Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Testing Python and C# Code

Monkey patching and reflection are just a couple of ways to test complex systems.

February 06, 2013
URL:http://www.drdobbs.com/testing/testing-python-and-c-code/240147927

In part one of this three-part series on testing complex systems, I covered the practical aspects of deep testing and demonstrated how to test difficult areas of code, such as the user interface, networking, and asynchronous code. In part two, I discussed some techniques and utilities I have used to successfully test complex systems in C++.

In this final installment, I discuss similar complex testing for Python (with Swig C++/Python bindings) and high-volume, highly available Web services in C# and .NET.

Python

Python is strongly typed *and* dynamically typed. It is also an interpreted language (compiled just-in-time). This means that type errors are discovered at runtime, rather than compile time. For example, if you pass an `int` to a function that expects a string, your program will still run and maybe even run successfully to completion if your code never reaches the bad call:

```
def sayHello(name):
    print 'Hello, ' + name

def sayHelloToMyLittleFriend():
    sayHello(5)

sayHello('Jebediah')
```

Output:

```
Hello, Jebediah
```

The `sayHelloToMyLittleFriend()` function containing the bad call to `sayHello(5)` is never invoked, so the program runs fine. So, should you do something different than when writing programs in a statically typed language? No! If you didn't discover the bad call, it means you didn't test the code path.

If you don't test a code path, you can't tell whether it's broken. And it really doesn't matter whether it's broken due to type errors or some other kinds of errors. (By the way, in statically typed languages, there is a very similar problem that crops up a lot. The dreaded `NULL/null/nill/None` or whatever you want to call it. If you pass a `NULL` pointer to a C++ function that tried to dereference it, you will cause a crash: This is actually much more common than passing an `int` to a function expecting a string.)

With Python, if you are really paranoid, you can test the type of arguments. This may be useful in libraries that are used a lot and where you want to provide a good error message to the user instead of just crashing with a type error. You can either wrap the suspicious code in a `try-except` block and catch `TypeError`, or you can use the `instanceof()` function to dynamically determine the type of arguments (but this second option is pretty brittle). Here is how it looks with `try-except`:

```
def sayHello(name):
    try:
        print 'Hello, ' + name
    except TypeError:
        print 'Oh, no!', name, 'is not a string'

def sayHelloToMyLittleFriend():
    sayHello(5)

sayHelloToMyLittleFriend()
```

Output:

```
Oh, no! 5 is not a string
```

Monkey Patching

Testing Python code (and other dynamic languages) is a pleasure. There is no need to design testability into your code or create special interfaces or fight with legacy code. With Python, you can just reach into the guts of any object and replace its members with whatever you like. You can even replace functions. It doesn't get any better than that. This time honored practice is called monkey patching. Monkey patching works even if you don't use dependency injection because you can replace the dependencies at any time. In the following code example, class `A` instantiates its own `Dependency`:

```
class Dependency(object):
    def __init__(self):
        self._state = 'complex state'


    def bar(self):
        print 'Doing something expansive here...'
        print 'state: ' , self._state

class A(object):
    def __init__(self):
        self._b = Dependency()

    def foo(self):
        self._b.bar()

a = A()
a.foo()
```

Output:

```
Doing something expansive here... state: complex state
```

If you want to test `A`, but you don't want it to invoke the real expansive `Dependency.bar()` method you have two options:

1. You can replace the `_b` member of `A` with a mock object that has a `bar()` method
2. You can replace the `bar()` method of the `Dependency`

Method 1 is useful if you want to mock the entire dependency, often with multiple methods and internal state. Method 2 is useful if you want to use the real `Dependency` object with its logic and initialized state, but just replace a few methods.

Method 1:

```
class MockDependency(object):
    def __init__(self):
        """No state"""

    def bar(self):
        print 'Doing something cheap here...'

a = A()
a._b = MockDependency()
a.foo()
```

Output:

```
Doing something cheap here...
```

Method 2:

```
def mock_bar(self):
    print 'Doing something cheap here...'
    print 'state: ' , self._state

a = A()
Dependency.bar = mock_bar
a.foo()
```

Output:

```
Doing something cheap here... state: complex state
```

You'd be wise to restrict your usage of monkey patching to testing and other special circumstances. A lot of people get excited about the power of monkey patching and use it willy-nilly all over the place, which leads to buggy and unmaintainable code.

Python Test Frameworks

Python has multiple test frameworks. The xUnit-like unittest package has been part of the Python standard library since Python 2.1, but it was never as powerful and user-friendly and some other test frameworks until Python 2.7. I like nose a lot. With nose, you don't need to write test classes derived from a base class, but can instead write simple test functions and nose will discover and run them. In Python 2.7, the standard unittest package acquired many features available in third-party test frameworks and became a much more formidable contender (although nose has evolved, too, and there is nose2).

There is a lot of material available around the Web on unittest, so I'll just quickly list the main features of the Python 2.7 (and Python 3.2) unittest package:

- Abundance of `assert` functions
- Test discovery
- Test organization
- Reporting
- Command-line interface
- Fine-grained control on running tests

Python also has an interesting module called doctest, which allows you to write tests embedded in your docstrings. I don't use doctest because I like to separate the code under test from the code that tests it. I also aim for more comprehensive testing, which is difficult to do in a doc string (not to mention that it will clutter the code).

Python Extensions

Like any self-respecting dynamic language, Python (CPython) has an extension and embedding mechanism. That means you can extend Python with native C libraries and even embed Python in a native program. Other Python implemenations that target specific runtime environments, like Jython on the JVM and IronPython on .NET, provide similar facilities. Here, I'll discuss only CPython.

First things first — Python is slow. It is, of course, slow compared with native languages, but it is also slow compared with other dynamic languages. I will not delve into the particulars (super-flexible, super-dynamic, GIL, multithreading issues). There have been various efforts over the years to improve Python's performance. The most notable is the PyPy project. In the CPython ecosystem, the recommended way to extend Python with native code is to use Cython, which is a Python-like language that lets you write extensions to Python mostly in Python. If you need to interface with existing C/C++ code, there are several other tools available such as Boost.Python and Py++.

Dealing with Language Bindings

My language binding experience mostly involves using Swig. Swig is a veteran project that can create C/C++ bindings for almost any programming language. I have used it to create language bindings for the NuPIC 2.0 API for Python, Java, and C#. I have also had to debug through its interface code for NuPIC 1.0 Python bindings. Testing such code is no fun. It is all about marshaling data types, matching calling conventions, and adapting error-handling mechanisms. The best advise I can give here is to keep the binding code as lean as possible. Don't give in to the temptation to add syntactic sugar or try to make the binding code follow the idioms of the target language. Use a layered approach — the bindings themselves, whether hand coded or autogenerated, should closely match the native code. Later, once you are back in your target language world, write a wrapper or syntactic sugar layer on top of it. You can write your tests using the wrapper layer.

Testing C#/.NET Code

I have followed the evolution of the .NET framework and the C# language since the original beta 2. I wrote many programs at home and published several articles on the framework and language, but until recently, I never had the the chance to use them in a professional capacity. These days, I work for Roblox, where I write highly scalable and highly distributed services using C# and the .NET framework, and I enjoy every minute. C# is a wonderful programming language. It started out as a better Java, and in each revision, it has added more and more good stuff (generics, anonymous functions, attributes, LINQ, async support) with great balance. For some reason (great language designers?), the power and richness of C# never gave me a feeling of bloat like other languages do. When you throw in the .NET framework and Visual Studio, you get a superb development and runtime environment.

Visual Studio provides a good unit test framework (Microsoft.VisualStudio.TestTools.UnitTesting), which you can configure with attributes. It allows you to run any subset of tests in your solution. My go-to technique is to place the mouse cursor on a test I want to run and then click CTRL+R+T to launch just that test. In the past, I used to write little console programs to exercise different parts of the code. Not anymore.

While there are also many other great test frameworks (both commercial and free), I will use the built-in Visual Studio support in the following examples, but the concepts are not related to any particular test framework.

Reflection

The .NET framework keeps excellent metadata for every object and allows reflection. You can take any object even without knowing its type (all objects are derived from the `object` class) and, at runtime, discoverits fields, properties, methods, etc. You can also access fields and properties, and invoke methods. The most important thing about reflection is that it allows you to ignore access modifiers. You can access private state and invoke private methods, access internal types, as well as instantiate private classes and nested classes. The following example shows a `class A` with a private data member `_x` and a private method `DoubleIt()`:

```
class A
        {
                private int _x;

                public A(int x = 5)
                {
                        _x = x;
                }

                private void DoubleIt()
                {
                        _x *= 2;
                }
        }
```

`Class A` happens to be a nested class. This is important because, when accessing nested classes, you qualify their name with `+`. If you are not sure what is the exact type name of the class you want to test, you can use the `GetTypes()` of the assembly to list all the types it contains. Here is a unit test for `class A` using raw reflection:

```
[TestMethod]
        public void RawReflectionTest()
        {
                var assembly = GetType().Assembly;

                // Get all the types in the assembly
                var types = assembly.GetTypes();

                var typeName = "ReflectionDemo.ReflectionTest+A";
                var t = assembly.GetType(typeName);
                var instance = Activator.CreateInstance(t, 4);

                // Get the value of X
```

```
                var filed = t.GetField("_x", BindingFlags.NonPublic | BindingFlags.Instance);
                var value = filed.GetValue(instance);
                Assert.AreEqual(4, value);

                // Invoke DoubleIt
                var method = t.GetMethod("DoubleIt", BindingFlags.NonPublic | BindingFlags.Instance);
                method.Invoke(instance, new object[] {});

                // Verify result
                value = filed.GetValue(instance);
                Assert.AreEqual(8, value);
        }
```

Obviously, it is pretty cumbersome, so I created a little utility class called ReflectedObject that makes it a little more straightforward. Here is the same test using `ReflectedObject`:

```
[TestMethod]
        public void ReflectedObjectTest()
        {
                var assembly = GetType().Assembly;
                var typeName = "ReflectionDemo.ReflectionTest+A";
                var instance = new ReflectedObject(assembly, typeName, 4);

                var value = instance.GetField("_x");
                Assert.AreEqual(4, value);

                instance.InvokeMethod("DoubleIt");

                value = instance.GetField("_x");
                Assert.AreEqual(8, value);
    }
```

If you're using .NET 4.x, it is possible to go even further using the `DynamicObject` class, then writing a subclass that allows you to directly access private fields, properties, and methods.

I recently worked on a effort to integrate RabbitMQ into Roblox's technology stack. RabbitMQ is an awesome message queue with built-in clustering support. The RabbitMQ C# Client provides a full-featured API to talk to the RabbitMQ server. Unfortunately, it is not cluster-aware. If you want to talk to a cluster and handle situations like nodes going up and down, temporary disconnects, and recovery, you need to write a lot of extra code. I did just that and I wanted to test it.

Because RabbitMQ was a new third-party piece of software to be used as a critical component of our system, I wanted to test its integration throughly. That involved multiple tests against a local cluster of three nodes (all running on my local machine), as well as the same tests running against a remote RabbitMQ cluster. The tests involved tearing down, recreating, and configuring the cluster in different ways, and then stress-testing it. Setting up and configuring a remote RabbitMQ cluster involves multiple steps, each normally taking less than a second. But, on occasion, one can take up to 30 seconds. Here is a typical list of the necessary steps for configuring a remote RabbitMQ cluster:

- Shut down every node in the cluster
- Reset the persistent metadata of every node
- Launch every node in isolated mode
- Cluster the nodes together
- Start the application on each node
- Configure virtual hosts, exchanges, queues, and bindings

I created a Python program called Elmer that uses Fabric to remotely interact with the cluster. Due to the way RabbitMQ manages metadata across the cluster, you have to wait for each step to complete for every node in the cluster before you can execute the next step; and checking the result of each step requires parsing the console output of shell commands (yuck!). Couple that with node-specific issues and network hiccups and you get a process with high time variation. In my tests, in addition to graceful shutdown and restart of the whole cluster, I often want to violently kill or restart a node.

From an operations point of view, this is not a problem. Launching a cluster, or replacing a node, are rare events and it's OK if it takes a few seconds. It is quite a different story for a developer who want to run a few dozen cluster tests after each change. Another complication is that some use cases require testing unresponsive nodes, which can lead to the halting problem (is it truly unresponsive or just slow?). After suffering through multiple test runs where each test was blocked for a long time waiting for the remote cluster, I ended up with the following approach:

1. Elmer (the Python/Fabric cluster remote control program) exposes every step of the process
2. A C# class called `Runner` can launch Python scripts and Fabric commands and capture the output
3. A C# class called `RabbitMQ` utilizes the `Runner` class to control the cluster
4. A C# class called `Wait` can dynamically wait for an arbitrary operation to complete

The key was the `Wait` class. The `Wait` class has a static method called `Wait.For()` that allows you to wait for an arbitrary operation to complete until a certain timeout. If the operation completes quickly, you will not have to wait for the time to expire, and `Wait` will bail out quickly. If the operation doesn't complete in time, `Wait.For()` will return after the timeout expires. `Wait.For()` accepts a duration (either a `TimeSpan` or number of milliseconds), and a function returns `bool`. It also has a `Nap` member variable that defaults to 50 milliseconds. When you call `Wait.For()`, it calls your function in a loop until it returns `true` or until the duration expires (napping between calls). If the function returns `true`, then `Wait.For()` returns `true`; but if the duration expires, it returns `false`.

```
public class Wait
    {
        public static TimeSpan Nap = TimeSpan.FromMilliseconds(50);
        public static bool For(TimeSpan duration, Func<bool> func)
        {
```

```
            var end = DateTime.Now + duration;
            if (end <= DateTime.Now)
            {
                return false;
            }
            while (DateTime.Now < end)
            {
                if (func.Invoke())
                {
                    return true;
                }

                Thread.Sleep(Nap);
            }
            return false;
        }

        public static bool For(int duration, Func<bool> func)
        {
            return For(TimeSpan.FromMilliseconds(duration), func);
        }
    }
```

Now, you can efficiently wait for processes that may take highly variable times to complete. Here is how I use `Wait.For()` to check whether a RabbitMQ node is stopped:

```
private bool IsRabbitStopped()
        {
            var ok = Wait.For(TimeSpan.FromSeconds(10), () =>
            {
                var s = rmq("status", displayOutput: false);
                return !s.Contains("{mnesia,") && !s.Contains("{rabbit,");
            });

            return ok;
        }
```

I call `Wait.For()` with a duration of 10 seconds, which I wouldn't want to block on every time I check whether a node is down (since it happens all the time). The anonymous function I pass in calls the `rmq()` method with the `status` command. The `rmq()` method runs the `status` command on the remote cluster, then returns the command-line output as text. Here is the output when the Rabbit is running:

```
Status of node rabbit@GIGI ...
[{pid,8420},
 {running_applications,
     [{rabbitmq_management,"RabbitMQ Management Console","2.8.2"},
      {xmerl,"XML parser","1.3"},
      {rabbitmq_management_agent,"RabbitMQ Management Agent","2.8.2"},
      {amqp_client,"RabbitMQ AMQP Client","2.8.2"},
      {rabbit,"RabbitMQ","2.8.2"},
      {os_mon,"CPO  CXC 138 46","2.2.8"},
      {sasl,"SASL  CXC 138 11","2.2"},
      {rabbitmq_mochiweb,"RabbitMQ Mochiweb Embedding","2.8.2"},
      {webmachine,"webmachine","1.7.0-rmq2.8.2-hg"},
      {mochiweb,"MochiMedia Web Server","1.3-rmq2.8.2-git"},
      {inets,"INETS  CXC 138 49","5.8"},
      {mnesia,"MNESIA  CXC 138 12","4.6"},
      {stdlib,"ERTS  CXC 138 10","1.18"},
      {kernel,"ERTS  CXC 138 10","2.15"}]},
 {os,{win32,nt}},
 {erlang_version,"Erlang R15B (erts-5.9) [smp:8:8] [async-threads:30]\n"},
 {memory,
     [{total,19703792},
      {processes,6181847},
      {processes_used,6181832},
      {system,13521945},
      {atom,495069},
      {atom_used,485064},
      {binary,81216},
      {code,9611946},
      {ets,628852}]},
 {vm_memory_high_watermark,0.10147532588839969},
 {vm_memory_limit,858993459},
 {disk_free_limit,8465047552},
 {disk_free,15061905408},
 {file_descriptors,
     [{total_limit,924},{total_used,4},{sockets_limit,829},{sockets_used,2}]},
 {processes,[{limit,1048576},{used,181}]},
 {run_queue,0},
 {uptime,62072}]
...done.
```

The function is making sure that the `mnesia` and `rabbit` components don't show up in the output. Note that if the node is still up, the function will return `false` and `Wait.For()` will continue to execute it multiple times. `Wait.For()` decreases the sensitivity of my tests to occasional spikes in response time (I can `Wait.For()` longer without slowing down the test in the common case), and has reduced the runtime of the whole test suite from minutes to seconds.

Conclusion

The sum total of this series of articles has shown a variety of design principles and testing techniques to deal with hard-to-test systems. Nontrivial code will

always contain bugs, but deep testing is guarantied to reduce the number of undiscovered issues.

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems, and is a long-time contributor to Dr. Dobb's.

Related Articles

Testing Complex Systems

Testing Complex C++ Systems