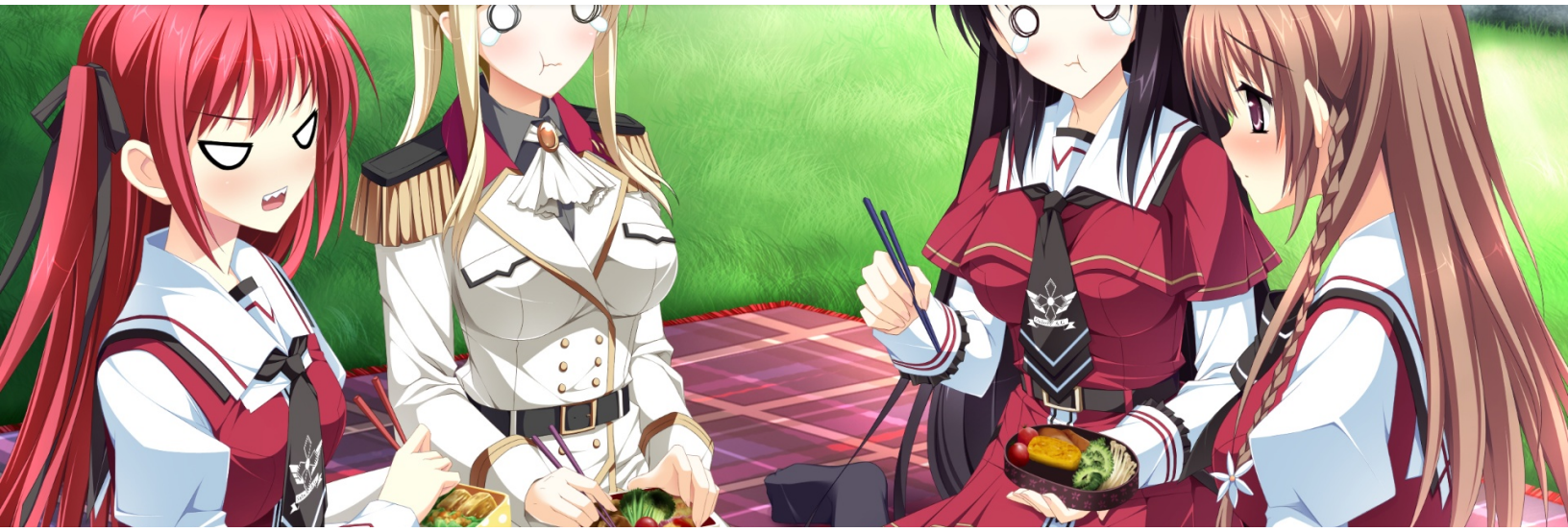

GHC API 系列笔记（2）：关于 Pipeline 与 Hooks

Felis sap...
函数式编程、编程语言、编程 话题的优秀回答者
已关注

上篇地址：GHC API 系列笔记（1）：入门篇 - 知乎专栏

上篇笔记中我们了解到了：

- 使用 GHC API 的意义
- GHC API 相关的几个基本类型与函数的用法：Ghc monad、DynFlags 等
- 如何使用 GHC.load 自动编译和加载 Haskell 源代码
- 如何设置 interactive context 并将字符串动态 eval 到一个 Haskell 值
- 封面妹子很萌

这篇笔记将注意力从 interactive evaluation 转到 compilation pipeline，介绍 GHC 的一些内部原理，然后演示如何在标准的 GHC compilation pipeline 中夹带私货，捞出我们需要的中间表示。

关于 package 的编译和装载

上次 demo 代码里，runGhc 内部的 Ghc monad 最开始的操作是通过 getSessionDynFlags >>= setSessionDynFlags 初始化整个 Session。初始化的过程中 GHC 会自动加载 package database 以及 exposed packages。

DynFlags 类型中有设置 package database 的 field（packageDBFlags），也有设置 package 的 field（packageFlags、ignorePackageFlags 、pluginPackageFlags ），它们都有直接对应的 ghc 命令行参数。平时如果抛开 stack/cabal-install 等基于 Cabal 框架的 Haskell 包管理器，手动调用 ghc 去编译和注册一个 package 的话，每次在编译单个 module （-c 选项）或者一群

sandbox，防止与其他项目的依赖相互污染），以及从这些 package database 中 expose 出哪些 package（否则在 type-check 时找不到依赖的 module）。

默认情况下，全局 package database 总是被加载的，内含 ghc boot libraries（如 ghc-prim、base 等标准库），这些 ghc boot libraries 跟 Hackage 上的第三方 package 相比有着老干部待遇：不能撤换和重编译；package id 是 “wired-in” （焊死）的，第三方 package 在 package database 里注册时，其标识符为 “name-version-hash”，而 boot libraries 不需要 hash 也能引用。除了全局 package database 以外，stack/cabal-install 等工具会通过环境变量/命令行参数等方式加载由工具自行管理的 package database（比如一个 stack snapshot，或者 cabal sandbox）。

这里的 package 指的是 package known by GHC，而不是 package known by Cabal，虽然前者的 metadata 是后者的子集（ghc 娘：Hackage 是啥？能吃吗？）。GHC API 中，PackageConfig/Packages 模块实现了 package 相关的类型和操作，同时 ghc-boot 库中包含了一些 ghc 和 ghc-pkg 工具共享的代码，使用它们即可手动暴力实现 Haskell 包管理器。

最后，GHC 8.2 起还有 Backpack module，不过现在估计能熟练使用这个特性的只有 Edward Yang 了，以后我会用了再单独写笔记吧……

本篇笔记并不涉及多重 package 的同时编译和链接，不过了解一下 package 相关的一点架构知识还是有必要的——如果需要基于 GHC API 定制自己的 Haskell 编译器，为了用户体验，跨 package 的编译几乎肯定要做。

关于 module 的 compilation pipeline

现在，我们要编译和装载一个（或者一群） module 。上次的 demo 代码中，初始化完了以后，我们只做了两件微小的工作：调用 setTargets 设定当前 Session 的编译目标；调用 load 启动整个 pipeline。这个 pipeline 具体做了什么工作呢?

首先是 dependency tracking。我们在设定 target module 时，仅仅指定了顶层的 target，这些 target 的源代码里，可能会 import 一些其他未指定为 target 的 module，这些 module 同样需要计入 target list。所以 load 会首先调用 GhcMake.depenal ，进行依赖分析，分析的结果进行拓扑排序，以决定编译顺序（一个 module 在 type check 阶段要求所有依赖的 module 已经 type check 通过并生成 interface 文件）。

值得注意的是，依赖分析过程中会对 Haskell 源代码进行预处理以及并不完整的解析（需要拿到 import list）。这个预处理和解析的工作，与之后的 module pipeline 中的预处理和解析是相互独立的，有轻微的重复计算。这算是 GHC 架构上的一个小小缺陷。

接下来是按正确的顺序，为每个待编译的 module 启动一个 pipeline。负责单个 module 的 compilation pipeline 的顶层接口在 DriverPipeline 模块中，核心逻辑是 runPhase 函数。runPhase 函数的类型签名是 PhasePlus -> FilePath -> DynFlags -> CompPipeline

DynFlags 的原因很简单——因为编译每个 module 时不能使用 Session 的全局 DynFlags 了，不同 module 可能会指定不同的编译选项。runPhase 会反复地迭代，直到返回 StopLn 正常终止，代表该 module 的编译工作告终（或者检查 timestamp 以后发现根本不用编译）

一般情况下，一个 module pipeline 会经历以下的流程：

1. Unlit 预处理。对于 Literate Haskell module，会调用一个 GHC 自带的 unlit 程序，将其恢复为普通的 Haskell module。
2. C 预处理。Haskell 有 CPP 扩展，可以使用 C 预处理器宏，所以对用到 CPP 的 module，需要先调用系统的 cpp 进行展开。
3. 文法解析。这一步会生成 HsModule RdrName。在 GHC 的架构中，凡是带有 identifier 的 AST 类型，全部使用一个类型变量来支持重载不同类型的 identifier，比如这里刚从 parser 出来的热乎着的 AST，带上的 identifier 类型基本上就是个（来自源代码的）字符串。
4. 作用域解析（renaming）。在函数式语言中，如何在作用域安全的前提下表示 AST 是个老生常谈的话题了，考虑到 AST 是从文本解析出来的，HOAS/PHOAS 肯定不现实，而 first-order 的表示中，最常见的是 de-bruijn indices，用绑定层数来替换变量名。而 GHC 采用的是 Barendregt convention，简而言之，给所有的绑定变量名加上唯一的后缀，保证在全局的命名空间内不冲突。《Secrets of the Glasgow Haskell Compiler inliner》一文中有介绍怎样优化这个 renamer 的性能。
5. 类型检查。
6. 转换到 Core（desugaring）。Haskell 本身的语言特性（与 AST）都十分复杂，所以在进一步的分析和优化之前，会转换到一个极小的核心语言（Core），Core 的 AST 和 typing rules 都比较简单。desugar 之前，Haskell AST 已经通过类型检查，而在 Core 层面，Core 也有自己的类型检查系统（CoreLint），作为一门强制标注类型的 System F 方言，CoreLint 能够确保“经过复杂变换以后，只要通过 CoreLint，就不会在运行时 segfault”。
7. Core 优化（simplifying）。重量级的 Haskell 编译优化都是在这一步实现的。
8. Core 后处理（tidying）。经过优化的 Core，在后处理之后，可以作为 interface file 被序列化到文件，日后在其他 module 依赖这个 module 时，GHC 可以直接读取 interface file，加载这个 module。
9. 转换到 STG。首先通过 CorePrep，将 Core 转换到 ANF（A-Normal Form）形式，然后通过 CoreToStg 将其转换为 STG。STG 详细可以参考 How to make a fast curry: push/enter vs eval/apply 一文，可以看作 Haskell 的操作语义定义；转换到 STG 以后，类型信息被大多数舍去，只留下足够指导代码生成的信息。
10. 转换到 Cmm。Cmm 是“C--”，最初（在 LLVM 未诞生的时代）作为一种“可移植的汇编语言”提出，作为生成机器码之前使用的底层中间表示。
11. 根据 DynFlags 中指定的后端，从 Cmm 分出不同路径：默认情况下是 assembly 后端，从 Cmm 生成平台汇编码，然后调用 assembler 生成 object file；古代的 GHC 默认是 C 后端，直接将 Cmm 按照 C syntax 序列化以后调用 gcc 生成 object file；而 LLVM 后端则会将 Cmm 转换为 LLVM 的 SSA IR，然后调用 llc 和 opt 编译和优化，最后生成 object file。
12. 链接。可以生成可执行文件或静态/动态链接库。

的文档看起，在依赖分析完成后，对每个 target 手动把整个 pipeline 一步步执行一遍（注意避免重复编译）

观察 GHC pipeline

文档和 wiki 太长？没有注释的代码看到抓狂？与其看我在专栏发萌图 & 瞎扯淡，不如自己做实验观察学习 GHC 的工作原理。

首先是 verbosity 选项。初始化 Session DynFlags 时，将 verbosity 设为 3 就好，输出的调试信息量适中，4 和 5 的话会把各种 GHC 的中间表示混在调试信息里输出，干扰视线。

需要阅读 GHC 生成的中间表示也很容易——GHC 提供了许多 -ddump-xx 选项，从前端到后端无所不包，如果使用 GHC API 的话，在 DynFlags 模块里，有专门的 DumpFlag 类型可以控制 dump 哪些中间表示。

从 GHC pipeline 获取中间表示

说了这么半天，一开始的目的是啥来着？哦，为了实现与 GHC 共享语言特性的 Haskell 编译器，我们需要从 GHC pipeline 中捞出中间表示，然后走自己的代码生成路径。大致有哪些方法呢？

- 命令行调用 ghc，使用 -ddump-xx 选项将中间表示 dump 到文本文件，然后手动解析。别笑，如果打算使用 Haskell 以外语言实现 Haskell 编译器，这是唯一的选项——最近两天刚刚开源的 IntelLabs/flrc 就是使用 SML 实现的 Haskell 编译器，所以他们还做了 Core 的 parser。
- 自己维护一份 ghc fork，加入自定义后端自然就不是个事了。不过呢，个人的努力固然重要，也要看 upstream 的进程。。（

所以最现实的选项仍然是设法使用 GHC API，毕竟直接操作结构化的数据是最舒服的。现有的 GHC API 框架内，我们的选项大致包括：

- 手动调用 HscMain 中的各个 entry function，运行不同的 phase。这样做的好处是我们可以做得比 ghc --make 更好一些，比如缓存预处理和依赖分析的结果，etc，同时能够取得经过任意 phase 的中间表示。缺点是工作量较大。
- 使用 Plugins API。目前的 Plugins API 分为 3 类：Frontend Plugin、Core Plugin、Typechecker Plugin。Plugins 的话题又大了，等后续的笔记吧（如果有的话
- 使用 Hooks API。工作量可大可小，下面讲一个例子。

Hooks API 的定义在 Hooks 模块中。DynFlags 默认初始化时会设置为 emptyHook，所以初始化时可以注册 Hooks 模块中提到的不同种类的 Hook。

我们最关心的 Hook 是 runPhaseHook，这个 Hook 可以用来劫持默认的 runPhase，控制 module pipeline。考虑到 runPhase 的原版实现有 600+ 行，直接复制粘贴然后魔改太费劲，所

```
import Control.Monad
import Control.Monad.IO.Class
import qualified DriverPipeline as GHC
import qualified DynFlags as GHC
import qualified GHC
import qualified GHC.Paths as GHC
import qualified Hooks as GHC
import qualified HscTypes as GHC

data Flags = Flags
    { importPaths :: [FilePath]
    , targets :: [String]
    , fatalMessager :: GHC.FatalMessager
    , coreHook :: GHC.CgGuts -> GHC.ModSummary -> IO ()
    }

defaultFlags :: Flags
defaultFlags =
    Flags
    { importPaths = []
    , targets = []
    , fatalMessager = GHC.defaultFatalMessager
    , coreHook = \_ _ -> pure ()
    }

run :: Flags -> IO ()
run flags =
    GHC.defaultErrorHandler (fatalMessager flags) GHC.defaultFlushOut $
    GHC.runGhc (Just GHC.libdir) $ do
        dflags <- GHC.getSessionDynFlags
        void $
            GHC.setSessionDynFlags
                dflags
                { GHC.verbosity = 3
                , GHC.optLevel = 2
                , GHC.importPaths = importPaths flags ++ GHC.importPaths dflags
                , GHC.hooks =
                      GHC.emptyHooks {GHC.runPhaseHook = Just run_phase_hook}
                , GHC.language = Just GHC.Haskell2010
                }
        sequence [GHC.guessTarget t Nothing | t <- targets flags] >>=
            GHC.setTargets
```

```
where
  run_phase_hook
      :: GHC.PhasePlus
      -> FilePath
      -> GHC.DynFlags
      -> GHC.CompPipeline (GHC.PhasePlus, FilePath)
  run_phase_hook phase_plus input_fn dflags = do
      result@(_, output_fn) <- GHC.runPhase phase_plus input_fn dflags
      liftIO $
          case phase_plus of
              GHC.HscOut _ _ (GHC.HscRecomp cg_guts mod_summary) -> coreHook flags c
              _ -> pure ()
      pure result
```

我们定义了一个 Flags 类型，以及一个 run 函数，类型为 Flags -> IO ()。可以理解为一个 Flag 代表一次编译任务，编译一个/多个 module。Flags 中可以指定除当前目录外的其他 import path（即可支持将代码分散在不同目录中）、本次编译的顶层 target（可以是 module name 或者 file name），编译如果顺利完成，GHC 会将 interface file 和 object file 也写到 Haskell 源代码的同一目录，下次编译会自动读取 interface。

为了获取我们想要的 Core 代码，我们的 Flags 包含一个 callback，其参数包含了 module summary（用于辨识 module 身份）以及经过 simplifier 优化、即将交给 code generator 的 Core module。在初始化 Session DynFlags 时，我们将自己的 run_phase_hook 注册到 hooks 中，这个 run_phase_hook 多数情况下会安安静静地调用普通的 runPhase 完成任务，但是当检测到“有个 module 被编译到 Core 了！新鲜热乎的 Core 要交给后端了！”的时候，会自动调用 Flags 中的这个 callback，报告新发现的 Core 的身份和完整代码。

通过 Hooks API，我们可以免于手动实现 make 相关逻辑的困扰，通过以上方式拿到 Core 代码用于我们自己的 codegen。如果希望拿到其他的中间表示，而该中间表示被 runPhase 函数内部吞掉了的话，我们可以用同样的思路，部分移植 runPhase 的逻辑——在不关心的阶段，直接调用 runPhase，而在需要拿到中间表示的阶段，再手动实现调用 HscMain 相关函数的逻辑。

相关轮子

- ghc-simple: Simplified interface to the GHC API：这是 haste-compiler 背后的框架，通过 GHC API 获取 Core/STG 表示，且实现了缓存功能。
- puellascript/Run.hs at master · TerrorJack/puellascript：这是目前我手上的一个坑，今天刚实现从 GHC pipeline 中导出所有种类的中间表示：Core (simplified / ANF)、STG (from Core / simplified)、Cmm (from STG / simplified / raw)，以供第三方 codegen 使用。缓存管理还没有做，用的是 ghc --make 的默认 pipeline，所以未修改模块在重编译时，相应的 dumper hook 不会触发。

参考资料

git.haskell.org - ghc.git/summary

编辑于 2017-04-15

「真诚赞赏，手留余香」

赞赏

还没有人赞赏，快来当第一个赞赏的人吧！

Haskell GHC（编程套件） 函数式编程

文章被以下专栏收录

推荐阅读

NeverMoes 1 年前

配图？？？某公司的女儿系列。

赞

Alexander Melody 1 年前

我以为推荐了某番更新

赞

胡必武 1 年前

封面妹子很萌。

赞

WiDeCourage 1 年前

新手上路，竟然不懂配图是谁

赞