可爱的 Python: 用 Psyco 让 Python 运行得像 C 一样快

使用 Psyco：Python 专用编译器

David Mertz (mertz@gnosis.cx), 博士,报章作家, Gnosis Software, Inc.

简介： Python 的设计在很多方面都类似于 Java 的设计。两者都利用了解释专门的伪编译字节码的虚拟机。JVM 比 Python 更高级的一个方面在于优化了字节码的执行。Psyco，一种 Python 专用编译器，帮助平衡了这一竞争。Psyco 现在是个外部模块，但是在将来的某一天它可能会包括到 Python 本身中去。只需极少量的额外编程，通常就可以使用 Psyco 将 Python 代码的速度提高好几个数量级。在本文中，David Mertz 研究了 Psyco 是什么，并在一些应用程序中对它进行了测试。

发布日期： 2002 年 5 月 09 日
级别： 初级

Python 对于您想让它做的事来说通常够快了。编程新手对于类似 Python 这样的解释型／字节编译型语言，将 90% 的关注点集中在执行速度方面，是相当幼稚的。在最新的硬件上，大多数非优化的 Python 程序运行的速度和所需要达到的速度一样快，而且，花费额外的编程工作以使应用程序运行得更快实在没什么意义。

因此，在本文，我只对其它的百分之十感兴趣。有时，Python 程序（或用其它语言编写的程序）也会运行得极其缓慢。不同的目的所需要的改进差异很大；提高只运行几毫秒的任务的性能极少能引人注目，但是加快那些需运行几分钟、几小时、几天甚至几星期的任务的速度通常是很值得的。而且，应该注意到并不是所有任务运行缓慢的原因都是由 CPU 引起的。例如，如果完成一个数据库查询要花费几个小时，那么处理结果数据集要花费一分钟还是两分钟就没什么差别了。本文同样不讨论与 I/O 有关的问题。

有很多方法可以加速 Python 程序。每个程序员都应当想到的第一种技术就是改进所使用的算法和数据结构。对低效算法步骤进行细微的优化是徒劳无益的事情。例如，如果当前技术的复杂性等级是 O(n**2)，那么将这些步骤加速 10 倍远不及寻找 O(n) 替代品来得有用。即使在考虑用汇编语言重写算法这种极端情况时，这种思想也都适用：Python 中正确的算法通常会比手工调优的汇编语言中的错误算法快得多。

第二种您应当首先考虑的技术是概要分析您的 Python 应用程序，要着眼于将关键部分重写成 C 扩展模块。使用像 SWIG 这样的扩展封装器（请参阅 参考资料），可以创建 C 扩展，它将程序中最耗时元素作为 C 代码执行。以这种方式扩展 Python 相对简单，但要花些时间学习（并且需要了解 C 的知识）。您经常会发现执行 Python 应用程序所花费的时间绝大部分只是花在了几个函数上，因此，采用这种扩展可能会有很可观的“成果”。

第三种技术建立在第二种技术的基础之上。Greg Ewing 已经创建了名叫 Pyrex 的语言，该语言融合了 Python 和 C。特别地，要使用 Pyrex，需要用类似 Python 的语言编写函数，这种语言将类型声明添加到所选变量中。Pyrex（工具）将“.pyx”文件处理成“.c”扩展名的文件。一旦用 C 编译器进行了编译，就可以将这些 Pyrex（语言）模块导入常规的 Python 应用程序并使用。由于 Pyrex 使用的语法和 Python 本身的语法（包括循环、分支和异常语句、赋值方式、块缩进等等）几乎一样，因此 Pyrex 程序员不需要学会用 C 去编写扩展。而且，与直接用 C 编写扩展相比，Pyrex 允许在同一代码中更无缝地混合 C 级别的变量和 Python 级别的变量（对象）。

最后一种技术就是本文的主题。扩展模块 Psyco 可以插入 Python 解释器的内部，而且可以有选择性地用优化的机器代码去替换部分 Python 的解释型字节码。和所描述的其它技术不同，Psyco 是严格地在 Python 运行时进行操作的。也就是说，Python 源代码是通过 `python` 命令编译成字节码的，所用的方式和以前完全相同（除了为调用 Psyco 而添加的几个 `import` 语句和函数调用）。但是当 Python 解释器运行应用程序时，Psyco 会不时地检查，看是否能用一些专门的机器代码去替换常规的 Python 字节码操作。这种专门的编译和 Java 即时编译器所进行的操作非常类似（一般地说，至少是这样），并且是特定于体系结构的。到现在为止，Psyco 只可用于 i386 CPU 体系结构。Psyco 的妙处在于可以使用您一直在编写的 Python 代码（完全一样！），却可以让它运行得更快。

Psyco 是如何工作的

要完全理解 Psyco，您可能需要很好地掌握 Python 解释器的 `eval_frame()` 函数和 i386 汇编语言。遗憾的是，我自己不能对其中任何一项发表专家性的意见 – 但是我想我可以大致不差地概述 Psyco。

在常规的 Python 中， `eval_frame()` 函数是 Python 解释器的内循环。 `eval_frame()` 函数主要察看执行上下文中的当前字节码，并将控制向外切换到一个适合实现该字节码的函数。支持函数将做什么的具体细节通常取决于保存在内存中的各种 Python 对象的状态。简单点说，

添加 Python 对象“2”和“3”和添加对象“5”和“6”会产生不同的结果，但是这两个操作都以类似的方式分派。

Psyco 用复合求值单元替代 `eval_frame()` 函数。Psyco 有几种方法可以用来改进 Python 所进行的操作。首先，Psyco 将操作编译成有点优化的机器码；由于机器码需要完成的工作和 Python 的分派函数所要做的事一样，所以其本身只有些许改进。而且，Psyco 编译中的“专门的”内容不仅仅是对 Python 字节码的选择，Psyco 也要对执行上下文中已知的变量值进行专门化。例如，在类似于下面的代码中，变量 `x` 在循环持续时间内是可知的：

```
x = 5
l = []
for i in range(1000):
    l.append(x*i)
```

该段代码的优化版本不需要用“x 变量／对象的内容”乘每个 `i` ，与之相比，简单地用 5 乘以每个 `i` 所用的开销较少，省略了查找／间接引用这一步。

除为小型操作创建特定于 i386 的代码之外，Psyco 还高速缓存这个已编译的机器码以备今后重用。如果 Psyco 能够识别出特定的操作和早先所执行的（“专门化的”）操作一样，那么，它就能依靠这个高速缓存的代码而不需要再次编译代码段。这样就节省了一些时间。

但是，Psyco 中真正省时的原因在于 Psyco 将操作分成三个不同的级别。对于 Psyco，有“运行时”、“编译时”和“虚拟时”变量。Psyco 根据需要提高和降低变量的级别。运行时变量只是常规 Python 解释器处理的原始字节码和对象结构。一旦 Psyco 将操作编译成机器码，那么编译时变量就会在机器寄存器和可直接访问的内存位置中表示。

最有意思的级别是虚拟时变量。在内部，一个 Python 变量就是一个有许多成员组成的完整结构 – 即使当对象只代表一个整数时也是如此。Psyco 虚拟时变量代表了需要时可能会被构建的 Python 对象，但是这些对象的详细信息在它们成为 Python 对象之前是被忽略的。例如，考虑如下赋值：

```
x = 15 * (14 + (13 - (12 / 11)))
```

标准的 Python 会构建和破坏许多对象以计算这个值。构建一个完整的整数对象以保存（12/11）这个值；然后从临时对象的结构中“拉”出一个值并用它计算新的临时对象（13-PyInt）。而 Psyco 跳过这些对象，只计算这些值，因为它知道“如果需要”，可以从值创建一个对象。

使用 Psyco

解释 Psyco 相对比较困难，但是 *使用*Psyco 就非常容易了。基本上，其全部内容就是告诉 Psyco 模块哪个函数／方法要“专门化”。任何 Python 函数和类本身的代码都不需进行更改。

有几种方法可以指定 Psyco 应该做什么。“猎枪（shotgun）”方法使得随处都可使用 Psyco 即时操作。要做到这点，把下列行置于模块顶端：

```
import psyco ; psyco.jit()
from psyco.classes import *
```

第一行告诉 Psyco 对所有全局函数“发挥其魔力”。第二行（在 Python 2.2 及以上版本中）告诉 Psyco 对类方法执行相同的操作。为了更精确地确定 Psyco 的行为，可以使用下列命令：

```
psyco.bind(somefunc)          # or method, class
newname = psyco.proxy(func)
```

第二种形式把 `func` 作为标准的 Python 函数，但是优化了涉及 `newname` 的调用。除了测试和调试之外的几乎所有的情况下，您都将使用 `psyco.bind()` 形式。

Psyco 的性能

尽管 Psyco 如此神奇，使用它仍然需要一点思考和测试。主要是要明白 Psyco 对于处理多次循环的块是很有用的，而且它知道如何优化涉及整数和浮点数的操作。对于非循环函数和其它类型对象的操作，Psyco 多半只会增加其分析和内部编译的开销。而且，对于含有大量函数和类的应用程序来说，在整个应用程序范围启用 Psyco，会在机器码编译和用于这一高速缓存的内存使用方面增加大量的负担。有选择性地

绑定那些可以从 Psyco 的优化中获得最大收益的函数，这样会好得多。

我以十分幼稚的方式开始了我的测试过程。我仅仅考虑了我近来运行的、但还未考虑加速的应用程序。想到的第一个示例是用来将我即将出版的书稿（ *Text Processing in Python*）转换成 LaTeX 格式的文本操作程序。该应用程序使用了一些字符串方法、一些正则表达式和一些主要由正则表达式和字符串匹配所驱动的程序逻辑。实际上将它用作 Psyco 的测试候选是很糟的选择，但是我还是使用了，就这么开始了。

第一遍测试中，我所做的就是将 `psyco.jit()` 添加到脚本顶端。这做起来一点都不费力。遗憾的是，结果（意料当中）很令人失望。原先脚本运行要花费 8.5 秒，经过 Psyco 的"加速"后它大概要运行 12 秒。真差劲！我猜测大概是即时编译所需的启动开销拖累了运行时间。因此接下来我试着处理一个更大的输入文件（由原来那个输入文件的多个副本组成）。这次获得了小小的成功，将运行时间从 120 秒左右减到了 110 秒。几次运行中的加速效果比较一致，但是效果都不显著。

文本处理候选项的第二遍测试中。我只添加了 `psyco.bind(main)` 这一行，而不是添加一个总的 `psyco.jit()` 调用，因为 `main()` 函数 *确实* 要循环多次（但是仅利用了最少的整数运算）。这里的结果名义上要比前面好。这种方法将正常的运行时间削减了十分之几秒，在较大的输入版本的情况下削减了数秒钟。但是仍然没有引入瞩目的结果发生（但也没产生什么害处）。

为进行更恰当的 Psyco 测试，我搜寻出我在以前的文章里编写的一些神经网络代码（请参阅"参考资料"）。这个"代码识别器（code_recognizer）"应用程序可以经"训练"用于识别不同编程语言编写的不同 ASCII 值的可能分布情况。类似于这样的东西可能在猜测文件类型方面（比方说丢失的网络信息包）将很有用；但是，关于"训练"些什么，代码实际上完全是通用的 – 它能很容易地学会识别面孔、声音或潮汐模式。任何情况下，"代码识别器"都基于 Python 库 `bpnn` ，Psyco 4.0 分发版也包含（以修正的形式）了该库作为测试用例。在本文中，对"代码识别器"要重点了解它做了许多浮点运算循环并花费了很长的运行时间。这里我们已经有了一个能用于 Psyco 测试的好的候选用例。

使用了一段时间后，我建立了有关 Psyco 用法的一些详细信息。对于这种只有少量类和函数的应用程序，使用即时绑定还是目标绑定没有太大区别。但最佳的结果是，通过有选择性地绑定最优化类，仍可得到几个百分点的改进。然而，更值得注意的是要理解 Psyco 绑定的作用域，这一点很重要。

`code_recognizer.py` 脚本包括类似于下面的这些行：

```
从 bpnn 导入 NN


    class NN2(NN):
    # customized output methods, math core inherited
```

也就是说，从 Psyco 的观点来看，有趣的事情在类 `bpnn.NN` 之中。把 `psyco.jit()` 或 `psyco.bind(NN2)` 添加到 `code_recognizer.py` 脚本中起不了什么作用。要使 Psyco 进行期望的优化，需要将 `psyco.bind(NN)` 添加到 `code_recognizer.py` 或者将 `psyco.jit()` 添加到 `bpnn.py` 。与您可能假设的情况相反，即时优化不在创建实例时或方法运行时发生，而是在定义类的作用域内发生。另外，绑定派生类不会专门化其从其它地方继承的方法。

一旦找到适当的 Psyco 绑定的细微的详细信息，那么加速效果是相当明显的。使用参考文章中提供的相同测试用例和训练方法（500 个训练模式，1000 个训练迭代），神经网络训练时间从 2000 秒左右减到了 600 秒左右 – 提速了 3 倍多。将迭代次数降到 10，加速的倍数也成比例降低（但对神经网络的识别能力无效），迭代的中间数值也会如此变化。

我发现使用两行新代码就能将运行时间从超过半小时减到 10 分钟左右，效果非常显著。这种加速仍可能比 C 编写的类似应用程序的速度慢，而且它肯定比几个独立的 Psyco 测试用例所反映出的 100 倍加速要慢。但是这种应用程序是相当"真实的"，而且在许多环境中这些改进已经是够显著的了。

Psyco 将何去何从？

Psyco 现在不执行任何类型的内部统计或概要分析，只对生成的机器码进行最小优化。可能今后的版本会知道如何针对那些能真正最大获益 Python，并且丢弃为不可优化部分高速缓存的机器码。另外，也许今后的 Psyco 可能会决定对费力运行的操作进行更广泛（但也更昂贵）的优化。这种运行时分析可能类似于 Sun 的 HotSpot 技术为 Java 所做的工作。Java 不像 Python，它有类型声明，但这一事实实际上没有许多人所设想的那样重要（但是先前在 Self、SmallTalk、Lisp 和 Scheme 的优化方面所做的工作也说明了这一点）。

若将 Psyco 类型的技术集成到 Python 本身的某个未来版本中去，会多么令人激动，尽管我怀疑这永远不会真正发生。添加几行导入和绑定代码不需要做很多工作，但却可以轻易地让 Python 比以前运行得快得多。我们将看到这一点。

参考资料

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文.
- 在 SourceForge 的 Psyco 主页和 项目页面上查找更多信息。

- 简化的封装器和接口生成器（Simplified Wrapper and Interface Generator(SWIG)）是广泛（或许是占统治地位的）使用的为 Python 和其它"脚本编制"语言编写 C/C++ 模块的工具。

- Greg Ewing 已经创建了 Pyrex 语言，用它来编写 Python 扩展模块。Pyrex 的主旨是定义一种看起来很接近 Python 本身的语言，它允许 Python 和 C 数据类型的混合，但最终要将其转换并编译成 Python C 扩展。

- John Max Skaller 的 Vyper 语言打算作为一种增强的 Python，它是用 OCaml 实现的。这个项目中希望得到的结果是编译出与 OCaml 生成的机器码相同的机器码，该机器码通常能和 C 的速度相媲美。遗憾的是，Vyper 是个夭折的项目，从未完成编译版本。请重读该项目在开发中时 David 对 Skaller 的采访（ *developerWorks*，2000 年 10 月）。

- David 和 Andrew Blais 合著了 An Introduction to neural networks（ *developerWorks*，2001 年 7 月）。在那篇文章里，他们提供了一些基于 Neil Schemenauer 的 Python 模块 bpnn 的代码。现在这篇文章利用了其中的神经网络代码来演示 Psyco 的功能。

- 在 *developerWorks*Linux 专区查找 更多 Linux 文章。

关于作者

David Mertz 作为猎人、渔夫和牧羊人的失败导致了他的评论性批评生涯。明天他也许会尝试点别的。可以通过 mertz@gnosis.cx和 David 联系；在 http://gnosis.cx/publish/上了解他的生活。欢迎提出关于本栏过去、现在和将来的意见和建议。

当您初次登录到 developerWorks 时，将会为您创建一份概要信息，您需要指定一个昵称。您的昵称将和您在 developerWorks 发布的内容显示在一起。

昵称长度在 3 至 31 个字符之间。 您的昵称在 developerWorks 社区中必须是唯一的，并且出于隐私保护的原因，不能是您的电子邮件地址。

昵称： （长度在 3 至 31 个字符之间）

单击提交则表示您同意developerWorks 的条款和条件。 使用条款.

提交 取消

所有提交的信息确保安全。

平均分 (8个评分)

1 星

1 星
2 星

2 星
3 星

3 星
4 星

4 星
5 星

5 星

提交

添加评论:

请 登录 或 注册 后发表评论。

注意：评论中不支持 HTML 语法

有新评论时提醒我剩余 1000 字符

发布

快来添加第一条评论

打印此页面 分享此页面 关注 **developerWorks**