你好，我是李兵。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

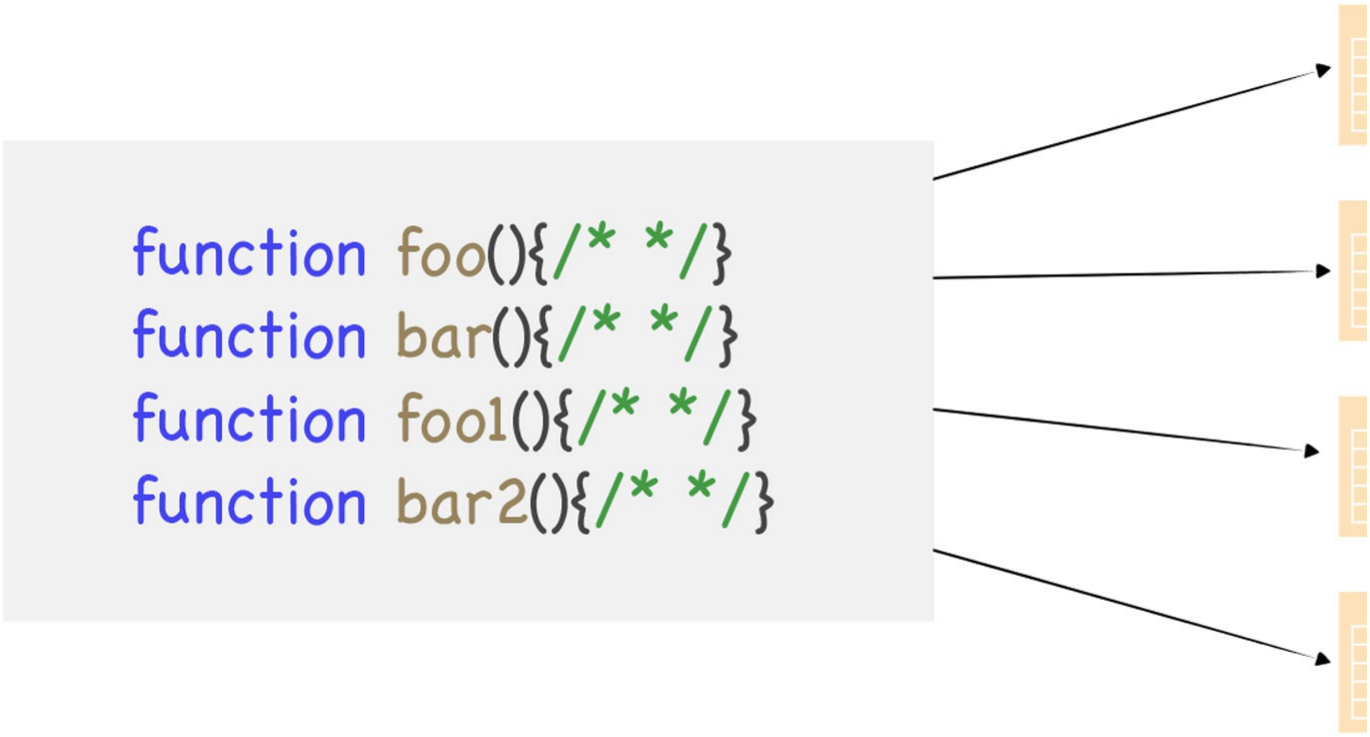
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

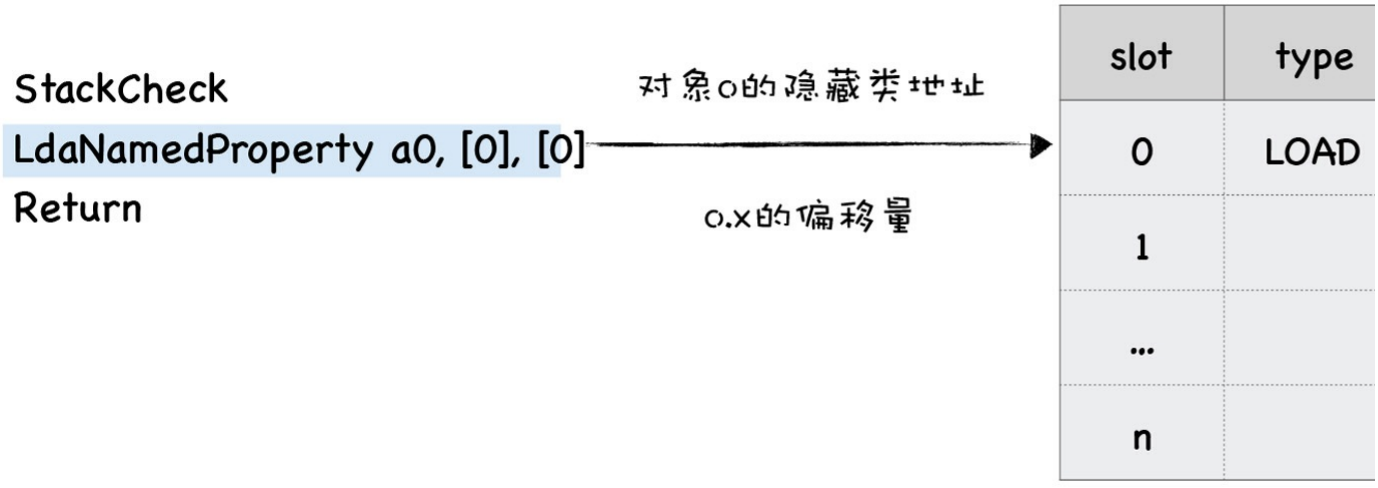
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

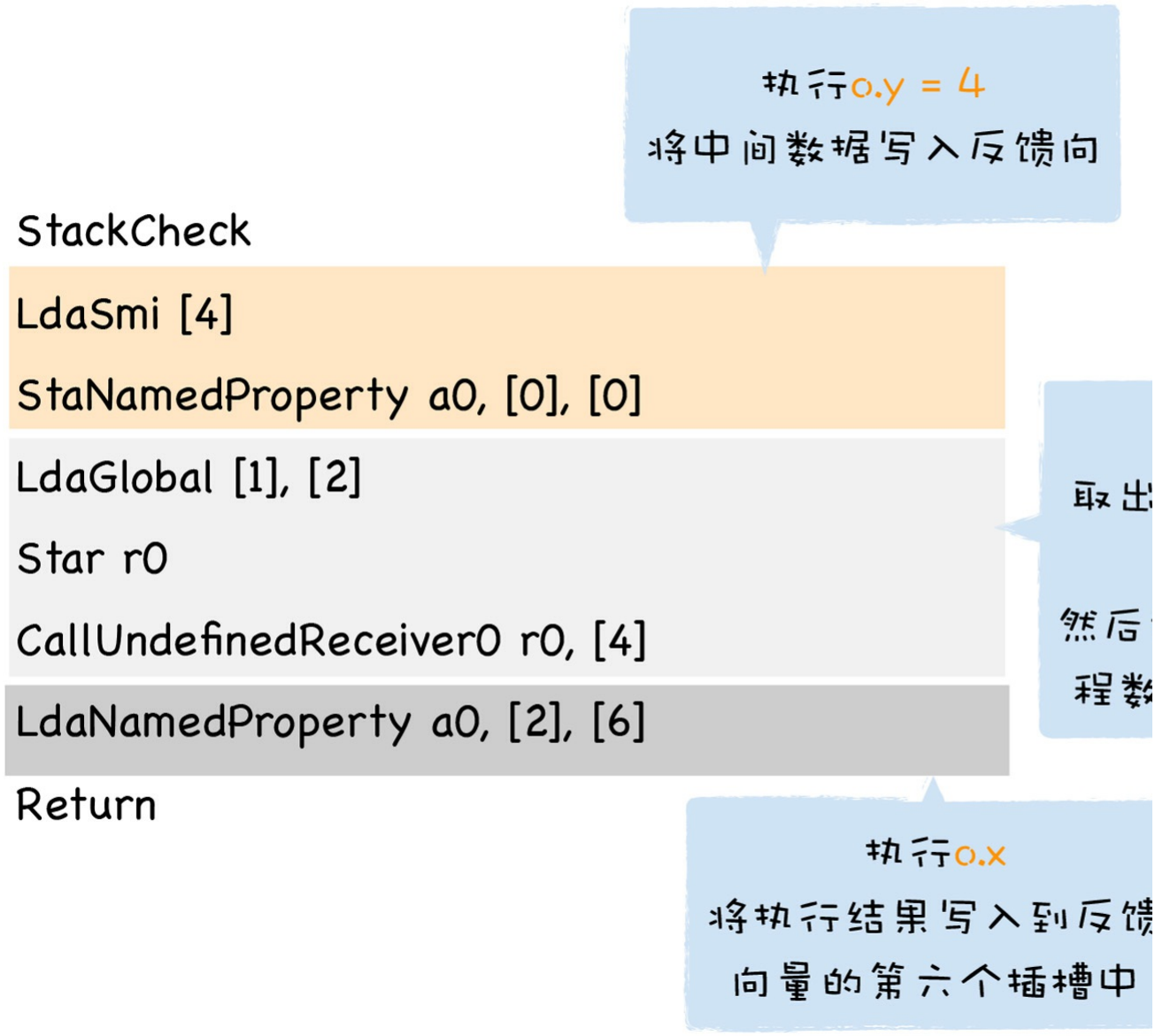
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

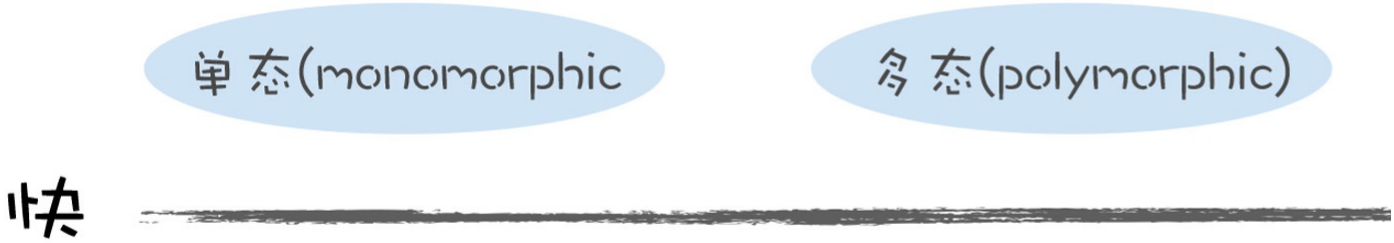
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

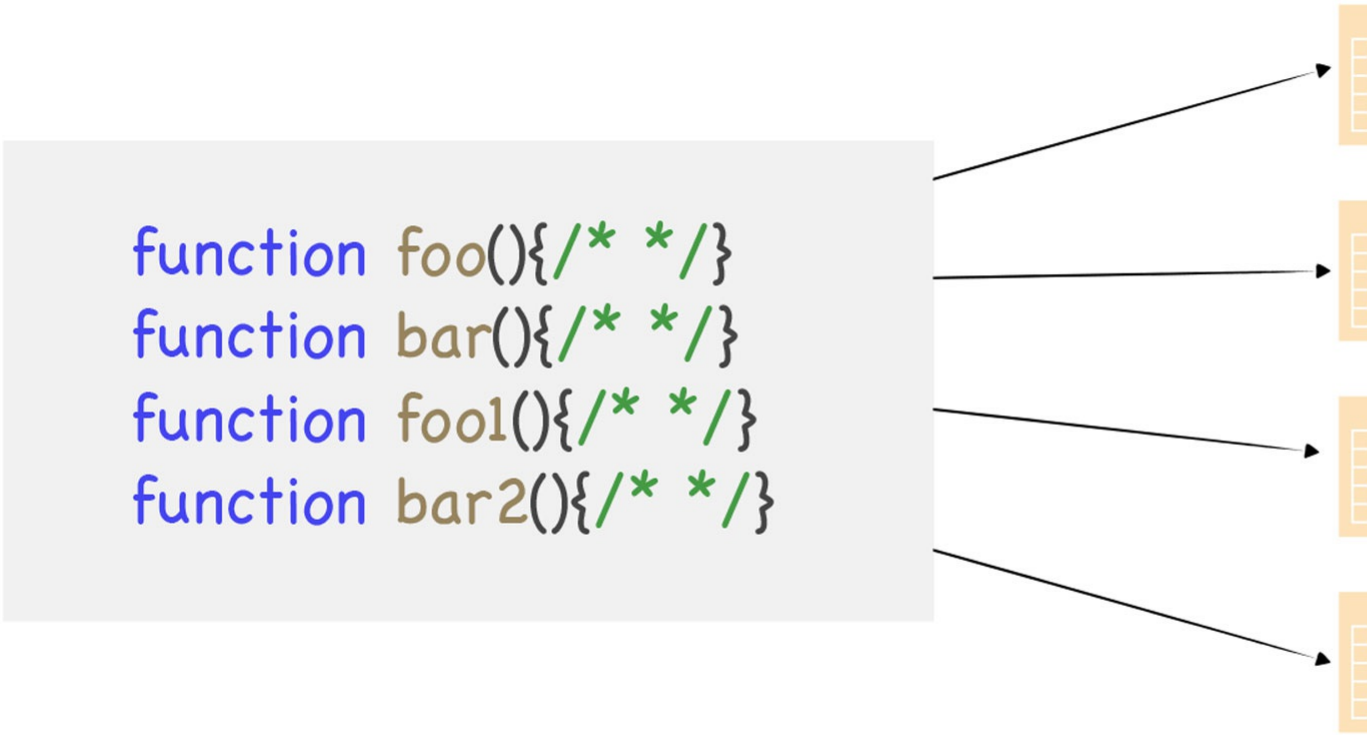
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

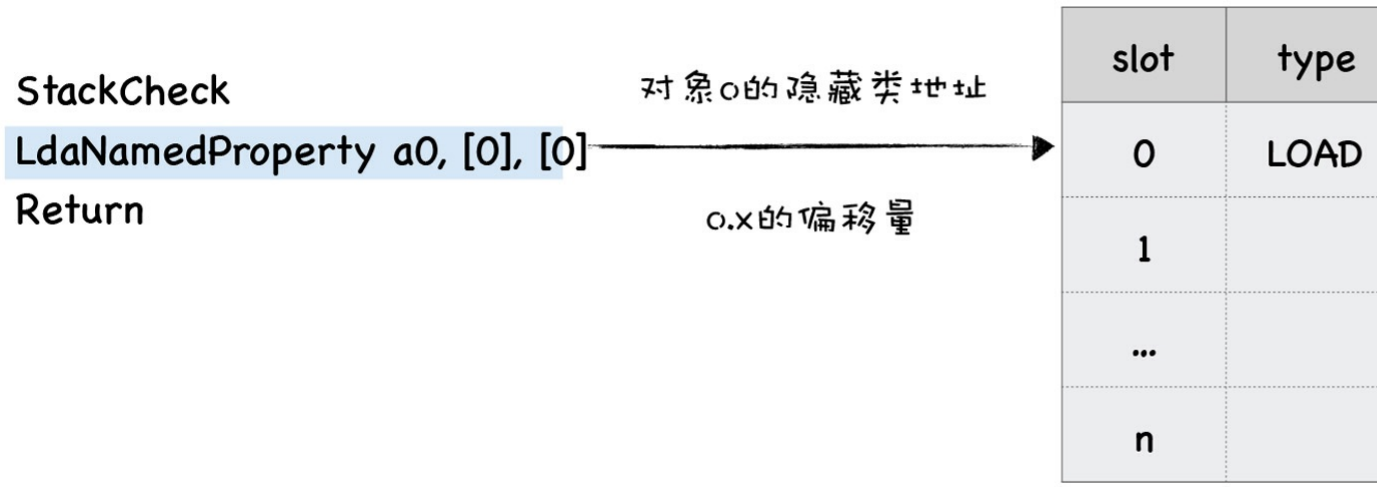
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

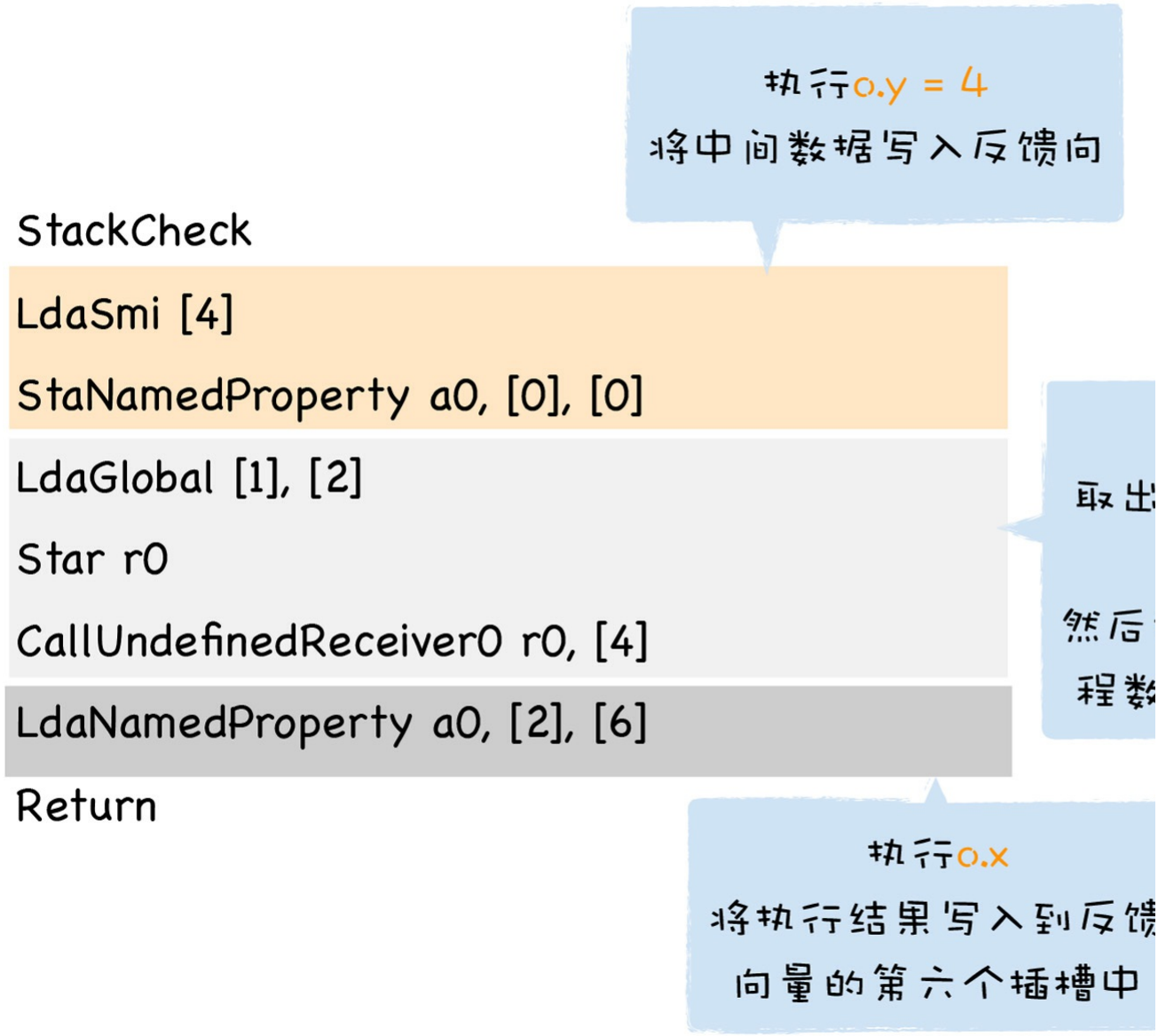
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

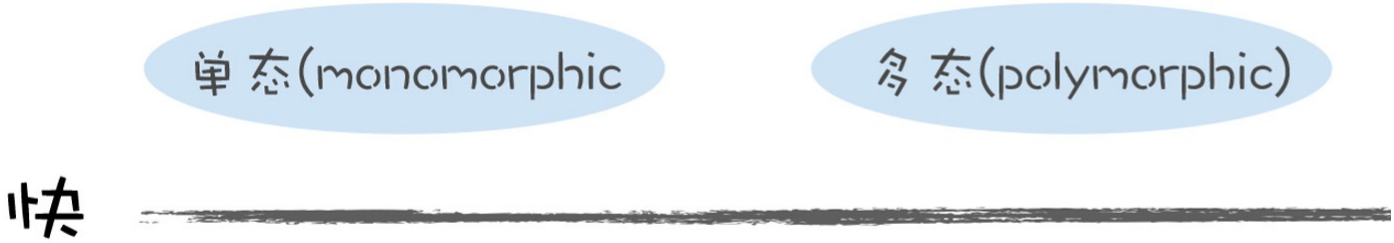
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

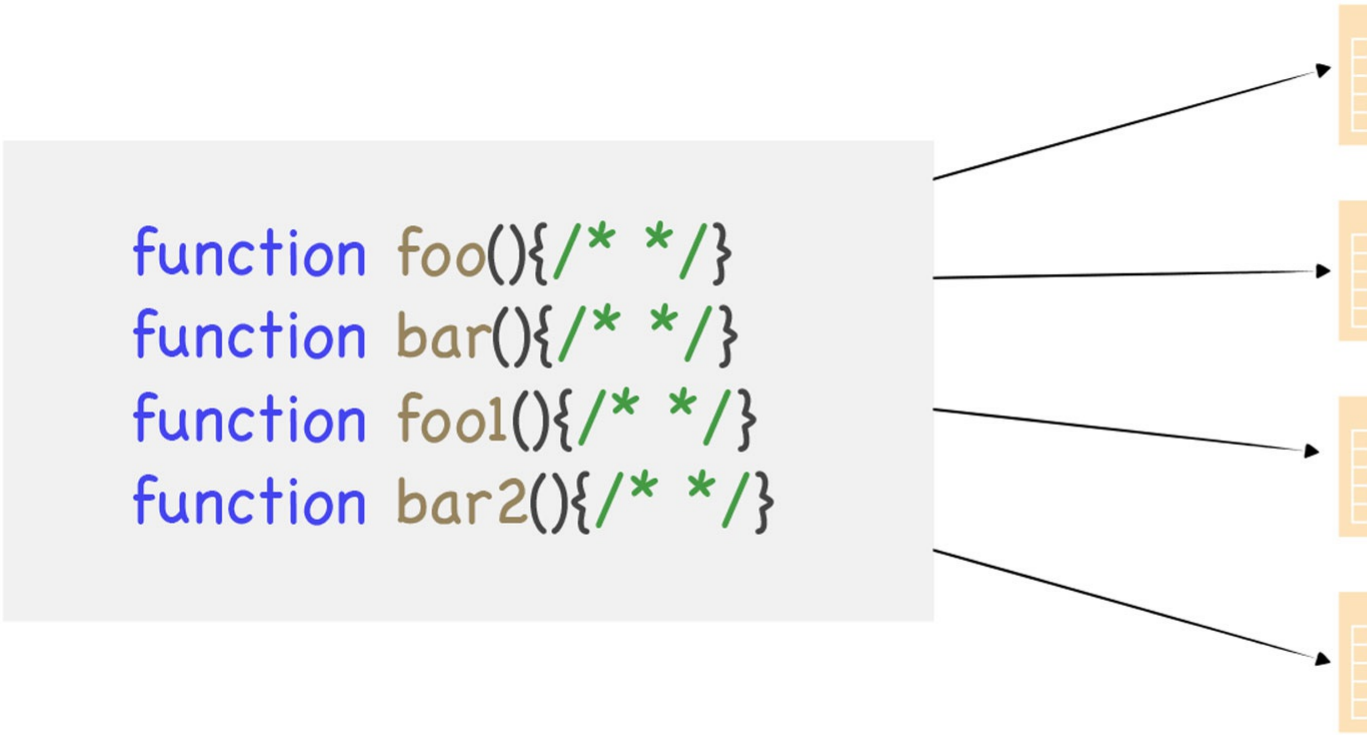
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

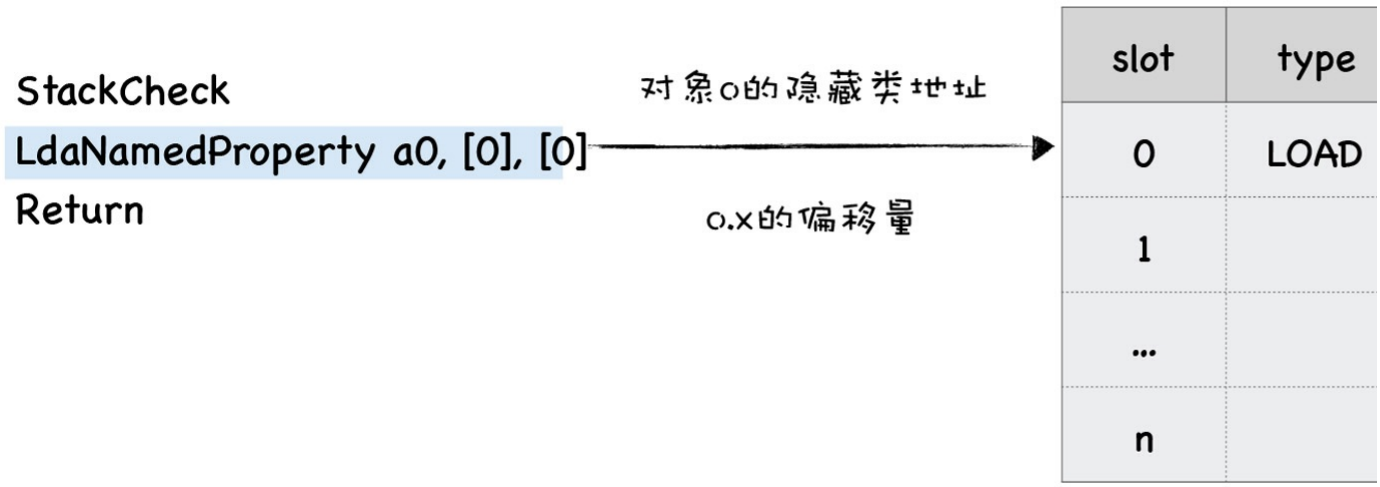
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

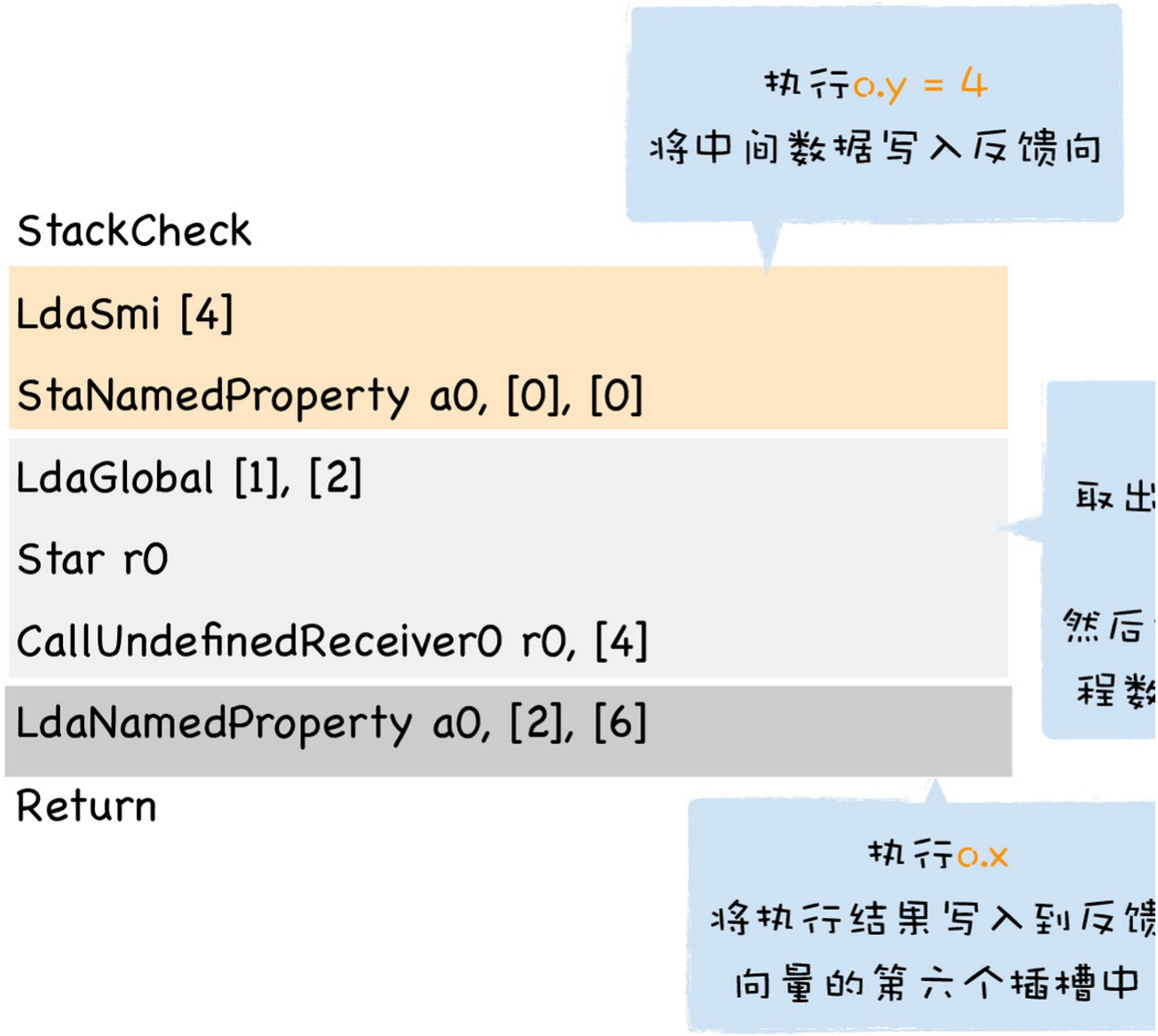
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

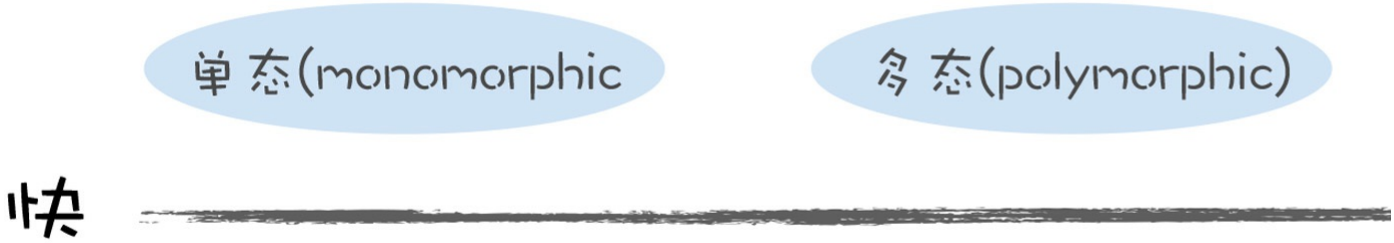
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

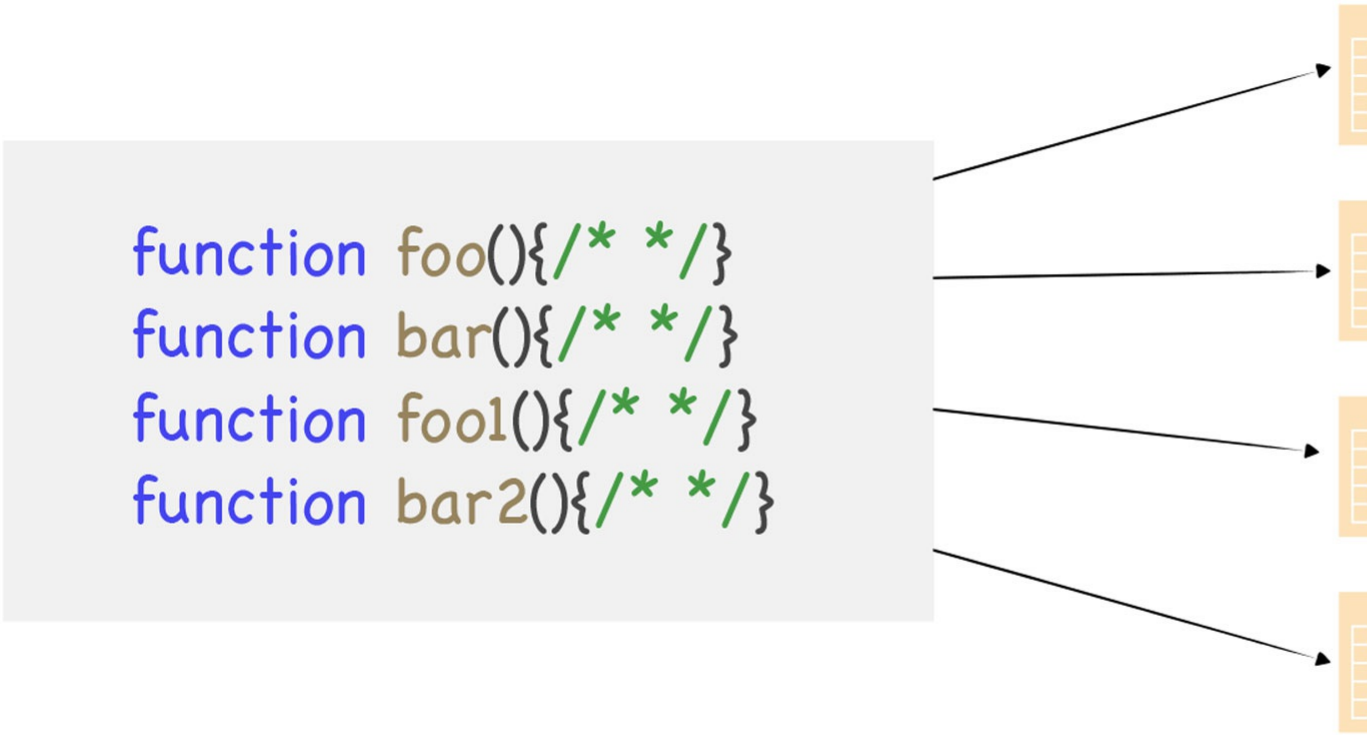
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

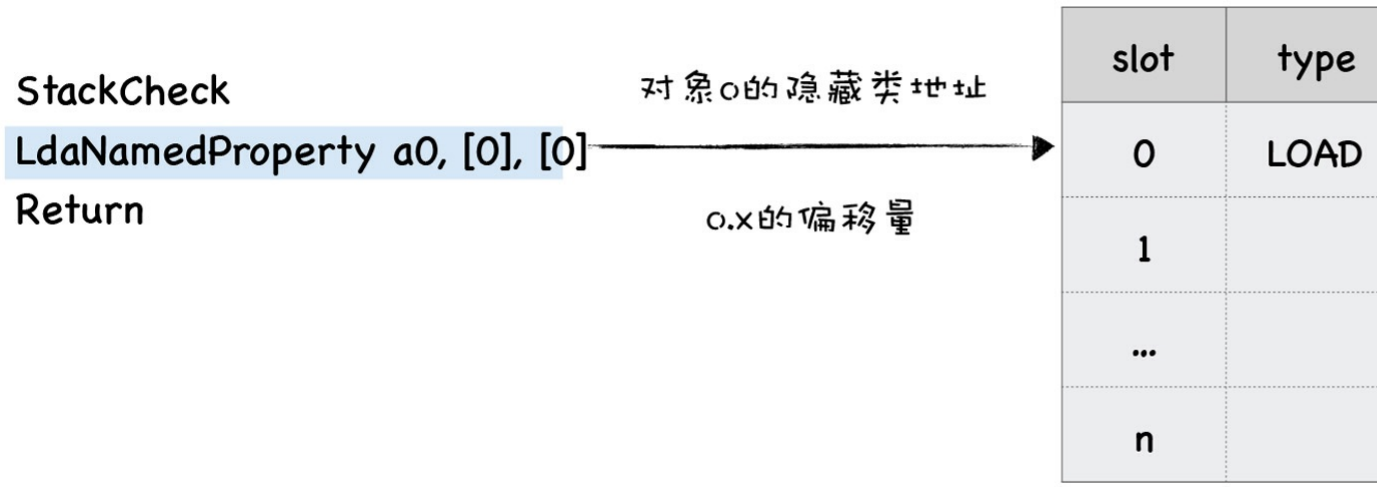
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

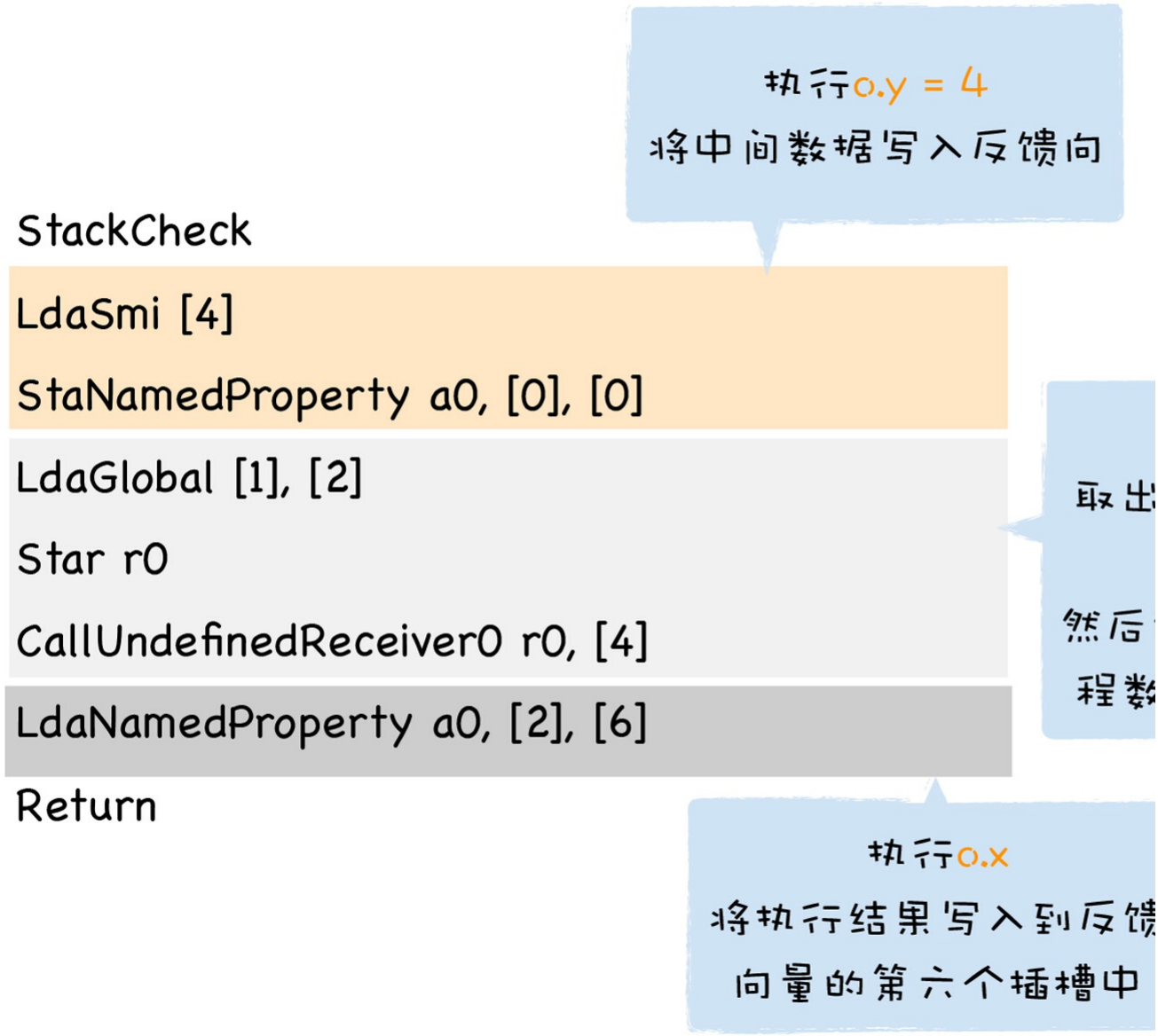
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：

尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

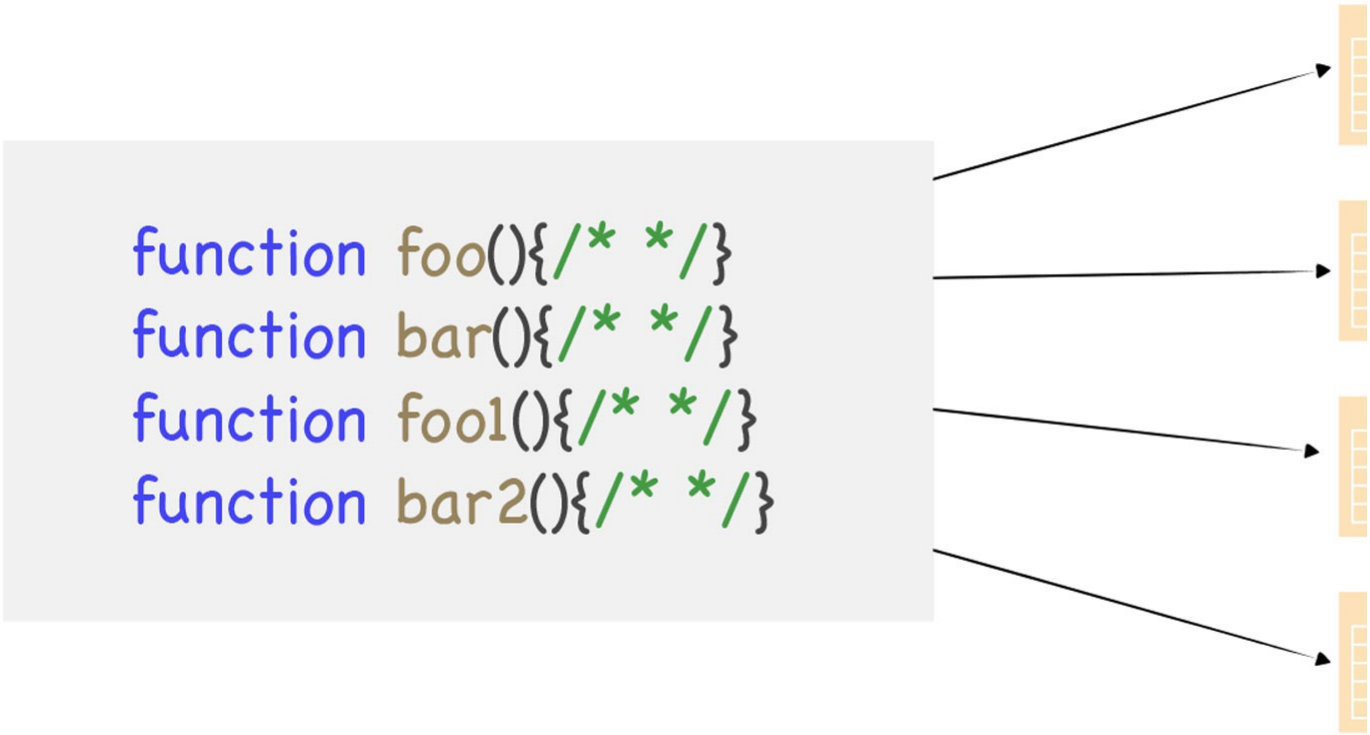
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

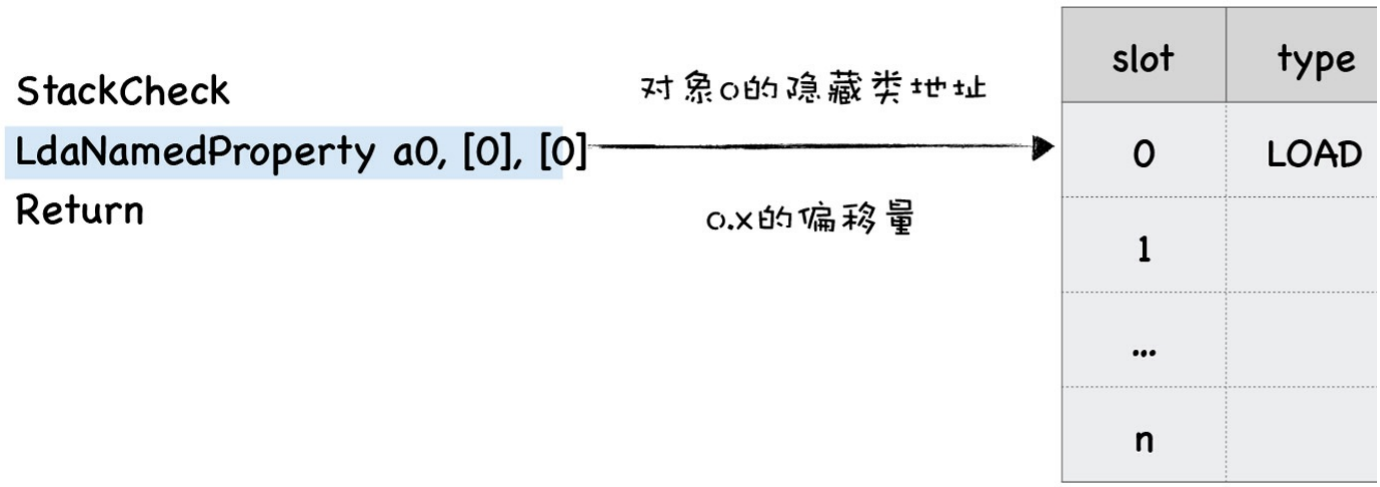
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

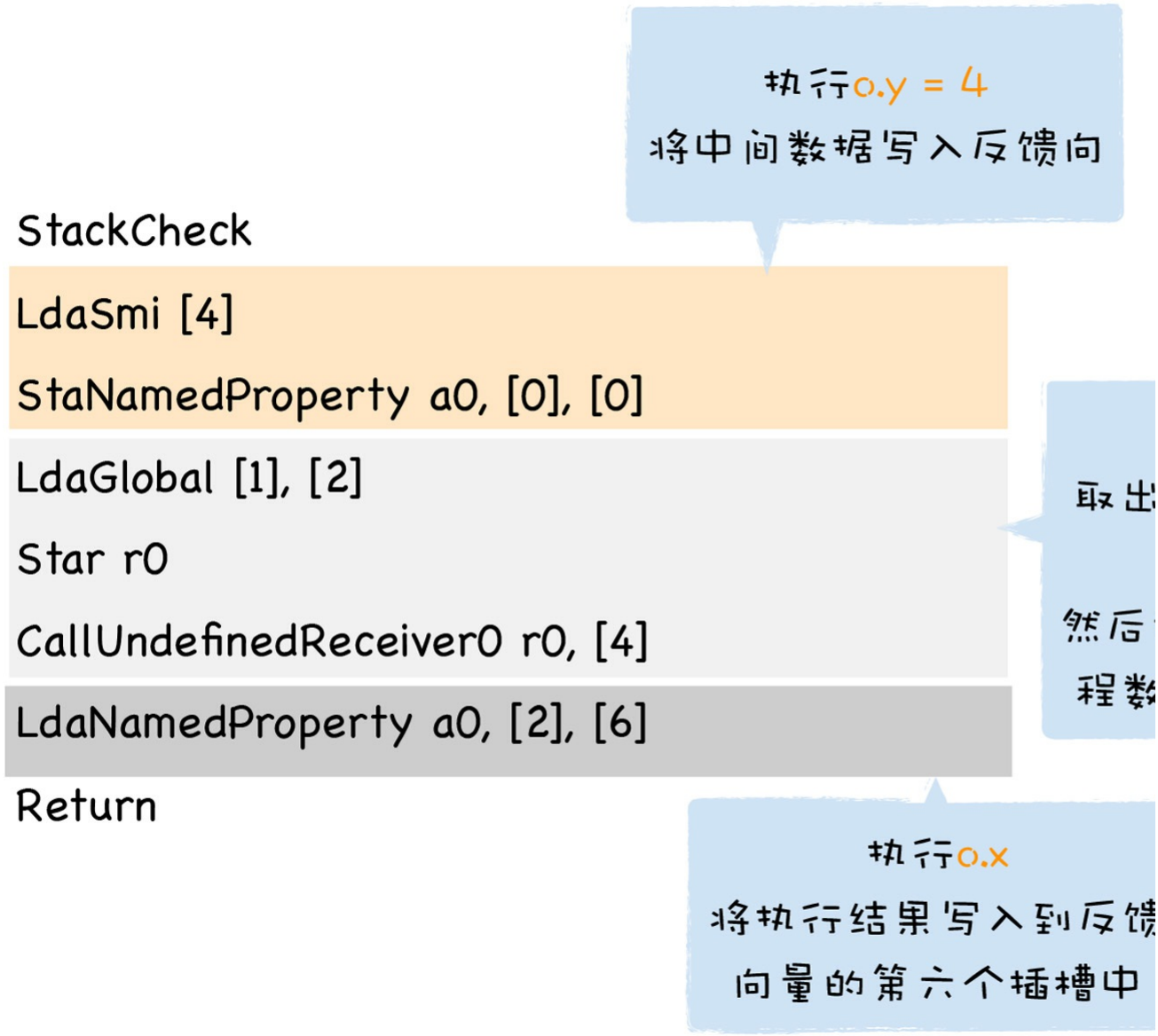
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：

尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

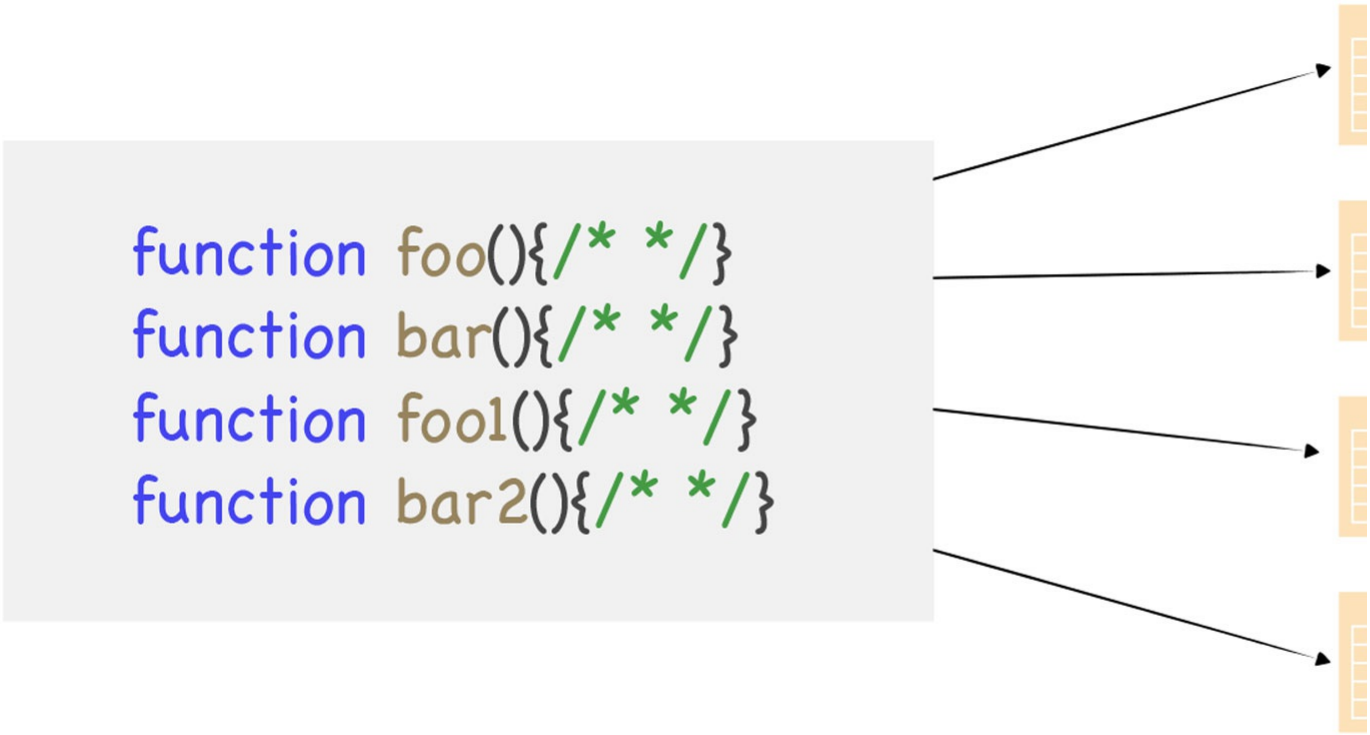
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

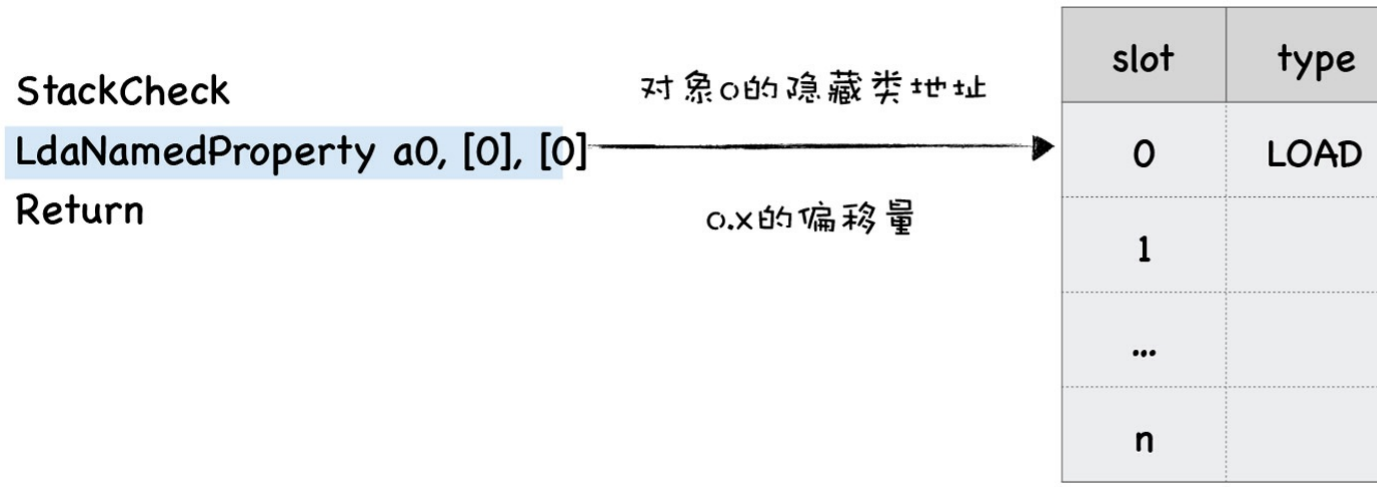
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

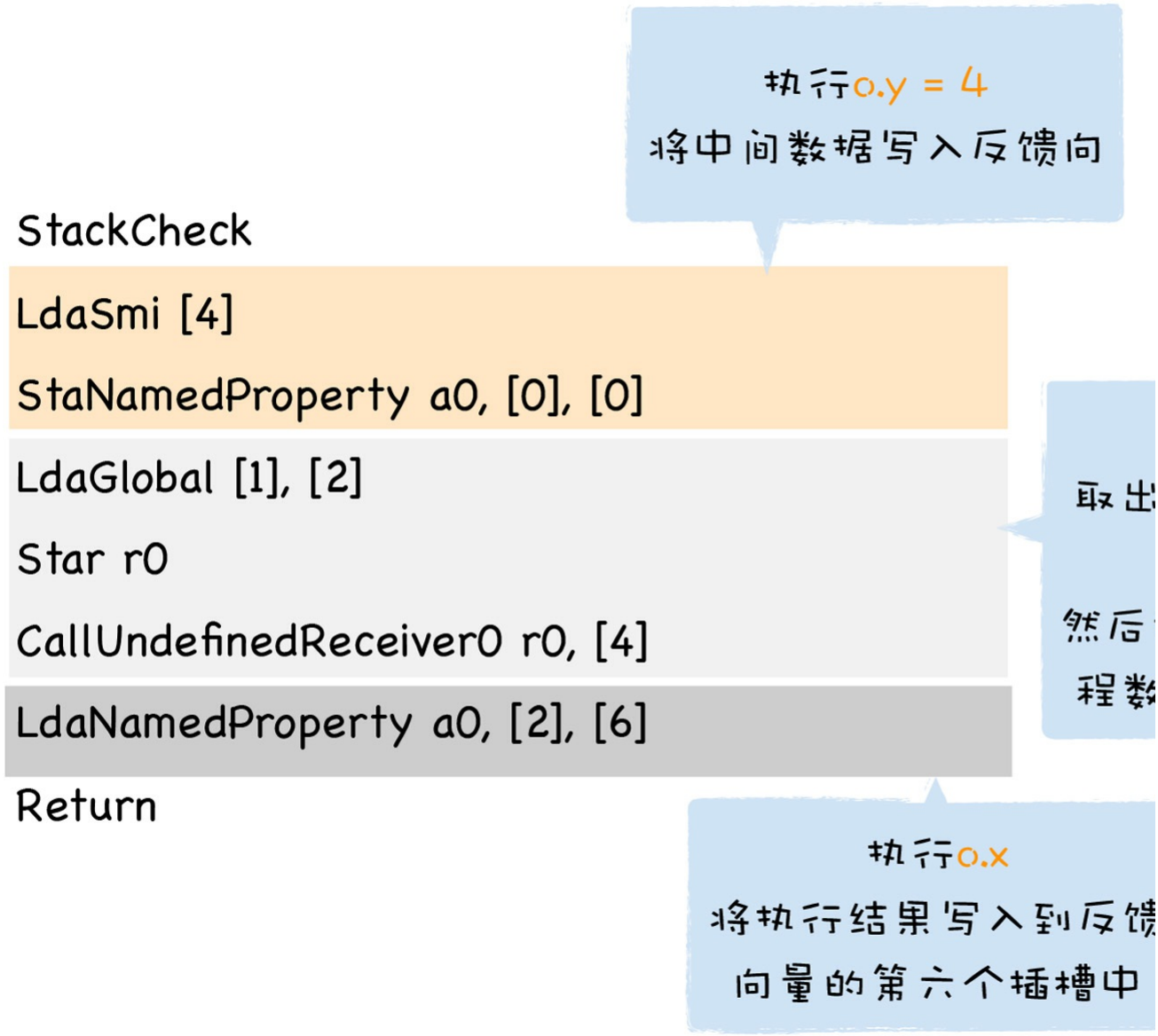
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：

尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

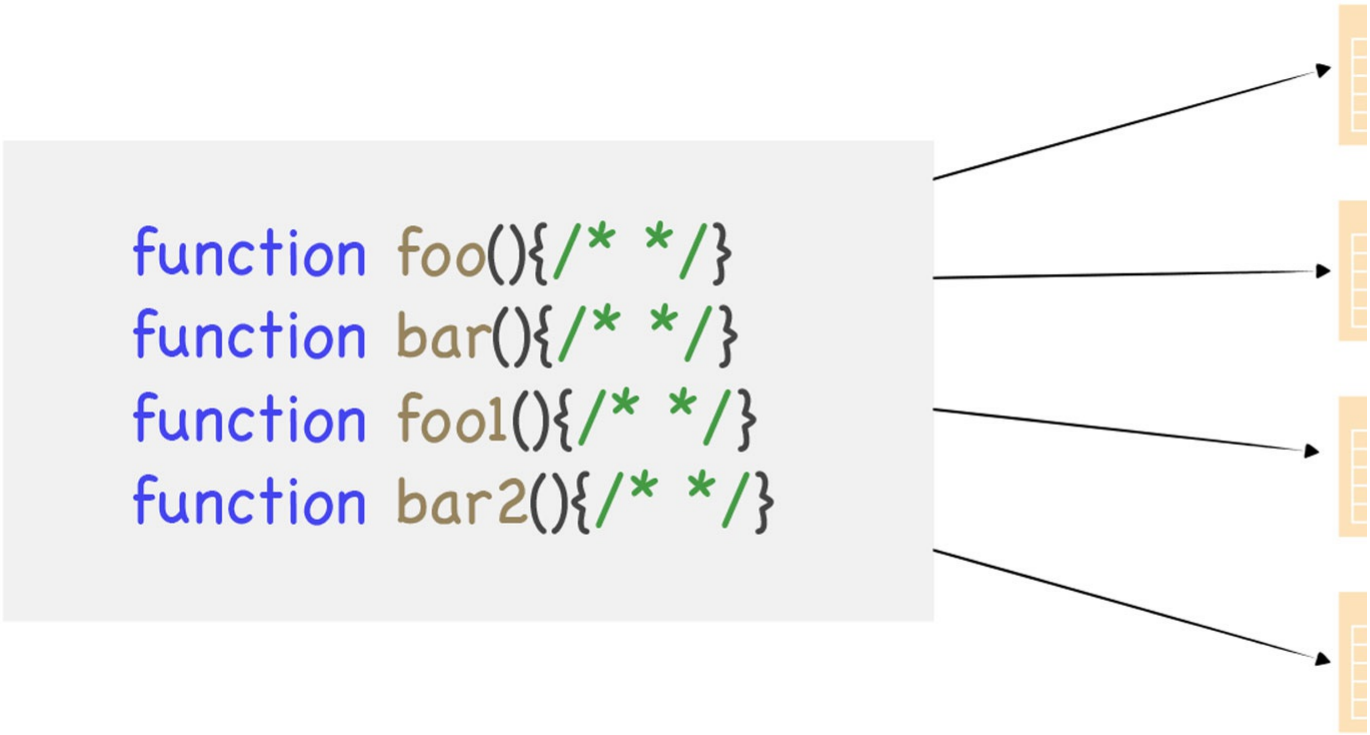
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

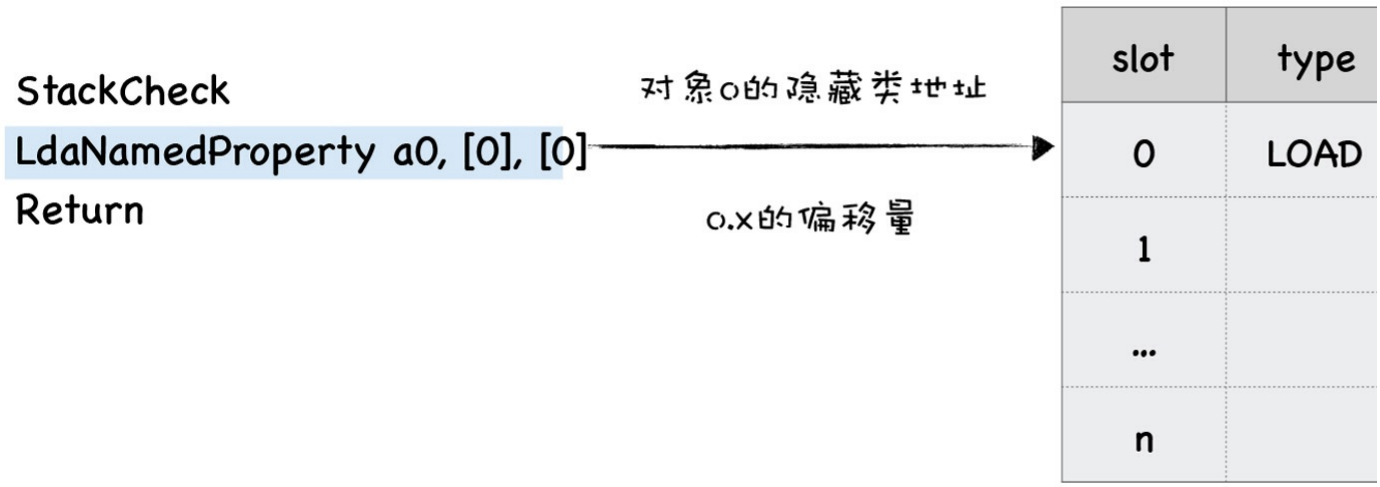
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

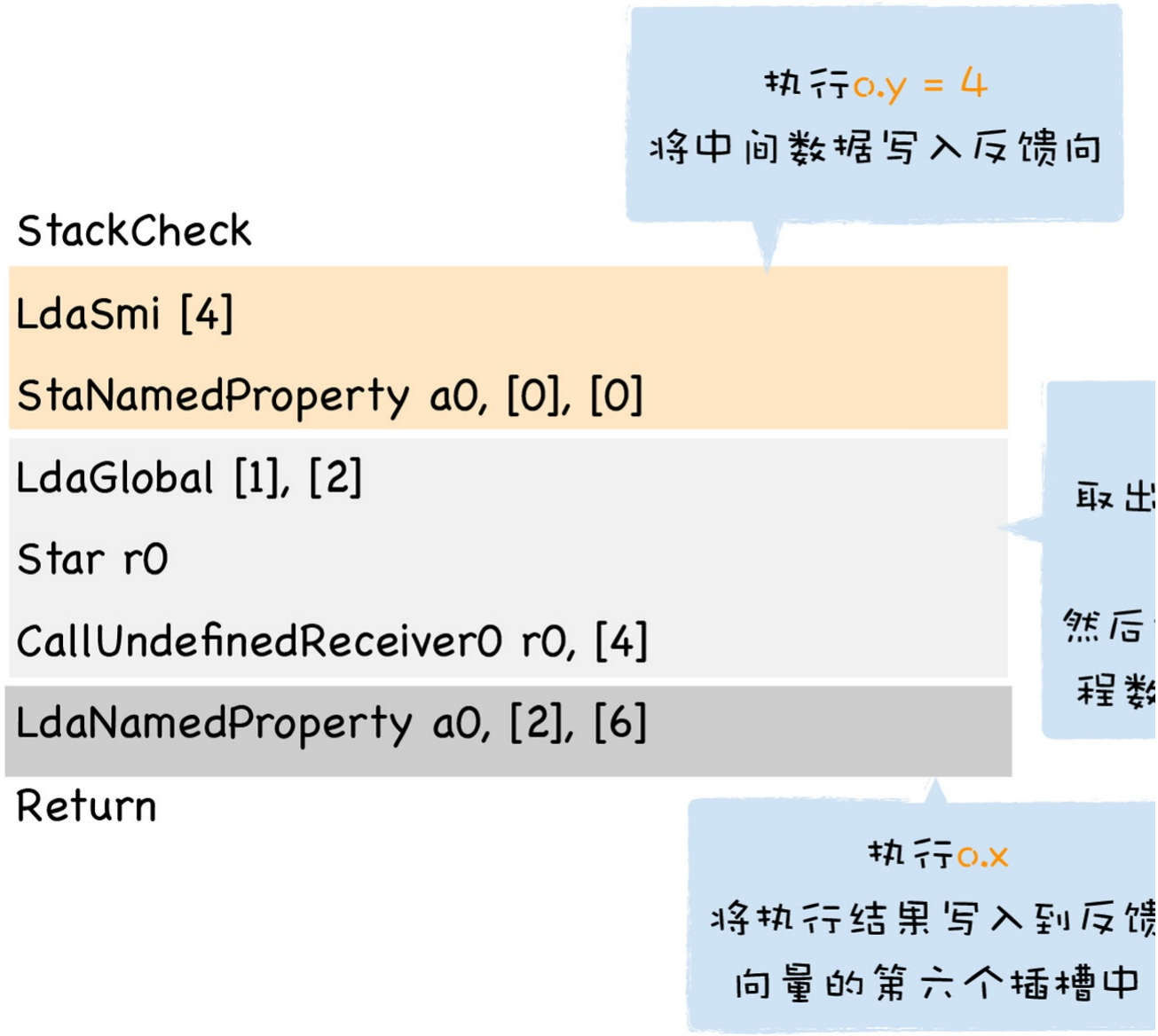
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

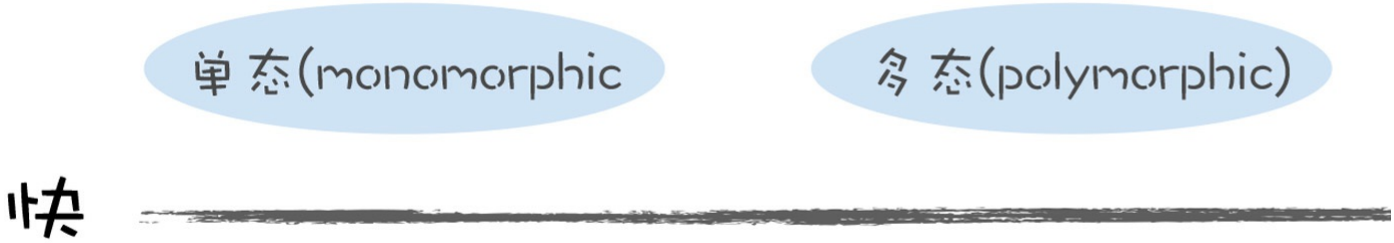
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

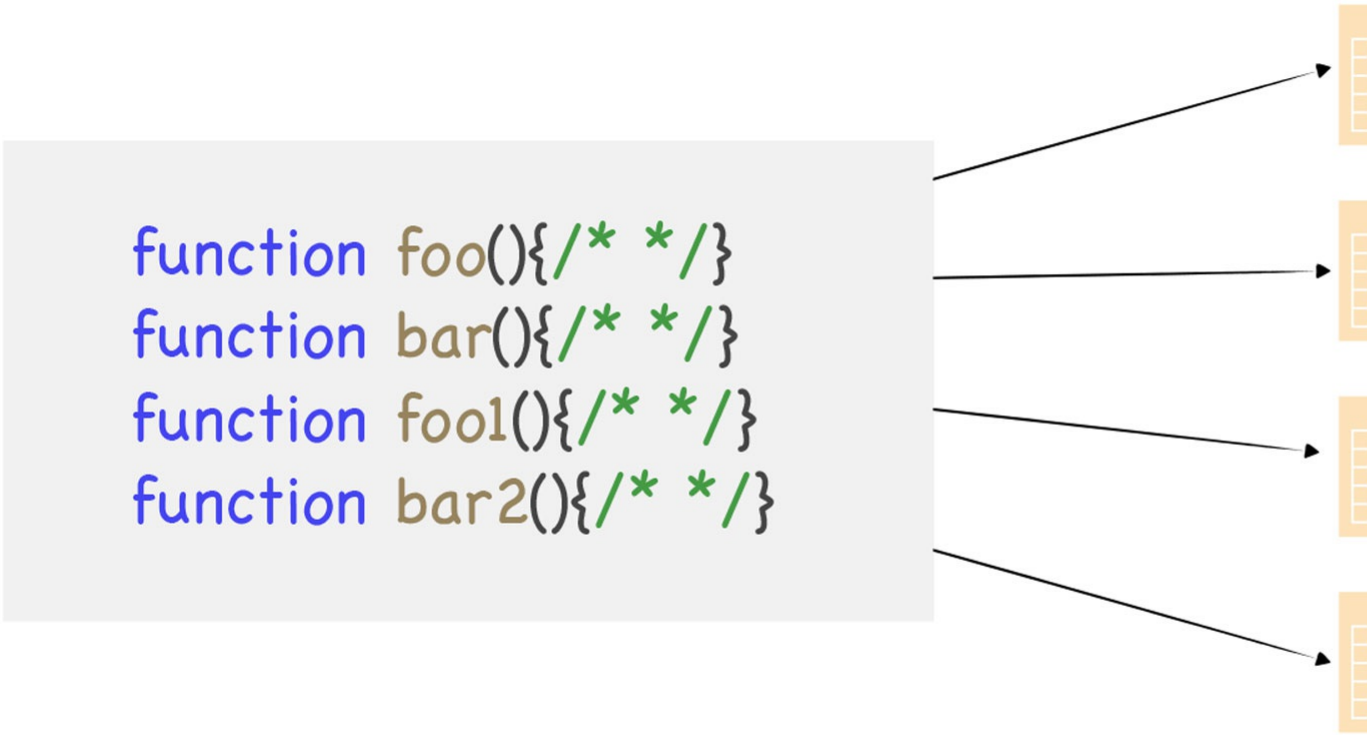
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

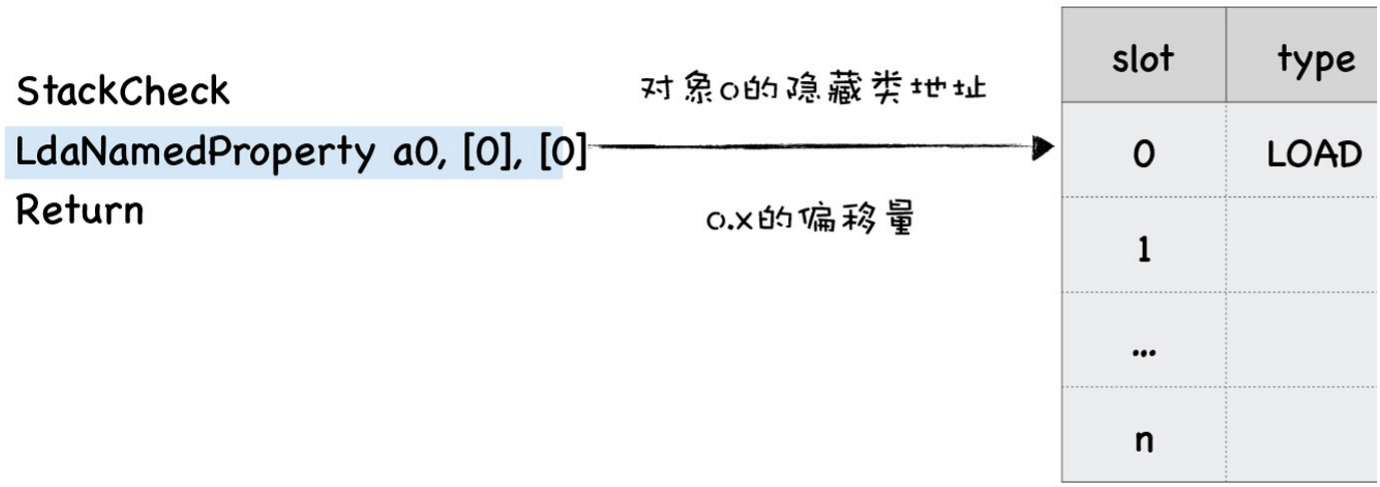
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

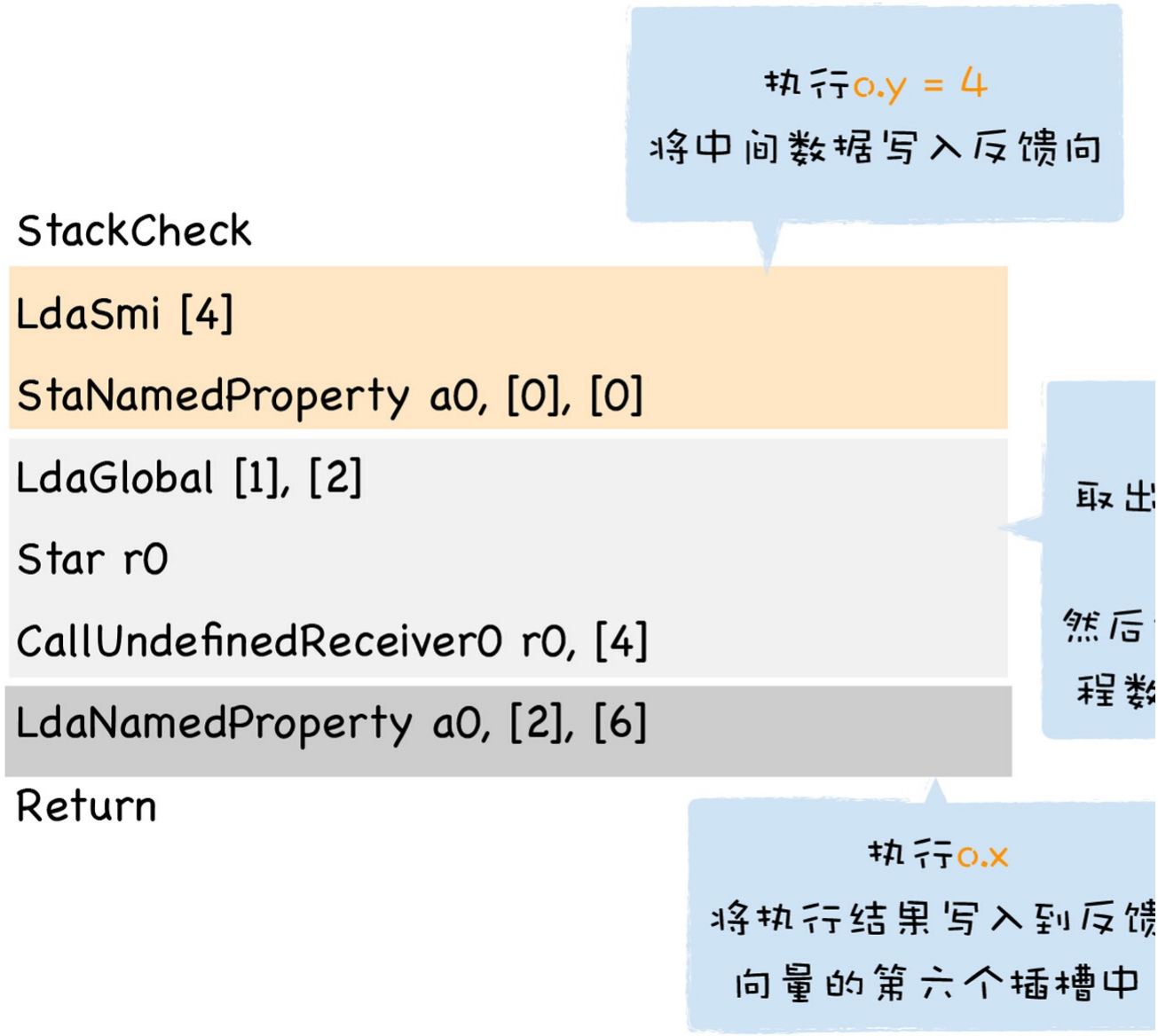
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

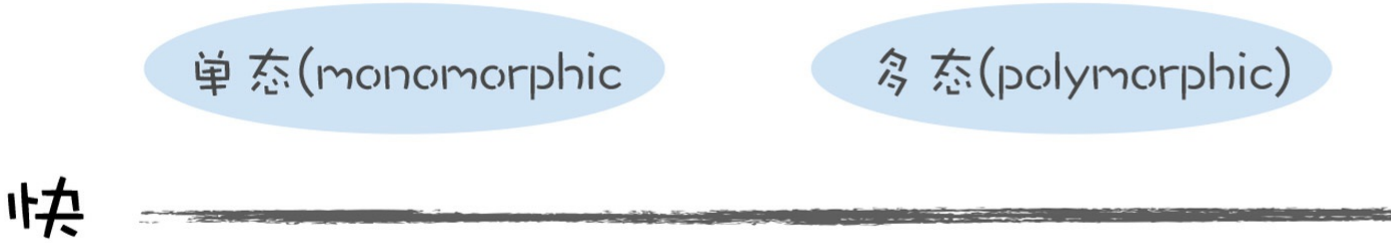
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

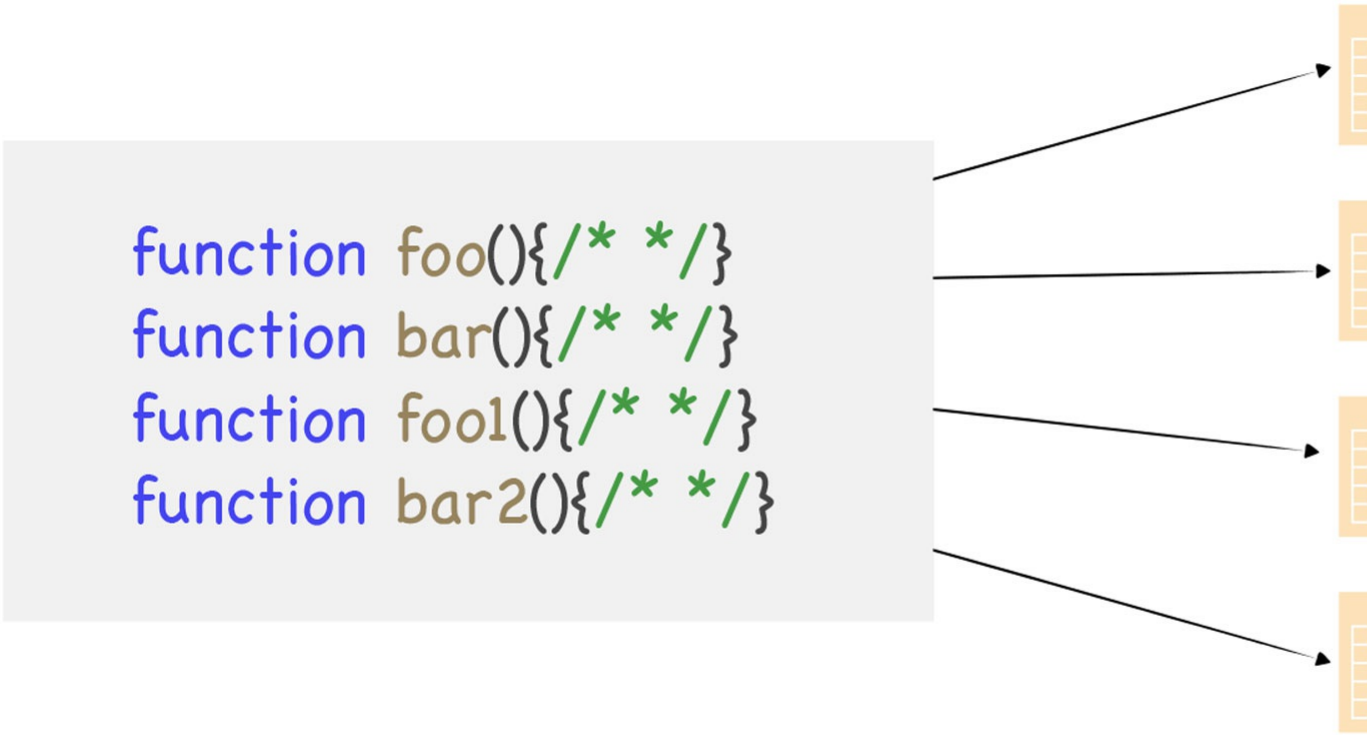
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

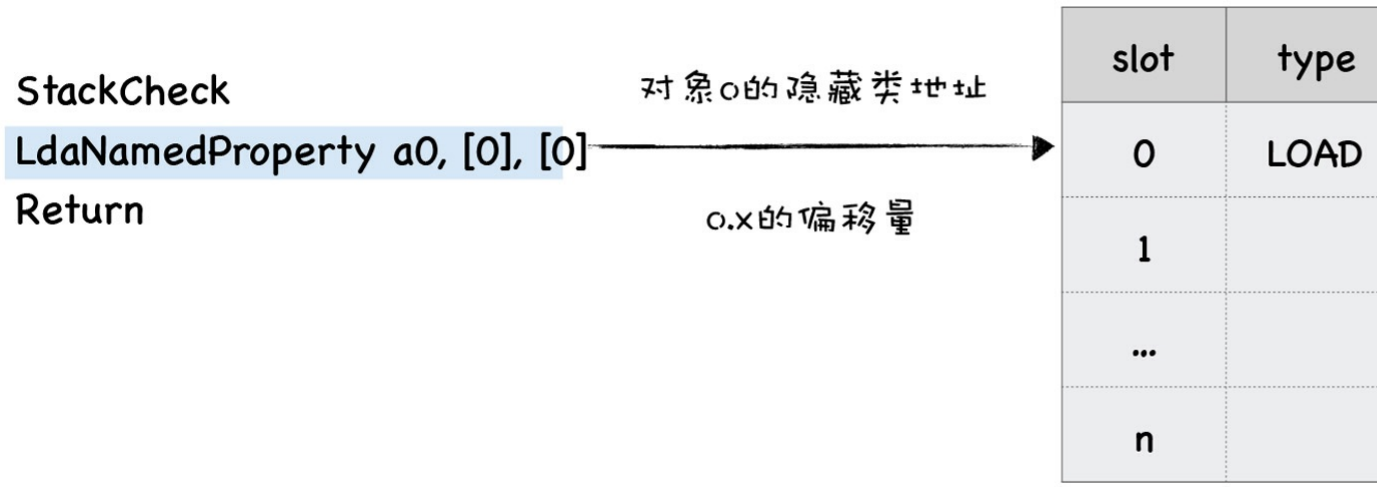
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

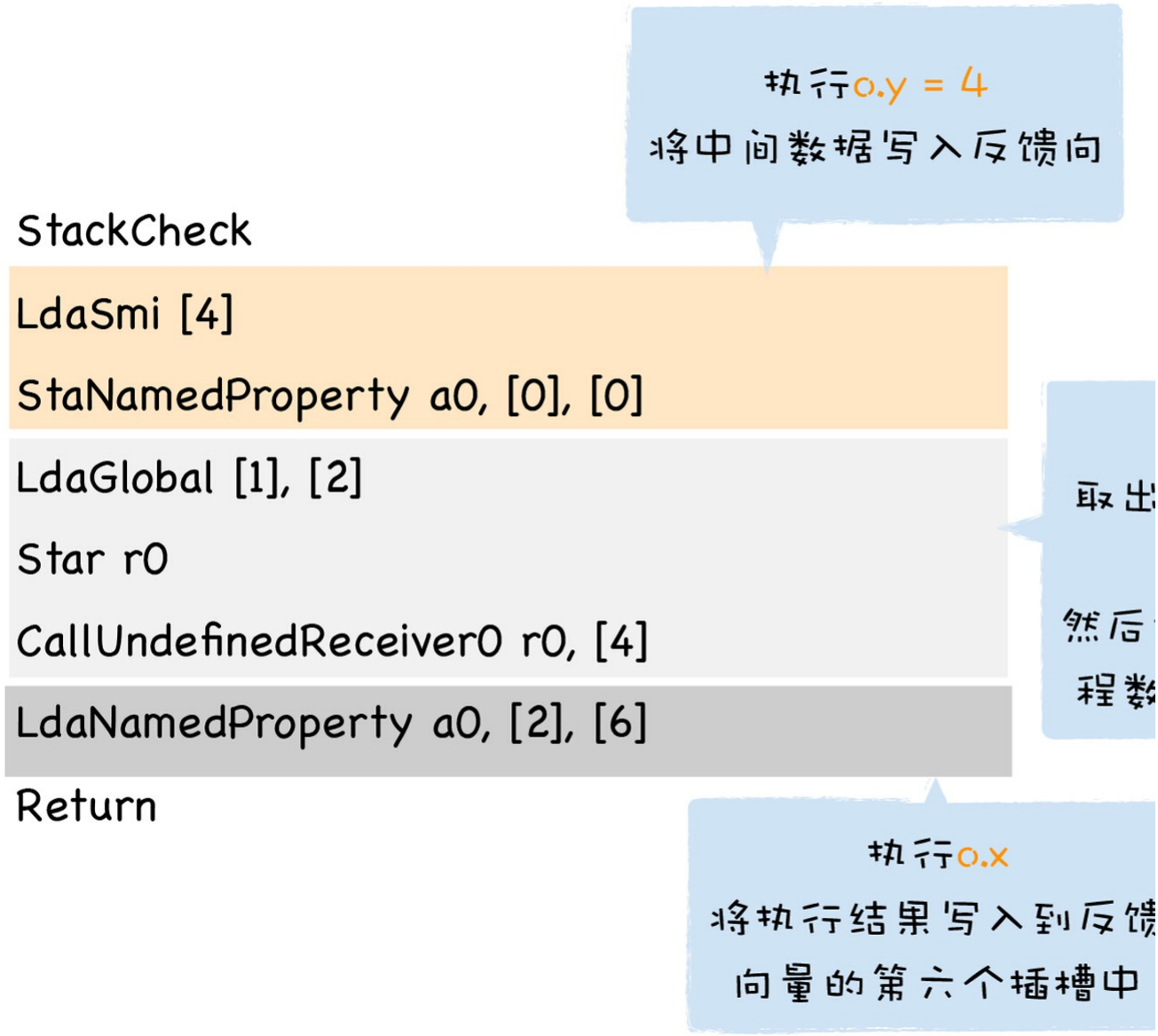
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：

尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态**，所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC**。

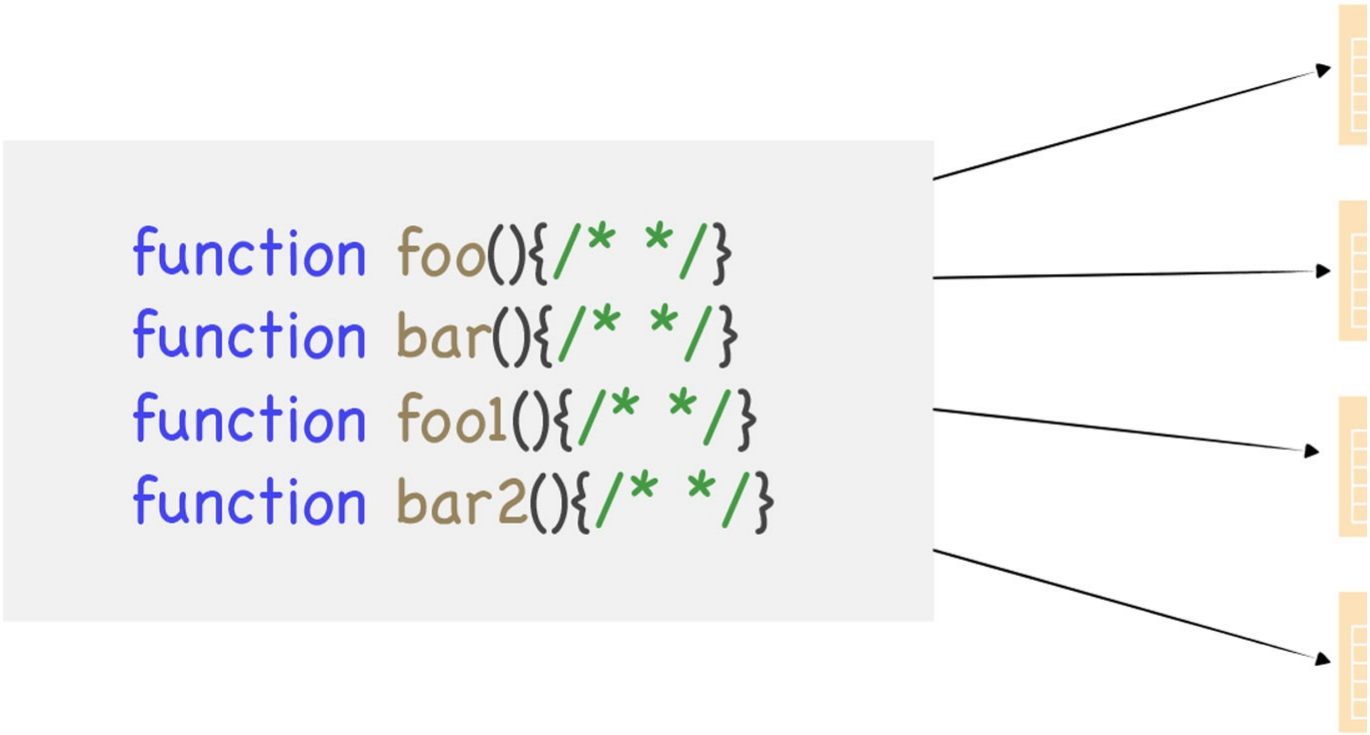
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

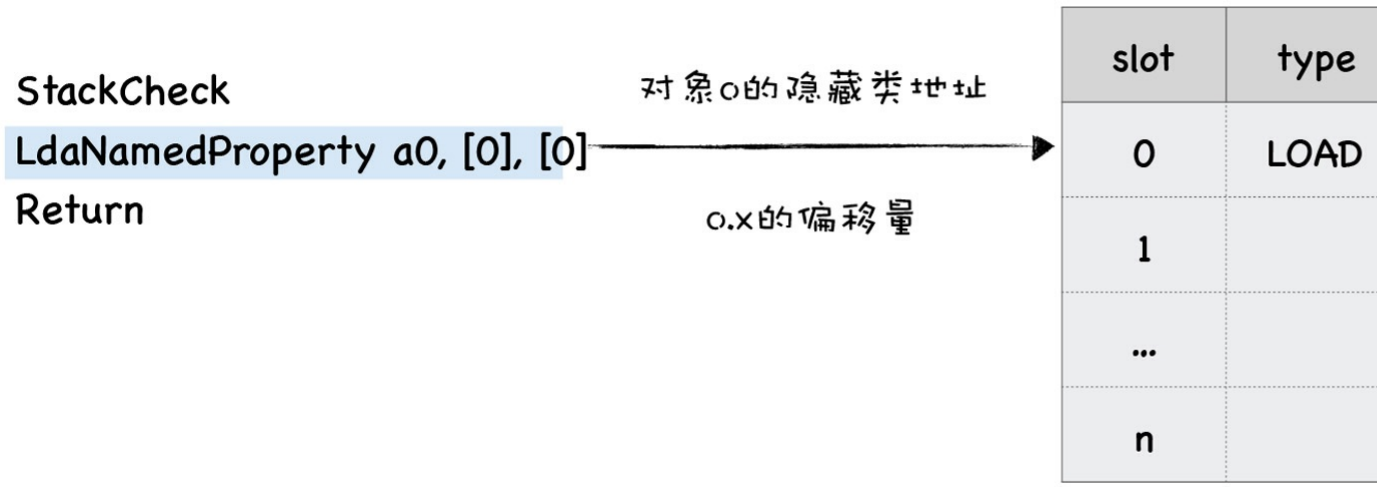
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

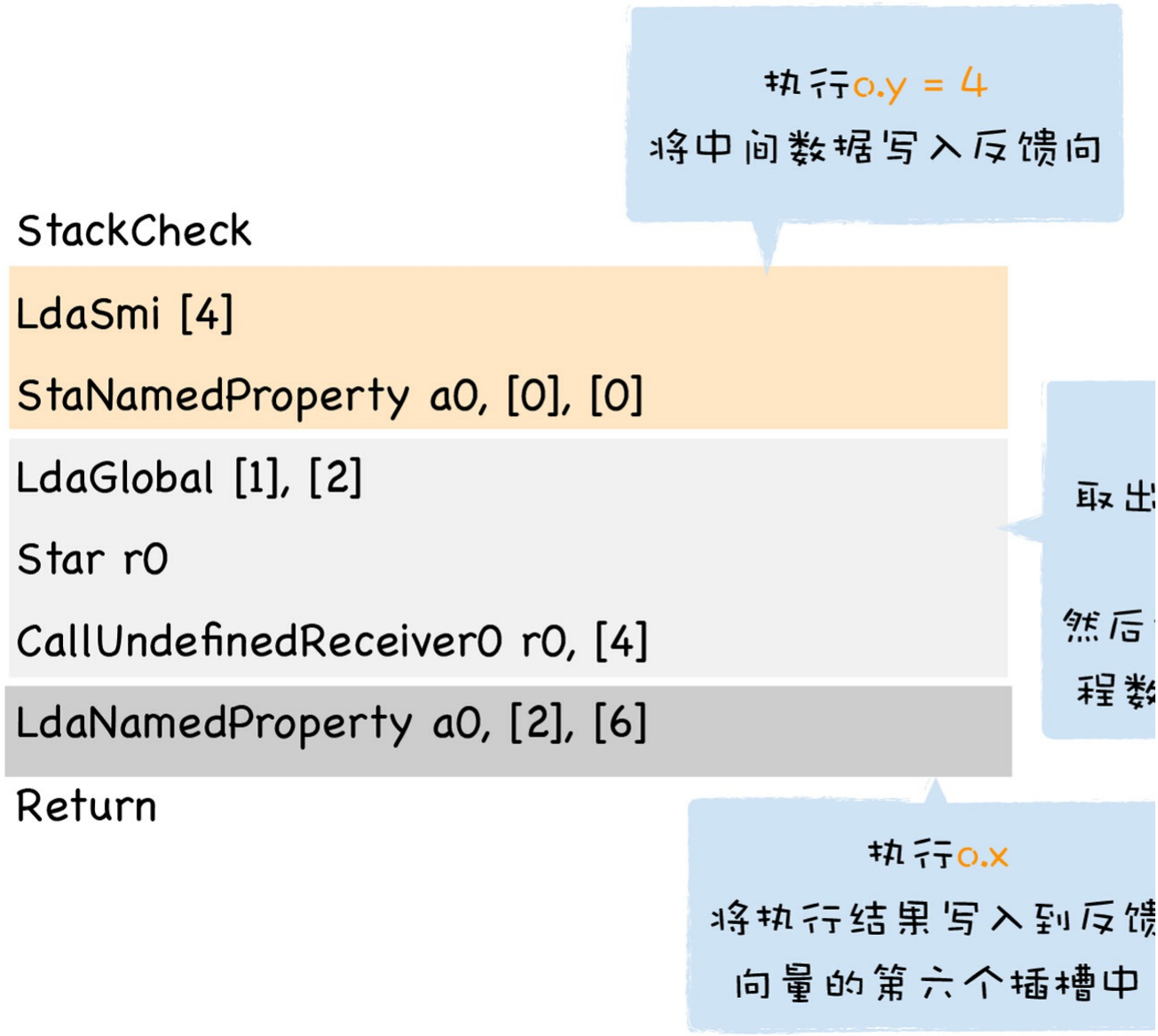
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：

尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

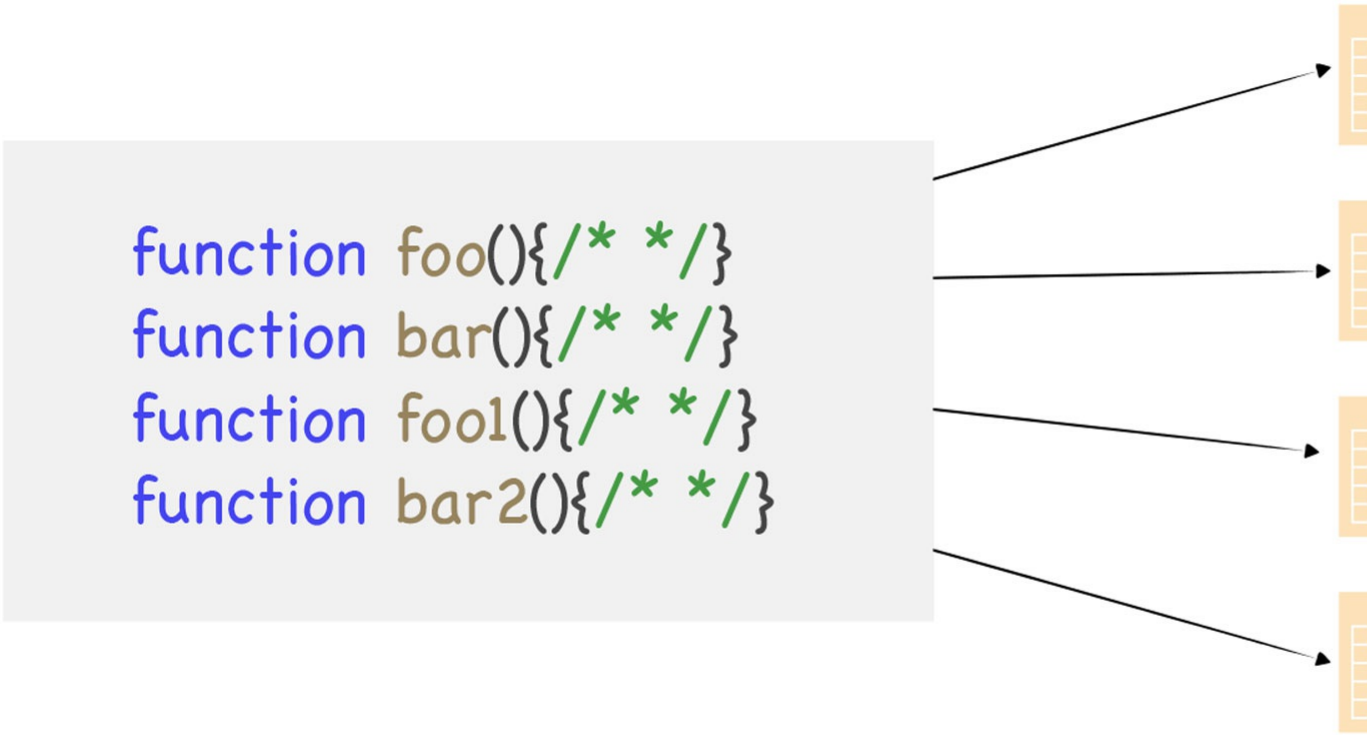
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

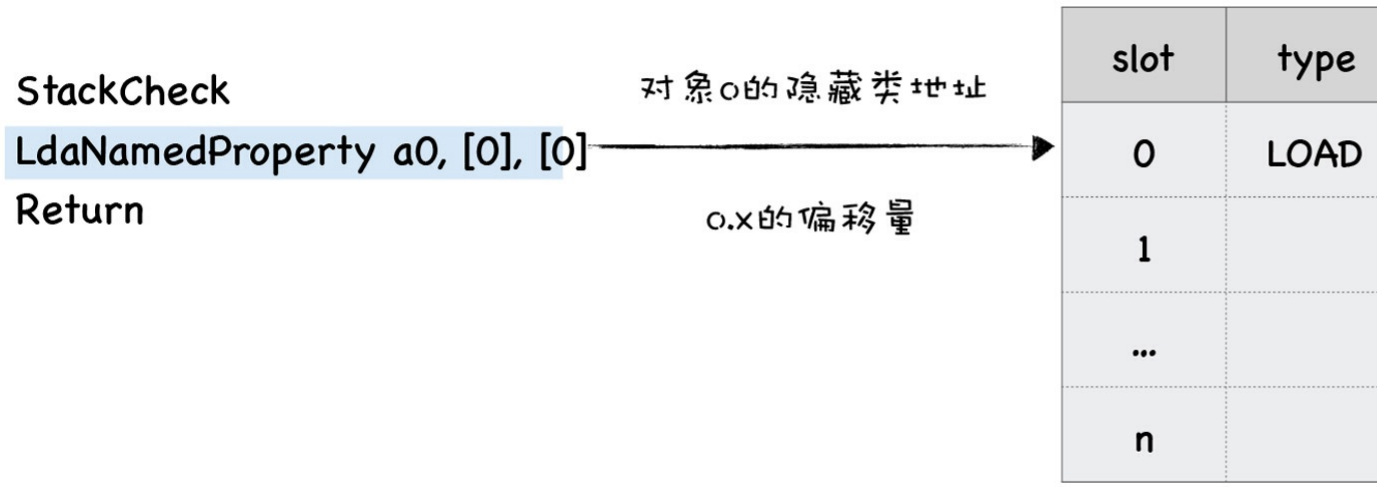
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

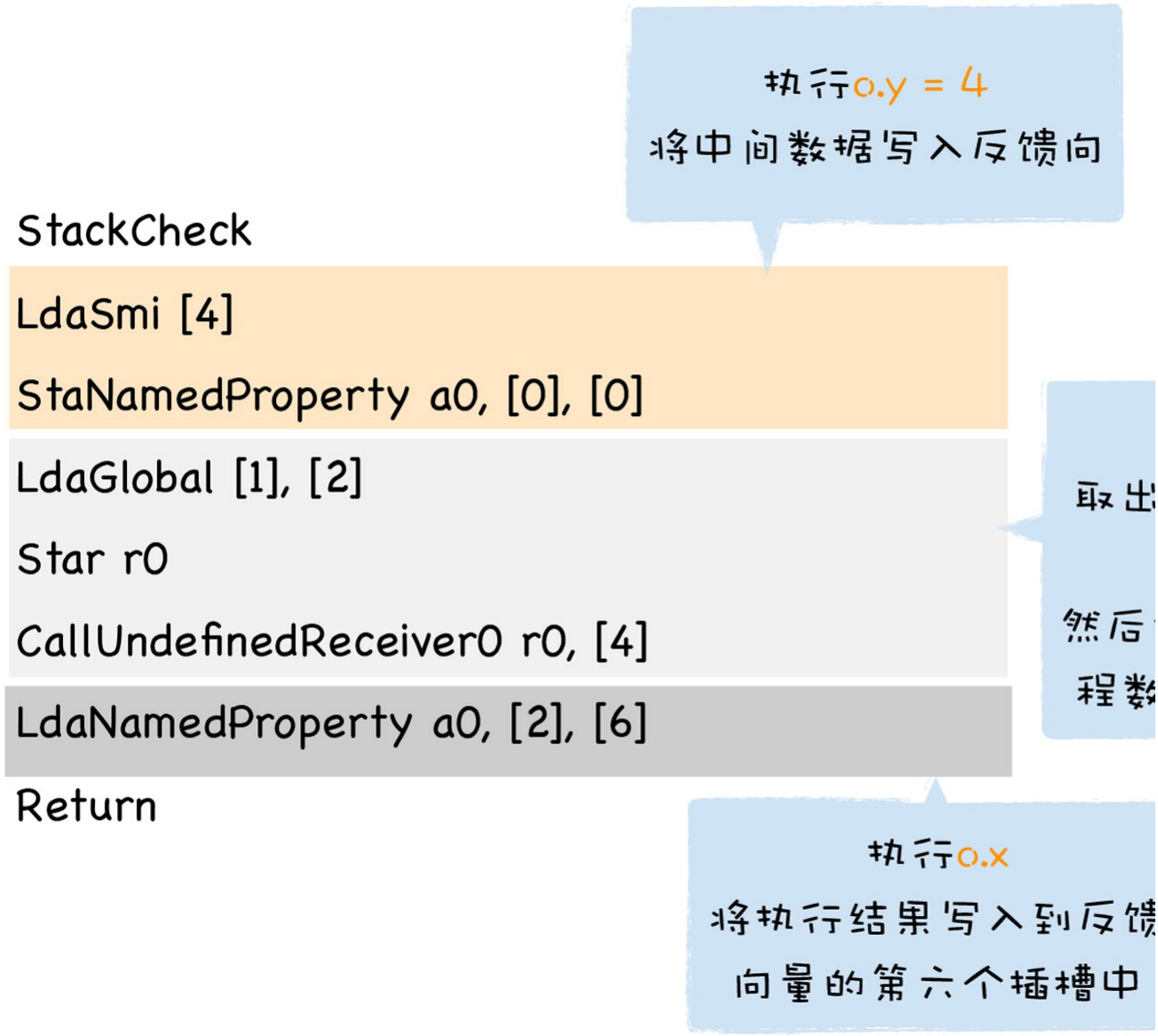
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

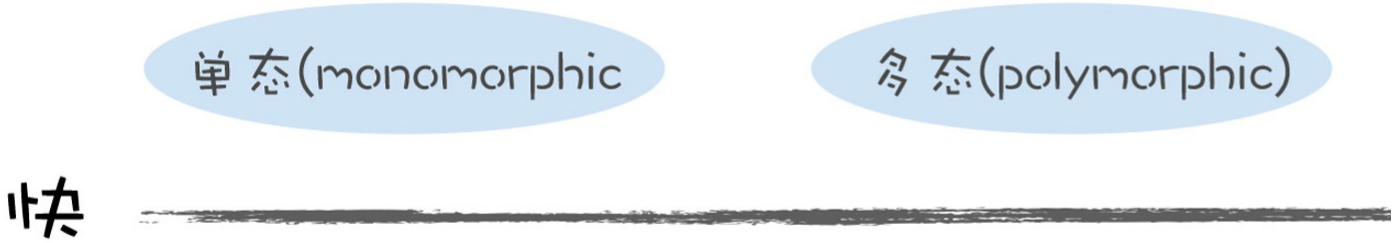
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

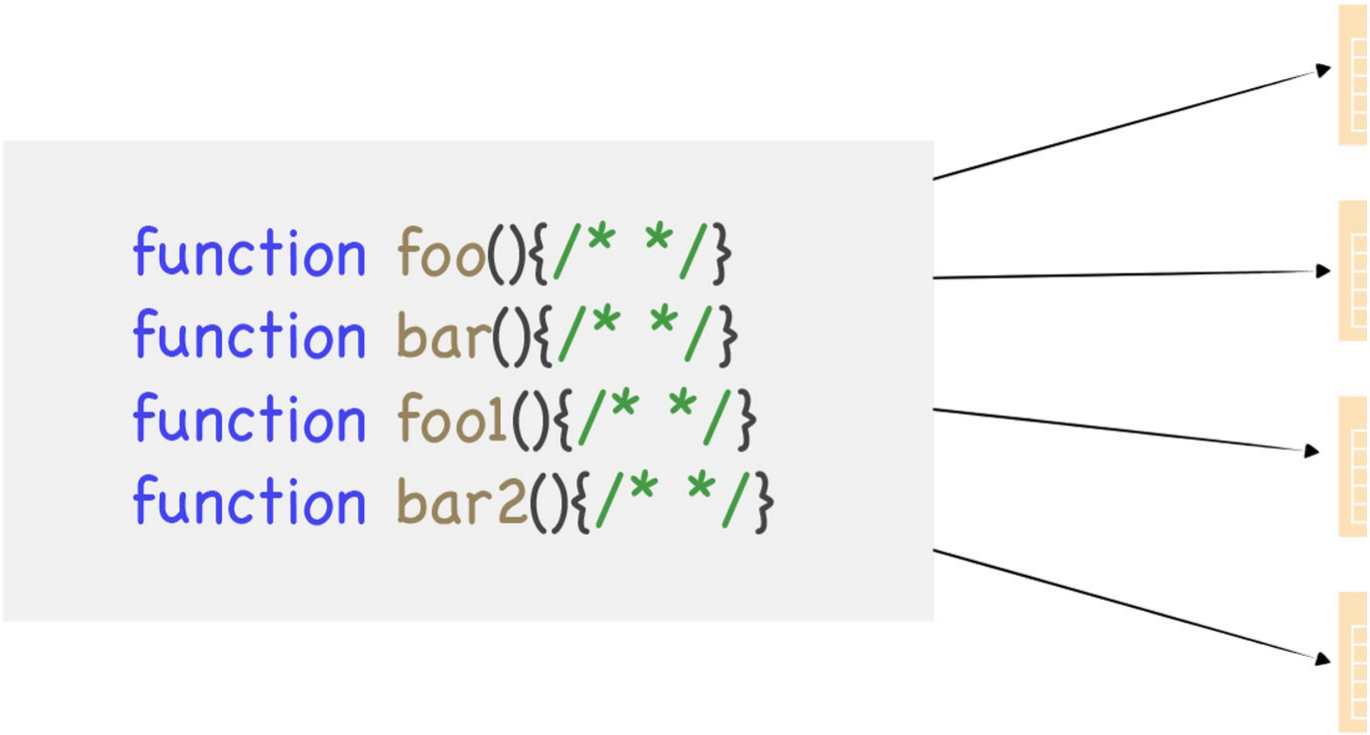
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

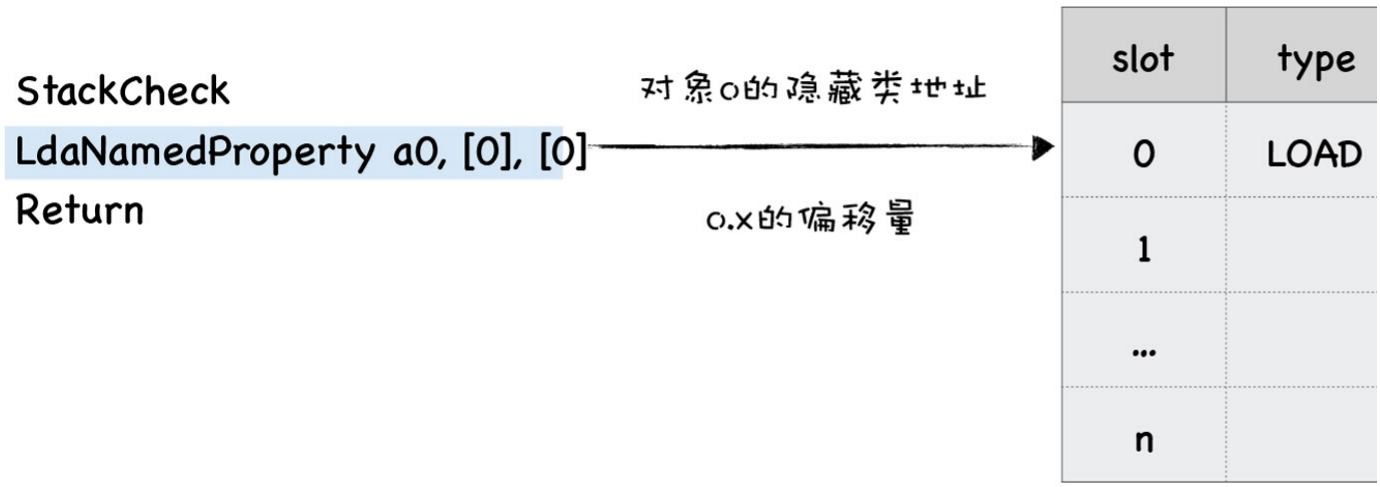
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

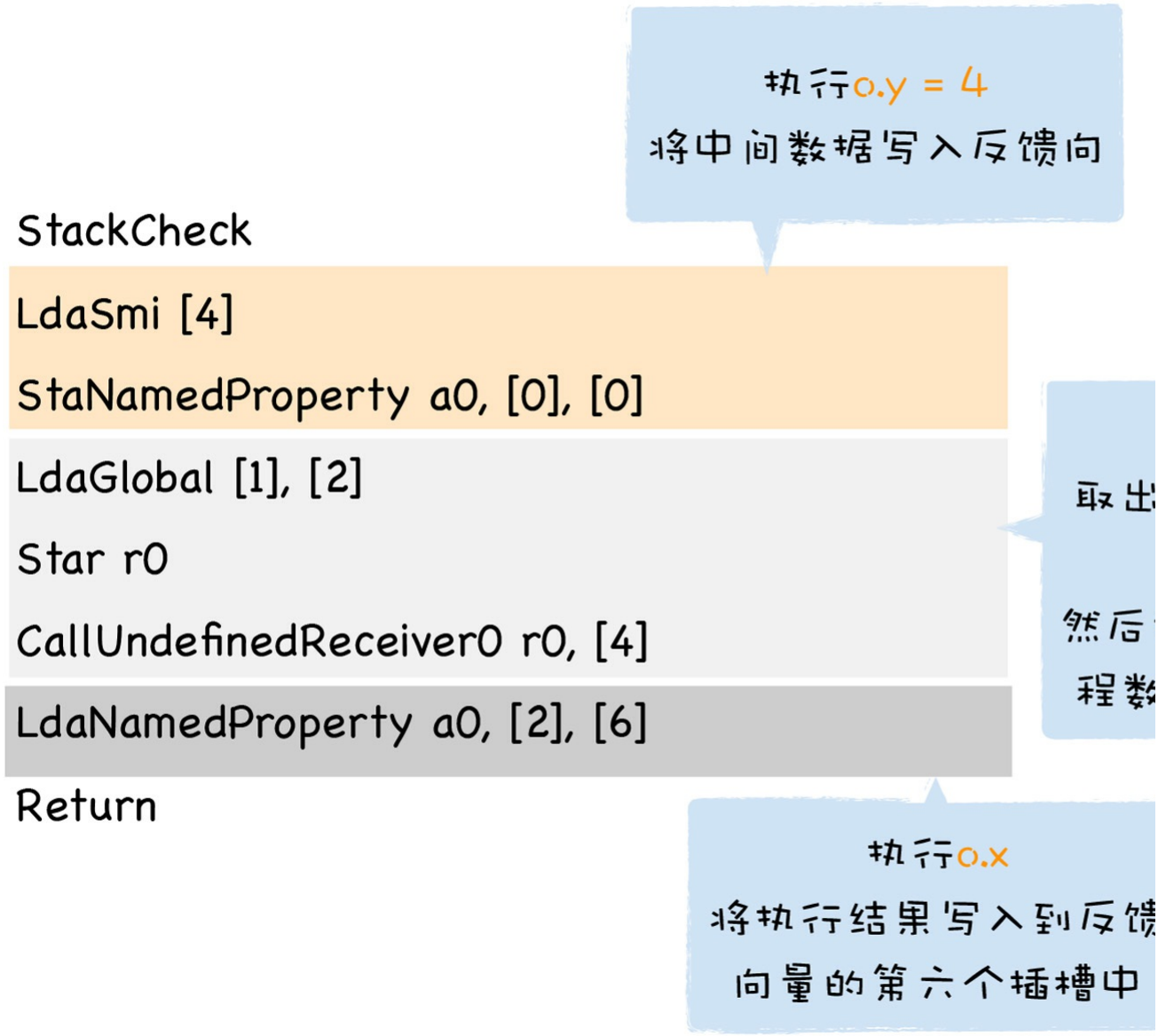
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

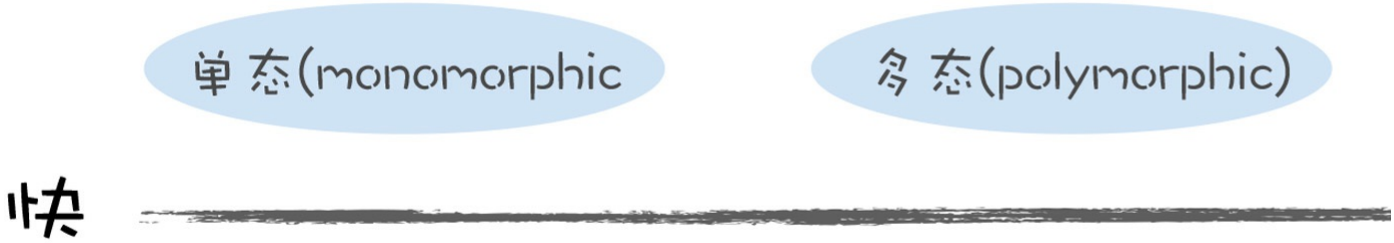
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

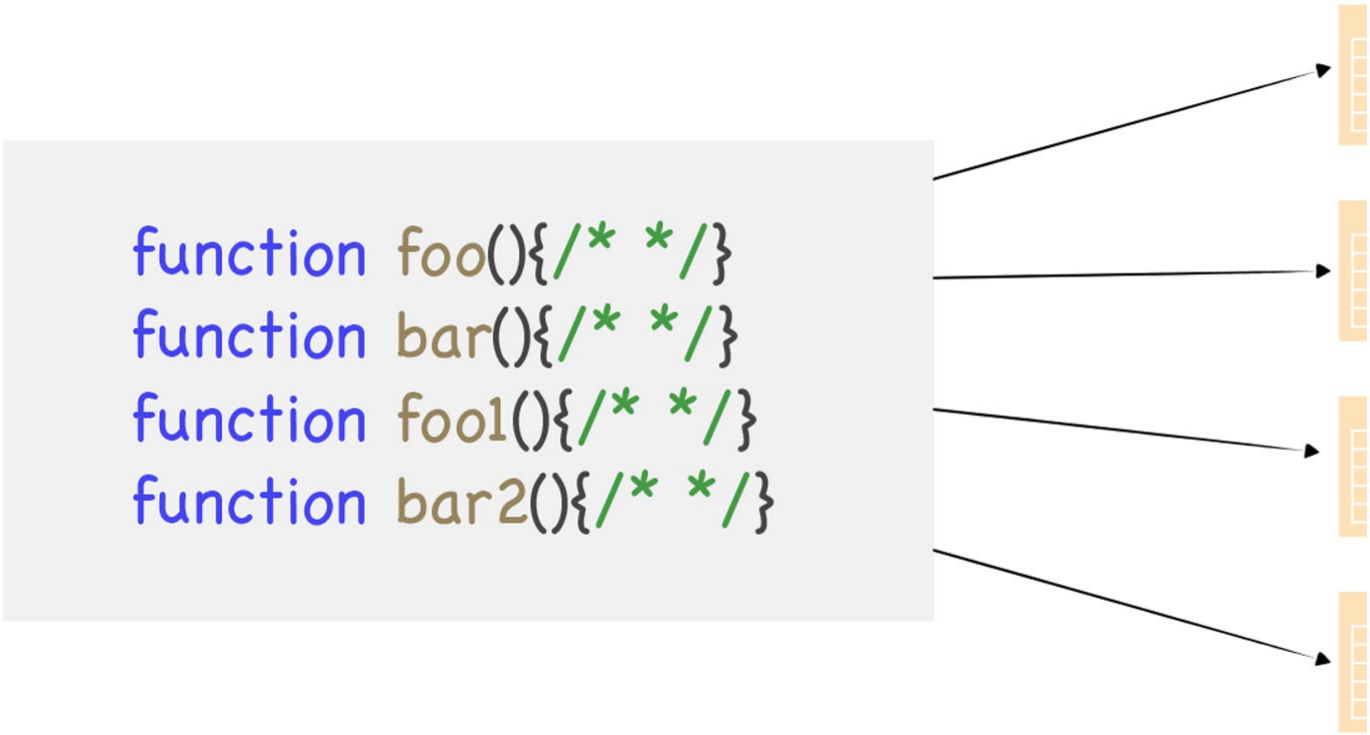
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

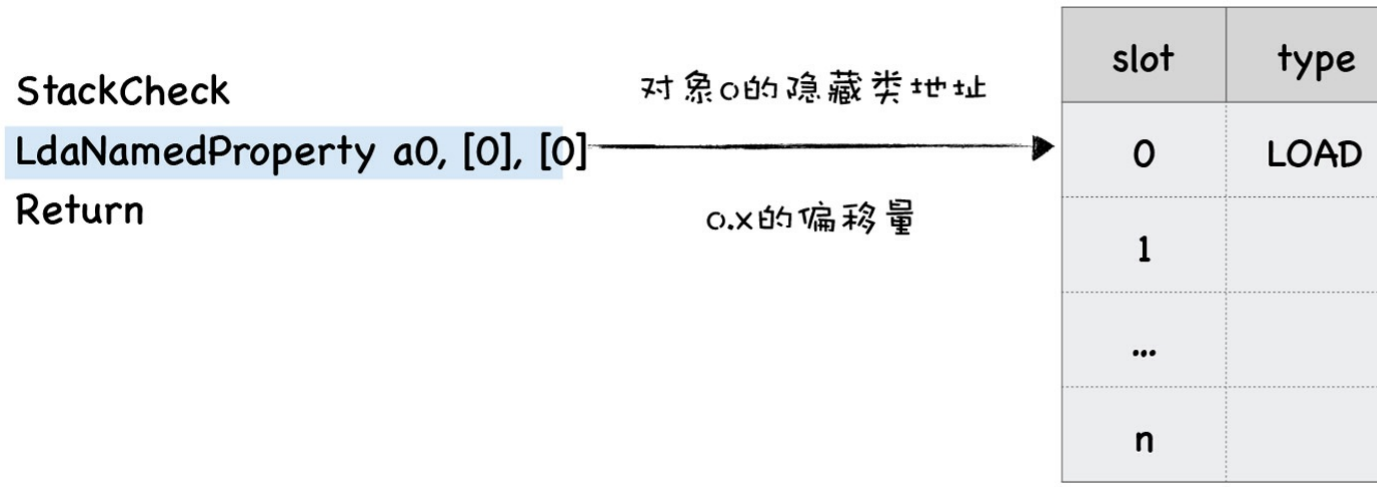
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

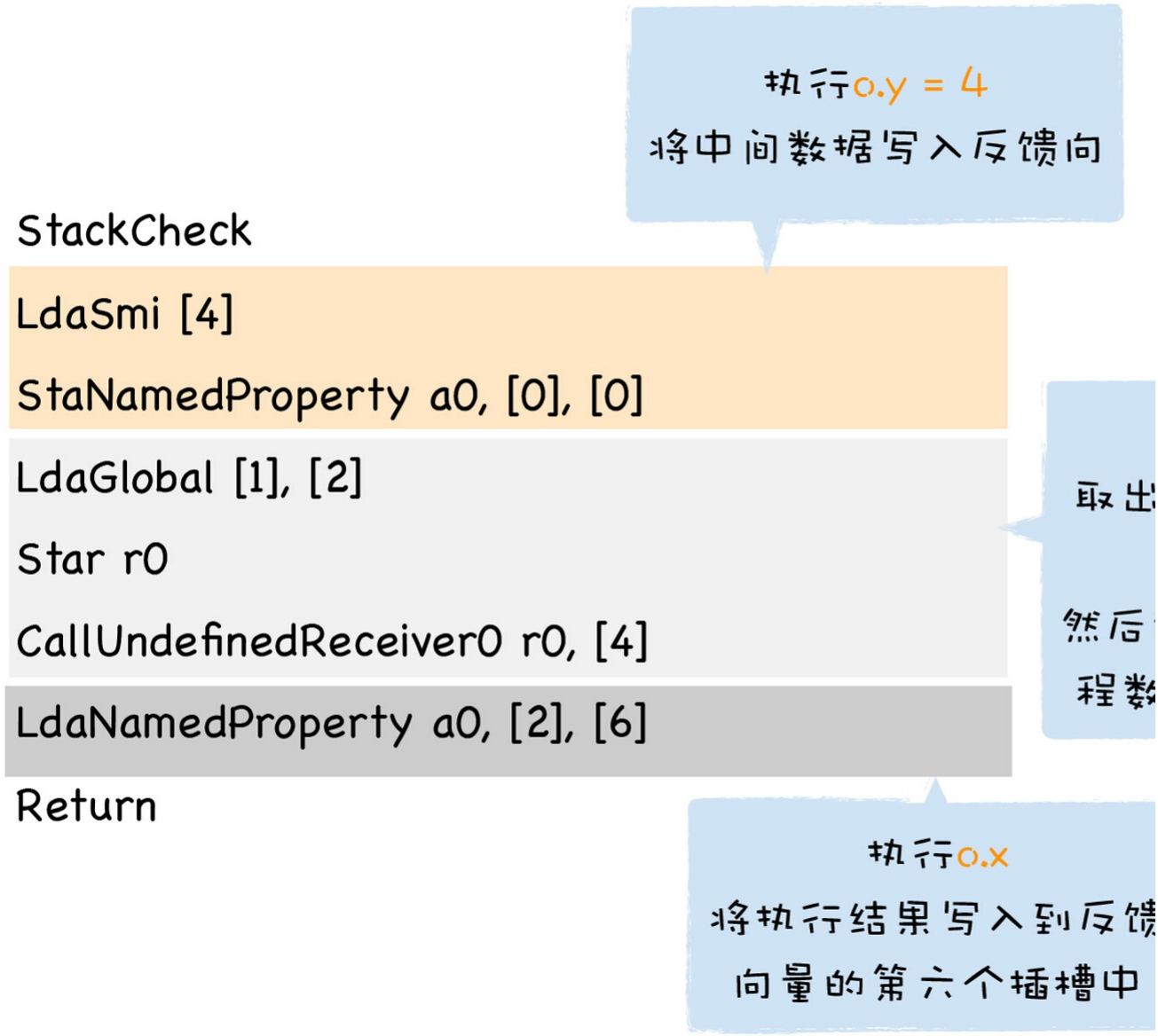
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

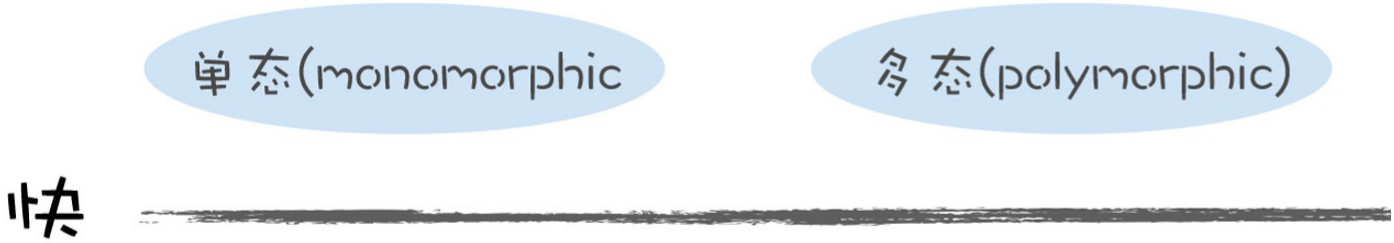
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

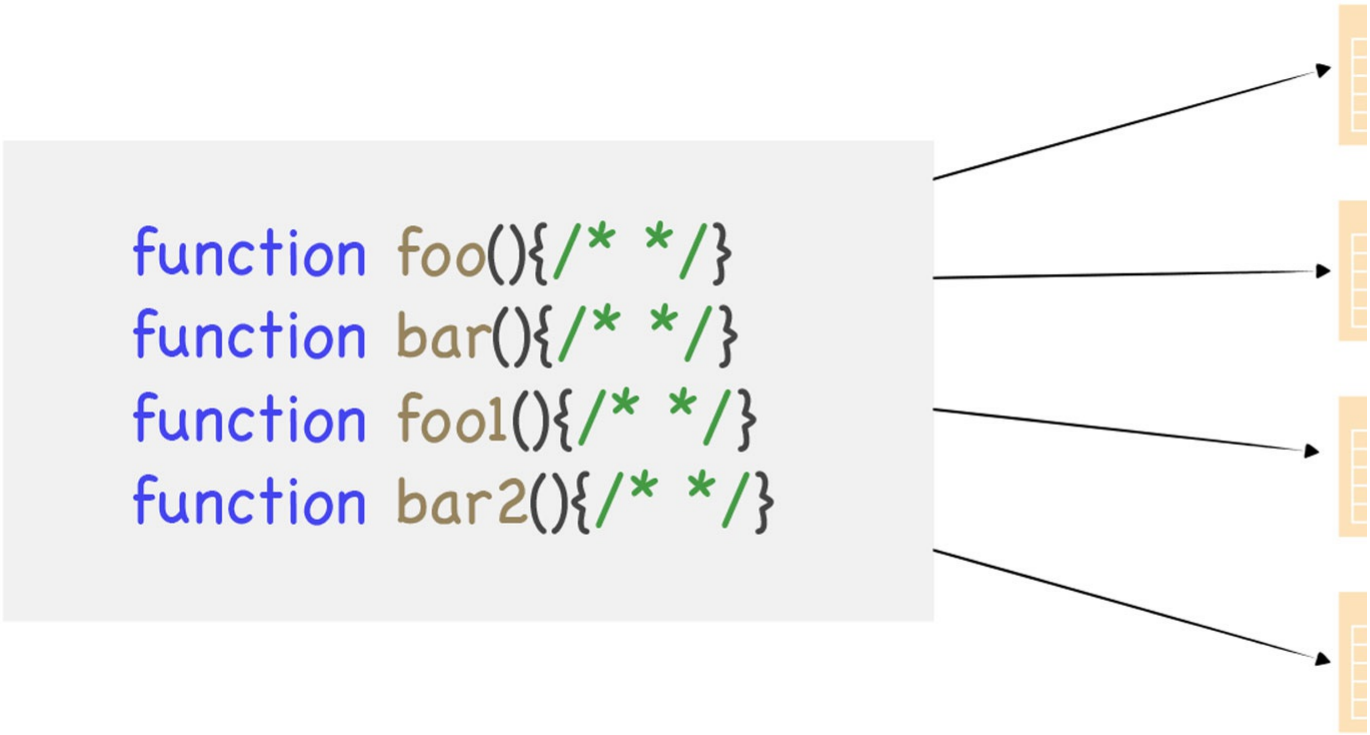
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

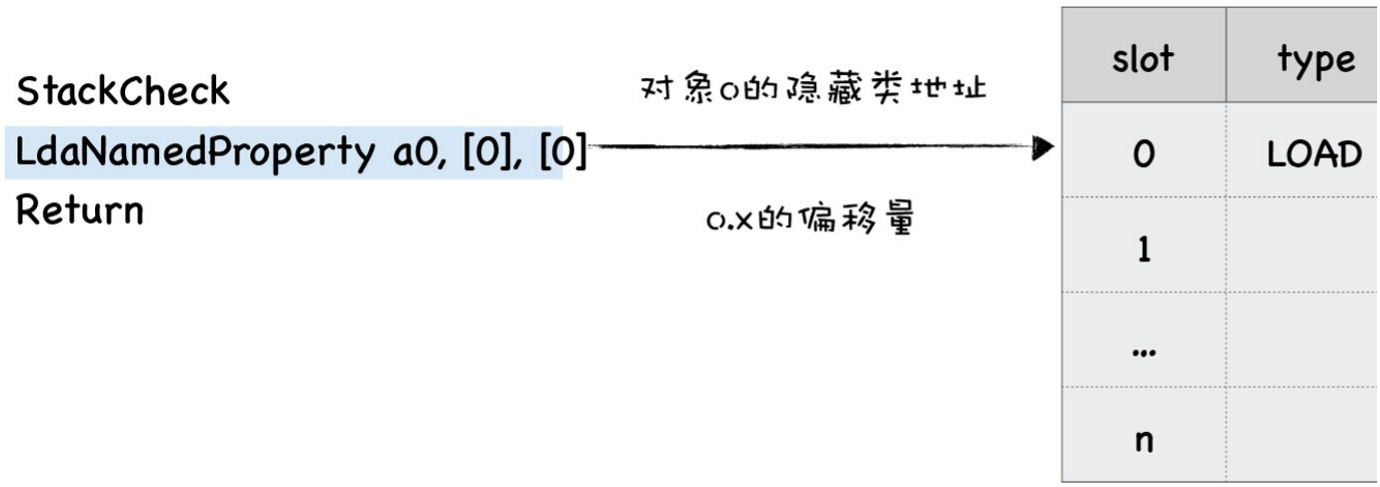
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

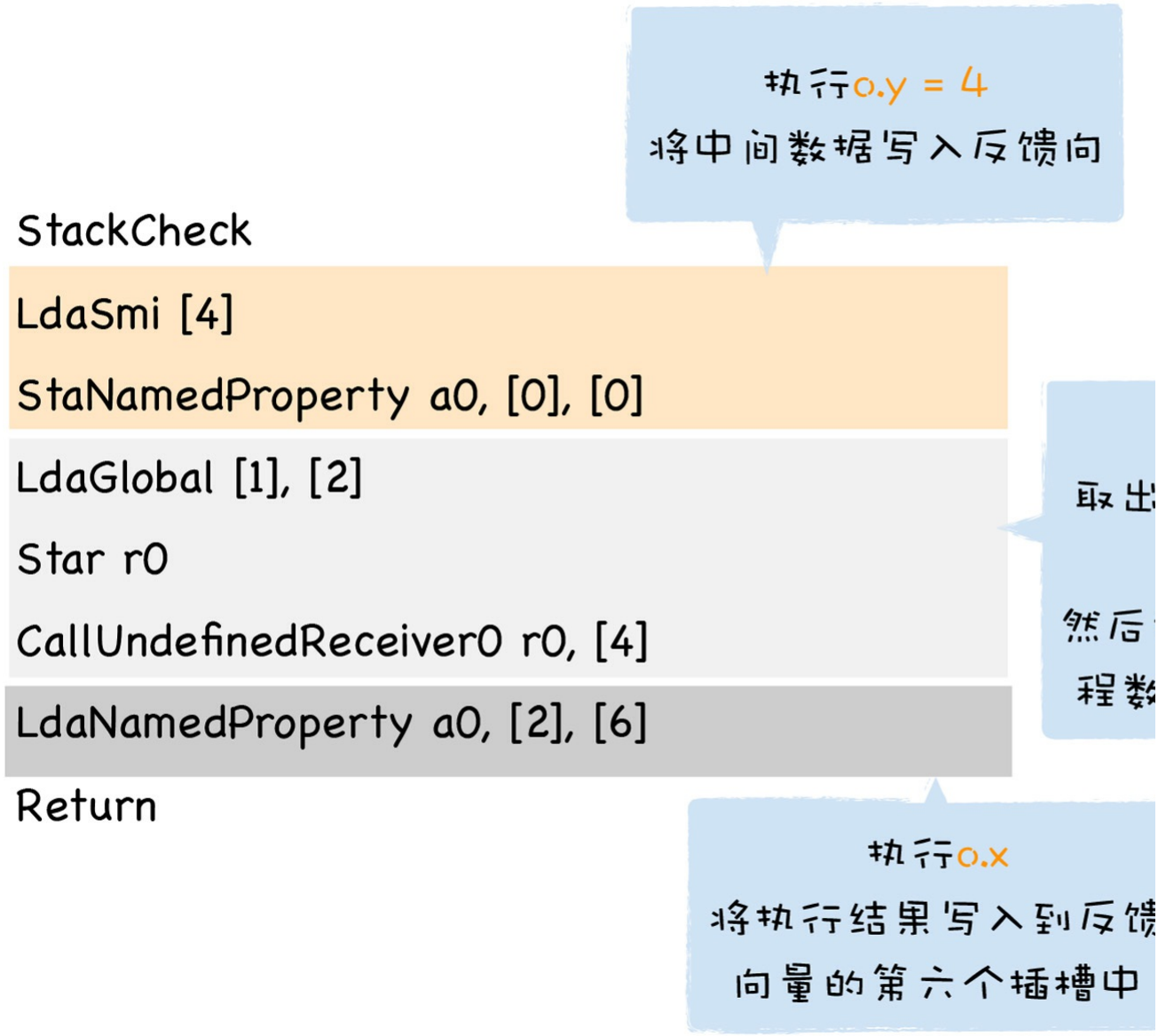
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

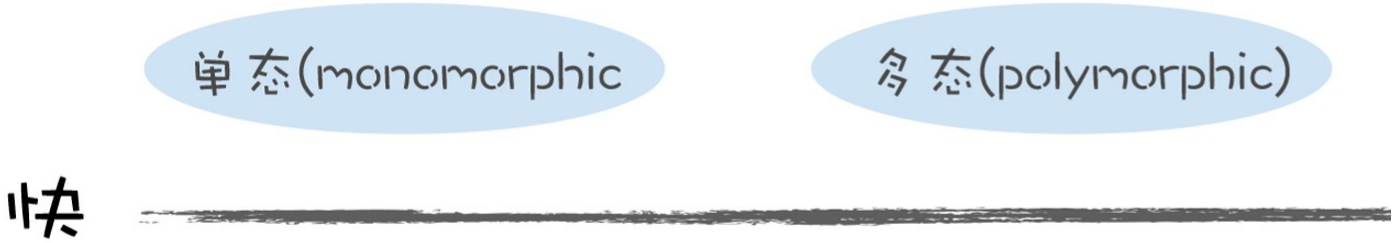
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

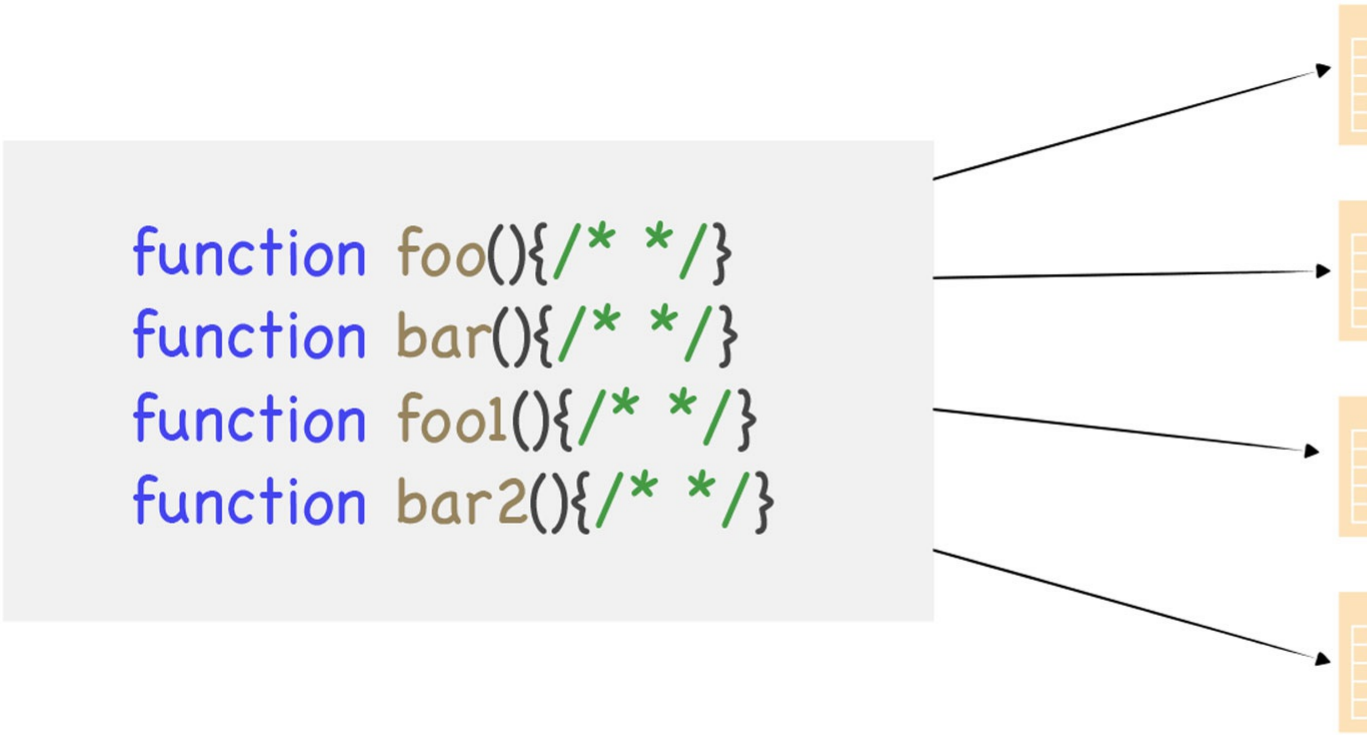
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

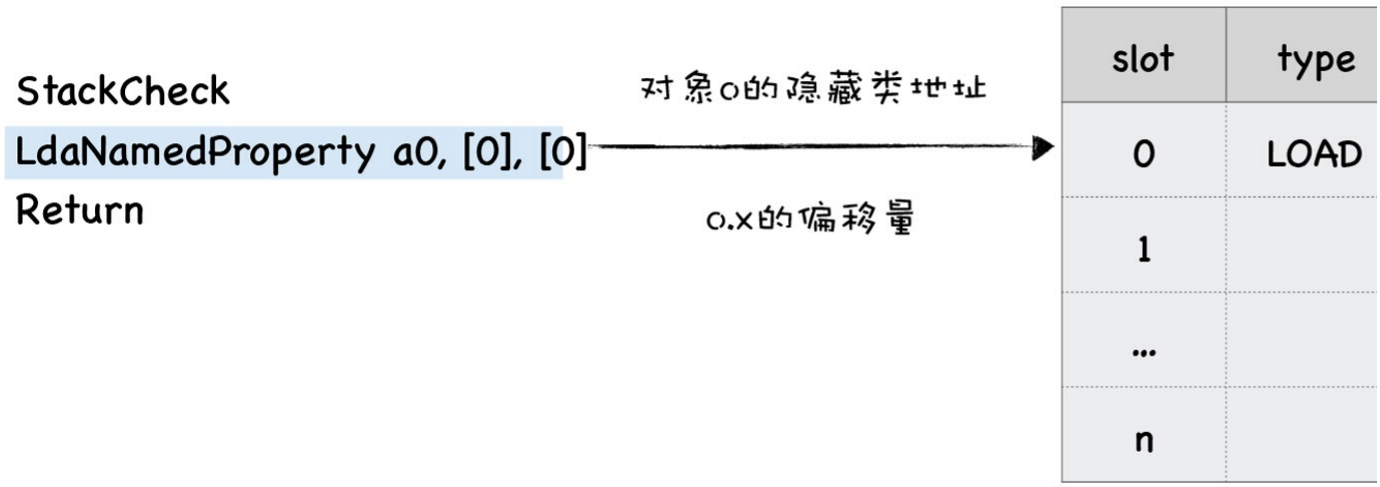
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

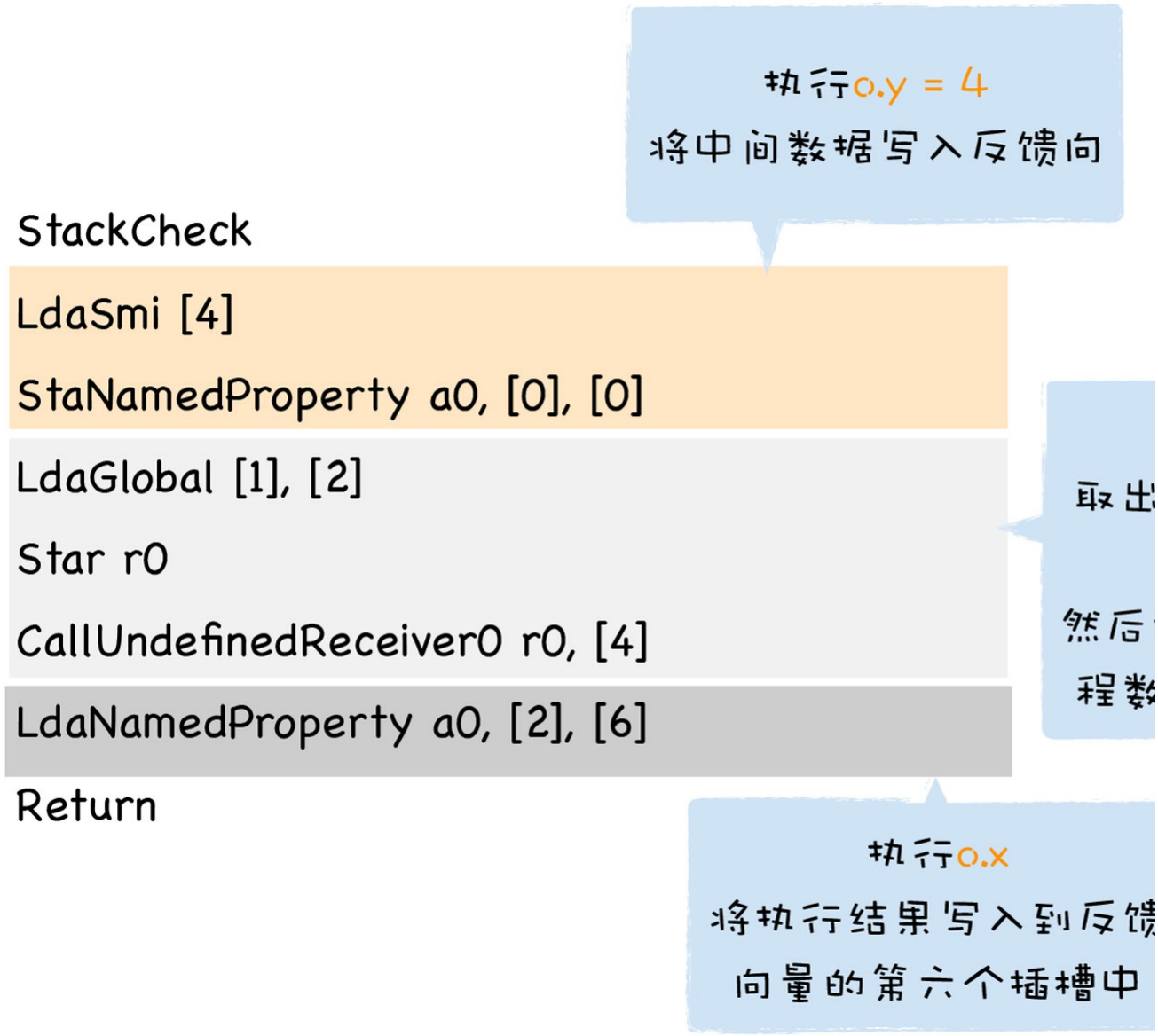
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：

尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())

let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。

上节我们留了个思考题，提到了一段代码是这样的：

```
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)
}
```

我们定义了一个loadX函数，它有一个参数o，该函数只是返回了o.x。

通常V8获取o.x的流程是这样的：**查找对象o的隐藏类，再通过隐藏类查找x属性偏移量，然后根据偏移量获取属性值**，在这段代码中loadX函数会被反复执行，那么获取o.x流程也需要反复被执行。我们有没有办法再度简化这个查找过程，最好能一步到位查找到x的属性值呢？答案是，有的。

其实这是一个关于内联缓存的思考题。我们可以看到，函数loadX在一个for循环里面被重复执行了很多次，因此V8会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存(Inline Cache)**，简称为**IC。**

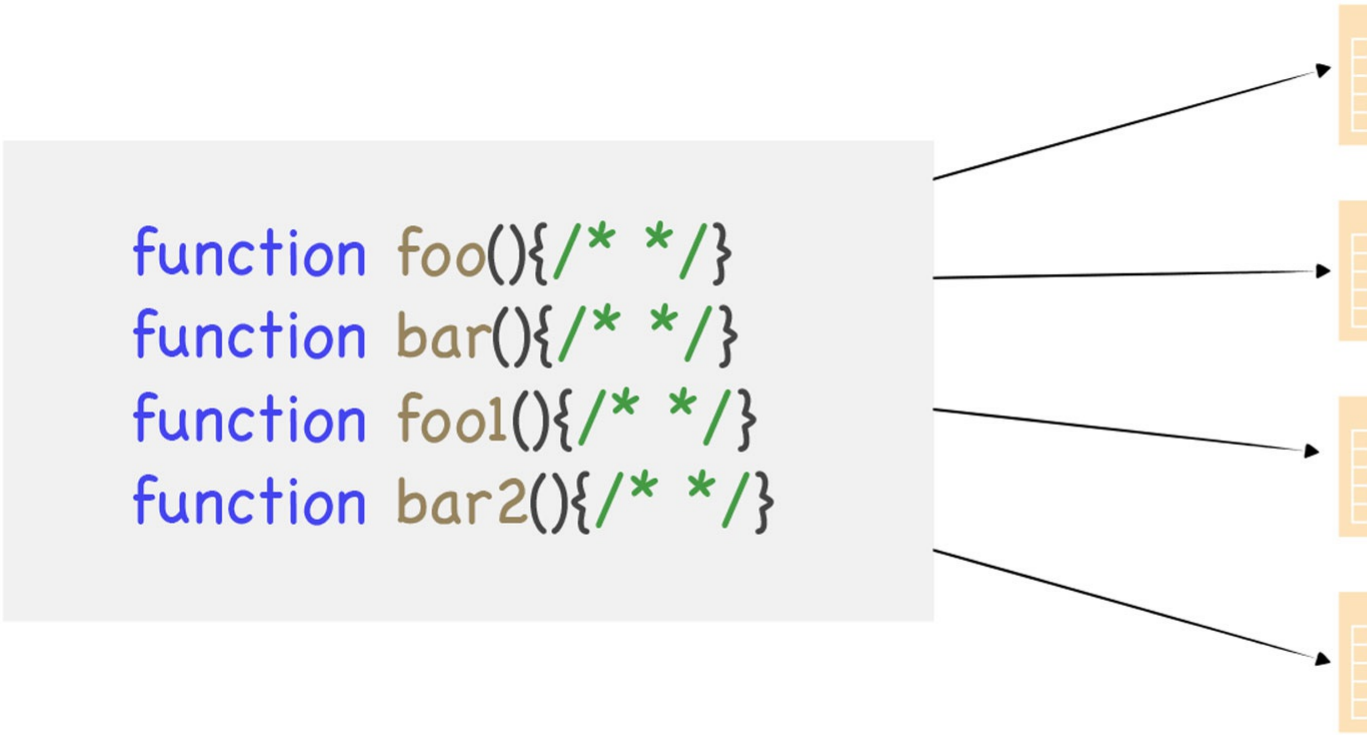
这节课我们就来解答下，V8是怎么通过IC，来加速函数loadX的执行效率的。

什么是内联缓存？

要回答这个问题，我们需要知道IC的工作原理。其实IC的原理很简单，直观地理解，就是在V8执行函数的过程中，会观察函数中一些**调用点(CallSite)上的关键的中间数据**，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此V8利用IC，可以有效提升一些重复代码的执行效率。

接下来，我们就深入分析一下这个过程。

IC会为每个函数维护一个**反馈向量(FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。关于函数和反馈向量的关系你可以参看下图：



反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个**插槽(Slot)**，V8会依次将执行loadX函数的中间数据写入到反馈向量的插槽中。

比如下面这段函数：

```
function loadX(o) {
 o.y = 4
 return o.x
}
```

当V8执行这段函数的时候，它会判断 o.y = 4和 return o.x这两段是**调用点(CallSite)**，因为它们使用了对象和属性，那么V8会在loadX函数的反馈向量中为每个调用点分配一个插槽。

每个插槽中包括了插槽的索引(slot index)、插槽的类型(type)、插槽的状态(state)、隐藏类(map)的地址、还有属性的偏移量，比如上面这个函数中的两个调用点都使用了对象o，那么反馈向量两个插槽中的map属性也都是指向同一个隐藏类的，因此这两个插槽的map地址是一样的。

slot	type	state	
0	LOAD	MONO	34C6
1	STORE	MONO	34C6
...		...	
n	...	...	

了解了反馈向量的大致结构，我们再来看下当V8执行loadX函数时，loadX函数中的关键数据是如何被写入到反馈向量中。

loadX的代码如下所示：

```
function loadX(o) {
    return o.x
}
loadX({x:1})
```

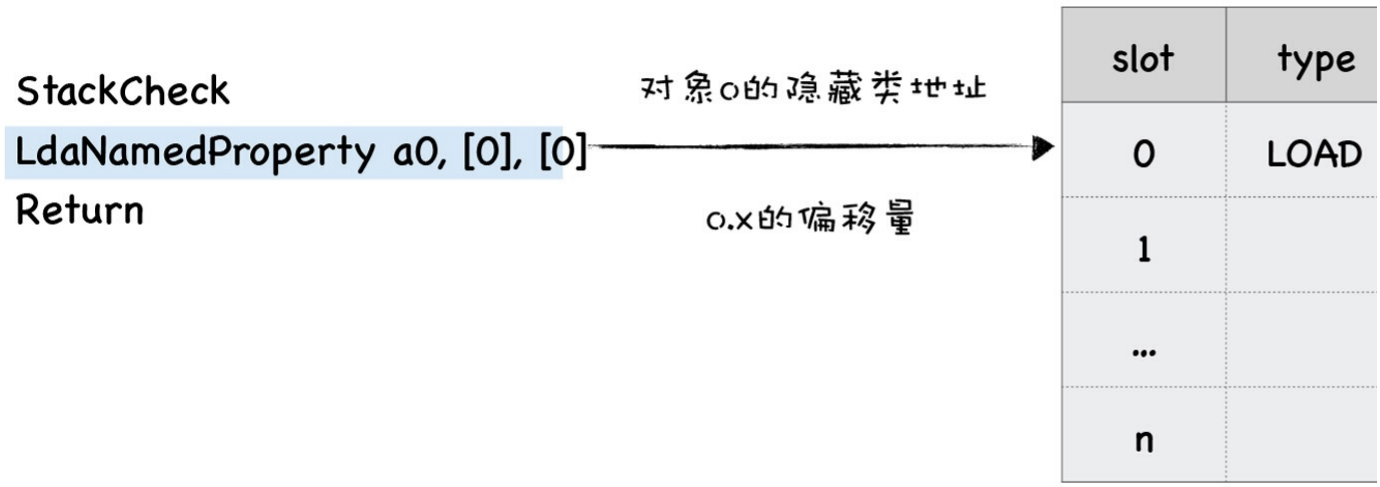
我们将loadX转换为字节码：

```
StackCheck
LdaNamedProperty a0, [0], [0]
Return
```

loadX函数的这段字节码很简单，就三句：

- 第一句是检查栈是否溢出；
- 第二句是LdaNamedProperty，它的作用是取出参数a0的第一个属性值，并将属性值放到累加器中；
- 第三句是返回累加器中的属性值。

这里我们重点关注LdaNamedProperty这句字节码，我们看到它有三个参数。a0就是loadX的第一个参数；第二个参数[0]表示取出对象a0的第一个属性值，这两个参数很好理解。第三个参数就和反馈向量有关了，它表示将LdaNamedProperty操作的中间数据写入到反馈向量中，方括号中间的0表示写入反馈向量的第一个插槽中。具体你可以参看下图：



观察上图，我们可以看出，在函数loadX的反馈向量中，已经缓存了数据：

- 在map栏，缓存了o的隐藏类的地址；
- 在offset一栏，缓存了属性x的偏移量；
- 在type一栏，缓存了操作类型，这里是LOAD类型。在反馈向量中，我们把这种通过o.x来访问对象属性值的操作称为LOAD类型。

V8除了缓存o.x这种LOAD类型的操作以外，还会缓存**存储(STORE)类型**和**函数调用(CALL)类型**的中间数据。

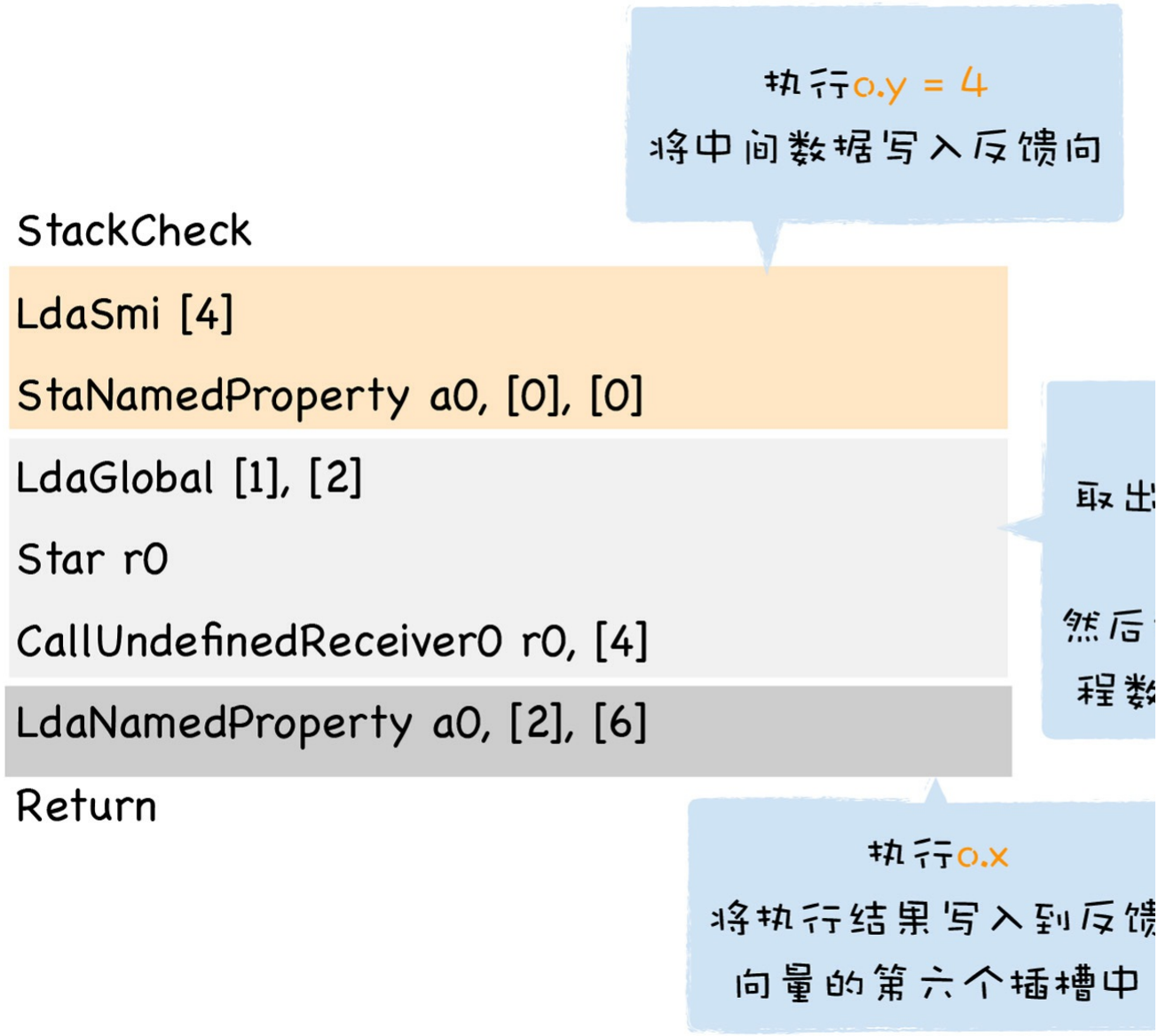
为了分析后面两种存储形式，我们再来看下面这段代码：

```
function foo(){}
function loadX(o) {
    o.y = 4
    foo()
    return o.x
}
loadX({x:1,y:4})
```

相应的字节码如下所示：

```
StackCheck
LdaSmi [4]
StaNamedProperty a0, [0], [0]
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
LdaNamedProperty a0, [2], [6]
Return
```

下图是我画的这段字节码的执行流程：



从图中可以看出，`o.y = 4` 对应的字节码是：

```
LdaSmi [4]
StaNamedProperty a0, [0], [0]
```

这段代码是先使用LdaSmi [4]，将常数4加载到累加器中，然后通过StaNamedProperty的字节码指令，将累加器中的4赋给o.y，这是一个**存储(STORE)类型**的操作，V8会将操作的中间结果存放到反馈向量中的第一个插槽中。

调用foo函数的字节码是：

```
LdaGlobal [1], [2]
Star r0
CallUndefinedReceiver0 r0, [4]
```

解释器首先加载foo函数对象的地址到累加器中，这是通过LdaGlobal来完成的，然后V8会将加载的中间结果存放到反馈向量的第3个插槽中，这是一个存储类型的操作。接下来执行CallUndefinedReceiver0，来实现foo函数的调用，并将执行的中间结果放到反馈向量的第5个插槽中，这是一个**调用(CALL)类型**的操作。

最后就是返回o.x，return o.x仅仅是加载对象中的x属性，所以这是一个**加载(LOAD)类型**的操作，我们在上面介绍过的。最终生成的反馈向量如下图所示：

slot	type	state	r
0	STORE	MONO	34C60
2	LOAD	MONO	10CC0
4	CALL	MONO	
6	LOAD	MONO	

现在有了反馈向量缓存的数据，那V8是如何利用这些数据的呢？

当V8再次调用loadX函数时，比如执行到loadX函数中的return o.x语句时，它就会在对应的插槽中查找x属性的偏移量，之后V8就能直接去内存中获取o.x的属性值了。这样就大大提升了V8的执行效率。

多态和超态

好了，通过缓存执行过程中的基础信息，就能够提升下次执行函数时的效率，但是这有一个前提，那就是多次执行时，对象的形状是固定的，如果对象的形状不是固定的，那V8会怎么处理呢？

我们调整一下上面这段loadX函数的代码，调整后的代码如下所示：

```
function loadX(o) {
    return o.x
}
var o = { x: 1,y:3}
var o1 = { x: 3, y:6,z:4}
for (var i = 0; i < 90000; i++) {
    loadX(o)
    loadX(o1)
}
```

我们可以看到，对象o和o1的形状是不同的，这意味着V8为它们创建的隐藏类也是不同的。

第一次执行时loadX时，V8会将o的隐藏类记录在反馈向量中，并记录属性x的偏移量。那么当再次调用loadX函数时，V8会取出反馈向量中记录的隐藏类，并和新的o1的隐藏类进行比较，发现不是一个隐藏类，那么此时V8就无法使用反馈向量中记录的偏移量信息了。

面对这种情况，V8会选择将新的隐藏类也记录在反馈向量中，同时记录属性值的偏移量，这时，反馈向量中的第一个槽里就包含了两个隐藏类和偏移量。具体你可以参看下图：

slot	type	state	
0	LOAD	POLY	34C6 10CC
...		...	
n	...	...	

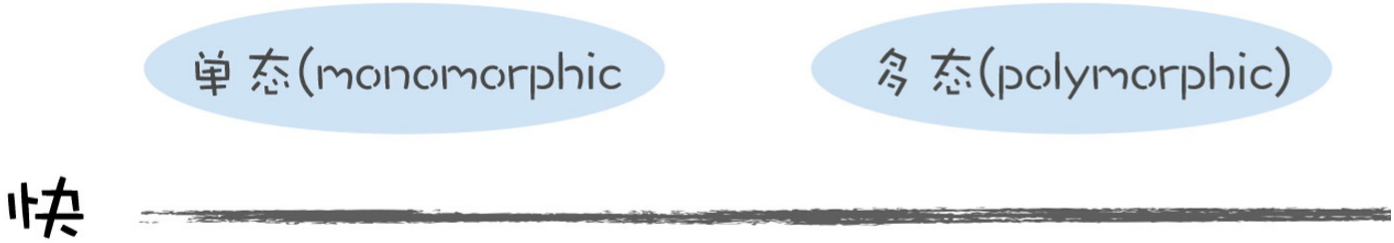
当V8再次执行loadX函数中的o.x语句时，同样会查找反馈向量表，发现第一个槽中记录了两个隐藏类。这时，V8需要额外做一件事，那就是拿这个新的隐藏类和第一个插槽中的两个隐藏类来一一比较，如果新的隐藏类和第一个插槽中某个隐藏类相同，那么就使用该命中的隐藏类的偏移量。如果没有相同的呢？同样将新的信息添加到反馈向量的第一个插槽中。

现在我们知道了，一个反馈向量的一个插槽中可以包含多个隐藏类的信息，那么：

- 如果一个插槽中只包含1个隐藏类，那么我们称这种状态为**单态(monomorphic)**；
- 如果一个插槽中包含了2～4个隐藏类，那我们称这种状态为**多态(polymorphic)**；
- 如果一个插槽中超过4个隐藏类，那我们称这种状态为**超态(magamorphic)**。

如果函数loadX的反馈向量中存在多态或者超态的情况，其执行效率肯定要低于单态的，比如当执行到o.x的时候，V8会查询反馈向量的第一个插槽，发现里面有多个map的记录，那么V8就需要取出o的隐藏类，来和插槽中记录的隐藏类一一比较，如果记录的隐藏类越多，那么比较的次数也就越多，这就意味着执行效率越低。

比如插槽中包含了2～4个隐藏类，那么可以使用线性结构来存储，如果超过4个，那么V8会采取hash表的结构来存储，这无疑会拖慢执行效率。单态、多态、超态等三种情况的执行性能如下图所示：



尽量保持单态

这就是IC的一些基础情况，非常简单，只是为每个函数添加了一个缓存，当第一次执行该函数时，V8会将函数中的存储、加载和调用相关的中间结果保存到反馈向量中。当再次执行时，V8就要去反馈向量中查找相关中间信息，如果命中了，那么就直接使用中间信息。

了解了IC的基础执行原理，我们就能理解一些最佳实践背后的道理，这样你并不需要去刻意记住这些最佳实践了，因为你已经从内部理解了它。

总的来说，我们只需要记住一条就足够了，那就是**单态的性能优于多态和超态，**所以我们需要稍微避免多态和超态的情况。

要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个loadX(o)的函数，那么当传递参数时，尽量不要使用多个不同形状的o对象。

总结

这节课我们通过分析IC的工作原理，来介绍了它是如何提升代码执行速度的。

虽然隐藏类能够加速查找对象的速度，但是在V8查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。

如果一个函数中利用了对象的属性，并且这个函数会被多次执行，那么V8就会考虑，怎么将这个查找过程再度简化，最好能将属性的查找过程能一步到位。

因此，V8引入了IC，IC会监听每个函数的执行过程，并在一些关键的地方埋下监听点，这些包括了加载对象属性(Load)、给对象属性赋值(Store)、还有函数调用(Call)，V8会将监听到的数据写入一个称为**反馈向量(FeedBack Vector)**的结构中，同时V8会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8就可以缩短对象属性的查找路径，从而提升执行效率。

但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。我们在实际项目中，要尽量避免出现多态或者超态的情况。

最后我还想强调一点，虽然我们分析的隐藏类和IC能提升代码的执行速度，但是在实际的项目中，影响执行性能的因素非常多，**找出那些影响性能瓶颈才是至关重要**的，**你不需要过度关注微优化，你也不需要过度担忧你的代码是否破坏了隐藏类或者IC的机制**，因为相对于其他的性能瓶颈，它们对效率的影响可能是微不足道的。

思考题

观察下面两段代码：

```
let data = [1, 2, 3, 4]
data.forEach((item) => console.log(item.toString())
```

```
let data = ['1', 2, '3', 4]
data.forEach((item) => console.log(item.toString())
```

你认为这两段代码，哪段的执行效率高，为什么？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。