《源码探秘 CPython》85. Python线程的创建、销毁、调度，以及 GIL 的实现原理

原创 古明地觉 古明地觉的编程教室 2022-05-10 08:30 发表于北京

初识 _thread 模块

下面来说一下 Python 线程的创建，我们知道在创建多线程的时候会使用 threading 这个标准库，这个库以一个 py 文件的形式存在，不过这个模块依赖于 _thread 模块，我们来看看它长什么样子。

_thread 是真正用来创建线程的模块，这个模块由 C 编写，内嵌在解释器里面。我们可以 import 导入，但是在 Python 安装目录里面则是看不到的。像这种底层由 C 编写、内嵌在解释器里面的模块，以及那些无法使用文本打开的 pyd 文件，PyCharm 都会给你做一个抽象，并且把注释给你写好。

记得我们之前说过 Python 源码中的 Modules 目录，这个目录里面存放了大量使用 C 编写的模块，它们在编译完 Python 之后就内嵌在解释器里面了。而这些模块都是针对那些性能要求比较高的，而要求不高的则由 Python 语言编写，存放在 Lib 目录下。

像我们平时调用 random、collections、threading，其实它们背后会调用 C 实现的 _random、_collections、_thread。再比如我们使用的 re 模块，真正用来做正则匹配的逻辑实际上位于 Modules/_sre.c 里面。

而 _thread 的底层实现是在 _threadmodule.c中，我们来看看它都提供了哪些接口。

```
static PyMethodDef thread_methods[] = {
    {"start_new_thread",        (PyCFunction)thread_PyThread_start_new_t
     METH_VARARGS, start_new_doc},
    {"start_new",               (PyCFunction)thread_PyThread_start_new_t
     METH_VARARGS, start_new_doc},
    {"allocate_lock",           thread_PyThread_allocate_lock,
     METH_NOARGS, allocate_doc},
    {"allocate",                thread_PyThread_allocate_lock,
     METH_NOARGS, allocate_doc},
    {"exit_thread",             thread_PyThread_exit_thread,
     METH_NOARGS, exit_doc},
    {"exit",                    thread_PyThread_exit_thread,
     METH_NOARGS, exit_doc},
    {"interrupt_main",          thread_PyThread_interrupt_main,
     METH_NOARGS, interrupt_doc},
    {"get_ident",               thread_get_ident,
     METH_NOARGS, get_ident_doc},
#ifdef PY_HAVE_THREAD_NATIVE_ID
    {"get_native_id",           thread_get_native_id,
     METH_NOARGS, get_native_id_doc},
#endif
    {"_count",                  thread__count,
     METH_NOARGS, _count_doc},
    {"stack_size",              (PyCFunction)thread_stack_size,
     METH_VARARGS, stack_size_doc},
    {"_set_sentinel",           thread__set_sentinel,
     METH_NOARGS, _set_sentinel_doc},
    {"_excepthook",              thread_excepthook,
     METH_O, excepthook_doc},
    {NULL,                      NULL}           /* sentinel */
};
```


显然PyCharm 抽象出来的 _thread.py，和底层的这些接口是一样的。而创建一个线程会调用 start_new_thread，在底层会调用 thread_PyThread_start_new_thread，当然这里截图没有截全。

当我们使用 threading 模块创建一个线程的时候，threading 会调用 _thread 模块的 start_new_thread 来创建。而它对应 thread_PyThread_start_new_thread，下面我们就来看看这个函数。

```
1  //Modules/_threadmodule.c
2  static PyObject *
3  thread_PyThread_start_new_thread(PyObject *self, PyObject *fargs)
4  {
5      PyObject *func, *args, *keyw = NULL;
6      struct bootstate *boot;
7      unsigned long ident;
8
9      //下面都是参数检测逻辑
10     //thread.Thread()里面我们一般传递target、args、kwargs
11     if (!PyArg_UnpackTuple(fargs, "start_new_thread", 2, 3,
12                            &func, &args, &keyw))
13         return NULL;
14     //target必须可调用
15     if (!PyCallable_Check(func)) {
16         PyErr_SetString(PyExc_TypeError,
17                         "first arg must be callable");
18         return NULL;
19     }
```

```
    //args是个元组
    if (!PyTuple_Check(args)) {
        PyErr_SetString(PyExc_TypeError,
                        "2nd arg must be a tuple");
        return NULL;
    }
    //kwargs是个字典
    if (keyw != NULL && !PyDict_Check(keyw)) {
        PyErr_SetString(PyExc_TypeError,
                        "optional 3rd arg must be a dictionary");
        return NULL;
    }

    //创建bootstate结构体实例
    /*
    struct bootstate {
        PyInterpreterState *interp;
        PyObject *func;
        PyObject *args;
        PyObject *keyw;
        PyThreadState *tstate;
    };
    */
    boot = PyMem_NEW(struct bootstate, 1);
    if (boot == NULL)
        return PyErr_NoMemory();
    //获取进程状态对象、函数、args、kwargs
    boot->interp = _PyInterpreterState_Get();
    boot->func = func;
    boot->args = args;
    boot->keyw = keyw;
    boot->tstate = _PyThreadState_Prealloc(boot->interp);
    if (boot->tstate == NULL) {
        PyMem_DEL(boot);
        return PyErr_NoMemory();
    }
    Py_INCREF(func);
    Py_INCREF(args);
    Py_XINCREF(keyw);
    //初始化多线程环境，记住这一步
    PyEval_InitThreads();

    //创建线程，返回id
    ident = PyThread_start_new_thread(t_bootstrap, (void*) boot);
    if (ident == PYTHREAD_INVALID_THREAD_ID) {
        PyErr_SetString(ThreadError, "can't start new thread");
        Py_DECREF(func);
        Py_DECREF(args);
        Py_XDECREF(keyw);
        PyThreadState_Clear(boot->tstate);
        PyMem_DEL(boot);
        return NULL;
    }
    return PyLong_FromUnsignedLong(ident);
}
```

因此在这个函数中，我们看到虚拟机通过三个主要的动作完成一个线程的创建。

- **1. 创建并初始化 bootstate 结构体实例对象 boot，在 boot 中，会保存一些相关信息；**
- **2. 初始化 Python 的多线程环境；**
- **3. 以 boot 为参数，创建子线程，子线程也会对应操作系统的原生线程；**

而在源码中，有这么一行：**boot->interp = _PyInterpreterState_Get();**，说明 boost 保存了 Python 的 PyInterpreterState 对象，这个对象中携带了 Python 的模块对象池(module pool)

这样的全局信息，而所有的 thread 都会保存这些全局信息。

然后我们还看到了多线程环境的初始化动作，这一点需要注意，Python 在启动的时候是不支持多线程的。换言之，Python 中支持多线程的数据结构、以及 GIL 都还没有创建。

因为对多线程的支持是需要代价的，如果上来就激活了多线程，但是程序却只有一个主线程，那么Python仍然会执行所谓的线程调度机制，只不过调度完了还是它自己，所以这无异于在做无用功。因此 Python 将开启多线程的权利交给了程序员，自己在启动的时候是单线程，既然是单线程，自然就不存在线程调度了、当然也没有GIL。

而一旦我们调用了**threading.Thread(...).start()**，底层对应 _thread.start_new_thread()，则代表明确地指示虚拟机要创建新的线程。这个时候虚拟机就知道自己该创建与多线程相关的东西了，比如：数据结构、环境、以及那个至关重要的GIL。

建立多线程环境

多线程环境的建立，说的直白一点，主要就是创建 GIL。我们已经知道了 GIL 对于 Python 多线程机制的重要意义，但是这个 GIL 是如何实现的呢？这是一个比较有趣的问题，下面就来看看 GIL 长什么样子。

```
//include/internal/pycore_pystate.h
struct _ceval_runtime_state {
    //递归限制，可以通过sys.getrecursionlimit()查看
    int recursion_limit;
    //记录是否对任意线程启用跟踪
    //同时计算 tstate->c_tracefunc 为空的线程数
    //如果该值为0，那么将不会检查该线程的 c_tracefunc
    //这会加快 PyEval_EvalFrameEx()中 fast_next_opcode之后的if语句
    //关于这个字段，我们就不深入讨论了
    int tracing_possible;

    //eval循环中所有跳出快速通道的请求数，不深入讨论
    _Py_atomic_int eval_breaker;

    //是否被要求放弃 GIL
    _Py_atomic_int gil_drop_request;

    //线程调度相关，比如：加锁
    struct _pending_calls pending;

    //信号检测相关
    _Py_atomic_int signals_pending;

    //重点来了，GIL
    //我们看到 GIL 是一个 struct _gil_runtime_state 结构体实例
    struct _gil_runtime_state gil;
};
```

所以GIL在Python的底层就是一个 _gil_runtime_state 结构体实例，来看看这个结构体长什么样子。

```
//Python/ceval_gil.h
#define DEFAULT_INTERVAL 5000

//include/internal/pycore_gil
struct _gil_runtime_state {
    //一个线程拥有 GIL 的间隔，默认是 5000 微妙
```

```
    //也就是调用 sys.getswitchinterval()得到的 0.005
    unsigned long interval;

    //最后一个持有 GIL 的 PyThreadState
    //这有助于我们知道在丢弃 GIL 后是否还有其他线程被调度
    _Py_atomic_address last_holder;
    //GIL是否被获取，这个是原子性的，
    //因为在ceval.c中不需要任何锁就能够读取它
    _Py_atomic_int locked;

    //从GIL创建之后，总共切换的次数
    unsigned long switch_number;
    //cond允许一个或多个线程等待，直到GIL被释放
    PyCOND_T cond;

    /* mutex则是负责保护上面的变量 */
    PyMUTEX_T mutex;
#ifdef FORCE_SWITCHING
    //"GIL等待线程"在被调度获取 GIL 之前
    //"GIL释放线程"一直处于等待状态
    PyCOND_T switch_cond;
    PyMUTEX_T switch_mutex;
#endif
};
```

所以我们看到 GIL 就是 _gil_runtime_state 结构体实例，而该结构体又内嵌在结构体 _ceval_runtime_state 里面。

GIL 有一个 locked 字段用于判断 GIL 有没有被获取，这个 locked 字段可以看成是一个布尔变量，其访问受到 mutex 字段保护，是否改变则取决于 cond 字段。在持有 GIL 的线程中，主循环(_PyEval_EvalFrameDefault)必须能通过另一个线程来按需释放 GIL。

而在创建多线程的时候，首先是需要调用 PyEval_InitThreads 进行初始化的。我们就来看看这个函数，位于 Python/ceval.c 中。

```
void
PyEval_InitThreads(void)
{
    //获取运行时状态对象
    _PyRuntimeState *runtime = &_PyRuntime;
    //拿到 ceval，它是 struct _ceval_runtime_state类型
    //而 GIL 对应的字段就内嵌在里面
    struct _ceval_runtime_state *ceval = &runtime->ceval;
    //获取 GIL
    struct _gil_runtime_state *gil = &ceval->gil;

    //如果 GIL 已经创建，那么直接返回
    if (gil_created(gil)) {
        return;
    }

    //线程的初始化
    PyThread_init_thread();
    //创建gil
    create_gil(gil);
    //获取线程状态对象
    PyThreadState *tstate = _PyRuntimeState_GetThreadState(runtime);
    //GIL 创建了，那么就要拿到这个 GIL
    take_gil(ceval, tstate);

    //我们说这个是和线程调度相关的
    struct _pending_calls *pending = &ceval->pending;
    //如果拿到 GIL 了，其它线程就不能获取了
```

```
    //那么不好意思这个时候要加锁
    pending->lock = PyThread_allocate_lock();
    if (pending->lock == NULL) {
        Py_FatalError("Can't initialize threads for pending calls");
    }
}
```

关于 GIL 有四个函数，分别是 gil_created, create_gil, take_gil, drop_gil，含义如下：

- gil_created：GIL 是否已被创建；
- create_gil：创建 GIL；
- take_gil：获取创建的 GIL；
- drop_gil：释放持有的 GIL；

```
//Python/ceval_gil.h
static int gil_created(struct _gil_runtime_state *gil)
{
    //检测 GIL 有没有被创建
    return (_Py_atomic_load_explicit(&gil->locked, _Py_memory_order_acqu
ire) >= 0);
}


static void create_gil(struct _gil_runtime_state *gil)
{
    //创建 GIL，下面是负责初始化 GIL 里面的字段
    MUTEX_INIT(gil->mutex);
#ifdef FORCE_SWITCHING
    MUTEX_INIT(gil->switch_mutex);
#endif
    COND_INIT(gil->cond);
#ifdef FORCE_SWITCHING
    COND_INIT(gil->switch_cond);
#endif
    _Py_atomic_store_relaxed(&gil->last_holder, 0);
    _Py_ANNOTATE_RWLOCK_CREATE(&gil->locked);
    _Py_atomic_store_explicit(&gil->locked, 0, _Py_memory_order_release);
}


static void
take_gil(struct _ceval_runtime_state *ceval, PyThreadState *tstate)
{
    if (tstate == NULL) {
        Py_FatalError("take_gil: NULL tstate");
    }

    struct _gil_runtime_state *gil = &ceval->gil;
    int err = errno;
    MUTEX_LOCK(gil->mutex);

    //判断 GIL 是否被释放
    //如果被释放，那么直接跳转到_ready
    if (!_Py_atomic_load_relaxed(&gil->locked)) {
        goto _ready;
    }

    //走到这里说明 GIL 没有被释放，还被某个线程所占有
    //那么会阻塞在这里，一直请求获取 GIL
    //直到 GIL 被释放，while 条件为假，结束循环
    while (_Py_atomic_load_relaxed(&gil->locked)) {
        int timed_out = 0;
        unsigned long saved_switchnum;
```

```
        //代表 GIL 已被占有，会一直循环请求获取 GIL
        //.....
        //.....
    }
_ready:
#ifdef FORCE_SWITCHING
    //.....
    //GIL一次只能被一个线程获取，因此获取到 GIL 的时候，要进行独占
    //于是会通过_Py_atomic_store_relaxed对其再次上锁
    _Py_atomic_store_relaxed(&gil->locked, 1);
    _Py_ANNOTATE_RWLOCK_ACQUIRED(&gil->locked, /*is_write=*/1);

    //.....
}
```

Python线程在获取 GIL 的时候会调用 take_gil 函数，在里面会检查当前 GIL 是否可用。而其中的 locked 字段就是指示当前 GIL 是否可用，如果这个值为 0，则代表可用，那么获取之后就必须要将 GIL 的 locked 字段设置为 1，表示当前 GIL 已被占用。

而当该线程释放 GIL 的时候，也一定要将该值减去 1，这样 GIL 的值才会从 1 变成 0，才能被其他线程使用，所以官方把 GIL 的 locked 字段说成是布尔类型也不是没有道理的。

另外，由于获取到 GIL，就将 locked 字段更新为 1；并且获取 GIL 之前，也会先检测 locked 字段是否为 1。这就说明，GIL 每次只能被一个线程获取，而一旦被某个线程获取，那么其它线程会因 locked 字段为 1，而阻塞在 while 循环处。

持有 GIL 的线程释放 GIL 之后，会通知所有在等待 GIL 的线程。但是会选择哪一个线程呢？之前说了，这个时候 Python 会直接借用操作系统的调度机制随机选择一个。

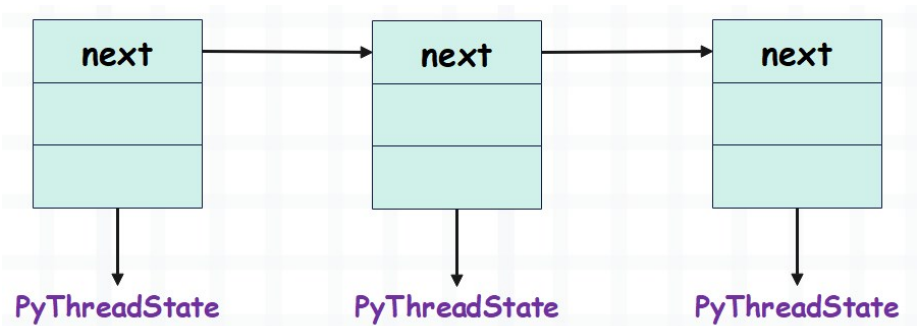
线程状态对象的保护机制

线程状态对象中都保存着当前正在执行的栈帧对象、线程id 等信息，因为这些信息是需要被线程访问的。但是要考虑到安全问题，比如线程 A 访问线程对象，但是里面存储的却是线程 B 的 id，这样的话就完蛋了。

因此 Python 内部必须有一套机制，这套机制与操作系统管理进程的机制非常类似。在线程切换的时候，会保存当前线程的上下文，并且还能够进行恢复。而在 Python 内部，会维护这一个变量（上一篇文章提到过），负责保存当前活动线程所对应的线程状态对象。当Python调度线程时，会将新的被激活线程所对应的线程状态对象赋给这个变量，总之它始终保存活动线程的状态对象。

但是这样就引入了一个问题：Python 在调度线程时，如何获得被激活线程对应的状态对象呢？其实 Python 内部会通过一个单向链表来管理所有的 Python 线程状态对象，当需要寻找一个线程对应的状态对象时，就会遍历这个链表。

而对这个状态对象链表的访问，则不必在 GIL 的保护下进行。因为对于这个状态对象链表，Python 会专门创建一个独立的锁，专职对这个链表进行保护，而且这个锁的创建是在 Python 初始化的时候就完成的。



从 GIL 到字节码

我们知道创建线程状态对象是通过 PyThreadState_New 函数创建的：

```
//Python/pystate.c
PyThreadState *
PyThreadState_New(PyInterpreterState *interp)
{
    return new_threadstate(interp, 1);
}


static PyThreadState *
new_threadstate(PyInterpreterState *interp, int init)
{
    _PyRuntimeState *runtime = &_PyRuntime;
    //创建线程对象
    PyThreadState *tstate = (PyThreadState *)PyMem_RawMalloc(sizeof(PyTh
readState));
    if (tstate == NULL) {
        return NULL;
    }

    //用于获取当前线程的 frame
    if (_PyThreadState_GetFrame == NULL) {
        _PyThreadState_GetFrame = threadstate_getframe;
    }

    //下面是线程的相关属性
    tstate->interp = interp;

    tstate->frame = NULL;
    tstate->recursion_depth = 0;
    tstate->overflowed = 0;
    tstate->recursion_critical = 0;
    tstate->stackcheck_counter = 0;
    tstate->tracing = 0;
    tstate->use_tracing = 0;
    tstate->gilstate_counter = 0;
    tstate->async_exc = NULL;
    tstate->thread_id = PyThread_get_thread_ident();

    tstate->dict = NULL;

    tstate->curexc_type = NULL;
    tstate->curexc_value = NULL;
    tstate->curexc_traceback = NULL;

    tstate->exc_state.exc_type = NULL;
    tstate->exc_state.exc_value = NULL;
    tstate->exc_state.exc_traceback = NULL;
    tstate->exc_state.previous_item = NULL;
    tstate->exc_info = &tstate->exc_state;

```

```
    tstate->c_profilefunc = NULL;
    tstate->c_tracefunc = NULL;
    tstate->c_profileobj = NULL;
    tstate->c_traceobj = NULL;

    tstate->trash_delete_nesting = 0;
    tstate->trash_delete_later = NULL;
    tstate->on_delete = NULL;
    tstate->on_delete_data = NULL;

    tstate->coroutine_origin_tracking_depth = 0;

    tstate->async_gen_firstiter = NULL;
    tstate->async_gen_finalizer = NULL;

    tstate->context = NULL;
    tstate->context_ver = 1;

    tstate->id = ++interp->tstate_next_unique_id;

    if (init) {
        _PyThreadState_Init(runtime, tstate);
    }

    HEAD_LOCK(runtime);
    tstate->prev = NULL;
    tstate->next = interp->tstate_head;
    if (tstate->next)
        tstate->next->prev = tstate;
    interp->tstate_head = tstate;
    HEAD_UNLOCK(runtime);

    return tstate;
}


//这一步 _PyThreadState_Init 就表示
//将线程对应的线程对象放入到我们刚才说的那个"线程状态对象链表"当中
void
_PyThreadState_Init(_PyRuntimeState *runtime, PyThreadState *tstate)
{
    _PyGILState_NoteThreadState(&runtime->gilstate, tstate);
}
```

这里有一个特别需要注意的地方，就是当前活动的 Python 线程不一定获得了 GIL。比如主线程获得了 GIL ，但是子线程还没有申请 GIL，那么操作系统也不会将其挂起。由于主线程和子线程都对应操作系统的原生线程，所以操作系统系统是可能在主线程和子线程之间切换的，因为操作系统级别的线程调度和 Python 级别的线程调度是不同的。

而当所有的线程都完成了初始化动作之后，操作系统的线程调度和 Python 的线程调度才会统一。那时 Python 的线程调度会迫使当前活动线程释放 GIL，而这一操作会触发操作系统内核用于管理线程调度的对象，进而触发操作系统对线程的调度。

所以我们说，Python对线程的调度是交给操作系统的，它使用的是操作系统内核的线程调度机制，当操作系统随机选择一个 OS 线程的时候，Python 就会根据这个 OS 线程去线程状态对象链表当中找到对应的线程状态对象，并赋值给那个保存当前活动线程的状态对象的变量。从而获取 GIL，执行字节码。

在执行一段时间之后，该线程会被强迫释放 GIL，然后操作系统再次调度，选择一个线程。而 Python 也会再次获取对应的线程状态对象，然后获取 GIL，执行一段时间字节码。而执行一段时间后，同样又会被被强迫释放 GIL，然后操作系统同样继续随机选择，依次往复。。。。。。

> 不过这里有一个问题，线程是如何得知自己被要求释放 GIL 呢？还记得 gil_drop_request 这个字段

> 吗？线程在执行字节码之前，会检测这个字段的值是否为 1，如果为 1，那么就知道自己要释放 GIL 了。

显然，当子线程还没有获取 GIL 的时候，相安无事。然而一旦 PyThreadState_New 之后，多线程机制初始化完成，那么子线程就开始争夺话语权了。

```
//Modules/_threadmodule.c
static void
t_bootstrap(void *boot_raw)
{
    //线程信息都在里面
    struct bootstate *boot = (struct bootstate *) boot_raw;
    //线程状态对象
    PyThreadState *tstate;
    PyObject *res;
    //获取线程状态对象
    tstate = boot->tstate;
    //拿到线程id
    tstate->thread_id = PyThread_get_thread_ident();
    _PyThreadState_Init(&_PyRuntime, tstate);

    //下面说
    PyEval_AcquireThread(tstate);

    //进程内部的线程数量+1
    tstate->interp->num_threads++;
    //执行字节码
    res = PyObject_Call(boot->func, boot->args, boot->keyw);
    if (res == NULL) {
        if (PyErr_ExceptionMatches(PyExc_SystemExit))
            /* SystemExit is ignored silently */
            PyErr_Clear();
        else {
            _PyErr_WriteUnraisableMsg("in thread started by", boot->func
);
        }
    }
    else {
        Py_DECREF(res);
    }
    Py_DECREF(boot->func);
    Py_DECREF(boot->args);
    Py_XDECREF(boot->keyw);
    PyMem_DEL(boot_raw);
    tstate->interp->num_threads--;
    PyThreadState_Clear(tstate);
    PyThreadState_DeleteCurrent();
    PyThread_exit_thread();
}
```

这里面有一个 PyEval_AcquireThread ，之前我们没有说，但如果我要说它是做什么的你就知道了。在里面子线程进行了最后的冲刺，并通过 PyThread_acquire_lock 争取GIL。

由于 GIL 现在被主线程持有，所以子线程会发现自己获取不到，于是会将自己挂起。而操作系统没办法靠自己的力量将其唤醒，只能等待 Python 的线程调度机制强迫主线程放弃 GIL、被子线程获取，然后触发操作系统内核的线程调度之后，子线程才会被唤醒。

然而当子线程被唤醒时，主线程却又陷入了苦苦的等待当中，同样苦苦地等待着 Python 强迫子线程放弃 GIL 的那一刻，假设我们这里只有一个主线程和一个子线程。

另外当子线程被线程调度机制唤醒之后，它所做的第一件事就是通过 PyThreadState_Swap 将维护当前线程状态对象的变量设置为其自身的状态对象，就如同操作系统进程的上下文环境恢复一样。这个 PyThreadState_Swap 我们就不展开说了，因为有些东西我们只需要知道是干什么的就行。

子线程获取了 GIL 之后，还不算成功，因为它还没有进入字节码解释器，就是那个大大的 for 循环，里面有一个巨大的 switch。于是子线程将回到 t_bootstrap，并进入 PyObject_Call ，从这里一路往前，最终调用 PyEval_EvalFrameEx ，才算是成功。

在 PyEval_EvalFrameEx 里面调用 _PyEval_EvalFrameDefault，执行字节码指令，所以此时才算是真正的执行，之前的都只能说是初始化。

而当进入 PyEval_EvalFrameEx 的那一刻，子线程就和主线程一样，完全受 Python 线程调度机制控制了。

Python 的线程调度

当主线程和子线程都进入了解释器后，Python线程之间的切换就完全由Python的线程调度机制掌控了。Python的线程调度机制肯定是在 PyEval_EvalFrameEx 里面的，因为线程是在执行字节码的时候切换的，那么肯定是在 PyEval_EvalFrameEx 里面。

而在分析字节码的时候，我们看到过 PyEval_EvalFrameEx ，尽管说它是字节码执行的核心，但它实际上是调用了 _PyEval_EvalFrameDefault。不过毕竟是从它开始的，所以我们还是说字节码核心是 PyEval_EvalFrameEx。

总之，在分析字节码的时候，我们并没有看线程的调度机制，那么下面就来分析一下。

```
PyObject* _Py_HOT_FUNCTION
_PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
{
    //......
    //大大的 for 循环
    for (;;) {
        //......
        //执行字节码之前先检测 gil_drop_request 是否为 1
        //如果为 1，则表示时间片用尽了，该释放 GIL 了
        //如果为 0，则表示时间片没用尽，还可以继续执行字节码
        if (_Py_atomic_load_relaxed(&ceval->gil_drop_request)) {
            if (_PyThreadState_Swap(&runtime->gilstate, NULL) != tstate) {
                Py_FatalError("ceval: tstate mix-up");
            }
            //如果 if 语句成立，说明要释放 GIL 了
            //于是调用 drop_gil 将 GIL 释放掉
            drop_gil(ceval, tstate);

        //释放 GIL 之后怎么办呢？显然还要继续争取，等待下一次调用
        //于是再次尝试获取 GIL，而这一步是阻塞的
        //假设总共有N个线程，那么会有 N-1 个线程阻塞在此处
        //因为 GIL 每次只能被一个线程获取
        //一旦当持有GIL的线程的时间片用尽、调用drop_gil释放GIL之后
        //那么阻塞在此处的 N-1 个线程，会有一个因获取到 GIL 而停止阻塞
        //至于刚刚释放 GIL 的线程会因为要再次获取而阻塞在这里
            take_gil(ceval, tstate);

            /* Check if we should make a quick exit. */
            exit_thread_if_finalizing(runtime, tstate);

            if (_PyThreadState_Swap(&runtime->gilstate, tstate) != NULL) {
                Py_FatalError("ceval: orphan tstate");
            }
        }
```

```
35          //......
36          //巨型 switch
37      }
38  }
```

所以相信现在应该明白，为什么 GIL 被称为是字节码层面上的互斥锁了。因为虚拟机就是以字节码为核心一条一条执行的，也就是说字节码是虚拟机执行的基本单元，但线程在执行字节码之前要先判断 gil_drop_request 是否为 0，也就是自己还能不能继续执行字节码指令。

如果不能执行，那么该线程就调用 drop_gil 函数将 GIL 释放掉（还会将那个维护线程状态对象的变量设置为 NULL），然后调用 take_gil 再次获取 GIL，等待下一次被调度。但是当该线程调用 drop_gil 之后，早已阻塞在 take_gil 处的等待线程会有一个获取到 GIL（并且会将那个变量设置为自身对应的线程状态对象）。而等到该线程再调用 take_gil 时，GIL 已被别的线程获取，那么该线程就会成为等待线程中新的一员。

也正因为如此，Python 才无法利用多核，因为 GIL 的存在使得每次只能有一个线程去执行字节码，而字节码又是执行的基本单元。并且还可以看出，每条字节码执行的时候不会被打断，因为一旦开始了字节码的执行，那么就必须等到当前的字节码指令执行完毕、进入下一次循环时才有可能释放 GIL。所以线程切换要么发生在字节码执行之前，要么发生在字节码执行之后，不会存在字节码执行到一半时被打断。

另外，释放 GIL 并不是立刻就让活跃线程停下来，因为活跃线程此时正在执行字节码指令，而字节码在执行的过程中不允许被打断。其实释放 GIL 的本质是线程调度机制发现活跃线程的执行时间达到 0.05 秒，于是将其 gil_drop_request 设置为 1。这样等到活跃线程将当前的字节码指令执行完毕、进入下一次循环时，看到 gil_drop_request 为 1、调用 drop_gil 之后，才会真正释放 GIL（将 locked 字段设置为 0）。

就这样通过 GIL 的释放、获取，每个线程都执行一会，依次往复。于是，Python 中无法利用多核的多线程机制，就这么实现了。

最后再补充一下，当一个 Python 线程在失去 GIL 时，它对应的 OS 线程依旧是活跃线程（此时会存在一个短暂的并行时间）。然后继续申请 GIL，但是 GIL 已被其它线程持有，于是触发操作系统的线程调度机制，将线程进行休眠。所以我们发现，线程释放 GIL 之后并不是马上就被挂起的，而是在释放完之后重新申请 GIL、但发现申请不到的时候才被挂起。

而当它再次申请到 GIL 时，那么又会触发操作系统的线程调度机制，将休眠的 OS 线程唤醒。然后遍历线程状态对象链表，找到对应的线程状态对象，并交给变量进行保存。

上面的线程调度被称为**标准调度**，标准调度是Python的调度机制掌控的，每个线程都是相当公平的，它适用于 CPU 密集型。

但是如果仅仅只有标准调度的话，那么可以说Python的多线程没有任何意义，但为什么又有很多场合适合使用多线程呢？就是因为调度方式除了标准调度之外，还存在阻塞调度。

阻塞调度是指，当某个线程遇到 IO 阻塞时，会主动释放 GIL，让其它线程执行，因为 IO 是不耗费 CPU 的。比如 time.sleep，或者从网络上请求数据等等，这些都是 IO 阻塞，那么会发生线程调度。

当阻塞的线程可以执行了，比如 sleep 结束、请求的数据成功返回，那么再切换回来。除了这一种情况之外，还有一种情况，也会导致线程不得不挂起，那就是 input 函数等待用户输入，这个时候也不得不释放 GIL。

而阻塞调度，则是借助操作系统实现的。

Python 子线程的销毁

我们创建一个子线程的时候，往往是执行一个函数，或者重写一个类继承自threading.Thread。而当一个子线程执行结束之后，Python 肯定要把对应的子线程销毁，当然销毁主线程和销毁子线程是不同的。销毁主线程必须要销毁 Python 的运行时环境，因为销毁主线程就意味着程序执行完毕了；而子线程的销毁则不需要这些动作，因此我们只看子线程的销毁。

通过前面的分析我们知道，线程的主体框架是在t_bootstrap中：

```
//Modules/_threadmodule.c
static void
t_bootstrap(void *boot_raw)
{
    struct bootstate *boot = (struct bootstate *) boot_raw;
    PyThreadState *tstate;
    PyObject *res;

    //......
    Py_DECREF(boot->func);
    Py_DECREF(boot->args);
    Py_XDECREF(boot->keyw);
    PyMem_DEL(boot_raw);
    tstate->interp->num_threads--;
    PyThreadState_Clear(tstate);
    PyThreadState_DeleteCurrent();
    PyThread_exit_thread();
}
```

Python 首先会将进程内部的线程数量减 1，然后通过 PyThreadState_Clear 清理当前线程所对应的线程状态对象。所谓清理实际上比较简单，就是改变引用计数。随后，再通过 PyThreadState_DeleteCurrent 函数释放gil。

```
//Modules/pystate.c
void
PyThreadState_DeleteCurrent()
{
    _PyThreadState_DeleteCurrent(&_PyRuntime);
}


static void
_PyThreadState_DeleteCurrent(_PyRuntimeState *runtime)
{
    struct _gilstate_runtime_state *gilstate = &runtime->gilstate;
    PyThreadState *tstate = _PyRuntimeGILState_GetThreadState(gilstate);
    if (tstate == NULL)
        Py_FatalError(
            "PyThreadState_DeleteCurrent: no current tstate");
    tstate_delete_common(runtime, tstate);
    if (gilstate->autoInterpreterState &&
        PyThread_tss_get(&gilstate->autoTSSkey) == tstate)
    {
        PyThread_tss_set(&gilstate->autoTSSkey, NULL);
    }
    _PyRuntimeGILState_SetThreadState(gilstate, NULL);
    PyEval_ReleaseLock();
}
```

过程很简单，首先会删除当前的线程状态对象，然后通过 PyEval_ReleaseLock 释放 GIL。不过这只是完成了绝大部分的销毁工作，而剩下的收尾工作就依赖于对应的操作系统了，当然这跟我们也就没关系了。

Python线程的用户级互斥与同步

我们知道在 GIL 的控制之下，线程之间对 Python 提供的 C API 访问都是互斥的，并且每次在字节码执行的过程中不会被打断，这可以看做是 Python 内核级的用户互斥。但是这种互斥不是我们能够控制的，内核级通过 GIL 的互斥保护了内核共享资源，比如**del obj**，它对应的指令是 DELETE_NAME，这个是不会被打断的。

但是像 **n = n + 1** 这种一行代码对应多条字节码，是可以被打断的，因为 GIL 是字节码层面的互斥锁，不是代码层面的互斥锁。如果在执行到一半的时候，碰巧 GIL 释放了，比如执行完 n + 1，但还没有赋值给 n，那么也会出岔子。所以我们还需要一种互斥，也就是用户级互斥。

实现用户级互斥的一种方法就是加锁，我们来看看Python提供的锁。

```
static PyMethodDef lock_methods[] = {
    {"acquire_lock", (PyCFunction)(void(*)(void))lock_PyThread_acquire_lock,
     METH_VARARGS | METH_KEYWORDS, acquire_doc},
    {"acquire",      (PyCFunction)(void(*)(void))lock_PyThread_acquire_lock,
     METH_VARARGS | METH_KEYWORDS, acquire_doc},
    {"release_lock", (PyCFunction)lock_PyThread_release_lock,
     METH_NOARGS, release_doc},
    {"release",      (PyCFunction)lock_PyThread_release_lock,
     METH_NOARGS, release_doc},
    {"locked_lock",  (PyCFunction)lock_locked_lock,
     METH_NOARGS, locked_doc},
    {"locked",       (PyCFunction)lock_locked_lock,
     METH_NOARGS, locked_doc},
    {"__enter__",    (PyCFunction)(void(*)(void))lock_PyThread_acquire_lock,
     METH_VARARGS | METH_KEYWORDS, acquire_doc},
    {"__exit__",    (PyCFunction)lock_PyThread_release_lock,
     METH_VARARGS, release_doc},
    {NULL,           NULL}              /* sentine
};
```


这些方法我们肯定都见过，acquire 表示上锁、release 就是解锁。假设有两个线程 A 和 B，A 线程先执行了 lock.acquire()，然后继续执行后面的代码。

这个时候依旧会进行线程调度，等到线程 B 执行的时候，也遇到了 lock.acquire()，那么不好意思 B 线程就只能在这里等着了。没错，是轮到 B 线程执行了，但由于我们在用户级层面上又设置了一把锁，而这把锁已经被 A 线程获取了。那么即使切换到 B 线程，但只要 A 还没有 lock.release()，B 也只能卡在 lock.acquire() 上面。因为A先拿到了锁，那么只要 A 不释放，B 就拿不到锁。

所以 GIL 是内核层面上的锁，我们使用 Python 开发时是控制不了的，把握不住，并且它提供的是以字节码为粒度的保护。而 threading.Lock 是用户层面上的锁，它提供的是以代码为粒度的保护，什么时候释放也完全由我们来控制，并且可以保护的代码数量没有限制。也就是说，在 lock.acquire() 和 lock.release() 之间写多少行代码都是可以的，而 GIL 每次只能保护一条字节码。

一句话，用户级互斥就是即便你拿到了GIL，也无法执行。

小结

以上就是 Python 的线程，以及 GIL 的实现原理。现在是不是对 GIL 有一个清晰的认识了呢？其实 GIL 没有什么神秘的，非常简单，就是一把字节码层面上的互斥锁。

而且通过 GIL，我们也知道了为什么 Python 不能利用多核。另外这里再提一个框架叫 Dpark，是模仿 Spark 的架构设计的，但由于 Python 多线程利用不了多核，于是将多线程改成了多进程。根据测试，Dpark 的表现还不如 Hadoop 的 MapReduce，所以 Python 的性能劣势抵消了 Spark 架构上带来的优势。

当然啦，Python 慢归慢，但是凭借着语法灵活、和 C 的完美兼容，以及丰富的第三方库，依旧走出了自己的社会主义道路，在编程语言排行榜上一直独领风骚。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》86. 内置函数解析

下一篇 ›
《源码探秘 CPython》84. 初识GIL、以及多个线程之间的调度机制