谈谈单元测试中的测试桩实践

写程序的时候，有时可能遇到这样的情况。比如我们开发了一个数据处理模块，被处理的数据需要调用其他模块(由其他团队开发，或者是第三方提供，总之测试的责任不在你)，从数据库或者文件或者通过网络从其他进程中获取。为了对该数据处理模块进行单元测试，我们通常得相应的配置起一个数据库或者文件系统，甚至是相关进程，以求正常的得到数据，但这样做的代价往往较大。

这里想讨论一种我以前曾经使用过的简化单元测试的思路。通过接口来封装对外部模块的调用，在单元测试时，用调试实现代替外部实现。**受 myworkfirst 指点，又 google 了一下，才知道这是单元测试里早已成熟的“测试桩”**。但我仍然想把我的实践和大家分享一下。

我们用一个简单的例子来说明。比如我实现了一个 SystemTimeSynchronizer 类，周期性的查询 NTP 标准时间，和本地系统时间进行比较。

```
/**shannon.demo is the package for the demonstration, in which,
 * there's all the codes of unit test target module.
 */
package shannon.demo;

import thirdparty.any.NtpClock;

/**
 * <code>SystemTimeSynchronizer</code> is our unit test target,
 * which acts as if calibrating the system time firmly in
 * compliance with the standard time.
 * @author Shannon Qian
 */
public class SystemTimeSynchronizer {
    /**Compares the local system time with the standard time.
     * @return - 1 if system time is ahead of standard time,
     * 0 if it's on standard time and -1 if it's behind standard
     * time.
     */
    public int syncTime() {
        long currentTime = new NtpClock().getTime();
        long interval = System.currentTimeMillis()-currentTime;
        if(interval == 0) {
            return 0;
        } else if(interval > 0) {
            return 1;
        } else {
            return -1;
        }
```

```
    }
}
```

SystemTimeSynchronizer#syncTime() 调用的 NtpClock 类，属于外部模块。NtpClock#getTime()在这里只是一个示意，说明在没有预设 NTP 服务器的情况下，它将抛出异常(这和我们在单元测试时实际遇到的情况类似)。但是请你想象其内部实现通过访问预设的 NTP 服务器获取标准时间。要让 NtpClock 类正常的运行起来，需要一个 NTP 服务器，并事先进行比较复杂的设置。

```
/**package thirdparty.any plays the role to contain all the codes
 * as if from thrid party.
 */
package thirdparty.any;

/**
 * <code>NtpClock</code> is a demenstrating class for this unit test
firewall
 * example. it acts as if a third-party-provided adaptor with access
to the
 * NTP server.
 * @author Shannon Qian
 */
public class NtpClock {

    /**Returns the standard time from NTP server.
     * @return - the standard time from NTP server
     * @throws IllegalStateException - if there's no NTP server
available.
     */
    public long getTime() {
        //if there's no NTP server available.
        throw new IllegalStateException("NTP server is not ready.");
    }
}
```

在不配置 NTP 服务器的情况下，单元测试肯定会因为异常抛出而中断。为了避免麻烦，我们首先想到的是如果不调用 NtpClock 就好了。但如果不调用，就无法获取标准时间。这样我们只能另外造一个类，在单元测试时替代 NtpClock，能够方便的提供标准时间。新的问题是 SystemTimeSynchronizer 需要知道在不同时机调用不同的对象－在单元测试时，调用我们自定义的类，而在正常运行时仍然调用 NtpClock.

首先定义一个 Clock 接口。并为 Clock 实现两个具体类，一个是 NtpClockWrapper，顾名思义其实就是实现了 Clock 的 NtpClock，另一个是 SystemClock，它就提供系统当前时间作为标准时间。

```
package shannon.demo;

/**
 * <code>Clock</code> is an interface for all the clock to provide
time.
 * @author Shannon Qian
 */
public interface Clock {
    /**Returns the time in millusecond.
     * @return - the time in millusecond
     */
    public long getTime();
}
```

同时，我们定义一个 UnitTestFirewall 类，维护一个 debugging 标记。并提供一个 getClock() 类工厂方法，返回 Clock 对象。

```
package shannon.demo;

import thirdparty.any.NtpClock;

/**
 * <code>UnitTestFirewall</code> is the facility to
 * ease unit test
 * @author Shannon Qian
 */
public final class UnitTestFirewall {
    private static boolean debugging=false;

    /**Returns true if it's in debugging mode, else false.
     * @return the debugging
     */
    public static boolean isDebugging() {
        return debugging;
    }

    /**Sets Debugging flag as true if it's time to unit test.
     * @param on - the debugging to set, true for on and false
     * for off
     */
    public static void setDebugging(boolean on) {
        UnitTestFirewall.debugging = on;
    }
```

```
    private final static NtpClock _ntpClock=new NtpClock();

    private static class NtpClockWrapper implements Clock {
        public long getTime() {
            return _ntpClock.getTime();
        }
    }

    private static class SystemClock implements Clock {
        public long getTime() {
            return System.currentTimeMillis();
        }
    }

    private static SystemClock sysClock = null;
    private static NtpClockWrapper ntpClock = null;

    /**Returns the Clock instance for <code>SystemTimeSynchronizer
     * </code>'s invocation.
     * @return - Clock instance
     */
    public static Clock getClock() {
            if(debugging) {
                if(sysClock == null)
                    sysClock = new SystemClock();
                return sysClock;
            }
            else {
                if(ntpClock == null)
                    ntpClock = new NtpClockWrapper();
                return ntpClock;
            }
    }
}
```

相应的 SystemTimeSynchronizer#syncTime()也做了修改，不再直接调用 NtpClock。

```
package shannon.demo;

import thirdparty.any.NtpClock;

public class SystemTimeSynchronizer {

    public int syncTime() {
```

```
        //long currentTime = new NtpClock().getTime();
        long currentTime = UnitTestFirewall.getClock().getTime();
        long interval = System.currentTimeMillis()-currentTime;
        if(interval == 0) {
            return 0;
        } else if(interval > 0) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

在正常运行时，debugging 为 false，SystemTimeSynchronizer 调用的是 NtpClockWrapper 实例。在单元测试时，可以在测试代码里将 debugging 设为 true，这样 SystemTimeSynchronizer 实际调用的是 SystemClock 实例。

```
/**shannon.demo.unittest is the unit test codes
 * package for the demonstration.
 */
package shannon.demo.unittest;

import shannon.demo.SystemTimeSynchronizer;
import shannon.demo.UnitTestFirewall;
import junit.framework.TestCase;

/**<code>SystemTimeSynchronizerTest</code> is the
 * unit test class for SystemTimeSynchronizer
 * @author Shannon Qian
 */
public class SystemTimeSynchronizerTest extends TestCase {

    protected void setUp() throws Exception {
        super.setUp();
        UnitTestFirewall.setDebugging(true);
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testSyncTime() {
        int result = new SystemTimeSynchronizer().syncTime();
        assertTrue(result==0||result==1||result==-1);
```

```
    }
}
```

另一方面，其实我们也看到了，如果 NtpClock 对象后面连着真正的 NTP 服务器，那么它永远只能返回正确的时间，而 SystemTimeSynchronizer#syncTime()实际上提供了系统时间快于、慢于和恰好等于标准时间三种情况的处理逻辑。如果要测全三种场景，修改 NTP 服务器会是件麻烦的事情。或者为此等很长时间，还要看运气。所以目前为止，我们只能在 `SystemTimeSynchronizerTest#testSyncTime()` 里验证 `SystemTimeSynchronizer#syncTime()` 的返回值是`{0, 1, -1}`中的任何一个。

要方便的测试所有的处理逻辑分支，关键就是要能够随心所欲的控制 Clock#getTime() 的返回值。在此定义一个 DebugClock 替换原来的 SystemClock 类。

```
package shannon.demo;

/**
 * <code>DebugClock</code> is a Clock for debugging.
 * It accepts arbitrary value as candidate time for
 * the next return of {@link #getTime()}
 * @author Shannon Qian
 */
public class DebugClock implements Clock {
    private long time=-1L;

    /**Sets candidate time for debugging.
     * @param t - the candidate time value in millisecond
     * @see #getTime()
     */
    public void setTime(long t) {
        this.time=t;
    }

    /** Returns the time in millisecond.
     * By default, it returns system time if the
     * candidate time is not preset. else it will
     * return the candidate time after reseting it.
     * @return - time in millisecond
     * @see #setTime(long)
     */
    public long getTime() {
        if(time<0L)
            return System.currentTimeMillis();
        long t=this.time;
        this.time=-1L;
        return t;
```

```
    }
}
```

在 UnitTestFirewall 中也要做相应的修改，以允许在测试时，定制 Clock 实例。

```
package shannon.demo;

import thirdparty.any.NtpClock;

/**
 * <code>UnitTestFirewall</code> is the facility to
 * ease unit test
 * @author Shannon Qian
 */
public final class UnitTestFirewall {
    private static boolean debugging=false;

    /**Returns true if it's in debugging mode, else false.
     * @return the debugging
     */
    public static boolean isDebugging() {
        return debugging;
    }

    /**Sets Debugging flag as true if it's time to unit test.
     * @param on - the debugging to set, true for on and false
     * for off
     */
    public static void setDebugging(boolean on) {
        UnitTestFirewall.debugging = on;
    }

    private final static NtpClock _ntpClock=new NtpClock();

    private static class NtpClockWrapper implements Clock {
        public long getTime() {
            return _ntpClock.getTime();
        }
    }

    private static NtpClockWrapper ntpClock = null;
    private static Clock clock = null;

    /**sets the Clock instance for debugging.
```

```
     * If candidate is not null, {@link #getClock()} will
     * return it when debugging is true.
     * @param candidate - the Clock instance for debugging
     */
    public static void presetClock(Clock candidate) {
        clock=candidate;
    }

    /**Returns the Clock instance for <code>SystemTimeSynchronizer
     * </code>'s invocation.
     * @return - Clock instance
     */
    public static Clock getClock() {
        if(debugging && clock != null) {
            return clock;
        } else {
                if(ntpClock == null)
                    ntpClock = new NtpClockWrapper();
                return ntpClock;
        }
    }
}
```

接着，SystemTimeSynchronizerTests 就可以作很大的改进，通过预设标准时间，测试 SystemTimeSynchronizer#syncTime()的各个处理分支了。

```
/**shannon.demo.unittest is the unit test codes
 * package for the demonstration.
 */
package shannon.demo.unittest;

import shannon.demo.DebugClock;
import shannon.demo.SystemTimeSynchronizer;
import shannon.demo.UnitTestFirewall;
import junit.framework.TestCase;

/**<code>SystemTimeSynchronizerTest</code> is the
 * unit test class for SystemTimeSynchronizer
 * @author Shannon Qian
 */
public class SystemTimeSynchronizerTest extends TestCase {

    protected void setUp() throws Exception {
        super.setUp();
```

```
        UnitTestFirewall.setDebugging(true);
        UnitTestFirewall.presetClock(debugClock);
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    private static DebugClock debugClock=new DebugClock();

    public void testSyncTime() {
        SystemTimeSynchronizer sync=new SystemTimeSynchronizer();
        assertTrue(sync.syncTime()==0);

        debugClock.setTime(System.currentTimeMillis()-10L);
        assertTrue(sync.syncTime()==1);

        debugClock.setTime(System.currentTimeMillis()+10L);
        assertTrue(sync.syncTime()==-1);
    }
}
```

~~这是我在具体工作中一个图省力的法子，所谓单元测试中的“防火墙”就是将外部模块的影响屏蔽在模块单元测试之外，不知是否违背单元测试的精神。~~

再总结一下这种测试桩方法的好处，我体会到的有两个：

1) 免去许多配置和调试外部模块的工作；

2) 可以方便的模拟其他模块的各种行为，提供各种场景测试条件。

当然，实际问题要复杂的多，所以我们可能要比这个例子做得更多。但正因为实际问题复杂，我们省的力气也要多得多。

另外有同事告诉我 JMock 和 easyMock 也提供和我上面所讨论的法子类似的功能。我花了极少的时间试着了解过 JMock，但还没入门，所知很少，还望有经验的同行指教。再次感谢 myworkfirst。