5.3 asm.js

asm.js（http://asmjs.org）这个标签是指 JavaScript 语言中可以高度优化的一个子集。通过小心避免某些难以优化的机制和模式（垃圾收集、类型强制转换，等等），asm.js 风格的代码可以被 JavaScript 引擎识别并进行特别激进的底层优化。

和本章前面讨论的其他程序性能机制不同，asm.js 并不是 JavaScript 语言规范需要采纳的某种东西。虽然 asm.js 规范的确存在（http://asmjs,org/spec/latest/），但它主要是用来追踪一系列达成一致的备选优化方案而不是对 JavaScript 引擎的一组要求。

目前还没有提出任何新的语法。事实上，asm.js 提出了一些识别满足 asm.js 规则的现存标准 JavaScript 语法的方法，并让引擎据此实现它们自己的优化。

浏览器提供者之间在关于程序中应如何激活 asm.js 这一点上有过一些分歧。早期版本的 asm.js 实验需要一个 `"use asm"`；pragma（类似于严格模式的 `"use strict"`;）帮助提醒 JavaScript 引擎寻找 asm.js 优化机会。另外一些人认为，asm.js 应该就是一个启发式的集合，引擎应该能够自动识别，无需开发者做任何额外的事情。这意味着，从理论上说，现有的程序可以从 asm.js 风格的优化得益而无需特意做什么。

5.3.1 如何使用 asm.js 优化

关于 asm.js 优化，首先要理解的是类型和强制类型转换（参见本书的“类型和语法”部分）。如果 JavaScript 引擎需要跟踪一个变量在各种各样的运算之间的多个不同类型的值，才能按需处理类型之间的强制类型转换，那么这大量的额外工作会使得程序优化无法达到最优。

为了解释明了，我们在这里将使用 asm.js 风格代码，但你要清楚，通常并不需要手工编写这样的代码。asm.js 通常是其他工具的编译目标，比如 Emscripten（https://github.com/kripken/emscripten/wiki）。当然，你也可以自己编写 asm.js 代码，但一般来说，这想法并不好，因为这是非常耗时且容易出错的过程。尽管如此，可能还是会有一些情况需要你修改代码，以便于 asm.js 优化。

还有一些技巧可以用来向支持 asm.js 的 JavaScript 引擎暗示变量和运算想要的类型是什么，使它可以省略这些类型转换跟踪步骤。

比如：

```
var a = 42;

// ..
```

```
var b = a;
```

在这个程序中，赋值 `b = a` 留下了变量类型二义性的后门。但它也可以换一种方式，写成这样：

```
var a = 42;

// ..

var b = a | 0;
```

此处我们使用了与 `0` 的 `|`（二进制或）运算，除了确保这个值是 32 位整型之外，对于值没有任何效果。这样的代码在一般的 JavaScript 引擎上都可以正常工作。而对支持 asm.js 的 JavaScript 引擎来说，这段代码就发出这样的信号，`b` 应该总是被当作 32 位整型来处理，这样就可以省略强制类型转换追踪。

类似地，可以这样把两个变量的加运算限制为更高效的整型加运算（而不是浮点型）：

```
(a + b) | 0
```

另一方面，支持 asm.js 的 JavaScript 引擎可以看到这个提示并推导出这里的 + 运算应该是 32 位整型加，因为不管怎样，整个表达式的结果都会自动规范为 32 位整型。

5.3.2 asm.js 模块

对 JavaScript 性能影响最大的因素是内存分配、垃圾收集和作用域访问。asm.js 对这些问题提出的一个解决方案就是，声明一个更正式的 asm.js“模块”，不要和 ES6 模块混淆。请参考本系列《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分。

对一个 asm.js 模块来说，你需要明确地导入一个严格规范的命名空间——规范将之称为 `stdlib`，因为它应该代表所需的标准库——以导入必要的符号，而不是通过词法作用域使用全局的那些符号。基本上，`window` 对象就是一个 asm.js 模块可以接受的 `stdlib` 对象，但是，你能够而且可能也需要构造一个更加严格的。

你还需要声明一个*堆*（heap）并将其传入。这个术语用于表示内存中一块保留的位置，变量可以直接使用而不需要额外的内存请求或释放之前使用的内存。这样，asm.js 模块就不需要任何可能导致内存扰动的动作了，只需使用预先保留的空间即可。

一个堆就像是一个带类型的 `ArrayBuffer`，比如：

```
var heap = new ArrayBuffer( 0x10000 );  // 64k堆
```

由于使用这个预留的 64k 二进制空间，asm.js 模块可以在这个缓冲区存储和获取值，不需

要付出任何内存分配和垃圾收集的代价。举例来说，可以在模块内部使用堆缓冲区备份一个 64 位浮点值数组，就像这样：

```
var arr = new Float64Array( heap );
```

用一个简单快捷的 asm.js 风格模块例子来展示这些细节是如何结合到一起的。我们定义了一个 `foo(..)`。它接收一个起始值（x）和终止值（y）整数构成一个范围，并计算这个范围内的值的所有相邻数的乘积，然后算出这些值的平均数：

```
function fooASM(stdlib,foreign,heap) {
    "use asm";

    var arr = new stdlib.Int32Array( heap );

    function foo(x,y) {
        x = x | 0;
        y = y | 0;

        var i = 0;
        var p = 0;
        var sum = 0;
        var count = ((y|0) - (x|0)) | 0;

        // 计算所有的内部相邻数乘积
        for (i = x | 0;
            (i | 0) < (y | 0);
            p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            // 存储结果
            arr[ p >> 3 ] = (i * (i + 1)) | 0;
        }

        // 计算所有中间值的平均数
        for (i = 0, p = 0;
            (i | 0) < (count | 0);
            p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            sum = (sum + arr[ p >> 3 ]) | 0;
        }

        return +(sum / count);
    }

    return {
        foo: foo
    };
}

var heap = new ArrayBuffer( 0x1000 );
var foo = fooASM( window, null, heap ).foo;

foo( 10, 20 );      // 233
```

出于展示的目的，这个 asm.js 例子是手写的，所以它并不能代表由目标为 asm.js 的编译工具产生的同样功能的代码。但是，它确实显示了 asm.js 代码的典型特性，特别是类型提示以及堆缓冲区在存储临时变量上的使用。

第一个对 `fooASM(..)` 的调用建立了带堆分配的 asm.js 模块。结果是一个 `foo(..)` 函数，我们可以按照需要调用任意多次。这些 `foo(..)` 调用应该被支持 asm.js 的 JavaScript 引擎专门优化。很重要的一点是，前面的代码完全是标准 JavaScript，在非 asm.js 引擎中也能正常工作（没有特殊优化）。

显然，使 asm.js 代码如此高度可优化的那些限制的特性显著降低了这类代码的使用范围。asm.js 并不是对任意程序都适用的通用优化手段。它的目标是对特定的任务处理提供一种优化方法，比如数学运算（如游戏中的图形处理）。

5.4 小结

本部分的前四章都是基于这样一个前提：异步编码模式使我们能够编写更高效的代码，通常能够带来非常大的改进。但是，异步特性只能让你走这么远，因为它本质上还是绑定在一个单事件循环线程上。

因此，在这一章里，我们介绍了几种能够进一步提高性能的程序级别的机制。

Web Worker 让你可以在独立的线程运行一个 JavaScript 文件（即程序），使用异步事件在线程之间传递消息。它们非常适用于把长时间的或资源密集型的任务卸载到不同的线程中，以提高主 UI 线程的响应性。

SIMD 打算把 CPU 级的并行数学运算映射到 JavaScript API，以获得高性能的数据并行运算，比如在大数据集上的数字处理。

最后，asm.js 描述了 JavaScript 的一个很小的子集，它避免了 JavaScript 难以优化的部分（比如垃圾收集和强制类型转换），并且让 JavaScript 引擎识别并通过激进的优化运行这样的代码。可以手工编写 asm.js，但是会极端费力且容易出错，类似于手写汇编语言（这也是其名字的由来）。实际上，asm.js 也是高度优化的程序语言交叉编译的一个很好的目标，比如 Emscripten 把 C/C++ 转换成 JavaScript（https://github.com/kripken/emscripten/wiki）。

JavaScript 还有一些更加激进的思路已经进入非常早期的讨论，尽管本章并没有明确包含这些内容，比如近似的直接多线程功能（而不是藏在数据结构 API 后面）。不管这些最终会不会实现，还是我们将只能看到更多的并行特性偷偷加入 JavaScript，但确实可以预见，未来 JavaScript 在程序级别将获得更加优化的性能。

第 6 章

性能测试与调优

本部分的前四章都是关于（异步与并发）编码模式的性能，第 5 章是关于宏观程序架构级的性能。这一章要讨论的主题则是微观性能，关注点在单个表达式和语句。

最能引发普遍好奇心的领域之一（真的，有些开发者可能会沉迷于此）就是分析和测试编写一行或一块代码的多个选择，然后确定哪一种更快。

我们将要来探讨这些问题中的一部分，但从一开始你就要明白，本章的目的不是为了满足对微观性能调优的沉迷，比如某个 JavaScript 引擎上运行 `++a` 是不是会比 `a++` 快。本章更重要的目标是弄清楚哪些种类的 JavaScript 性能更重要，哪些种类则无关紧要，以及如何区分。

但是，在得出结论之前，首先需要探讨如何最精确可靠地测试 JavaScript 性能，因为我们的知识库中充满了大量的误解和迷思。需要筛选掉所有垃圾以得到清晰的概念。

6.1 性能测试

好，现在是时候消除一些误解了。我敢打赌，如果被问到如何测试某个运算的速度（执行时间），绝大多数 JavaScript 开发者都会从类似下面的代码开始：

```
var start = (new Date()).getTime(); // 或者Date.now()

// 进行一些操作

var end = (new Date()).getTime();

 console.log( "Duration:", (end - start) );
```

如果这大致上就是你首先想到的，请举手。嗯，我想就是如此。这种方案有很多错误，不过别难过，我们会找到正确方法的。

这个测量方式到底能告诉你什么呢？理解它做了什么以及关于这个运算的执行时间不能提供哪些信息，就是学习如何正确测试 JavaScript 性能的关键所在。

如果报告的时间是 `0`，可能你会认为它的执行时间小于 1ms。但是，这并不十分精确。有些平台的精度并没有达到 1ms，而是以更大的递增间隔更新定时器。比如，Windows（也就是 IE）的早期版本上的精度只有 15ms，这就意味着这个运算的运行时间至少需要这么长才不会被报告为 `0`！

还有，不管报告的时长是多少，你能知道的唯一一点就是，这个运算的这次特定的运行消耗了大概这么长时间。而它是不是总是以这样的速度运行，你基本上一无所知。你不知道引擎或系统在这个时候有没有受到什么影响，以及其他时候这个运算会不会运行得更快。

如果时长报告是 `4` 呢？你能更加确定它的运行需要大概 4ms 吗？不能。它消耗的时间可能要短一些，而且在获得 `start` 或 `end` 时间戳之间也可能有其他一些延误。

更麻烦的是，你也不知道这个运算测试的环境是否过度优化了。有可能 JavaScript 引擎找到了什么方法来优化你这个独立的测试用例，但在更真实的程序中是无法进行这样的优化的，那么这个运算就会比测试时跑得慢。

那么，能知道的是什么呢？很遗憾，根据前面提出的内容，我们几乎一无所知。这样低置信度的测试几乎无力支持你的任何决策。这个性能测试基本上是无用的。更坏的是，它是危险的，因为它可能提供了错误的置信度，不仅是对你，还有那些没有深入思考带来测试结果的条件的人员。

6.1.1 重复

“好吧，”你现在会说，“那就用一个循环把它包起来，这样整个测试的运行时间就会更长一些了。”如果重复一个运算 100 次，然后整个循环报告共消耗了 137ms，那你就可以把它除以 100，得到每次运算的平均用时为 1.37ms，是这样吗？

并不完全是这样。

简单的数学平均值绝对不足以对你要外推到整个应用范围的性能作出判断。迭代 100 次，即使只有几个（过高或过低的）的异常值也可以影响整个平均值，然后在重复应用这个结论的时候，你还会扩散这个误差，产生更大的欺骗性。

你也可以不以固定次数执行运算，转而循环运行测试，直到达到某个固定的时间。这可能会更可靠一些，但如何确定要执行多长时间呢？你可能会猜测，执行时间应该是你的运算执行的单次时长的若干倍。错。

实际上，重复执行的时间长度应该根据使用的定时器的精度而定，专门用来最小化不精确性。定时器的精度越低，你需要运行的时间就越长，这样才能确保错误率最小化。15ms 的定时器对于精确的性能测试来说是非常差劲的。要最小化它的不确定性（也就是出错率）到小于 1%，需要把你的每轮测试迭代运行 750ms。而 1ms 定时器时只需要每轮运行 50ms 就可以达到同样的置信度。

但是，这只是单独的一个例子。要确保把异常因素排除，你需要大量的样本来平均化。你还会想要知道最差样本有多慢，最好的样本有多快，以及最好和最差情况之间的偏离度有多大，等等。你需要知道的不仅仅是一个告诉你某个东西跑得有多快的数字，还需要得到某个可以计量的测量值告诉你这个数字的可信度有多高。

还有，你可能会想要把不同的技术（以及其他方面）组合起来，以得到所有可能方法的最佳平衡。

这仅仅是个开始。如果你过去进行性能测试的方法比我刚才提出的还要不正式的话，好吧，那么可以说你完全不知道：正确的性能测试。

6.1.2 Benchmark.js

任何有意义且可靠的性能测试都应该基于统计学上合理的实践。此处并不打算撰写一章关于统计学的内容，所以我要和如下术语挥手作别：标准差、方差、误差幅度。如果你不知道这些术语的意思——我回大学上了一门统计学课程，但对这些还是有点糊涂——那么实际上你还不够资格编写自己的性能测试逻辑。

幸运的是，像 John-David Dalton 和 Mathias Bynens 这样的聪明人了解这些概念，并编写了一个统计学上有效的性能测试工具，名为 Benchmark.js（http://benchmarkjs.com/）。因此，对于这个悬而未决的问题，我的答案就是："使用这个工具就好了。"

我并不打算复述他们的整个文档来介绍 Benchmark.js 如何运作。他们的 API 很不错，你应该读一读。还有一些很棒的文章介绍了更多的细节和方法，比如这里（http://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks）和这里（http://monsur.hosa.in/2012/12/11/benchmarksjs.html）。

但为了简单展示一下，下面介绍应该如何使用 Benchmark.js 来运行一个快速的性能测试：

```
function foo() {
    // 要测试的运算
}

var bench = new Benchmark(
    "foo test",             // 测试名称
    foo,                    // 要测试的函数(也即内容)
```

```
    {
        // ..              // 可选的额外选项(参见文档)
    }
);

bench.hz;                 // 每秒运算数
bench.stats.moe;          // 出错边界
bench.stats.variance;     // 样本方差
// ..
```

除了这里我们介绍的一点内容，关于 Benchmark.js 的使用还有很多要学的。但是，关键在于它处理了为给定的一段 JavaScript 代码建立公平、可靠、有效的性能测试的所有复杂性。如果你想要对你的代码进行功能测试和性能测试，这个库应该最优先考虑。

这里我们展示了测试一个像 X 这样的单个运算的使用方法，不过很可能你还想要比较 X 和 Y。通过在一个 suite（Benchmark.js 组织特性）中建立两个不同的测试很容易做到这一点。然后，可以依次运行它们，比较统计结果，得出结论，判断 X 和 Y 哪个更快。

Benchmark.js 当然可以用在浏览器中测试 JavaScript（参见 6.3 节），它也可以在非浏览器环境中运行（Node.js 等）。

Benchmark.js 有一个很大程度上还未开发的潜在用例，就是你可以将其用于开发或测试环境中，针对应用中 JavaScript 的关键路径部分运行自动性能回归测试。这和你可能在部署之前运行的单元测试套件类似，你也可以与之前的版本进行性能测试比较，以监控应用性能是提高了还是降低了。

setup/teardown

在前面的代码片段中，我们忽略了“额外选项”`{ .. }` 对象。这里有两个选项是我们应该讨论的：`setup` 和 `teardown`。

这两个选项使你可以定义在每个测试之前和之后调用的函数。

有一点非常重要，一定要理解，`setup` 和 `teardown` 代码不会在每个测试迭代都运行。最好的理解方法是，想像有一个外层循环（一轮一轮循环）还有一个内层循环（一个测试一个测试循环）。`setup` 和 `teardown` 在每次外层循环（轮）的开始和结束处运行，而不是在内层循环中。

为什么这一点很重要呢？设想你有一个像这样的测试用例：

```
a = a + "w";
b = a.charAt( 1 );
```

然后，你建立了测试 `setup` 如下：

```
var a = "x";
```

你的目的可能是确保每个测试迭代开始的 a 值都是 "x"。但并不是这样！只有在每一轮测试开始时 a 值为 "x"，然后重复 + "w" 链接运算会使得 a 值越来越长，即使你只是访问了位置 1 处的字符 "w"。

对某个东西，比如 DOM，执行产生副作用的操作的时候，比如附加一个子元素，常常会刺伤你。你可能认为你的父元素每次都清空了，但是，实际上它被附加了很多元素，这可能会严重影响测试结果。

6.2 环境为王

对特定的性能测试来说，不要忘了检查测试环境，特别是比较任务 X 和 Y 这样的比对测试。仅仅因为你的测试显示 X 比 Y 快，并不能说明结论 X 比 Y 快就有实际的意义。

举例来说，假定你的性能测试表明 X 运算每秒可以运行 10 000 000 次，而 Y 每秒运行 8 000 000 次。你可以说 Y 比 X 慢了 20%。数学上这是正确的，但这个断言并不像你想象的那么有意义。

让我们更认真地思考这个结果：每秒 10 000 000 次运算就是每毫秒 10 000 次运算，每微妙 10 次。换句话说，单次运算需要 0.1μs，也就是 100ns。很难理解 100ns 到底有多么短。作为对比，据说人类的眼睛通常无法分辨 100ms 以下的事件，这要比 X 运算速度的 100ns 慢一百万倍了。

即使最近的科学研究表明可能大脑可以处理的最快速度是 13ms（大约是以前结论的 8 倍），这意味着 X 的运算速度仍然是人类大脑捕获一个独立的事件发生速度的 125 000 倍。X 真的非常非常快。

不过更重要的是，我们来讨论一下 X 和 Y 的区别，即每秒 2 000 000 次运算差距的区别。如果 X 需要 100ns，而 Y 需要 80ns，那么差别就是 20ns，这在最好情况下也只是人类大脑所能感知到的最小间隙的 65 万分之一。

我要说的是什么呢？这些性能差别无所谓，完全无所谓！

但是稍等，如果这些运算将要连续运行很多次呢？那么这个差别就会累加起来，对不对？

好吧，那我们要问的就是，这个运算 X 要一个接一个地反复运行多次的可能性有多大呢，得运行 650 000 次才能有一点希望让人类感知到。更可能的情况是，它得在一个紧密循环里运行 5 000 000~10 000 000 次才有意义。

你脑子里的计算机科学家可能抗议说，这是可能的；但你脑子里那个现实的的你会更大声说还是应该检查一下这个可能性到底有多大。即使在很少见的情况下是有意义的，但在绝大数情况下它却是无关紧要的。

对于微小运算的绝大多数测试结果，比如 ++x 对比 x++ 的迷思，像出于性能考虑应该用 X 代替 Y 这样的结论都是不成立的。

引擎优化

你无法可靠地推断，如果在你的独立测试中 X 比 Y 要快上 10μs，就意味着 X 总是比 Y 要快，就应该总是使用 X。性能并不是这样发挥效力的。它要比这复杂得多。

举例来说，设想一下（纯粹假设）你对某些微观性能行为进行了测试，比如这样的比较：

```
var twelve = "12";
var foo = "foo";

// 测试1
var X1 = parseInt( twelve );
var X2 = parseInt( foo );

// 测试2
var Y1 = Number( twelve );
var Y2 = Number( foo );
```

如果理解与 Number(..) 相比 parseInt(..) 做了些什么，你可能会凭直觉以为 parseInt(..) 做的工作可能更多，特别是在 foo 用例下。或者你可能会直觉认为它们的工作量在 foo 用例下应该相同，两个都应该能够在第一个字符 f 处停止。

哪种直觉是正确的呢？老实说，我不知道。不过，对于我举的这个例子，哪个判断正确并不重要。测试结果可能是什么？这里我再次单纯假设，并没有实际进行过测试，也没必要那么做。

让我们假装测试结果返回的是从统计上来说完全相同的 X 和 Y。那么你能够确定你关于 f 字符的直觉判断是否正确吗？不能。

在我们假设的情况下，引擎可能会识别出变量 twelve 和 foo 在每个测试中只被使用了一次，因此它可能会决定把这些值在线化。那么它就能识别出 Number("12") 可以直接替换为 12。对于 parseInt(..)，它可能会得出同样的结论，也可能不会。

也有可能引擎的死代码启发式去除算法可能会参与进来，它可能意识到变量 X 和 Y 并没有被使用，因此将其标识为无关紧要的，故而在整个测试中实际上什么事情都没有做。

所有这些都只是根据单个测试所做的假设的思路。现代引擎要比我们凭直觉进行的推导复杂得多。它们会实现各种技巧，比如跟踪记录代码在一小段时期内或针对特别有限的输入集的行为。

如果引擎由于固定输入进行了某种优化，而在真实程序中的输入更加多样化，对优化决策影响很大（甚至完全没有）呢？或者如果引擎看到测试由性能工具运行了数万次而进行优

化，但是在真实程序中只会运行数百次，而这种情况下引擎认为完全不值得优化呢？

我们设想的所有这些优化可能性在受限的测试中都有可能发生，而且在更复杂的程序中（出于各种各样的原因），引擎可能不会进行这样的优化。也可能恰恰相反，引擎可能不会优化这样无关紧要的代码，但是在系统已经在运行更复杂的程序时可能会倾向于激进的优化。

这里我要说明的就是，你真的不能精确知道底下到底发生了什么。你能进行的所有猜想和假设对于这样的决策不会有任何实际的影响。

这是不是意味着无法真正进行任何有用的测试呢？绝对不是！

这可以归结为一点，测试不真实的代码只能得出不真实的结论。如果有实际可能的话，你应该测试实际的而非无关紧要的代码，测试条件与你期望的真实情况越接近越好。只有这样得出的结果才有可能接近事实。

像 ++x 对比 x++ 这样的微观性能测试结果为虚假的可能性相当高，可能我们最好就假定它们是假的。

6.3 jsPerf.com

尽管在所有的 JavaScript 运行环境下，Benchmark.js 都可用于测试代码的性能，但有一点一定要强调，如果你想要得到可靠的测试结论的话，就需要在很多不同的环境（桌面浏览器、移动设备，等等）中测试汇集测试结果。

比如，针对同样的测试高端桌面机器的性能很可能和智能手机上 Chrome 移动设备完全不同。而电量充足的智能手机上的结果可能也和同一个智能手机但电量只有 2% 时完全不同，因为这时候设备将会开始关闭无线模块和处理器。

如果想要在不止一个环境下得出像“X 比 Y 快”这样的有意义的结论成立，那你需要在尽可能多的真实环境下进行实际测试。仅仅因为在 Chrome 上某个 X 运算比 Y 快并不意味着这在所有的浏览器中都成立。当然你可能还想要交叉引用多个浏览器上的测试运行结果，并有用户的图形展示。

有一个很棒的网站正是因这样的需求而诞生的，名为 jsPerf（http://jsperf.com）。它使用我们前面介绍的 Benchmark.js 库来运行统计上精确可靠的测试，并把测试结果放在一个公开可得的 URL 上，你可以把这个 URL 转发给别人。

每次测试运行的时候，测试结果就会被收集并持久化，累积的测试结果会被图形化，并展示到一个页面上以供查看。

在这个网站上创建测试的时候，开始需要先填写两个测试用例，但是你可以按需增添加任意多的测试。你还可以设定在每个测试循环开始时运行的 `setup` 代码，以及每个测试循环结束时运行的 `teardown` 代码。

可以通过一个技巧实现只用一个测试用例（如果需要测试单个方法的性能，而不需要对比的话），就是在首次创建的时候在第二个测试输入框填入占位符文字，然后编辑测试并把第二个测试清空，也就是删除了它。你总是可以在以后增加新的测试用例。

可以定义初始页面设置（导入库、定义辅助工具函数、声明变量，等等）。还有选项可以在需要的时候定义 `setup` 和 `teardown` 行为，参见 6.1.2 节。

完整性检查

jsPerf 是一个很好的资源，但认真分析的话，出于本章之前列出的多种原因，公开发布的测试中有大量是有缺陷或无意义的。

考虑：

```
// 用例1
var x = [];
for (var i=0; i<10; i++) {
    x[i] = "x";
}

// 用例2
var x = [];
for (var i=0; i<10; i++) {
    x[x.length] = "x";
}

// 用例3
var x = [];
for (var i=0; i<10; i++) {
    x.push( "x" );
}
```

这个测试场景的一些需要思考的现象如下。

- 对开发者来说，极常见的情况是：把自己的循环放入测试用例，却忘了 Benchmark.js 已经实现了你所需的全部重复。非常有可能这些情况下的 `for` 循环完全是不必要的噪音。
- 每个测试用例中 `x` 的声明和初始化可能是不必要的。回忆一下之前的内容，如果 `x = []` 放在 `setup` 代码中，它并不会在每个测试迭代之前实际运行，而是只在每轮测试之前运行一次。这意味着 `x` 将会持续增长到非常大，而不是 `for` 循环中暗示的大小——`10`。

所以，其目的是为了确定测试只局限于 JavaScript 引擎如何处理小数组（大小为 10）吗？目的可能是这样，而如果确实是的话，你必须考虑这是否过多关注了微秒的内部实现细节。

另一方面，测试的目的是否包含数组实际上增加到非常大之后的环境？与真实使用情况相比，JavaScript 处理大数组的行为是否适当和精确呢？

- 目的是否是找出 `x.length` 或 `x.push(..)` 对向数组 `x` 添加内容的操作的性能的影响有多大？好吧，这可能是有效的测试目标。但话说回来，`push(..)` 是一个函数调用，所以它当然要比 `[..]` 访问慢。可以证明，用例 1 和 2 要比用例 3 公平得多。

以下是另一个例子，展示了典型的不同类型对比的缺陷：

```
// 用例1
var x = ["John","Albert","Sue","Frank","Bob"];
x.sort();

// 用例2
var x = ["John","Albert","Sue","Frank","Bob"];
x.sort( function mySort(a,b){
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
} );
```

这里，很明显测试目标是找出自定义的比较函数 `mySort(..)` 比内建默认比较函数慢多少。但是，通过把函数 `mySort(..)` 指定为在线函数表达式，你已经创建了一个不公平 / 虚假的测试。这里，第二个用例中测试的不只是用户自定义 JavaScript 函数，它还在每个迭代中创建了一个新的函数表达式。

如果运行一个类似的测试，但是将其更新为将创建一个在线函数表达式与使用预先定义好的函数对比，如果发现在线函数表达式创建版本要慢 2% ~ 20%，你会不会感到吃惊？！

除非你这个测试特意要考虑在线函数表达式创建的代价，否则更好更公平的测试就是将 `mySort(..)` 的声明放在页面 `setup` 中——不要把它放在测试 `setup` 中，因为那会在每一轮不必要地重新声明——只需要在测试用例中通过名字引用它：`x.sort(mySort)`。

根据前面的例子，还有一个陷阱是隐式地给一个测试用例避免或添加额外的工作，从而导致“拿苹果与橘子对比”的场景：

```
// 用例1
var x = [12,-14,0,3,18,0,2.9];
x.sort();

// 用例2
var x = [12,-14,0,3,18,0,2.9];
```

```
x.sort( function mySort(a,b){
    return a - b;
} );
```

除了前面提到的在线函数表达式陷阱，第二个用例的 `mySort(..)` 可以工作。因为你给它提供的是数字，但如果是字符串的话就会失败。第一个用例不会抛出错误，但它的行为不同了，输出结果也不同！这应该很明显，但是两个测试用例产生不同的输出几乎肯定会使整个测试变得无效！

不过，在这种情况下，除了不同的输出之外，内建的比较函数 `sort(..)` 实际上做了 `mySort()` 没有做的额外工作，包括内建的那个把比较值强制类型转化为字符串并进行字典序比较。第一段代码结果为 `[-14, 0, 0, 12, 18, 2.9, 3]`，而第二段代码结果（基于目标而言可能更精确）为 `[-14, 0, 0, 2.9, 3, 12, 18]`。

所以，这个测试是不公平的，因为对于不同的用例，它并没有做完全相同的事情。你得到的任何结果都是虚假的。

同样的陷阱可能会更加不易察觉：

```
// 用例1
var x = false;
var y = x ? 1 : 2;

// 用例2
var x;
var y = x ? 1 : 2;
```

这里，目的可能是测试对 Boolean 值进行强制类型转换对性能的冲击：如果 x 表达式并不是 Boolean 运算符，`? :` 就会进行强制类型转换（参见本书的“类型和语法”部分）。所以，你显然可以接受如下事实：第二个用例中有额外的类型转换工作要做。

那么不易察觉的问题是什么呢？在第一个用例中设定了 x 的值，而在另一个中则没有设定，所以实际上你在第一个用例中做了在第二个用例中没有做的事。要消除这个潜在的（虽然很小的）影响，可以试着这样：

```
// 用例1
var x = false;
var y = x ? 1 : 2;

// 用例2
var x = undefined;
var y = x ? 1 : 2;
```

现在两种情况下都有赋值语句了。所以你想要测试的内容（有无对 x 的类型转换）很可能就更加精确地被独立出来并被测试到了。

6.4 写好测试

我看看能不能讲清楚我在这里想要说明的更重要的一点。

要写好测试，需要认真分析和思考两个测试用例之间有什么区别，以及这些区别是有意还是无意的。

有意的区别当然是正常的，没有问题，可我们太容易造成会扭曲结果的无意的区别。你需要非常小心才能避免这样的扭曲。还有，你可能有意造成某个区别，但是，对于这个测试的其他人来说，你的这个意图可能不是那么明显，所以他们可能会错误地怀疑（或信任！）你的测试。如何解决这样的问题呢？

编写更好更清晰的测试。但还有，花一些时间来编写文档（使用 jsPerf.com 上的 Description 字段和 / 或代码注释）精确表达你的测试目的，甚至对于那些微小的细节也要如此。找出那些有意的区别，这会帮助别人和未来的你更好地识别出那些可能扭曲测试结果的无意区别。

通过在页面或测试 setup 设置中预先声明把与测试无关的事情独立出来，使它们移出测试计时的部分。

不要试图窄化到真实代码的微小片段，以及脱离上下文而只测量这一小部分的性能，因为包含更大（仍然有意义的）上下文时功能测试和性能测试才会更好。这些测试可能也会运行得慢一点，这意味着环境中发现的任何差异都更有意义。

6.5 微性能

到目前为止，我们一直在围绕各种微性能问题讨论，并始终认为沉迷于此是不可取的。现在我要花费一点时间直面这个问题。

在考虑对代码进行性能测试时，你应该习惯的第一件事情就是你所写的代码并不总是引擎真正运行的代码。在第 1 章讨论编译器语句重排序问题时，我们简单介绍过这个问题。不过这里要说的是，有时候编译器可能会决定执行与你所写的不同的代码，不只是顺序不同，实际内容也会不同。

来考虑下面这段代码：

```
var foo = 41;

(function(){
    (function(){
        (function(baz){
            var bar = foo + baz;
            // ..
        })(1);
```