介绍 WebKit FTL JIT

2014 年 5

作者：Fi

WebKit 的 FTL JIT（超越光速的即时编译器）已切换到新的后端 - Bare Bones Backend（B3）取代 LLVM 成为 FTL JIT 中的低级优化器。

就在十年前，JavaScript（用于驱动网页交互的编程语言）被认为速度太慢，不适合进行严肃的应用程序开发。但由于持续的优化努力，现在可以使用符合可移植标准的 JavaScript 和 HTML5 编写复杂的高性能应用程序，甚至是图形密集型游戏。这篇文章介绍了 JavaScript 优化的一项新进展：WebKit 项目已将其现有的 JavaScript 编译基础架构与最先进的LLVM优化器统一起来。这将使 JavaScript 程序能够利用以前仅对使用 C++ 或 Objective-C 等语言编写的本机应用程序可用的复杂优化。

所有主流浏览器引擎都具有复杂的 JavaScript 优化功能。在 WebKit 中，我们在针对当今网络上的 JavaScript 应用程序的优化和针对下一代网络内容的优化之间取得了平衡。当今的网站提供大量高度动态的 JavaScript 代码，这些代码通常运行时间相对较短。此类代码的主要成本是加载它所花费的时间和用于存储它所用的内存。这些开销在JSBench中有所说明，这是一个直接基于现有网络应用程序（如 Facebook 和 Twitter）的基准测试。WebKit 在该基准测试中的表现一直非常出色。

WebKit 还具有更高级的编译器，可消除动态语言开销。此编译器有利于长时间运行的程序，并且大部分执行时间都集中在相对较少的代码中。此类代码的示例包括图像过滤器、压缩编解码器和游戏引擎。

在努力改进 WebKit 的优化编译器的过程中，我们发现我们越来越多地重复传统提前 (AOT) 编译器中已经存在的逻辑。我们没有继续复制数十年的编译器专业知识，而是研究将 WebKit 的编译器基础架构与现有的低级编译器基础架构 LLVM 统一起来。从r167958开始，该项目不再是调查。我很高兴地报告，我们基于 LLVM 的即时 (JIT) 编译器，称为 FTL（第四层 LLVM 的缩写），已在 Mac 和 iOS 端口上默认启用。

这篇文章总结了过去一年进行的 FTL 工程。它首先回顾了 WebKit 的 JIT 编译器在 FTL 之前的工作原理。然后描述了 FTL 架构以及我们如何解决使用 LLVM 作为动态语言 JIT 的一些基本挑战。最后，这篇文章展示了 FTL 如何实现一些特定于 JavaScript 的优化。

WebKit JavaScript 引擎概述

本节概述了在添加 FTL JIT 之前 WebKit 的 JavaScript 引擎的工作原理。如果您已经熟悉配置文件导向类型推断和分层编译等概念，则可以跳过本节。

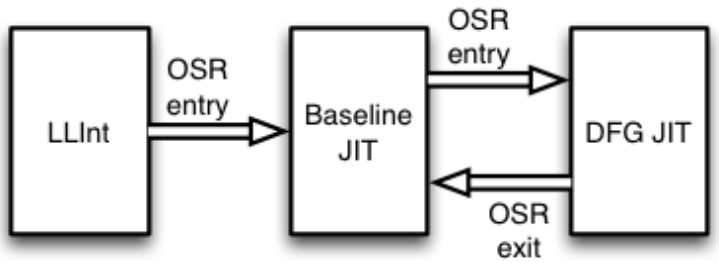

图 1. WebKit 三层架构。箭头表示堆栈上替换（简称 OSR）。

在 FTL JIT 之前，WebKit 使用三层策略来优化 JavaScript，如图 1 所示。运行时可以根据每个函数从三个执行引擎或层中进行选择。每一层都将我们的内部 JavaScript 字节码作为输入，并使用不同级别的优化来解释或编译字节码。有三个可用的层：LLInt （低级解释器）、Baseline JIT和DFG JIT（数据流图 JIT）。LLInt 针对低延迟启动进行了优化，而 DFG 针对高吞吐量进行了优化。任何函

数的第一次执行始终从解释器层开始。只要函数中的任何语句执行超过 100 次，或者函数被调用超过 6 次（以先到者为准），执行就会转移到由 Baseline JIT 编译的代码中。这消除了一些解释器的开销，但缺乏任何重要的编译器优化。一旦任何语句在 Baseline 代码中执行超过 1000 次，或者 Baseline 函数被调用超过 66 次，我们就会再次将执行转移到 DFG JIT。通过使用一种称为栈上替换（简称 OSR）的技术，几乎可以在任何语句边界转移执行。OSR 允许我们在字节码指令边界解构字节码状态，而不管执行引擎是什么，并为任何其他引擎重建它以继续使用该引擎执行。如图 1 所示，我们可以使用它来将执行转换到更高的层级（*OSR 入口*）或降级到更低的层级（*OSR 出口*；更多信息见下文）。

三层策略在低延迟和高吞吐量之间实现了良好的平衡。在每个 JavaScript 函数上运行优化 JIT（例如 DFG）的成本将高得令人望而却步；这就像每次要运行应用程序时都必须从源代码编译它一样。优化编译器需要时间才能运行，JavaScript 优化 JIT 编译器也不例外。多层策略可确保我们用于优化 JavaScript 函数的 CPU 时间和内存量始终与该函数以前使用的 CPU 时间量成正比。我们的引擎对 Web 内容做出的唯一先验假设是，各个函数的过去执行频率可以很好地预测这些函数未来的执行频率。这三个层级各有独特的优势。多亏了 LLInt，我们不必浪费时间编译和存储仅运行一次的代码。这可能是顶级标记中的初始化代码，`<script>` 也可能是一些偷偷溜过我们专门的 JSON 路径的 JSONP 内容。这节省了总执行时间，因为如果每个语句仅执行一次，则编译比解释花费的时间更长； JIT 编译器本身仍必须循环并切换字节码中的所有指令，就像解释器一样。只要任何指令执行多次，LLInt 相对于 JIT 的优势就会开始减弱。对于频繁执行的代码，LLInt 中大约四分之三的执行时间花在间接分支上，以分派到下一个字节码指令。这就是为什么我们在仅 6 次函数调用或语句重新执行 100 次后就升级到 Baseline JIT：此 JIT 主要负责消除解释器分派开销，但除此之外，它生成的代码与解释器的代码非常相似。这使 Baseline JIT 能够非常快速地生成代码，但它留下了一些性能优化，例如寄存器分配和类型特化。我们将高级优化留给 DFG JIT，它仅在代码在 Baseline JIT 中重新执行足够多次后才被调用。

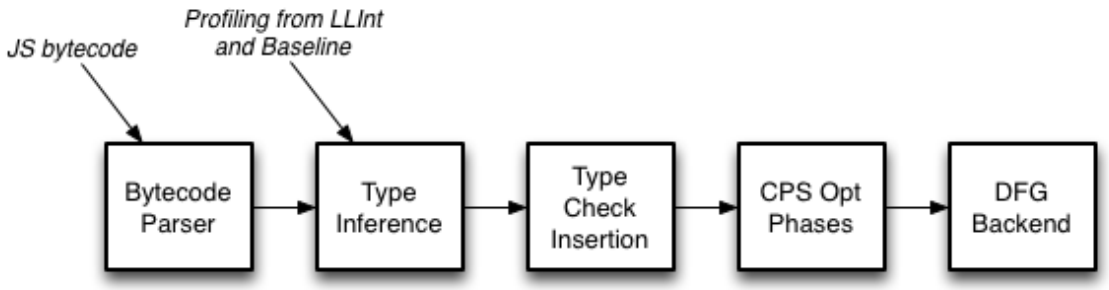

图 2. DFG JIT 优化管道。DFG 首先将字节码转换为 DFG CPS 形式，从而揭示变量和临时变量之间的数据流关系。然后使用分析信息推断类型的猜测，并使用这些猜测插入一组最小的类型检查。接下来是传统的编译器优化。最后，编译器直接从 DFG CPS 形式生成机器代码。

DFG JIT 将字节码转换为更易于优化的形式，试图揭示操作之间的数据流关系；DFG 中间代码表示的细节最接近函数式语言编译器中经常使用的经典延续传递样式(CPS) 形式。DFG 结合了传统的编译器优化，例如寄存器分配、控制流图简化、公共子表达式消除、死代码消除和稀疏条件常量传播。但严肃的编译器需要了解变量的类型以及堆中对象的结构才能执行优化。因此，DFG JIT 使用来自 LLInt 和 Baseline JIT 的分析反馈来构建关于变量类型的预测。这允许 DFG 对任何传入值的类型进行猜测。请考虑以下简单程序作为示例：

```
function foo(a, b) { return a + b + 42; }
```

仅从源代码来看，无法判断 `a` 和 `b` 是数字、字符串还是对象；如果它们是数字，我们无法判断它们是整数还是双精度数。因此，我们让 LLInt 和 Baseline JIT 为每个变量收集值配置文件，这些变量的值源自一些不明显的事物，例如函数参数或来自堆的负载。DFG 只有 `foo` 在执行了很多次后才会编译，并且每次执行都会为这些参数的配置文件贡献值。例如，如果参数是整数，那么 DFG 将能够使用廉价的“你是整数吗？”检查两个参数，并对两个加法进行溢出检查来编译此函数。不需要 `valueOf()` 生成双精度、字符串连接或对对象方法的调用，这样既节省了空间，又节省了总执行时间。如果这些检查失败，DFG 将使用堆栈替换(OSR) 将执行转移回

Baseline JIT。这样，当发现 DFG 的任何推测优化无效时，Baseline JIT 就可以作为永久的后备。

另一种分析反馈形式是 Baseline JIT 的多态内联缓存。多态内联缓存是 Smalltalk 社区中用于优化动态调度的经典技术。考虑以下 JavaScript 函数：

```
function foo(o) { return o.f + o.g; }
```

在此示例中，属性访问可能导致任何情况，从堆中众所周知的位置的简单加载到 getter 调用，甚至复杂的 DOM 陷阱，例如，如果 `o` 是文档对象，而“f”是页面中元素的名称。Baseline JIT 最初会将这些属性访问作为完全多态调度执行。但在执行此操作时，它会记录所采取的步骤，然后将就地修改堆访问，以缓存*必要*步骤，以便将来重复类似的访问。例如，如果对象在距离对象基址偏移量 16 处具有属性“f”，则代码将被修改为首先快速检查传入对象是否包含偏移量 16 处的属性“f”，然后执行加载。这些缓存被称为*内联*，因为它们完全表示为生成的机器代码。它们被称为*多态*，因为如果遇到不同的对象结构，机器代码将被修改为在执行完全动态属性查找之前打开先前遇到的对象类型。当 DFG 编译此代码时，它将检查内联缓存是否为*单态*（仅针对一个对象结构进行优化），如果是，它将仅对该对象结构进行检查，然后进行直接加载。在此示例中，如果 `o` 始终是具有属性“f”和“g”且偏移量不变的对象，则 DFG 只需进行一次类型检查，`o` 然后进行两次直接加载。

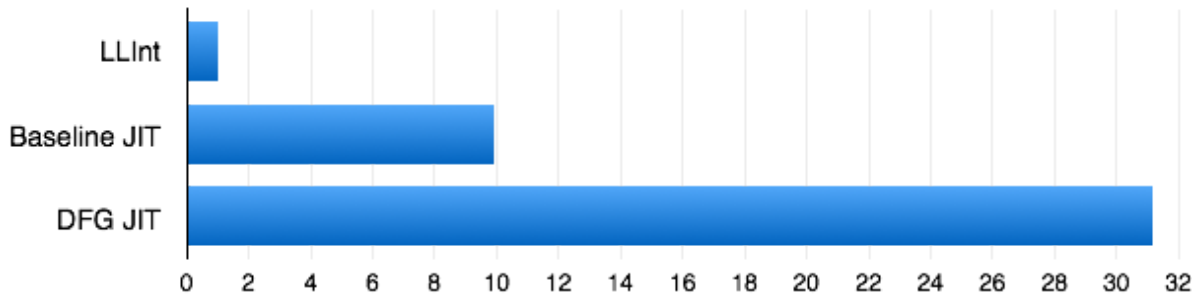

图 3.三个层级在 Richards 基准测试中的相对加速（越高越好）。

图 2 显示了 DFG 优化管道的概览。DFG 使用值配置文件和内联缓存作为类型推断的起点。这使它能够生成专门针对我们之前观察到的类型的代码。然后，DFG 插入一组最小的类型检查，以防止程序将来使用不同的类型。当这些检查失败时，DFG 将*OSR 退出*回到 Baseline JIT 的代码。由于任何检查失败都会导致控制权离开 DFG 代码，因此 DFG 编译器可以在假设不需要重复检查的情况下优化检查后的代码。这使 DFG 能够积极地最小化类型检查集。变量的大多数重用都不需要任何检查。

图 3 显示了每个层级在一个代表性基准测试（Martin Richards 进行的操作系统模拟）上的相对加速情况。每个层级生成代码所需的时间都比其下方的层级更长，但每个层级带来的吞吐量增加足以弥补这一不足。

三层策略使 WebKit 能够适应 JavaScript 代码的要求和特性。我们根据代码预计运行的时间来选择层。我们使用较低的层来生成用于引导 DFG JIT 中的类型推断的分析信息。这很有效 - 但即使有三层，我们也不得不做出权衡。我们经常发现，调整 DFG JIT 以进行更积极的优化会提高长时间运行代码的性能，但会损害短时间运行的程序，因为 DFG 的编译时间在总运行时间中的作用比 DFG 生成的代码的质量更大。这让我们相信我们应该添加第四层。

#

构建第四层 JIT

WebKit 的优势在于它能够动态适应不同的 JavaScript 工作负载。但这种适应性仅与我们必须选择的层级一样强大。DFG 无法同时成为低延迟优化编译器和生成高吞吐量代码的编译器——任何让 DFG 生成更好代码的尝试都会减慢它的速度，并会增加运行时间较短的代码的延迟。添加第四层并将其用于更繁重的优化，同时保持 DFG 相对轻量级，使我们能够平衡运行时间较长和运行时间较短的代码的要求。

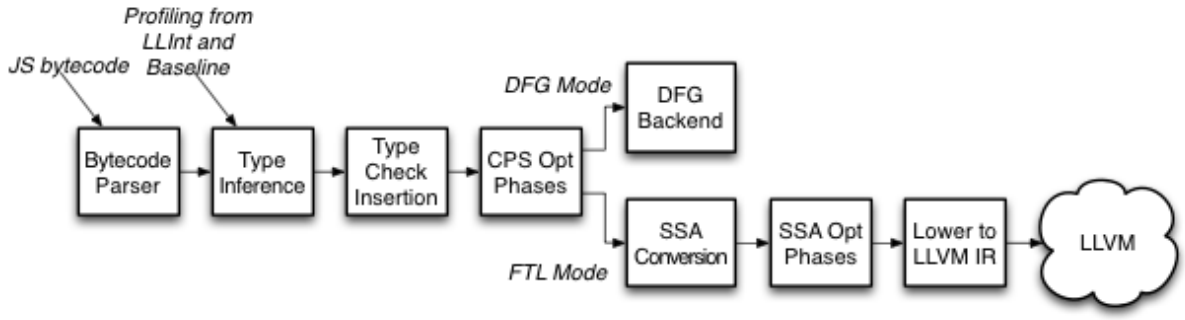

图 4. DFG 和 FTL JIT 优化管道。我们重用了大部分 DFG 阶段，包括其基于 CPS 的优化。新的 FTL 管道是第三层 DFG 后端的直接替代品。它涉及对 DFG SSA 格式进行额外的 JavaScript 感知优化，然后是将 DFG IR（中间表示）降低到 LLVM IR 的阶段。然后我们调用 LLVM 的优化管道和 LLVM 的 MCJIT 后端来生成机器代码。

FTL JIT旨在为 JavaScript 带来类似 C 语言的积极优化。我们希望确保我们在 FTL 中所做的工作能够影响最广泛的 JavaScript 程序（而不仅仅是用受限子集编写的程序），因此我们重用了 DFG JIT 中现有的类型推断引擎以及现有的 OSR 出口和内联缓存基础架构来处理动态类型。但我们也希望获得最积极、最全面的低级编译器优化。为此，我们选择低级虚拟机 (LLVM) 作为我们的编译器后端。LLVM 拥有复杂的生产质量优化管道，其中包含我们想要的许多功能，例如全局值编号、成熟的指令选择器和复杂的寄存器分配器。

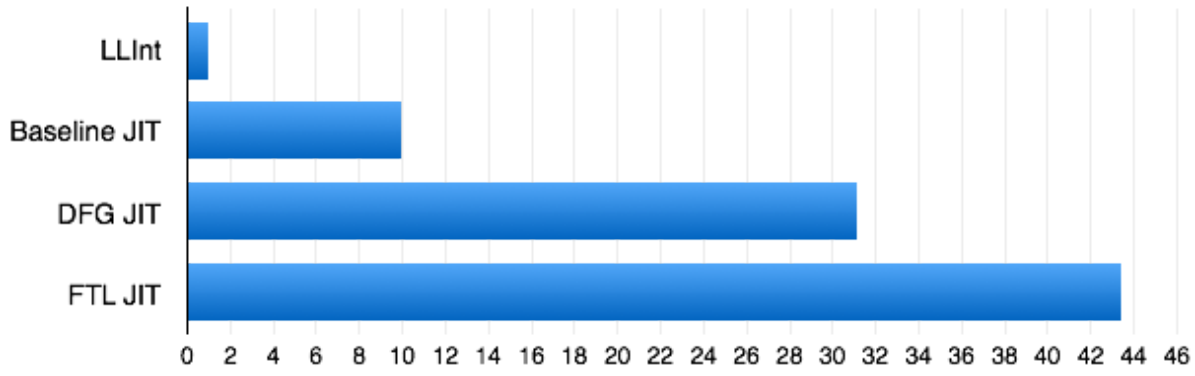

图 5. Richards 基准测试中所有四个层级的相对加速（越高越好）。

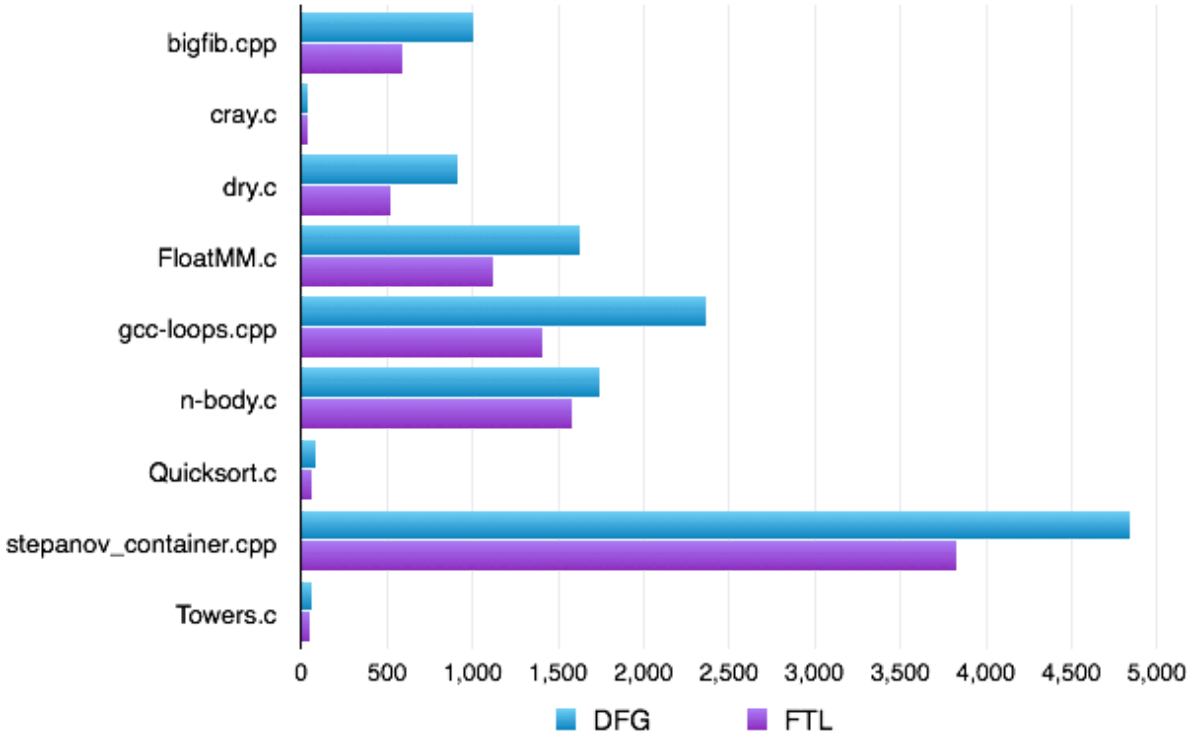

图 6.使用 DFG 和 FTL 进行的一系列 asm.js 基准测试的执行时间（以毫秒为单位）（越低越好）。*FTL 平均快 35%。*

图 4 显示了 FTL 架构的概览。FTL 主要以 DFG 的备用后端的形式存在。我们不会调用 DFG 的机器代码生成器（速度非常快但几乎不进行低级优化），而是将 DFG 的 JavaScript 函数表示转换为静态单赋值(SSA) 形式并执行其他优化。转换为 SSA 比 DFG 的原始 CPS 转换更昂贵，但它可以实现更强大的优化，例如循环不变代码移动。完成这些优化后，我们将代码从 DFG SSA 形式转换为 LLVM IR。由于 LLVM IR 也是基于 SSA 的，因此这是一个直接的线性和一对多转换。它只需要对代码进行一次传递，并且 DFG SSA 形式中的每条指令都会产生一条或多条 LLVM IR 指令。此时，我们从代码中消除了任何特定于 JavaScript 的知识。所有 JavaScript 习语都被降低为使用 LLVM IR 的 JavaScript 习语的内联实现，或者如果我们知道代码路径不常见，则降低为过程调用。

这种架构让我们可以自由地向 FTL 添加重量级优化，同时保留旧的 DFG 编译器作为第三层。我们甚至可以添加在 DFG 和 FTL 中都能运行的新优化，方法是将它们添加到编译管道中的“分叉”之前。它还可以与 LLVM IR 的设计很好地集成：LLVM IR 是静态类型的，因

此我们需要在转换到 LLVM IR 之前执行动态语言优化。幸运的是，DFG 已经擅长推断类型 - 因此当 LLVM 看到代码时，我们将为所有变量赋予类型。使用我们的间接方法将 JavaScript 转换为 LLVM IR 变得非常自然 - 首先源代码转换为 WebKit 的字节码，然后转换为 DFG CPS IR，然后是 DFG SSA IR，然后才是 LLVM IR。每次转换都会消除 JavaScript 的一些动态，到我们进入 LLVM IR 时，我们将消除所有动态。这样，我们就可以充分利用 LLVM 的优化功能，即使是针对普通的人工编写的 JavaScript；例如，在图 3 中的 Richards 基准测试中，FTL 为我们提供了比 DFG 额外 40% 的性能提升，如图 5 所示。图 6 显示了 FTL 在 asm.js 基准测试中为我们带来的性能摘要。请注意，FTL 不会通过识别编译指令来为 asm.js 提供特殊情况`"use asm"`。所有性能都来自 DFG 的类型推断和 LLVM 的低级优化能力。

WebKit FTL JIT 是第一个使用 LLVM JIT 基础架构对动态语言进行配置文件导向编译的大型项目。为了实现这一点，我们需要在 WebKit 和 LLVM 中进行一些重大更改。与我们现有的 JIT 相比，LLVM 需要更多时间来编译代码。WebKit 使用复杂的分代垃圾收集器，但 LLVM 不支持侵入式 GC 算法。配置文件驱动的编译意味着我们可能会在函数运行时调用优化编译器，并且我们可能希望在循环中间将函数的执行转移到优化代码中；据我们所知，FTL 是第一个执行堆栈替换以将热循环转移到 LLVM 编译代码的编译器。最后，LLVM 以前不支持我们依赖来处理动态代码的自修改代码和去优化技巧。这些问题将在下面详细描述。

避免 LLVM 编译时间

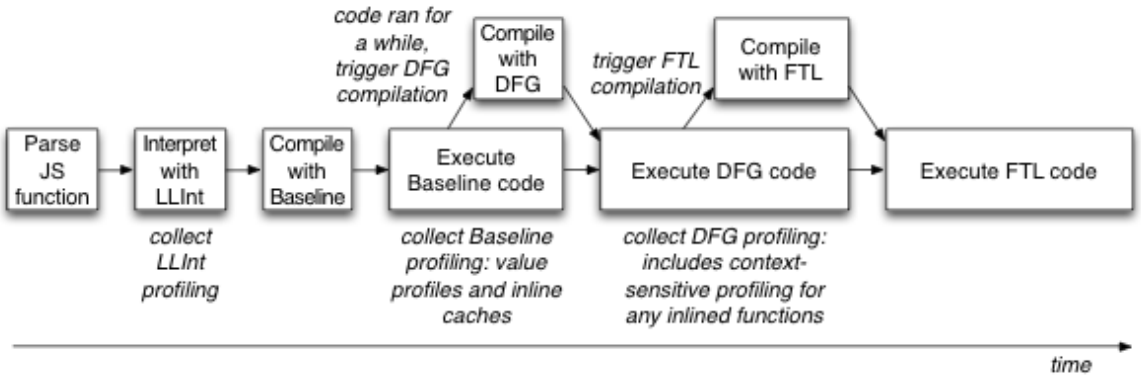

图 7.代码在 LLInt、Baseline JIT 和 DFG JIT 中运行一段时间后，才会调用 FTL。当我们从并发线程调用 FTL 编译器时，相关函数已经在执行由 DFG JIT 生成的相对较快的代码。然后，FTL 可以提取从 LLInt、Baseline JIT 和 DFG JIT 收集的分析数据，以最大

编译器设计受制于经典的延迟-吞吐量权衡。很难设计出能够快速生成优秀代码的编译器——生成代码最快的编译器也会生成最差的代码，而像 LLVM 这样生成优秀代码的编译器通常很慢。此外，编译器在延迟-吞吐量连续体中的位置往往是系统性的；很难设计出可以从低延迟变为高吞吐量或反之亦然的编译器。

例如，DFG 的后端可以非常快速地生成普通代码。它生成代码的速度很快，正是因为它只对相对高级的表示进行一次传递；但同样的特性意味着它留下了许多未利用的优化机会。LLVM 则相反。为了最大限度地增加优化机会，LLVM 对最有可能暴露冗余和低效习语的非常低级的代码进行操作。在将该代码转换为机器代码的过程中，LLVM 通过涉及不同表示的几个阶段执行多次降低转换。每个表示都会揭示不同的加速机会。但这也意味着没有办法缩短 LLVM 与 DFG 相比的缓慢性。仅将代码转换为 LLVM 的 IR 就比运行整个 DFG 后端更昂贵，因为 LLVM IR 是如此低级。这就是权衡：编译器架构师必须先验地选择他们的编译器是具有低延迟还是高吞吐量。

FTL 项目旨在将高吞吐量和低延迟编译的优势结合起来。对于短时间运行的代码，我们确保不会为调用 LLVM 付出代价，除非执行计数分析证明有必要这样做。对于长时间运行的代码，我们通过使用并发性和并行性的组合来隐藏调用 LLVM 的成本。

我们的第二层和第三层——Baseline 和 DFG JIT——仅当函数变得足够“热”时才会调用。FTL 也不例外。除非函数已经由 DFG 编译过，否则 FTL 不会编译该函数。然后，DFG 重用来自 Baseline JIT 的执行计数分析方案。每个循环后沿都会将 1 加到内部每个函数计数器上。返回会将 15 加到同一个计数器上。如果计数器过零，我们将触发编译。计数器开始时是一个负值，该负值对应于在触发更高层的编译器之前我们希望的函数“执行”次数的估计值。对于 Baseline-to-DFG 层级，我们将计数器设置为 $-1000 \times C$，其中C是编译单元大小和可用可执行内存量的函数。C通常接近于 1。DFG-to-FTL 层级更激进；我们将计数器设置为 $-100000 \times C$。这

可确保短时间运行的代码不会导致基于 LLVM 的昂贵编译。在现代硬件上运行时间超过约 10 毫秒的任何函数都将由 FTL 编译。

除了保守的编译策略外，我们还通过使用并发和并行编译来最大限度地减少 FTL 对启动时间的影响。DFG 已经作为并发编译器运行了近一年。DFG 管道的每个部分都是并发运行的，包括字节码解析和分析。借助 FTL，我们将这一点提升到了一个新的水平：我们启动了多个 FTL 编译器线程，FTL 管道的 LLVM 部分甚至可以与 WebKit 的垃圾收集器并发运行。我们还保留了一个以更高优先级运行的专用 DFG 编译器线程，以确保即使队列中有 FTL 编译，任何新发现的 DFG 编译机会都具有更高的优先级。

图 7 展示了函数从 LLInt 一直到 FTL 的时间线。解析和 Baseline JIT 编译仍然不是并发的。一旦 Baseline JIT 代码开始运行，主线程将永远不会等待函数编译，因为所有后续编译都是并发进行的。此外，如果函数运行频率不够高，DFG 就不会编译该函数，更不用说 FTL 了。

将 LLVM 与垃圾收集集成

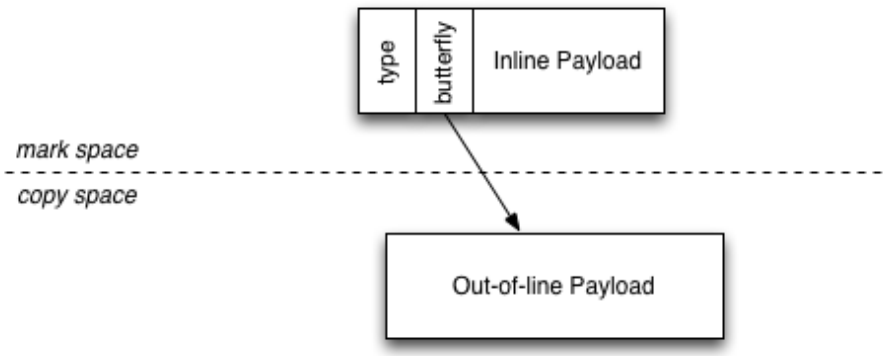

图 8. WebKit 的 JavaScript 对象模型。固定大小的对象及其推断出的属性存储在隔离的空闲列表标记-清除空间中的一个小型非移动单元中。可变大小的对象（例如数组和字典）的有效负载存储在碰撞指针快速释放标记区域复制空间中的可移动*Butterfly*中。我们允许 WebKit 中的任何代码（包括我们运行时的 C++ 代码和由我们的任何 JIT（包括 LLVM）生成的代码）直接引用单元和 Butterfly，并拥有指向两者的内部指针。

WebKit 使用分代垃圾收集器，为对象单元提供非移动空间，用于存放固定大小的数据，并为可变大小的数据提供混合标记区域/复制

空间（见图 8）。对象的生成由粘性标记位跟踪。我们的收集器还具有其他优化功能，例如并行收集和精心调整的分配和屏障快速路径。

我们将垃圾收集与 LLVM 集成的方法旨在最大限度地提高吞吐量和收集器精度，同时消除 LLVM 了解垃圾收集器的任何需求。我们使用的方法归功于 Joel Bartlett，自 20 世纪 80 年代末以来就已为人所知，我们已经使用多年了。我们认为，对于像 WebKit 这样的高性能系统，这种方法比完全保守的收集器（如Boehm-Demers-Weiser）或各种基于强制变量溢出到堆栈上的精确收集策略更可取。完全 Boehm 式的保守主义意味着堆和堆栈中的每个指针大小的字都被保守地认为可能是一个指针。这既慢（每个字都需要经过指针测试算法），又非常不精确，有可能保留大量死对象。另一方面，强制任何东西（指针或疑似指针的值）进入堆栈都会导致速度变慢，因为需要额外的指令将这些值存储到内存中。我们使用 LLVM 的原因之一是获得更好的寄存器分配；将东西放在堆栈上违背了这一目标。幸运的是，Bartlett 垃圾收集器提供了一种优雅的中间立场，仍然允许我们使用复杂的分代复制收集器。

在 Bartlett 收集器中，堆中的对象应具有类型映射，以告诉我们哪些字段具有指针以及应如何解码这些指针。只能从其他堆对象访问的对象可能会被移动，指向它们的指针可能会被更新。但堆栈不需要具有堆栈映射，并且必须固定被认为从堆栈引用的对象。收集器的堆栈扫描必须：(1) 确保它除了读取堆栈本身之外还读取所有寄存器，(2) 保守地认为每个指针宽度字都可能是指针，以及 (3) 固定任何看起来可能从堆栈访问的对象。这意味着我们用来编译 JavaScript 的编译器可以对值执行任何操作，并且它们不必知道这些值是否是指针。

由于程序堆栈通常比堆的其余部分小几个数量级，因此堆栈上的恶意值意外导致大量死堆对象被保留的可能性非常低。事实上，典型堆中超过 99% 的对象仅被其他堆对象引用，这允许 Bartlett 收集器移动这些对象（无论是为了改善缓存局部性还是对堆进行碎片整理），就像完全准确的收集器一样。保留一些死对象当然是可能的，只要它只对应少量的空间开销，我们就可以容忍它。在实践中，我们发现，只要我们使用额外的堆栈清理技巧（例如确保我们

的编译器使用压缩的堆栈框架并偶尔将已知未使用的堆栈区域归零），死保留量就可以忽略不计，不值得担心。

我们最初选择使用 Bartlett 垃圾收集方法，因为它允许我们的运行时用普通 C++ 编写，直接引用局部变量中的垃圾收集对象。这既更高效（没有对象句柄的开销），也更容易维护。这种方法还允许我们采用 LLVM 作为编译器后端，而无需通过堆栈溢出来影响性能，这一事实只会增强我们对这种著名算法的信心。

热循环传输

使用多个执行引擎运行 JavaScript 代码的挑战之一是用另一个编译器编译的代码替换一个编译器编译的代码，例如当我们决定从 DFG 升级到 FTL 时。这通常很容易。大多数函数的生命周期都很短——它们往往在被调用后不久就返回。即使函数占总执行时间的很大一部分，情况也是如此。此类函数往往被频繁调用，而不是在一次调用期间运行很长时间。因此，我们替换代码的主要机制是编辑用于虚拟调用解析的数据结构。此外，对于任何已取消虚拟化的调用，我们会取消所有传入调用与函数旧代码的链接，然后将它们重新链接到新编译的代码。对于大多数函数来说，这已经足够了：在新优化的代码可用后不久，所有先前对该函数的调用都会返回，所有未来的调用都会进入优化代码。

虽然这对大多数函数都很有效，但有时函数会运行很长时间而不返回。这种情况在基准测试中最为常见 - 尤其是微基准测试。但有时它也会影响真实代码。我们发现的一个例子是类型数组初始化循环，它在 FTL 中的运行速度比在 DFG 中快 10 倍，差异如此之大以至于它在 DFG 中的总运行时间明显大于使用 FTL 编译该函数所需的时间。对于这样的函数，在函数运行时将执行从 DFG 编译的代码转移到 FTL 编译的代码是有意义的。这个问题称为热循环转移（因为这只会在函数有循环时发生），在 WebKit 的代码中我们将其称为*OSR 条目*。

值得注意的是，LLInt-to-Baseline 和 Baseline-to-DFG 层级已经支持 OSR 进入。OSR 进入 Baseline JIT 利用了这样一个事实：我们很容易知道 Baseline JIT 如何在每条指令边界表示所有变量。事实上，它的表示与 LLInt 相同，因此 LLInt-to-Baseline OSR 进入仅涉及跳转到适当的机器代码地址。OSR 进入 DFG JIT 稍微复杂

一些，因为 DFG JIT 通常以不同于 Baseline JIT 的方式表示状态。因此，DFG 通过将函数的控制流图视为具有多个入口点来实现 OSR 进入：一个入口点用于函数的开始，另一个入口点位于每个循环头。OSR 进入 DFG 的操作与特殊函数调用非常相似。这会抑制许多优化，但会使 OSR 进入非常便宜。

不幸的是，FTL JIT 无法使用这两种方法。我们希望 LLVM 在表示状态的方式上拥有最大的自由度，并且不存在任何机制来询问 LLVM 如何在任意循环头中表示状态。这意味着我们无法执行 OSR 条目的 Baseline 样式。此外，LLVM 假设函数只有一个入口点，而更改这一点将需要进行重大的架构更改。因此，我们也无法执行 OSR 条目的 DFG 样式。但更根本的是，让 LLVM 假设执行可以通过除函数开头之外的任何路径进入函数会使某些优化（如循环不变代码移动，简称 LICM）变得更加困难。我们希望 FTL JIT 能够将代码移出循环。如果循环有两个入口点（一个沿着函数开头的路径，另一个通过 OSR 条目），那么 LICM 必须复制它移出循环的任何代码，并将其提升到两个可能的入口点。这将带来重大的架构挑战，实际上，这意味着向 FTL 添加新的优化比在可以做出单入口点假设的编译器中添加这些优化更具挑战性。简而言之，多个入口点的复杂性加上 LLVM 目前不支持它的事实意味着我们不想在 FTL 中使用这种方法。

我们最终用于热循环传输的方法是为每个我们要使用的入口点单独复制一个函数。当一个函数变得足够热以保证 FTL 编译时，我们会投机取巧地假设该函数最终会返回并通过再次调用重新进入 FTL。但是，如果在我们已经进行 FTL 编译之后，该函数的 DFG 版本继续变热，并且我们检测到它因为循环中的执行计数而变热，那么我们假设通过该循环进入是最有利的。然后，我们在一种称为 的特殊模式下触发该函数的第二次编译`FTLForOSREntry`，其中我们使用循环预头作为函数的入口点。这涉及对 DFG IR 的转换，我们为该函数创建一个新的起始块，并让该块从全局暂存缓冲区加载循环中的所有状态。然后我们让块跳转到循环中。之前在循环之前的所有代码最终都死了。因此，从 LLVM 的角度来看，该函数不接受任何参数。然后，我们的运行时的 OSR 入口机制让 DFG 将循环头处的状态转储到该全局暂存缓冲区中，然后我们“调用”LLVM 编译的函数，它将从该暂存缓冲区加载状态并继续执行。

FTL OSR 入口可能涉及多次调用 LLVM，但我们只在有充分证据表明该函数绝对需要它的情况下才会这样做。我们采用的方法不需要 LLVM 的任何新功能，并且可以最大限度地利用 FTL 可用的优化机会。

去优化和内联缓存

自修改代码是高性能虚拟机设计的基石。我们希望优化的 JavaScript 编译器能够处理以下三种情况：

部分编译。我们通常会将函数的某些部分保留为未编译状态。最明显的好处是节省内存和编译时间。这在使用涉及类型检查的推测优化时会发挥作用。在 DFG 中，原始 JavaScript 代码中的每个操作通常都会导致至少一次推测检查 - 要么验证某个值的类型，要么确保某些条件（如数组索引在边界内）已知成立。在一些精心编写的 JavaScript 程序中，或者在由Emscripten等转译器生成的程序中，通常会优化掉这些检查中的大多数。但在简单的人工编写的 JavaScript 代码中，许多这些检查将保留，并且进行这些检查仍然比不执行任何推测优化要好。因此，对于 FTL 来说，能够处理可能具有数千条触发 OSR 到基线 JIT 的“侧出口”路径的代码非常重要。让 LLVM 急切地编译这些路径是没有意义的。相反，我们希望通过使用自修改代码来懒惰地修补这些路径。

失效。DFG优化管道能够在堆中的对象和运行时状态的各个方面安装观察点。例如，可以对用作原型的对象进行观察点设置，以便对原型实例的任何访问都可以对原型的条目进行常量折叠。我们还使用这种技术来推断哪些变量是常量，这使得开发人员可以轻松地以与使用`static final` Java 中的变量相同的方式使用 JavaScript 变量。当这些观察点触发时（例如，因为写入了推断的常量变量），我们将使任何对这些变量的值做出假设的优化代码失效。失效意味着取消对这些函数的任何调用的链接，并确保函数内的任何调用点在其被调用方返回后立即触发 OSR 退出。失效和部分编译都是我们去优化策略的大致组成部分。

多态内联缓存。FTL有许多不同的策略来为堆访问或函数调用生成代码。这两种 JavaScript 操作都可能是多态的。它们可以对多种类型的对象进行操作，并且流入操作的类型集可能是无限的——在 JavaScript 中，可以以编程方式创建新的对象或函数，如果无法推

断出通用类型，我们的类型推断可能会为每个对象归因于不同的类型。大多数堆访问和调用最终都会产生一组有限（且较小的）流入的类型，因此 FTL 通常可以发出相对直接的代码。但我们的分析可能会告诉我们流入操作的类型集很大。在这种情况下，我们发现处理操作的最佳方式是使用内联缓存，就像 Baseline JIT 所做的那样。内联缓存是一种动态去虚拟化，我们为操作发出的代码可能会随着操作使用的类型的变化而重新修补。

这些场景中的每一个都可以被认为是自修改内联汇编，但在较低级别上具有编程接口来修补机器代码内容并选择应使用哪些寄存器或堆栈位置。与内联汇编非常相似，实际内容由 LLVM 的客户端而不是 LLVM 本身决定。我们将这种新功能称为修补点，它在 LLVM 中作为内在函数公开`llvm.experimental.patchpoint`。使用内在函数的工作流程如下：

1. WebKit（或任何 LLVM 客户端）决定哪些值需要提供给补丁点，并选择这些值表示的约束。可以指定以任何形式给出值，在这种情况下，它们可能最终成为编译时常量、寄存器、带加数的寄存器（可以通过获取某个寄存器并向其添加一个常量值来恢复该值）或内存位置（可以通过从获取某个寄存器并向其添加一个常量计算出的地址加载来恢复该值）。还可以通过调用约定传递值，然后选择调用约定来指定更细粒度的约束。可用的调用约定之一是`anyreg`，这是一个非常轻量级的约束——值始终在寄存器中可用，但 LLVM 可以选择使用哪些寄存器。
   WebKit 还必须选择机器代码片段的大小（以字节为单位）以及返回类型。

2. LLVM 发出一个*nop sled*（一系列 nop 指令），其长度由 WebKit 选择。使用 LLVM 移交给 WebKit 的单独数据部分，LLVM 描述 nop sled 在代码中出现的位置以及 WebKit 可用于查找传递给补丁点的所有参数的位置。位置可能是常量、带加数的寄存器或内存位置。

   LLVM 还会报告每个补丁点上哪些寄存器确实在使用。WebKit 可以使用任何未使用的寄存器，而不必担心保存它们的值。

3. WebKit 在每个补丁点的 nop sled 中发出它想要的任何机器代码。WebKit 稍后可能会以它想要的任何方式重新修补该机器代码。

除了修补点之外，我们还有一个更受约束的内在函数，称为 `llvm.experimental.stackmap`。此内在函数可用作反优化的优化。它向 LLVM 保证，我们只会用无条件跳转到外部代码来覆盖 nop sled。要么不修补堆栈图，在这种情况下它什么也不做，要么修补堆栈图，在这种情况下执行不会落到堆栈图之后的指令。这可以让 LLVM 完全优化掉 nop sled。如果 WebKit 覆盖堆栈图，它将覆盖紧接在堆栈图所在位置之后的机器代码。我们设想“无落到”规则还有其他潜在用途，因为我们将继续与 LLVM 项目合作来完善堆栈图内在函数。

有了补丁点和堆栈图，FTL 可以使用 DFG 和 Baseline JIT 已经用于处理完全多态代码的所有自修改代码技巧。这保证了从 DFG 升级到 FTL 不会减慢速度。

FTL 特定的高级优化

到目前为止，这篇文章详细介绍了我们如何与 LLVM 集成，并设法利用其低级优化功能，而不会失去 DFG JIT 的功能。但是，添加更高级别的 JIT 还使我们能够进行优化，如果我们的分层策略以 DFG 结束，这些优化将无法实现。接下来的部分展示了第四层实现的两项功能，这些功能并非 LLVM 独有。

多变量去虚拟化

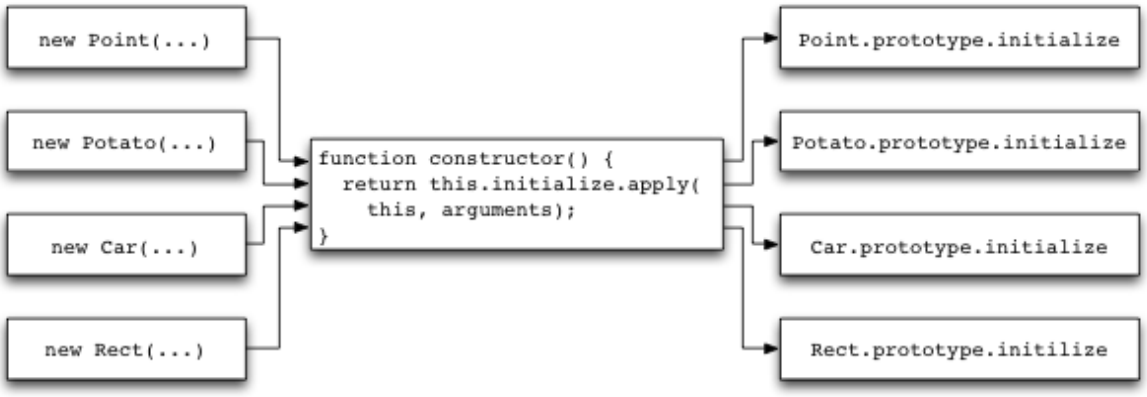

图 9。诸如inheritance.js或Prototype之类的 JS 习语将通过辅助程序（例如本图中的对象构造函数）引导执行。这导致辅助程序看起来是多态的。请注意，如果它是内联的，它就不会是多态的：在调用站点`constructor`内`new Point(...)`联会导致对的调用`initialize`始终调用`Point`的`initialize`方法。

为了理解多变性的强大功能，我们来考虑以下在 JavaScript 中创建类的常用习惯用法：

```
var Class = {
    create: function() {
        function constructor() {
            this.initialize.apply(this, arguments);
        }
        return constructor;
    }
}
```

Prototype等框架使用了这种习语的更复杂形式。它允许通过 来创建类对象`Class.create()`。然后可以将返回的“类”用作构造函数。例如：

```
var Point = Class.create();
Point.prototype = {
    initialize: function(x, y) {
        this.x = x;
        this.y = y;
    }
};
var p = new Point(1, 2); // Creates a new object p suc
```

使用此习语实例化的每个对象都将涉及对 的调用`constructor`，并且构造函数对 的使用`this.initialize`将看起来是多态的。

它可能调用`Point`类的`initialize`方法，也可能调用`initialize`创建此习语的任何其他类的方法。图 9 说明了这个问题。对 的调用看起来是多态的，从而阻止我们在调用 的位置`initialize`完全内联 的构造函数主体。DFG 将成功内联 的主体，但它将在那里停止。它将使用虚拟调用来调用。`Point``new Point(1, 2)``constructor()``this.initialize`

但是想象一下，如果我们要求 DFG 以与 Baseline JIT 相同的方式分析其虚拟调用，会发生什么。在这种情况下，DFG 将能够报告，如果`constructor`内联到表示 的表达式中`new Point()`，则对 的调用`this.initialize`总是调用`Point.prototype.initialize`。我们将此称为多变量分析，这也是为什么尽管 FTL JIT 更强大但仍保留 DFG JIT 的原因之一——DFG JIT 通过在内联后运行分析来充当多变量分析器。这意味着 FTL JIT 不仅比 DFG 执行更多传统的编译器优化。它还可以看到更丰富的 JavaScript 代码配置文件，这使它能够进行更多的去虚拟化和内联。

值得注意的是，拥有两个不同的编译器层（如 DFG 和 FTL）对于获得多变量分析并非绝对必要。例如，我们可以完全消除 DFG 编译器，并在收集其他分析后让 FTL “分层”到自身。我们甚至可以使用相同的优化编译器进行无限次重新编译，使用分析工具编译任何多态习语。一旦该工具发现任何额外优化的机会，我们就可以再次编译代码。这可能有效，但它依赖于多态代码始终具有分析，以防它最终被发现是单态的。分析可能非常便宜 - 例如在内联缓存的情况下 - 但没有免费的午餐。下一节将展示积极“锁定”多态访问的所有路径并允许 LLVM 将其视为普通代码的好处。这就是为什么我们更喜欢使用一种方法，在执行一定次数之后，函数将不再对其任何代码路径进行分析，并且我们使用低延迟优化编译器（DFG）作为分析的“最后机会”。

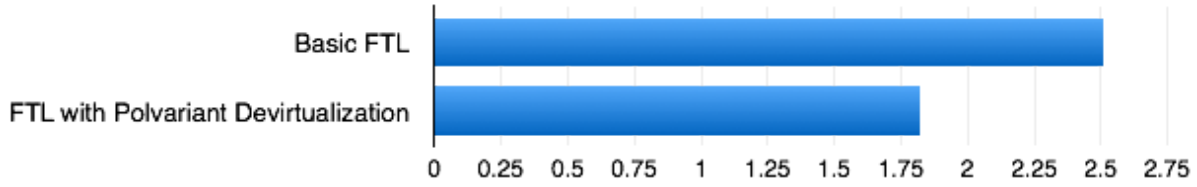

图 10.在 Raytrace 基准测试中，有和没有多变量去虚拟化的执行时间（以毫秒为单位）（越低越好）。

图 10 显示了我们从多变量去虚拟化中获得的基准测试加速之一。此基准测试使用原型样式的类系统，并对构造函数进行多次调用，如果我们可以通过构造函数助手进行去虚拟化，这些调用很容易内联。多变量去虚拟化在此基准测试中将速度提高了 38%。

多态内联

DFG 有两种优化堆访问的策略。要么已知访问是完全单态的，在这种情况下访问的代码是内联的，要么访问被转换为与 Baseline JIT 使用的相同类型的内联缓存。内联访问是有利可图的，因为 DFG IR 可以明确表示访问所需的所有步骤，例如类型检查、任何中间指针（如 Butterfly）的加载（参见图 8）、任何 GC 屏障，以及最后的加载或存储。但如果根据可用的分析认为访问是多态的，那么 DFG 最好使用内联缓存，原因有二：

- 访问可能被认为是多态的，但实际上可能最终是单态的。如果访问最终是单态的，内联缓存可能会非常快，这可能是因为分析不精确。如上一节所述，DFG 有时会看到不精确的分析，因为 Baseline JIT 对调用上下文一无所知。
- FTL 希望对多态访问进行额外的分析，以便执行多变量去虚拟化。多态内联缓存是一种自然的分析形式——我们懒惰地只发出我们知道我们需要的、我们知道看到的类型的代码，因此枚举一组可见的类型就像查看内联缓存最终发出了哪些情况一样简单。如果我们内联多态访问，我们就会丢失这些信息，因此多变量分析将更加困难。

出于这些原因，DFG 将多态操作（即使传入类型数量被认为很少且有限）表示为内联缓存。但 FTL 可以很好地内联多态操作。事实上，这样做会在 LLVM 中带来一些有趣的优化机会。

我们将多态堆访问表示为 DFG IR 中的单个指令。这些指令携带元数据，总结了执行该指令时可能发生的所有可能情况，但多态操作的实际控制和数据流并未显示。这使得 DFG 能够以低成本对多态堆访问执行“宏”优化。例如，DFG 知道多态堆加载是纯粹的，因此可以将其提升出循环，而无需推理其组成控制流。但是，当我们将

DFG IR 降低到 LLVM IR 时，我们将多态堆访问表示为`switch`预测类型的一个，以及每个类型的一个基本块。然后，LLVM 可以对此切换进行优化。它可以采用其众多技术之一来高效地发出切换代码。如果我们连续对同一对象进行多次访问，LLVM 的控制流简化可以线程化控制流并消除冗余切换。更常见的是，LLVM 会在不同的切换案例中找到通用代码。考虑这样的访问`o.f`，我们知道`o`要么具有类型*T*并且"f"位于偏移量 24，要么具有类型*U*并且"f"位于偏移量 32。我们已经看到 LLVM 足够聪明，它将访问变成如下形式：

```
o + 24 + ((o->type == U) < < 3)
```

换句话说，我们的编译器认为的控制流，LLVM 可以转换为数据流，从而产生更快、更简单的代码。

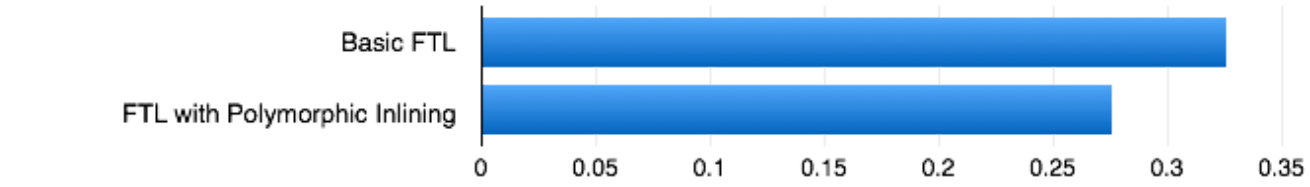

图 11.在 Deltablue 基准测试中，有和没有多态内联的执行时间（以毫秒为单位）（越低越好）。

受益于我们的多态内联的基准测试之一是 Deltablue，如图 11 所示。此程序对具有不同结构的对象执行许多堆访问。揭示这种多态性的性质以及每个字段上所有不同访问模式的特定代码路径允许 LLVM 执行额外的优化，从而将速度提高 18%。

结论

统一 WebKit 和 LLVM 的编译器基础架构是一个非凡的过程，我们很高兴终于在r167958 版本中启用了 WebKit 主干中的 FTL 。从很多方面来看，FTL 的工作才刚刚开始 - 我们仍然需要增加 FTL 可以

编译的 JavaScript 操作集，并且我们仍然有未开发的性能机会。在以后的文章中，我将展示有关 FTL 实现的更多细节以及改进的机会。现在，您可以随时在 WebKit Nightly 中亲自试用。并且，一如既往，请务必提交错误！ ■

下一个

速度计：
Web 应用响应能力的基准

了解更多

之前

WebKit 的 CSS JIT 编译器概述

了解更多