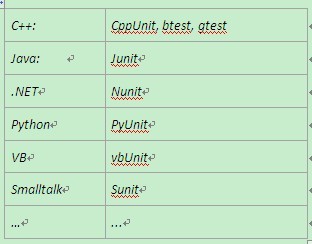
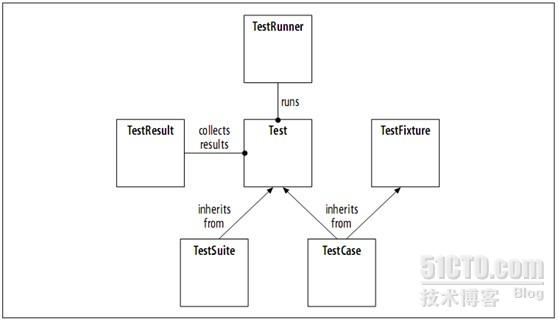
**1 现有的单元测试框架**单元测试是保证程序正确性的一种有效的测试手段，对于不同的开发语言，通常都能找到相应的单元框架。

[](http://img1.51cto.com/attachment/201112/191839171.jpg)

借助于这些单测框架的帮助，能够使得我们编写单元测试用例的过程变得便捷而优雅。框架帮我们提供了case的管理，执行，断言集，运行参数，全局事件工作，所有的这些使得我们只需关注：于对于特定的输入，被测对象的返回是否正常。

那么，这些xUnit系列的单元测试框架是如何做到这些的了？分析这些框架，发现所有的单元测试框架都是基于以下的一种体系结构设计的。

[](http://img1.51cto.com/attachment/201112/191913800.jpg)

如上图所示，单测框架中通常包括TestRunner, Test, TestResult, TestCase, TestSuite, TestFixture六个组件。

**TestRuner：**负责驱动单元测试用例的执行，汇报测试执行的结果，从而简化测试

**TestFixture：**以测试套件的形式提供setUp()和tearDown()方法，保证两个test case之间的执行是相互独立，互不影响的。

**TestResult：**这个组件用于收集每个test case的执行结果

**Test：**作为TestSuite和TestCase的父类暴露run()方法为TestRunner调用

**TestCase：**暴露给用户的一个类，用户通过继承TestCase，编写自己的测试用例逻辑

**TestSuite：**提供suite功能管理testCase

正因为相似的体系结构，所以大多数单元测试框架都提供了类似的功能和使用方法。那么在单测中引入单元测试框架会带来什么好处，在现有单元测试框架下还会存在什么样不能解决的问题呢？

**2 单元测试框架的优点与一些问题**在单元测试中引入单测框架使得编写单测用例时，不需要再关注于如何驱动case的执行，如何收集结果，如何管理case集，只需要关注于如何写好单个测试用例即可；同时，在一些测试框架中通过提供丰富的断言集，公用方法，以及运行参数使得编写单个testcase的过程得到了最大的简化。

那这其中会存在什么样的疑问了？

我在单元测试框架中写一个TestCase，与我单独写一个cpp文件在main()方法里写测试代码有什么本质却别吗？用了单元测试框架，并没有解决我在对复杂系统做单测时遇到的问题。

没错，对于单个case这两者从本质上说是没有区别的。单元测试框架本身并没有告诉你如何去写TestCase，在这一点上他是没有提供任何帮助的。所以对于一些复杂的场景，只用单元测试框架是有点多少显得无能为力的。

使用单元测试框架往往适用于以下场景的测试：单个函数，一个class，或者几个功能相关class的测试，对于纯函数测试，接口级别的测试尤其适用，如房贷计算器公式的测试。

但是，对于一些复杂场景：

 被测对象依赖复杂，甚至无法简单new出这个对象 对于一些failure场景的测试 被测对象中涉及多线程合作 被测对象通过消息与外界交互的场景 …单纯依赖单测框架是无法实现单元测试的，而从某种意义上来说，这些场景反而是测试中的重点。

以分布式系统的测试为例，class 与 function级别的单元测试对整个系统的帮助不大，当然，这种单元测试对单个程序的质量有帮助；分布式系统测试的要点是测试进程间的交互：一个进程收到客户请求，该如何处理，然后转发给其他进程；收到响应之后，又修改并应答客户；同时分布式系统测试中通常更关注一些异常路径的测试，这些场景才是测试中的重点，也是难点所在。

Mock方法的引入通常能帮助我们解决以上场景中遇到的难题。

**3 Mock的引入带来了什么**在维基百科上这样描述Mock：In object-oriented programming, mock objects are simulated objects that mimic the behavior of real objects in controlled ways. A computer programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior. of a human in vehicle impacts.

Mock通常是指，在测试一个对象A时，我们构造一些假的对象来模拟与A之间的交互，而这些Mock对象的行为是我们事先设定且符合预期。通过这些Mock对象来测试A在正常逻辑，异常逻辑或压力情况下工作是否正常。

引入Mock最大的优势在于：Mock的行为固定，它确保当你访问该Mock的某个方法时总是能够获得一个没有任何逻辑的直接就返回的预期结果。

Mock Object的使用通常会带来以下一些好处：

 隔绝其他模块出错引起本模块的测试错误。 隔绝其他模块的开发状态，只要定义好接口，不用管他们开发有没有完成。 一些速度较慢的操作，可以用Mock Object代替，快速返回。对于分布式系统的测试，使用Mock Object会有另外两项很重要的收益：

 通过Mock Object可以将一些分布式测试转化为本地的测试 将Mock用于压力测试，可以解决测试集群无法模拟线上集群大规模下的压力**4 Mock的应用场景**在使用Mock的过程中，发现Mock是有一些通用性的，对于一些应用场景，是非常适合使用Mock的：

 真实对象具有不可确定的行为(产生不可预测的结果，如股票的行情) 真实对象很难被创建(比如具体的web容器) 真实对象的某些行为很难触发(比如网络错误) 真实情况令程序的运行速度很慢 真实对象有用户界面 测试需要询问真实对象它是如何被调用的(比如测试可能需要验证某个回调函数是否被调用了) 真实对象实际上并不存在(当需要和其他开发小组，或者新的硬件系统打交道的时候，这是一个普遍的问题)当然，也有一些不得不Mock的场景：

 一些比较难构造的Object：这类Object通常有很多依赖，在单元测试中构造出这样类通常花费的成本太大。 执行操作的时间较长Object：有一些Object的操作费时，而被测对象依赖于这一个操作的执行结果，例如大文件写操作，数据的更新等等，出于测试的需求，通常将这类操作进行Mock。 异常逻辑：一些异常的逻辑往往在正常测试中是很难触发的，通过Mock可以人为的控制触发异常逻辑。在一些压力测试的场景下，也不得不使用Mock，例如在分布式系统测试中，通常需要测试一些单点（如namenode，jobtracker）在压力场景下的工作是否正常。而通常测试集群在正常逻辑下无法提供足够的压力（主要原因是受限于机器数量），这时候就需要应用Mock去满足。

在这些场景下，我们应该如何去做Mock的工作了，一些现有的Mock工具可以帮助我们进行Mock工作。

**5 Mock工具的介绍**手动的构造 Mock 对象通常带来额外的编码量，而且这些为创建 Mock 对象而编写的代码很有可能引入错误。目前，有许多开源项目对动态构建 Mock 对象提供了支持，这些项目能够根据现有的接口或类动态生成，这样不仅能避免额外的编码工作，同时也降低了引入错误的可能。

C++: GoogleMock <http://code.google.com/p/googlemock/>

Java: EasyMock <http://easymock.org/>

通常Mock工具通过简单的方法对于给定的接口生成 Mock 对象的类库。它提供对接口的模拟，能够通过录制、回放、检查三步来完成大体的测试过程，可以验证方法的调用种类、次数、顺序，可以令 Mock 对象返回指定的值或抛出指定异常。通过这些Mock工具我们可以方便的构造 Mock 对象从而使单元测试顺利进行，能够应用于更加复杂的测试场景。

以EasyMock为例，通过 EasyMock，我们可以为指定的接口动态的创建 Mock 对象，并利用 Mock 对象来模拟协同模块，从而使单元测试顺利进行。这个过程大致可以划分为以下几个步骤：

 使用 EasyMock 生成 Mock 对象 设定 Mock 对象的预期行为和输出  将 Mock 对象切换到 Replay 状态 调用 Mock 对象方法进行单元测试 对 Mock 对象的行为进行验证EasyMock的使用和原理： <http://www.ibm.com/developerworks/cn/opensource/os-cn-easymock/>

EasyMock 后台处理的主要原理是利用 java.lang.reflect.Proxy 为指定的接口创建一个动态代理，这个动态代理，就是我们在编码中用到的 Mock 对象。EasyMock 还为这个动态代理提供了一个 InvocationHandler 接口的实现，这个实现类的主要功能就是将动态代理的预期行为记录在某个映射表中和在实际调用时从这个映射表中取出预期输出。

借助类似于EasyMock这样工具，大大降低了编写Mock对象的成本，通常来说Mock工具依赖于单元测试框架，为用户编写TestCase提供便利，但是本身依赖于单元测试框架去驱动，管理case，以及收集测试结果。例如EasyMock依赖于JUint，GoogleMock依赖于Gtest。

那么有了单元测试框架和相应的Mock工具就万事俱备了，还有什么样的问题？正如单元测试框架没有告诉你如何写TestCase一样，Mock工具也没有告诉你如何去选择Mock的点。

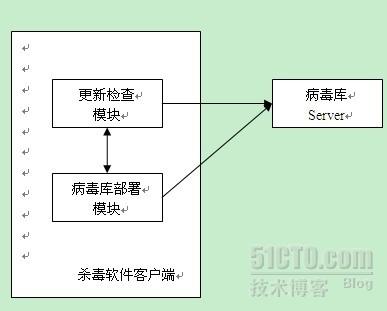
**6 如何选择恰当的mock点**对于Mock这里存在两个误区，1.是Mock的对象越多越好；2.Mock会引入巨大的工作量，通常得不偿失。这都是源于不恰当的Mock点的选取。

这里说的如何选择恰当的mock点，是说对于一个被测对象，我们应当在外围选择恰当的mock对象，以及需要mock的接口。因为对于任意一个对象，任意一段代码逻辑我们都是有办法进行Mock的，而Mock点选择直接决定了我们Mock的工作量以及测试效果。从另外一种意义上来说，不恰当Mock选择反而会对我们的测试产生误导，从而在后期的集成和系统测试中引入更多的问题。

在mock点的选择过程中，以下的一些点会是一些不错的选择

 网络交互：如果两个被测模块之间是通过网络进行交互的，那么对于网络交互进行Mock通常是比较合适的，如RPC 外部资源：比如文件系统、数据源，如果被测对象对此类外部资源依赖性非常强，而其行为的不可预测性很可能导致测试的随机失败，此类的外部资源也适合进行Mock。 UI：因为UI很多时候都是用户行为触发事件，系统本身只是对这些触发事件进行相应，对这类UI做Mock，往往能够实现很好的收益，很多基于关键字驱动的框架都是基于UI进行Mock的 第三方API：当接口属于使用者，通过Mock该接口来确定测试使用者与接口的交互。当然如何做Mock一定是与被系统的特性精密关联的，一些强制性的约束和规范是不合适的。这里介绍几个做的比较好的mock的例子。

**1. 杀毒软件更新部署模块的Mock**这个例子源于一款杀毒产品的更新部署模块的测试。对于一个杀毒软件客户端而言，需要通过更新检查模块与病毒库Server进行交互，如果发现病毒库有更新则触发病毒库部署模块的最新病毒库的数据请求和部署工作，要求部署完成后杀毒软件客户端能够正常工作。

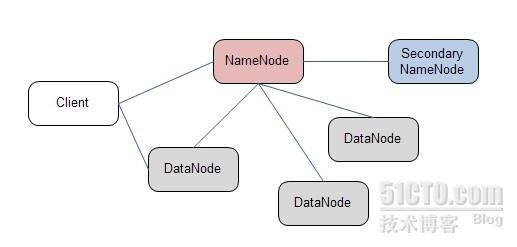
[](http://img1.51cto.com/attachment/201112/191927452.jpg)

对于这一场景的测试，当时受限于这样一个条件，通常的病毒库server通常最多一天只更新一次病毒库，也就是说如果使用真实的病毒库server，那么针对更新部署模块的测试一天只能被触发一次。这是测试中所不能容忍的，通过对病毒库server进行mock可以解决这个问题。

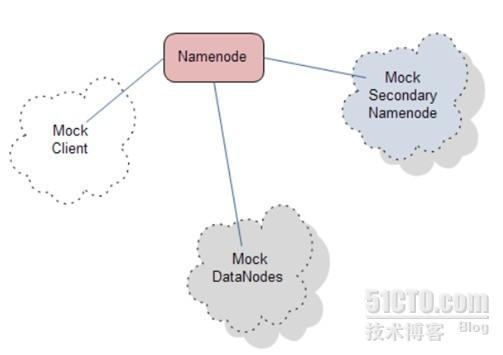
对于这个场景可以采取这样一种Mock方式：用一个本地文件夹来模拟病毒库server，选择更新部署模块与病毒库server之间交互的两个函数checkVersion()，reqData()函数进行Mock。

checkVersion()工作原先的工作是检查病毒库Server的版本号，以决定是否触发更新，将其行为Mock为检查一个本地文件夹中病毒库的版本号；reqData()原有的行为是从病毒库Server拖取病毒库文件，将其Mock为从本地文件夹中拖取病毒库文件。通过这种方式我们用一个本地文件夹Mock病毒库Server的行为，其带来的产出是：我们可以随意的触发病毒库更新操作以及各种异常。通过这种方式发现了一个在更新部署过程中，病毒库Server的病毒库版本发生改变造成出错的严重bug，这个是在原有一天才触发一次更新操作的情况下永远也无法发现的。

**2. 分布式系统中对NameNode模块的测试**

[](http://img1.51cto.com/attachment/201112/191936107.jpg)

在测试NameNode模块的过程中存在这样一个问题，在正常逻辑无压力条件下NameNode模块都是工作正常的。但是线上集群在大压力的情况下，是有可能触发NameNode的问题的。但是原有的测试方法下，我们是无法对NameNode模拟大压力的场景的（因为NameNode的压力主要来源于DateNode数量，而我们测试集群是远远无法达到线上几千台机器的规模的），而NameNode单点的性能瓶颈问题恰恰是测试的重点，真实的DataNode是无法满足测试需求的，我们必须对DataNode进行Mock。

[](http://img1.51cto.com/attachment/201112/191948572.jpg)

如何对DateNode进行Mock了，最直观的想法是选择NameNode与DataNode之间的交互接口进行Mock，也就是他们之间的RPC交互，但是由于NameNode与DataNode之间的交互信息种类很多，所以其实这并不是一种很好的选择。

换个角度来想，NameNode之上的压力是源于对HDFS的读写操作造成的NameNode上元数据的维护，也就是说，对于NameNode而言，其实他并不关心数据到底写到哪里去了，只关心数据是否读写成功。如果是这种场景Mock就可以变的简单了，我们可以直接将DataNode上对块的操作进行mock，比如，对一次写请求，DataNode并不触发真实的写操作，而直接返回成功。通过这种方式，DataNode去除了执行功能，只保留了消息交互功能，间接的实现了我们的测试需求，且工作量比之第一种方案小很多。

**3. 开源社区提供的MRUnit测试框架**在原有框架下，对于MapReduce程序的测试通常是无法在本地验证的，更不用说对MapReduce程序进行单测了。而MRUnit通过一个简单而优雅的Mock，却实现了一个基于MapReduce程序的单测框架。

基于MRUINT框架可以将单测写成如下形式：

[](http://img1.51cto.com/attachment/201112/192014604.jpg)

在这个框架中定义了MapDriver，ReducerDriver，MapReduceDriver三个有点类似容器的driver，通过driver来驱动map，reduce或者整个mapreduce过程的执行。

如上例，在driver中设定mapper为IdentityMapper，通过withInput方法设定输入数据，通过withOutput方法设定预期结果，通过runTest方法来触发执行并进行结果检测

他的实现原理是将outputCollector做Mock，outputCollectort中的emit方法实现的逻辑是将数据写到文件系统中，Mock后是通过另外一个进程去收集数据并保存在内存中，从而实现最终结果的可检验（在自己的数据结构中比对结果）。

实现的原理很简单，这样做mock就会精巧，只选择最底层的一些简单却又依赖广泛的点（依赖广泛指模块间的数据流通常都走这样的点过）做mock，这样通常效果很好且简单

当然这个例子中也有一些缺陷：1.因为在outputcollector层做mock的数据截取，使得无法过partition的分桶逻辑；2.这个框架是写内存的，无法最终改成压力性能测试工具。