12.14 在Unix系统上面启动守护进程¶

问题¶

你想编写一个作为一个在Unix或类Unix系统上面运行的守护进程运行的程序。

解决方案¶

创建一个正确的守护进程需要一个精确的系统调用序列以及对于细节的控制。 下面的代码展示了怎样定义一个守护进程，可以启动后很容易的停止它。

```
#!/usr/bin/env python3
# daemon.py

import os
import sys

import atexit
import signal

def daemonize(pidfile, *, stdin='/dev/null',
                          stdout='/dev/null',
                          stderr='/dev/null'):

    if os.path.exists(pidfile):
        raise RuntimeError('Already running')

    # First fork (detaches from parent)
    try:
        if os.fork() > 0:
            raise SystemExit(0)   # Parent exit
    except OSError as e:
        raise RuntimeError('fork #1 failed.')

    os.chdir('/')
    os.umask(0)
    os.setsid()
    # Second fork (relinquish session leadership)
    try:
        if os.fork() > 0:
            raise SystemExit(0)
    except OSError as e:
        raise RuntimeError('fork #2 failed.')

    # Flush I/O buffers
    sys.stdout.flush()
    sys.stderr.flush()

    # Replace file descriptors for stdin, stdout, and stderr
    with open(stdin, 'rb', 0) as f:
        os.dup2(f.fileno(), sys.stdin.fileno())
    with open(stdout, 'ab', 0) as f:
        os.dup2(f.fileno(), sys.stdout.fileno())
    with open(stderr, 'ab', 0) as f:
        os.dup2(f.fileno(), sys.stderr.fileno())

    # Write the PID file
    with open(pidfile,'w') as f:
        print(os.getpid(),file=f)

    # Arrange to have the PID file removed on exit/signal
    atexit.register(lambda: os.remove(pidfile))

    # Signal handler for termination (required)
    def sigterm_handler(signo, frame):
        raise SystemExit(1)
```

```
    signal.signal(signal.SIGTERM, sigterm_handler)

def main():
    import time
    sys.stdout.write('Daemon started with pid {}\n'.format(os.getpid()))
    while True:
        sys.stdout.write('Daemon Alive! {}\n'.format(time.ctime()))
        time.sleep(10)

if __name__ == '__main__':
    PIDFILE = '/tmp/daemon.pid'

    if len(sys.argv) != 2:
        print('Usage: {} [start|stop]'.format(sys.argv[0]), file=sys.stderr)
        raise SystemExit(1)

    if sys.argv[1] == 'start':
        try:
            daemonize(PIDFILE,
                      stdout='/tmp/daemon.log',
                      stderr='/tmp/dameon.log')
        except RuntimeError as e:
            print(e, file=sys.stderr)
            raise SystemExit(1)

        main()

    elif sys.argv[1] == 'stop':
        if os.path.exists(PIDFILE):
            with open(PIDFILE) as f:
                os.kill(int(f.read()), signal.SIGTERM)
        else:
            print('Not running', file=sys.stderr)
            raise SystemExit(1)

    else:
        print('Unknown command {!r}'.format(sys.argv[1]), file=sys.stderr)
        raise SystemExit(1)
```

要启动这个守护进程，用户需要使用如下的命令：

```
bash % daemon.py start
bash % cat /tmp/daemon.pid
2882
bash % tail -f /tmp/daemon.log
Daemon started with pid 2882
Daemon Alive! Fri Oct 12 13:45:37 2012
Daemon Alive! Fri Oct 12 13:45:47 2012
...
```

守护进程可以完全在后台运行，因此这个命令会立即返回。 不过，你可以像上面那样查看与它相关的pid文件和日志。 要停止这个守护进程，使用：

```
bash % daemon.py stop
bash %
```

讨论¶

本节定义了一个函数 `daemonize()` ，在程序启动时被调用使得程序以一个守护进程来运行。 `daemonize()` 函数只接受关键字参数，这样的话可选参数在被使用时就更清晰了。 它会强制用户像下面这样使用它：

```
daemonize('daemon.pid',
          stdin='/dev/null',
          stdout='/tmp/daemon.log',
          stderr='/tmp/daemon.log')
```

而不是像下面这样含糊不清的调用：

```
# Illegal. Must use keyword arguments
daemonize('daemon.pid',
          '/dev/null', '/tmp/daemon.log','/tmp/daemon.log')
```

创建一个守护进程的步骤看上去不是很易懂，但是大体思想是这样的， 首先，一个守护进程必须要从父进程中脱离。这是由 `os.fork()` 操作来完成的，子进程创建之后，父进程立即被终止。

在子进程变成孤儿后，调用 `os.setsid()` 创建了一个全新的进程会话，并设置子进程为首领。 它会设置这个子进程为新的进程组的首领，并确保不会再有控制终端。 如果这些听上去太魔幻，因为它需要将守护进程同终端分离开并确保信号机制对它不起作用。 调用 `os.chdir()` 和 `os.umask(0)` 改变了当前工作目录并重置文件权限掩码。 修改目录通常是个好主意，因为这样可以使得它不再工作在被启动时的目录。

另外一个调用 `os.fork()` 在这里更加神秘点。 这一步使得守护进程失去了获取新的控制终端的能力并且让它更加独立（本质上，该daemon放弃了它的会话首领地位，因此再也没有权限去打开控制终端了）。 尽管你可以忽略这一步，但是最好不要这么做。

一旦守护进程被正确的分离，它会重新初始化标准I/O流指向用户指定的文件。 这一部分有点难懂。跟标准I/O流相关的文件对象的引用在解释器中多个地方被找到 （sys.stdout, sys.__stdout__等）。 仅仅简单的关闭 `sys.stdout` 并重新指定它是行不通的， 因为没办法知道它是否全部都是用的是 `sys.stdout` 。 这里，我们打开了一个单独的文件对象，并调用 `os.dup2()` ， 用它来代替被 `sys.stdout` 使用的文件描述符。 这样，`sys.stdout` 使用的原始文件会被关闭并由新的来替换。 还要强调的是任何用于文件编码或文本处理的标准I/O流还会保留原状。

守护进程的一个通常实践是在一个文件中写入进程ID，可以被其他程序后面使用到。 `daemonize()` 函数的最后部分写了这个文件，但是在程序终止时删除了它。 `atexit.register()` 函数注册了一个函数在Python解释器终止时执行。 一个对于SIGTERM的信号处理器的定义同样需要被优雅的关闭。 信号处理器简单的抛出了 `SystemExit()` 异常。 或许这一步看上去没必要，但是没有它， 终止信号会使得不执行 `atexit.register()` 注册的清理操作的时候就杀掉了解释器。 一个杀掉进程的例子代码可以在程序最后的 `stop` 命令的操作中看到。

更多关于编写守护进程的信息可以查看《UNIX 环境高级编程》, 第二版 by W. Richard Stevens and Stephen A. Rago (Addison-Wesley, 2005)。 尽管它是关注与C语言编程，但是所有的内容都适用于Python， 因为所有需要的POSIX函数都可以在标准库中找到。