计算缓存、优化算法和加速 Python 执行 第三部分

用列表计算进行优化

Python 性能的确一般，但是 Python 的常规函数的性能还是不错的， Python 解释器是用 C 语言实现的，在基本操作层面，已经优化了很多年，所以虽然有 requests 这样石破天惊的扩展包来取代 Python 内部的函数包，但是基本上没有人会去自己实现 Python 的列表计算等最常规的功能。在我们实践中的大部分场景，包括以 server 服务为主的场景，Python 的性能、稳定性都非常在线，也没有 Java 的 JVM 带来的很多不可控制的问题。我们用程序是去解决更多的业务逻辑，所以易学、易维护这些特性也是要关注的。

我们来看看用列表操作实现斐波那契数列的版本：

```
def list_fib(n):

    list_f = []
    f = 1
    list_f.append(f)
    list_f.append(f)
    for i in range(n-2):
        f = list_f[-1] + list_f[-2]
        list_f.append(f)
    return f


if __name__ == "__main__":
    x = 47
    start = time.time()
    res = list_fib(x)
    elapsed = time.time() - start
    print("Python Computed fib(%s)=%s in %0.8f seconds" % (x, res,
elapsed))
```

执行结果就是秒开：

```
Python Computed fib(47)=2971215073 in 0.00002408 seconds
```

基本是所有执行方法中最快的之一，因为执行几十次的列表运算对于 Python 来说实在很轻松。

Python 的列表运算我们在《Python 机器学习》一书中有专门章节讨论过，列表是 Python 中非常重要的数据类型，不能小看，像这样一个数学运算的场景，用列表也可以简单而高效的完成。

谈谈 PyPy

CPython，也就是我们平时使用的 Python 解释器，因为这个解释器是用 C 写的，所以我们称之为 CPython，它并不是 Python 的唯一发行版，比如还有 PyPy http://pypy.org ，它通过即时编译器（JIT）来加快代码执行速度。在之前斐波那契数列的计算例子中有过比较，要比使用 CPython 快 5 倍左右。PyPy 完全支持 Python 标准库，但它不是支持所有的第三方扩。其平均可以提升 2-5 倍性能，并且你的应用所使用的扩展包都兼容，那么 PyPy 是可以尝试一下的。

维基百科中文版 https://zh.wikipedia.org/wiki/PyPy 关于 PyPy 的资料比较老，可以参考英文版 https://en.wikipedia.org/wiki/PyPy。

PyPy 是 Armin Rigo 开发的，一己之力，能做这么多，真的很厉害！

PyPy 的前端是个严格的 Python 子集，称之为 RPython，这里的 R 是严格(Restricted)的意思。RPython 对 Python 语言做了一些约束，以便在之后编译时可以推断出变量类型等诸多优化的前提。PyPy 项目开发了一个工具链，用于分析 RPython 代码并将其转换为字节代码形式，最后用 C 语言来编译，达到将大部分代码编译成机器代码以提高执行速度的目的。它有设计精良的垃圾收集器和内存管理。最后，它包含了 JIT 引擎，在解释器层面上构造“即时的优化代码”。

概念很多，我们一一道来。

什么是 RPython

RPython 有自己的文档站点：https://rpython.readthedocs.io/en/latest/

你写的 Python 程序首先会经过一个 PyPy 的解释器来生成 RPython 代码，前面说了，生成的 RPython 代码比较严格，这样后面的编译工作就好开展，真正的 CPython 是用 C 写的解释器，但是并没有将你的 Python 代码编译成机器码，所以 Python 一般情况下会被诟病执行速度慢。

所以说 PyPy 实际上是包含两个组件：

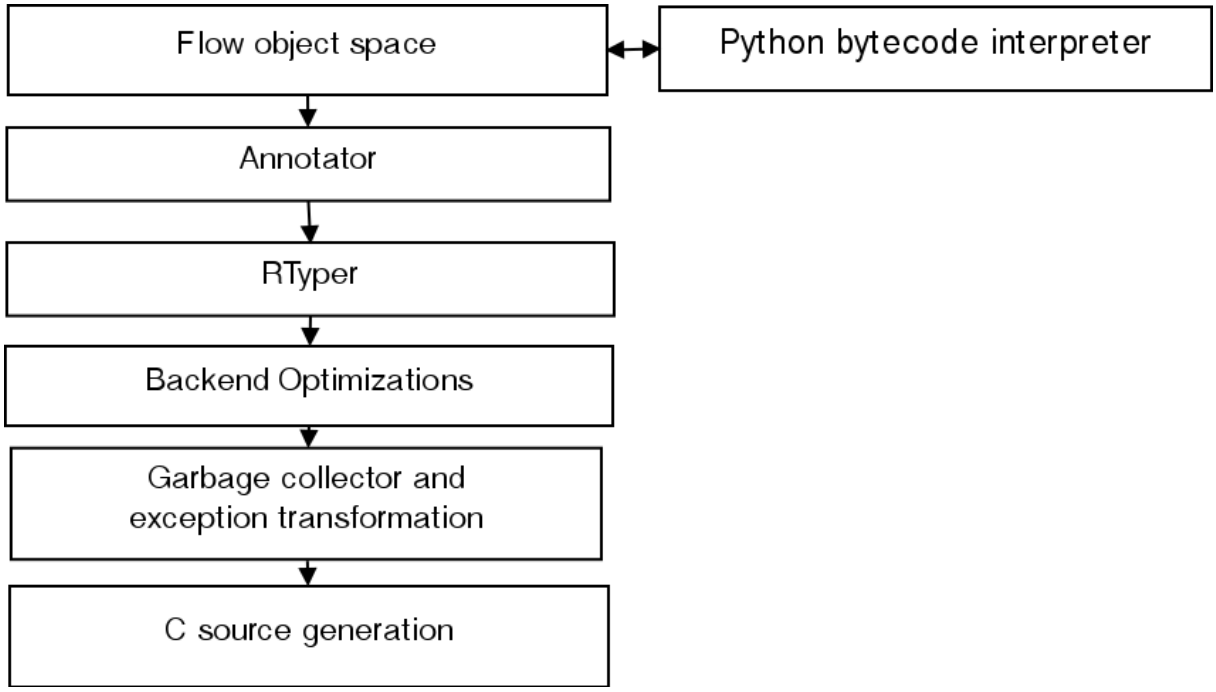
1. Python 解释器，输入是常规的 Python 程序，输出 RPython 代码，且这个 Python 解释器本身使用 RPython 写的。

2. RPython 编译工具链，包括将 Python 代码转换成字节码和 C 代码，编译成本地二进制代码的一系列工具和步骤。

所以大家理解为什么说 PyPy 对 Python 兼容性很好，但是安装一些复杂的本身通过 C 优化过的扩展包兼容性又不太好的原因了。

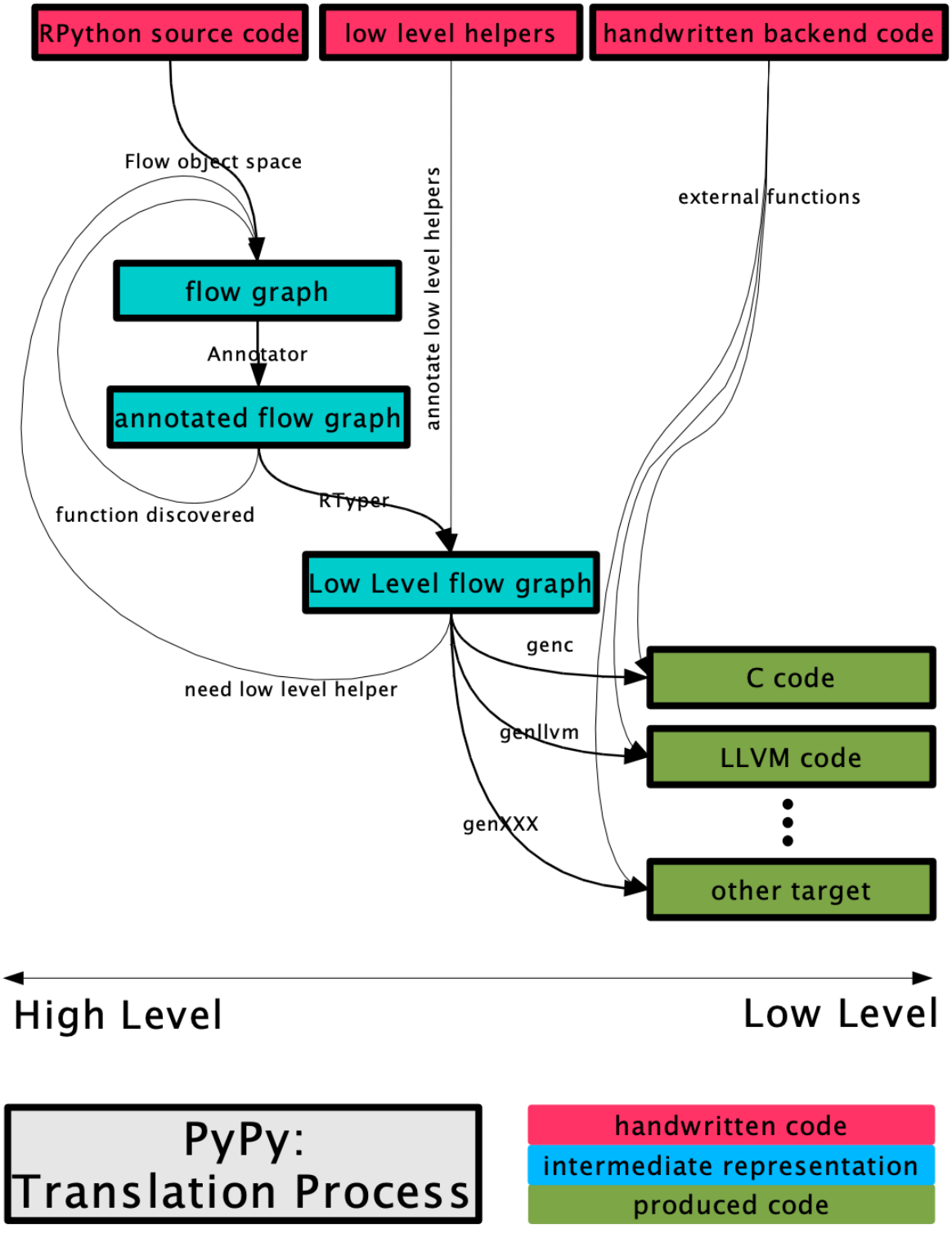
RPython 编译过程

在官方文档中，PyPy 的整个工作不叫“编译”，而是叫“翻译”（translate）。RPython 编译器是一组工具链，将 RPython 程序编译到目标语言，比如 C。这个编译器本身是用 Python 写的。流程如下：



编译器读入转换到 RPython 样式的代码。RPython 会对普通的、带有一些动态性能的 Python 代码做一些限制，比如函数不能动态被生成、变量不能够不确定到底存储什么类型等，一些在解释执行时候没有问题的细节，在这里都会被严格对待。

编译器通过一个称之为抽象解释器（abstract interpretation）的工具构建 RPython程序的流程图。这个抽象解释器使用 PyPy 的 Python 解释器来解析 RPython 程序，这里开始的流程非常复杂，涉及到很多编译相关的技术细节，在 RPython 官方文档这里有一个更加详细的流程图：



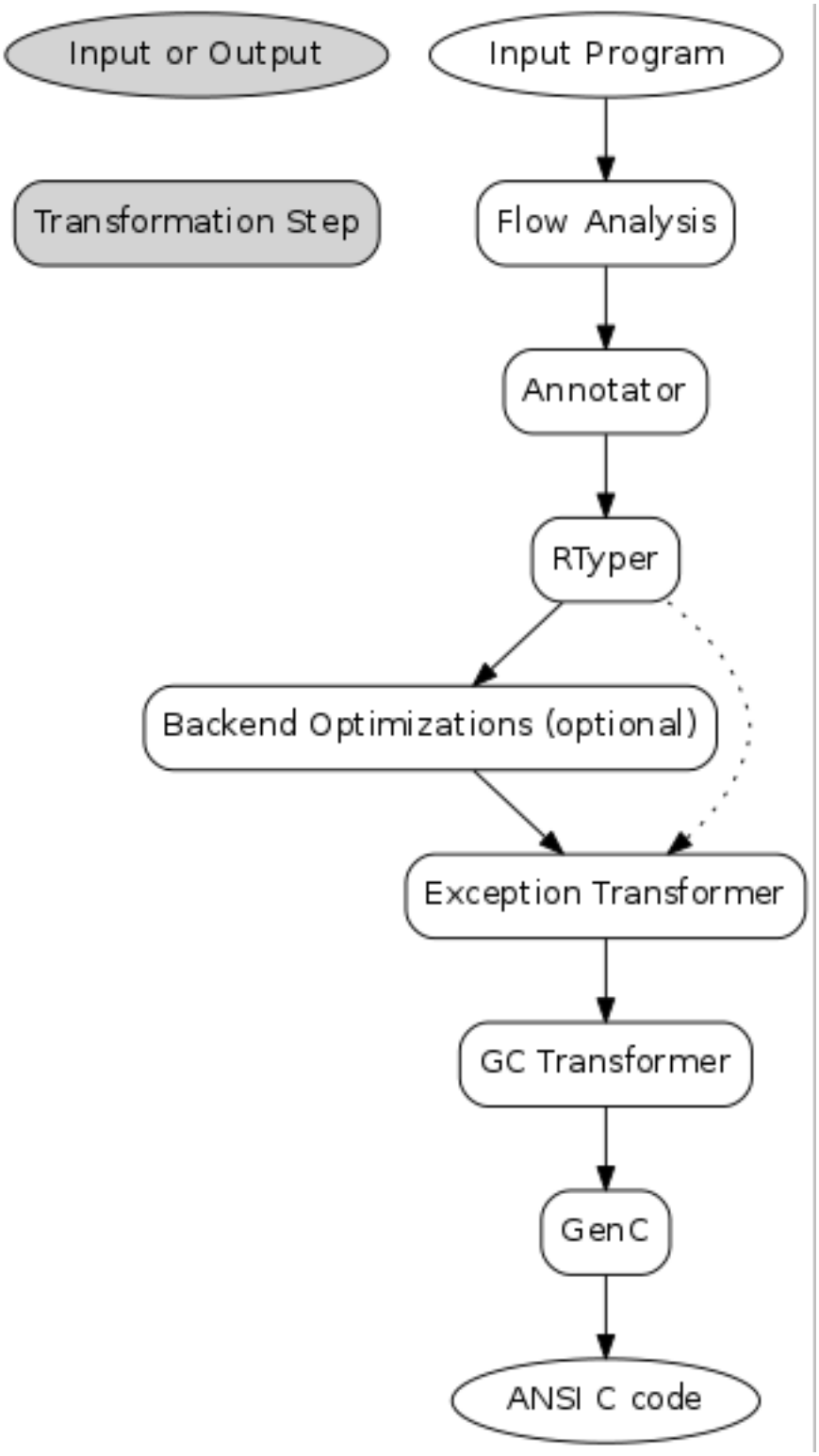
笔者才疏学浅，就不再逐一展开了，有兴趣的朋友可以参考这些资料：

https://rpython.readthedocs.io/en/latest/translation.html

https://www.aosabook.org/en/pypy.html

有更加完整的说明和解释。

编译器一步步通过生成的流程图，最后生成基于 C 语言的代码，当然，在生成 C 代码的时候还要加入很多异常处理、内存管理等。通过下面这一连串的动作，终于可以将 Python 代码转换到 C 代码了。


Input or Output
Input Program
Transformation Step
Flow Analysis
Annotator
RTyper
Backend Optimizations (optional)
Exception Transformer
GC Transformer
GenC
ANSI C code

什么是 *JIT*

什么是即时编译 JIT，Wiki 的定义(https://zh.wikipedia.org/wiki/即时编译) 很清晰：即时编译是动态编译的一种形式，是一种提高程序运行效率的方法。通常，程序有两种运行方式：静态编译与动态解释。静态编译的程序在执行前全部被翻译为机器码，而动态解释执行则是一句一句边运行边翻译。

即时编译器则混合了这二者，一句一句编译源代码，但是会将编译过的代码缓存起来以提高性能。相对于静态编译代码，即时编译的代码可以处理延迟绑定并增强安全性。

即时编译器有两种类型，一是字节码翻译，二是动态编译翻译。微软的.NET Framework，还有绝大多数的 Java 实现，都依赖即时编译以提供高速的代码执行。Ruby 的第三方实现 Rubinius 和 Python 的第三方实现 PyPy 也都通过 JIT 来明显改善了解释器的性能。

PyPy 的 JIT 机制比较与众不同的地方在于，别的编译器直接执行程序代码，其编译器中支持 JIT 特性，而 PyPy 则是在将 Python 代码转换为 RPython 代码的时候加入的，然后将其编译成可执行代码后就自带 JIT 了。这样的方式也算是比较巧妙的设计了，我理解，PyPy 对性能提升很大程度就在这里了。只是，通过这么复杂的两次转换，Python 代码的性能虽然有了几倍的提升，但和 Java、C 等还是有不小的差距。而随着硬件服务器资源的摩尔定律，引入 PyPy 本身增加的项目复杂度以及其对于第三方扩展包的支持问题，似乎还是使用 Python 标准的 CPython 解释器比较好，综合使用成本低很多。PyPy 解决问题的思路非常不错，但还是有很长的路要走。