1.python的\_\_new\_\_

http://www.cnblogs.com/ifantastic/p/3175735.html

**python类型也是对象，其他对象都是object的基类，定义一个自己的新类，object会 先传进去cls和其他参数调用\_\_new\_\_，返回新对象，然后把对象当作self以及其他的 参数传进去给\_\_init\_\_。**

**单例模式好处：1.资源；2.全局唯一（递增唯一id）**

**class S(object):**

**def \_\_new\_\_(cls, \*args, \*\*kw):**

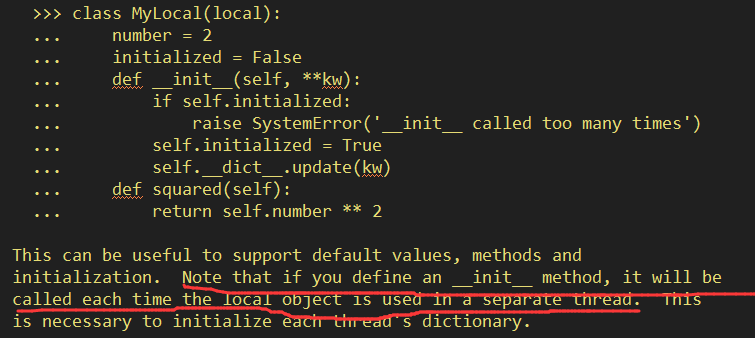
**if not hasattr(cls, '\_instance'):**

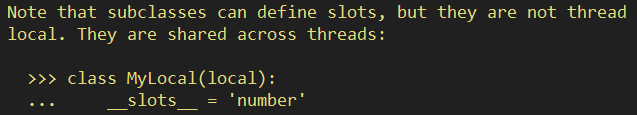
**obj = object.\_\_new\_\_(cls)**

**cls.\_instance = obj**

**return cls.\_instance**

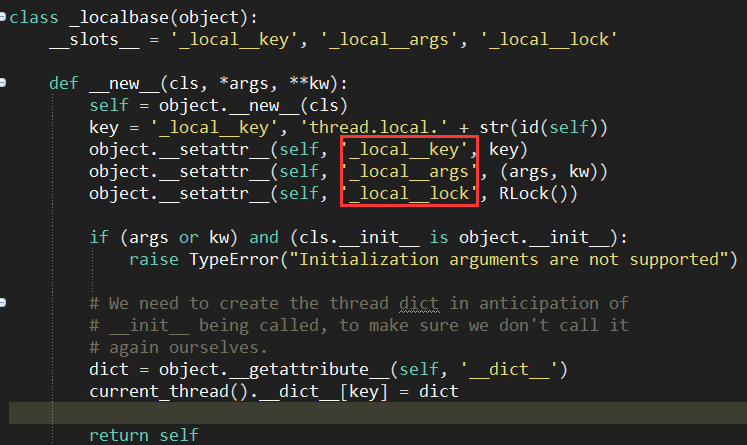
2.python的threading模块的local关键注释：





3.代码分析

a) 基类：



1.设置变量（生成对象，然后初始化参数）：

I) id(object)对象唯一的key;

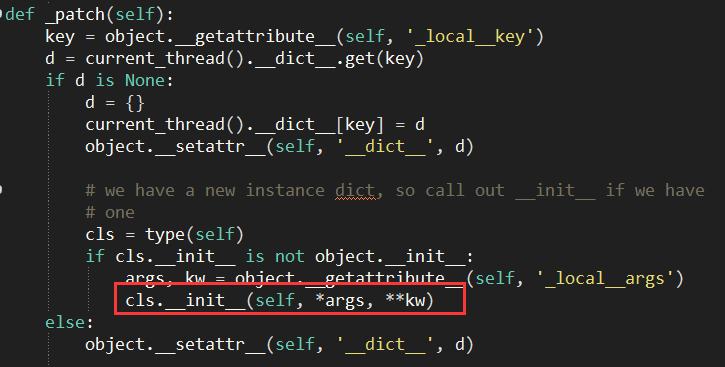
II) 参数；

III) 线程锁

2.把变量设置到当前线程的\_\_dict\_\_：

I) \_\_new\_\_之后系统自动调用\_\_init\_\_，而\_patch如果发现当前线程没有key， 会再调用一次\_\_init\_\_，因此要设置当前线程的key，防止重复\_\_init\_\_。

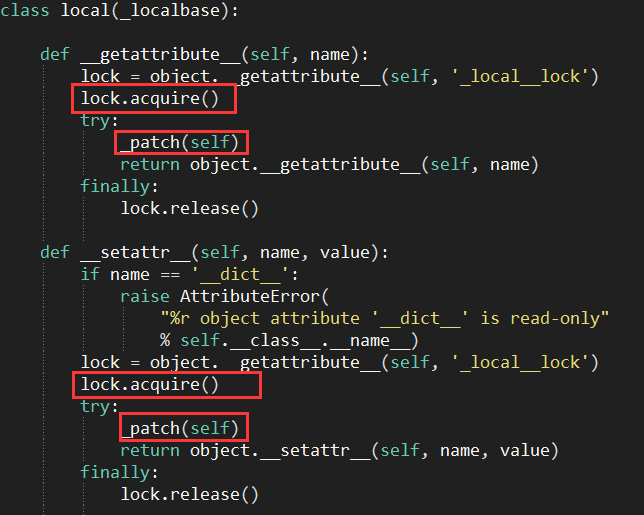
b) \_patch



如果发现当前线程没有该local变量，就会创建一个空的\_\_dict\_\_，设置到变量对象，

然后调用\_\_Init\_\_（如果切换到一个新的线程，就会调用\_\_init\_\_，但是如果之前已经切 换过，则不会重复\_\_init\_\_）。

c) local类：



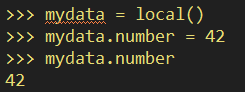
不管是set还是get，都是：

I) 获取锁；

II) 获取当前线程保存的该变量的\_\_dict\_\_

4.流程分析：

a.直接使用local：



-> 本地线程

-> \_localbase::\_\_new\_\_

-> object.\_\_init\_\_

-> 其他线程

-> \_\_getattribute\_\_

-> lock

-> self.\_\_dict\_\_ = {}

--因为没有自定义\_\_init\_\_，所以不用初始化

b.继承local，有自定义\_\_init\_\_

-> 本地线程

-> \_localbase::\_\_new\_\_

-> self.\_\_init\_\_

-> 其他线程

-> \_\_getattribute\_\_

-> lock

-> self.\_\_dict\_\_ = {}

-> \_local\_\_args

-> self.\_\_init\_\_(\_local\_\_args)

5.分析线程锁：

a).线程安全和可重入

I) 可重入：

一个函数被同一个线程调用2次以上，得到的结果具有可再现性(多次调用函数， 得到的结果是一样的)。那么我们说这个函数是可重入的。可函数可重入的概念最 先是在讲signal的handler的时候提出的。此时进程(线程)正在执行函数fun()，在 函数fun()还未执行完的时候， 突然进程接收到一个信号sig, 此时，需要暂停执行 fun(),要转而执行sig信号的处理函数sig\_handler()，那么，如果在sig\_handler()中， 也恰好调用了函 数fun().信号的处理是以软终端的形式进行的，那么，当 sig\_handler()执行完返回之后，CPU会继续从fun()被打断的地方往下执行。 这里讲 的比较特殊，最好的情况是，进程中调用了fun()，函数，信号处理函数sig\_handle() 中也调用了fun()。如果fun()函数是可重 入的，那么，多次调用fun()函数就具有 可再现性。从而，两次调用fun()的结果是正确的预期结果。非可重入函数，则恰 好相反。

II) 线程安全：

对于可竞争资源，有没有加锁保护。

b).GIL和线程：

http://zhuoqiang.me/python-thread-gil-and-ctypes.html

在 Python 语言的主流实现 CPython 中，GIL 是一个货真价实的全局线程锁，在 解释器解释执行任何 Python 代码时，都需要先获得这把锁才行，在遇到 I/O 操 作时会释放这把锁。如果是纯计算的程序，没有 I/O 操作，解释器会每隔 100 次 操作就释放这把锁，让别的线程有机会执行（这个次数可以通过 sys.setcheckinterval 来调整）。所以虽然 CPython 的线程库直接封装操作系 统的原生线程，但 CPython 进程做为一个整体，同一时间只会有一个获得了 GIL 的线程在跑，其它的线程都处于等待状态等着 GIL 的释放。这也就解释了我们上 面的实验结果：虽然有两个死循环的线程，而且有两个物理 CPU 内核，但因为 GIL 的限制，两个线程只是做着分时切换，总的 CPU 占用率还略低于 50％。

解决方法是：多进程和ctypes（调用C拓展都会释放锁）。

c).原子操作：

GIL 保证的是 python 操作码的原子性（用dis模块dis.dis就可以看）。

GIL好处是保证整个python执行都是串行的，但是GIL只是保护到字节码（一个加 法也有几条字节码），所以还 是要靠加锁来保证线程安全（list就不是线程 安全）。

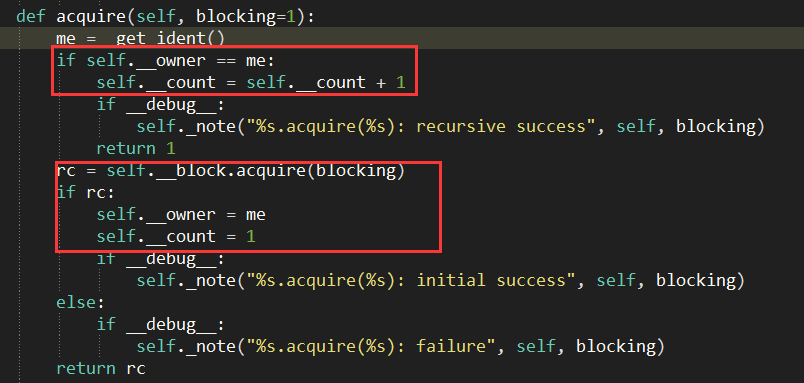
d).加锁操作的线程安全：

因为这个锁是针对线程的，所以在加锁和解锁的过程中，就已经假设是在某一个线 程里面，是不会有竞争的。

5.细节分析：

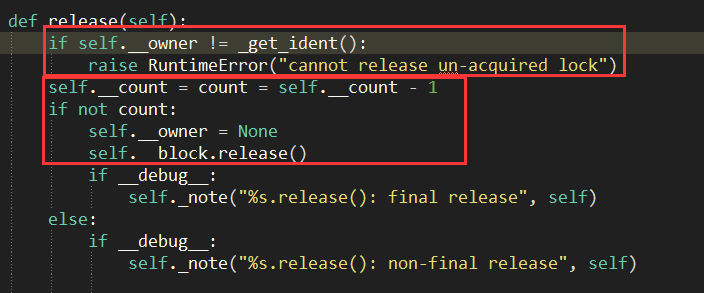
a).锁

I) 加锁：



递归锁

II) 解锁：



III) 为什么要锁：

-> \_\_getattribute\_\_

-> 线程1

-> get \_local\_\_lock

-> acquire lock

-> \_patch

-> 线程2

-> get \_local\_\_lock

-> acquire lock (fail)

-> 线程1

-> return reusult

V) 为什么要递归锁：

获取某个类成员变量，可能还要获取其他成员变量。

b).\_\_slots\_\_的作用：

???现在还搞不清楚，语言规则，先不管了。