Python 对自由线程的支持

从 3.13 发布版开始，CPython 支持 free threading 的 Python 构建，其禁用 global interpreter lock (GIL)。 自由线程化的执行允许在可用的 CPU 核心上并行运行线程，充分利用可用的处理能力。 尽管并非所有软件都能自动地从中受益，但是考虑到线程设计的程序在多核硬件上运行速度会更快。

自由线程模式目前可用并在不断改进，但与常规构建相比，单线程工作负载会产生一些额外的开销。此外，第三方软件包，特别是带有 extension module 的软件包，可能无法在自由线程构建中使用，并将重新启用 GIL。

本文档描述了自由线程对 Python 代码的影响。 请参阅 自由线程的 C API 扩展支持 了解如何编写支持自由线程构建的 C 扩展。

参见： **PEP 703** —— 查阅《在 CPython 中使全局解释器锁成为可选项》以了解对自由线程 Python 的整体描述。

安装

从 Python 3.13 开始，官方 macOS 和 Windows 安装器提供了对可选安装自由线程 Python 二进制文件的支持。 安装器可在 https://www.python.org/downloads/ 获取。

有关其他平台的信息，请参阅 Installing a Free-Threaded Python，这是一份由社区维护的针对安装自由线程版 Python 的安装指南。

当从源码构建 CPython 时，应使用 `--disable-gil` 配置选项以构建自由线程 Python 解释器

识别自由线程 Python

要判断当前解释器是否支持自由线程，可检查 `python -VV` 和 `sys.version` 是否包含 "free-threading build"。 新的 `sys._is_gil_enabled()` 函数可用于检查在运行进程中 GIL 是否确实被关闭。

`sysconfig.get_config_var("Py_GIL_DISABLED")` 配置变量可用于确定构建是否支持自由线程 。如果该变量设置为 `1`，则构建支持自由线程。 这是与构建配置相关的决策的推荐机制。

自由线程版 Python 中的全局解释器锁

CPython 的自由线程构建版支持在运行时使用环境变量 `PYTHON_GIL` 或命令行选项 `-X gil` 选择性地启用 GIL。

GIL 也可能在导入未显式标记为支持自由线程模式的 C-API 扩展模块时被自动启用。 在这种情况下将会打印一条警告。

在单独软件包的文档以外，还有下列网站在追踪热门软件包对自由线程模式的支持状态：

- https://py-free-threading.github.io/tracking/
- https://hugovk.github.io/free-threaded-wheels/

线程安全

自由线程构建的 CPython 旨在 Python 层级提供与默认全局解释器锁启用构建相似的线程安全行为。内置类型（如 `dict` 、 `list` 和 `set` 等）使用内部上锁来防止并发修改，其行为方式与全局解释器锁相似。 但是，Python 历来不对这些内置类型的并发修改提供特定的行为提供保证，因此这应被视为对当前实现的描述，而不是对当前或未来行为的保证。

备注： 建议尽可能使用 `threading.Lock` 或其他同步的原语，而不是依赖内置类型的内部上锁 。

已知的限制

本节介绍自由线程 CPython 构建的已知限制。

永生化

3.13 版本的自由线程构建使某些对象 immortal。 永生对象不会被重新分配，其引用计数永远不会被修改。 这样做是为了避免引用计数发生争夺，以免妨碍高效的多线程扩展。

当主线程运行后首次启动新的线程时，对象将被永生化。 以下对象将被永生化：

- 在模块中声明的 函数 对象
- 方法 描述器
- 代码对象
- module 对象及其字典
- 类 (类型对象)

由于永生对象永远不会被重新分配，因此应用如果创建了许多此类对象，可能会增加内存使用。预计 3.14 版将解决这个问题。

此外，代码中的数字和字符串字面值以及 `sys.intern()` 返回的字符串也将永久化。 预计在 3.14 自由线程构建中将保留这一行为 。

帧对象

从其他线程访问 帧 对象是不安全的，这样做可能会导致程序崩溃。 这意味着，在自由线程构建中使用 `sys._current_frames()` 一般是不安全的。 函数（如 `inspect.currentframe()` 和 `sys._getframe()` 等）只要不将生成的帧对象传递给另一个线程，一般都是安全的。

迭代器

在多个线程之间共享同一个迭代器对象通常是不安全的，线程在迭代时可能会出现元素重复或缺失的情况，或使解释器崩溃。

单线程性能

与启用默认全局解释器锁的构建相比，自由线程构建在执行 Python 代码时有额外的开销。 在 3.13 中，pyperformance 套件的开销约为 40%。大部分时间花在 C 扩展或 I/O 上的程序受到的影响较小。影响最大的原因是在自由线程构建中禁用了特化自适应解释器 (**PEP 659**) 。 我们希望在 3.14 中以线程安全的方式重新启用它。在即将发布的 Python 版本中，这一开销有望减少。 我们的目标是，与启用默认全局解释器锁的构建相比，pyperformance 套件的开销不超过 10%。

行为的变化

本节描述CPython在自由线程构建时的行为变化。

上下文变量

在自由线程构建中，`thread_inherit_context` 标志默认设置为 true，这会导致使用 `threading.Thread` 创建的线程以 `start()` 的调用程序的 `Context()` 的副本启动。在默认启用 GIL 的构建中，该标志默认为 false，因此线程以空 `Context()` 启动。

警告过滤器

在自由线程构建中，`context_aware_warnings` 标志默认设置为 true。 在默认启用 GIL 的构建中，该标志默认设置为 false。 如果该标志为 true，则 `warnings.catch_warnings` 上下文管理器使用上下文变量用于警告过滤器。 如果该标志为 false，则 `catch_warnings` 修改全局过滤器列表，这不是线程安全的。 详情请参阅 `warnings` 模块。