JavaScript Profiler

Roy Adams and David Belinger

October 18, 2013

1 Overview

Our profiler takes as input a block of JavaScript code as a string, adds instrumentation code to all user defined functions, and then runs the code to collect profile data. The profiler may be called either as an imported library or via a simple HTML interface that allows the user to paste in JavaScript code he or she wants to profile. During a run of the input script, the profiler collects timing, call-path, and call frequency information which can then be accessed as a formatted output.

2 Design and Functionality

Our approach to the problem involves two steps. First, the code to be profiled is parsed using the esprima parser (¡esprima url¿) which returns a syntax tree including line numbers and colums. This syntax tree is then traversed recursively and all function definitions, return statements, timing functions, and eval statements are detected and modified to include instrumentation code. Modifications are made directly to the syntax tree and so when the modifications are complete, the modified syntax tree is simply used to generate new code using the escodegen package(¡escodegen url¿). In the second main step, this code is evaluated and profile information is gathered including call counts, function timing, call edges, and call paths. This second step can be done within our standalone profiler, profiler.html, or can be done within the user's application by replacing the code to be profiled with the instrumented code and importing profiling.js. In the first case, profile information is reported in the main page. In the second case, a popup window is opened which reports the same information.

2.1 Code Modification

To modify the syntax tree we traverse it recursively, applying a modification function at each node. All function declarations and function expressions are modified by adding code to the beginning and end of each function. All return statements are modified to run some instrumentation code prior to returning. In this way all user defined functions will execute profiling code regardless of how or from where they are called. As part of the instrumentation code, a function name is passed. If the user declares the function with a name, this name is retrieved during parsing and used when reporting profile information, otherwise the function is named anon_line_(line_num)_col_(col_num) where (line_num) and (col_num) are the line and column where the function is defined. An advantage of this strategy is that function statistics are agregated regardless of whether the function is assigned to a new variable. A downside is that the programmer must explicitly give the functions names is he/she want those names to show up in the profile. Also caught during parsing are

any calls to setTimeout and setInterval. These are modified to indicate that the callback function passed is being called from the a timing function and indicate the line and column number of the timing function call.

2.2 Profiling

The profiler is defined completely in profiling.ts (which typescript then compiles to profiling.js) and consists of three main classes: ProfileFromSource, Profiler, and Profile. The top level class is ProfileFromSource which serves as an interface to the user. It has three main functions, a constructor which modifies the input code, startUp which runs the modified code and collects profile information, and getReport which compiles the profile information and returns a formatted string. Underneath this is the Profiler class which does most of the leg-work in our profiler. The Profiler class stores and maintains all global profile objects such as the call stack, call paths, and a list of defined functions. Finally, the Profile class maintains information about a specific function such as average time spent in the function and number of invocations. When a user defined function, call it Foo, is called it calls the Profiler member function getProfile which searches the list of profile functions already created for a matching profile and creates a new profile if one is not found. Foo then calls Profile.start which pushes Foo onto the call stack and begins timing. Finally, just prior to returning, Foo calls Profile.end which pops Foo from the call stack and does some timing calculations. All functions passed to a timeing function are wrapped in a function that pushes a dummy function indicating the type of timing function that is being set and its location in the original code.

2.3 User Interfaces

There are two main ways to interface with the profiler. The first and simplest way is to use profiler.html which can be run in any browser. This allows the user to paste in any JS code. It parses the code and then evaluates the modified code and updates profile statistics every second. This is obviously not intended as a server side interface as simply evaling arbitrary code is about as unsecure as it is possible to be. Another caveat to this interface is that the JavaScript code can only have very limited if any interaction with the DOM without messing up the profiling environment. Much JavaScript is heavily interactive so this will not be the interface of choice for many situations. It is instead intended as an interface to quickly and simply profile standalone code fragments. We consider the second interface to be the typical use case. To profile code, the user may open gen_prof_code.html and paste in any code as before. The code is then modified and the modified code is output to the window. The user then copies this new code to a new JS file to be used in place of the original file. To run the profiler, the user simply imports profiling.js and replaces the original code file with the new modified code and runs his/her html application as normal. The profiler will then open a popup that report profile information every second while the original program is running. This method allows the user to run the profiler in any html application and allow the user to interact with the DOM in any way he/she wants (subject to not overwriting window.report which is used to pass information to the report window).

3 Testing