剖析V8底层引擎　探究JS优化机制

Haitao Feng

Intel Open Source Technology Center

August 11, 2013

Outline

Type Feedback

- Modern JavaScript engines collect type information for primitive operation or create types for objects they run the JavaScript codes. Those type information are used by inline caching or adaptive optimizing compiler.
- In V8, each JavaScript object is assigned with an unique map (the created type). Map describes the object layout and makes object property access and method invocation efficient. Map is created or transitioned when a new property added into an object.

Inline Cache (IC)

V8 generates inline cache stub function call for primitive operation, object property access and method invocation in the JavaScript code in the non-optimizing compiler. The map information could be inlined in the corresponding stub calls. The IC state could be:

- Monomorphic (one map seen from type feedback)
- Polymorphic (2-4 maps seen from type feedback)
- Megamorphic (>4 maps seen from type feedback)

V8 modifies the generated code to use the correct IC stub when there is a state transition.

Adaptive Optimization

V8 adaptively optimizes the hot functions or hot loops in the JavaScript code. It does the following steps:

- Instrument the non-optimizing code for profiling
- Collect type information by mining the IC stubs
- Generate compiler IR (Intermediate Language)
- Optimize compiler IR by using classic compiler technology
- Generate machine code

When the future type information violates the type feedback in the optimizing code, V8 de-optimizes the function to the non-optimizing code and re-collect type feedback.

Practice 1. Be aware of objects' layout

- Initialize all the properties in the constructor
- Initialize all the properties in the same order
- Avoid object layout modification if necessary

And V8 could generate efficient codes in IC and optimizing compiler.

Example 1. Be aware of objects' layout

In-efficient code:

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

for (var i = 1; i < N; i++) {
  var p = new Point(i, i);
  if (i % 2) p.z = i/2;
  ...
}
```

Efficient code:

```
function Point2(x, y) {
  this.x = x;
  this.y = y;
}

function Point3(x, y, z) {
  this.x = x;
  this.y = y;
  this.z = z;
}

for (var i = 1; i < N; i++) {
  var p = (i % 2) ?
    new Point3(i, i, i/2) :
    new Point2(i, i);
  ...
}
```

Practice 2. Get hot traces faster

- Be aware of non-optimizing JavaScript language features
- Abstract hot traces into functions
- Use function local variables instead of global variables if possible

V8 could generate efficient codes if you do not put your hot traces into a non-optimizing scope, for example, inside "with" statement.

Example 2. Get hot traces faster

In-efficient code:

```
try {
  for (var i = 1; i < N; i++) {
    var p = (i % 2) ?
      new Point3(i, i, i/2) :
      new Point2(i, i);
    ...
  }
} catch(e) {
  ...
}
```

Efficient code:

```
function initialize(N) {
  for (var i = 1; i < N; i++) {
    var p = (i % 2) ?
      new Point3(i, i, i/2) :
      new Point2(i, i);
    ...
  }
}

try {
  initialize(N);
} catch(e) {
  ...
}
```

Practice 3. Measure performance consistently

- Use Chrome devtools to profiler the JavaScript code
- Use V8 builtin tools:
  - Timeline plot could give an overview of the JavaScript execution
  - Statistical profile could give detailed information at function level
  - Low level profiler could give detailed information at assembly level

Those tools help you understand how your JavaScript code executed by the engine.

Example 3. Use the V8 on-line profviz tool

The on-line profviz tool includes the timeline plot and statistical profile. For example, to analyze string-base64.js from SunSpider:

- Run ./out/x64.release/d8 −−log-timer-events −−prof SunSpider/tests/sunspider-1.0/string-base64.js −−logfile string-base64.txt
- Upload string-base64.txt to `http://v8.googlecode.com/svn/branches/bleeding_edge/tools/profviz/profviz.html`