代码覆盖率

“当我们在说：我们代码的覆盖率为100%的时候，我们在说什么”

演讲者：feixiang

什么是代码覆盖率呢？

代码覆盖率 = 代码的覆盖程度，一种度量方式。

是不是经常看到这两个图标？

Evan You
@youyuxi

正在关注

Vue.js 0.11 unit test coverage... finally

File	Statements	
src/	100.00%	(298 / 298)
src/api/	100.00%	(324 / 324)
src/compile/	100.00%	(278 / 278)
src/directives/	100.00%	(504 / 504)
src/directives/model/	100.00%	(171 / 171)
src/filters/	100.00%	(75 / 75)
src/instance/	100.00%	(210 / 210)
src/observer/	100.00%	(160 / 160)
src/parse/	100.00%	(368 / 368)
src/transition/	100.00%	(147 / 147)
src/util/	100.00%	(257 / 257)

看看例子

```
function far(n) {
    if (n > 100) {
        return true;
    }
    else {
        return false;
    }
}
```


```
    assert.equal(far(100),false);
    assert.equal(far(200),true);
```

看看一个覆盖率不全的例子

passes: *1* failures: *0* duration: *0.02*s 100%

test

✓ should return false

Blanket.js results	Covered/Total Smts.	Coverage (%)
1. http://192.168.1.101:8080/runner/test.js	3/4	75 %

```
1  function far(n) {
2     if (n > 100) {
3             return true;
4     }
5     else {
6             return false;
7     }
8  }
```

Global total	3/4	75 %

再来看看一个覆盖率100%的例子

test

✓ should return false when pass 100

```
100
```

✓ should return true when pass 200

```
200
```

passes: *2* failures: *0* duration: *0.02*s 100%

Blanket.js results	Covered/Total Smts.	Coverage (%)
1. http://192.168.1.101:8080/runner/test.js	4/4	100 %

```
1  function far(n) {
2     if (n > 100) {
3             return true;
4     }
5     else {
6             return false;
7     }
8  }
```

webcache

passes: *12* failures: *0* duration: *2.26*s 100%

webcache

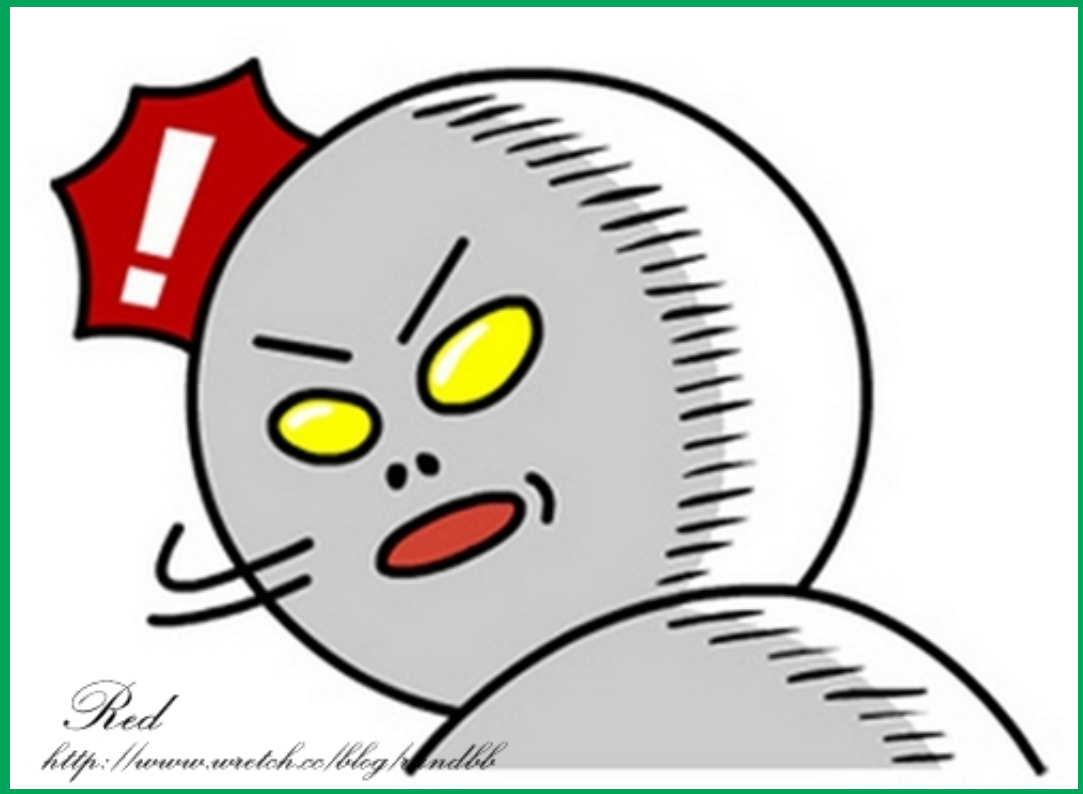
- ✓ 初始化为空
- ✓ put 方法
- ✓ 空间溢出，清除最早数据
- ✓ 不存在的值返回 undefined
- ✓ 最近访问的键值提前
- ✓ 添加新键值，抛弃最早引用的值
- ✓ 最近更新的键值提前
- ✓ 更新的键值不存在，不做任何操作
- ✓ 更新的键值存在则修改键值并提前
- ✓ 创建不同尺寸缓存时自动清空
- ✓ 超时丢弃 334ms
- ✓ 性能测试 1892ms

Blanket.js results	Covered/Total Smts.	Coverage (%)
1. http://192.168.1.101:8080/runner/webcache.js	99/107	92.52 %
Global total	99/107	92.52 %

为什么要关注代码覆盖率

- 代码覆盖率（Code Coverage）是反映测试用例对被测软件覆盖程度的重要指标，也是衡量测试工作进展情况的重要指标。
- 通过代码覆盖率结果，能够比较直观的了解到哪些代码未被测试，哪些分支未被覆盖，进而补充相应的测试案例。
- 代码覆盖率给程序员和测试人员以信心。

今天我们不谈这些工程上的话题。



工程化的东西需要自己在实践中感受痛苦，再去领悟。

只是单纯来说说代码覆盖率背后的原理

举个栗子

- 看看最简单的情况

```
function far(n){
  return 100;
}
```

- 如果想要统计far函数哪些语句被覆盖到了
- 最直观的方法就是给这个函数加一个列表来保存哪些语句被执行了，然后在每条语句前都往这些列表添加上当前的行号，写出来是这样

```
cover[1] = true;function far(n){
cover[2] = true;    return 100;
cover[3] = true;}
```

- Done!!!

原理是不是很简单

我们要做的就是在跑程序之前，把所有JS源码每一行前加上代码行数统计的代码

那怎么来做呢？

好像还是有点难度啊！

- 手动暴力的通过字符串拼接或者正则表达式替换很容易出错
- 所有源码写在一行怎么搞？
- `if (a || b || c || d) return;` javascript的惰性计算的特性，这样的语句怎么搞？
- 马达，不行了吗？

今天的另一个主角

JavaScript AST

AST 又名抽象语法树，是将源码解析成一种特定的树状结构。

Esprima (../index.html) **Demo** **Project** **Documentation (../doc/index.html)**

Parser produces the (beautiful) syntax tree

```
1  // Life, Universe, and Everything
2  var answer = 6 * 7;
3
```

Syntax Tree Tokens


```
{
    "type": "Program",
    "body": [
        {
            "type": "VariableDeclaration",
            "declarations": [
                {
                    "type": "VariableDeclarat
                    "id": {
                        "type": "Identifier",
                        "name": "answer"
                    },
                    "init": {
                        "type": "BinaryExpres
                        "operator": "*",
                        "left": {
                            "type": "Literal"
                            "value": 6,
                            "raw": "6"
                        },
                        "right": {
                            "type": "Literal"
                            "value": 7,
                            "raw": "7"
                        }
                    }
```


No error

Syntax node location info (start, end):

☐ **Index-based range**

为什么说它？

它能帮我们更轻松的生成可追踪的代码

before

```
var a = 1 || 2;
var b = [a]
```

after

```
var cover = (Function('return this'))();
//省略部分结构体信息
cover.s['1']++;
var a = (cover.b['1'][0]++, 1) || (cover.b['1'][1]++, 2);
cover.s['2']++;
var b = [a];
```

基本工作原理

- HTML => DOM => HTML
- JavaScript => AST => JavaScript

常见的生成JavaScript抽象语法树的工具

AST在JavaScript中常见的应用

- 压缩
- 很精准的代码格式化
- 语法提示，自动补全
- 提取注释 JSDOC
- 其他语言生成JavaScript代码
- 代码追踪:
    - 统计每一行语句执行速度
    - 统计每一行代码执行前后所占的内存
    - 代码执行覆盖率

```
Pystone(1.1) time for 50000 passes = 2.48
This machine benchmarks at 20161.3 pystones/second
Wrote profile results to pystone.py.lprof
Timer unit: 1e-06 s

File: pystone.py
Function: Proc2 at line 149
Total time: 0.606656 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def Proc2(IntParIO):
   151     50000        82003      1.6     13.5      IntLoc = IntParIO + 10
   152     50000        63162      1.3     10.4      while 1:
   153     50000        69065      1.4     11.4          if Char1Glob == 'A':
   154     50000        66354      1.3     10.9              IntLoc = IntLoc - 1
   155     50000        67263      1.3     11.1              IntParIO = IntLoc - IntGlo
   156     50000        65494      1.3     10.8              EnumLoc = Ident1
   157     50000        68001      1.4     11.2          if EnumLoc == Ident1:
   158     50000        63739      1.3     10.5              break
   159     50000        61575      1.2     10.1      return IntParIO
```

执行过程中活动的代码覆盖

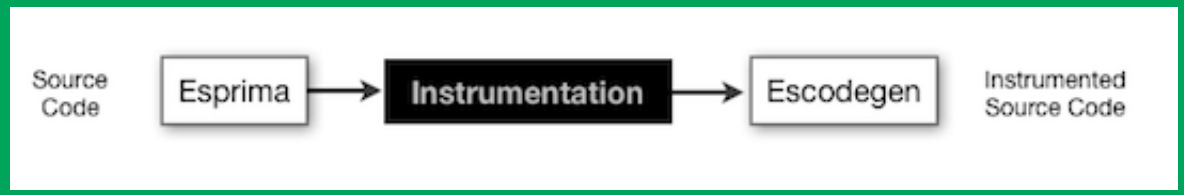
Tracker

继续回到我们的主题

代码执行覆盖率

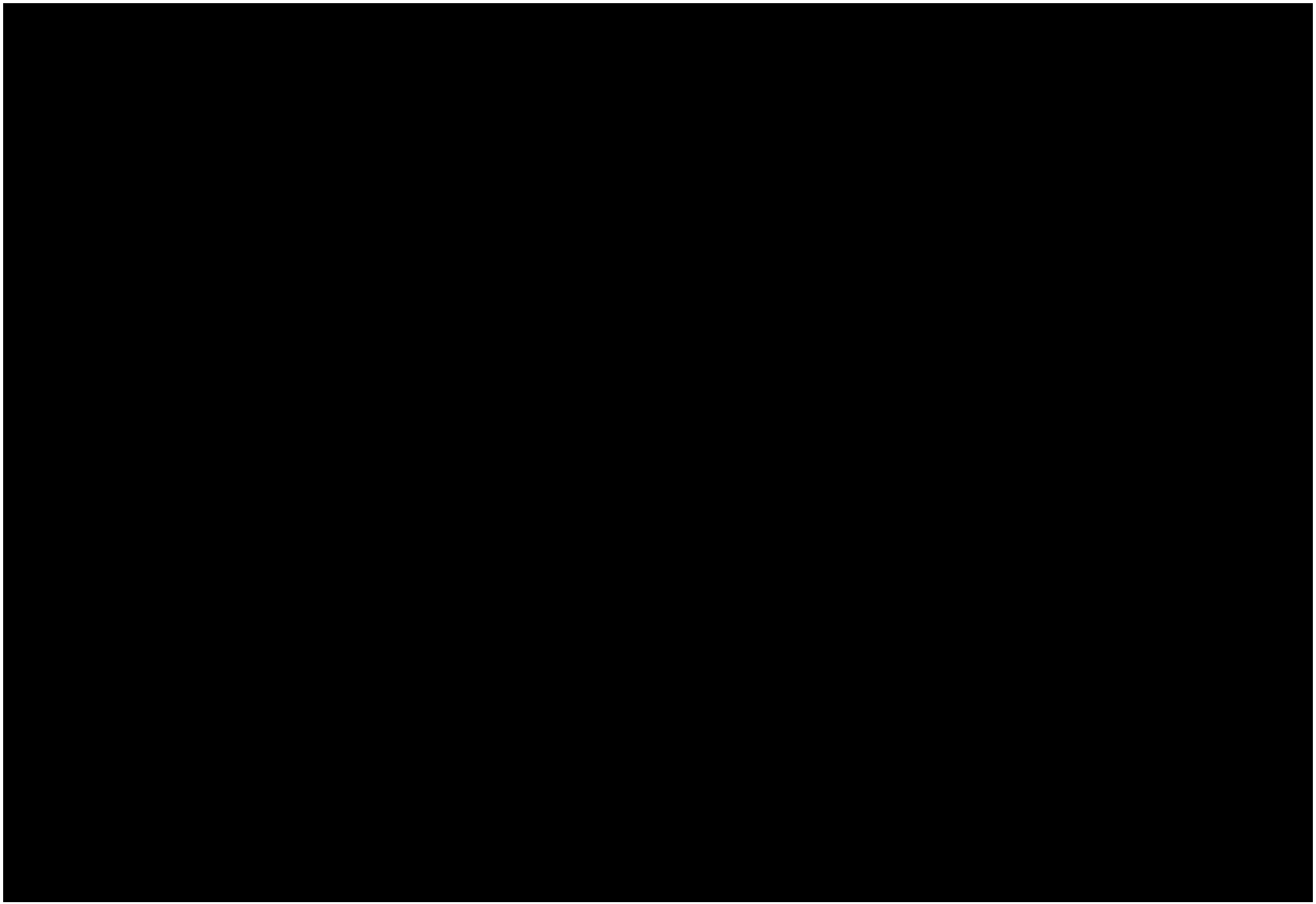
现成的工具

- jscoverage
- blanket
- istanbul



- 装载源码
- 用Esprima生成AST，针对各种语法节点，修改语法树
- 用Escodegen把语法树还原成可追踪源码
- 运行可追踪的源码，结束后生成报告。

演示istanbul



Powered By nodePPT v0.9.5