译器会直接将x相对于start的地址写进汇编指令中，那么当使用了对象start中的x属性时，CPU就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

什么是隐藏类(Hidden Class）？

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到V8中呢？

答案是**可行**的。

目前所采用的一个思路就是将JavaScript中的对象静态化，也就是V8在运行JavaScript的过程中，会假设JavaScript中的对象是静态的，具体地讲，V8对每个对象做如下两点假设：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8就可以对JavaScript中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

- 对象中所包含的所有的属性；
- 每个属性相对于对象的偏移量。

有了隐藏类之后，那么当V8访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了V8查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
let point = {x:100,y:200}
```

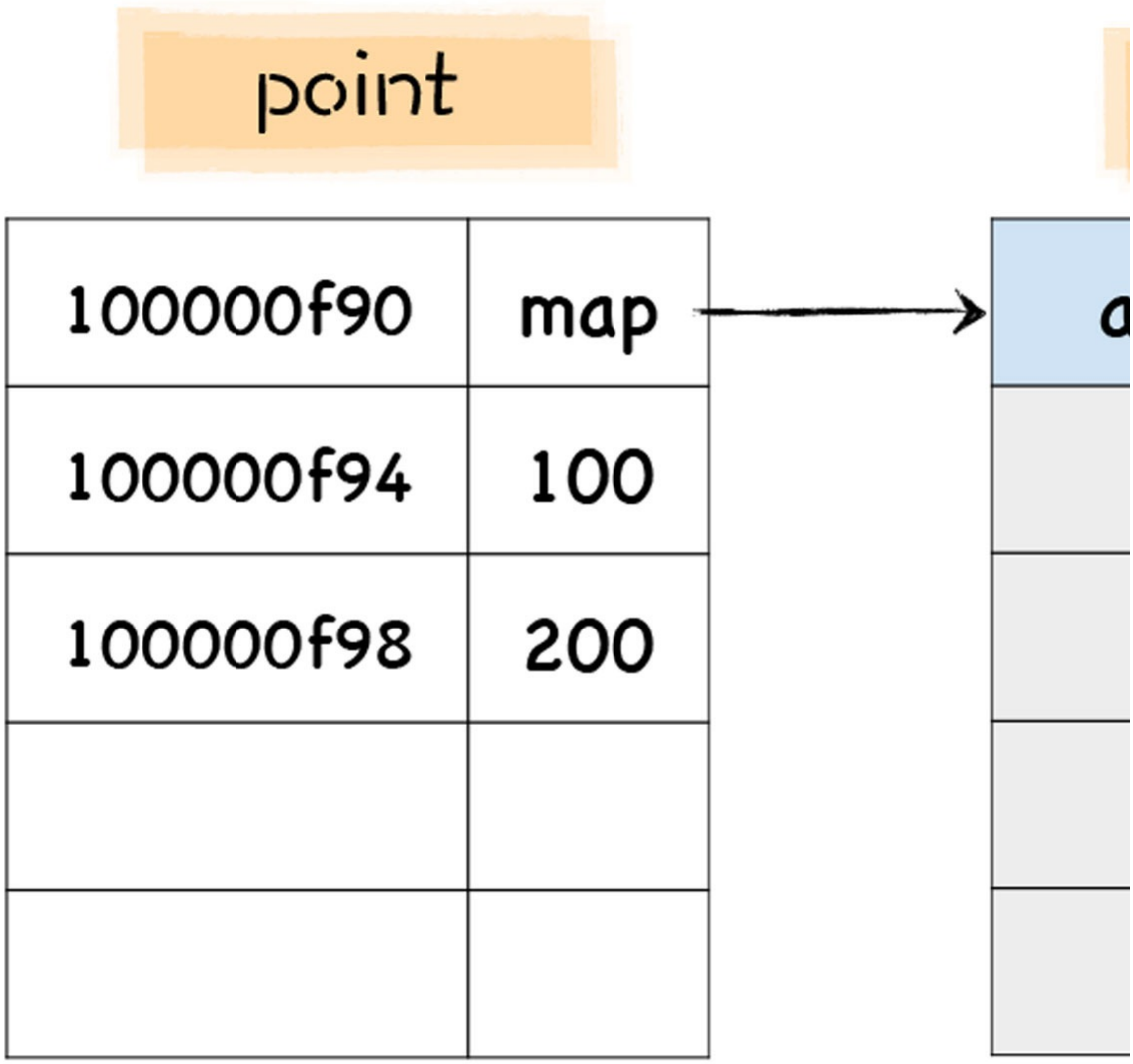
当V8执行到这段代码时，会先为point对象创建一个隐藏类，在V8中，把隐藏类又称为**map**，每个对象都有一个map属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如point对象的隐藏类就包括了x和y属性，x的偏移量是4，y的偏移量是8。

point对象的隐藏

attr	offse
x	4
y	8

注意，这是point对象的map，它不是point对象本身。关于point对象和map之间的关系，你可以参看下图：



在这张图中，左边的是point对象在内存中的布局，右边是point对象的map，我们可以看到，point对象的第一个属性就指向了它的map，关于如何通过浏览器查看对象的map，我们在《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》这节课也做过简单的分析，你可以回顾下这节内容。

有了map之后，当你再次使用point.x访问x属性时，V8会查询point的map中x属性相对point对象的偏移量，然后将point对象的起始位置加上偏移量，就得到了x属性的值在内存中的位置，有了这个位置也就拿到了x的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8通过引入隐藏类，模拟C++这种静态语言的机制，从而达到静态语言的执行效率。

实践：通过d8查看隐藏类

了解了隐藏类的工作机制，我们可以使用d8提供的API DebugPrint来查看point对象中的隐藏类。

```
let point = {x:100,y:200};
%DebugPrint(point);
```

这里你需要注意，在使用d8内部API时，有一点很容易出错，就是需要为JavaScript代码加上分号，不然d8会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存test.js文件中，再执行：

```
 d8 --allow-natives-syntax test.js
```

执行这段命令，就可以打印出point对象的基础结构了，打印出来的结果如下所示：

```
DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
 - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19dc080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }
0x19dc08284d11: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
 - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
 - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
 - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
 - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

从这段point的内存结构中，我们可以看到，point对象的第一个属性就是map，它指向了0x19dc08284d11这个地址，这个地址就是V8为point对象创建的隐藏类，除了map属性之外，还有我们之前介绍过的prototype属性，elements属性和properties属性（关于这些属性的函数，你可以参看《03丨快属性和慢属性：V8是怎样提升对象属性访问速度的？》和《05丨原型链：V8是如何实现对象继承的？》这两节的内容）。

多个对象共用一个隐藏类

现在我们知道了在V8中，每个对象都有一个map属性，该属性值指向该对象的隐藏类。不过如果两个对象的形状是相同的，V8就会为其复用同一个隐藏类，这样有两个好处：

1. 减少隐藏类的创建次数，也间接加速了代码的执行速度；
2. 减少了隐藏类的存储空间。

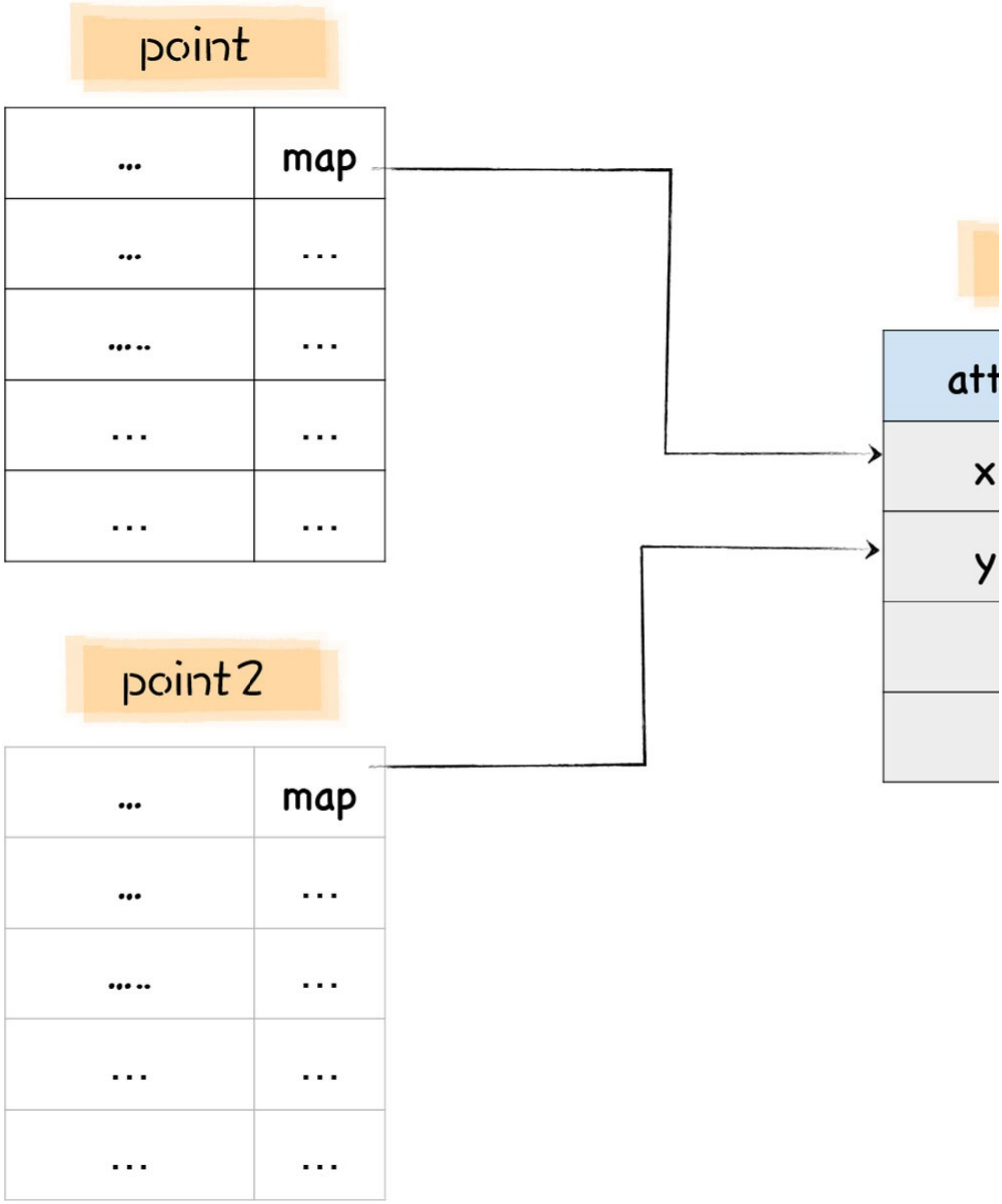
那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

接下来我们就来创建两个形状一样的对象，然后看看它们的map属性是不是指向了同一个隐藏类，你可以参看下面的代码：

```
let point = {x:100,y:200};
let point2 = {x:3,y:4};
%DebugPrint(point);
%DebugPrint(point2);
```

当V8执行到这段代码时，首先会为point对象创建一个隐藏类，然后继续创建point2对象。在创建point2对象的过程中，发现它的形状和point是一样的。这时候，V8就会将point的隐藏类给point2复用，具体效果你可以参看下图：



你也可以使用d8来证实下，同样使用这个命令：

```
 d8 --allow-natives-syntax test.js
```

打印出来的point和point2对象，你会发现它们的map属性都指向了同一个地址，这也就意味着它们共用了同一个map。

重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8为了实现隐藏类，需要两个假设条件：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

但是，JavaScript依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着V8要为新改变的对象重新构建新的隐藏类，这对于V8的执行效率来说，是一笔大的开销。

通俗地理解，给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发V8为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
let point = {};
```

```
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);
```

将这段代码保存到test.js文件中，然后执行：

```
 d8 --allow-natives-syntax test.js
```

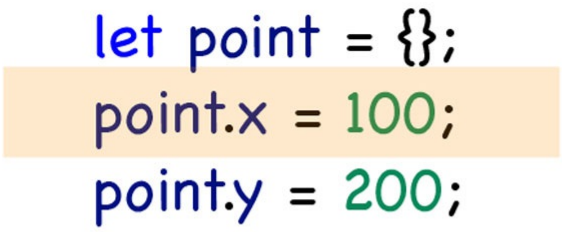
执行这段命令，d8会打印出来不同阶段的point对象所指向的隐藏类，在这里我们只关心point对象map的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

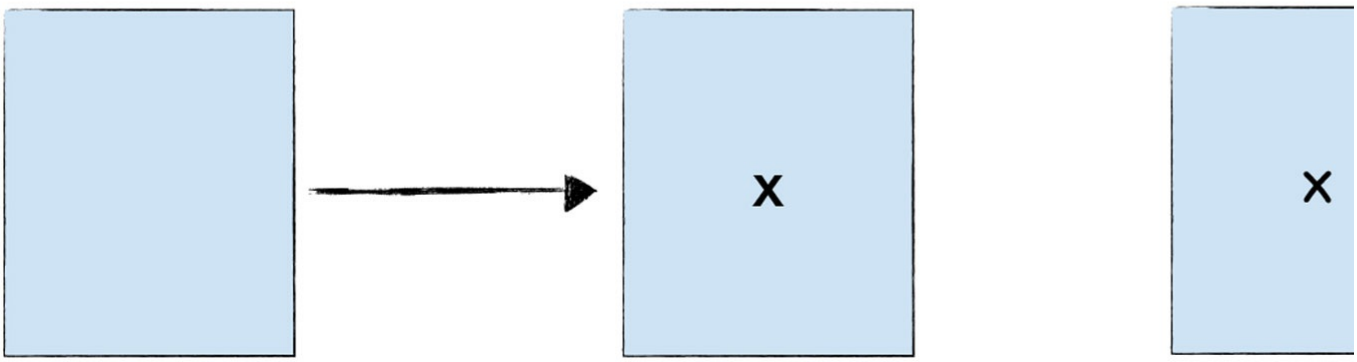
```
DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
 }


DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]
 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - p
 - ...
 - properties: 0x0986080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：





同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时V8也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
let point = {x:100,y:200};
delete point.x
```

我们再次使用d8来打印这段代码中不同阶段的point对象属性，移除多余的信息，最终打印出来的结果如下所示

```
DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 -...
 - properties: 0x1c2f080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
    #y: 200 (const data field 1)
 }


DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - ...
 - properties: 0x1c2f08045567 <FixedArray[0]> {
    #y: 200 (const data field 1)
 }
```

最佳实践

好了，现在我们知道了V8会为每个对象分配一个隐藏类，在执行过程中：

- 如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；
- 如果对象的形状发生了改变，那么V8会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发V8重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

一，使用字面量初始化对象时，要保证属性的顺序是一致的。比如先通过字面量x、y的顺序创建了一个point对象，然后通过字面量y、x的顺序创建一个对象point2，代码如下所示：

```
let point = {x:100,y:200};
let point2 = {y:100,x:200};
```

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

二，尽量使用字面量一次性初始化完整对象属性。因为每次为对象添加一个属性时，V8都会为该对象重新设置隐藏类。

三，尽量避免使用delete方法。delete方法会破坏对象的形状，同样会导致V8为该对象重新生成新的隐藏类。

总结

这节课我们介绍了V8中隐藏类的工作机制，我们先分析了V8引入隐藏类的动机。因为JavaScript是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8无法知道对象的完整形状，那么当查找对象中的属性时，V8就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

- 对象创建好了之后就不会添加新的属性；
- 对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着V8需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

- 在V8中，每个对象都有一个隐藏类，隐藏类在V8中又被称为map。
- 在V8中，每个对象的第一个属性的指针都指向其map地址。
- map描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？
- 如果添加新的属性，那么需要重新构建隐藏类。
- 如果删除了对象中的某个属性，同样也需要构建隐藏类。

思考题

现在我们知道了V8为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数loadX，loadX有一个参数，然后返回该参数的x属性值：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3 ,y:6}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

当V8调用loadX的时候，会先查找参数o的隐藏类，然后利用隐藏类中的x属性的偏移量查找到x的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果loadX在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是V8的设计者，你会采用什么措施来提高loadX函数的执行效率？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。