

Monitoring and Debugging Live Applications

Robert Coup

Koordinates

robert@coup.net.nz

<http://rob.coup.net.nz/>

Abstract

Ever wondered what is going on inside your app? Learn some tools and ideas you can use for sussing it out.

Debugging issues in live applications can be a real nightmare. Add some more logging, restart it, wait for the failure/load case and try and deduce a bit more. Even with awesome test coverage you still need to debug those tricky problems. We introduce some ideas for monitoring and debugging your live Python applications.

- the standard logging module and learning how to drive it.
- how to set up a remote console session into our running application, so we can interrogate variables and run code to figure out what's going on.
- how to incorporate IM bots so we can interact with our apps from the desktop, and how they can interact with us.

1. Introduction

When we encounter a problem with our live applications (on the web or elsewhere), we often have no opportunity to follow the normal debugging process through to its conclusion. The app needs to get back up, stat! And trying to piece together logs afterwards is hard.

2. Logging

I believe in logging virtually everything - if it helps during development it'll probably help during debugging. Certainly beats "turning up" or adding logging, restarting the app, and waiting for the problem to happen again.

The most common logging pattern is:

```
print "name=", name
```

For those who get sick of the logging interfering with the program output, the usual next step is to output the log messages to standard error:

```
print >>sys.stderr, "name=", name
```

The problem with the above methods is that it becomes difficult to adjust the detail of logging, and we end up commenting out the logging lines.

Using Python's standard logging library, we can do the same thing via:

```
log.debug("name=%s", name)
```

2.1. Python's logging module

The key advantage of the builtin logging module is the separation it provides between where the log messages are generated (throughout the code) and where the log output goes to. And this is all configurable at runtime.

This separation means we can change the detail of the logging and where it ends up, without touching the code that creates log messages.

2.2. Logger

The first key component of the logging module is the Logger object, which is responsible for generating log messages. Loggers are named by the developer, usually based on where the code is located (eg. mypackage.mymodule.myobject). The names form a hierarchy using the dots, which can be filtered on later.

The key methods on a Logger object create log messages at different levels:

```
debug(), info(), warn(), error(), critical(), exception()
```

All the logging methods accept % formatting via their arguments. In addition, the `exception()` method will automatically pick up on the current exception and log a traceback.

2.3. Handler

Handlers do something with log messages. They might write it to the console, to a file, send to a web server, post an email, write it to the syslog or the NT event log, or a multitude of other things. Handlers can filter out which messages they apply to and which level messages they will act on, so we can log everything to a file as well as email just the critical errors.

2.4. Formatter

The final component of the logging module is the formatter. It describes how log messages are going to look like for specific handlers. The formatter has access to the code location where the log message came from (file, module, line, method), as well as process and thread information.

2.5. Logging Example

Lets show an example of configuring and running a logger for our example “machine” application:

- log everything to /tmp/machine.log
- log startup and shutdown events to syslog
- quieten some noisy libraries (log warnings & errors only)
- email errors in machine.core to admin@example.com

The code example for doing the logging:

```
#!/usr/bin/env python

import logging, logging.config

# configure using our ini file
logging.config.fileConfig("l.ini")

# this will be logged to syslog
L = logging.getLogger("machine.startstop.start")
L.info("Starting the machine...")

# normal log messages get logged to the file
L = logging.getLogger("machine.wangdoodle")
L.info("work work work")

L = logging.getLogger("muppets")
L.info("bork bork bork")

# any errors under machine.core will be emailed too
L = logging.getLogger("machine.core.reactor")
L.error("Meltdown in Sector 7-G!")

# noisy code we suppress
L = logging.getLogger("chatty")
L.debug("rhubarb rhubarb rhubarb")

# this will be logged to syslog
L = logging.getLogger("machine.startstop.stop")
L.info("Stopping the machine...")
```

And we'll use the ini-file style configuration of the logging module:

```
[loggers]
keys=root,noisylibs,startstop,core
[handlers]
keys=file,syslog,email,stderr
[formatters]
keys=normal,email

# our normal formatter
[formatter_normal]
format=%(asctime)s machine[%%(process)d]: %(name)s %(module)s(%
(lineno)d): %(message)s

# format emails into >1 line...
[formatter_email]
format=Level: %(levelname)s
Created: %(asctime)s
Logger: %(name)s
Module: %(module)s (line %(lineno)d)
Process: %(process)d (thread %(thread)d %(threadName)s)
%(message)s
```

```

# root logger, send everything to the file log
[logger_root]
level=NOTSET
handlers=file

# ignore messages below WARN for noisy libraries
[logger_noisylibs]
level=WARN
qualname=noisy,chatty,loud
propagate=0
handlers=

# send machine.core errors to the email handler
[logger_core]
qualname=machine.core
level=ERROR
handlers=email

# send startup/shutdown notices to syslog
[logger_startstop]
qualname=machine.startstop
handlers=syslog

# log to syslog
[handler_syslog]
class=handlers.SysLogHandler
formatter=normal
args=('/dev/log',)

# log to a file
[handler_file]
class=FileHandler
formatter=normal
args=('/tmp/machine.log','a')

# log to email
[handler_email]
class=handlers.SMTPHandler
formatter=email
args=('localhost', 'logbots@example.com', ['admin@example.com'],
'LOG NOTICE')

```

2.6. Other Logging Tools

Once we are producing piles of useful logs for our applications, it's worth introducing some further tools to help us manage the logs.

Tools:

- **syslog-ng** (<http://www.balabit.com/network-security/syslog-ng/>) pulls logs from multiple machines into one coherent structure.
- **ack** (<http://betterthangrep.com/>) is a fast and flexible grep-like tool which is excellent for searching logs.
- **less** (<http://www.greenwoodsoftware.com/less/>) is a powerful pager to help navigate and search log files.
- **Splunk** (<http://www.splunk.com/>) allows merging and matching numerous logs – tracing events across multiple services and servers and producing detailed metrics.

3. Remote Consoles

The next thing in our debugging arsenal is a remote console. Don't you wish you could just look inside your app and find out what it's thinking, rather than hypothesising? Remote consoles certainly beats trial and error, or reconstructing chains of events from log files. We have the technology to just ask our apps what they're doing!

3.1. Twisted Manhole

Twisted (<http://www.twistedmatrix.com>) is an asynchronous programming library that has great support for numerous network protocols. Manhole is a component that provides an SSH or Telnet server into a running Python environment. Once connected, you can read and change variables, call methods, and interact with your running application.

```
#!/usr/bin/env python

"""
Example of a twisted Manhole server
Connect via Telnet to localhost:4777, login as admin:admin
"""

from twisted.application import service, app
from twisted.internet import reactor, task
from twisted.conch import manhole_tap

class ManholeServer(object):
    def __init__(self, application, namespace):
        self.manhole = manhole_tap.makeService({
            'namespace': namespace,
            'telnetPort' : 'tcp:4777:interface=127.0.0.1',
            'sshPort' : None,
            # path to passwd-style file (1 line per entry,
            # username:password in plaintext)
            'passwd' : 'passwd',
        })
        self.manhole.setServiceParent(application)

class Adder(object):
    def __init__(self, application):
        self.value = 0
        task.LoopingCall(self.callback).start(1.0, False)

    def callback(self):
        self.value += 1
        print self.value

if __name__ == "__main__":
    application = service.Application('Example Two')

    adder = Adder(application)
    console = ManholeServer(application, {'adder': adder})

    # run it all
    print "Starting reactor..."
    app.startApplication(application, False)
    reactor.run()
```

The above example shows how simple it is to add a console into your application. But what if you're not running a Twisted application (ie. most of the time). We can instead run Twisted (and our console) in a second thread, leaving our existing code

to work as normal. The only downside is that we need to handle concurrency if we're doing any writes into the application data.

```
#!/usr/bin/env python

"""
Example of a twisted Manhole server running as a thread in
a normal python program.

Connect via Telnet to localhost:4777, login as admin:admin
"""

import threading

from twisted.application import service, app
from twisted.internet import reactor
from twisted.conch import manhole_tap

class ThreadConsole(threading.Thread):
    def __init__(self, namespace):
        super(ThreadConsole, self).__init__()
        self.setDaemon(True)
        self.application =
service.Application('TwistedThreadConsole')

        self.manhole = manhole_tap.makeService({
            'namespace': namespace,
            'telnetPort' : 'tcp:4777:interface=127.0.0.1',
            'sshPort' : None,
            # path to passwd-style file (1 line per entry,
username:password in plaintext)
            'passwd' : 'passwd',
        })
        self.manhole.setServiceParent(self.application)

    def run(self):
        app.startApplication(self.application, False)
        reactor.run(installSignalHandlers=0)

class PrimeFinder(object):
    # THE dumbest prime-finding algorithm
    def __init__(self):
        self.c = 2
        self.primes = []

    def run(self):
        while True:
            for i in xrange(2, self.c):
                if self.c % i == 0:
                    break
            else:
                self.primes.append(self.c)
                print self.c
                self.c += 1

if __name__ == "__main__":
    primeFinder = PrimeFinder()
    console = ThreadConsole(globals()).start()
    primeFinder.run()
```

4. Bots

The third idea i want to talk about is robots. We can review our application's progress with logs. We can telnet right inside it for hardcore inspection. But we can also get it talking to us via instant messaging.

Everybody runs IM of some sort, it is already open on the desktop, and it just works. With some IM integration, our applications can say to us, in real time: "hey, Rob, I'm not happy". Plus, the marketing guy down the hall might like to get pinged if someone orders 100 copies of our awesome software. Bots can do this.

Among its arsenal of networking awesomeness, twisted also does XMPP (aka. Jabber). It'll also do IRC and some other stuff, but we'll focus on XMPP for now. Basically XMPP is an asynchronous protocol for passing bits of XML around, with no polling required. To make building XMPP clients even easier, use the Wokkel library (<http://wokkel.ik.nu/>):

```
#!/usr/bin/env python

"""
A simple example of an IM bot using
the Twisted Words and wokkel libraries.
"""
from wokkel import client, xmppim
from twisted.application import service, app
from twisted.internet import reactor
from twisted.words.protocols.jabber.jid import JID
from twisted.words.xish import domish

class TalkProtocol(xmppim.MessageProtocol):
    def onMessage(self, message):
        # we've received a message!
        reply = "nah, you're a " + str(message.body)

        response = domish.Element((None, "message"))
        response["to"] = message['from']
        response.addElement((None, 'body'), content=reply)
        self.send(response)

if __name__ == "__main__":
    application = service.Application('TalkBot')

    client = client.XMPPClient(JID("talkbot@jabber.org"), "talkbot")
    client.logTraffic = True
    client.setServiceParent(application)

    presence = xmppim.PresenceClientProtocol()
    presence.setHandlerParent(client)
    presence.available()

    talkProtocol = TalkProtocol()
    talkProtocol.setHandlerParent(client)

    app.startApplication(application, False)
    reactor.run()
```

As shown above, the actual code which receives messages and composes replies is relatively short and simple. Calling `talkProtocol.send()` at any point will dispatch a message, and Jabber will take care of routing the request to all the users specified.

5. Summary

Arm yourself with some great tools and debugging and keeping an eye on your apps will be a lot easier. Trial and error sucks, move on!

- Use logging
- Talk to your apps (console, bots)
- Have your apps talk to you (bots)
- You don't need to have a Twisted app!