Project 1: JSProf

Bruce Spang

October 21, 2013

Approach

JSProf is divided into two main components: a program profiler and a statistics reporter. The profiler instruments a user-supplied program in order to maintain an accurate call graph while the program executes. The reporter manipulates the call graph to figure out information about how the program executes.

Instrumenting the program source was the trickiest part of the project. Once I had an accurate call graph, reporting just involved a lot of iterating over the call graph and manipulating the DOM.

I used UglifyJS [9] to parse and transform the AST, and wrote the instrumentation logic by hand. There were two ways to do this: I could have either inserted profiling calls in function definitions, or inserted profiling calls at every call site. Call-site profiling is simple and would allows the system to profile calls to other libraries. Unfortunately, it doesn't work when instrumented callbacks are executed by non-instrumented libraries (e.g. setInterval). This results in calls in a callback being embedded in unexpected nodes in the call graph, which is confusing. Because javascript libraries love callbacks, I decided to implement definition-site instrumentation instead.

System Design

The JSProf profiler consists of a program instrumentor and a runtime statistics collector. The JSProf reporter consists of a set of functions to extract statistics from the call graph, and a front-end for visualization.

Profiler

See src/prof.js

The profiler uses UglifyJS [9] to instrument the AST with `enter()` and `exit()` calls. At runtime, the `enter` and `exit` calls update a linked data structure that represents the program's call graph. Every node of this data structure corresponds to a stack frame in the program execution, and includes timing details and all calls to child functions.

The instrumentor traverses the AST and inserts instrumentation calls. If it sees a function definition, it inserts `enter` at the beginning of the body and `exit` at the end of the body. If it sees an exit node (either a `return` or a `throw`), it inserts an `exit` call.

There are a few subtle parts of the instrumentor. The system can't just insert an exit call before all return statements, because they may have lots of computation (e.g. `return reallyLongFunction()`). It also needs to properly handle exceptions.

To avoid the issue with return statements, the system stores the result of the return expression in a new variable, inserts an exit call, and then returns the new variable. It wraps all this in an immediately evaluated anonymous function so that the new variable doesn't override some other program variable.

Handling exceptions properly is tricky, since program flow can jump any number of levels up the stack. To deal with them, the system wraps all function calls in a try/catch block. When the program throws an exception, each try/catch block calls `exit` before propagating the exception up the stack. Once the program flow hits a catch block in the original program, the system re-enters the current function and continues executing.

Reporter

See src/app.js, src/visualizer.js

The profiler system includes a reporter that visualizes the program execution. Every 250ms, it traverses the call graph and updates a call graph visualization, a set of statistics about each function, and information about hot paths.

The main advantage to this method is that it's very flexible: it is possible to extract new statistics from the call graph without modifying the runtime system. While traversing the entire call graph every 250ms could be slow for very large programs, it seemed fast enough in my testing. If reporter speed was an issue, it may be better to compute all the necessary statistics at runtime (e.g. every time a function is called, increment the frequency).

Results

This design seems very good for small javascript programs. It is functional, fast, and the visualizations are useable. While developing the system, I was able to find bugs in the instrumentation logic by examining the call graph and bugs in the timing logic by examining the timing reports. Being able to see a live visualization of program behavior, and being able to update that behavior on the fly is very useful for development.

If this system was going to be used primarily for larger javascript applications, I would have made different design decisions. I probably would either have written a node.js script to instrument the code statically, or modify a script loader to instrument libraries automatically. I would have made the visualization component write to an overlay, so that it could be embedded onto

existing websites. Fortunately, none of these changes would involve significant code modifications.

It is annoying that the profiler cannot instrument library functions. It may be possible to combine call-site and definition-site instrumentation in order to profile them. I'm not sure how to do that correctly without generating massive amounts of code.

References

[1] binary trees benchmark. http://benchmarksgame.alioth.debian.org/u32/program.php?test=binarytrees&lang=v8&id=1.

[2] Bootstrap. http://getbootstrap.com/.

[3] Google code prettify. https://code.google.com/p/google-code-prettify/.

[4] jquery. http://jquery.com/.

[5] jquery.sparklines. http://omnipotent.net/jquery.sparkline/.

[6] linedtextarea. https://github.com/aw20/JQueryLinedText.

[7] n-body benchmark. http://benchmarksgame.alioth.debian.org/u32/program.php?test=nbody&lang=v8&id=2.

[8] reverse complement benchmark. http://benchmarksgame.alioth.debian.org/u32/program.php?test=revcomp&lang=v8&id=4.

[9] Uglifyjs2. https://github.com/mishoo/UglifyJS2.