Backtrader 使用指南

1. 总览

1.1 Backtrader 主要模块

Backtrader主要由以下几个数据模块组成：

- **DataFeeds 数据模块**
  - CSVDataBase 导入CSV
  - PandasData 导入
  - YahooFinanceData 导入网站数据
- **Broker 经纪商模块**
  - cash 初始资金
  - commission 手续费
  - slippage 滑点
- **Orders 订单模块**
  - `buy()` 买入
  - `sell()` 卖出
  - `close()` 平仓
  - `cancel()` 取消订单
- **Sizers 仓位模块**
- **Strategy 策略模块**
  - `next()` 主策略函数
  - notify_order, notify_trade 打印订单、交易信息
- **Indicators 指标模块**
  - SMA, EMA 移动均线
  - Ta-lib 技术指标库
- **Cerebro 回测执行模块**
  - `adddata()` 输入数据
  - `addsizer()` 添加仓位
  - `addstrategy()` 添加策略
  - `addanalyzer()` 添加分析器
  - `addobserver()` 添加观察器
  - `run()` 启动回测
  - `plot()` 可视化
- **Analyzers 策略分析模块**
  - AnnualReturn 年化收益
  - SharpeRatio 夏普比率
  - DrawDown 回撤
  - PyFolio 分析工具
- **Observers 观察器模块**
  - Broker 资金市值曲线

- Trades 盈亏走势
- BuySell 买卖信号

1.2 基本模版

要使用backtrader进行回测，首先要准备回测数据并编写策略；在实例化`cerebro = Cerebro()`的基础上，设置回测参数和绩效分析指标；最终运行回测`cerebro.run()`，获取回测数据。

下面是一个最基本的使用模版，从中可以反映出backtrader的使用流程。

```
import backtrader as bt # 导入 Backtrader
import backtrader.indicators as btind # 导入策略分析模块
import backtrader.feeds as btfeeds # 导入数据模块

# 创建策略
class TestStrategy(bt.Strategy):
    # 可选，设置回测的可变参数：如移动均线的周期
    params = (
        (...,...), # 最后一个“,”最好别删！
    )
    def log(self, txt, dt=None):
        '''可选，构建策略打印日志的函数：可用于打印订单记录或交易记录等'''
        dt = dt or self.datas[0].datetime.date(0)
        print('%s, %s' % (dt.isoformat(), txt))

    def __init__(self):
        '''必选，初始化属性、计算指标等'''
        pass

    def notify_order(self, order):
        '''可选，打印订单信息'''
        pass

    def notify_trade(self, trade):
        '''可选，打印交易信息'''
        pass

    def next(self):
        '''必选，编写交易策略逻辑'''
        sma = btind.SimpleMovingAverage(...) # 计算均线
        pass

# 实例化 cerebro
cerebro = bt.Cerebro()
# 通过 feeds 读取数据
data = btfeeds.BacktraderCSVData(...)
# 将数据传递给 “大脑”
cerebro.adddata(data)
# 通过经纪商设置初始资金
cerebro.broker.setcash(...)
# 设置单笔交易的数量
cerebro.addsizer(...)
```

```
# 设置交易佣金
cerebro.broker.setcommission(...)
# 添加策略
cerebro.addstrategy(TestStrategy)
# 添加策略分析指标
cerebro.addanalyzer(...)
# 添加观测器
cerebro.addobserver(...)
# 启动回测
cerebro.run()
# 可视化回测结果
cerebro.plot()
```

2. 数据处理和调用

2.1 数据馈送对象

数据馈送对象（Data Feed）指的是数据表格或数据表格集合，给策略有序提供数据以及数据的索引位置。

Backtrader提供了各种有效的数据索引方法：

- 带中括号的常规方式：`self.datas[X]`；
- 不带中括号的缩写方式：`self.dataX`；
- 使用负向索引位置编号，仅有`self.datas[-1]`；
- 默认指向第一个导入的数据集，即`self.data`等价于`self.datas[0]`或者`self.data0`；
- 通过表格名称调用数据：`self.getdatabyname('stockN')`，名称在导入数据时通过`name`参数设置。

下面是一个示例：

```
class TestStrategy(bt.Strategy):
    def __init__(self):
        # 打印数据集和数据集对应的名称
        print("-------------self.datas-------------")
        print(self.datas)
        print("-------------self.data-------------")
        print(self.data._name, self.data) # 返回第一个导入的数据表格，缩写形式
        print("-------------self.data0-------------")
        print(self.data0._name, self.data0) # 返回第一个导入的数据表格，缩写形式
        print("-------------self.datas[0]-------------")
        print(self.datas[0]._name, self.datas[0]) # 返回第一个导入的数据表格，常规形式
        print("-------------self.datas[1]-------------")
        print(self.datas[1]._name, self.datas[1]) # 返回第二个导入的数据表格，常规形式
        print("-------------self.datas[-1]-------------")
        print(self.datas[-1]._name, self.datas[-1]) # 返回最后一个导入的数据表格
        print("-------------self.datas[-2]-------------")
        print(self.datas[-2]._name, self.datas[-2]) # 返回倒数第二个导入的数据表格

cerebro = bt.Cerebro()
st_date = datetime.datetime(2019,1,2)
ed_date = datetime.datetime(2021,1,28)
# 添加 600466.SH 的行情数据
```

```
datafeed1 = bt.feeds.PandasData(dataname=data1,
                                fromdate=st_date,
                                todate=ed_date)
cerebro.adddata(datafeed1, name='600466.SH')
# 添加 603228.SH 的行情数据
datafeed2 = bt.feeds.PandasData(dataname=data2,
                                fromdate=st_date,
                                todate=ed_date)
cerebro.adddata(datafeed2, name='603228.SH')
cerebro.addstrategy(TestStrategy)
rasult = cerebro.run()
```

2.2 数据的导入

Backtrader 支持导入各式各样的数据：第三方网站加载数据（Yahoo、VisualChart、Sierra Chart、Interactive Brokers 盈透、OANDA、Quandl）、CSV 文件、Pandas DataFrame、InfluxDB、MT4CSV 等，其中最基础或最常见的就是导入 CSV 和导入 DataFrame了。

导入数据大致分 2 步：首先调用 DataFeeds 模块中的方法读取数据，其次将读取的数据传给大脑。

以下是一个示例：

```
# 读取和导入 CSV 文件
data = bt.feeds.GenericCSVData(dataname='filename.csv', ...)
cerebro.adddata(data, name='XXX')
# 读取和导入 dataframe 数据框 - 方式1
data = bt.feeds.PandasData(dataname=df, ...)
cerebro.adddata(data, name='XXX')
# 读取和导入 dataframe 数据框 - 方式2
data = bt.feeds.PandasDirectData(dataname=df, ...)
cerebro.adddata(data, name='XXX')

# 以 GenericCSVData 为例进行参数说明（其他导入函数参数类似）
bt.feeds.GenericCSVData(
    dataname='daily_price.csv', # 数据源，CSV文件名 或 Dataframe对象
    fromdate=st_date, # 读取的起始时间
    todate=ed_date, # 读取的结束时间
    nullvalue=0.0, # 缺失值填充
    dtformat=('%Y-%m-%d'), # 日期解析的格式
    # 下面是数据表格默认包含的 7 个指标，取值对应指标在 daily_price.csv 中的列索引位置
    datetime=0, # 告诉 GenericCSVData， datetime 在 daily_price.csv 文件的第1列
    high=3,
    low=4,
    open=2,
    close=5,
    volume=6,
    openinterest=-1) # 如果取值为 -1 , 告诉 GenericCSVData 该指标不存在
```

如果你觉得每次都要设置这么多参数来告知指标位置很麻烦，那你也可以重新自定义数据读取函数，自定义的方式就是继承数据加载类 GenericCSVData、PandasData 再构建一个新的类，然后在新的类里统一设置参数。

自定义的函数，不会修改 Backtrader 底层的数据表格内 lines 的排列规则，只是规定了一个新的数据读取规则。调用这个函数，就按函数里设置的规则来读数据，而不用每次都设置一堆参数：

```
class My_CSVData(bt.feeds.GenericCSVData):
    params = (
    ('fromdate', datetime.datetime(2019,1,2)),
    ('todate', datetime.datetime(2021,1,28)),
    ('nullvalue', 0.0),
    ('dtformat', ('%Y-%m-%d')),
    ('datetime', 0),
    ('time', -1),
    ('high', 3),
    ('low', 4),
    ('open', 2),
    ('close', 5),
    ('volume', 6),
    ('openinterest', -1)
)

cerebro = bt.Cerebro()
data = My_CSVData(dataname='daily_price.csv')
cerebro.adddata(data, name='600466.SH')
rasult = cerebro.run()
```

在回测时，除了常规的高开低收成交量这些行情数据外，还会用到别的指标，比如选股回测时会用到很多选股因子（PE、PB 、PCF、.....），可以给数据表格新增列，也就是给数据表格新增 line。以导入 DataFrame 为例，在继承原始的数据读取类 bt.feeds.PandasData 的基础上，设置 lines 属性和 params 属性，新的 line 会按其在 lines 属性中的顺序依次添加进数据表格中，具体对照下面例子的输出部分：

```
class PandasData_more(bt.feeds.PandasData):
    lines = ('pe', 'pb', ) # 要添加的线
    # 设置 line 在数据源上的列位置
    params=(
        ('pe', -1),
        ('pb', -1),
           )
    # -1表示自动按列明匹配数据，也可以设置为线在数据源中列的位置索引 (('pe',6),('pb',7),)
class TestStrategy(bt.Strategy):
    def __init__(self):
        print("--------- 打印 self.datas 第一个数据表格的 lines ----------")
        print(self.data0.lines.getlinealiases())
        print("pe line:", self.data0.lines.pe)
        print("pb line:", self.data0.lines.pb)

data1['pe'] = 2 # 给原先的data1新增pe指标（简单的取值为2）
data1['pb'] = 3 # 给原先的data1新增pb指标（简单的取值为3）
# 导入的数据 data1 中
cerebro = bt.Cerebro()
st_date = datetime.datetime(2019,1,2)
ed_date = datetime.datetime(2021,1,28)
datafeed1 = PandasData_more(dataname=data1,
```

```
                            fromdate=st_date,
                            todate=ed_date)
cerebro.adddata(datafeed1, name='600466.SH')
cerebro.addstrategy(TestStrategy)
result = cerebro.run()
```

2.3 数据中的行和列

Backtrader 数据表格的行，可以看做是蜡烛图中的一个个 bar ，只不过这个 bar 包含的信息并不局限于“高开低收”4 个指标，可以指向在这个时间点上的所有信息。回测其实就是按时间先后顺序依次循环遍历各个带有历史行情信息的 bar，检验策略在历史行情上的表现。

Backtrader 将数据表格的列拆成了一个个 line 线对象，一列就是一个指标，就是该指标的时间序列，就是一条线 line。默认情况下，数据表格要包含 7 个字段：`datetime`、`open`、`high`、`low`、`close`、`volume`、`openinterest` ，回测过程中用到的时间序列行情数据可视化后就是一条条曲线，一个字段就对应了一条条线。

因为可以将 Data Feed 对象看做是数据表格，而表格中又包含列，所以每一个 Data Feed 对象都有一个 lines 属性。可以将 lines 属性看作是 line 的集合，所以想要调用具体的某一条线，就通过 lines 属性来调用：

- 访问 lines 属性`xxx.lines`，可简写成 `xxx.l`；
- 访问 lines 属性中具体某条线：`xxx.lines.name`，可简写成 `xxx.lines_name`；
- 套用“先调用某张数据表格，再调用这张表格中具体的某根 line”的逻辑依次编写代码；
- 可以通过 `getlinealiases()` 方法查看 Data Feed 对象包含哪些线。

```
# 访问第一个数据集的 close 线
self.data.lines.close # 可省略 lines 简写成：self.data.close
self.data.lines_close # 可省略 lines 简写成：self.data_close
# 访问第二个数据集的 open 线
self.data1.lines.close # 可省略 lines 简写成：self.data1.close
self.data1.lines_close # 可省略 lines 简写成：self.data1_close
# 注：只有从 self.datas 调用 line 时可以省略 lines，调用 indicators 中的 line 时不能省略
```

也可以通过索引位置的办法，X 对应数据表格在数据表格集合中的索引位置，Y 对应线在数据表格中的索引位置：

- 完整形式：`self.datas[X].lines[Y]`
- 简写形式：`self.dataX.lines[Y]`, `self.dataX_Y`

进一步还可以提取line上面数据点，同样可以根据索引位置访问需要的数据：

- 索引规则：索引位置编号结合了时间信息，0 号位置永远指向当前时间点的数据，-1 号位置指向前一个时间点的数据，然后依次回退 （backwards）-2、-3、-4、-5、……；1 号位置指向下一天的数据，然后依次向前（forwards）2、3、4、……；
- 切片方法：`get(ago=0, size=1)` 函数，其中 ago 对应数据点的索引位置，即从 ago 时间点开始往前取 size 个数据点。默认情况下是取当前最新时点（ago=0）的那一个数据（size=1）。

值得注意的是，对数据点的索引切片操作一般在 `next()` 函数中涉及较多，而 `__init__()` 中涉及较少，因为 `__init__()` 中一般是对一整条 line 进行操作。这是一个示例：

```
class TestStrategy(bt.Strategy):
    def __init__(self):
        self.count = 0 # 用于计算 next 的循环次数
        # 打印数据集和数据集对应的名称
        print("------------- init 中的索引位置-------------")
        print("0 索引：",'datetime',self.data1.lines.datetime.date(0),
'close',self.data1.lines.close[0])
        print("-1 索引：",'datetime',self.data1.lines.datetime.date(-1),'close',
self.data1.lines.close[-1])
        print("-2 索引",'datetime', self.data1.lines.datetime.date(-2),'close',
self.data1.lines.close[-2])
        print("1 索引：",'datetime',self.data1.lines.datetime.date(1),'close',
self.data1.lines.close[1])
        print("2 索引",'datetime', self.data1.lines.datetime.date(2),'close',
self.data1.lines.close[2])
        print("从 0 开始往前取3天的收盘价：", self.data1.lines.close.get(ago=0, size=3))
        print("从-1开始往前取3天的收盘价：", self.data1.lines.close.get(ago=-1, size=3))
        print("从-2开始往前取3天的收盘价：", self.data1.lines.close.get(ago=-2, size=3))
        print("line的总长度：", self.data1.buflen())

    def next(self):
        print(f"------------- next 的第{self.count+1}次循环 --------------")
        print("当前时点（今日）：",'datetime',self.data1.lines.datetime.date(0),'close',
self.data1.lines.close[0])
        print("往前推1天（昨日）：",'datetime',self.data1.lines.datetime.date(-1),'close',
self.data1.lines.close[-1])
        print("往前推2天（前日）", 'datetime',self.data1.lines.datetime.date(-2),'close',
self.data1.lines.close[-2])
        print("前日、昨日、今日的收盘价：", self.data1.lines.close.get(ago=0, size=3))
        print("往后推1天（明日）：",'datetime',self.data1.lines.datetime.date(1),'close',
self.data1.lines.close[1])
        print("往后推2天（明后日）", 'datetime',self.data1.lines.datetime.date(2),'close',
self.data1.lines.close[2])
        print("已处理的数据点：", len(self.data1))
        print("line的总长度：", self.data0.buflen())
        self.count += 1

cerebro = bt.Cerebro()
st_date = datetime.datetime(2019,1,2) # 起始时间
ed_date = datetime.datetime(2021,1,28) # 结束时间
datafeed1 = bt.feeds.PandasData(dataname=data1,
                                fromdate=st_date,
                                todate=ed_date)
cerebro.adddata(datafeed1, name='600466.SH')
datafeed2 = bt.feeds.PandasData(dataname=data2,
                                fromdate=st_date,
                                todate=ed_date)
cerebro.adddata(datafeed2, name='603228.SH')
cerebro.addstrategy(TestStrategy)
rasult = cerebro.run()
```

在涉及到`datetime`线的操作中，默认存的是数字形式的时间，可以通过一定的方式转换：

- 可以通过 `bt.num2date()` 方法将其转为“xxxx-xx-xx xx:xx:xx”这种形式；
- 对 datetime 线进行索引时，`xxx.date(X)` 可以直接以“xxxx-xx-xx xx:xx:xx”的形式返回，X 就是索引位置

除此以外，如果需要获取总的或者已经回测的line长度，可以选择：

- `self.data0.buflen()` 返回整条线的总长度，固定不变；
- 在 `next()` 中调用 `len(self.data0)`，返回的是当前已处理（已回测）的数据长度。

3. 指标计算

3.1 指标的基本计算和调用

在编写策略时，除了常规的高开低收成交量等行情数据外，还会用到各式各样的指标（变量），比如宏观经济指标、基本面分析指标、技术分析指标、另类数据等等。除了前面描述的直接通过 DataFeeds 模块导入已经计算好的指标，在编写策略时还可以调用 Indicators 指标模块临时计算指标，比如 5 日均线、布林带等 。

回顾基本流程，发现只有在编写策略Strategy 时才会涉及到指标的计算和使用，而且是 Strategy 中的 `__init__()` 和 `next()` 方法涉及的最多。因此建议在 `__init__()`中一次性提前计算指标，并在`next()` 会每个交易日依次循环调用已经算好的指标，避免指标的重复计算，提高回测运行速度。下面是最基本的例子：

```
import backtrader.indicators as btind # 导入策略分析模块

class MyStrategy(bt.Strategy):
  # 先在 __init__ 中提前算好指标
    def __init__(self):
        sma1 = btind.SimpleMovingAverage(self.data)
        ema1 = btind.ExponentialMovingAverage()
        close_over_sma = self.data.close > sma1
        close_over_ema = self.data.close > ema1
        sma_ema_diff = sma1 - ema1
        # 生成交易信号
        buy_sig = bt.And(close_over_sma, close_over_ema, sma_ema_diff > 0)
    # 在 next 中直接调用计算好的指标
    def next(self):
        if buy_sig:
            self.buy()
```

计算指标时，有不同的简写形式，默认是对`self.datas`数据对象中的第一张表格中的第一条`line`（默认第一条line是 close）计算相关指标。调用指标时，也有不同的简写方式， 在`next()` 中调用当前时刻指标值时，可以省略索引`[0]`：`self.sma5[0]`就是`self.sma5`，就是`self.data.close[0]`，也就是`self.data.close`。

下面是一个例子：

```
class TestStrategy(bt.Strategy):

    def __init__(self):
        self.sma5 = btind.SimpleMovingAverage(period=5) # 5日均线
        self.sma10 = btind.SimpleMovingAverage(period=10) # 10日均线
        self.buy_sig = self.sma5 > self.sma10 # 5日均线上穿10日均线
```

```
    def next(self):
        # 提取当前时间点
        print('datetime', self.datas[0].datetime.date(0))
        # 打印当前值
        print('close', self.data.close[0], self.data.close)
        print('sma5', self.sma5[0], self.sma5)
        print('sma10', self.sma10[0], self.sma10)
        print('buy_sig', self.buy_sig[0], self.buy_sig)
        # 比较收盘价与均线的大小
        if self.data.close > self.sma5:
            print('------收盘价上穿5日均线------')
        if self.data.close[0] > self.sma10:
            print('------收盘价上穿10日均线------')
        if self.buy_sig:
            print('------ buy ------')

cerebro = bt.Cerebro()
st_date = datetime.datetime(2019,1,2)
end_date = datetime.datetime(2021,1,28)
datafeed1 = bt.feeds.PandasData(dataname=data1, fromdate=st_date, todate=end_date)
cerebro.adddata(datafeed1, name='600466.SH')
cerebro.addstrategy(TestStrategy)
result = cerebro.run()
```

3.2 指标的运算关系和符号

在计算指标或编写策略逻辑时，离不开算术运算、关系运算、逻辑运算、条件运算……，为了更好的适用于 Backtrader 框架的语法规则，backtrader对一些常用的运算符做了优化和改进。在 `__init__` 中事先通过 `bt.And`、`bt.Or`、`bt.If`、`bt.All`、`bt.Any`、`bt.Max`、`bt.Min`、`bt.Sum` 计算返回的结果与在 `next()` 中对当前时点通过常规 python 运算语法返回的结果是一致的。

经过优化之后，在`__init__` 中调用这些函数是基于整条 line 进行运算，返回的结果也是 lines ，能在 `next ()` 中循环调用。以 `bt.Max(self.data, self.sma10, self.sma5)` 为例，`bt.Max` 函数会站在3 条 line 的相同时间节点上求出最大值（各个横截面上求最大值），返回的结果就是由各个时间节点上最大值组成的 line ：

```
class TestStrategy(bt.Strategy):

    def __init__(self):
        self.sma5 = btind.SimpleMovingAverage(period=5) # 5日均线
        self.sma10 = btind.SimpleMovingAverage(period=10) # 10日均线
        # bt.And 中所有条件都满足时返回 1；有一个条件不满足就返回 0
        self.And = bt.And(self.data>self.sma5, self.data>self.sma10, self.sma5>self.sma10)
        # bt.Or 中有一个条件满足时就返回 1；所有条件都不满足时返回 0
        self.Or = bt.Or(self.data>self.sma5, self.data>self.sma10, self.sma5>self.sma10)
        # bt.If(a, b, c) 如果满足条件 a，就返回 b，否则返回 c
        self.If = bt.If(self.data>self.sma5,1000, 5000)
        # bt.All,同 bt.And
        self.All = bt.All(self.data>self.sma5, self.data>self.sma10, self.sma5>self.sma10)
        # bt.Any，同 bt.Or
        self.Any = bt.Any(self.data>self.sma5, self.data>self.sma10, self.sma5>self.sma10)
```

```
        # bt.Max，返回同一时刻所有指标中的最大值
        self.Max = bt.Max(self.data, self.sma10, self.sma5)
        # bt.Min，返回同一时刻所有指标中的最小值
        self.Min = bt.Min(self.data, self.sma10, self.sma5)
        # bt.Sum，对同一时刻所有指标进行求和
        self.Sum = bt.Sum(self.data, self.sma10, self.sma5)
        # bt.Cmp(a,b), 如果 a>b ，返回 1；否则返回 -1
        self.Cmp = bt.Cmp(self.data, self.sma5)

    def next(self):
        print('---------- datetime',self.data.datetime.date(0), '------------------')
        print('close:', self.data[0], 'ma5:', self.sma5[0], 'ma10:', self.sma10[0])
        print('close>ma5',self.data>self.sma5, 'close>ma10',self.data>self.sma10, 'ma5>ma10',
self.sma5>self.sma10)
        print('self.And', self.And[0], self.data>self.sma5 and self.data>self.sma10 and
self.sma5>self.sma10)
        print('self.Or', self.Or[0], self.data>self.sma5 or self.data>self.sma10 or
self.sma5>self.sma10)
        print('self.If', self.If[0], 1000 if self.data>self.sma5 else 5000)
        print('self.All',self.All[0], self.data>self.sma5 and self.data>self.sma10 and
self.sma5>self.sma10)
        print('self.Any', self.Any[0], self.data>self.sma5 or self.data>self.sma10 or
self.sma5>self.sma10)
        print('self.Max',self.Max[0], max([self.data[0], self.sma10[0], self.sma5[0]]))
        print('self.Min', self.Min[0], min([self.data[0], self.sma10[0], self.sma5[0]]))
        print('self.Sum', self.Sum[0], sum([self.data[0], self.sma10[0], self.sma5[0]]))
        print('self.Cmp', self.Cmp[0], 1 if self.data>self.sma5 else -1)

cerebro = bt.Cerebro()
st_date = datetime.datetime(2019,1,2)
ed_date = datetime.datetime(2021,1,28)
datafeed1 = bt.feeds.PandasData(dataname=data1, fromdate=st_date, todate=ed_date)
cerebro.adddata(datafeed1, name='600466.SH')
cerebro.addstrategy(TestStrategy)
result = cerebro.run()
```

通常情况下，操作的都是相同周期的数据，比如日度行情数据计算返回各类日度指标、周度行情数据计算返回各类周度指标、……，行情数据和指标的周期是一致的，时间也是对齐的。但有时候也会遇到操作不同周期数据的情况，比如拿日度行情与月度指标作比较，日度行情每天都有数据，而月度指标每个月只有一个，2 条数据在时间上是没有对齐的。

这时，可以使用`( )`语法操作来对齐不同周期的数据，对齐的方向是“大周期向小周期对齐”，可以选择指标对象中的某条 line 进行对齐，也可以对整个指标对象进行对齐。在使用该语法时，要将`cerebro.run()`中的`runonce`设置为 False，才能实现对齐操作。

这样的操作符本质是线的切片操作`get (ago=-1, size=1)`，然后在更细的时间点上始终取当前最新的指标值。比如对于月度指标，向日度对齐时，月中的那些时间点的数据取得是当前最新的数据（即：月初的指标值），直到下个月月初新的指标值计算出来为止。

下面是一个例子：

```
# self.data0 是日度行情、self.data1 是月度行情
self.month = btind.xxx(self.data1) # 计算返回的 self.month 指标也是月度的
# 选择指标对象中的第一条 line 进行对齐
self.sellsignal = self.data0.close < self.month.lines[0]()
# 对齐整个指标对象
self.month_ = self.month()
self.signal = self.data0.close < self.month_.lines[0]

cerebro.run(runonce=False)
```

3.3 内置、外部或自定义指标

Indicators 指标模块提供了 140 多个技术分析指标计算函数，大部分指标与 TA-Lib 库里的指标是一致的，各函数的用途、算法、参数、返回的结果等信息可以查阅官方文档对于内置技术指标的说明。在文档中，可以根据这样的结构了解每一个函数：

- Alias：函数别名，如果一个指标函数包含多个别名，那这些名称都可以作为这个函数的函数名。

  如简单移动均线函数 MovingAverageSimple，别名有 SMA,SimpleMovingAverage，那调用该函数时可以有 3 种写法`btind.MovingAverageSimple()`、`btind.SimpleMovingAverage()`或者`btind.SMA()`。

- Formula：技术指标算法说明，如 MACD 函数的算法为

  ```
  macd = ema(data, me1_period) - ema(data, me2_period)
  signal = ema(macd, signal_period)
  ```

- Lines: 说明函数返回的指标对象中包含哪些 lines，如 MACD 函数返回的指标对象就包含 2 条线：macd 线和 signal 线，可通过`xxxx.lines.macd`或者`xxxx.macd`调用具体的线，`.lines`有时可以省略。

- Params：指标函数可以设置的参数，如移动均线`MovingAverageSimple`包含一个参数：`period (30)`，括号里是该参数的默认值，默认情况下是计算 30 日均值。

- PlotInfo：绘制指标时，支持设置的图形参数，常用绘图参数有以下所示。

  - `plot = True`，是否显示这个指标值，True的显示，False不显示 ；
  - `subplot = True`，是否把指标显示到另一个窗口，True显示到另一个窗口，False显示在主图；
  - `plotname = ""`， 显示 line 的名称，默认是 class.**name**；
  - `plotabove = False`， 指标绘制的位置，False 指标画在主图下方，True 指标画在主图上方；
  - `plotlinelabels = False`，False 显示的是指标函数的名称，True 显示指标线的名称；
  - `plotymargin=0.0`，画图的时候距离顶部和底部的距离；
  - `plotyticks=[ ]`， y 轴刻度范围，取值为空列表时会自动计算；
  - `plothlines=[ ]`，用于绘制水平线；
  - `plotyhlines=[ ]`，用同一个参数，控制 plotyticks 和 plothlines 的取值；
  - `plotforce=False`，如果该绘制的指标没有被绘制，就将 plotforce 设置为 True 进行 强制绘图。
  - `PlotLines`，绘制的曲线样式 。

为了满足大家的使用习惯，Backtrader 还接入了 TA-Lib 技术指标库，具体信息可以查阅官方文档对于TA-Lib支持的描述，文档中同样对各个函数的输入、输出，以及在 Backtrader 中特有的绘图参数、返回的 lines 属性等信息都做了介绍和说明。

TA-Lib 指标函数的调用形式为 `bt.talib.xxx`：

```
class TALibStrategy(bt.Strategy):
    def __init__(self):
        # 计算 5 日均线
        bt.talib.SMA(self.data.close, timeperiod=5)
        bt.indicators.SMA(self.data, period=5)
        # 计算布林带
        bt.talib.BBANDS(self.data, timeperiod=25)
        bt.indicators.BollingerBands(self.data, period=25)
```

在 backtrader 中，如果涉及到自定义操作，一般都是通过继承原始的父类，然后在新的子类里自定义属性，比如之前介绍的自定义数据读取函数 `class My_CSVData (bt.feeds.GenericCSVData)`，就是继承了原始 `GenericCSVData` 类，自定义新指标也类似，需要继承原始的 bt.Indicator 类，然后在新的子类里构建指标。新的子类里通常可以设置如下属性：

- `lines = ('xxx', 'xxx', 'xxx',)`：定义指标函数返回的 lines 名称，方便后面通过名称调用具体的指标，如 `self.lines.xxx` 、`self.l.xxx` 、`self.xxx` ；
- `params = (('xxx', n),)`：定义参数，方便在子类里全局调用，也方便在使用指标函数时修改参数取值；
- `__init__()` 方法：同策略 Strategy 里的 `__init__()` 类似，对整条 line 进行运算，运算结果也以整条 line 的形式返回；
- `next()` 方法：同策略 Strategy 里的 `next()` 类似，每个 bar 都会运行一次，在 `next()` 中是对数据点进行运算；
- `once()` 方法：这个方法只运行一次，但是需要从头到尾循环计算指标；
- 指标绘图相关属性的设置，例如 `plotinfo = dict()` 通过字典形式修改绘图参数；`plotlines = dict()` 设置曲线样式 等等

```
class DummyInd(bt.Indicator):
    # 将计算的指标命名为 'dummyline'，后面调用这根 line 的方式有：
    # self.lines.dummyline ↔ self.l.dummyline ↔ self.dummyline
    lines = ('dummyline',)
    # 定义参数，后面调用这个参数的方式有：
    # self.params.xxx ↔ self.p.xxx
    params = (('value', 5),)

    def __init__(self):
        self.l.dummyline = bt.Max(0.0, self.p.value)

    def next(self):
        self.l.dummyline[0] = max(0.0, self.p.value)

    def once(self, start, end):
        dummy_array = self.l.dummyline.array
        for i in xrange(start, end):
            dummy_array[i] = max(0.0, self.p.value)
    plotinfo = dict(...)
    plotlines = dict(...)
```

通过这个例子，可以比较在 `__init__()`、`next()`、`once()` 中计算指标的区别：

- `__init__()` 中是对 line 进行运算，最终也以 line 的形式返回，所以运算结果直接赋值给了 `self.l.dummyline`；
- `next()` 中是对当前时刻的数据点进行运算（用了常规的 `max()` 函数），返回的运算结果也只是当前时刻的值，所以是将结果赋值给 dummyline 的当前时刻`self.l.dummyline[0]`， 然后依次在每个 bar 都运算一次；
- `once()` 也只运行一次，适合于不直接对指标 line 进行操作，而单纯的 python 运算和赋值的习惯场景；
- 自定义指标时，建议首选 `__init__()`，因为其能自动实现 `next()` 和 `once()` 的功能。

4. 模拟算法交易