psutil

1. psutil（python 系统和进程实用程序）是一个跨平台库，用于检索 Python 中运行的进程和系统利用率（CPU，内存，磁盘，网络，传感器）的信息。主要用于系统监控，分析，限制流程资源和运行流程管理。它实现了诸如 ps，top，lsof，netstat，ifconfig，who，df，kill，free，nice，ionice，iostat，iotop，uptime，pidof，tty，taskset，pmap 等命令行工具提供的许多功能。（psutil、进程和系统利用率）
2. 安装 psutil 最简单的方式是通过 pip ：

```
pip install psutil
```

在 UNIX 上，这需要安装 C 编译器（例如 gcc）。

a) CPU

1. psutil.cpu_times(percpu = False)　（系统 CPU）

将返回一个指定的系统 CPU 时间元组。每个属性表示 CPU 在给定模式下花费的秒数。属性可用性因平台而异：

- user：正常进程在用户模式下执行的时间; 在 Linux 上也包括客人时间
- system：进程在内核模式下执行的时间
- idle：时间花费什么都不做

(1)　平台特定字段：

- nice(UNIX)：在用户模式下执行的被划分（优先级）的进程花费的时间; 在 Linux 上也包括 guest_nice 的时间
- iowait(Linux)：等待 I/O 完成的时间
- irq(Linux，BSD)：维护硬件中断所花费的时间
- softirq(Linux)：用于维护软件中断的时间
- steal(Linux 2.6.11+)：在虚拟化环境中运行的其他操作系统花费的时间
- guest(Linux 2.6.24+)：在 Linux 内核控制下为客户机操作系统运行虚拟 CPU 的时间
- guest_nice(Linux 3.2.0+)：运行一个接收到的客户机的时间（在 Linux 内核控制的客户机操作系统的虚拟 CPU）
- interrupt(Windows)：用于维护硬件中断的时间（在 UNIX 上类似于“irq”）
- dpc(Windows)：为延迟过程调用（DPC）服务的时间; DPC 是比标准中断优先级更低的中断。

当 percpu 为 True 时返回系统上每个逻辑 CPU 的命名元组列表。列表的第一个元素是指第一个 CPU，第二个元素到第二个 CPU，依此类推。列表的顺序和调用中的顺序是一致的。linux 上的示例：（percpu、每个逻辑 CPU）

```
>>> import psutil
>>> psutil.cpu_times()
scputimes(user=17411.7, nice=77.99, system=3797.02, idle=51266.57,
iowait=732.58, irq=0.01, softirq=142.43, steal=0.0, guest=0.0, guest_nice=0.0)
```

2. psutil.cpu_percent(interval = None,percpu = False)　（利用率、百分比）

返回一个浮点数，代表当前 cpu 的利用率的百分比，包括 system 和 user. 当 interval 为 0 或者 None 时，表示的是 interval 时间内的 sys 的利用率。interval 指定的是计算 cpu 使用率的时间间隔，percpu 则指定是选择总的使用率还是每个 cpu 的使用率。

```
>>> import psutil
>>> # blocking
>>> psutil.cpu_percent(interval=1)
2.0
>>> # non-blocking (percentage since last call)
>>> psutil.cpu_percent(interval=None)
2.9
>>> # blocking, per-cpu
>>> psutil.cpu_percent(interval=1, percpu=True)
[2.0, 1.0]
>>>
```

3. psutil.cpu_times_percent(interval=None，percpu=False)（特定CPU、利用率）

与cpu_percent()相同，但提供了由psutil.cpu_times(percpu=True)返回的每个特定CPU时间的利用率。interval和percpu参数与cpu_percent()含义相同。

4. psutil.cpu_count(logical=True)　　（cpu数量）

返回系统中逻辑CPU的数量（与Python 3.4中的os.cpu_coun()相同）。示例：

```
>>> import psutil
>>> psutil.cpu_count()
4
>>> psutil.cpu_count(logical=False)
2
```

5. psutil.cpu_stats()　　（cpu统计信息）

返回各种CPU统计信息作为一个指定的元组：

- ctx_switches：启动后的上下文切换（自愿+非自愿）的数量。
- interrupts：启动后的中断次数。
- soft_interrupts：启动后的软件中断数。 在Windows和SunOS上始终设置为0 。
- syscalls：启动后的系统调用次数。 在Linux上始终设置为0 。

示例（Linux）：

```
>>> import psutil
>>> psutil.cpu_stats()
scpustats(ctx_switches=20455687, interrupts=6598984, soft_interrupts=2134212,
syscalls=0)
```

6. psutil.cpu_freq(percpu = False)　　（cpu频率）

返回CPU频率的元组，元组包括以Mhz表示的当前、最小和最大频率。在Linux上，当前频率报告实时值，在所有其他平台上，它代表着名义上的“固定”值。 如果percpu为True ，并且系统支持per-cpu频率检索（仅限于Linux），则会为每个CPU返回一个频率列表，否则返回带有单个元素的列表。 如果最小和最大不能被确定，它们被设置为0 。示例（Linux）：

```
>>> import psutil
>>> psutil.cpu_freq()
scpufreq(current=931.42925, min=800.0, max=3500.0)
>>> psutil.cpu_freq(percpu=True)
```

```
[scpufreq(current=2394.945, min=800.0, max=3500.0),
 scpufreq(current=2236.812, min=800.0, max=3500.0),
 scpufreq(current=1703.609, min=800.0, max=3500.0),
 scpufreq(current=1754.289, min=800.0, max=3500.0)]
```

b) Memory

1. psutil.virtual_memory()　　（内存统计信息）

返回有关系统内存使用情况的统计信息的元组，包括以字节表示的以下字段。主要指标：

- total：总物理内存
- available：可以立即给予进程的内存，而不需要系统进行交换。 这通过根据平台求和不同的内存值来计算，并且它应该用于以跨平台方式监视实际的内存使用。

其他指标：

- used：已经使用的内存，根据平台计算不同，仅供参考。 total 减去 used 不一定等于 used。
- free：空闲内存。请注意，这并不反映实际的可用内存（可用）。total 减去 used 不一定等于 free。
- active(UNIX)：当前正在使用或最近使用的内存。
- inactive(UNIX)：标记为未使用的内存。
- buffers(Linux，BSD)：用于缓存文件系统元数据。
- cached(Linux，BSD)：缓存的各种东西。
- shared(Linux，BSD)：可以供多个进程同时访问的内存。
- wired(BSD，OSX)：标记为始终保留在 RAM 中的内存。 它永远不会移动到磁盘。

used 和 available 的总和不一定等于 total。 在 Windows 上 available 和 free 是一样的。

```
>>> import psutil
>>> mem = psutil.virtual_memory()
>>> mem
svmem(total=10367352832, available=6472179712, percent=37.6, used=8186245120,
free=2181107712, active=4748992512, inactive=2758115328, buffers=790724608,
cached=3500347392, shared=787554304)
>>>
>>> THRESHOLD = 100 * 1024 * 1024  # 100MB
>>> if mem.available <= THRESHOLD:
...     print("warning")
...
>>>
```

2. psutil.swap_memory()　　（交换内存）

返回系统交换内存统计信息的元组，其中包括以下字段：

- total：总交换内存（以字节为单位）
- used：以字节为单位的已经使用的交换内存
- free：以字节为单位的空闲交换内存
- percent：利用率，计算公式为(total-available)/total * 100
- sin：系统从磁盘（累积）中换入的字节数
- sout：系统从磁盘（累积）中换出的字节数

```
>>> import psutil
>>> psutil.swap_memory()
sswap(total=2097147904L, used=886620160L, free=1210527744L, percent=42.3,
sin=1050411008, sout=1906720768)
```

c) Disks

1. psutil.disk_partitions(all = False)　　（磁盘分区、列表）

在UNIX上类似于“df”命令，将所有安装的磁盘分区作为名称元组的列表（包括设备，安装点和文件系统类型）返回。 如果all参数都为False，它将尝试仅区分并返回物理设备（例如硬盘，CD-ROM驱动器，USB密钥），并忽略所有其他设备（例如，内存分区，如/dev/shm ）。

```
>>> import psutil
>>> psutil.disk_partitions()
[sdiskpart(device='/dev/sda3', mountpoint='/', fstype='ext4',
opts='rw,errors=remount-ro'),
 sdiskpart(device='/dev/sda7', mountpoint='/home', fstype='ext4', opts='rw')]
```

2. psutil.disk_usage(path)　　（磁盘统计信息）

返回将给定路径的磁盘使用统计信息的元组，包括以字节表示的total，used和free以及利用率。 如果路径不存在，则引发OSError 。 从Python 3.3开始，这也可以作为shutil.disk_usage()使用。

```
>>> import psutil
>>> psutil.disk_usage('/')
sdiskusage(total=21378641920, used=4809781248, free=15482871808, percent=22.5)
```

注意　UNIX通常为root用户保留总磁盘空间的5%。 UNIX上的total和used的字段指的是总的和已经占用的空间，而free表示用户可用的空间，百分比表示用户利用率（参见源代码 ）。 这就是为什么百分比值可能比您预期的要多5%。

3. psutil.disk_io_counters(perdisk = False)　（磁盘I/O信息）

返回磁盘I/O统计信息的元组，包括以下字段：

- read_count ：读取次数
- write_count ：写入次数
- read_bytes ：读取的字节数
- write_bytes ：写入的字节数

平台特定字段：

- read_time：（除NetBSD和OpenBSD之外的所有数据）从磁盘读取的时间（以毫秒为单位）
- write_time：（除了NetBSD和OpenBSD之外）写入磁盘的时间（以毫秒为单位）
- busy_time:(Linux，FreeBSD）实际I/O花费的时间（以毫秒为单位）
- read_merged_count(Linux)：合并读取数（见iostat doc ）
- write_merged_count(Linux)：合并写入数（见iostats doc ）

如果perdisk为True ，则将系统上安装的每个物理磁盘返回相同的信息，分区名称作为键的字典，并将上述名为元组的值作为值返回。

```
>>> import psutil
```

```
>>> psutil.disk_io_counters()
sdiskio(read_count=8141, write_count=2431, read_bytes=290203,
write_bytes=537676, read_time=5868, write_time=94922)
>>>
>>> psutil.disk_io_counters(perdisk=True)
{'sda1': sdiskio(read_count=920, write_count=1, read_bytes=2933248,
write_bytes=512, read_time=6016, write_time=4),
 'sda2': sdiskio(read_count=18707, write_count=8830, read_bytes=6060,
write_bytes=3443, read_time=24585, write_time=1572),
 'sdb1': sdiskio(read_count=161, write_count=0, read_bytes=786432,
write_bytes=0, read_time=44, write_time=0)}
```

d) Network

1. psutil.net_io_counters(pernic=False)　　（网络 I/O 信息）
   返回系统范围的网络 I/O 统计信息作为命名元组，其中包括以下属性：

- bytes_sent：发送的字节数
- bytes_recv：接收的字节数
- packets_sent：发送的报文数
- packets_recv：接收到的数据包数
- errin：接收时总错误数
- errout：发送时总错误数
- dropin：丢弃的传入数据包的总数
- dropout：丢弃的传出数据包的总数（在 OSX 和 BSD 上始终为 0）

如果 pernic 为 True，则将系统上安装的每个网络接口的相同信息作为字典返回，该字典具有网络接口名称作为键，上述名为元组的值为值。

```
>>> import psutil
>>> psutil.net_io_counters()
snetio(bytes_sent=14508483, bytes_recv=62749361, packets_sent=84311,
packets_recv=94888, errin=0, errout=0, dropin=0, dropout=0)
>>>
>>> psutil.net_io_counters(pernic=True)
{'lo': snetio(bytes_sent=547971, bytes_recv=547971, packets_sent=5075,
packets_recv=5075, errin=0, errout=0, dropin=0, dropout=0),
'wlan0': snetio(bytes_sent=13921765, bytes_recv=62162574, packets_sent=79097,
packets_recv=89648, errin=0, errout=0, dropin=0, dropout=0)}
```

2. psutil.net_connections(kind ='inet')　　（套接字）
   将系统范围的套接字连接作为元组返回。 每个元组提供 7 个属性：

```
>>> import psutil
>>> psutil.net_connections()
[pconn(fd=115, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_STREAM:
1>, laddr=('10.0.0.1', 48776), raddr=('93.186.135.91', 80),
status='ESTABLISHED', pid=1254),
 pconn(fd=117, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_STREAM:
1>, laddr=('10.0.0.1', 43761), raddr=('72.14.234.100', 80), status='CLOSING',
```

```
pid=2987),
 pconn(fd=-1, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_STREAM:
1>, laddr=('10.0.0.1', 60759), raddr=('72.14.234.104', 80),
status='ESTABLISHED', pid=None),
 pconn(fd=-1, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_STREAM:
1>, laddr=('10.0.0.1', 51314), raddr=('72.14.234.83', 443), status='SYN_SENT',
pid=None)
 ...]
```

3. psutil.net_if_addrs()　　（NIC 地址字典）

返回与系统上安装的每个 NIC（网络接口卡）相关联的地址的字典，键是 NIC 名，值为分配给 NIC 的每个地址的指定元组列表。 每个命名元组包括 5 个字段：

- family：地址系列， AF_INET ， AF_INET6 或 psutil.AF_LINK ，它是指 MAC 地址。
- address：主 NIC 地址（始终设置）。
- netmask：网络掩码地址（可能是 None ）。
- broadcast：广播地址（可能为 None ）。
- ptp：代表“点对点”；它是点对点接口（通常是 VPN）的目标地址。 广播和 ptp 是相互排斥的。 可能是 None 。

```
>>> import psutil
>>> psutil.net_if_addrs()
{'lo': [snic(family=<AddressFamily.AF_INET: 2>, address='127.0.0.1',
netmask='255.0.0.0', broadcast='127.0.0.1', ptp=None),
        snic(family=<AddressFamily.AF_INET6: 10>, address='::1',
netmask='ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff', broadcast=None, ptp=None),
        snic(family=<AddressFamily.AF_LINK: 17>, address='00:00:00:00:00:00',
netmask=None, broadcast='00:00:00:00:00:00', ptp=None)],
 'wlan0': [snic(family=<AddressFamily.AF_INET: 2>, address='192.168.1.3',
netmask='255.255.255.0', broadcast='192.168.1.255', ptp=None),
           snic(family=<AddressFamily.AF_INET6: 10>,
address='fe80::c685:8ff:fe45:641%wlan0', netmask='ffff:ffff:ffff:ffff::',
broadcast=None, ptp=None),
           snic(family=<AddressFamily.AF_LINK: 17>, address='c4:85:08:45:06:41',
netmask=None, broadcast='ff:ff:ff:ff:ff:ff', ptp=None)]}
>>>
```

4. psutil.net_if_stats()　　（NIC 信息）

将系统上安装的每个 NIC（网络接口卡）的信息作为字典返回，其关键字是 NIC 名称，值是具有以下字段的元组：

- isup：指示 NIC 是否启动并运行的 bool。
- duplex：双工通信类型；它可以是 NIC_DUPLEX_FULL ， NIC_DUPLEX_HALF 或 NIC_DUPLEX_UNKNOWN 。
- speed：NIC 速度以兆比特（MB）表示，如果无法确定（例如“localhost”），它将被设置为 0 。
- mtu：NIC 的最大传输单位，以字节表示。

```
>>> import psutil
```

```
>>> psutil.net_if_stats()
{'eth0': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_FULL: 2>, speed=100,
mtu=1500),
 'lo': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_UNKNOWN: 0>, speed=0,
mtu=65536)}
```

e) 传感器

1. psutil.sensors_temperatures(fahrenheit = False)　　（硬件温度）

返回硬件温度。每个条目是一个表示某个硬件温度传感器的命名元组（它可能是一个CPU，一个硬盘或别的东西，这取决于操作系统及其配置）。所有温度均以摄氏度表示，除非华氏度设为True 。如果OS不支持传感器，则返回一个空的dict。

```
>>> import psutil
>>> psutil.sensors_temperatures()
{'acpitz': [shwtemp(label='', current=47.0, high=103.0, critical=103.0)],
 'asus': [shwtemp(label='', current=47.0, high=None, critical=None)],
 'coretemp': [shwtemp(label='Physical id 0', current=52.0, high=100.0,
critical=100.0),
              shwtemp(label='Core 0', current=45.0, high=100.0, critical=100.0),
              shwtemp(label='Core 1', current=52.0, high=100.0, critical=100.0),
              shwtemp(label='Core 2', current=45.0, high=100.0, critical=100.0),
              shwtemp(label='Core 3', current=47.0, high=100.0,
critical=100.0)]}
```

2. psutil.sensors_fans()　　（风扇速度）

返回硬件风扇的速度。每个条目是一个表示某个硬件传感器风扇的命名元组。风扇速度以RPM（每分钟回合）表示。如果操作系统不支持传感器，则返回一个空的命令。 例：

```
>>> import psutil
>>> psutil.sensors_fans()
{'asus': [sfan(label='cpu_fan', current=3200)]}
```

3. psutil.sensors_battery()　　（电池状态）

将电池状态信息作为一个指定的元组返回，包括以下值。 如果没有安装电池或无法确定None返回。

- percent ：电池电量剩余百分比。
- secsleft ：电池电量耗尽之前剩下多少秒的粗略近似值。 如果连接了交流电源线，则设置为psutil.POWER_TIME_UNLIMITED 。 如果无法确定它设置为psutil.POWER_TIME_UNKNOWN 。
- power_plugged ：如果连接了交流电源线， True否则为“否”，如果无法确定，则为“ None 。

```
>>> import psutil
>>>
>>> def secs2hours(secs):
...     mm, ss = divmod(secs, 60)
...     hh, mm = divmod(mm, 60)
...     return "%d:%02d:%02d" % (hh, mm, ss)
```

```
...
>>> battery = psutil.sensors_battery()
>>> battery
sbattery(percent=93, secsleft=16628, power_plugged=False)
>>> print("charge = %s%%, time left = %s" % (batt.percent,
secs2hours(batt.secsleft)))
charge = 93%, time left = 4:37:08
```

f) 其他系统信息

1. psutil.boot_time()　　（电池启动时间）

返回系统启动时间，以秒为单位表示。 例：

```
>>> import psutil, datetime
>>> psutil.boot_time()
1389563460.0
>>> datetime.datetime.fromtimestamp(psutil.boot_time()).strftime("%Y-%m-%d %H:
%M:%S")
'2014-01-12 22:51:00'
```

2. psutil.users()　　（当前连接的用户）

将系统上当前连接的用户作为名称元组返回，其中包括以下字段：

- user：用户的名称。
- terminal：与用户关联的 tty 或伪 tty（如果有的话）否则 None 。
- host：与条目相关联的主机名，如果有的话。
- started：创建时间作为浮点数，以秒为单位表示。

例：

```
>>> import psutil
>>> psutil.users()
[suser(name='giampaolo', terminal='pts/2', host='localhost',
started=1340737536.0),
 suser(name='giampaolo', terminal='pts/3', host='localhost',
started=1340737792.0)]
```

g) 进程--函数

1. psutil.pids()　　（PID 列表）

返回当前运行 PID 的列表。为了迭代所有进程并避免竞争条件， process_iter()应该是首选的。

```
>>> import psutil
>>> psutil.pids()
[1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, ..., 32498]
```

2. psutil.pid_exists(pid)　　（PID 是否存在）

检查给定 PID 是否存在于当前进程列表中。这比直接使用“pid in psutil.pids()”语句更快，应该是首选。

3. psutil.process_iter()　　（Process 迭代器）

返回一个迭代器，为本地机器上的所有正在运行的Process创建一个Process类实例。每个实例只创建一次，然后缓存到一个内部表，每次元素生成时都会更新。检查缓存Process实例的身份，以便在其他进程重用PID的情况下安全，在这种情况下更新缓存的实例。对于迭代过程，这应该优于psutil.pids()。使用示例

```
import psutil
for proc in psutil.process_iter():
    try:
        pinfo = proc.as_dict(attrs=['pid', 'name'])
    except psutil.NoSuchProcess:
        pass
    else:
        print(pinfo)
```

4. psutil.wait_procs(procs，timeout=None，callback=None)（等待实例终止）

等待一系列Process实例终止。返回一个(gone, alive)元组，指示哪些进程已经消失，哪些进程还活着。那些将会有一个新的returncode属性指示进程退出状态（它可能是None）。callback是每次进程终止时被调用的函数（Process实例作为回调参数传递）。一旦所有进程终止或发生超时，功能将返回。典型用例是：

- 将SIGTERM发送到进程列表
- 给他们一些时间终止
- 发送SIGKILL到那些还活着

终止并等待此过程的所有子项的示例：

```
import psutil

def on_terminate(proc):
    print("process {} terminated with exit code {}".format(proc,
proc.returncode))

procs = psutil.Process().children()
for p in procs:
    p.terminate()
gone, still_alive = psutil.wait_procs(procs, timeout=3, callback=on_terminate)
for p in still_alive:
    p.kill()
```

h) Exceptions

1. class psutil.Error[source]　　（基类异常类）
   基类异常类。 所有其他异常都从此继承。
2. class psutil.NoSuchProcess(pid, name=None, msg=None)
   在当前进程列表中没有找到具有给定pid的进程时，或者进程不再存在时，由Process类方法引发。name是进程在消失之前的名称，只有在Process.name()之前被调用时才会设置。
3. class psutil.ZombieProcess(pid, name=None, ppid=None, msg=None）（僵尸进程）
   当在UNIX上查询僵尸进程（Windows没有僵尸进程）时，这可能由Process类方法引

发。

4. class psutil.AccessDenied(pid=None, name=None, msg=None)（被拒绝时引发）

   当执行操作的权限被拒绝时，由 Process 类方法引发。“name”是进程的名称（可以是 None ）。

5. class psutil.TimeoutExpired(seconds, pid=None, name=None, msg=None)

   由 Process.wait()引发，如果超时到期且进程仍然存在。

i) 进程类

1. class psutil.Process(pid=None)　　（给定 pid 的进程）

   表示具有给定 pid 的操作系统进程。如果 pid 被省略，则使用当前进程 pid（os.getpid()）。如果 pid 不存在，则触发 NoSuchProcess。

(1) oneshot()

实用上下文管理器，可以大大加快多进程信息的检索。 可以通过使用相同的例程来获取内部不同的进程信息（例如 name()， ppid()， uids()， create_time()，...），但只返回一个值，而其他值被丢弃。当使用这个上下文管理器时，内部程序被执行一次（在下面的例子中，name()），返回感兴趣的值，其他的被缓存。共享相同内部例程的后续调用将返回缓存值。 当退出上下文管理器块时，缓存被清除。建议是在每次检索有关过程的多个信息时使用。例：

```
>>> import psutil
>>> p = psutil.Process()
>>> with p.oneshot():
...     p.name()  # execute internal routine once collecting multiple info
...     p.cpu_times()  # return cached value
...     p.cpu_percent()  # return cached value
...     p.create_time()  # return cached value
...     p.ppid()  # return cached value
...     p.status()  # return cached value
...
>>>
```

2. 这里列出了可以优化的方法，具体取决于您所使用的平台。在下表中，水平空行表示什么过程方法可以在内部高效地组合在一起。最后一列（加速）显示加速，如果你调用所有的方法（最好的情况下），你可以得到的近似值。

Linux	Windows	OSX	BSD	SunOS
cpu_num()	cpu_percent()	cpu_percent()	cpu_num()	name()
cpu_percent()	cpu_times()	cpu_times()	cpu_percent()	cmdline()
cpu_times()	io_counters()	memory_info()	cpu_times()	create_time()
create_time()	ionice()	memory_percent()	create_time()	
name()	memory_info()	num_ctx_switches()	gids()	memory_info()
ppid()	nice()	num_threads()	io_counters()	memory_percent()
status()	memory_maps()		name()	nice()
terminal()	num_ctx_switches()	create_time()	memory_info()	num_threads()
	num_handles()	gids()	memory_percent()	ppid()
gids()	num_threads()	name()	num_ctx_switches()	status()
num_ctx_switches()	username()	ppid()	ppid()	terminal()
num_threads()		status()	status()	
uids()		terminal()	terminal()	gids()
username()		uids()	uids()	uids()
		username()	username()	username()
memory_full_info()				
memory_maps()				
speedup: +2.6x	*speedup: +1.8x / +6.5x*	*speedup: +1.9x*	*speedup: +2.0x*	*speedup: +1.3x*

(1) pid：该进程的PID。这是类的唯一（只读）属性。
(2) ppid()：进程的父PID。
(3) name()：进程名称。
(4) exe()：进程可执行文件的绝对路径。在一些系统上，这也可能是一个空字符串。返回值是第一次调用后缓存。

```
>>> import psutil
>>> psutil.Process().exe()
'/usr/bin/python2.7'
```

(5) cmdline()
调用命令行进程。返回值不缓存，因为进程的命令行可能会改变。

```
>>> import psutil
>>> psutil.Process().cmdline()
['python', 'manage.py', 'runserver']
```

(6) environ()
保存进程的环境变量，是一个字典。
(7) create_time()
进程的创建时间，是一个以秒为单位的浮点数，UTC。返回值是第一次调用后缓存。

```
>>> import psutil, datetime
>>> p = psutil.Process()
>>> p.create_time()
1307289803.47
>>> datetime.datetime.fromtimestamp(p.create_time()).strftime("%Y-%m-%d %H:%M:%S")
'2011-03-05 18:03:52'
```

(8) as_dict(attrs=None, ad_value=None)

检索多个处理信息的字典。如果指定ATTRS，它必须是一个反映可用列表Process类的属性名称，否则所有的公共（只读）假定属性。ad_value是被分配到的情况下的字典键的值或异常检索该特定处理信息时上升。

(9) parent()

它返回父进程作为Process对象，检查PID是否已被重新使用。如果PPID是未知的，返回None。

(10) status()

作为一个字符串的当前进程状态。返回的字符串是一个psutil.STATUS_*常数。

(11) cwd()

进程当前工作目录的绝对路径。

(12) username()

拥有该进程的用户名。在UNIX上，这是通过使用实际进程UID计算。

(13) uids()

这个过程是一个名为元组的真实，有效和保存的用户ID。这是相同的os.getresuid（） ，但可用于任何过程PID。

(14) gids()

一个真实、有效和保存组ID的元组。

(15) nice(value=None)[source]

获取或设置进程nice值（优先级）。在UNIX上，这是一个数量通常从去-20到20。该nice值越高，进程的优先级较低。

```
>>> import psutil
>>> p = psutil.Process()
>>> p.nice(10)  # set
>>> p.nice()  # get
10
>>>
```

(16) ionice(ioclass=None, value=None)

获取或设置过程I/O优先级。

```
>>> import psutil
>>> p = psutil.Process()
>>> p.ionice(psutil.IOPRIO_CLASS_IDLE)  # set
>>> p.ionice()  # get
pionice(ioclass=<IOPriority.IOPRIO_CLASS_IDLE: 3>, value=0)
>>>
```

(17) rlimit(resource, limits=None)

获取或设置进程资源限制。

```
>>> import psutil
>>> p = psutil.Process()
>>> # process may open no more than 128 file descriptors
>>> p.rlimit(psutil.RLIMIT_NOFILE, (128, 128))
>>> # process may create files no bigger than 1024 bytes
>>> p.rlimit(psutil.RLIMIT_FSIZE, (1024, 1024))
>>> # get
>>> p.rlimit(psutil.RLIMIT_FSIZE)
```

```
(1024, 1024)
>>>
```

(18) io_counters()

返回进程中的I/O统计数据的元组。

- read_count：执行的读取操作的数目（累计）
- write_count：执行的写操作的数目（累计）
- read_bytes：字节数读（累计）
- write_bytes：写入的字节数（累计）

(19) num_fds()

返回文件描述符的数量。

(20) num_threads()

返回线程的数量。

(21) threads()

返回由进程打开的元组，元组包括线程ID和线程的CPU时间。

(22) cpu_times()

返回（用户、系统、children_user、children_system的）累积处理时间的元组，以秒（见说明）为单位。

(23) cpu_percent()

返回表示该进程的CPU利用率。

(24) cpu_affinity(cpus=None)

获取或设置进程当前CPU亲和力。CPU亲和力在于告诉OS只运行哪些CPU。在Linux上，这是通过完成taskset命令。如果没有传递参数，返回当前CPU亲和力整数列表。如果通过了它必须是整数指定新的CPU亲和力的列表。

```
>>> import psutil
>>> psutil.cpu_count()
4
>>> p = psutil.Process()
>>> # get
>>> p.cpu_affinity()
[0, 1, 2, 3]
>>> # set; from now on, process will run on CPU #0 and #1 only
>>> p.cpu_affinity([0, 1])
>>> p.cpu_affinity()
[0, 1]
>>> # reset affinity against all eligible CPUs
>>> p.cpu_affinity([])
```

(25) cpu_num()

返回当前正在运行的CPU。

(26) memory_info()

返回依赖平台的进程的内存信息的变量字段的元组。

(27) memory_full_info()

该方法返回的信息和memory_info()相同，再加上一些平台（Linux操作系统，OSX，Windows）中，提供了额外的度量（USS，PSS和swap）。

(28) memory_percent(memtype="rss")

比较进程内存总物理系统内存和计算进程的内存利用率百分比。mem 规定要不比较的进程内存类型。

(29) memory_maps(grouped=True)

返回进程中的映射内存区域的元组，其字段是在不同的平台是不同的。

(30) children(recursive=False)

返回此子进程列表，抢先检查 PID 是否已被重用。

(31) open_files()

返回由进程打开的常规文件的元组，包括以下字段：

- path：绝对文件名。
- fd：文件描述符号；在 Windows 上，这是永远-1。
- position（Linux）：文件（偏移）的位置。
- mode（Linux）：指示文件的打开方式，同样一个字符串开放的 mode 论点。可能的值是'r'，'w'，'a'，'r+'和'a+'。
- flags（Linux）：它被传递到底层标志 os.open 调用 C 时，文件被打开（例如 os.O_RDONLY，os.O_TRUNC 等）。

(31) connections(kind="inet")

返回由进程打开的套接字连接的元组。

(32) is_running()

返回当前进程是否正在运行。

(33) send_signal()

将信号发送到进程。

(34) suspend()

暂停处理执行与 SIGSTOP 信号

(35) resume()

恢复进程执行。

(36) terminate()

终止进程。

(37) kill()

通过 SIGKILL 信号杀死进程。

(38) wait()

等待进程终止，如果进程是当前进程的子进程，返回退出代码，否则 None。

j) POPEN 类

1. class psutil.Popen(*args, **kwargs) （进程信息）

一个更方便的接口，用于获取用户启动的应用程序进程信息，以便跟踪程序进程的运行状态。

```
>>> import psutil
>>> from subprocess import PIPE
>>>
>>> p = psutil.Popen(["/usr/bin/python", "-c", "print('hello')"], stdout=PIPE)
>>> p.name()
'python'
>>> p.username()
'giampaolo'
>>> p.communicate()
('hello\n', None)
```

```
>>> p.wait(timeout=2)
0
>>>
```

Rrdtool

a) rrdtool 功能

1. create

   设立一个新的循环数据库（RRD）。由 create()函数完成。

2. update

   将新数据值存储到 RRD 中。向守护进程发送值，而不是将其写入磁盘本身。由 update()函数完成。

3. updatev

   在操作上相当于更新，同时返回输出。由 updatev()函数完成。

4. graph

   根据存储在一个或多个 RRD 中的数据创建图。除了生成图形外，还可以将数据提取到 stdout。由 graph()函数完成。

5. graphv

   根据存储在一个或多个 RRD 中的数据创建图。与 graph 相同，但 graphv 会在创建图形之前打印元数据。由 graphv()函数完成。

6. dump

   以普通 ASCII 转储 RRD 的内容。 关于 restore，您可以使用它将 RRD 从一个计算机体系结构移动到另一个计算机体系结构。由 dump()函数完成。

7. restore

   将 XML 格式的 RRD 恢复为二进制 RRD。由 restore()函数完成。

8. fetch

   从 RRD 获取特定时间段的数据。 图形函数使用 fetch 从 RRD 中检索其数据。由 fetch()函数完成。

9. tune

   改变 RRD 的设置和结构。由 tune()函数完成。

10. first

    找到 RRD 的第一个更新时间。由 rrdgraph()函数完成。

11. last

    查找 RRD 的最后更新时间。由 last()函数完成。

12. lastupdate

    查找 RRD 的最后更新时间。 它还返回最近更新中为每个基准存储的值。由 lastupdate

()函数完成。

13. info

    获取有关 RRD 的信息。由 info()函数完成。

14. resize

    更改单个 RRA 的大小。 这很危险！由 resize()函数完成。

15. xport

    导出从一个或多个 RRD 检索的数据。由 xport()函数完成。

16. flushcached

    刷新缓存，更新特定 RRD 文件的值。由 flushcached()函数完成。

b) Rrdcached

1. Rrdcached 是一个守护进程，它接收现有 RRD 文件的更新，累加它们，如果已经收到足够的数据或已经定义的时间过去，将更新写入 RRD 文件。 可以使用 flush 命令来强制将值写入磁盘，以便图形化设备和类似设备可以使用最新的数据。
2. update 命令向守护进程发送值，而不是将其写入磁盘本身。 所有其他命令可以在访问文件之前向守护程序发送一个 FLUSH 命令，因此即使缓存超时较大，它们也可以使用最新的数据。 （update、不写磁盘）

Paramiko

a) Channel

跨越 SSH 传输的安全通道。Channel 的行为就像套接字一样，并且具有与 Python 套接字 API 不可区分的 API。 （Channel、安全通道、套接字）

1. __repr__()

   返回此对象的字符串表示形式，供调试。

2. Close() （关闭）

   关闭 Channel。

3. exec_command(* args ， ** kwds) （执行命令）

在服务器上执行命令。 如果服务器允许，则该通道将直接连接到正在执行的命令的 stdin，stdout 和 stderr。命令执行结束后，通道将被关闭，不能重复使用。 如果要执行其他命令，则必须打开新通道。

4. exit_status_ready() （退出并返回、true）

如果远程进程退出并返回退出状态，则返回 true。 如果您不想在 recv_exit_status 中阻止，您可以使用它来轮询进程状态。 请注意，在某些情况下，服务器可能无法返回退出状态（例如不良服务器）。

5. Fileno() （文件描述符）

   返回一个 OS 级文件描述符，可以用于轮询，但不能用于读取或写入。

6. get_id() （int ID）

   返回此通道的 int ID。Channel ID 在传输中是唯一的，通常是一小部分。

7. get_name() （名称）

获取以前由 set_name 设置的此 Channel 的名称。

8. get_pty()　　（伪终端）

从服务器请求伪终端。

9. get_transport()　　（返回、transport）

返回与此通道关联的 transport。

10. Getpeername()　　（远端地址）

如果可能，返回此通道的远端的地址。

11. Gettimeout()　　（超时）

返回与套接字操作相关联的超时（以秒为单位）（如 float），如果未设置超时，则返回 None 。

12. invoke_shell(* args ， ** kwds)　（请求 shell 会话）

在此 Channel 上请求交互式 shell 会话。 如果服务器允许，则通道将直接连接到 shell 的 stdin，stdout 和 stderr。

13. invoke_subsystem(* args ， ** kwds)　（子系统）

在服务器上请求一个子系统（例如， sftp ）。如果服务器允许，则通道将直接连接到所请求的子系统。子系统完成后，通道将关闭，不能重复使用。

14. makefile(* params)　　（类文件对象）

返回与此通道关联的类文件对象。可选 mode 和 bufsize 参数的解释方式与 Python 中内置的 file()函数相同。

15. makefile_stderr(* params)　　（stderr、类文件对象）

返回与此通道的 stderr 流相关联的类文件对象。

16. recv(nbytes)　　（接收数据）

从 Channel 接收数据。 返回值是表示接收到的数据的字符串。 一次接收的最大数据量由 nbytes 指定。 如果返回长度为零的字符串，则通道流已关闭。

17. recv_exit_status( )　　（返回退出状态）

从服务器上的进程返回退出状态。

18. recv_ready( )　　（缓存并读取）

如果数据被缓冲并准备从该通道中读取，则返回 true。 False 结果并不意味着渠道关闭；这意味着您可能需要等待更多数据到达。

19. recv_stderr(nbytes)　　（stderr 流、接收数据）

从 Channel 的 stderr 流接收数据。 只有在没有 pty 的情况下使用 exec_command 或 invoke_shell Channel exec_command 在 stderr 数据流上拥有数据。 返回值是表示接收到的数据的字符串。 一次接收的最大数据量由 nbytes 指定。 如果返回长度为零的字符串，则通道流已关闭。

20. recv_stderr_ready( )　　（被缓冲、stderr 流中读取）

如果数据被缓冲并准备从该通道的 stderr 流中读取，则返回 true。

21. request_forward_agent(* args ， ** kwds)　　（请求 SSH 代理）

在此 Channel 上请求转发 SSH 代理。 这只适用于 OpenSSH 的 ssh 代理！

22. request_x11(* args ， ** kwds)　　（请求 X11 会话）

在此 Channel 上请求 x11 会话。 如果服务器允许，则在 shell 会话中运行 x11 应用程序时，可以从服务器向客户端进一步执行 x11 请求。

23. resize_pty(* args ， ** kwds)　　（调整大小）

调整伪终端的大小。 这可以用来改变先前 get_pty 调用中创建的终端仿真的宽度和高度。

24. send　　（发送数据）

发送数据到 Channel。 返回发送的字节数，如果通道流关闭，则返回 0。 应用程序负责检查所有数据是否已发送：如果仅传输了一些数据，则应用程序需要尝试传送剩余的数据。

25. send_exit_status(status)　　（退出状态发送给客户端）

将执行的命令的退出状态发送给客户端。 许多客户端希望在完成后从执行的命令中获取某种状态代码。

26. send_ready( )　　（可写而不阻塞）

如果数据可以写入此通道而不阻塞，则返回 true。 这意味着通道关闭（所以任何写入尝试都会立即返回），或者出站缓冲区中至少有一个字节的空间。 如果出站缓冲区中至少有一个字节的空间，则 send 调用将立即成功，并返回实际写入的字节数。

27. send_stderr(s)　　（发送到）

将数据发送到“stderr”流上的 Channel。 这通常仅由服务器用于从 shell 命令发送输出 - 客户端不会使用此命令。 返回发送的字节数，如果通道流关闭，则返回 0。 应用程序负责检查所有数据是否已发送：如果仅传输了一些数据，则应用程序需要尝试传送剩余的数据。

28. sendall(s)　　（所有数据）

将所有数据发送到通道。 与 send 不同，此方法继续从给定的字符串发送数据，直到所有数据已发送或发生错误为止。 没有任何东西返回。

29. sendall_stderr(s)　　（所有数据、“stderr”流）

将所有数据发送到 Channel 的“stderr”流。 与 send_stderr 不同，该方法继续从给定的字符串发送数据，直到发送所有数据或发生错误。 没有任何返回。

30. set_combine_stderr(combine)　　（是否合并）

设置 stderr 是否应该合并到此通道的 stdout 中。 默认值为 False ，但在某些情况下，将两个流组合起来可能很方便。

31. set_environment_variable(* args ， ** kwds)　　（设置）

设置环境变量的值。

32. set_name(name)　　（通道名称）

设置此通道的名称。 目前它只用于在日志文件条目中设置通道的名称。 可以使用 get_name 方法获取该名称。

33. settimeout(timeout)　　（设置超时）

在阻止读/写操作时设置超时。 timeout 参数可以是表示秒的非负的浮点数，也可以是 None 。 如果给定了浮点数，则在操作完成之前，如果超时周期值已经过去，后续通道读/写操作将引发超时异常。 设置超时为 None 禁用套接字操作超时。

34. shutdown(how)　　（关闭）

关闭一个或两个连接中的一个。 如果是 0，则不接受进一步的接收。 如果是 1，则不允许进一步发送。 如果是 2，进一步发送和接收是不允许的。 这将在一个或两个方向上关闭流。

35. shutdown_write( )　　（关闭发送端）

关闭发送端，关闭流出方向的流。

36. update_environment(* args ， ** kwds)　（更新）

更新此通道的远程 shell 环境。

37. __repr__( )

返回此对象的字符串表示形式，供调试。

b) SSHClient 类

1. __init__()

创建一个新的 SSHClient。

2. close()　　（关闭 SSHClient）

关闭此 SSHClient 及其底层 Transport 。

3. connect()　　（连接）

连接到 SSH 服务器并进行身份验证。

4. exec_command()　　（执行命令）

在 SSH 服务器上执行命令。 打开一个新 Channel ，并执行所请求的命令。 命令的输入和输出流作为表示 stdin，stdout 和 stderr 的 Python file 样对象返回。

5. get_host_keys()　　（获取）

获取本地的 HostKeys 对象。 这可以用于检查本地主机密钥或更改它们。

6. get_transport()　　（返回连接）

返回此 SSH 连接的底层 Transport 对象。 这可以用于执行较低级别的任务，如打开特定类型的通道。

7. invoke_shell()　　（启动 shell）

在 SSH 服务器上启动交互式 shell 会话。 打开新 Channel 并使用所请求的终端类型和大小连接到伪终端。

8. load_host_keys(filename)　　（加载密钥）

从本地主机密钥文件加载主机密钥。

9. load_system_host_keys(filename = None)　（加载主机密钥）

从系统（只读）文件加载主机密钥。 使用此方法读取的主机密钥将不会被 save_host_keys 保存。

10. open_sftp()　　（打开 SFTP 会话）

在 SSH 服务器上打开 SFTP 会话。

11. set_log_channel(name)　　（设置）

设置记录通道。 默认值为"paramiko.transport"但可以设置为任何您想要的。

12. set_missing_host_key_policy(policy)　　（设置、没有主机密钥、策略）

设置连接到没有已知主机密钥的服务器时使用的策略。

c) Message 类

1. __init__ (content = None)

创建一个新的 SSH2 消息。

2. __repr__( )

返回此对象的字符串表示形式，供调试。

3. __str__( )

返回此消息的字节流内容。

4. __weakref__

列表对对象的弱引用（如果已定义）

5. add()

向流中添加一个项目序列。 这些值根据它们的类型进行编码：str，int，bool，list 或 long。

6. add_adaptive_int(n)

向流中添加一个整数。

7. add_boolean(b)

向流中添加一个布尔值。

8. add_byte(b)

向流中写入一个字节，无需任何格式化。

9. add_bytes(b)

将字节写入流，而不进行任何格式化。

10. add_int(n)

向流中添加一个整数。

11. add_int64(n)

向流中添加 64 位 int。

12. add_list(l)

向流中添加一个字符串列表。 它们与使用逗号分隔的单个字符串。

13. add_mpint(z)

为流添加一个长整型，编码为无限精度整数。 此方法仅适用于正数。

14. add_string(s)

向流中添加一个字符串。

15. asbytes( )

将该消息的字节流内容返回为字节。

16. get_adaptive_int( )

从流中获取一个 int。

17. get_binary( )

从流中获取字符串。 这可能是一个字节字符串，可能包含不可打印的字符。（字符串不包含另一个字节流消息）

18. get_boolean( )

从流中获取布尔值。

19. get_byte( )

返回消息的下一个字节。 这相当于 get_bytes(1) 。

20. get_bytes(n)

返回消息的下一个 n 个字节（作为一个 str ），而不会分解成 int，解码的字符串等。只是返回原始字节。 如果消息中不存在 n 个字节，则返回一个 n 个零字节的字符串。

21. get_int( )

从流中获取一个 int。

22. get_int64( )

从流中获取 64 位 int。

23. get_list( )

从流中获取 strings list 。

24. get_mpint( )

从流中获取长整型（mpint）。

25. get_remainder( )

返回尚未解析并返回的消息的字节（作为 str ）。

26. get_so_far( )

返回已解析并返回的消息的 str 字节。 传入消息构造函数的字符串可以通过连接 get_so_far 和 get_remainder 。

27. get_string( )

从流中获取一个 str 。 这可能是一个字节字符串，可能包含不可打印的字符。（字符串不包含另外的字节流消息）。

28. get_text( )

从流中获取 Unicode 字符串。

29. rewind( )

将消息倒回到开始，好像没有任何项目已被解析出来

d) 数据包处理 **Packetizer** 类

1. __weakref__

弱引用对象的列表（如果已定义）。

2. complete_handshake( )

告诉 Packetizer 握手已经完成。

3. handshake_timed_out( )

检查握手是否超时。

4. need_rekey( )

如果需要协商一组新的密钥，则返回 True 。 这将在数据包读取或写入期间触发，因此，应在每次读取或写入后进行检查，或至少每隔几个月检查一次。

5. read_all()

读取尽可能接近 N 个字节，只要必要时阻止。

6. read_message( )

   此功能中只能有一个线程（没有其他锁定完成）。

7. readline()

   从套接字读取一行。 我们假设没有数据在线后面挂起，所以可以尝试大量读取。

8. send_message()

   使用当前密码写入一个数据块，作为 SSH 块。

9. set_inbound_cipher()

   切换入站数据密码。

10. set_keepalive()

    打开/关闭回调 keepalive。 如果 interval 秒数通过，没有从套接字读取或写入数据，则回调将被执行，定时器将被复位。

11. set_log()

    设置用于日志记录的 Python 日志对象。

12. set_outbound_cipher()

    切换出站数据密码。

13. start_handshake()

    告诉 Packetizer ，握手过程开始。 启动一个可以在握手过程中发出超时信号的保持计时器。

e) 传输类 **Transport**

SSH 传输连接到流（通常为套接字），协商加密会话，认证，然后在会话中创建称为 channels 流隧道。 多个通道可以跨单个会话进行复用（通常在端口转发的情况下）。

1. Accept()　　（返回、下一个通道）

   以服务器模式返回客户端通过此传输打开的下一个通道。 如果在给定超时之前没有频道被打开，则返回 None 。

2. add_server_key()　　（添加到、密钥列表）

   将主机密钥添加到用于服务器模式的密钥列表中。 当作为服务器运行时，主机密钥用于在 SSH2 协商期间签署某些数据包，以便客户端可以相信我们是我们所说的。 因为这是用于签名，密钥必须包含私钥信息，而不仅仅是公开的一半。 只保留每种类型的一个密钥（RSA 或 DSS）。

3. Atfork()　　（终止、不关闭）

   终止此运输而不关闭会话。 在 posix 系统上，如果传输在进程分叉期间打开，则父进程和子节点都将共享底层套接字，但只有一个进程可以使用连接（不会破坏会话）。 使用此方法清理传输对象，而不会中断其他进程。

4. auth_gssapi_keyex()　　（GSS-API / SSPI 验证）

   如果使用 GSS-API kex，则使用 GSS-API / SSPI 验证服务器。

5. auth_gssapi_with_mic ()　　（GSS-API / SSPI 验证）

   使用 GSS-API / SSPI 验证服务器。

6. auth_interactive()　　（交互、验证）

   以交互方式验证服务器。 处理程序用于从服务器回答任意问题。 在许多服务器上，

这只是 PAM 周围的一个愚蠢的包装。

7. auth_interactive_dumb() （交互、自动化）

   以交互方式对服务器进行自动化，但是垃圾。

8. auth_none() （身份验证）

   尝试使用任何身份验证向服务器进行身份验证。 这几乎总是会失败。

9. auth_password () （密码、身份验证）

   使用密码对服务器进行身份验证。 用户名和密码通过加密链接发送。

10. auth_publickey() （私钥验证）

    使用私钥对服务器进行身份验证。 密钥用于从服务器签署数据，因此它必须包括私有部分。

11. cancel_port_forward() （取消、转发请求）

    请求服务器取消先前的端口转发请求。

12. Close() （关闭）

    关闭此会话以及任何与之相关的公开渠道。

13. Connect () （连接）

    连接服务器。协商 SSH2 会话，并可选择验证服务器的主机密钥，并使用密码或私钥进行身份验证。 这是 start_client ， get_remote_server_key 和 Transport.auth_password 或 Transport.auth_publickey 的快捷方式。

14. get_banner() （返回）

    连接后返回服务器提供的横幅。 如果没有提供横幅，则此方法返回 None 。

15. get_exception() （返回、异常）

    返回上次服务器请求期间发生的任何异常。

16. get_log_channel() （返回、通道名称）

    返回用于此运输记录的通道名称。

17. get_remote_server_key() （返回、主机密钥）

    返回服务器的主机密钥（在客户端模式下）。

18. get_server_key() （返回、主机密钥）

    在服务器模式下返回活动主机密钥。 在与客户端协商后，该方法将返回协商的主机密钥。

19. get_username() （返回、用户名）

    返回用于此连接的用户名。

20. Getpeername() （远程端地址）

    如果可能，返回此 Transport 的远程端的地址。 这实际上是底层套接字上'getpeername'的包装。

21. global_request() （发出请求）

    向远程主机发出全局请求。

22. is_active() （处于活动状态）

    如果此会话处于活动状态（打开），则返回 true。

23. is_authenticated()　　（处于活动状态）

如果此会话处于活动状态并通过身份验证，则返回 true。

24. static load_server_moduli()　　（加载）

加载主模块的文件，用于在服务器模式下进行组交换密钥协商。

25. open_channel()　　（请求新的 Channel）

向服务器请求新的 Channel。

26. open_forward_agent_channel()　　（请求新的 Channel）

向客户请求新的频道，类型为"auth-agent@openssh.com" 。这只是 open_channel('auth-agent@openssh.com')的别名。

27. open_forwarded_tcpip_channel()　　（请求新的 Channel）

请求一个新的频道回到客户端，类型为"forwarded-tcpip" 。 这是在客户端请求端口转发后使用的，用于将传入的连接发送回客户端。

28. open_sftp_client()　　（打开、创建、通道）

从打开的传输中创建 SFTP 客户端通道。 成功后，将与远程主机一起打开 SFTP 会话，并返回一个新的 SFTPClient 对象。

29. renegotiate_keys()　　（强制、切换）

强制此会话切换到新密钥。 通常这是在会话发送或接收到一定数量的数据包或字节后自动完成的，但是这种方法可以让您选择强制新的密钥。 双方交换密钥并进行计算，协商新密钥会导致流量暂停。 当会话切换到新密钥时，此方法返回。

30. request_port_forward()　　（请求、转发 TCP 连接）

请求服务器通过此 SSH 会话从服务器上的监听端口转发 TCP 连接。

31. set_keepalive()　　（打开、关闭）

打开/关闭 keepalive 数据包（默认为关闭）。 如果这样设置，在 interval 秒之后，不通过连接发送任何数据，“keepalive”数据包将被发送（并被远程主机忽略）。 例如，这可以有助于通过 NAT 来保持连接的生效。

32. set_log_channel()　　（设置、通道）

设置此运输记录的通道。 默认值为"paramiko.transport"但可以设置为任何您想要的。（有关详细信息，请参阅 logging 模块。）SSH 通道将记录指定的子通道。

33. set_subsystem_handler()　　（设置）

在服务器模式下设置子系统的处理程序类。

34. start_server ()　　（设置、服务器）

设置新的 SSH2 会话为服务器。 这是创建新的 Transport 和设置服务器主机密钥后的第一步。 创建一个单独的线程用于协议协商。

35. use_compression()　　（压缩）

打开/关闭压缩。 这只会在启动传输之前（例如在调用 connect 之前）等方面产生影响。 默认情况下，压缩已关闭，因为它会影响交互式会话。

f) SFTPClient

1. chdir()

更改此 SFTP 会话的“当前目录”。

2. chmod()

   更改文件的模式（权限）。

3. chown()

   更改文件的所有者（ uid ）和组（ gid ）。

4. close()

   关闭 SFTP 会话及其基础通道。

5. file()

   在远程服务器上打开一个文件。

6. classmethod from_transport()

   从打开的 Transport 创建 SFTP 客户端通道。

7. get()

   将远程文件（远程 remotepath ）从 SFTP 服务器复制到本地主机作为本地 localpath 。

8. get_channel()

   返回此 SFTP 会话的基础 Channel 对象。

9. getcwd()

   由 Paramiko 仿真，返回此 SFTP 会话的“当前工作目录”。 如果没有使用 chdir 设置 chdir ，则此方法将返回 None 。

10. getfo()

    从 SFTP 服务器复制一个远程文件（远程路径），并写入一个打开的文件或类似文件的对象。

11. listdir()

    返回一个包含给定 path 条目名称的列表。

12. listdir_attr()

    返回包含与给定 path 中的文件相对应的 SFTPAttributes 对象的列表。 列表是按任意顺序排列的。

13. listdir_iter()

    listdir_attr 的生成器版本。

14. lstat()

    检索有关远程系统上的文件的信息，而不需要遵循符号链接（快捷方式）。

15. mkdir()

    以数字模式 mode 创建名为 path 的文件夹（目录）。 默认模式是 0777（八进制）。

16. normalize()

    返回给定路径的标准化路径（在服务器上）。

17. open()

    在远程服务器上打开一个文件。

18. put()

    将本地文件（本地路径）复制到 SFTP 服务器作为 remotepath 。

19. putfo()

将打开的文件对象的内容复制到 SFTP 服务器。

20. readlink()

返回符号链接的目标（快捷方式）。 您可以使用 symlink 创建这些。 结果可能是绝对路径名或相对路径名。

21. remove()

删除给定路径上的文件。 这只适用于文件；要删除文件夹（目录），请使用 rmdir 。

22. rename()

将文件或文件夹从 oldpath 重命名为 newpath 。

23. rmdir()

删除名为 path 的文件夹。

24. stat()

检索有关远程系统上文件的信息。 返回值是一个对象，其属性对应于 os.stat 返回的 Python 的 stat 结构的 stat ，除了它包含较少的字段。

25. truncate()

更改由 path 的文件的大小。

26. unlink()

删除给定路径上的文件。 这只适用于文件; 要删除文件夹（目录），请使用 rmdir 。

27. utime()

设置由 path 的文件的访问和修改时间。

g) SFTPServer

1. static convert_errno()

将 errno 值转换为标准 SFTP 结果代码。 这是一个方便的功能，用于捕获服务器代码中的异常并返回适当的结果。

2. static set_file_attr()

更改本地文件系统上的文件属性。