Just-in-time Compiler in JavaScriptCore - browser 0x03

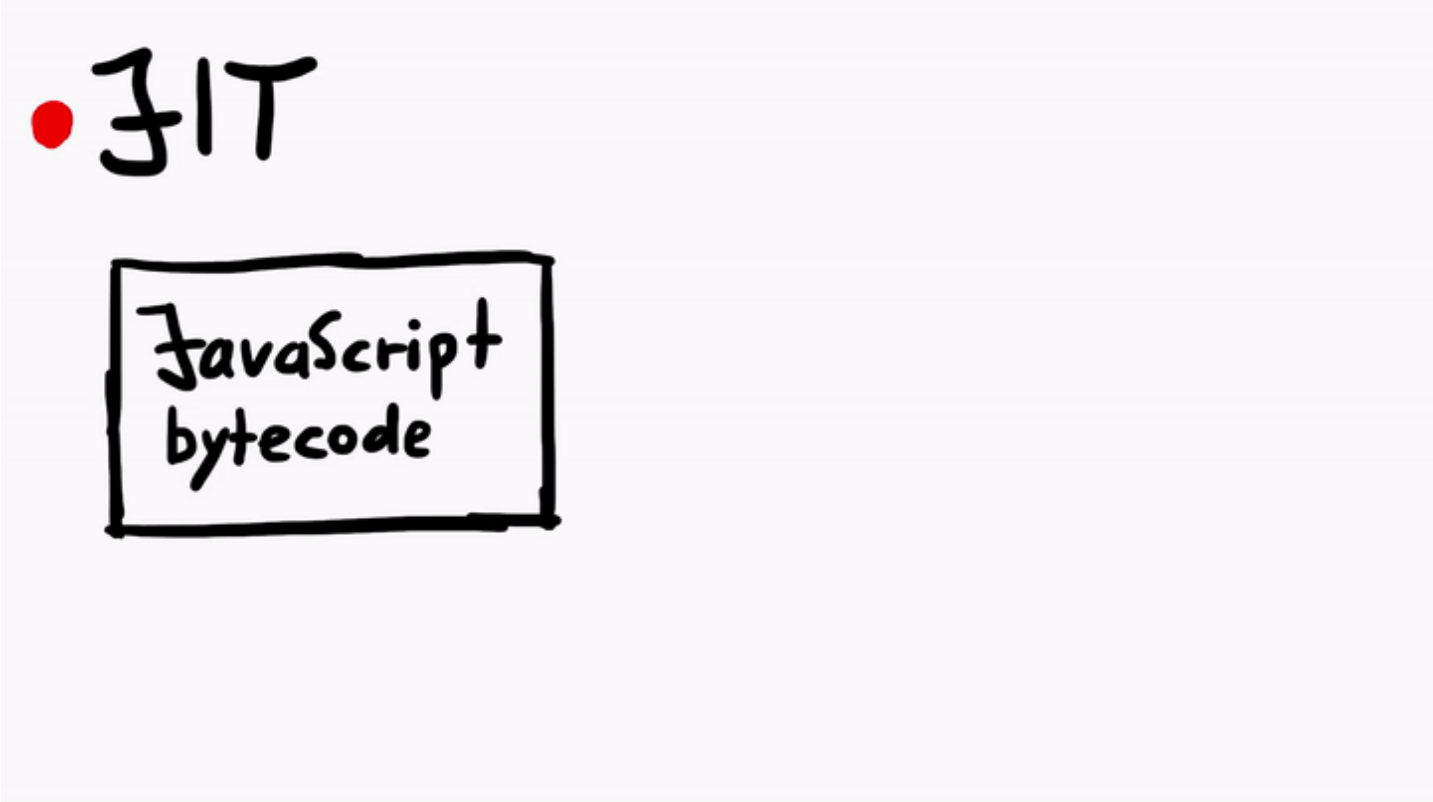
原文地址：https://liveoverflow.com/just-in-time-compiler-in-javascriptcore-browser-0x03/

Introduction

在上一篇文章中，我们探讨了JavaScriptCore（来自WebKit的JavaScript引擎）是如何在内存中存储对象和数值的。在这篇文章中，我们将跟大家一起来探索JIT，即Just-In

The Just-In-Time compiler

说到Just-In-Time编译器，这可是一个复杂的主题。但简单地说，JIT编译器的作用就是将JavaScript字节码（由JavaScript虚拟机执行）编译为本机机器代码（程序集）。虽



为了深入学习JIT，我曾经专门就此向Linus请教，他给出的建议是，多看官方的WebKit资源，例如JavaScriptCore CSI：A Crash Site Investigation Story，这篇文章在如何调试崩溃并诊断错误以查找根本原因方面给出了具体的操作方法——当我们在JavaScriptCore中挖掘漏洞的时候，这方面的内容是非常有用的。除此

```
$ cd webkitDIr
$ ./Tools/Scripts/set-webkit-configuration --asan
$ ./Tools/Scripts/build-webkit --debug
```

这些内容虽然很酷，但我们对JIT方面的内容更感兴趣，所以，让我们找一些对我们的研究有用的东西。在上面提到的文章中，我们可以找到有关JIT编译器相关介绍。

JSC■■■■■■■■■■■■■■■■

实际上，它共有4级：

- 第1级：LLInt解释器
- 第2级：Baseline JIT编译器
- 第3级：DFG JIT
- 第4级：FTL JIT

第1级：LLInt解释器

这是常规的JavaScript解释器，实际上就是一个基本的JavaScript虚拟机。为了加深读者的立即，我们将通过源代码进行讲解。通过快速浏览LowLevelInterpreter.cpp源文件

```
//============================================================================
// The llint C++ interpreter loop:
// LowLevelInteroreter.cpp
```

```
JSValue Cloop::execute(OpcodeID entry OpcodeID, void* executableAddress,
VM* vm, ProtoCallFrame* protoCallFrame, bool isInitializationPass)
{
   // Loop Javascript bytecode and execute each instruction
   // [...] snip
}
```

第2级：Baseline JIT编译器

当一个函数被多次调用的时候，我们就可以称这个函数时一个"热门的（hot）"函数。这个术语用于表示其执行次数较多，而JavaScriptCore会将这种类型的函数JIT化。如果

```
void JIT::privateCompileMainPass()
{
   ...
   // When the LLInt determines it wants to do OSR entry into the baseline JIT in a loop,
   // it will pass in the bytecode offset it was executing at when it kicked off our
   // compilation. We only need to compile code for anything reachable from that bytecode
   // offset.
   ...
}
```

当前栈替换（On Stack
Replacement，OSR）是一种用于在同一函数的不同实现之间进行切换的技术。例如，OSR可以在执行期间从动态解释代码（未优化代码）切换为对应的JIT编译代码。关于

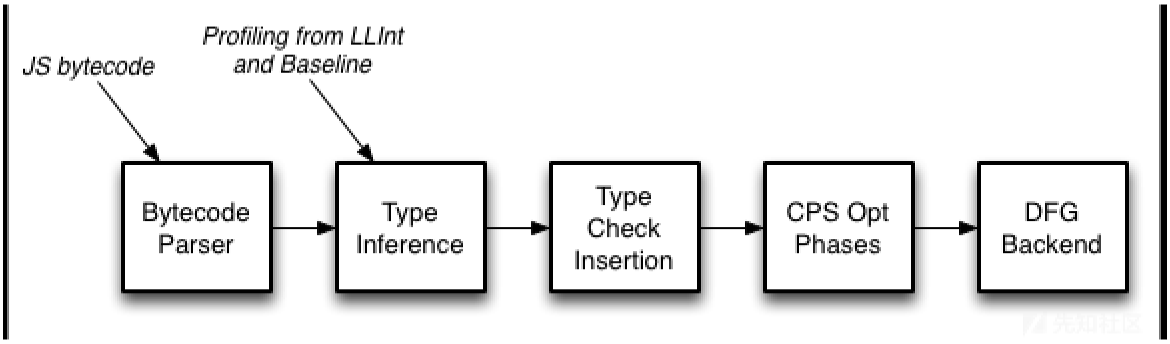
然而，在这个阶段，机器代码仍然与原始字节码非常兼容的（我不知道使用这个术语进行描述是否恰当）的，并且还没有进行真正的优化处理。

第3级：DFG JIT

另一篇介绍WebKit FTL JIT的文章，曾经指出：

```
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■100■■■■■■■■■■■■■■■6■■■■■■■■■■■■■■■■■■Baseline JIT■■■■■■■■■■■■
```

这里，DFG代表数据流图（Data Flow Graph）。此外，上面提到的这篇文章还给出了一些DFG优化流程图。



DFG JIT优化流程图

也就是说，DFG首先会将字节码转换为DFG CPS格式。

期中，CPS表示Continuation-Passing
Style，这意味着您的代码并不使用return语句返回，而是继续并将执行结果传递给下一个函数。我认为可以把它看作是express框架中的next()函数：

```
app.use('/hello', function (req, res, next) {
 console.log('Hello, World!')
 next() // continues execution by calling next() instead if returning.
})
```

继续阅读上面提到的文章，我们会看到下面的内容：

```
DFG■■■■■■■■■DFG CPS■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

如您所见，事情开始变得有趣起来。JIT编译器会猜测类型，如果JIT认为类型没有发生变化，就会省略某些检查工作。很明显，如果一个函数被大量调用的话，这当然可以显

第4级：FTL（Faster Than Light）JIT

在这一级中，会使用著名的编译器后端LLVM来完成各种典型的编译器优化处理。

```
FTL JIT■■■JavaScript■■■■■■■■■■C■■■■■■■
```

在某些时候，LLVM可以被B3后端所取代，但是背后的思想都是一致的。所以，为了进一步优化，JIT编译器会对代码做出更多的假设。

下面，我们从更加贴近实战的角度来考察这个过程，这时，这篇介绍崩溃调查的文献就有了用武之地了。这篇文章引入了几个环境变量，可用于控制JIT的行为并启用调试输

Option	Description	Tiers that can run	Tiers that cannot run
JSC_useJIT=false	Disables all JITs	LLInt	Baseline, DFG, FTL
JSC_useDFGJIT=false	Disables the DFG and above	LLInt, Baseline	DFG, FTL
JSC_useFTLJIT=false	Disables the FTL	LLInt, Baseline, DFG	FTL

来源：https：//webkit.org/blog/6411/javascriptcore-csi-a-crash-site-investigation-story/

这篇文章中还列出了其他一些环境选项，如JSC_reportCompileTimes = true，这样会报告所有JIT编译时间。

我们也可以在lldb中进行相关的设置，使其转储所有JIT编译函数的反汇编代码。

```
(lldb) env JSC_dumpDisassembly=true
(lldb) r
There is a running process, kill it and restart?: [Y/n] Y
...
Genereated JIT code for Specialized thunk for charAt:
   Code at [0x59920a601380, 0x59920a601440]
       0x59920a601380: push %rbp
...
```

这样，我们就可以看到JIT优化调试输出的内容了。看看JIT化的函数的名称，貌似charAt()、abs()等函数已经被优化了。可是，我们希望自己的函数也被优化，为此，可以创

function liveoverflow(n) {

```
let result = 0;
   for(var i=0; i<=n; i++) {
       result += n;
   }
   return result;
}
```

现在，我们需要让这个函数变成一个"热门的"函数，为此，只需重复多次调用这个函数函数，这样一来，JIT就会认为针对这个函数的需求很旺盛，所以，必须将其JIT化。下

```
>>> for(var j=0; j<4; j++) {
   liveoverflow(j);
}
```

我们确实获得了一些输出信息，但是这些信息都与我们的函数的JIT化无关，这意味着该函数还不是"热门的"，所以，接下来将函数的调用次数增加到10次。

```
>>> for(var j=0; j<10; j++) {
   liveoverflow(j);
}
90
>>> // nothing yet
>>> for(var j=0; j<10; j++) {
   liveoverflow(j);
}
90
>>> // nothing yet, let's call it again
>>> for(var j=0; j<10; j++) {
   liveoverflow(j);
}
```

```
90
>>> Generated JIT code for fixup arity:
   Code at [0x59920a601880, 0x59920a601900]
   ...
   //Baseline JIT
   ...
   Optimized liveoverflow#...
```

这下可以了！这里的调用次数使其进入Baseline JIT优化级别。接下来，我们将再次提高调用次数，以触发更高的JIT优化级别。

```
>>> for(var j=0; j<100; j++) {
   liveoverflow(j);
}
Generated Baseline JIT code for liveoverflow#...
...
//optimized function code in assembly
...
```

由此看来，JSC的确会根据函数的调用次数来决定不同的JIT优化级别。现在，让我们"丧心病狂地"将调用次数增加至100,000次，看看会发生什么情况。

```
>>> for(var j=0; j<100000; j++) {
   liveoverflow(j);
}
Optimized liveoverflow#...using FLTMode with FTL ... with B3 generated code ...
...
```

看到没，这次达到了FTL JIT级别！

当然，除了上面介绍的调试输出信息之外，还有大量的其他调试信息，由于本人对它们也不是很熟悉，所以这里就不多说了。不过，到此为止，我们已经介绍了深入挖掘浏

攻击理念

现在，我们已经对JIT有了一定的了解；并且，上一篇文章中我们也对JavaScript对象进行了介绍，下面，我们开始讲解攻击理念。

既然JIT编译器会对代码中的数据类型进行猜测和假定，并删除相关的检查，例如直接从指定的内存偏移处开始移动，那么，攻击者是否能够对其加以利用呢？

假设某些经过JIT处理的代码预期的对象是一个含有双精度浮点数的JavaScript数组，并会直接对这些值进行操作。并且，JIT编译器将所有检查都优化掉了……同时，您找到了

接下来，我们来聊聊JIT如何防止发生这种状况？事实证明，开发人员应设法对每个可能对JIT编译器所做假设产生影响的函数进行评估。因此，对于任何能够改变数组的内存

下面是引用自"Inverting your assumptions: a guide to JIT"中的一句话：

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■clobberWorld■■■■■■■■■■■■■■■clobberWorld■■■■■■■■■■■■■■■■■■■■■■■■

由于JavaScript引擎会否定可能导致副作用的各种假设，因此，它会将可能影响数据安全性的所有函数都标记为"危险的"，并通过调用clobberWorld()函数来完成这些任务。

```
1186
1187            makeHeapTopForNode(node);
1188            break;
1189        }
1190
1191        case StringValueOf: {
1192            clobberWorld();
1193            setTypeForNode(node, SpecString);
1194            break;
1195        }
1196
1197        case StringSlice: {
1198            setTypeForNode(node, SpecString);
1199            break;
1200        }
1201
1202        case ToLowerCase: {
1203            setTypeForNode(node, SpecString);
1204            break;
1205        }
1206
1207        case LoadKeyFromMapBucket:
1208        case LoadValueFromMapBucket:
1209        case ExtractValueFromWeakMapGet:
1210            makeHeapTopForNode(node);
1211            break;
1212
```

下面的代码引自clobberWorld()函数。

```
// JavascriptCore/dfg/DFGAbstractInterpreterInlines.h
template <typename AbstractStateType>
void AbstractInterpreter<AbstractStateType>::clobberWorld()
{
   clobberStructures();
}
```

函数clobberWorld()会调用clobberStructures(), 其定义位于同一个文件中.

```
// JavascriptCore/dfg/DFGAbstractInterpreterInlines.h
template <typename AbstractStateType>
void AbstractInterpreter<AbstractStateType>::clobberStructures()
{
   m_state.clobberStructures();
   m_state.mergeClobberState(AbstractInterpreterClobberState::ClobberedStructures);
   m_state.setStructureClobberState(StructuresAreClobbered);
}
```

所以，JIT为了防止出现副作用，就必须谨慎对待能够改变对象的结构的代码。例如，假设代码访问对象的属性obj.x后，又突然删除了该属性，那么，JIT必须将这个结构标记

好了，本文就说到这里了。在下一篇文章中，我们将研究Linus的exploit，它利用的就是这种漏洞。

Resources

- JavaScriptCore CSI: A Crash Site Investigation Story
- Inverting your assumptions: A Guide to JIT comparisons
- Video Explanation

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板