**超过十年以上，没有比解释器全局锁（GIL）让Python新手和专家更有挫折感或者更有好奇心。**

**Python的底层**

    要理解**GIL**的含义，我们需要从**Python的基础讲起**。像C++这样的语言是**编译型语言**，所谓编译型语言，是指程序**输入到编译器**，编译器再根据语言的语法**进行解析**，然后翻译成语言独立的中间表示，最终链接成具有**高度优化的机器码的可执行程序**。编译器之所以可以深层次的对代码进行优化，是因为它可以看到整个程序（或者一大块独立的部分）。这使得它可以对不同的语言指令之间的**交互进行推理**，从而给出更有效的优化手段。

    与此相反，**Python是解释型语言**。程序被输入到**解释器**来运行。解释器在程序**执行之前对其并不了解**；它所知道的只是**Python的规则**，以及在执行过程中怎样去动态的**应用这些规则**。它也有一些优化，但是这基本上只是另一个级别的优化。由于**解释器没法很好的对程序进行推导**，Python的大部分优化其实**是解释器自身的优化**。更快的解释器自然意味着程序的运行也能**“免费”**的更快。也就是说，解释器优化后，Python程序不用做修改就可以享受优化后的好处。

    这一点很重要，让我们再强调一下。如果其他条件不变**，Python程序的执行速度直接与解释器的“速度”相关**。不管你怎样优化自己的程序，你的程序的执行速度还是**依赖于解释器执行你的程序的效率**。这就很明显的解释了为什么我们需要对**优化Python解释器**做这么多的工作了。对于Python程序员来说，这恐怕是与免费午餐最接近的了。

**免费午餐结束了**

    还是没有结束？摩尔定律给出了硬件速度会按照确定的时间周期增长，与此同时，整整一代程序员学会了如何编码。如果一个人写了比较慢的代码，最简单的结果通常是更快的处理器去等待代码的执行。显然，摩尔定律仍然是正确的，并且还会在很长一段时间生效，不过它提及的方式有了根本的变化。并非是时钟频率增长到一个高不可攀的速度，而是通过多核来利用晶体管密度提高带来的好处。在新处理器上运行的程序要想充分利用其性能，必须按照并发方式进行重写。

    大部分开发者听到**“并发”**通常会立刻想到**多线程的程序**。目前来说，**多线程执行还是利用多核系统最常用的方式**。尽管多线程编程大大好于“顺序”编程，不过即便是仔细的程序员也没法在**代码中将并发性做到最好**。编程语言在这方面应该做的更好，大部分应用广泛的**现代编程语言都会支持多线程编程**。

**意外的事实**

    现在我们来看一下问题的症结所在。要想利用多核系统，**Python必须支持多线程运行**。作为**解释型语言**，**Python的解释器必须做到既安全又高效**。我们都知道**多线程编程会遇到的问题**。解释器要留意的**是避免在不同的线程操作内部共享的数据**。同时它还要保证在管理用户线程时保证总是有最大化的计算资源。

    那么，不同**线程同时访问时**，数据的保护机制是怎样的呢？答案是**解释器全局锁**。从名字上看能告诉我们很多东西，很显然，这是一个**加在解释器上的全局（从解释器的角度看）锁（**从互斥或者类似角度看）。这种方式当然很安全，但是它有**一层隐含的意思（Python初学者需要了解这个）**：对于**任何Python程序，不管有多少的处理器，任何时候都总是只有一个线程在执行**。

    许多人都是**偶然发现这个事实的**。网上的很多**讨论组和留言板都充斥着来自Python初学者和专家的类似这样的问题**——”为什么我全新的多线程**Python程序运行得比其只有一个线程的时候还要慢**？“许多人在问这个问题时还是非常犯晕的，因为显然一个**具有两个线程的程序**要比其只有一个线程时要快（**假设该程序确实是可并行的**）。事实上，这个问题被问得如此频繁以至于Python的专家们精心制作了一个标准答案：”**不要使用多线程，请使用多进程**。“但这个答案比那个问题更加让人困惑。难道我不能在Python中使用多线程？在Python这样流行的一个语言中使用多线程究竟是有**多糟糕，连专家都建议不要使用**。难道我真的漏掉了一些东西？

    很遗憾，没有**任何东西被漏掉**。由于**Python解释器的设计**，使用多线程以提高性能应该算是一个困难的任务。在最坏的情况下，它将会降低（有时很明显）你的**程序的运行速度**。一个**计算机科学与技术专业的大学生新手**可能会告诉你当多个线程都在竞争一个共享资源时将会发生什么。结果通常不会非常理想。很多情况下多线程都能很好地工作，可能对于解释器的实现和内核开发人员来说，没有关于**Python多线程性能的过多抱怨**。

**现在该怎么办？惊慌？**

    那么，这又能怎样？问题解决了吗？**难道我们作为Python开发人员就意味着要放弃使用多线程来探索并行的想法了**？为什么无论怎样，GIL需要保证只有一个线程在某一时刻处于运行中？难道不可以**添加细粒度的锁来阻止多个独立对象的同时访问**？并且为什么之前没有人去尝试过类似的事情？

    这些实用的问题有着十分有趣的回答。**GIL**对诸如当前线程状态和为垃圾回收而用的堆分配对象这样的东西的访问提供着保护。然而，这对Python语言来说没什么特殊的，它需要使用一个GIL。这是该实现的一种典型产物。现在也有其它的Python解释器（和编译器）并不使用GIL。虽然，对于CPython来说，自其出现以来已经有**很多不使用GIL的解释器**。

    那么**为什么不抛弃GIL呢**？许多人也许不知道，在1999年，针对Python 1.5，一个经常被提到但却不怎么理解的“free threading”补丁已经尝试实现了这个想法，该补丁来自**Greg Stein**。在这个补丁中，GIL被完全的移除，且用细粒度的锁来代替。然而，GIL的移除给单线程程序的执行速度带来了一定的代价。当用单线程执行时，**速度大约降低了40%**。使用两个线程展示出了在速度上的提高，但除了这个提高，这个收益并没有随着核数的增加而线性增长。由于执行速度的降低，这一补丁被拒绝了，并且几乎被人遗忘。

**移除GIL非常困难，让我们去购物吧！**

    译者注：XXX is hard. Let's go shopping!在英语中类似于中文的咆哮体。其隐含意思为想成功完成某件事情非常困难，我们去直接寻找第三方的产品替代吧。）

    不过，“**free threading**”这个补丁是有**启发性意义**的，其证明了一个关于**Python解释器的基本要点**：移除GIL是非常困难的。由于该补丁发布时所处的年代，解释器变得依赖更多的全局状态，这使得想要移除当今的GIL变得更加困难。值得一提的是，也正是因为这个原因，许多人对于尝试移除GIL变得更加有兴趣。困难的问题往往很有趣。

    但是这可能有点被误导了。让我们考虑一下：如果我们有了一个神奇的补丁，**其移除了GIL**，并且没有对单线程的Python代码产生性能上的下降，那么什么事情将会发生？我们将会获得我们一直想要的：一**个线程API可能会同时利用所有的处理器**。那么现在，我们已经获得了我们希望的，但这确实是一个好事吗？

    基于线程的编程毫无疑问是困难的。每当某个人觉得他了解关于线程是如何工作的一切的时候，总是会悄无声息的出现一些新的问题。因为在这方面想要**得到正确合理的一致性真的是太难了**，因此有一些非常知名的语言设计者和研究者已经总结得出了一些线程模型。就像某个写过多线程应用的人可以告诉你的一样，不管是**多线程应用的开发还是调试都会比单线程的应用难上数倍**。程序员通常所具有的顺序执行的思维模恰恰就是与并行执行模式不相匹配。GIL的出现无意中帮助了**开发者免于陷入困境**。在使用多线程时仍然需要同步原语的情况下，GIL事实上帮助我们保持**不同线程之间**的**数据一致性问题**。

    那么现在看起来**讨论Python**最难得问题是有点问错了问题。我们有非常好的理由来说明为什么**Python专家**推荐我们使用**多进程代替多线程**，而不是去**试图隐藏Python线程实现的不足**。更进一步，我们**鼓励开发者使用更安全更直接的方式实现并发模型**，同时保留使用多线程进行开发除非你觉的真的非常必要的话。对于大多数人来说什么是最好的**并行编程模型可能并不是十分清楚**。但是目前我们清楚的是**多线程的方式可能并不是最好的**。

    至于GIL，不要认为它在那的存在就是**静态的和未经分析过的**。Antoine Pitrou 在Python**3.2**中实现了一个新的GIL，并且带着一些积极的结果。这是自1992年以来，**GIL的一次最主要改变**。这个改变非常巨大，很难在这里解释清楚，但是从一个更高层次的角度来说，旧的GIL通过对Python指令**进行计数来确定何时放弃GIL**。这样做的结果就是，单条Python指令将会包含大量的工作，即它们并没有被1:1的翻译成机器指令。在新的GIL实现中，用一个**固定的超时时间来指示当前的线程以放弃这个锁**。在当前线程保持这个锁，且当第二个线程请求这个锁的时候，当前线程就会在5ms后被强制释放掉这个锁（这就是说，**当前线程每5ms就要检查其是否需要释放这个锁**）。当任务是可行的时候，这会使得线程间的切换更加可预测。

    然而，这并不是一个完美的改变。对于在各种类型的任务上**有效利用GIL这个领域里**，最活跃的研究者可能就是David Beazley了。除了对Python 3.2之前的GIL研究最深入，他还研究了这个最新的GIL实现，并且发现了很多有趣的程序方案。对于这些程序，即使是新的GIL实现，其表现也相当糟糕。他目前仍然通过一些实际的研究和发布一些实验结果来引领并推进着有关GIL的讨论。

    不管某一个人**对Python的GIL感觉如何**，它仍然是**Python语言里最困难的技术挑战**。想要理解它的实现需要对**操作系统设计、多线程编程、C语言、解释器设计和CPython解释器**的实现有着非常彻底的理解。单是这些所需准备的就妨碍了很多开发者去更彻底的研究GIL。虽然如此，并没有迹象表明GIL在不久以后的任何一段时间内会远离我们。目前，它将继续给那些**新接触Python**，并且与此同时又对解决非常困难的技术问题感兴趣的人带来困惑和惊喜。

    以上内容是基于我目前对Python解释器所做出的研究而写。虽然我还希望写一些有关解释器的其它方面内容，但是没有任何一个比全局解释器锁（GIL）更为人所知。虽然我认为这里有些内容是不准确的，但是这些技术上的细节与CPython的很多资源条目是不同的。如果你发现了不准确的内容，请及时告知我，这样我就会尽快对其进行改正。（本文来自：http://www.oschina.net/translate/pythons-hardest-problem）