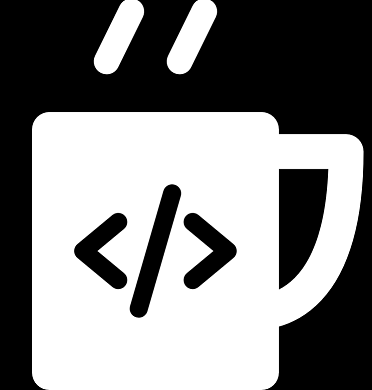
C#编程魔法书

第七章 工程编程

截止目前我们已经讨论了C#的不少知识，本章讨论如何用这些知识和功能搭建一个完整的C#应用程序，我们将继续完善前面几章演示的交易示例程序，添加自动化测试、可持续集成、版本控制和与用户交互的命令行和图形界面等模块。

# 自动化测试

软件开发过程大致分为：需求分析、概要设计、详细设计、软件编码、单元测试、集成测试、系统测试和验收测试等几个阶段步骤，其模型构图形似字母V，因此又称为V模型。其中单元测试是指对软件中最小可测试单元进行检查和验证，最小可测单元的定义要根据实际情况来判定，单元是人为规定的最小被测功能模块。单元测试是软件开发过程中进行的最低级别的测试活动，软件的独立单元在与程序的其他部分隔离的情况下进行测试。由于单元测试的粒度很小，被测的代码通常无法组成一个可供人机交互的程序，所以都是由程序员编写代码自动化执行。为了保证所交付的程序质量，大型软件开发团队通常都会强制要求只有程序通过所有的单元测试之后，才会开始进行集成测试等后续测试工作。

虽然自动化的单元测试对软件的交付提供了强大的质量保证，但是编写单元测试代码的要求很高，其不仅要求开发人员有较高的编码水平，而且很多团队在项目工期紧张的情况下往往会跳过这个编码环节。为了强制在开发过程中编写单元测试，有些团队则使用测试驱动开发的理念，这种开发的做法是在编码之前先写单元测试用例，再通过修复编译错误和测试失败的方法，逐步补齐代码完成整个工程的开发。单元测试要求被测系统的需求是稳定的，经常变化的需求会导致写好的单元测试代码失去作用，无谓加大团队的开发工作量。如果被测系统只是一次性交付的话，编写单元测试代码的意义也不大，因为一次性交付的软件经过交付后再次开发的频率很小，执行手工测试就可以满足需求，相同的测试场景，编写自动化测试相对于手工执行一次测试的代价高昂的多，只有需要多次频繁测试的情况下，自动化测试的性价比才会超过手工测试。从笔者个人的开发经验来看，单元测试对于长期维护不断升级的软件系统非常有用，如.NET框架这样的软件产品、电商平台中中间件模块、通过WEB服务或RESTful接口向外提供服务的模块都属于这种软件系统。 而很多业务系统，一次性交付的ERP、MES等项目，特别是ERP软件上层的业务系统网站，就没有必要编写单元测试，一方面性价比较低，另一方面业务系统本身就是集成了很多软件模块和系统组成的，其最基本的功能测试已经到集成、甚至是系统测试范畴了，自动化的难度也很高。

## 在Visual Studo中编写单元测试

与JAVA等编程语言将单元测试和被测代码放在同一个工程里的做法不同，.NET中单元测试和被测的代码是分开在两个工程里放置的，这种做法笔者觉得较Java更好一点。单元测试独立作为一个工程，与被测代码分开可以保证两个工程独立开发，在团队管理和版本控制方面会容易一些，而且独立的单元测试工程，可以引用多个组件，同时对多个工程进行测试。Visual Studio内置了单元测试模板，虽然其名称是单元测试，但实际上这个模板可以用在任意一个自动化测试场景，比如说UI自动化测试也可以用这个模板编写。VS中有两套单元测试模板，一套是基于VS团队开发的MSTest框架的模板，一套是基于开源的NUnit框架的模板，两个框架的用法大同小异，本书采用MSTest框架开发单元测试。在VS中创建新项目时，选择”MSTest测试项目（.NET Core）”，如图7 - 1所示，VS中内置的项目模板太多，可以通过搜索文本框或者下面的筛选下拉框检索到模板，根据创建向导填写必要的项目信息后就创建了一个单元测试项目了。

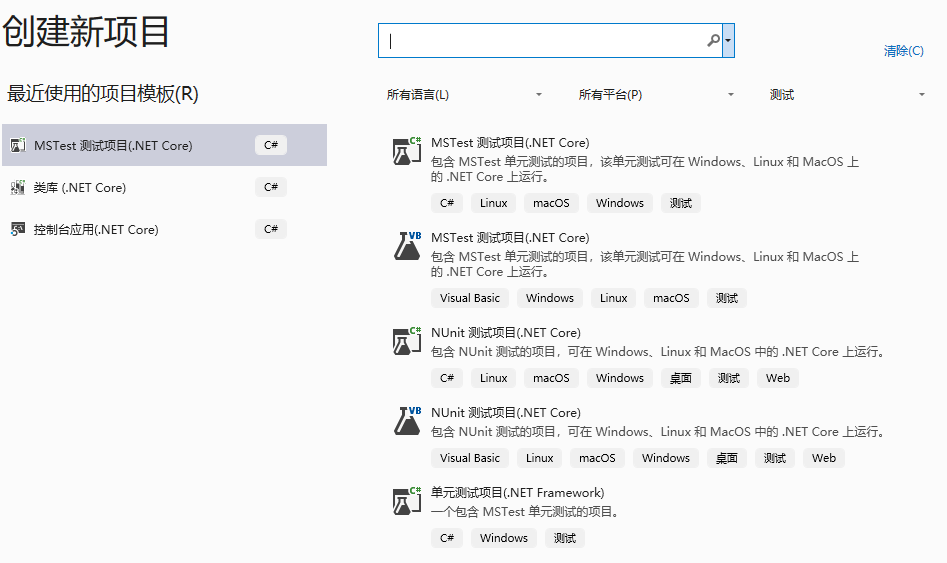


图7 - 1 在Visual Studio里创建单元测试工程

新建的工程预先创建了一个空的单元测试类型，代码清单7 - 1是更完整的测试类型结构，基本上所有语言的单元测试框架都有类似的结构，MSTest通过特性来识别单元测试的元素，包含单元测试用例的类型需要标注上TestClass特性，如第1行，这个类型通常可以看做包含很多测试用例的一个测试集合（Test Suite）。而实际的单元测试则使用TestMethod特性标注，如第14行的方法就是一个完整的单元测试用例。一般测试类型只包含测试用例方法就足够完成测试目的了，但有些测试场景需要在执行测试的前后做一些准备和清理工作，如果在每个测试方法里都重复编写和执行这些辅助性的工作就很没有必要了，单元测试框架都会提供辅助方法用来编码这些通用工作。如第7行的TestInitializer特性，标注了这个特性的方法被用来执行测试之前的准备工作，MSTest会自动在运行每个单元测试用例方法之前先运行这个方法。而第10行的TestCleanup特性，则标注了测试用例运行完毕后执行清理工作的通用方法。如果要在测试集合所有的用例运行之前做准备工作，可以用ClassInitializer特性标注，如第5行的方法，这个方法除了用ClassInitializer特性标注之外，必须是静态方法，且必须接受一个类型为TestContext的参数，才会被MSTest识别为有效的测试集合范围的准备方法。

代码清单7 - 1 MSTest单元测试用例结构

|  |
| --- |
| 01 [TestClass]  02 public class 开发数据库中测试  03 {  04     [ClassInitialize]  05     public static void 准备依赖注入环境(TestContext tc) {}  06  07     [TestInitialize]  08     public void 重新构建数据库() {}  09  10     [TestCleanup]  11     public void 删除数据库() {}  12  13     [TestMethod]  14     public void 验证使用实际数据库根据策略下达买单() {}  15 } |

代码清单7 - 1中测试用例的类型和方法的命名笔者特意使用中文命名，这是一个小技巧，将测试方法作为测试用例的标题命名，方便在VS的测试资源管理器中快速了解用例的测试目的，如图7 - 2所示，其中类型名作为测试用例集合的名称，而方法名当做测试标题使用，便于管理。图7 - 2中的测试用例还标注了TestCategory特性，这是MSTest提供的另一个归类测试用例的特性，VS的测试资源管理器中的“特征”列会列出附加在测试用例上的“TestCategory”，程序员可以用此列过滤测试用例，并执行选中“特征”的测试用例。

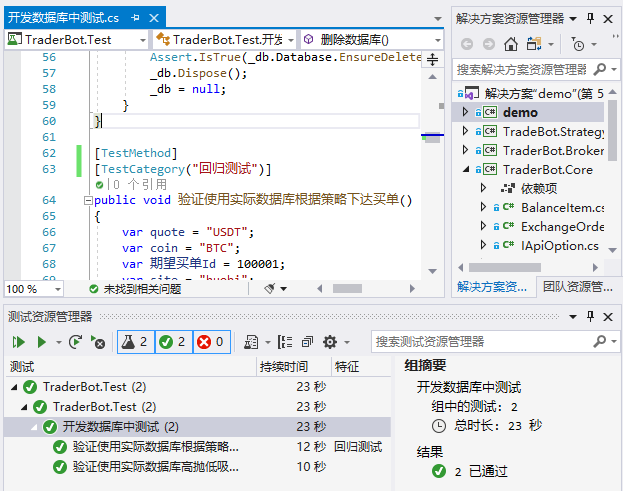


图7 - 2 VS测试资源管理器

一般测试开发中在VS图形环境里运行测试用例，当用例写完之后会加入可持续集成环境在每日构建（daily build）过程中自动化执行：

* 很多开发团队都要求程序员将代码签入到代码版本仓库之前，必须确保所有单元测试用例执行通过。在开发过程中要执行所有测试用例的话，在解决方案资源管理器中打开测试工程中任意一个测试用例源码文件，点击菜单栏上的“测试”->“运行所有测试”命令运行工程里所有的测试用例；

当新的代码签入到代码版本仓库的时候，特别是在可持续集成环境中，往往会用钩子程序自动编译最新代码并执行单元测试以验证代码的质量，对于.net core工程，在测试工程的根目录下使用下面的命令行运行所有的用例：

dotnet test

对于.net工程，需要先编译工程，再使用MSTest命令加载生成的dll文件运行所有测试用例 – 需要确保MSTest命令在命令行的PATH搜索目录中：

mstest /testcontainer:tests.dll

* 对于大型软件项目，可能有很多测试用例，完全跑一遍所有测试用例会耗费很多时间，在开发过程中程序员可以选择只执行一小部分测试用例，例如执行一个测试集合进行小规模验证，通过后再签入到代码仓库中。这种情况下，可以打开VS的“测试资源管理器”，右键选择一个测试集合（即测试类型），然后点击右键菜单中的“运行”项执行其中的所有测试用例；
* 如果一个测试类型里的测试用例覆盖面不够的话，也可以通过“特征”来归类用例，比如说将最重要的测试场景归类为“回归测试”一类，每次进行代码签入之前至少要保证这些测试场景通过。这种情况下，打开“测试资源管理器”，根据“特征”列过滤出期望运行的测试集合，然后点击右键菜单中的“运行”项执行选中特征中的所有测试用例；
* 在用例开发过程中，没有必要执行其他测试用例，只要运行正在开发的用例就可以了，在VS中打开要执行的测试用例源码，在测试用例方法的代码范围内单击右键，选择右键菜单中的“运行测试”就可以运行当前的单个测试。

## 测试驱动开发

前面的小节只谈了VS中测试用例的结构和运行测试用例的方法，这一小节里我们补齐测试用例代码。这里笔者要实现一个简单的策略交易程序，基本功能是在一个交易所里重复执行低买高卖的交易策略，首先针对某些交易对设定策略，策略包括几个信息：目标买入价、目标卖出价（或者是目标利润）和初始买入量。当某个交易对的市场价进入买入价范围时下达买单，并将买单信息保存到数据中便于事后的盈利分析。买单成交后，则持续监控该交易对的行情，当行情上涨到目标卖出价（或者满足预期利润）后，卖出套利，然后再重复该套利操作。

到目前为止，策略程序只对接了交易所的API，本节采用测试驱动开发完成交易策略的编码，编码逻辑大致分为以下几个步骤：

1. 需要在数据里保存用户的交易策略，即保存目标买入价、目标卖出价和初始买入量等信息。
2. 当程序启动后，不停地接收指定交易对的最新行情。
3. 如果行情到达目标买入价，先判断用户是否还有未成交的买单，如果有的话则不需要进行重复购买，没有的话则下达买入指令。
4. 继续监控交易对的最新行情，针对用户的每一个成交的买单，如果行情到达预设的目标卖出价，则下达卖出指令获利。

从前面的步骤描述里，我们大致可以将持续划分为几个模块：保存用户交易策略和订单历史的数据库模块，对接交易所API的接口模块，执行交易策略的引擎模块以及用户交互的界面模块，几个模块之间的关系如图7 - 3所示。为了避免整个程序跟接口模块绑定的太紧，程序抽象出一个接口抽象模块，模块定义在TradeBot.Core工程里，里面只包含必要的接口和数据类型定义，执行具体交易所API交互的工作则放在TradeBot.Brokers工程里。Trader.Strategy.Band工程保存了执行交易策略的引擎模块和保存用户数据的数据库模块，这两个模块最好是分在两个工程里定义，这里出于简化工程的目的将两个模块放在了一起。将接口的具体实现TradeBot.Brokers和依赖其的客户模块TradeBot.Strategy.Band通过一个抽象层TradeBot.Core分开，是一个常用的软件架构技巧，一般要将两个耦合很紧的模块解耦，常见的技巧是把两个模块耦合的地方抽象成接口并封装成一个新的抽象层模块，就实现架构的解耦了。如本例中要支持新交易所，替换TradeBot.Brokers模块，TradeBot.Strategy.Band模块不需要更改代码就可以直接使用新的模块。最后与用户交互的界面模块引用这四个模块响应用户的指令，在本章最后才会实现界面模块，图7 - 3将其标为虚线。单元测试模块需要测试接口和引擎的逻辑，因此也需要引用到这四个模块，单元测试模块在TradeBot.Test工程中。

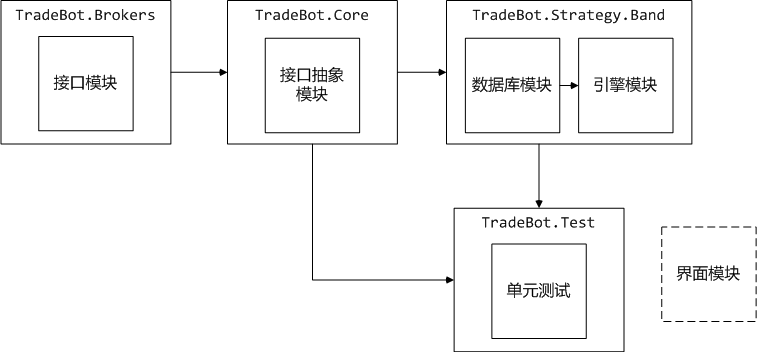


图7 - 3 交易程序中各个模块之间的架构

首先我们先编写一个测试用例来描述用户制定交易策略并由引擎执行的全过程，通过这个用例来定义模块之间的接口和数据类型，如代码清单7 - 2就是为了这个目的定义的，随着代码细节的逐步完善，最终的测试用例不一定跟代码清单7 - 2完全一样，因此笔者将其定义为伪码。

首先程序需要用户制定交易策略才能运行，而交易策略将会保存到数据库中，第9 - 11行代码用来定义数据库中保存交易策略的表 – BandStrategies，表结构使用BandStrategy的构造方法定义必填字段。我们希望多个用户使用策略交易系统，因此在BandStrategies中添加了UserId列用来关联用户信息，其他几个字段用来保存用户具体的交易策略。由于我们还没有在工程里定义BandStrategy类型，所以在调用BandStrategy的构造方法时，使用了命名参数列表。第11行将用户制定的策略保存进数据库，如果测试写的细致一些的话，可以单独写一个用例专门验证策略是否能正确保存，这个用例留给有兴趣的读者自己实现。

接下来需要按照用户的设置监控交易所的行情执行策略，首先在14行创建策略交易引擎实例，将依赖注入容器作为参数传递给引擎，这样引擎实例可以在必要的时候从容器中获取依赖项实例。BandStrategyEngine定义了Start和Stop方法，以避免构造方法执行耗时操作，而且用户也可以通过这两个方法启动和停止引擎。由于系统支持多用户设置策略，引擎需要支持多个交易所，所以Start方法里接受一个可变参数列表，用以传入需要监控的交易所实例列表。在测试用例里我们只用了一个交易所做例子，所以只传入了一个实例，从API设计来看，建议读者将这个参数设置为数组类型的参数，一方面IDE能更准确的生成Start方法的声明，另一方面也增加了代码的可读性，笔者只是出于演示C#语法的目的将其定义为可变数组列表。

代码清单7 - 2 测试驱动开发策略交易程序的测试用例伪码

|  |
| --- |
| 01 [TestMethod]  02 public void 验证使用实际数据库根据策略下达买单()  03 {  04     var quote = "USDT";  05     var coin = "BTC";  06     var site = "huobi";  07     var uid = 1;  08  09     \_db.BandStrategies.Add(new BandStrategy(userId:uid,quote: quote,coin: coin,  10         bidPrice: 2000, tradeVolume: 1, askProfit: null, askPrice: 8000));  11     \_db.SaveChanges();  12  13     var broker = new Huobi("apikey", "apisecret");  14     var engine = new BandStrategyEngine(\_container);  15  16     engine.Start(quote, coin, broker);  17     Assert.AreEqual(1, broker.BuyLimitCallCount);  18     var openOrders = broker.GetOpenOrders(quote, coin);  19     Assert.AreEqual(1, openOrders.Length);  20 } |

代码清单7 - 2已经足够描述系统的完整工作过程了，在这个基础上我们通过补齐用例中未定义的类型和方法来实现策略交易程序，VS提供了自动生成未实现方法的声明可以帮我们节省很多时间，如图7 - 4中，将鼠标放在Start方法上面，VS会自动弹出一个浮动菜单以便生成方法声明，当然对于IDE辅助程序员生成的方法声明，还是需要审核一下。比如本例中， 测试用例中调用Start方法的第三个参数broker的类型是Huobi，VS生成的Start方法声明会将第三个参数的类型定义为Huobi，显然这不是我们想要的，需要人工干预将参数类型改为IBroker。

代码清单7 - 3 策略交易程序的下单逻辑

|  |
| --- |
| // 源码位置：第7章\demo\TradeBot.Strategy.Band\BandStrategyEngine.cs  01 private void PlaceBidOrder(BandStrategy strategy,  02     IBroker authBroker, BandTradeContext context) {  03     // 如果网络异常无法获取订单表信息，跳过处理该策略  04     var orderbook = authBroker.GetOrderBook(strategy.Quote, strategy.Coin);  05     if (orderbook == null) return;  06  07     // 最新卖价已经低到合适的位置，则下达买单  08     if (strategy.BidPrice > orderbook.LowestAsk) return;  09  10     // 判断用户账户的余额是否足够下买单  11     var balances = authBroker.GetBalances();  12     if (balances == null || balances.Length == 0) return;  13  14     // 买单需要判断计价币是否有足够余额  15     var balance = balances.SingleOrDefault(b => b.Currency == strategy.Quote);  16     if (balance == null || balance.Available == 0) return;  17  18     if (balance.Available <  19         (strategy.TradeVolume \* strategy.BidPrice \* (1 + authBroker.Fee)))  20         return;  21  22     // 余额足够，下达买单  23     var bid = authBroker.BuyLimit(  24         strategy.Quote, strategy.Coin, strategy.TradeVolume,strategy.BidPrice);  25     if (bid == null) return;  26  27     // 将最新的买单记录到数据库中  28     context.DbPlacedOrders.Add(new DbPlacedOrder() { /\* 保存订单 \*/}});  29 } |

代码清单7 - 3是下单逻辑的简单实现，其工作流程是在交易引擎启动后，不停的从交易所获取指定交易对的最新行情，由于要获取的是能够直接下单的行情，所以调用OrderBook方法获取的是实时报价单。每次拿到最新行情就跟用户预定的买入价进行比较，当行情满足条件时，从交易所获取用户的余额，只有在余额足够的前提下才会下达买单。最后当交易所返回买单订单号，表示成功下单后才把这笔买单记录到数据库中，以便执行后续的高卖套利策略。代码清单7 - 3中关键逻辑代码笔者加了一些注释，对于比较复杂的逻辑，笔者一般是先写注释描述想法再实现代码，建议读者在编码时也可以参考这种做法。

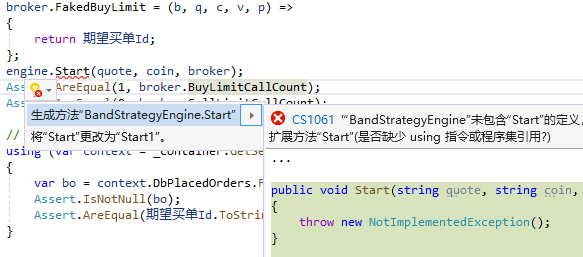


图7 - 4 使用VS的智能提示生成未实现的方法声明

一个完整的测试用例包含测试步骤和结果验证，前面的代码是组成测试步骤的代码，用例的最后就是结果验证代码，当行情满足策略的条件时，系统应该向交易所下达相应的指令，代码清单7 - 2第17 – 20行的代码就是用来执行这些验证工作的。在现实情况中行情是一直波动的，使用实际的行情来验证策略引擎工作正确与否不大现实，我们需要有一个模拟机制来按照测试需求伪造行情数据。根据图7 - 3展示的系统架构，引擎模块是跟接口抽象模块交互，再间接向实际的交易所接口模块下达指令，在测试用例中我们可以更换实际的交易所模块，使用模拟交易所模块来达到伪造行情的目的。代码清单7 - 3是示例代码中模拟交易所实现的代码摘要，实现思路是将伪造行情数据的工作交给测试用例，因此在第33行FakeBroker定义了一个名为FakedOrderBook的委托，36行IBroker的OrderBook方法的实现就是调用这个委托获得行情数据，测试用例通过实现这个委托方法来达到模拟行情数据的目的。由于测试用例可能会模拟各种交易对的交易场景，所以FakeBroker中Support方法的实现就是简单地对任意交易对返回true。

除了模拟行情数据以外，我们的测试用例还需要验证引擎是否在行情满足策略预定的条件下正确下单，因此FakeBroker也实现了BuyLimit和SellLimit下单方法。有几种模拟下单的方法：

1. 最简单的方案是在FakeBroker类维护自增的买单和买单订单号数字，这两个数字是静态变量，只要BuyLimit或者SellLimit被调用了，就累加相应的订单号数字。
2. 第二种方案是定义一个委托，由测试用例返回订单号，便于测试用例验证下单方法。
3. 第三种方案是在FakeBroker维护两个订单列表，分别是买单和卖单列表，调用BuyLimit时将买单添加进买单列表，而SellLimit则将卖单加进卖单列表，买卖单列表的Count可以作为最新的订单Id使用。而维护两个订单列表的好处是同时也实现了GetOpenOrders等订单相关的接口方法。

代码清单7 - 3里使用了最复杂的方案，将3个方案合并在一起，FakeBroker中定义了BuyLimitCallCount用来跟踪买单方法被调用的测试，当然同时也可以用作最新的订单号。第6行定义的FakedBuyLimit委托，测试用例可以在这里模拟包括下单失败在内的很多场景，在第14行FakedBuyLimit只要返回一个非正数值就可以模拟下单失败的场景了。最后FakeBroker里定义了\_orders实例变量，统一管理买卖方向的订单，用来在第30行实现GetOpenOrders方法。

代码清单7 - 4 模拟行情接口实现

|  |
| --- |
| // 源码位置：第7章\demo\TraderBot.Test\Mock\FakeBroker.cs  01 public class FakeBroker : IBroker  02 {  03     private List<ExchangeOrder> \_orders = new List<ExchangeOrder>();  04  05     public int BuyLimitCallCount { get; private set; }  06     public Func<FakeBroker, string, string, decimal, decimal, int>  07         FakedBuyLimit { get; set; }  08  09     public object BuyLimit(  10         string quote, string coin, decimal quantity, decimal price) {  11         if (FakedBuyLimit == null) return 0;  12  13         BuyLimitCallCount++;  14         var ret = FakedBuyLimit(this, quote, coin, quantity, price);  15         if (ret > 0) {  16             \_orders.Add(new ExchangeOrder() {  17                 Id = ret, Side = TradeSide.Buy, Site = Name,  18                 Quote = quote, Coin = coin, Price = price,  19                 PlacedTimestamp = DateTime.Now,  20                 Quantity = quantity, QuantityRemaining = quantity  21             });  22  23             return ret;  24         }  25         else {  26             return 0;  27         }  28     }  29  30     public ExchangeOrder[] GetOpenOrders(  31         string quote, string coin) => \_orders.ToArray();  32  33     public Func<FakeBroker, string, string, OrderBook>  34         FakedOrderBook { get; set; }  35  36     public OrderBook GetOrderBook(string quote, string coin,  37         int limit = 10) => FakedOrderBook(this, quote, coin);  38  39     public bool Support(string quote, string coin) => true;  40  41     public IBroker CreateAuthenticatedInstance(  42         string apiKey, string apiSecret) => this;  43 } |

代码清单7 - 5在测试用例中通过实现FakeBroker的几个委托实现模拟场景，如第6行创建FakeBroker对象时，第二个字典参数用来输入模拟用户在交易所中的余额，可以用来测试用户余额不足时引擎的处理方式。第13行通过FakedOrderBook委托设置了模拟的行情数据，最低卖价刚好满足策略中设置的2000的买价，这样引擎应该下达买单，否则就说明代码实现有错误。第21行的FakedBuyLimit委托返回了测试用例特意设置的买单Id，便于验证引擎是根据策略设置下达的买单，而不是随意下单。最后第25行启动引擎后，在26、27行判断是否执行了买单操作，并在第30行获取交易所未成交订单列表，根据预设的买单Id验证引擎的确根据我们交易策略下达了买单。

代码清单7 - 5 在测试用例中使用模拟交易所的方法

|  |
| --- |
| 01 var quote = "USDT";  02 var coin = "BTC";  03 var 期望买单Id = 100001;  04 var site = "huobi";  05  06 var broker = new FakeBroker(site, new Dictionary<string, decimal>()  07 {  08     { "USDT", 10000 },  09     { "BTC", 1 }  10 });  11 var engine = new BandStrategyEngine(\_container);  12  13 broker.FakedOrderBook = (b, q, c) =>  14 {  15     var ob = CreateOrderBook(site, q, c,  16         new decimal[,] { { 2000, 2 }, { 2001.88m, 1.2m } },  17         new decimal[,] { { 1999, 1 }, { 1998.99m, 2 } });  18  19     return ob;  20 };  21 broker.FakedBuyLimit = (b, q, c, v, p) =>  22 {  23     return 期望买单Id;  24 };  25 engine.Start(quote, coin, broker);  26 Assert.AreEqual(1, broker.BuyLimitCallCount);  27 Assert.AreEqual(0, broker.SellLimitCallCount);  28 var openOrders = broker.GetOpenOrders(quote, coin);  29 Assert.AreEqual(1, openOrders.Length);  30 Assert.IsNotNull(openOrders.SingleOrDefault(o => (int)o.Id == 期望买单Id)); |

## 模拟数据库

与手工测试用例不一样的地方是，自动化测试用例在执行过程中不能修改测试环境。在执行手工测试的时候，测试人员很喜欢将增改查删这几个操作的测试用例顺序执行，这样做可以大大提高测试效率。但在自动化测试中，改、查和删这三个测试用例不能依赖测试增加数据的测试用例的执行结果，这是因为在自动化测试中用例可能会单独执行，也可以会放在放在一个测试集合里批量执行，但执行不一定会按照增改查删的顺序，因此编写自动化测试用例的原则是避免对测试环境的修改。例如测试添加记录的用例，在测试完毕后应该删除新增的这条记录，这是因为同一个用例有可能会再次执行 – 例如开发调试用例的阶段。相应的，删除记录的用例应该在执行前先添加好要删除的记录。代码清单7 - 6是7.1.2小节中测试用例的准备和清理代码，如第2行的“重新构建数据库”方法，在每个测试用例执行之前创建一个新的数据库，并在数据库中准备好用户在模拟数据库中身份信息等必要的测试数据。而第13行的“删除数据库”方法是测试环境清理代码，它的工作是在每个测试用例执行完毕后，将数据库删除，在执行下一个测试用例之前，MSTest会再调用“重新构建数据库”方法创建一个全新的数据库。这样操作的目的是保证每个测试用例都是在全新的数据库里运行，避免之前测试用例执行完毕后遗留的影响。

代码清单7 - 6 使用实体数据库时的测试准备和清理代码

|  |
| --- |
| 01 [TestInitialize]  02 public void 重新构建数据库()  03 {  04     var factory = new DesignTimeBandTradeContextFactory();  05     \_db = factory.CreateDbContext(null);  06     Assert.IsTrue(\_db.Database.EnsureCreated());  07  08     \_db.UserTradeApiSettings.Add(new UserTradeApiSetting("1", "1-secrete"));  09     \_db.SaveChanges();  10 }  11  12 [TestCleanup]  13 public void 删除数据库()  14 {  15     if (\_db != null)  16     {  17         Assert.IsTrue(\_db.Database.EnsureDeleted());  18         \_db.Dispose();  19         \_db = null;  20     }  21 } |

在执行多个用例的时候频繁构建删除数据库是一个很耗资源的操作，而且数据库软件独立于.NET环境，并不是很可控，笔者在编写前面的示例测试代码时碰到好几次删除数据库不成功导致测试集合失败的情况，而EntityFrameworks Core提供的内存模拟数据库就非常适用于我们的需要。使用下面的命令按照EF Core的内存模拟数据库：

dotnet add package Microsoft.EntityFrameworkCore.InMemory

为了支持使用内存模拟数据库做ORM映射，需要修改DbContext的一些代码实现，代码清单7 - 7列出了这些更改。第1）要增加一个接受DbContextOptions参数的构造方法，如第2行，这个构造方法用来在从依赖注入容器获取DbContext对象时，有机会配置为使用内存进行映射。由于使用了内存模拟数据库，也不需要太多配置，而且使用也是Linq to Object查询，对日志的支持也不足，所以第2）是修改OnConfiguring方法，当使用内存模拟数据库时，options参数的IsConfigured属性值为false，通过判断这个字段来选择是否使用真实数据库进行映射，如第10行的if语句的做法。

代码清单7 - 7 在DbContext中支持内存模拟数据库

|  |
| --- |
| // 源码位置：第7章\demo\TradeBot.Strategy.Band\Db\BandTradeContext.cs  01 public class BandTradeContext : DbContext {  02     public BandTradeContext(DbContextOptions<BandTradeContext> options) // 1)  03         : base(options) { }  04  05     /\* 省略其他代码 ……  \*/  06  07     protected override void OnConfiguring(  08         DbContextOptionsBuilder options)  09     {  10         if (!options.IsConfigured)  // 2)  11         {  12             options.UseSqlServer(\_config["db"])  13                    .UseLoggerFactory(\_loggerFactory);  14         }  15     }  16 } |

在初始化依赖注入容器的时候，第1）需要构建调用DbContext方法的options参数，构建时指定使用内存数据库，如第10行。第2）使用这个options对象构造要返回的DbContext对象，如13行，这一行也演示了在依赖注入容器获取构造新对象时其他依赖项（如IConfiguration和ILoggerFactory）实例的方法。

代码清单7 - 8 初始化依赖注入容器时配置使用内存数据库

|  |
| --- |
| // 源码位置：第7章\demo\TraderBot.Test\内存数据库中测试.cs  01 [ClassInitialize]  02 public static void 准备依赖注入环境(TestContext tc)  03 {  04     \_container = new ServiceCollection()  05         .AddLogging(config => config.AddDebug())  06         .AddSingleton<IConfiguration>(configuration)  07         .AddTransient<BandTradeContext>(sp =>  08         {  09             var options = new DbContextOptionsBuilder<BandTradeContext>()  10                 .UseInMemoryDatabase(databaseName: "bandmemorydb")  // 1  11                 .Options;  12  13             return new BandTradeContext(options,    // 2  14               sp.GetService<IConfiguration>(),sp.GetService<ILoggerFactory>());  15         })  16         .BuildServiceProvider();  17 } |

## Mock外部接口

在7.1.2小节的例子中，我们单独定义了一个模拟的FakeBroker类型，用来模拟交易所的不同行为，为了模拟不同的实时行情，FakeBroker采用委托的办法由测试用例自己设计满足测试场景的行情数据。IBroker的接口比较简单，所需实现的操作也较少，定义一个FakeBroker类型足够在测试工程里使用了，在一些更大型的软件项目中，一个测试场景可能涉及到实现多个接口，甚至是根据测试场景需要定义不同的接口实现类型，这种情况下，测试工程里会包含有很多测试类型，从代码维护的角度来说是灾难性的，我们可以使用模拟对象（Mock）技术来实现FakeBroker类型相似的效果。

.NET技术社区里有几个成熟的模拟对象包，其中流行的且支持.NET Core，截止到笔者写作时是Moq这个类库。一般Moq类库只会在测试工程中使用，进入测试工程所在的文件夹，并输入以下命令使用Moq：

dotnet add package Moq

或者直接在解决方案的文件夹中，通过指定测试工程的文件名添加Moq包，如以下的命令不仅指定了测试工程的位置，还同时指定了安装的Moq包的版本号：

dotnet add .\TraderBot.Test\TraderBot.Test.csproj package Moq --version 4.13.1

代码清单7 - 9中的测试用例演示了Moq的用法，与7.1.2小节的做法不同，这里不需要单独定义一个实现IBroker接口的类型，而是通过Moq在运行时实时定义伪造的类型并返回相应的实例：

1. 通过Mock泛型类直接创建IBroker的模拟对象，如14行。新创建的模拟对象同样也要针对接口的方法定义具体的实现，但跟FakeBroker类型需要实现IBroker接口的每个方法不同的是，用Moq模拟的对象只需要实现测试场景中会用到的接口方法就可以了。
2. Setup方法在新创建的模拟对象定义要实现的接口方法的行为，如第15行实现了IBroker的GetBalances方法，在Moq中实现接口方法的流程是向Setup方法传入描述接口方法调用方式的表达式树，Moq通过分析表达式树在运行时伪造接口方法。如果实现的接口方法有返回值，则需要使用Returns扩展方法设置伪造接口方法的返回数据，如第15行表明在测试过程中，调用模拟对象的GetBalances方法时，返回值包含两个币种的余额明细。
3. Setup方法还可以通过区分方法参数值来设置不同的实现，第20行在实现接口的GetOrderBook方法时通过指定具体的quote和coin参数来设置，只有测试用例在试图获取“BTC/USDT”交易对的行情时才使用这个方法实现，而获取其他交易对的行情时则不会使用这个实现。
4. 如果不需要根据方法的参数值设置不同的实现，可以使用It.IsAny方法来匹配任意参数值，如第25行Support方法的实现，Returns方法里返回的true，表明我们的模拟对象支持任意交易对。
5. Moq的Setup方法保存的是被实现的接口方法的表达式树，除了可以用来实现模拟方法的定义之外，Moq还会通过它记录模拟方法被调用的次数，因此在第28行把模拟的BuyLimit方法记录在一个表达式树变量中，以便在后面验证测试用例的执行结果时使用。
6. 当设置好模拟对象需要用到的接口方法实现后，通过mock.Object字段获取到实际的模拟对象，如第32行，这个对象在运行时由Moq按照前面Setup方法的规则实现IBroker接口，对于待测系统的其他代码来说，它跟正常的IBroker对象没有任何区别。
7. 在很多测试场景中，除了验证最终结果是否满足测试预期之外，对待测系统的执行过程进行验证也是很有必要的，以策略交易程序为例，当实时行情满足预定的交易策略条件时，我们只希望一条策略只会执行一次，而不是没有节制的重复下单。这种情况下，Moq内置的方法被调用次数统计就显得很有必要了，如第39行测试用例验证当策略条件满足时，BuyLimit方法只会被调用一次，即策略只会下单一次。
8. 正如前面提到的，系统其他模块将Moq模拟的对象当做正常的IBroker实例使用，因此在第46行可以在数据库里查询到第30行模拟BuyLimit方法返回的订单Id值。

代码清单7 - 9 使用Moq模拟IBroker接口

|  |
| --- |
| // 源码位置：第7章\demo\TraderBot.Test\使用Moq测试.cs  01 [TestMethod]  02 public void 验证使用实际数据库根据策略下达买单()  03 {  04     var 期望买单Id = 100001;  05     var site = "mock";  06     using (var context = \_container.GetService<BandTradeContext>())  07     {  08         context.BandStrategies.Add(new BandStrategy(  09             userId: 1, quote: "USDT", coin: "BTC",  10             bidPrice: 2000, tradeVolume: 1, askProfit: null, askPrice: 8000));  11         context.SaveChanges();  12     }  13  14     var mock = new Mock<IBroker>();                                     // 1)  15     mock.Setup(m => m.GetBalances()).Returns(new BalanceItem[]          // 2)  16     {  17         new BalanceItem { Currency = "USDT", Available = 10000},  18         new BalanceItem { Currency = "BTC", Available = 1},  19     });  20     mock.Setup(m => m.GetOrderBook("USDT", "BTC", 10)).Returns(         // 3)  21         开发数据库中测试.CreateOrderBook(site, "USDT", "BTC",  22             new decimal[,] { { 2000, 2 }, { 2001.88m, 1.2m } },  23             new decimal[,] { { 1999, 1 }, { 1998.99m, 2 } })  24         );  25     mock.Setup(m => m.Support(  26         It.IsAny<string>(), It.IsAny<string>())).Returns(true);         // 4)  27  28     Expression<Func<IBroker, object>> buyExpression =                   // 5)  29         m => m.BuyLimit("USDT","BTC",It.IsAny<decimal>(),It.IsAny<decimal>());  30     mock.Setup(buyExpression).Returns(期望买单Id);  31     var engine = new BandStrategyEngine(\_container);  32     var broker = mock.Object;                                           // 6)  33     mock.Setup(m => m.CreateAuthenticatedInstance(  34                         It.IsAny<string>(), It.IsAny<string>()))  35         .Returns(broker);  36  37     engine.Start("USDT", "BTC", broker);  38  39     mock.Verify(buyExpression, Times.Once);                             // 7)  40  41     // \_db是之前创建的，需要重新查询一下数据库获取最新数据  42     using (var context = \_container.GetService<BandTradeContext>())  43     {  44         var bo = context.DbPlacedOrders.FirstOrDefault();  45         Assert.IsNotNull(bo);  46         Assert.AreEqual(期望买单Id.ToString(), bo.BidOrderId);          // 8)  47     }  48 } |

# 交互界面

前面的示例代码均是简单的命令行程序，无法完成一个人机交互需求，本节综合前面章节的内容编写图形化人机交互界面 – 一个Web应用。

## ASP.NET Core MVC应用简介

Web应用是基于HTTP协议的，协议基础原理是客户端采用GET、POST、PUT和DELETE等几种方法中的一个向服务器发送请求，请求中包含一个服务路径。在早期互联网时代，这个服务路径通常是一个文件地址，服务器根据请求中的方法来处理服务路径指定的文件，例如GET方法表示读取路径指定的文件内容，DELETE方法则表示删除路径上的文件，POST和PUT则分别表示增加和修改对应路径上的文件，后两个方法通常情况下客户端会在HTTP请求包中附带一些数据，服务器接收之后修改文件内容。在这个早期时代，Web应用的编写也较为简单，服务器只需要解析HTTP请求中的路径并根据具体的方法执行命令即可，随着网络的逐步发展，Web应用越来越复杂，开发人员需要更加强大的编程手段来适应这些复杂的编程场景。因此目前主流Web应用开发采用的是“模型-视图-控制器”（Model-View-Controller），简称为MVC编程范式。在MVC里，HTTP请求中的路径被映射成面向对象编程中的类型和类型中的方法调用，这些处理HTTP请求的类型称为控制器类型，而具体处理请求的方法则称为功能（Action）。控制器对请求的处理结果采用视图的方式返回给客户端，在处理请求的过程中，会使用模型类型对请求和结果数据进行封装，其中控制器也会创建一些模型对象将封装好的数据传递给视图完成前端的呈现工作。而HTTP请求中的路径到控制器类型的映射由MVC框架提供的路由（routing）组件完成，这样将Web请求中的响应式操作流程转换成了面向对象编程的方式，大大方便了程序开发。在MVC应用中处理一个HTTP请求的流程一般是这样的：

1. MVC框架将HTTP请求消息通过路由组件确定需要实例化的控制器类型，并确定控制器中的处理消息的方法；其中HTTP请求消息附带的数据通常会由一个模型类型封装。
2. 控制器的方法处理完请求后，将处理结果由其他模型类型封装并传递给视图对象，视图中通常会包含一些服务器端代码处理模型对象，最后生成结果HTML返回给客户端浏览器。

ASP.NET Core MVC应用里，控制器类型都是继承自Controller基类，模型类型则由程序员视情况自己定义，视图则保存在使用Razor引擎的cshtml文件中，视图页面中C#代码嵌入在Html代码里，在运行时Razor引擎在服务器端编译和执行cshtml中嵌入的C#代码，并将执行结果和原始的Html代码合并返回到客户端的浏览器。

如图7 - 6就是一个典型的Razor页面的截图，可以看到文件名使用.cshtml作为后缀，在第1行使用**@model**指令告诉Razor引擎视图中使用的模型对象的类型是DbPlacedOrde的集合类型，控制器处理客户端请求后，会传递一个集合对象给Index.cshtml视图。在视图中，以**@**符号作为关键字，其包含的代码都是需要在服务器端执行的C#代码，有的C#代码不输出任何HTML代码，如第1行和第3行两行代码。有的C#代码则会输出HTML代码，如第11行和第14行分别会输出两个<label>标签，Razor引擎会根据控制器传递过来的模型对象执行视图里的C#代码，并将代码中输出的HTML代码与视图中原始的HTML代码合并，最终输出到客户的浏览器端。



图7 - 5 ASP.NET Core Razor页面

## 创建ASP.NET Core MVC应用

打开Visual Studio，在demo解决方案中添加一个ASP.NET core mvc应用，示例代码中工程命名为“TraderBot.Web”。ASP.NET Core应用支持寄宿在IIS中运行，也支持作为独立程序运行，甚至还支持寄宿在其他进程中运行。在Windows服务器部署ASP.NET Core应用的话，建议寄宿在IIS中运行，以便利用IIS的强大的进程监控、管理能功能。在Linux服务器部署ASP.NET Core，建议寄宿在Nginx或者Apache中。而在开发过程中，作为独立程序在命令行中启动更便于调试排错。

代码清单7 - 10 ASP.NET Core启动代码

|  |
| --- |
| 01 public static void Main(string[] args)  02 {  03     CreateHostBuilder(args).Build().Run();                       // 1)  04 }  05  06 public static IHostBuilder CreateHostBuilder(string[] args) =>  07     Host.CreateDefaultBuilder(args)                              // 2)  08         .ConfigureLogging(logging => logging.AddConsole())       // 3)  09         .ConfigureWebHostDefaults(webBuilder => {  10             webBuilder.UseStartup<Startup>();                    // 4)  11         }); |

代码清单7 - 10是Visual Studio的ASP.NET Core MVC模板自带的默认代码，笔者做了点简单修改：

1. 在ASP.NET Core中将Web应用进程封装成一个称做宿主（host）的对象，如果将网站寄宿在IIS、Apache等进程中运行，IIS、Apache等也被称为宿主进程，它们与这里的宿主对象不是相同的事物。ASP.NET Core中的宿主对象用来封装应用要使用到的资源，便于启动和关闭Web应用。封装的资源有：依赖注入环境、日志、配置和实现了IHostedService的对象。第3行的CreateHostBuilder方法用来配置宿主对象的资源，再通过Build方法构建宿主对象，最后Run方法启动Web应用。
2. CreateHostBuilder方法首先调用CreateDefaultBuilder方法使用默认配置创建初始的宿主对象，在这个基础上根据需要配置需要引入的资源。
3. 第8行向宿主对象添加日志配置，这行代码实际上是冗余的，这是因为默认的宿主配置就包含了向控制台、Debug窗口等地方输出日志。Visual Studio默认在Windows平台下使用IIS Express运行Web应用，日志输出不明显，其实ASP.NET Core在运行过程中，除了Web应用本身输出的日志，ASP.NET Core以及Entity Framework Core都会输出大量的日志便于开发人员排错，这些日志在使用命令行模式运行Web应用时会更明显。在Visual Studio中运行Web应用时，选择以工程名命名的调试配置就可以使用独立宿主——即命令行模式运行应用，如图7 - 6。

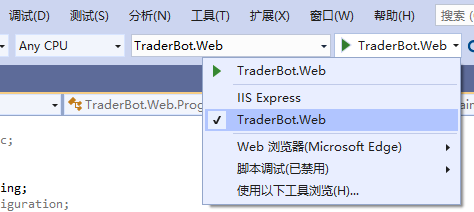


图7 - 6 使用命令行模式运行ASP.NET Core应用

1. 从ASP.NET Core 3.0开始，宿主对象的配置工作被拆分成两个步骤，分别由CreateDefaultBuilder和ConfigureWebHostDefaults方法完成。第一个方法用来初始化进程的配置文件、日志和依赖注入容器的配置。而第二个方法则用来完成构建一个ASP.NET Core应用所需要的其他步骤，例如使用程序员自定义的Startup类型配置依赖注入容器内的组件构建方法和生命周期，处理HTTP消息的中间件管道等。第10行使用的Startup类型也是MVC模板自带的，里面已经包含了诸如构建MVC编程模型、Web请求路由配置、权限管理、未捕获异常的默认处理页面以及静态文件的处理等构建Web应用必须的代码。

一般我们会修改Startup类型的ConfigurationServices方法来配置整个Web应用中的依赖注入逻辑。如代码清单7 - 11的配置代码，跟前面章节示例代码中配置的方式一致，配置了数据库DbContext、交易所类型的创建方式。第15行是ASP.NET Core MVC自带的方法，用来配置MVC基础架构相关的类库。

代码清单7 - 11 在ASP.NET Core中配置依赖注入逻辑

|  |
| --- |
| 01 public void ConfigureServices(IServiceCollection services)  02 {  03     services.AddTransient<BandTradeContext>();  04     services.AddSingleton<Huobi>();  05     services.AddSingleton<FakeBroker>();  06     services.AddTransient<BrokerFactoryDelegate>(serviceProvider => site =>  07         {  08             if (string.Compare(site, "huobi", true) == 0)  09                 return serviceProvider.GetService<Huobi>();  10             else if (string.Compare(site, "fake", true) == 0)  11                 return serviceProvider.GetService<FakeBroker>();  12             else  13                 throw new InvalidOperationException($"不支持的站点：{site}");  14         });  15     services.AddControllersWithViews();  16 } |

ASP.NET Core对依赖注入的支持是框架级别的，其很多组件服务都是通过依赖注入容器获取，例如控制器（Controller）类型有其他依赖时，ASP.NET Core自动在容器中解决依赖关系创建控制器实例。如代码清单7 - 12，TraderController需要使用到日志、配置和映射数据库的BandTradeContext等组件，还依赖创建交易所接口的BrokerFactoryDelegate类型，我们只需要把这些依赖作为参数定义在控制器的构造方法上，ASP.NET Core在路由组件定位到控制器类型后，会自动从依赖注入容器中解决这些依赖。

针对每个请求，ASP.NET Core MVC都会创建新的控制器实例对象，也就是说代码清单7 - 12中的构造方法都会被调用一次，如果并发请求很多的话，第4行的\_context对象会在内存中有很多的实例，在前面的章节中提到数据库链接是一个非托管对象，不及时释放链接的话会对数据库造成无谓的浪费，笔者建议在实际项目中不要直接注入类似BandTradeContext这样的数据库对象，而是注入创建数据库对象的委托，如创建交易所实例的BrokerFactoryDelegate方法。注入委托的方法还有一个好处就是，在控制器里可能需要多个数据库对象实例，直接注入数据库对象的话，没有办法创建新的数据库实例，而使用委托就可以解决这个问题。

代码清单7 - 12 ASP.NET Core使用依赖注入创建控制器实例

|  |
| --- |
| 01 private readonly ILogger<HomeController> \_logger;  02 private BrokerFactoryDelegate \_factory;  03 private IConfiguration \_config;  04 private BandTradeContext \_context;  05 public TradeController(BrokerFactoryDelegate factory, IConfiguration config,  06     BandTradeContext context, ILogger<HomeController> logger)  07 {  08     \_factory = factory;  09     \_config = config;  10     \_context = context;  11     \_logger = logger;  12 } |

代码清单7 - 13是处理下单界面的控制器代码，里面定义了两个Index的重载方法，第1行的Index是处理GET方法HTTP请求的控制器功能方法，它的参数可以通过URL的检索部分传入，如“/trade/?q=USDT&c=BTC”的GET请求就会被路由组件定位到第1行的Index方法，并将“USDT”和“BTC”分别传入其参数“q”和“c”。这里所有的参数都是可选的，因此在访问时，URL里也可以不附带任何参数，即GET “/trade”的请求也会被定位到这个Index方法。Index方法处理完请求后，需要向用户呈现交互视图， Controller类中的View方法就是用来返回视图的，如果需要在视图中显示数据，控制器创建一个模型对象 – 如第4行的TradeOrder对象传递给View方法，在.cshtml视图文件中就可以使用该模型对象进行针对性的显示处理，图7 - 8是最终呈现给用户的界面，注意地址栏上的URL中的检索参数，以及“交易对”和“数字币”两个文本框中的文字。



图7 - 7 下单处理界面

当用户在下单界面上输入自己的订单参数并点击提交时，浏览器会将这些信息打包到HTTP消息中并使用POST方法上传到服务器，在示例代码中处理这个消息的URL也是“/trade”，路由组件同样会定位到Index方法，然而这次是POST方法，路由会定位到标注了“HttpPost”特性的Action方法，即第14行的方法。方法标注了这个特性表示可以处理POST请求，而从客户端上传来的数据一般的做法是定义一个模型（TradeOrder）类来封装它们，MVC框架会自动将数据反序列化成相应的模型对象。

如果模型类型里定义了数据校验逻辑，MVC接收到客户端传来的数据，反序列化成对象并传递给功能（Action）方法之前，会执行这些校验逻辑，校验的结果保存在ModelState.IsValid变量里，如代码清单7 - 13第16行，上传的数据有效的话，我们就继续执行后面的交易处理逻辑。接下来先根据请求中的站点字段，通过\_factory委托创建交易所实例broker，再根据用户的指令确定是买单还是卖单，并下达订单。成功下单的话，就将这次下单以及订单Id保存到数据库中，否则就输出相应的错误消息到日志中。最后将下单结果通过ViewBag的ErrorMessage返回给视图页面，如第53行和第60行。ViewBag是Controller类专门用来向视图传递额外数据的动态对象，即它的类型是dynamic的，功能（Action）方法可以往里面放入任何类型的数据，由于其是动态的，所以可以跳过编译器的类型检查，很便于在包含大量Html代码的视图页面中使用。

代码清单7 - 13 处理下单交易的Action代码

|  |
| --- |
| 01 public IActionResult Index(string q = null, string c = null, string s = null,  02     decimal? p = null, decimal? v = null, int? d = null)  03 {  04     return View(new TradeOrder()  05     {  06         Quote = q, Coin = c, Site = s,  07         Side = d.HasValue && d.Value == (int)TradeSide.Sell ? "Sell" : "Buy",  08         Price = p,  09         Volume = v  10     });  11 }  12  13 [HttpPost]  14 public IActionResult Index(TradeOrder model)  15 {  16     if (ModelState.IsValid)  17     {  18         var broker = \_factory(model.Site);  19         var cfgbase = \_config["configdir"];  20         broker.Initialize(cfgbase);  21         decimal price = model.Price.Value;  22  23         object orderid = null;  24         string side = null;  25         if (model.Side == "Buy")  26         {  27             side = "买入";  28             orderid = broker.BuyLimit(  29                 model.Quote, model.Coin, model.Volume.Value, price);  30         }  31         else  32         {  33             side = "卖出";  34             orderid = broker.SellLimit(  35                 model.Quote, model.Coin, model.Volume.Value, price);  36         }  37  38         if (orderid != null)  39         {  40             var order = new DbPlacedOrder()  41             {  42                 BrokerSite = model.Site,  43                 Coin = model.Coin,  44                 TradingVolume = model.Volume.Value  45             };  46  47            // ...... 省略order赋值的代码 ......  48  49             \_context.DbPlacedOrders.Add(order);  50             \_context.SaveChanges();  51             var msg = "订单号：" + orderid;  52             \_logger.LogInformation(msg);  53             ViewBag.ErrorMessage = msg;  54             return View();  55         }  56         else  57         {  58             var msg = "执行失败，可能是网络原因，交易所返回数据太慢，请重试.....";  59             \_logger.LogWarning(msg);  60             ViewBag.ErrorMessage = msg;  61             return View();  62         }  63     }  64  65     ViewBag.ErrorMessage = "下单成功";  66     return View();  67 } |

## 部署ASP.NET Core MVC应用

前面提到过ASP.NET Core MVC有多种部署方式，寄宿在IIS、Apache和nginx等进程中运行，或者直接作为独立进程运行。本节我们采用独立进程的方式部署，其他寄宿部署方式请读者阅读官网文档：<https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/?view=aspnetcore-3.0>。

在Visual Studio的“解决方案资源管理器”中右键单击MVC工程，选择右键菜单中的“发布”，如果是第一次发布，会弹出 “选取发布目标”窗口，在窗口的左边列表中选择“文件夹”发布，最后点击“创建配置文件”按钮完成发布配置，窗口中有一个“选择文件夹”文本框用来更改最终的发布路径。后续发布时，“发布”右键菜单会列出所有可用的发布配置，选择一个配置点击“发布”按钮就可以完成项目的打包，再将发布文件夹中的文件整体拷贝到服务器上运行就可以了。

默认的发布采用的是框架依赖的部署方式，即假定部署的机器上已经安装了.NET Core运行环境。如果需要定制的话，可以在创建发布配置文件时弹出的“选取发布目标”窗口上点击“高级”链接，或者事后编辑配置文件修改配置，编辑界面如图7 - 9所示。

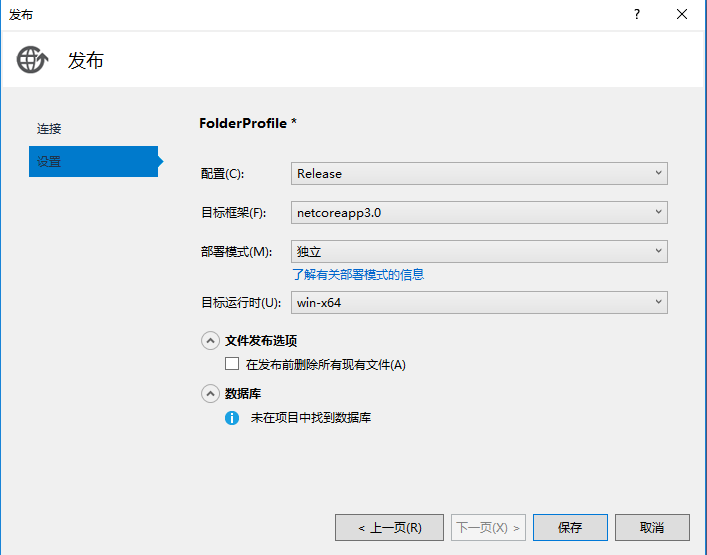


图7 - 8 定制发布配置

在部署配置文件编辑界面，“配置”下拉框用来选择发布时采用的编译配置，是“Debug”还是“Release”。“目标框架”下拉框设置运行程序使用的.NET框架版本。“部署模式”分为“框架依赖”和“独立”，前者需要目标部署机安装有相应的目标框架运行环境，或者则可以直接运行，读者可以根据实际情况选择。当“部署模式”设置为“独立”时，发布时会针对部署的操作系统打包依赖的库程序，如图7 - 9中选择的部署平台是Windows x64环境。

# 持续集成

当开发团队由多个成员组成，甚至是由多个子开发团队组成，为了尽早集成团队各个成员的工作，防止潜在的错误，很多开发团队都采用持续集成的做法来自动化构建并验证，保证交付质量并尽量做到随时交付。图7 - 5列出了持续集成开发的一般流程，持续集成系统会自动从代码仓库签出最新版本的代码，自动编译并执行自动化的单元测试，通过这些验证后进行自动化部署，如果是预备上生产的代码，则会执行集成和验收测试。最后的集成和验证测试步骤大部分团队采用手工执行，一些技术储备强的团队则会编写部分自动化的UI测试用例来验证。

持续集成.png

图7 - 9 持续集成流程

## Dotnet命令使用简介

.NET Core SDK的dotnet命令可以覆盖编译、单元测试和部署等多个过程，dotnet命令的使用语法包含四部分：主命令（dotnet本身）、子命令、子命令参数和选项。如下面的命令创建了一个新的C#工程：

dotnet new console –lang C#

这里面：

* dotnet是主命令；
* new是子命令；
* console是new子命令的参数；
* -lang则是选项。

dotnet的help子命令可以列出所有支持的子命令，表7 - 1列出了在持续集成环境中经常用到的子命令的作用，更详细的子命令说明请读者阅读文档：<https://docs.microsoft.com/en-us/dotnet/core/tools/?tabs=netcore2x>。读者要也可以输入“dotnet help 子命令”的方式查看某一个子命令的使用文档，如“dotnet help new”。

表7 - 1 持续集成中常用的dotnet子命令

|  |  |
| --- | --- |
| **命令** | **说明** |
| build | 编译工程或者解决方案，其“**-c**”选项可以选择编译采用的配置，即“**Debug**”和“**Release**”配置，默认使用的是“**Debug**”配置，程序员也可以在工程里定义新的配置并通过这个选项编译。  使用示例（编译demo.sln解决方法，使用发布配置）：  dotnet build demo.sln -c Release  或者进入demo解决方案的文件夹，执行命令（使用调试配置）：  dotnet build |
| clean | 清理工程中所有编译输出的文件，用法与build类似，只是作用是相反，其也支持“**-c**”选项指定要清理的配置输出。  使用示例：  dotnet clean demo.sln -c Release  或者进入demo解决方案的文件夹，执行：  dotnet clean |
| restore | 还原工程中依赖的装配件和NuGet包。在前面的示例中，经常用到“dotnet add package”命令给工程添加包引用， dotnet命令从[http://www.nuget.org](http://www.nuget.org/)下载被引用的最新版本的包，并将引用保存在csproj文件里。当程序员将代码签入到代码仓库时，通常不会签入下载的包，一方面是避免代码仓库同步时间过长，一方面是避免团队成员用不同版本的包覆盖了原始文件导致编译失败的问题。在持续集成或者团队其他成员同步项目源码时，执行“restore”子命令从NuGet包仓库中根据csproj文件中记录的版本号同步包引用。  使用示例：  dotnet restore |
| run | 编译并运行工程。 |
| test | 运行工程中的单元测试。如果所有的测试执行通过，则返回0，否则返回1表示至少有一个测试用例失败。  在持续集成系统中，会调用dotnet的相应子命令来逐步执行图7 - 5中列出的自动化步骤，系统会判断每个子命令的返回值来判断步骤是否成功执行，与test子命令类似，其他子命令都采用这种方式报告运行结果。 |
| publish | 打包编译输出的程序到一个文件夹中，或者部署到远程的服务器上，前文使用Visual Studio IDE发布MVC应用时实际上也是调用该子命令完成的。如果执行命令的当前目录有.csproj或者.sln文件的话，不带参数执行就是发布相应的工程或者解决方案，否则需要指明文件的路径，该子命令有几个关键的参数：  **-c**：指明发布使用的编译配置。  **-f：**指明部署的目标框架，此参数对应图7 - 9中的“目标框架”，.NET在二十年的历史中发布了很多版本，每个版本都有名字，截止本书写作时，最新.NET Core的版本是3.0，相应名字是“netcoreapp3.0”，而.NET框架的版本是4.8，名字则是“net48”。完整的目标框架列表请阅读：<https://docs.microsoft.com/en-us/dotnet/standard/frameworks>。  **-r：**设置发布的目标运行时，对应图7 - 9中的“目标运行时”。随着.NET特别是.NET Core对多种操作系统和平台的支持，可选的运行时矩阵也越来越大。完整列表请阅读：<https://docs.microsoft.com/en-us/dotnet/core/rid-catalog>。  **--self-contained：**指明是否框架依赖，对应图7 - 9中的“部署模式”下拉框。  使用示例：  dotnet publish -f netcoreapp3.0 -r linux-x64 --self-contained  上面的示例命令将7.2.2节中的工程以独立模式的部署，使用.NET Core 3.0作为目标框架，目标平台是linux x64，成功编译发布后，文件保持在工程目录的位置：TraderBot.Web\bin\Debug\netcoreapp3.0\linux-x64\publish。将publish文件夹打包上传到linux服务器上，在服务器上解压后，可执行文件通常以工程名命名——即TraderBot.Web（文件没有后缀名），使用**chmod**为程序加上可执行权限：  chmod +x TraderBot.Web  接下来在服务器上执行运行程序启动网站应用：  ./TraderBot.Web |

## 可持续集成示例

截止到目前，市面上有不少优秀的持续集成软件，这里采用笔者最熟悉的持续集成软件Jenkins为例演示.NET Core项目中的持续集成方法。Jenkins是流行的开源持续集成软件，持续集成项目的开发步骤基本类似，读者使用的是其他平台的话，只要稍微改动本书中的集成步骤应该就可以顺畅使用：

1. 首先在Jenkins官网下载软件：<https://jenkins.io/download/>，这是一个Java编写的软件，需要先安装JDK 8.0以上的开发环境，JDK环境请读者自行到Oracle官网下载。
2. Jenkins自带Web界面便于用户配置持续集成的步骤，整个系统都打包在一个war包里，可以直接使用java命令运行启动：

java -jar jenkins.war

1. Jenkins启动后，在命令行中会输出不少的日志，其中有一些重要的信息，如网站监听的端口号，主目录和初始管理员密码等信息，代码清单7 - 14列出了启动日志的摘要，其中关键日志笔者加粗显示，可以看到网站访问的端口号是8080，而初始管理员密码是bd8f62bb9439428798cb689f74231c89。

代码清单7 - 14 jenkins输出日志

|  |
| --- |
| webroot: $user.home/.jenkins  2019-11-18 08:09:27.349+0000 [id=1] INFO o.e.j.server.session.HouseKeeper#startScavenging: node0 Scavenging every 660000ms  **Jenkins home directory: C:\Users\Administrator\.jenkins** found at: $user.home/.jenkins  2019-11-18 08:09:28.532+0000 [id=1] INFO o.e.j.server.AbstractConnector#doStart: Started **ServerConnector@6f10d5b6{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}**  2019-11-18 08:09:28.532+0000 [id=1] INFO org.eclipse.jetty.server.Server#doStart: Started @10199ms  2019-11-18 08:09:38.852+0000 [id=31] INFO jenkins.install.SetupWizard#init:  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Jenkins initial setup is required. An admin user has been created and a password generated.  Please use the following password to proceed to installation:  **bd8f62bb9439428798cb689f74231c89**  This may also be found at: **C:\Users\Administrator\.jenkins\secrets\initialAdminPassword**  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  2019-11-18 08:10:41.846+0000 [id=31] INFO jenkins.InitReactorRunner$1#onAttained: Completed initialization  2019-11-18 08:10:41.918+0000 [id=22] INFO hudson.WebAppMain$3#run: **Jenkins is fully up and running** |

1. 打开浏览器访问“http://localhost:8080”,并在打开的页面输入前面日志里输出的初始管理员密码解锁系统，并使用默认配置安装相关插件，设置好网站管理员用户名和密码，完成最后的安装。
2. 安装后在主页点击“create new jobs”创建新的持续集成任务，或者直接访问url：<http://localhost:8080/newJob>。

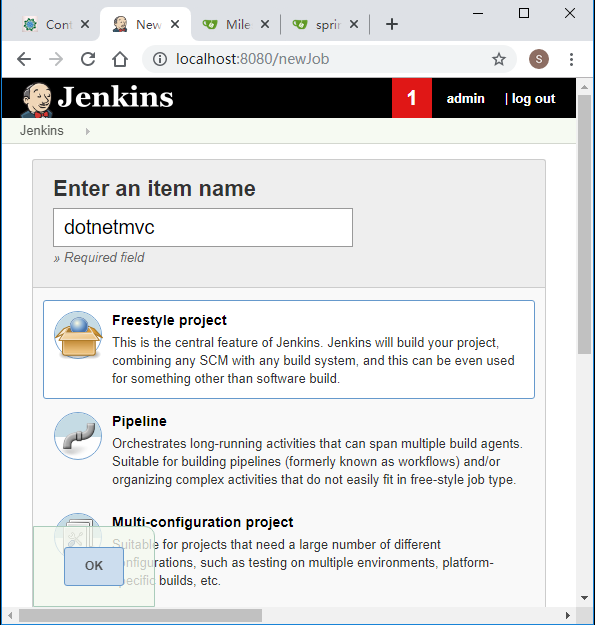


图7 - 10 Jenkins创建构建任务首界面

1. 配置界面上分几大块内容，每个配置项的右边基本上：
   1. 最上面的是一般性的配置，如配置介绍，是否需要保存前面的编译输出等。
   2. 源码管理信息，设置版本控制是git还是svn，选中其中一个会展开对应版本控制软件的详细设置界面，如代码仓库的链接等。在执行持续构建任务时，jenkins会自动从该链接下载最新的代码编译。
   3. 构建的触发器，如是周期性的构建，还是由外部脚本触发等等。
   4. 构建的环境设置。
   5. 构建脚本，可以由多个步骤串行组成。对于.NET工程的构建，将步骤放在一个批处理中执行。如果批处理文件路径包含空格，需要使用双引号包含起来，如图7 - 11。代码清单7 - 15是示例工程中笔者采用的批处理。这个批处理先进入源码文件夹，依次执行编译、单元测试和发布操作。
   6. 构建完成后要执行的收尾工作，例如给开发团队发送邮件通知构建情况等。

代码清单7 - 15 .NET构建批处理脚本

|  |
| --- |
| cd /d "D:\C# Programming Magic\sample-code\第7章\demo\"  dotnet build  pushd TraderBot.Test  dotnet test  popd  pushd TraderBot.Web  dotnet publish -f netcoreapp3.0 -r linux-x64 --self-contained  popd |

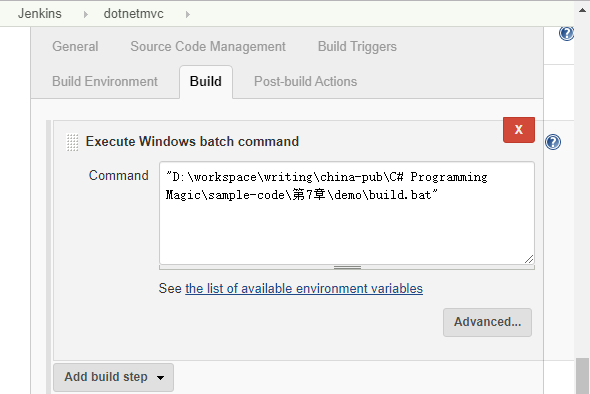


图7 - 11 设置构建批处理脚本路径

配置好构建任务后，可以在任务页面的左侧点击“Build Now”启动构建。Jenkins会保存每次构建的命令行输出，便于程序员在时候排错，每次构建的输出都以构建历史的形式保存，在任务页面的左下角会显示“Build History”（构建历史）列表，点击每次构建的序列号就可以查看构建任务执行时的命令行输出，如。

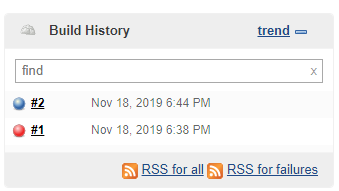


图7 - 12 构建历史记录列表

# 小结

到目前为止，笔者已经介绍了使用.NET技术编写商业化项目至少需要了解的知识，后面几章将介绍优化.NET应用需要了解的知识，如多线程、并行编程等。