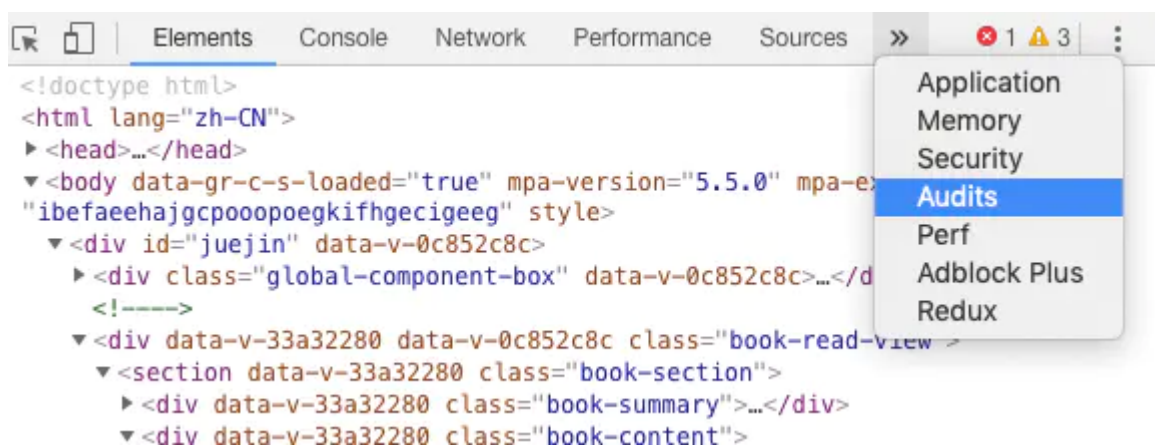


# 从 V8 中看 JS 性能优化

注意：该知识点属于性能优化领域。

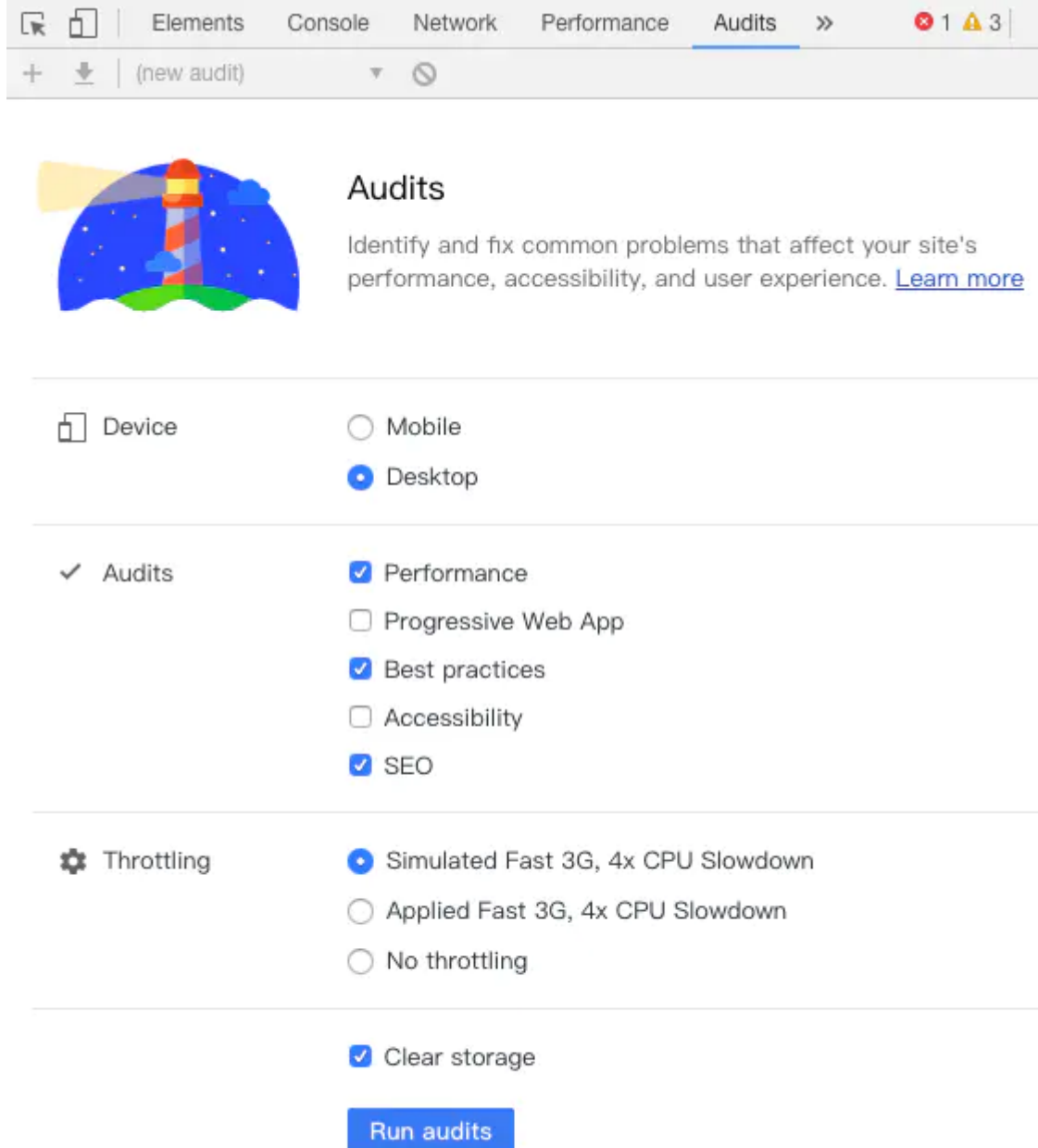
## 测试性能工具

Chrome 已经提供了一个大而全的性能测试工具 **Audits**



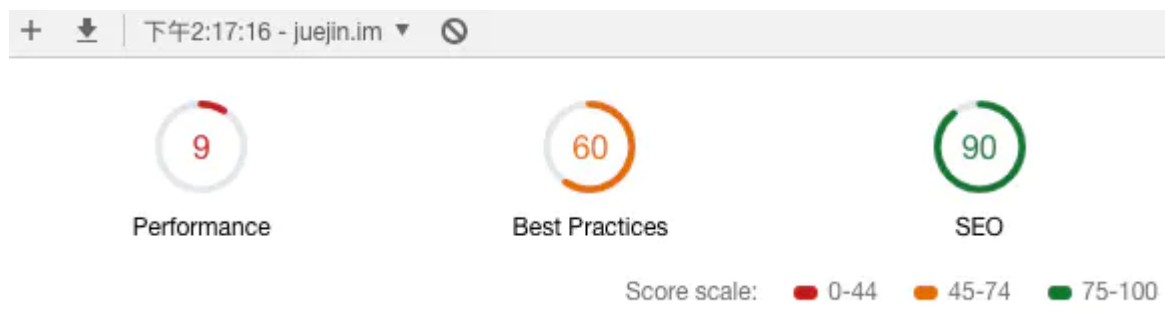
Audits 所处位置

点我们点击 Audits 后，可以看到如下的界面



Audits 界面

在这个界面中，我们可以选择想测试的功能然后点击 **Run audits**，工具就会自动运行帮助我们测试问题并且给出一个完整的报告



Audits 工具给出的报告

上图是给掘金首页测试性能后给出的一个报告，可以看到报告中分别为**性能**、**体验**、**SEO**都给出了打分，并且每一个指标都有详细的**评估**

## Performance

9

### Metrics

First Contentful Paint	5,460 ms ▲	First Meaningful Paint	6,970 ms ▲
Speed Index	11,970 ms ▲	First CPU Idle	14,590 ms ▲
Time to Interactive	15,740 ms ▲	Estimated Input Latency	1,982 ms ▲

View Trace

Values are estimated and may vary.



指标中的详细评估

评估结束后，工具还提供了一些**建议**便于我们提高这个指标的分数

### Opportunities

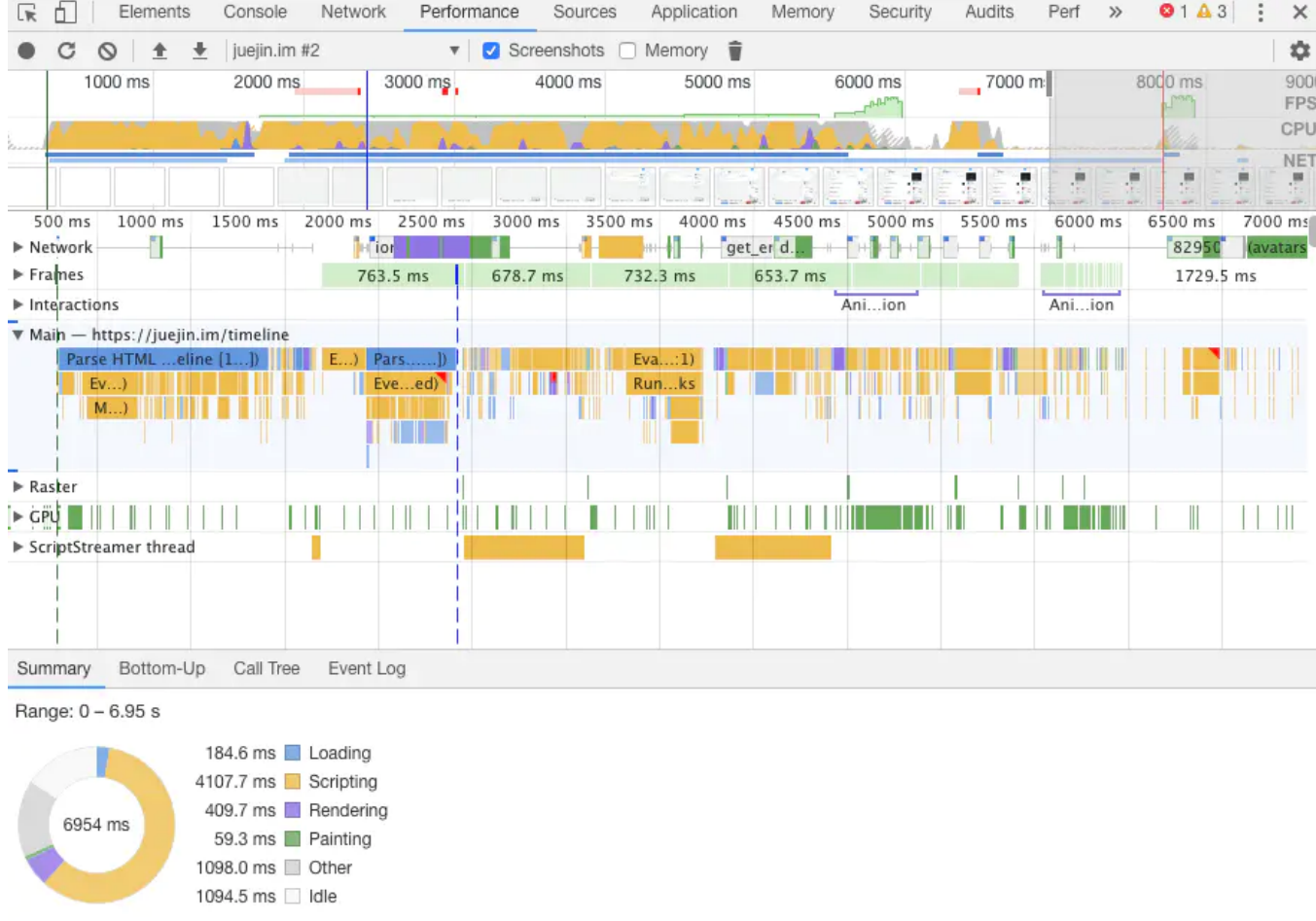
These are opportunities to speed up your application by optimizing the following resources.

	Resource to optimize		Estimated Savings
1	Defer offscreen images	<div></div>	1.7 s ▼
2	Serve images in next-gen formats	<div></div>	1.7 s ▼
3	Properly size images	<div></div>	1.38 s ▼
4	Defer unused CSS	<div></div>	0.9 s ▼
5	Avoid multiple, costly round trips to any origin	<div></div>	0.68 s ▼

优化建议

我们只需要一条条根据建议去优化性能即可。

除了 **Audits** 工具之外，还有一个 **Performance** 工具也可以供我们使用。



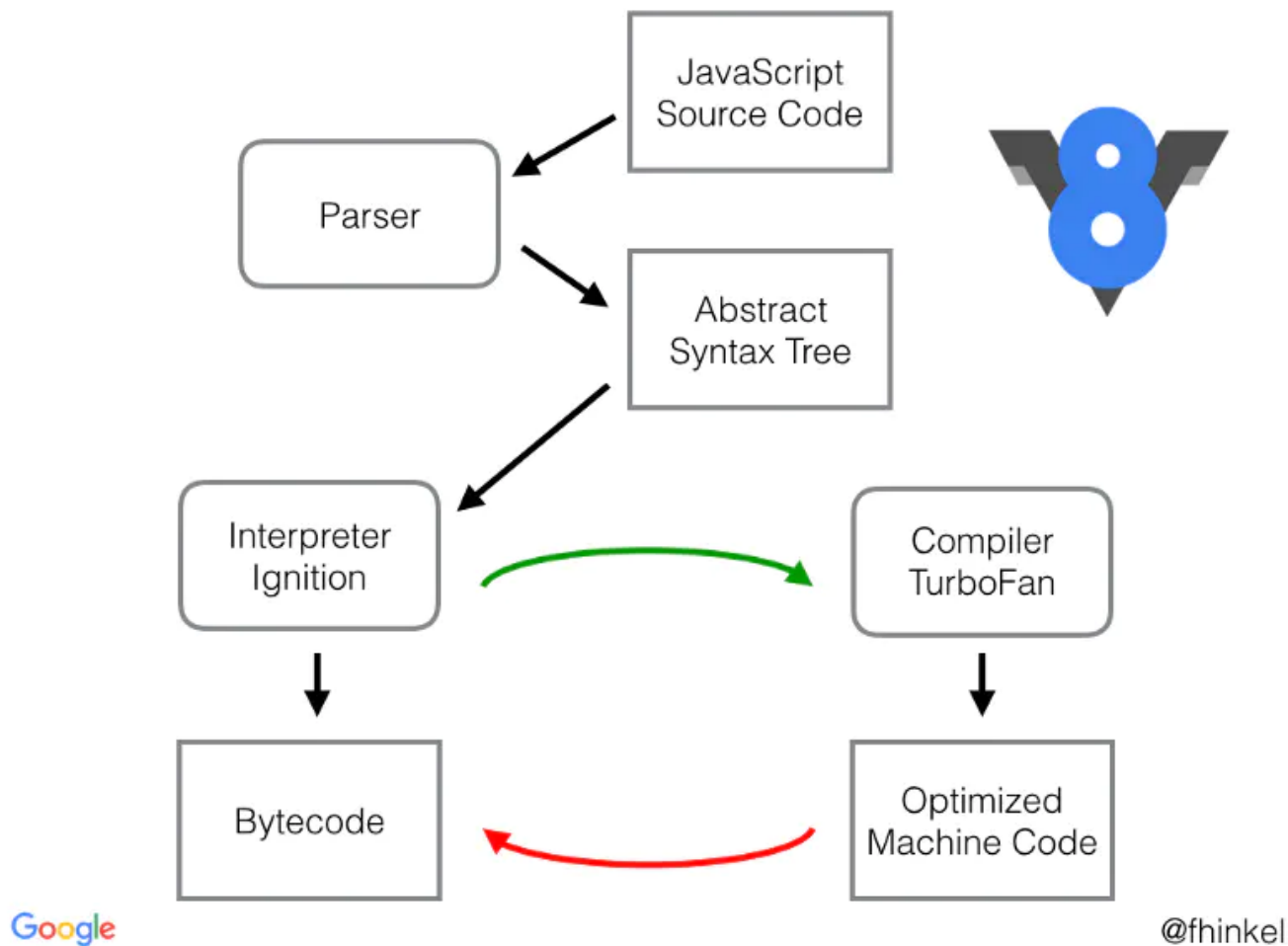
在这张图中，我们可以详细的看到每个时间段中浏览器在处理什么事情，哪个过程最消耗时间，便于我们更加详细的了解性能瓶颈。

## JS 性能优化

JS 是编译型还是解释型语言其实并不固定。首先 JS 需要有引擎才能运行起来，无论是浏览器还是在 Node 中，这是解释型语言的特性。但是在 V8 引擎下，又引入了 **TurboFan** 编译器，他会在特定的情况下进行优化，将代码编译成执行效率更高的 **Machine Code**，当然这个编译器并不是 JS 必须需要的，只是为了提高代码执行性能，所以总的来说 JS 更偏向于解释型语言。

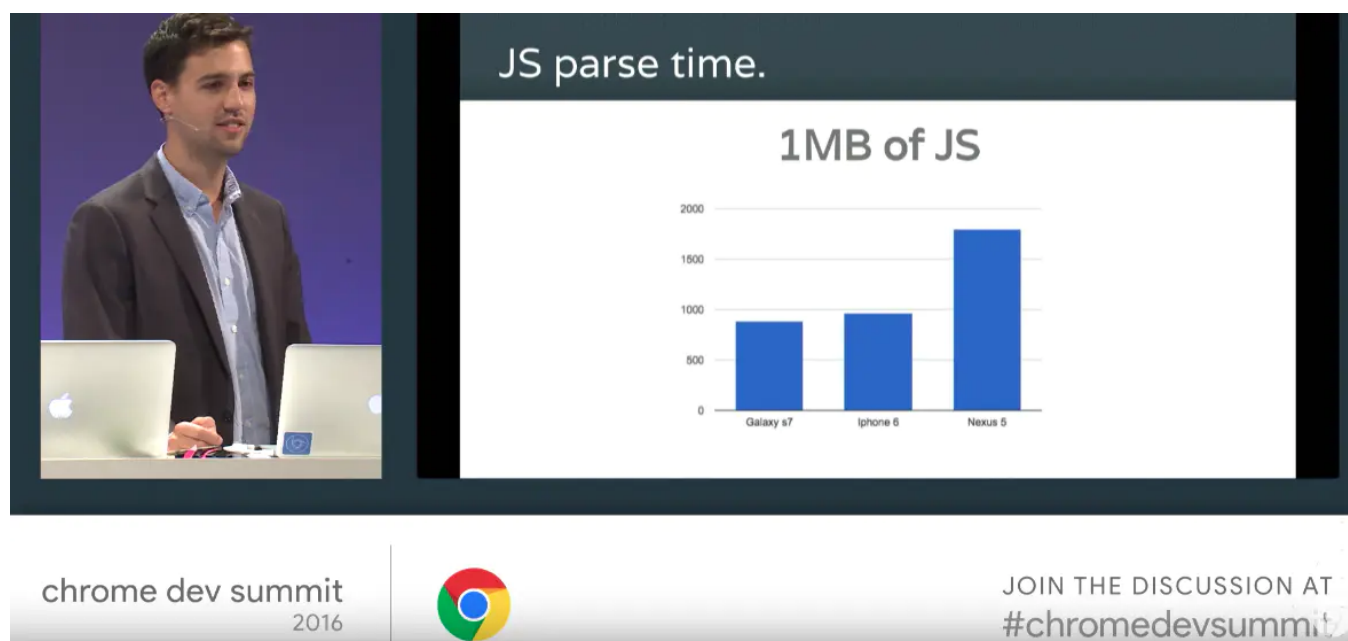
那么这一小节的内容主要会针对于 Chrome 的 **V8** 引擎来讲解。

在这一过程中，JS 代码首先会解析为抽象语法树（AST），然后通过解释器或者编译器转化为 **Bytecode** 或者 **Machine Code**



V8 转化代码的过程

从上图中我们可以发现，JS 会首先被解析为 AST，解析的过程其实是略慢的。代码越多，解析的过程也就耗费越长，这也是我们需要压缩代码的原因之一。另外一种减少解析时间的方式是预解析，会作用于未执行的函数，这个我们下面再谈。

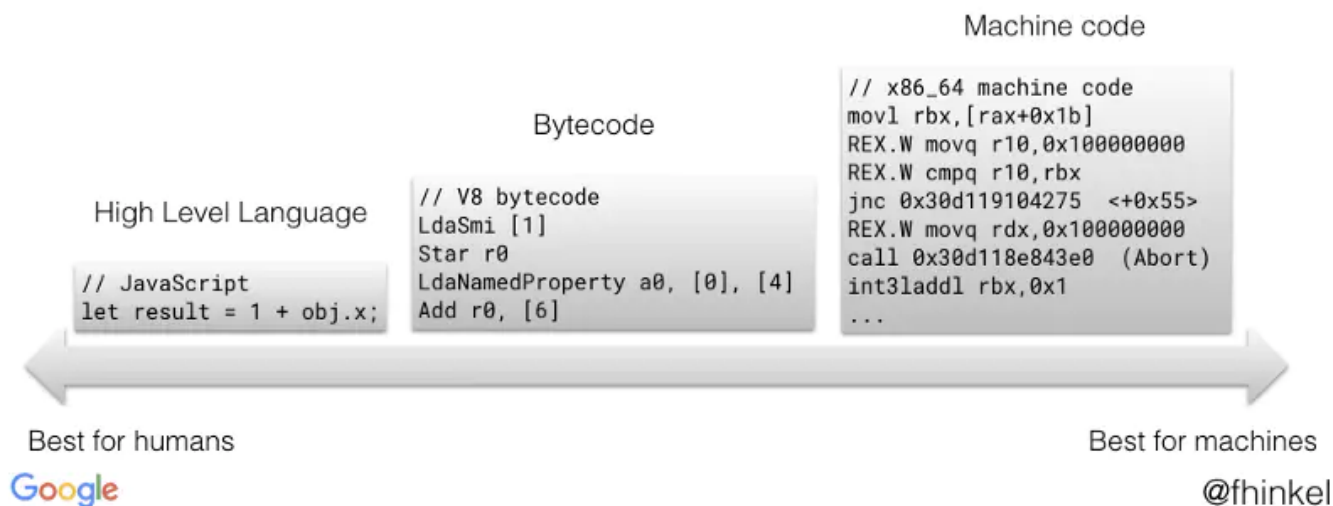


2016 年手机解析 JS 代码的速度

这里需要注意一点，对于函数来说，应该尽可能避免声明嵌套函数（类也是函数），因为这样会造成函数的重复解析。

```
function test1() {
  // 会被重复解析
  function test2() {}
}
```

然后 **Ignition** 负责将 AST 转化为 Bytecode, **TurboFan** 负责编译出优化后的 Machine Code, 并且 Machine Code 在执行效率上优于 Bytecode



那么我们就产生了一个疑问, 什么情况下代码会编译为 **Machine Code**?

JS 是一门**动态类型**的语言, 并且还有一大堆的规则。简单的加法运算代码, 内部就需要考虑好几种规则, 比如数字相加、字符串相加、对象和字符串相加等等。这样的情况也就势必导致了内部要增加很多判断逻辑, 降低运行效率。

```
function test(x) {
  return x + x
}
```

```
test(1)
test(2)
test(3)
test(4)
```

对于以上代码来说, 如果一个函数被**多次调用**并且参数一直传入 **number** 类型, 那么 V8 就会认为该段代码可以编译为 Machine Code, 因为你**固定了类型**, 不需要再执行很多判断逻辑了。

但是如果一旦我们传入的参数**类型改变**, 那么 Machine Code 就会被 **DeOptimized** 为 Bytecode, 这样就有性能上的一个损耗了。所以如果我们希望代码能多的编译为 Machine Code 并且 DeOptimized 的次数减少, 就应该尽可能保证传入的**类型一致**。

那么你可能会有一个疑问, 到底优化前后有多少的提升呢, 接下来我们就来实践测试一下到底有多少的提升。

```

const { performance, PerformanceObserver } = require('perf_hooks')

function test(x) {
  return x + x
}

// node 10 中才有 PerformanceObserver
// 在这之前的 node 版本可以直接使用 performance 中的 API
const obs = new PerformanceObserver((list, observer) => {
  console.log(list.getEntries())
  observer.disconnect()
})

obs.observe({ entryTypes: ['measure'], buffered: true })

performance.mark('start')

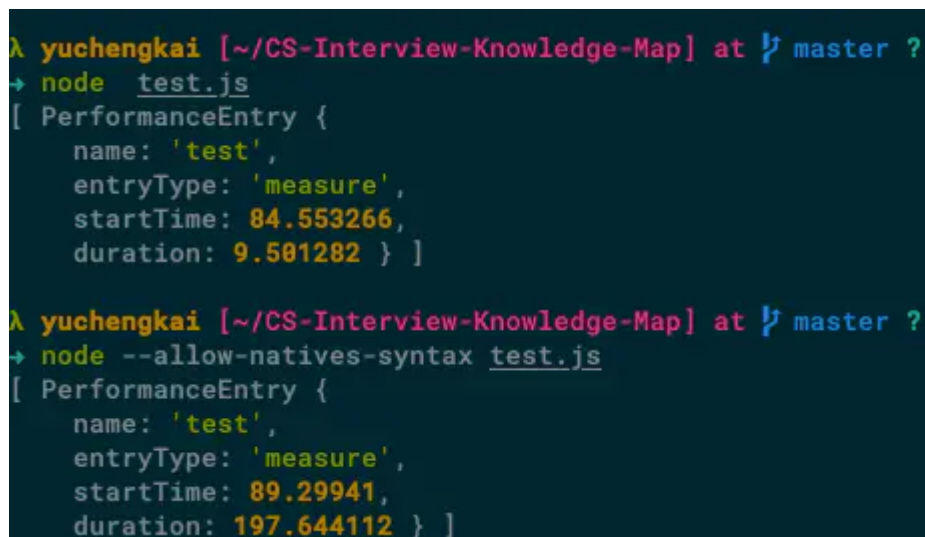
let number = 10000000
// 不优化代码
%NeverOptimizeFunction(test)

while (number--) {
  test(1)
}

performance.mark('end')
performance.measure('test', 'start', 'end')

```

以上代码中我们使用了 `performance` API，这个 API 在性能测试上十分好用。不仅可以用来测量代码的执行时间，还能用来测量各种网络连接中的时间消耗等等，并且这个 API 也可以在浏览器中使用。



```

λ yuchengkai [~/CS-Interview-Knowledge-Map] at 10:10 master ?
→ node test.js
[ PerformanceEntry {
  name: 'test',
  entryType: 'measure',
  startTime: 84.553266,
  duration: 9.501282 } ]

λ yuchengkai [~/CS-Interview-Knowledge-Map] at 10:10 master ?
→ node --allow-natives-syntax test.js
[ PerformanceEntry {
  name: 'test',
  entryType: 'measure',
  startTime: 89.29941,
  duration: 197.644112 } ]

```

优化与不优化代码之间的巨大差距

从上图中我们可以发现，优化过的代码执行时间只需要 9ms，但是不优化过的代码执行时间却是前者的二十倍，已经接近 200ms 了。在这个案例中，相信大家已经看到了 V8 的性能优化到底有多强，只需要我们符合一定的规则书写代码，引擎底层就能帮助我们自动优化代码。



另外，编译器还有个骚操作 **Lazy-Compile**，当函数没有被执行的时候，会对函数进行一次预解析，直到代码被执行以后才会被解析编译。对于上述代码来说，`test` 函数需要被预解析一次，然后在调用的时候再被解析编译。但是对于这种函数马上就被调用的情况来说，预解析这个过程其实是多余的，那么有什么办法能够让代码不被预解析呢？

其实很简单，我们只需要给函数**套上括号**就可以了

```
(function test(obj) {  
  return x + x  
})
```

但是不可能我们为了性能优化，给所有的函数都去套上括号，并且也不是所有函数都需要这样做。我们可以通过 [optimize-js](#) 实现这个功能，这个库会分析一些函数的使用情况，然后给需要的函数添加括号，当然这个库很久没人维护了，如果需要使用的話，还是需要测试过相关内容的。