JS Engine and JavaScript Optimization

JavaScript is one of the most popular programming languages in the world. It was started as a simple scripting language for web browsers. But now it becomes a language of preference for millions of developers. However, its interpretational nature doesn't always provide adequate performance.

To speed up execution of JavaScript programs there were developed several optimization techniques in recent years. One example of modern high-performing JavaScript engine is a V8 engine used in Google Chrome browser and node.js web server among others.

JavaScript Engine

The **JavaScript Engine** is a program or an interpreter which executes JavaScript code. A JavaScript engine can be implemented as a standard interpreter, or just-in-time compiler that compiles JavaScript to bytecode in some form.

The Interpreter

The interpreter uses a concept called ***REPL — read-eval-print-loop***. The real advantage of this method is immediate output and it's easy to implement. But this comes with the cost of execution speed.

The Compiler

Compared to the interpreter, the compiler translates all of the code to executable at once. As a result compilers can make optimization like sharing of machine code for repeated lines of code. But this has a con. It will have a slow start. For this reason, we have a 3rd version of compiler which uses the best of the two worlds.

JIT Compiler

JIT *stands for* ***Just-In-Time.*** So JIT starts with interpreting but it keeps track of *warm codes* which are run few times and also *hot codes* which runs way more times. It attempts to combine the best parts of both interpreters and compilers, making both translation and execution fast.

The basic idea is to avoid retranslation where possible. To start, a profiler simply runs the code through an interpreter. During execution, the profiler keeps track of warm code

segments, which run a few times, and hot code segments, which run many, many times. JIT sends warm code segments off to a baseline compiler, reusing the compiled code where possible.

The JIT compiler also comes with a cost of memory allocation during interpreting, because that is the key for tracking your code. But hey, now we have GB's of RAM so that is not a concern for general web application.

Three things engine does to help us out

- Speculative optimization
- Hidden classes for dynamic lookups
- Function inlining

Optimizations made by JS engine

JavaScript engines implement the JS object model and they use them to speed of accessing properties of JS objects. When any object is created, a lot of information about the object is stored about the object in the memory. Information like whether the object is writable, enumerable, configurable etc. are stored corresponding to an object. It will be a waste of memory to store this redundant information about the objects of the same type again and again. Hence, these pieces of information are stored separately and the object stores only value. This helps in optimizing the memory a lot.

Inline Caching

The main motivation behind shapes is the concept of Inline Caches or ICs. ICs are the key ingredient to making JavaScript run fast! JavaScript engines use ICs to memorize information on where to find properties on objects, to reduce the number of expensive lookups.

Garbage Collection

Garbage collection is **a form of memory management**. It's where we have the notion of a collector which attempts to reclaim memory occupied by objects that are no longer being used. In a garbage-collected language such as JavaScript, objects that are still referenced by your application are not cleaned up. By simply putting the variables where they need to be (ideally, as local as possible, i.e. inside the function where they are used versus an outer scope), things should just work.