GHC API 系列笔记（1）：入门篇

Felis sap...
函数式编程、编程语言、编程 话题的优秀回答者

为什么需要 GHC API

- 实现 Haskell 的 eval 功能 —— 不是 Template Haskell 那种躲在编译期运行的弱鸡扩展，而是最混沌、最不安全的那种，输入 String、输出 Haskell 值的 eval，而且可以动态检视其类型。在实现诸如 Online REPL、Online Judge 等面向 Haskell 学习者的工具时相当有用。
- 实现 Haskell 模块的热切换。Lisp 能做，Erlang 能做，凭什么 Haskell 不能做（
- 捞取 GHC Pipeline 中的各种中间表示，从而将 GHC 的前端挪用来实现自己的 Haskell 编译器。像 haste 和 ghcjs 就依赖 GHC API 实现从 Haskell 源代码编译到 STG 的逻辑，然后通过自己的代码生成器从 STG 编译到 JavaScript。当然，想要用 Haskell 搞前端开发，除了真的开坑做 Haskell 的 JavaScript 编译器以外，另一条道路是设计用 Haskell 来表示 JavaScript 的 monadic DSL，具体做法及其优缺点得等我另外写一篇讲 remote monad 设计模式的专栏了。
- 实现 GHC Plugin。可以实现自定义的 Core optimization pass 或 type checker，比如 liquidhaskell 通过 Plugin API 调用 SMT solver，为 Haskell 实现 refinement type 系统。
- 实现 Haskell 的 IDE，比如实时类型检查、自动补全等。Haskell IDE 不发达，我们都要支援它（

GHC 的架构中，大多数类型和相关函数都通过 GHC API 暴露给用户，而顶层的 ghc 程序只是一个 GHC API 的 wrapper 而已，这使得 GHC 能做到的所有事情，我们可以通过调用 GHC API 来做到。这里通过一个将 Haskell 代码用 native code generator 编译到 object code 并加载后对 String 进行 eval 的例子，演示 GHC API 的基本用法、一些相关类型与函数的含义。

下面这段代码，从 ./test/case/Fact.hs 中编译并加载 Fact 模块，然后将表达式 "fact 5" 求值后输出。

```
import Control.Monad.IO.Class
import Data.Functor
import DynFlags
import GHC
import GHC.Paths
import Unsafe.Coerce

main :: IO ()
main =
    defaultErrorHandler defaultFatalMessager defaultFlushOut $
    runGhc (Just libdir) $ do
        dflags <- getSessionDynFlags
        void $
            setSessionDynFlags
                dflags
                { ghcLink = LinkInMemory
                , hscTarget = HscAsm
                , importPaths = ["./test/case"]
                }
        setTargets
            [ Target
              { targetId = TargetFile "./test/case/Fact.hs" Nothing
              , targetAllowObjCode = True
              , targetContents = Nothing
              }
            ]
        void $ load LoadAllTargets
        setContext [IIDecl $ simpleImportDecl $ mkModuleName m | m <- ["Fact"]]
        v <- unsafeCoerce <$> compileExpr "fact 5"
        liftIO $ print $ (v :: Int)
```

./test/case/Fact.hs 的内容（一个简单的阶乘函数）：

```
module Fact where

fact :: Int -> Int
fact n
    | n < 0 = error "Expects non-negative n"
    | n == 0 = 1
    | otherwise = n * fact (n - 1)
```

```
$ stack ghc -- ghci.hs -package ghc -package ghc-paths
[1 of 1] Compiling Main             ( ghci.hs, ghci.o )
Linking ghci ...

$ ./ghci
120

$ ls ./test/case
Fact.hi  Fact.hs  Fact.o
```

可以看到表达式求值的结果。./test/case 目录中新增的 .hi 和 .o 文件是 native code generator 编译 Fact.hs 的产物。

demo 代码讲解

首先，如果需要用到 GHC API 的话，我们的依赖会增加两个包：ghc 和 ghc-paths，如果是 Cabal project，需要将其补充到 .cabal 中。ghc 包是 GHC 在 booting 阶段自动生成的，其版本号与 GHC 版本号严格对应，并不在 Hackage 上提供下载和文档，也并不支持通过普通的 Cabal 机制编译安装。ghc-paths 包是一个简单的 wrapper，通过 C 预处理器扩展来提供“编译 ghc-paths 所用到的 GHC 相关路径”，这个路径在使用 GHC API 时需要用到。

ghc 包的文档可以在线查看，也可以查看本地的版本（通过 stack 安装的 ghc 在 ~/.stack/programs 中自寻，而通过 hvr/ghc 安装的 ghc 需要单独安装 ghc-$GHCVER-htmldocs 包，$GHCVER 为 GHC 版本号）

ghc 包中的模块较多。入门时首先从 GHC 模块的文档看起。首先关注两个类型——Ghc 和 GhcT，以及一个类型类 GhcMonad 。GhcMonad 抽象了“维护 GHC Session、支持异常处理和日志输出、支持嵌入 IO action”的功能，几乎所有 GHC API 的接口基于 GhcMonad 提供，而 Ghc 和 GhcT 是提供了 GhcMonad 功能的 monad 和 monad transformer。在这个简单的 demo 中，我们只用到了 Ghc 类型。

运行一个 Ghc action 需要使用 runGhc 函数初始化一个 GHC Session，函数参数为 ghc-paths 提供的 GHC library directory。GHC Session 的类型是 HscEnv，这个 Session 不可以被多线程使用，但是单个 Haskell 程序可以初始化多个不同的 GHC Session 运行。runGhc 以外还需要一个 wrapper 来安装 exception handler，没有特殊需求的话使用 default 开头的默认 handler 即可。

接下来解释 Ghc monad 中做了什么工作。首先，我们需要设置 DynFlags 。一个 GHC Session 包含两个 DynFlags，分别代表 program/interactive 模式下的 GHC 设置，通过命令行调用 GHC 时的一堆 flags 的用途就是设定 DynFlags。GHC Session 初始化时默认的 DynFlags 值通过 defaultDynFlags 设定，不过这时的 DynFlags 值并不能直接用于后续的其他 GHC API 接口，需

package database）。getSessionDynFlags/setSessionDynFlags 会同时对 program/interactive DynFlags 进行设置，除此之外我们也可以单独设置 program/interactive DynFlags，这里使用 getSessionDynFlags/setSessionDynFlags 是为了省事。

我们的 demo 中对 GHC 选项稍作了一些修改。具体有哪些选项可以参考 DynFlags 的类型定义，这里我们指定使用 native code generator 后端、使用 in-memory linker，并将 Fact.hs 所在的目录加到 import 路径列表中。

后端通过 HscTarget 类型指定，除了 native code 后端以外，我们还可以选择 C 后端、LLVM 后端、bytecode 后端、啥也不生成后端。C 后端就是将 Cmm 代码转换为 C 然后调用 gcc 编译和链接，是古代的 GHC 使用的默认后端，优点是在将 GHC port 到未知平台上时有用，缺点是后端自身性能和生成代码性能都不如 native code 后端。LLVM 后端通过命令行调用 LLVM 的 llc 和 opt 工具，将 LLVM IR 进行编译和优化，优点是生成代码的性能较高，尤其是在进行数值计算的 Haskell 代码上提升明显，能够使用 SIMD primop；缺点是后端自身较慢，而且不同 GHC 版本支持的 LLVM 版本不一。bytecode 后端生成 ghci 解释器专用的 bytecode，速度较快。而啥也不生成后端的用途就是做类型检查，在实现 IDE 时有用，不过如果代码用到了 Template Haskell 就不行了，毕竟需要走 2 个 pass，第一个 pass 总不能啥也不生成。

在其他 GHC API 的 tutorial 中，需要演示实现 eval 时，通常选择 bytecode 后端，这里我选择 native code 后端以保证之后加载的模块行为与单独编译时完全一致，另外后端生成的 native code 性能比 bytecode 更佳。不过这样一来就无法使用 ghci 的断点调试了，demo 没有用到这个功能所以无妨。linker 选项的含义参考 GhcLink 说明，这里选择 in-memory linker，因为不需要生成 executable 或者 static/dynamic library。另一个选项是 importPaths，默认为当前目录，因为 Fact.hs 放在 ./test/case 目录中，所以需要加上这一项。

有一个选项这里没有设置，使用默认值：GhcMode。通过命令行调用 GHC 编译 Haskell 代码时，一般有 2 种模式选择—— one-shot 模式，以及 make 模式。one-shot 模式通过 -c 选项使用，用途为编译单个 Haskell 模块；make 模式通过 --make 选项使用，用途为将指定 Haskell 模块及其依赖模块一次性编译。通过 make 模式进行编译的性能，超过 one-shot 模式+第三方 make 工具，因为即使在 one-shot 模式下，ghc 也会自行做 dependency tracking，在内建的 make 模式下工作时可以避免许多冗余的 tracking 工作。这里我们使用默认的 make 模式，如果有其他未编译的模块被 Fact 依赖，它们会被一并编译。

以上是 GHC API 初始化相关的工作。接下来我们需要指定 GHC 的 target。setTargets 函数设定当前的 Target list，至于获取 Target 的方法，可以像 demo 中一样，通过 Target 类型定义手动指定，或者通过 guessTarget 函数生成。

指定完 target 以后，就可以启动 GHC Pipeline 进行编译和载入了。可以手动运行 dependency tracking、preprocessing、parsing、type checking、desugaring、code generating、linking 等 pass，不过最简单的方法是使用 load 函数自动运行整个 pipeline。load 成功后，Fact.hs 已经被编译完毕，interface file 和 object file 在同一目录下生成，可以加载。

的声明，需要手动将其加入。setContext 支持两种不同类型的 import，通过 InteractiveImport 类型指定，其中 IIDecl 将 module export 的定义 import 到当前 context，而 IIModule 则将整个 module 源代码的顶层环境 import 到当前 context。后一种 import 对应 ghci 中的 :load 命令，如果我们选择通过 bytecode 模式加载 Fact，则选择后一种，否则选择前一种。

interactive context 设置完毕后，总算可以做 ghci 也能做的各种事情了——这里先 eval 一个表达式试试看。compileExpr 将一个 String 编译成我们想要的 Haskell value。返回的类型是 HValue，而 HValue 是 Any 的 newtype wrapper，在我们知道表达式类型的前提下，可以通过 unsafeCoerce 强制转换回我们需要的类型。

如果有洁癖，觉得 unsafeCoerce 太辣眼，那么在表达式类型非多态、是 Typeable 实例的前提下，可以使用 dynCompileExpr 将 String 编译成 Dynamic。Haskell 有一定程度的动态类型支持，通过 Typeable class 实现运行时类型信息，通过 Data.Dynamic 提供了带运行时类型检查的普通 Haskell 值与 Dynamic 之间的转换。

demo 讲解完毕！

其他 GHC API 使用姿势

- 刚才的 demo 中，我们通过调用 load 函数，一次性完成了整个编译 pipeline。有时我们并不希望运行整个 pipeline，而是希望运行到某一个 pass 为止，将输出的中间表示进行处理，比如实现自己的 Haskell 编译器。为了在 pipeline 中捞到我们想要的中间表示，我们可以选择手动实现自己的 load，不过工作量不小，而 GHC API 提供了 Hooks 机制，就像先前介绍的 Cabal hooks 一样，可以通过初始化时在 DynFlags 中安插一些 callback，从而 load 过程中运行到特定 pass 时会触发 callback，允许我们获取和处理中间表示。详细可以参考 Ghc/Hooks - GHC
- 从 GHC 8 开始，ghci 提供了一个“remote interpreter”功能，相关代码来自 Luite Stegeman 在 ghcjs 项目中的相关工作，可以通过跨进程的 RPC 启动单独的 ghci 进程进行计算。remote interpreter 的特点参考 GHC 文档，这个 demo 里使用的是 old-style 的 ghci API，如果希望通过 remote interpreter 实现 interactive evaluation，可以参考 ghci 包的相关文档。
- GHC 提供了 Plugin API，包含 frontend plugin、typechecker plugin、core plugin，分别可以在 init 、type check、desugar 三个阶段搞事情。考虑到 Cabal 架构的复杂性，实现 plugin 一般比实现 standalone 的调用 GHC API 的程序工作量更小。

相关轮子

有不少人基于 GHC API 做了一些 wrapper 和有趣的应用，这里列举几个典型的：

- hint 和 mueval。实现 eval 功能，后者特地增强了安全性，在 Try Haskell! An interactive tutorial in your browser 有用到
- ghc-simple。实现了“带缓存的将一系列 Haskell 源代码编译到 Core/STG/Cmm”的功能，

我自己也写了一个 wrapper（还没有上 Hackage，施工现场在 TerrorJack/ghc-bones），目前提供的功能有：

- SessionT。这是一个提供了 GHC API 功能的 monad transformer，与 ghc 自带的 GhcT 相比，特点是与 mtl-style 的 monad transformer stack 兼容性好，实现了不少有用的 instance，同时调用相对简单一些，只要在 SessionPref 中指定好各种 handler 和选项即可；
- eval 函数。这个 eval 函数将单个 module 源代码在临时目录中编译成 object code 并加载到 interactive context，在该 context 下对表达式进行求值。支持限制求值所用时间和内存，同时会将表达式求值到 normal form，避免 time leak。这个 eval 是打算给以后的 Online Judge 坑用的。

具体用法可以参考 test suite 里的 test case。

参考资料

ghc（library）文档：ghc-8.0.2: The GHC API

ghc（compiler）文档：Welcome to the GHC Users Guide

The Architecture of Open Source Applications 书中 GHC 的章节，有点老不过仍然推荐：The Glasgow Haskell Compiler

GHC wiki，深入了解 GHC：Commentary - GHC

另一篇不错的 GHC API 教程：GHC API tutorial

Stephen Diehl 的 GHC API 教程：Dive into GHC: Pipeline

Edward Z. Yang 的 GHC API 集成 Cabal 教程：How to integrate GHC API programs with Cabal

祝大家。。什么节快乐好呢（沉思

编辑于 2017-04-14

Haskell GHC（编程套件） 函数式编程

文章被以下专栏收录

不动点高校现充部
一切与编程语言理论、函数式编程相关的杂谈。

推荐阅读

「每日一题」什么是 API?

如果你不知道 API 是什么，说明你在语言真的很差。 API 就是 Application Programming Interface （应用 编程 接口）三个单词，如果你不能顾名思义的话，我就举例说明。1. DOM APIDO...

与应杭 发表于前端学习指...

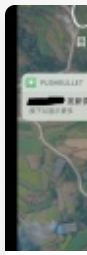
再谈 API 的撰写 - 子系统

陈天 发表于迷思

我的RxJava源码解读笔记

RxJava是一个用于处理异步任务的库，本篇文章我将把我在学习RxJava源码时的分析思路记载下来，一方面用来加强记忆，另一方面可以供大家参考。首先梳理一下RxJav主要功能的工作流程，然后...

靖然是你 发表于靖然有这样...



通过 准实

xlzd

17 条评论

切换为时间排序

写下你的评论...

Felis sapiens (作者) 回复 祖与占 1 年前

1. GHC API 里就有 DynamicLoading 模块；2. 这项目从我初学 Haskell 起就在了，现在还是一坑；3. 反正草稿列表里有一堆（其实只是想发图，顺便带点代码和讲解

1 查看对话

以上为精选评论

纯纯 1 年前

开头列的目的看起来好混乱邪恶。。

赞

> 实现 Haskell 模块的热切换 据 Simon Marlow 说他们用 Haxl 那个系统就可以热加载啊...据说是用 Linker 搞的
> 实现 Haskell 的 IDE，比如实时类型检查、自动补全等。Haskell IDE 不发达，我们都要支援它 Haskell 的 Language Server Protocol 支援需要你 > 另一条道路是设计用 Haskell 来表示 JavaScript 的 monadic DSL，具体做法及其优缺点得等我另外写一篇讲 remote monad 设计模式的专栏了 坐等

赞

Hypnoes Liu 1 年前

stack的文档太少了。我说的不是中文文档，我说的是所以语言的版本都很少。。。

赞

Hypnoes Liu 1 年前

其实想过给VScode做个stack插件来着……

赞

Belleve 1 年前

赞美

1

考古学家千里冰封 1 年前

IntelliJ有两个Haskell插件都可以做到运行，但是都是用命令行调用ghc运行的Orz
另外楼上修仙功力惊人

赞

凉宫礼 1 年前

相當於shell裏在直接運行ghc -c, runghc 之類的?

赞

圆角骑士魔理沙 1 年前

投稿吧0 0

赞

李约瀚 1 年前

新的IDE就叫金坷垃吧。。滑稽，，逃

赞

stack的文档在Haskell世界里算好的了（

赞 查看对话

开瓶少女 1 年前

这个时候、当然是复活节了

赞

Felis sapiens (作者) 回复 凉宫礼 1 年前

并不一样，可以直接操作结构化的数据

赞 查看对话

Felis sapiens (作者) 回复 圆角骑士魔理沙 1 年前

再说吧，感觉写得不太完整。。（

赞 查看对话

swordfeng 1 年前

在有.hi和.o，没有.hs的情况下怎么把模块载进来呢？我试图把.o当做target并没能成功

赞

Felis sapiens (作者) 回复 swordfeng 1 年前

加载模块需要的是.hi

赞 查看对话

白羽飘 1 年前

为什么我总是get不到题图要表达的东西，囧，还需继续学习

赞