第 3 章

CHAPTER 3

交易所 API 介绍

3.1 API 功能简述

API 是 Application Programming Interface 的缩写，中文名称为应用程序编程接口。交易所一般会提供平台基本信息、市场行情、用户的账户信息、现货下单、合约下单等功能的 API，有了这些 API 就可以编写代码，用程序执行策略，实现自动化交易。

API 连接方式有普通 HTTP 请求和 WebSocket 两种。WebSocket 是在 HTTP 的基础上升级的一种新的协议(Protocol)，它实现了全双工通信，一般用来获取行情数据，比普通 HTTP 方式速度更快，延迟更低，连接开销更小。

API 按权限分为两大类：公共 API 和私有 API。公共 API 包括交易所相关信息、费率信息、行情数据等，私有 API 包括用户钱包信息、订单信息、下单操作等功能。私有 API 访问需要带上 API Key 和 Secret Key，同时要求程序运行的云服务器 IP 地址是受信任的(参考第 2 章 API 设置界面)。

API 按功能可分为现货、杠杆、U 本位合约、币本位合约、期权等几类，本书只介绍现货和 U 本位合约中的账户查询、行情数据、交易等最基本的几个 API，读者完全掌握后，可以去交易所官网查看 API 文档，了解更多的功能。

编写交易策略程序，可以使用交易所官方提供的软件开发套件(Software Development Kit，SDK)，币安官方 SDK 支持的编程语言有 Python、Node. js、Java、PHP、Go、C＃。欧易没有官方的 SDK，只推荐了两个第三方的 SDK，这两个第三方的 SDK 支持编程语言 Python、Java 等主流编程语言。本书将教大家使用 Python 语言编写交易程序，即使没学过 Python 也不必担心，第 4 章将带大家掌握 Python 基础语法知识。

3.2 币安 API

使用币安 API 前要先注册 API Key 和 Secret Key。

安装币安 Python SDK，在命令行下输入的指令如下：

```
pip install binance-connector
```

本节将用这个 SDK 编写代码来演示 API 的使用方法。

以/api/*网址开头的 HTTP 接口(行情接口和交易接口)访问频率限制：每分钟6000 次。

以/sapi/*网址开头的 HTTP 接口(钱包和合约相关接口)访问频率限制：每分钟180 000 次。

WebSocket 接口访问频率限制：每 IP 地址每分钟最多可以发送 60 次连接请求。

币安的现货 API 与合约 API 是两个不同的模块,下面将分别介绍它们的使用方法。

3.2.1 币安现货 API

币安的现货 API 需要在程序的开头导入 SDK 的 Spot 包,代码如下：

```
#导入现货模块
from binance.spot import Spot
```

3.2.2 查询现货钱包余额 API

第 1 步,创建现货的客户端对象 client,输入 apikey 和 secret 参数,为 base_url 参数设置一个默认值 https://api1.binance.com/,这是币安 API 的一个基础网址,代码如下：

```
#现货账户
#apikey
key = "你的 API Key"
#密钥
secret = "你的 Secret"
#现货账户
client = Spot(key, secret, base_url="https://api1.binance.com/")
```

第 2 步,调用 SDK 现货模块中的 user_asset 方法获取账户余额,recvWindow 参数是时间空窗值,单位是毫秒,用来判断请求的最小延迟,如果延迟超过此值,则会被交易所认为无效,代码如下：

```
response = client.user_asset(asset="USDT", recvWindow=5000)
print(response)
```

图 3-1 币安账户返回结果

从币安交易所的 API 返回账户余额信息是一个数组包裹的字典数据,数组中只有一个元素,可以用 response[0]方法获取第 1 个元素,第 1 个元素中的资产名称用 response[0]["asset"]方法获取,第 1 个元素中的余额用 response[0]["free"]方法获取,如图 3-1 所示。

查询用户的交易所账户余额的完整代码如下：

```
#文件名:binanceSpotBalance.py
from binance.spot import Spot

#账户余额
def getBalance(symbol):
    key = "你的 key"
    secret = "你的 secret"
    print(key, secret)
    #现货账户
    client = Spot(key, secret, base_url = "https://api1.binance.com/")
    response = client.user_asset(asset = symbol, recvWindow = 5000)
    #返回的结果集是列表数据
    if len(response) > 0:
        return response[0]["free"]
    return 0

#主函数
def main():
    SpotBalance = getBalance("USDT")
    print("现货账户 USDT 的余额是", SpotBalance)

if __name__ == "__main__":
    main()

#运行结果:现货账户 USDT 的余额是 425.31667294
```

3.2.3　现货深度信息 API

交易所深度信息,也称为市场深度或交易深度,深度信息就是交易页面中订单簿里的数据,展示了买方和卖方愿意以不同价格进行交易的数量,币安的深度 API 最多可以返回 5000 条数据,通过观察深度数据,交易者可以了解市场的供需情况,判断价格的走势、潜在的支撑和阻力水平。深度信息参数见表 3-1。

表 3-1　深度信息参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称,例如 BTCUSDT
limit	Int	否	默认值为 100,最大值为 5000,可选值为[5,10,20,50,100,500,1000,5000]

查询深度信息,需要调用 SDK 现货模块中的 depth 方法,代码如下:

```
Symbol = "BTCUSDT"#交易对
Limit = 2         #数量
arr = client.depth(symbol,limit)
print(arr)
```

运行后返回的结果是字典数据(如果不明白字典数据的含义,则可以参考第 4 章内容),

bids是买入价数组的键值名称，数组内容的每个元素包含两个子元素，第1个数值是买入价格，第2个数值是挂单数量，按价格从高到低的顺序排列，价格高的在数组的最前面。asks是卖出价数组的键值，数组内容的每个元素也包含两个子元素，第1个数值是卖出价格，第2个数值是挂单数量，按价格从低到高的顺序排列，价格最低的在数组的最前面。第1个买入价是arr["bids"][0][0]，最后一个卖出价是arr["asks"][-1][0]，如图3-2所示。

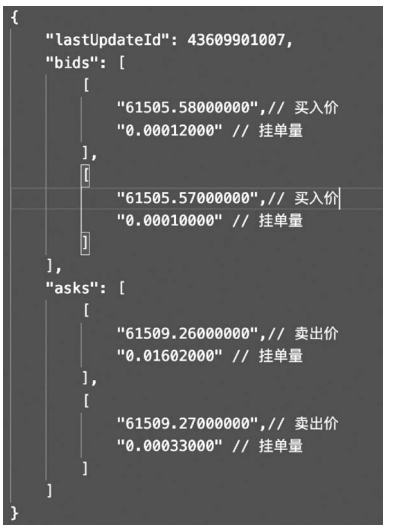

图3-2 返回的深度数据

查询深度信息的完整代码如下：

```
#文件名:binanceSpotDepth.py
#导入现货模块
from binance.spot import Spot

#现货账户
client = Spot()

#深度信息函数
def getDepth(symbol,limit):
    arr = client.depth(symbol, limit)

print("买入价", arr["bids"][0][0])     #取第1个买入价
print("卖出价", arr["asks"][-1][0])    #取最后一个卖出价

#主函数
if __name__ == "__main__":
    #调用深度信息的函数,传入交易对和数量
    getDepth("BTCUSDT",2)

#运行结果如下
#买入价 61505.58
#卖出价 61509.26
```

3.2.4　现货有限深度信息 WebSocket API

有限深度信息的 WebSocket 方式的接口效率更高，以每秒或者每 100ms 推送数据，延迟更低，HTTP 方式每次都要和交易所建立连接，而 WebSocket 只需建立一次长连接，就能不断地接收交易所的推送数据，所以连接开销更小。其参数与 HTTP 接口参数名称略有不同，见表 3-2。

表 3-2　有限深度信息参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
level	Int	否	默认值为 100，最大值为 5000，可选值为[5，10，20，50，100，500，1000，5000]
speed	Int	是	接收数据频率，单位为毫秒

查询有限深度信息，需要加载 SDK 现货模块的 websocket_stream 包和 json 包，使用 SpotWebsocketStreamClient 方法创建一个客户端变量，并指定一个名称为 message_handler 的函数接收推送数据，然后使用 partial_book_depth 方法发出请求，得到的推送数据是 JSON 格式，还需要导入 json 包进行解析，代码如下：

```
#文件名:binanceSpotWsDepth.py
From binance.websocket.spot.websocket_stream
import SpotWebsocketStreamClient
import json
```

创建客户端变量 client，并指定接收推送数据的函数名 message_handler，代码如下：

```
client = SpotWebsocketStreamClient(on_message = message_handler)
```

然后使用 partial_book_depth 方法发出请求，代码如下：

```
#发送请求，参数:交易对 = btcusdt，数量 = 5，频率 = 1000ms
client.partial_book_depth(symbol = "btcusdt", level = 5, speed = 1000)
```

定义一个 message_handler 函数，用于接收推送数据，代码如下：

```
#接收推送数据
def message_handler(_, msg):
    data = json.loads(msg)   #将收到的 JSON 格式数据转换为字典格式数据
    print(data)
```

查询有限深度信息的完整代码如下：

```
#文件名:binanceSpotWsDepth.py
From binance.websocket.spot.websocket_stream
import SpotWebsocketStreamClient
import json
```

```
#接收深度推送数据
def message_handler(_, msg):
    #把交易所推送过来的 JSON 数据转换为字典数据
    data = json.loads(msg)
    print("第 1 个买入价", data["bids"][0][0])
    print("最后 1 个卖出价", data["asks"][-1][0])

#主函数
def main():
    client = SpotWebsocketStreamClient(on_message = message_handler)
    #参数:交易对 = btcusdt,数量 = 5,频率 = 1000ms
    client.partial_book_depth(symbol = "btcusdt", level = 5, speed = 1000)

    #运行结果如下
    #买入价 61505.58
    #卖出价 61509.26

if __name__ == "__main__":
    main()
```

3.2.5 现货 K 线数据 API

K 线图是一种以图形化方式呈现给定时间范围内资产价格变化的金融图表。它由许多烛台图案组成,每个烛台图案表示一段相同的时间。烛台图案可以代表任何虚拟的时间范围,短到数秒,长到数天。K 线图的历史可以追溯到 17 世纪,最早由日本米商发明,后来经过改进和优化,成为现代技术分析的重要工具之一。

K 线图直接反映了资产的价格走势,帮助交易者了解市场的供需情况和价格变化。

K 线图显示了多空之间的博弈,通过观察不同形态的烛台图案,交易者可以判断市场的力量变化。

通过分析 K 线图,交易者可以预测价格的趋势,制定合适的交易策略。

查询 K 线数据需要使用 SDK 现货模块中的 klines 方法,参数见表 3-3。

表 3-3 K 线数据参数

参数名称	类型	是否是必需的	描 述
symbol	String	是	交易对名称,例如 BTCUSDT
interval	Enum	否	K 线间隔,可选值[1s,1m,3m,5m,15m,30m,1h,2h,4h,6h,8h,12h,1d,3d,1w,1M]
limit	Int	是	接收数据数量

查询有限深度信息,需要加载 SDK 现货模块的 klines 方法,代码如下:

```
#文件名:binanceSpotKline.py
from binance.spot import Spot
```

```
client = Spot()        #创建现货账户变量
symbol = "BTCUSDT"     #交易对
interval = "1m"        #更新频率为 1min
limit = 10             #K 线数量
#获得 K 线数据
arr = client.klines(symbol, interval, limit = limit)
print(arr)
```

K 线数据返回的结果是二维数组数据，运行结果如图 3-3 所示。

图 3-3　返回的 K 线数据

查询 K 线数据的完整代码如下：

```
#文件名:binanceSpotKline.py
from binance.spot import Spot

#现货账户
client = Spot()

def getKlines(symbol, interval, limit):
    arr = client.klines(symbol, interval, limit = limit)
    lastId = len(arr) - 1                #最后的 K 线数据,也就是最新数据
    print("开盘价", arr[lastId][1])      #开盘价
    print("最高价", arr[lastId][2])      #最高价
    print("最低价", arr[lastId][3])      #最低价
    print("收盘价", arr[lastId][4])      #收盘价
    #运行结果
    #开盘价 61740.30
    #最高价 61740.30
    #最低价 61659.72
    #收盘价 61666.68

#主函数
def main():
    symbol = "BTCUSDT"                   #交易对
    interval = "1m"                      #更新频率为 1min
    limit = 10                           #K 线数量
```

```
    getKlines(symbol, interval, limit)


if __name__ == "__main__":
    main()
```

3.2.6 现货 K 线数据 WebSocket API

逐秒推送的 K 线数据需要使用 SDK 现货模块中的 websocket_stream 包中的 kline 方法，参数见表 3-4。

表 3-4 K 线数据参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
interval	Enum	否	K 线间隔，可选值为[1s，1m，3m，5m，15m，30m，1h，2h，4h，6h，8h，12h，1d，3d，1w，1M]
limit	Int	是	接收数量

查询有限深度信息，需要加载 SDK 现货模块的 kline 方法，得到的数据格式是 JSON，还需要导入 json 包，代码如下：

```
#文件名:binanceSpotKline.py
from binance.websocket.spot.websocket_stream
import SpotWebsocketStreamClient
import json

def message_handler(_, msg):
    data = json.loads(msg)
    print(data)

client = SpotWebsocketStreamClient(on_message = message_handler)
#订阅 btcusdt 最新 K 线数据，参数：交易对 = btcusdt，频率 = 1s
client.kline(symbol = "btcusdt", interval = "1m")
```

运行结果如图 3-4 所示。

查询 K 线数据的完整代码如下：

```
#文件名:binanceSpotKline.py
from binance.websocket.spot.websocket_stream
import SpotWebsocketStreamClient
import json

#接收和处理 K 线推送数据
def message_handler(_, msg):
    data = json.loads(msg)
    print("交易对", data["k"]["s"])
```

```
    print("最后 1 笔成交价", data["k"]["c"])

# 主函数
def main():
    client = SpotWebsocketStreamClient(on_message = message_handler)
    # 订阅 btcusdt 最新 K 线数据,参数:交易对 = btcusdt,频率 = 1s
    client.kline(symbol = "btcusdt", interval = "1m")
    # 运行结果
    # 交易对 BTCUSDT
    # 最后 1 笔成交价 61350.44000000

if __name__ == "__main__":
    main()
```

```
{
    "e": "kline", //事件类型
    "E": 1709177218485, //事件时间
    "s": "BTCUSDT", //交易对
    "k": {
        "t": 1709177160000, //这根K线的起始时间
        "T": 1709177219999, //这根K线的结束时间
        "s": "BTCUSDT", //交易对
        "i": "1m", //K线间隔
        "f": 3442584285, //这根K线期间第一笔成交ID
        "L": 3442585791, // 这根K线期间末一笔成交ID
        "o": "61305.11000000", // 这根K线期间第一笔成交价
        "c": "61300.11000000", // 这根K线期间末一笔成交价
        "h": "61314.88000000", // 这根K线期间最高成交价
        "l": "61280.00000000", // 这根K线期间最低成交价
        "v": "34.92390000", // 这根K线期间成交量
        "n": 1507, // 这根K线期间成交笔数
        "x": False, // 这根K线是否完结(是否已经开始下一根K线)
        "q": "2140778.01323160", // 这根K线期间成交额
        "V": "16.87782000", // 主动买入的成交量
        "Q": "1034547.98146000", // 主动买入的成交量
        "B": "0" // 忽略此参数
    }
}
```

图 3-4　返回的 K 线数据

3.2.7　现货下单 API

下单操作需要用真实的数字币,如果代码有误,则会损失数字币和手续费。幸运的是,币安提供了测试网,可以使用免费的测试网来测试代码。代码无误后,把测试网址换成正式的网址。

为了更好地管理正式环境和测试环境的 API Key,避免每写一个程序都以明文方式写出 API Key,新建一个 config.ini 文件来保存实盘、现货测试网、合约测试网的 API Key 和 Secret。注意:文件内容不要包含单引号和双引号,文件的内容如下:

```
# 文件名:config.ini
[keys]
apiKey = aaaaaaaa                    # 正式环境的 apiKey
```

```
apiSecret = bbbbbbbbbb              # 正式环境的 apiSecret

testKey = ccccccc                   # 测试环境现货 apiKey
testSecret = ddddddddd              # 测试环境现货 apiSecret

testFuturesKey = eeeeee             # 测试环境合约 apiKey
testFuturesSecret = fffff           # 测试环境合约 apiSecret
```

然后我们写一个从配置文件 config.ini 中读取 API Key 和 Secret 的函数，文件的内容如下：

```
# 文件名:env.py
import os
import pathlib
from configparser import ConfigParser

# 获取 apikey 和 secret
def getApiKey(k,s):
    config = ConfigParser()
    # 获取 config.ini 文件的路径
    config_file_path = os.path.join(
        pathlib.Path(__file__).parent.resolve(), ".", "config.ini"
    )
    # 读取 config.ini 文件内容,放入字典中
    config.read(config_file_path)
    return config["keys"][k], config["keys"][s]
```

现货下单主要有限价单和市价单两种类型，限价单是达到指定价格才执行的订单，市价单是以当前市场价格立即执行的订单，参数见表 3-5。

表 3-5　下单参数

参 数 名 称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
side	String	是	方向：买入(BUY)或卖出(SELL)
type	String	是	类型：限价(Limit)或市价(Market)
price	String	否	价格，类型是限价时必须带此参数
quantity	String	是	基准币数量
quoteOrderQty	String	否	计价币数量，当 type=Market 时才可使用
timeInForce	String	否	有效方式，当类型为限价时必须带此参数

(1) 限价下单，首先获取正式环境的 API Key。注意：正式环境 API 的基础 URL 是 https://api.binance.com。代码如下：

```
key, secret = getApiKey("apiKey", "apiSecret")
client = Spot(key, secret, base_url = "https://api.binance.com/")
```

如果用测试环境的 API Key，则基础 URL 是 https://testnet.binance.vision，代码如下：

```
key, secret = getApiKey("testKey", "testSecret")
client = Spot(key, secret, base_url = "https://testnet.binance.vision")
```

希望以 380 的价格购买 0.05 个 BNB，下限价单的方法用 new_order 函数。注意：下单价格不能距离市场价太近，否则会被平台拒绝执行，必填参数为 symbol、side、type、timeInForce、quantity、price，代码如下：

```
#文件名:binanceSpotKline.py
#参数表都是必选项
params = {
      "symbol": "BNBUSDT",      //交易对
      "side": "BUY",            //方向是买
      "type": "LIMIT",          //类型是限价
      "timeInForce": "GTC",
      "quantity": "0.05",       //BNB 数量
      "price": "350",           //价格
}
#下单并返回结果
response = client.new_order( ** params)
print(response)
```

程序运行后，交易所 API 返回的是一个字典数据，其中最重要的 3 个参数是订单 ID、订单状态和成交数量。

订单 ID 用 response["orderId"]方法获得，获得订单 ID 后可以用订单查询 API 进一步查询订单的详情。

订单状态用 response["status"]方法获得，如果订单状态为 NEW，则表示未成交，如果订单状态为 FILLED，则表示完全成交，如果订单状态为 PARTIALLY_FILLED，则表示部分成交。

成交数量用 response["executedQty"]方法获得，返回结果如图 3-5 所示。

```
{
    "symbol": "BNBUSDT", //交易对
    "orderId": 2882514, //系统的订单ID
    "orderListId": -1, //OCO订单ID, 否则为-1
    "clientOrderId": "s8rqTlhuBY75VjYA8JW2vm", // 客户自定义的ID
    "transactTime": 1709262151069, //交易的时间戳
    "price": "380.00000000", //订单价格
    "origQty": "0.05000000", //用户设置的原始订单数量
    "executedQty": "0.00000000", //交易的订单数量
    "cummulativeQuoteQty": "0.00000000", // 累计交易的金额
    "status": "NEW", //订单状态
    "timeInForce": "GTC",  //订单的时效方式
    "type": "LIMIT", //订单类型, 比如市价单、限价单等
    "side": "BUY", //订单方向, 买还是卖
    "workingTime": 1709262151069, //订单添加到orderbook的时间
    "fills": [], //订单中交易的信息
    "selfTradePreventionMode": "EXPIRE_MAKER" //自我交易预防模式
}
```

图 3-5　限价单返回数据

下限价单的完整代码如下：

```
#文件名:binanceSpotLimitOrd.py
from binance.spot import Spot
from env import getApiKey

#下一个限价单
def limitOrder(symbol, side, qty, price):
    #正式环境
    #key, secret = getApiKey("apiKey", "apiSecret")
    #client = Spot(key, secret, base_url = "https://api.binance.com")
    #测试网
    testKey, testSecret = getApiKey("testKey", "testSecret")
    client = Spot(testKey, testSecret, base_url = "https://testnet.binance.vision")
    #参数字典
    params = {
        "symbol": symbol,              //交易对
        "side": side,                  //方向
        "type": "LIMIT",               //类型
        "timeInForce": "GTC",
        "quantity": qty,               //数量
        "price": price,                //价格
    }
    #下单
    response = client.new_order( ** params)
    print(response)

#主函数
def main():
    #限价方式以 380 为价格,购买 0.05 个 BNB
    limitOrder("BNBUSDT", "BUY", "0.05", "380")

if __name__ == "__main__":
    main()
```

(2) 以基准币市价下单,方法也是用 new_order 函数,必填参数为 symbol、side、type、quantity,代码如下:

```
#文件名:binanceSpotOrder.py
from binance.spot import Spot
from env import getApiKey

#下一个市价单
def marketOrder(symbol, side, qty):
    #测试网
    key, secret = getApiKey("testKey", "testSecret")
    client = Spot(key, secret, base_url = "https://testnet.binance.vision")
    params = {
      "symbol": symbol,      //交易对
      "side": side,          //方向
      "type": "MARKET",      //类型
```

```
        "quantity": qty                    //数量
    }
    response = client.new_order(**params)
    print(response)
    print("成交价",response["fills"][0]["price"])
    print("成交数量",response["fills"][0]["qty"])

def main():
    #以市价购买 0.1 个 BNB,订单立刻生效
    marketOrder("BNBUSDT", "BUY", "0.1")

if __name__ == "__main__":
    main()
```

程序运行后,交易所返回一个字典数据,和限价单不同的是 fills 字段里多了一个数组数据,里面保存的是交易结果数据,成交价格用 response["fills"][0]["price"]方法获得,成交数量用 response["fills"][0]["qty"]方法获得,结果如图 3-6 所示。


```
{
    "symbol": "BNBUSDT", //交易对
    "orderId": 2883468, //系统的订单ID
    "orderListId": -1, //OCO订单ID, 否则为-1
    "clientOrderId": "b4WiJR3m70ymS2T1Ca4tNo", // 客户自定义的ID
    "transactTime": 1709262904408, //交易的时间戳
    "price": "0.00000000", //订单价格,市价类型,此处都是0
    "origQty": "0.10000000", //用户设置的原始订单数量
    "executedQty": "0.10000000", //交易的订单数量
    "cummulativeQuoteQty": "40.56000000", // 累计交易的金额
    "status": "FILLED", //订单状态
    "timeInForce": "GTC", //订单的时效方式
    "type": "MARKET", //订单类型, 比如市价单、限价单等
    "side": "BUY", //订单方向, 买还是卖
    "workingTime": 1709262904408, //订单添加到order book的时间
    "fills": [  //订单中交易的信息
        {
            "price": "405.60000000", //交易的价格
            "qty": "0.10000000",  //交易的数量
            "commission": "0.04000000", //手续费金额
            "commissionAsset": "USDT", // 手续费的币种
            "tradeId": 104782 //交易ID
        }
    ],
    "selfTradePreventionMode": "EXPIRE_MAKER"
}
```


图 3-6　市价单返回数据

(3) 以计价币市价下单,方法也是用 new_order 函数,必填参数为 symbol、side、type、quoteOrderQty,代码如下:

```
#文件名:binanceSpotOrder.py
from binance.spot import Spot
from env import getApiKey
```

```
#用 180 个 USDT,下一个市价单
def marketOrder(symbol, side, qty):
    #key, secret = getApiKey("apiKey","apiSecret")
    #client = Spot(key, secret, base_url = "https://api.binance.com")
    #测试网
    testKey, testSecret = getApiKey("testKey", "testSecret")
    client = Spot(testKey, testSecret, base_url = "https://testnet.binance.vision")
    params = {"symbol": symbol, "side": side, "type": "MARKET", "quoteOrderQty": qty}
    response = client.new_order( ** params)
    print(response)

def main():
    #用 180 个 USDT 以市价换成尽可能多的 BNB 下单
    marketOrder("BNBUSDT", "BUY", "180")

if __name__ == "__main__":
    main()
```

运行结果如图 3-7 所示。


```
{
    "symbol": "BNBUSDT", //交易对
    "orderId": 5401866, //订单ID
    "orderListId": -1,
    "clientOrderId": "1X83XGnlvT2DxgEn9l9zP9",
    "transactTime": 1712548039660,
    "price": "0.00000000",
    "origQty": "0.30900000",
    "executedQty": "0.30900000", //交易的订单数量
    "cummulativeQuoteQty": "179.49810000", //消耗的usdt数量
    "status": "FILLED", //状态: 完成成交
    "timeInForce": "GTC",
    "type": "MARKET", //订单类型
    "side": "BUY", //方向: 买入
    "workingTime": 1712548039660,
    "fills": [
        {
            "price": "580.90000000", //价格
            "qty": "0.30900000",  //数量
            "commission": "0.00000000",
            "commissionAsset": "BNB",
            "tradeId": 250780
        }
    ],
    "selfTradePreventionMode": "EXPIRE_MAKER"
}
```


图 3-7　市价单返回数据

3.2.8　现货查询订单信息 API

下单操作后平台并不能马上返回结果,想要查询订单状态,需要调用查询订单信息的 API。参数见表 3-6。

表 3-6　查询订单参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
orderId	String	否	订单 ID

代码如下：

```
ord = client.get_order("BTCUSDT", orderId = "9783716")
print(ord)
```

程序运行后，交易所会返回一个字典数据，里面的重要参数有：price，表示下单价格；origQty，表示下单数量；status，表示订单状态；executedQty，表示成交数量；type，表示订单类型；side，表示买卖方向。

运行结果如图 3-8 所示。

图 3-8　查询订单返回数据

3.2.9　现货取消订单 API

取消订单 API 只能取消限价订单，并且订单的状态是 NEW（表示订单未成交），否则无法取消。参数见表 3-7。

表 3-7　取消订单参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
orderId	String	否	订单 ID

代码如下：

```
ord = client.cancel_order("BTCUSDT", orderId = "9783716")
print(ord)
```

3.2.10 应用示例：现货 API 综合应用

本节讲解了现货 API 中的查询行情、下单，以及查询订单、取消订单的基本操作，现在综合应用这些 API 写两个示例程序。

第 1 个示例程序的主要功能是下一个限价买单：以 60000 的价格购买 0.001 个 BTC，下单成功后会返回一个 orderId，然后用这个 orderId 查询订单的状态。如果状态等于 NEW，则调用取消订单 API，取消成功后返回结果字典中的状态值为 CANCELED，表示取消操作成功。

Python 语法部分，用到了函数定义和函数调用，我们把下单操作定义为 newLimitOrd 函数，将查询订单操作定义为 getOrder 函数，将取消订单操作定义为 cancelOrder 函数。主函数为 main，这是程序运行的起点。函数命名使用了驼峰规则，也就是函数名首字母小写，后面的每个单词首字母大写，这样的命名规则让人一看函数名就知其意。

第 1 个示例程序的完整代码如下：

```
文件名:binanceSpotGetOrd.py

from binance.spot import Spot
from env import getApiKey

#测试网
key, secret = getApiKey("testKey", "testSecret")
client = Spot(key, secret, base_url = "https://testnet.binance.vision")

#限价单
def newLimitOrd(symbol, side, price, qty):
    params = {
        "symbol": symbol,
        "side": side,
        "type": "LIMIT",
        "timeInForce": "GTC",
        "quantity": qty,
        "price": price,
    }
    response = client.new_order( ** params)   #返回的订单信息
    ordId = response["orderId"]               #订单 ID
    return ordId

#查询订单
def getOrder(symbol, ordId):
    ord = client.get_order(symbol, orderId = ordId)
```

```
    print(ord)
    print("订单状态", ord["status"])
    return ord

#取消订单
def cancelOrder(symbol, ordId):
    response = client.cancel_order(symbol, orderId = ordId)
    #取消成功,状态是 CANCELED
    print("取消订单结果", response["status"])

def main():
    #限价单,以 60000 的价格购买 0.001 个 BTC
    symbol = "BTCUSDT"
    orderId = newLimitOrd(symbol, "BUY", "60000", "0.001")
    ord = getOrder(symbol, orderId)
    print("限价单状态", ord["status"])
    if ord["status"] == "NEW":
        cancelOrder(symbol, orderId)

if __name__ == "__main__":
    main()
```

第 2 个示例程序更加接近实际应用,先获取行情数据,获得当前市价,确定一个有机会盈利的下单价格,下单价格=市价-市价×0.005%,以此价格下一个限价买单。

然后在循环结构中不断地查询订单状态,如果订单状态为 NEW,则表示未成交,程序就休眠 10s。如果订单状态为 FILLED,则表示完全成交。如果状态为 PARTIALLY_FILLED,则表示部分成交,立即下一个限价卖单,卖单价格为买入价+利润,卖出单的下单数量从查询订单接口返回数据中取 executedQty 字段值,表示实际成交数量,调用 API 部分的代码都使用 try/except 包裹起来,防止因 API 出错而导致程序中断运行。

精度计算也是一个非常重要的知识点,下单数量精度和价格精度(也就是小数位数),必须满足交易对的精度要求,否则下单会失败。例如下单数量 qty 为“2.001”,小数点后有 3 位小数,精度就是 3,如何计算精度呢? 可以使用 Python 的 split 函数对下单数量字符进行分隔,分隔符是“.”,arr=qty.split(".")得到的就是一个数组,数组名为 arr,arr 数组的第 1 个元素是整数部分“2”,arr 数组的第 2 个元素是小数点后的部分“001”,然后用 Python 的 len 函数计算“001”的长度,len(arr[1])就等于 3。注意:arr[1]指第 2 个元素,arr[0]指第 1 个元素。程序代码如下:

```
文件名:binanceSpotGetOrd.py

#测试币安买和卖流程
import json
import time
from binance.spot import Spot
from binance.websocket.spot.websocket_stream import SpotWebsocketStreamClient
```

```
from env import getApiKey


#获取 API Key 和 Secret
testKey, testSecret = getApiKey("testKey", "testSecret")
client = Spot(testKey, testSecret, base_url = "https://testnet.binance.vision")


#限价单
def newLimitOrd(symbol, side, price, qty):
    params = {
        "symbol": symbol,
        "side": side,
        "type": "LIMIT",
        "timeInForce": "GTC",
        "quantity": qty,
        "price": price,
    }
    print(params)
    response = client.new_order( ** params)   #返回的订单信息
    ordId = response["orderId"]               #订单 ID
    return ordId


#查询订单
def getOrder(symbol, ordId):
    try:
        ord = client.get_order(symbol, orderId = ordId)
        print(ord)
        print("订单状态", ord["status"])
        return ord
    except Exception as e:
        print(f"查询订单错误: {e}")
        return {}


#取消订单
def cancelOrder(symbol, ordId):
    try:
        response = client.cancel_order(symbol, orderId = ordId)
        #取消成功,状态是 CANCELED
        print("取消订单结果", response["status"])
    except Exception as e:
        print(f"取消订单错误: {e}")


#取消所有订单
def cancelAllOrder(symbol):
    try:
```

```
        response = client.cancel_open_orders(symbol)
        #取消成功,状态是 CANCELED
        print("取消所有订单", response["status"])
    except Exception as e:
        print(f"取消所有订单错误: {e}")


#计算下单数量的精度,也就是小数位
def setqtyDecimalNum(qty):
    global qtyDecimalNum
    arr = qty.split(".")
    if len(arr) > 0:
        qtyDecimalNum = len(arr[1])
    else:
        qtyDecimalNum = 0
    print(f"将小数位长度设置为{qtyDecimalNum}")


#计算价格的精度,也就是小数位
def getPriceDecimalNum(price):
    arr = price.split(".")
    if len(arr) > 0:
        num = arr[1].rstrip("0")
        numLen = len(num)
        return numLen
    else:
        return 0


def getKlines(symbol, interval, limit):
    arr = client.klines(symbol, interval, limit=limit)
    #print(arr)
    #返回数据格式[[开盘时间,开盘价,最高价,最低价,收盘价,成交量],[开盘时间,开盘价,最高
    #价,最低价,收盘价,成交量]...]
    #[[1499040000000', "61740.30","61740.30","61659.72","61666.68","148976.11427815"]]
    lastId = len(arr) - 1           #最后的 K 线数据,也就是最新数据
    print("开盘价", arr[lastId][1]) #开盘价
    print("最高价", arr[lastId][2]) #最高价
    print("最低价", arr[lastId][3]) #最低价
    print("收盘价", arr[lastId][4]) #收盘价
    return arr[lastId][4]


def main():
    symbol = "BTCUSDT"              #交易对
    interval = "1m"                 #将频率更新为 1min
    limit = 10                      #K 线数量
    cancelAllOrder(symbol)          #先取消所有订单
    close = getKlines(symbol, interval, limit)
```

```
    print("市价", close)
    closeF = float(close)
    #按低于市价0.03%价格购买
    closeF -= closeF * 0.0003
    #价格精度
    priceDecimalNum = getPriceDecimalNum(close)
    closeF = round(closeF, priceDecimalNum)
    qty = 0.001
    qtyDecimalNum = 3
    ordId = newLimitOrd(symbol, "BUY", f"{closeF}", qty)
    print(f"挂买单:订单id={ordId},交易对={symbol},方向=BUY,价格={closeF},数量=
{qty}")
    while True:
        try:
            #检查订单是否成交
            ord = getOrder(symbol, ordId)
            #如果状态是NEW,则表示未成交
            if ord["status"] == "NEW":
                time.sleep(10)
            elif ord["status"] == "FILLED" or ord["status"] == "PARTIALLY_FILLED":
                #命中
                print(f"订单id={ordId}命中")
                #获得订单执行价格
                orderPrice = float(ord["price"])
                #价格精度
                priceDecimalNum = getPriceDecimalNum(ord["price"])
                #计算盈利价格
                profitPrice = orderPrice + orderPrice * 0.0005
                #保持正确的小数位
                profitPrice = round(profitPrice, priceDecimalNum)
                #开始卖出
                #获取订单的执行数量
                executedQty = float(ord["executedQty"])
                print("执行数量", executedQty)
                #cummulativeQuoteQty = float(ord["cummulativeQuoteQty"])
                #保持精度
                executedQty = round(executedQty, qtyDecimalNum)
                print("执行数量,保持精度", executedQty)
                try:
                    ordId2 = newLimitOrd(
                        symbol, "SELL", f"{profitPrice}", f"{executedQty}"
                    )
                    print(
                        f"挂单信息:订单id={ordId2},交易对={symbol},方向=SELL,价格=
{profitPrice},数量={executedQty}"
                    )
                    print("=======完成挂卖出单,等待成交获利=======")
                    print("======退出======")
                    break
```

```
                except Exception as e:
                    print(" ======= 挂卖出单错误 ======= ")
                    print(e)
            except Exception as e:
                print(f"查询订单错误: {e}")
                time.sleep(10)


if __name__ == "__main__":
    print("测试币安买和卖的流程")
    main()
```

3.2.11　币安合约 API

现在的加密货币交易所,使用合约交易的用户比现货交易的用户多很多,从 API 限制频率来看:现货 API 交易每分钟被限制为 6000 次,而合约 API 交易每分钟被限制为 180000 次。手续费也是合约交易远低于现货交易,也就是平台鼓励用户使用合约交易。

合约 API 需要在程序的开头导入 SDK 的 UMFutures 包,代码如下:

```
#导入合约模块
from binance.um_futures import UMFutures
```

新建一个合约对象变量,代码如下:

```
client = UMFutures()
```

3.2.12　合约深度信息 API

获取合约深度信息的方法是 depth,需要的参数见表 3-8。

表 3-8　合约深度信息参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称,例如 BTCUSDT
limit	Int	否	默认值为 500,最大值为 1000,可选值为[5,10,20,50,100,500,1000]

使用 depth 方法传入交易对和数量两个参数,代码如下:

```
symbol = "BTCUSDT"
params = {
    "limit": 10,
}
arr = client.depth(symbol , **params)
print(arr)
```

返回的深度数据如图 3-9 所示。

图 3-9　合约深度数据

完整代码如下：

```
文件名:binanceFuturesDepth.py
from binance.um_futures import UMFutures

client = UMFutures()
symbol = "BTCUSDT"
params = {
   "limit": 10,
}

arr = client.depth(symbol, **params)
print("买入价", arr["bids"][0][0])     #取第1个买入价
print("卖出价", arr["asks"][-1][0])   #取最后一个卖出价

#返回结果
#买入价 61867.80
#卖出价 61868.40
```

3.2.13　合约有限深度信息 WebSocket API

有限深度信息的 WebSocket 方式的接口效率更高，以每秒或者每 100ms 推送数据，延迟更低，与 HTTP 接口参数名称略有不同，参数见表 3-9。

表 3-9　合约有限深度信息参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
level	Int	否	默认值为 100，最大值为 5000，可选值为[5，10，20，50，100，500，1000，5000]
speed	Int	是	更新频率，单位为毫秒，可选值为[100，250，500]

查询有限深度信息，需要加载 SDK 合约模块的 websocket_client 包和 json 包，代码如下：

```
#文件名:binanceSpotWsDepth.py
from binance.websocket.um_futures.websocket_client import
UMFuturesWebsocketClient
import json
```

使用 UMFuturesWebsocketStreamClient 方法创建一个合约对象变量，并指定一个名称为 message_handler 的函数接收推送数据，代码如下：

```
client = SpotWebsocketStreamClient(on_message=message_handler)
```

然后使用 partial_book_depth 方法发出请求，得到的推送数据是 JSON 格式，还需要导入 json 包进行解析，代码如下：

```
client.partial_book_depth(
    symbol="BTCUSDT",      #交易对
    level=20,              #数量
    speed=100,             #更新速度,单位为毫秒
)
```

接收推送数据，并用 json 包进行解析，代码如下：

```
#接收推送数据
def message_handler(_, msg):
    data = json.loads(msg)
    print(data)
```

WebSocket 接口返回的字段一般用简写，用 e 代替 event、用 s 代替 symbol、用 b 代替 bid、用 a 代替 ask，这样做的目的是最大限度地减少数据传输量，返回的深度数据如图 3-10 所示。

完整代码如下：

```
#文件名:binanceFuturesWsDepth.py
import time
from binance.websocket.um_futures.websocket_client import
UMFuturesWebsocketClient
import json


#接收推送数据
```

```
def message_handler(_, msg):
    data = json.loads(msg)
    #print(data)
    print("第 1 个买入价", data["b"][0][0])
    print("最后一个卖出价", data["a"][-1][0])
    #运行结果
    #第 1 个买入价 62066.30
    #最后一个卖出价 62070.80

client = UMFuturesWebsocketClient(on_message=message_handler)
client.partial_book_depth(
    symbol="BTCUSDT",    #交易对
    level=20,            #数量
    speed=100,           #更新速度,单位为毫秒
)

#time.sleep(10)         #休眠 10s
#client.stop()          #停止接收推送 print(data)
```


```
{
    "e": "depthUpdate", //事件类型
    "E": 1709437361616, //事件时间
    "T": 1709437361605, //交易时间
    "s": "BTCUSDT", //交易对
    "U": 4085989129934, //从上次推送至今新增的第一个 update Id
    "u": 4085989133230, //从上次推送至今新增的最后一个 update Id
    "pu": 4085989128744, //上次推送的最后一个update Id(上条消息的'u')
    "b": [ //买方
        [
            "62049.70", //价格
            "3.524" //数量
        ],
        [
            "62049.60", //价格
            "0.004" //数量
        ]
    ],
    "a": [ //卖方
        [
            "62052.20", //价格
            "0.011" //数量
        ],
        [
            "62052.40", //价格
            "0.117" //数量
        ]
    ]
}
```


图 3-10　合约 WebSocket 深度数据

3.2.14　合约 K 线 API

获取合约 K 线数据的方法是 klines,需要的参数见表 3-10。

表 3-10　合约 K 线 API 参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
interval	String	否	时间间隔，可选值为[1m，3m，5m，15m，30m，1h，2h，4h，6h，8h，12h，1d，3d，1w]
limit	Int	否	默认值为 500，最大值为 1000，可选值为[5，10，20，50，100，500，1000]

kline 使用方法，完整代码如下：

```
文件名:binanceFuturesKline.py
from binance.um_futures import UMFutures

client = UMFutures()
symbol = "BTCUSDT"
params = {
   "interval": "1m",              //间隔频率 1min
   "limit": 10
}
arr = client.klines(symbol, **params)
print(arr)
lastId = len(arr) - 1            #最后的 K 线数据，也就是最新数据
print("开盘价", arr[lastId][1])  #开盘价
print("最高价", arr[lastId][2])  #最高价
print("最低价", arr[lastId][3])  #最低价
print("收盘价", arr[lastId][4])  #收盘价

#运行结果
#开盘价 62059.60
#最高价 62065.30
#最低价 62059.60
#收盘价 62065.20
```

返回的 K 线数据如图 3-11 所示。

```
[
  [
    1709435340000, //开盘时间
    "61905.40", //开盘价
    "61905.50", //最高价
    "61879.00", //最低价
    "61881.50", //收盘价(当前K线未结束的即为最新价)
    "77.511", //成交量
    1709435399999, //收盘时间
    "4797612.99630", //成交额
    1279, //成交笔数
    "17.532", //主动买入成交量
    "1085192.40140", //主动买入成交额
    "0" //忽略该参数
  ]
]
```

图 3-11　合约 K 线数据

3.2.15 合约 K 线数据 WebSocket API

合约 K 线数据需要的参数见表 3-11。

表 3-11 合约 K 线数据参数

参数名称	类型	是否是必需的	描 述
symbol	String	是	交易对名称,例如 BTCUSDT
level	Int	否	默认值为 100,最大值为 5000,可选值为[5,10,20,50,100,500,1000,5000]
speed	Int	是	更新频率,单位为毫秒,可选值为[100,250,500]

创建一个合约对象,然后用 kline 方法获得 K 线推送数据,并指定一个名称为 message_handler 的函数接收推送数据,代码如下:

```
client = UMFuturesWebsocketClient(on_message = message_handler)
client.kline(
    symbol = "BTCUSDT",
    interval = "1m",
)
```

message_handler 函数接收到推送数据后解析 JSON 数据,代码如下:

```
def message_handler(_, msg):
    data = json.loads(msg)
    #print(data)
```

推送的 K 线数据如图 3-12 所示。

图 3-12 推送的合约 K 线数据

完整代码如下：

```
#文件名:binanceFuturesWsKline.py
import time
import json
from binance.websocket.um_futures.websocket_client import
UMFuturesWebsocketClient


#接收推送数据
def message_handler(_, msg):
    data = json.loads(msg)

print("交易对", data["k"]["s"])
    print("第 1 笔成交价", data["k"]["o"])
    print("最后一笔成交价", data["k"]["c"])
    print("最高成交价", data["k"]["h"])
    print("最低成交价", data["k"]["l"])
    print("成交量", data["k"]["v"])
    #运行结果
    #交易对 BTCUSDT
    #第 1 笔成交价 62040.00
    #最后一笔成交价 62060.70
    #最高成交价 62060.70
    #最低成交价 62039.90
    #成交量 41.368


client = UMFuturesWebsocketClient(on_message = message_handler)

client.kline(
    symbol = "BTCUSDT",
    interval = "1m",
)

#time.sleep(10)   #休眠 10s
#client.stop()    #停止接收推送
```

3.2.16　合约查询余额 API

合约账户包括多种资产，数据格式如图 3-13 所示。

对于 U 本位合约，只要查资产是 USDT 的余额即可，示例代码如下：

```
文件名:binanceFuturesBalance.py
from binance.um_futures import UMFutures
from env import getApiKey

key, secret = getApiKey("apiKey", "apiSecret")

#合约账户
```

```
client = UMFutures(key = key, secret = secret)


#合约账户余额
def getBalance(asset):
    arr = client.balance(recvWindow = 5000)
    print(arr)
    for item in arr:
        if item["asset"] == asset:
            return float(item["availableBalance"])
    return 0.00000000


def main():
    balance = getBalance("USDT")
    print("合约账户 USDT 余额", balance)


if __name__ == "__main__":
    main()
```


```
[
    {
        "accountAlias": "SgoCFzfWAuXqSgFz", //账户唯一识别码
        "asset": "BTC", //资产
        "balance": "0.00000000", //总余额
        "crossWalletBalance": "0.00000000", //全仓余额
        "crossUnPnl": "0.00000000", //全仓持仓未实现盈亏
        "availableBalance": "0.00000000", //下单可用余额
        "maxWithdrawAmount": "0.00000000", //最大可转出余额
        "marginAvailable": True, //是否可用作联合保证金
        "updateTime": 0 //更新时间
    },
    {
        "accountAlias": "SgoCFzfWAuXqSgFz", //账户唯一识别码
        "asset": "USDT", //资产
        "balance": "318.06463735", //总余额
        "crossWalletBalance": "318.06463735", //全仓余额
        "crossUnPnl": "0.00000000", //全仓持仓未实现盈亏
        "availableBalance": "318.06463735", //下单可用余额
        "maxWithdrawAmount": "318.06463735", //最大可转出余额
        "marginAvailable": True, //是否可用作联合保证金
        "updateTime": 1709357612889 //更新时间
    }
]
```


图 3-13　合约账户数据格式

3.2.17　合约设置逐仓全仓 API

设置逐仓全仓 API 需要的参数见表 3-12。

表 3-12　设置逐仓全仓 API 需要的参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
marginType	String	是	可选值为[CROSSED 全仓，ISOLATED 逐仓]

示例代码如下：

```
from binance.um_futures import UMFutures
from env import getApiKey

key, secret = getApiKey("apiKey", "apiSecret")
client = UMFutures(key = key, secret = secret)
#marginType 值选项:[CROSSED 全仓,ISOLATED 逐仓]
response = client.change_margin_type(
    symbol = "BTCUSDT", marginType = "CROSSED", recvWindow = 6000
)
print(response)
```

3.2.18　合约设置杠杆倍数 API

设置杠杆倍数 API 需要的参数见表 3-13。

表 3-13　设置杠杆倍数 API 需要的参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
leverage	Int	是	杠杆倍数

示例代码如下：

```
from binance.um_futures import UMFutures
from env import getApiKey

key, secret = getApiKey("apiKey", "apiSecret")
client = UMFutures(key = key, secret = secret)
response = client.change_leverage(
       symbol = "BTCUSDT", leverage = 10, recvWindow = 6000
)
print(response)
#返回结果
#leverage 杠杆倍数
#maxNotionalValue 最大名义价值
#{'symbol': 'BTCUSDT', 'leverage': 10, 'maxNotionalValue': '150000000'}
```

3.2.19　合约下单 API

合约下单 API 的方法是 new_order，这是一个难点，很多新手在这里容易写错，涉及如

何开仓、平仓、新建委托订单等操作，交易所文档只是罗列了参数，没有讲解如何使用，参数比较多，见表 3-14，我们将以示例代码的方式详细解读。

表 3-14　设置杠杆倍数 API 需要的参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称，例如 BTCUSDT
side	ENUM	是	买卖方向为 SELL 或 BUY
positionSide	ENUM	是	持仓方向，默认双持仓模式下的选择项为[做多 LONG，做空 SHOAT]
type	ENUM	是	订单类型为[LIMIT，MARKET，TAKE_PROFIT_MARKET，TRAILING_STOP_MARKET，TAKE_PROFIT]
reduceOnly	String	否	非双开模式选 True 或 False，双开模式不选此参数
quantity	DECIMAL	否	下单数量，使用 closePosition 时不填此参数
price	DECIMAL	否	委托价格
stopPrice	DECIMAL	否	触发价格
activatePrice	DECIMAL	否	追踪止损激活价格，仅当 type= TRAILING_STOP_MARKET 时使用
callbackRate	DECIMAL	否	追踪止损回调比例，取值范围为[0.1，10]，仅当 type = TRAILING_STOP_MARKET 时使用

开仓做多，side 一定要是 BUY，同时 positionSide 一定要是 LONG，示例代码如下：

```
#开仓做多
response = client.new_order(
    symbol = "BTCUSDT",      #交易对
    side = "BUY",            #交易方向
    type = "MARKET",         #类型
    quantity = 0.01,         #数量
    positionSide = "LONG",   #持仓方向
)
print("开仓做多")
print(response)
```

开仓做空，side 一定要是 SELL，同时 positionSide 一定要是 SHORT，示例代码如下：

```
#开仓做多
response = client.new_order(
    symbol = "BTCUSDT",      #交易对
    side = "SELL",           #交易方向
    type = "MARKET",         #类型
    quantity = 0.01,         #数量
    positionSide = "SHORT", #持仓方向
)
print("开仓做空")
print(response)
```

返回的订单数据如图 3-14 所示。

```
{
    "orderId": 3718946294, //系统订单号
    "symbol": "BTCUSDT", //交易对
    "status": "NEW", //订单状态
    "clientOrderId": "s6qlFxpKfq4qoxlD6vad4M", //用户自定义的订单号
    "price": "0.00", //委托价格
    "avgPrice": "63684.10", //平均成交价
    "origQty": "0.010", //原始委托数量
    "executedQty": "0.000", //成交量
    "cumQty": "0.000",
    "cumQuote": "0.00000", //成交金额
    "timeInForce": "GTC",
    "type": "MARKET", //订单类型
    "reduceOnly": False, //仅减仓
    "closePosition": False, //是否条件全平仓
    "side": "SELL", //买卖方向
    "positionSide": "SHORT", //持仓方向,SHORT是做空, LONG是做多
    "stopPrice": "0.00", //触发价
    "workingType": "CONTRACT_PRICE",
    "priceProtect": False,
    "origType": "MARKET", //触发前订单类型
    "priceMatch": "NONE",
    "selfTradePreventionMode": "NONE",
    "goodTillDate": 0,
    "updateTime": 1709521590663
}
```


图 3-14　订单数据

合仓：如果第 2 次执行 new_order，交易方向、持仓方向都和之前仓位一致，则将进行合仓操作，也就是合并在同一个仓位，两次的 quantity 值累加，代码如下：

```
#第 1 次开仓做多
Response1 = client.new_order(
    symbol = "BTCUSDT",      #交易对
    side = "SELL",           #交易方向
    type = "MARKET",         #类型
    quantity = 0.01,         #数量
    positionSide = "SHORT", #持仓方向
)
print("开仓做空 1")
print(response1)

#第 2 次开仓做多
Response2 = client.new_order(
    symbol = "BTCUSDT",      #交易对
    side = "SELL",           #交易方向
    type = "MARKET",         #类型
    quantity = 0.01,         #数量
    positionSide = "SHORT", #持仓方向
)
print("开仓做空 2")
print(response2)

#此时仓位中的 quantity 值是 0.02
```

平仓：交易所没有专门的平仓函数，需要再下一个订单，买卖方向和之前仓位买卖方向相反即可，代码如下：

```
#开仓做多
response = client.new_order(
    symbol = "BTCUSDT",      #交易对
    side = "BUY",            #交易方向
    type = "MARKET",         #类型
    quantity = 0.01,         #数量
    positionSide = "LONG",   #持仓方向
)
print("开仓做多")
print(response)

#平仓操作
Response2 = client.new_order(
    symbol = "BTCUSDT",      #交易对
    side = "SELL",           #交易方向和上面的相反即可平仓
    type = "MARKET",         #类型
    quantity = 0.01,         #数量
    positionSide = "LONG",   #持仓方向要和上面一致
)
print("平仓")
print(response2)
```

设置止盈和止损要点：买卖方向要和开仓买卖方向相反，持仓方向要一致，然后设置一个触发价格，当市场价到达这个触发价格时自动平仓，代码如下：

```
#开仓做多
response = client.new_order(
    symbol = "BTCUSDT",              #交易对
    side = "BUY",                    #交易方向为买入
    type = "MARKET",                 #类型
    quantity = 0.01,                 #数量
    positionSide = "LONG",           #持仓方向为做多
)
print("开仓做多")
print(response)

#设置止盈
response = client.new_order(
    symbol = "BTCUSDT",
    side = "SELL",                   #交易方向为卖出
    type = "TAKE_PROFIT_MARKET",     #订单类型为止盈
    timeInForce = "GTC",
    stopPrice = 64100,               #止盈触发价格
    quantity = qty,                  #数量
    positionSide = "LONG",           #持仓方向为做多
)
```

```
print("设置止盈")
print(response)

#设置止损,买卖方向要和开仓买卖方向相反,持仓方向要一致
response = client.new_order(
    symbol = "BTCUSDT",
    side = "SELL",                          #交易方向为卖出
    type = "TRAILING_STOP_MARKET",          #订单类型为止损
    timeInForce = "GTC",
    activatePrice = 62500,                  #止损激活价格
    callbackRate = 1,                       #止损回调比例
    quantity = qty,                         #数量
    positionSide = "LONG",                  #持仓方向为做多
)
print("设置止损")
print(response)
```

3.2.20　合约查询订单 API

查询订单 API 需要的参数见表 3-15。

表 3-15　查询订单 API 需要的参数

参数名称	类型	是否是必需的	描　　述
symbol	String	是	交易对名称,例如 BTCUSDT
orderId	String	是	订单 ID

代码如下：

```
symbol = "BTCUSDT"
orderId = "12345"
#查询单个订单
ord = client.query_order(symbol, orderId)
print(ord)
```

3.2.21　合约取消订单 API

查询订单 API 有两个,一个是按订单 ID 取消订单,另一个是取消所有打开的订单,代码如下：

```
#取消订单
response = client.cancel_order(
    symbol = "BTCUSDT", orderId = "123456", recvWindow = 2000
)
print("取消订单")
print(response)

#取消所有打开的订单
response = client.cancel_open_orders(
```

```
    symbol = "BTCUSDT", recvWindow = 2000
)
print("取消所有打开的订单")
print(response)
```

3.2.22 应用示例：合约 API 综合应用

把前面介绍的合约 API 结合在一起，做一个综合应用，代码如下：

```
#文件名:binanceFuturesCancel.py

from binance.um_futures import UMFutures
from env import getApiKey
import time

#正式环境
#key, secret = getApiKey("apiKey","apiSecret")
#client = UMFutures(key = key, secret = secret,
base_url = "https://api.binance.com")

#测试网
key, secret = getApiKey("testFuturesKey", "testFuturesSecret")
client = UMFutures(key = key, secret = secret,
base_url = "https://testnet.binancefuture.com")

qty = 0.01                          #数量
#开仓做多
response = client.new_order(
    symbol = "BTCUSDT",             #交易对
    side = "BUY",                   #买卖方向为买入
    type = "MARKET",                #订单类型为市价
    quantity = qty,                 #数量
    positionSide = "LONG",          #持仓方向为做多
    #timeInForce = "GTC",
)
print("开仓")
print(response)

#设置止盈
response = client.new_order(
    symbol = "BTCUSDT",             #交易对
    side = "SELL",                  #买卖方向为卖出
    type = "TAKE_PROFIT_MARKET",    #订单类型为市价止盈
    timeInForce = "GTC",
    stopPrice = 64100,              #触发价格
    quantity = qty,                 #数量
    positionSide = "LONG",          #持仓方向为做多
)
print("设置止盈")
```

```
print(response)
ordId = response["orderId"]
print(f"止盈订单 id{ordId}")
#休眠 10s
time.sleep(10)

#查询订单
response = client.query_order(symbol = "BTCUSDT", orderId = ordId)
print("查询订单结果")
print(response)
#订单状态
status = response["status"]


#如果订单状态为 NEW,则取消订单
if status == "NEW":
    response = client.cancel_order(symbol = "BTCUSDT", orderId = ordId, recvWindow = 2000)
    print("取消订单")
    print(response)


#取消所有打开的订单
#response = client.cancel_open_orders(
#symbol = "BTCUSDT", recvWindow = 2000
#)
#print("取消所有打开的订单")
#print(response)
```

3.3　欧易 API

和币安 API 不同,欧易 API 的现货和合约是同一套接口,欧易也有模拟盘,大部分 API 可以在模拟盘上进行测试,使用 API 前要先注册好实盘和模拟盘的 API Key、Secret Key、Passphrase,然后统一写到 config.ini 文件中,示例如下:

```
[keys]
#币安
apiKey = u3YMZu8pdzWXoy2rugrp301nEsgOjmYgxVkeEqn2QfOriMMwquryxpmTw6WCyf9O
apiSecret = olc6INfXgUbBNeZT3niNXsQ7OrcNpOD1EJvAaExD1jGcpdH5wKmt6N980OL1HUom
testKey = NaaGgBf104k0j5eU7bA6rmFmIQLChaRpUloYY7s84r6C13IEIjmxJcjhcgUxOzzu
testSecret = cRnSNAZYJsqp6e3cSy72BTfuXL9JypgEha0DJbDSRIUhJDIE9hv9Lm8LKNxOwQPv
testFuturesKey = 80fcfa736fc58faab1fe7a737afeef8ca0b3869fe853806ef9636463c1467bf3
testFuturesSecret = b9b688e52833da771a49ab2ed4df4eb4c5f9794c761438f77e8e8e986bb6b34a

#欧易
okApiKey = fa2c486a - 9387 - 483f - b074 - 6f3479ff9c59
okApiSecret = C0D68E91FB2AB0B6D7A1BFB16D8A30C9
```

```
okTestKey = 2a076334 - 82ca - 94a8 - 9971 - fbf556863d44
okTestSecret = EE51E9F072DE9B6DB7A41F4EF5E2CFB5
passphrase = Abcd8888!
```

欧易没有官方的 SDK，只推荐了两个第三方 SDK：第 1 个是 Python-OKX，这个 SDK 只支持 Python 语言；第 2 个是 Open-API-SDK-V5，支持 Java、C＃、PHP、Python 等主流编程语言，但是这个 SDK 的 Python 版需要指定在 3.6～3.8，对 3.11 版本支持不好，所以推荐大家使用 Python-OKX，在命令行窗口下输入下面的指令：

```
pip install python - okx
```

本节将讲解用这个 SDK 编写代码来演示欧易 API 的使用方法。

3.3.1 查询钱包余额 API

欧易查询账户余额 API 参数见表 3-16。

表 3-16 欧易查询账户余额 API 参数

参数名称	类型	是否是必需的	描　　述
ccy	String	是	币种，例如 USDT

第 1 步，导入 SDK 的账户模块，代码如下：

```
from okx import Account
```

第 2 步，导入 apiKey、apiSecretKey、passphrase，代码如下：

```
from env import getOkApiKey
apiKey, apiSecretKey, passphrase = getOkApiKey(
   "okTestKey", "okTestSecret", "passphrase"
)
```

第 3 步，使用账户模块的 get_account_balance 方法查询余额，代码如下：

```
result = accountAPI.get_account_balance(ccy = "USDT")
print(result)
```

返回的账户余额数据如图 3-15 所示。

查询钱包余额的完整代码如下：

```
＃文件名:okBalance.py
from okx import Account
from env import getOkApiKey

apiKey, apiSecretKey, passphrase = getOkApiKey(
   "okTestKey", "okTestSecret", "passphrase"
)
```

```
#0 为实盘,1 为模拟盘
accountAPI = Account.AccountAPI(apiKey, apiSecretKey, passphrase, False, flag = "1")
result = accountAPI.get_account_balance(ccy = "USDT")
print(result)
print("可用余额:", result["data"][0]["details"][0]["availBal"])
```

图 3-15　账户余额数据

3.3.2　设置逐仓模式 API

欧易设置逐仓模式 API 参数见表 3-17。

表 3-17　欧易设置逐仓模式 API 参数

参数名称	类型	是否是必需的	描　　述
isoMode	String	是	逐仓保证金划转模式,automatic 为开仓自动划转
type	String	是	业务类型,MARGIN 为币币；CONTRACTS 为合约

设置逐仓模式,代码如下：

```
#文件名:okMargin.py
from okx import Account
from env import getOkApiKey

apiKey, apiSecretKey, passphrase = getOkApiKey(
   "okTestKey", "okTestSecret", "passphrase"
)
```

```
#flag:0 为实盘;flag:1 为模拟盘
accountAPI = Account.AccountAPI(apiKey, apiSecretKey, passphrase, False, flag = "1")
#isoMode:逐仓保证金划转模式;type:业务线类型(MARGIN 币币杠杆,CONTRACTS 合约)
result = accountAPI.set_isolated_mode(isoMode = "automatic", type = "CONTRACTS")
print(result)
#返回结果
#{'code': '0', 'data': [{'isoMode': 'automatic'}], 'msg': ''}
```

3.3.3 设置杠杆倍数 API

欧易设置杠杆倍数 API 参数见表 3-18。

表 3-18 欧易设置杠杆倍数 API 参数

参数名称	类型	是否是必需的	描 述
instId	String	是	产品 ID,例如 BTC-USDT
lever	String	是	杠杆倍数
mgnMode	String	是	保证金模式,isolated 为逐仓; cross 为全仓

设置杠杆倍数,代码如下:

```
#文件名:okLeverage.py
from okx import Account
from env import getOkApiKey

apiKey, apiSecretKey, passphrase = getOkApiKey(
    "okTestKey", "okTestSecret", "passphrase"
)

#flag:0 为实盘;flag:1 为模拟盘
accountAPI = Account.AccountAPI(apiKey, apiSecretKey, passphrase, False, flag = "1")
#instId 为交易对,lever 为杠杆倍数, mgnMode 为逐仓模式
result = accountAPI.set_leverage(instId = "BTC-USDT", lever = "5", mgnMode = "isolated")
print(result)
#返回结果
#{'code': '0', 'data': [{'instId': 'BTC-USDT', 'lever': '5', 'mgnMode': 'isolated', 'posSide':
''}], 'msg': ''}
```

3.3.4 获取深度信息 API

欧易的深度数据需要通过 WebSocket 的订阅方式获取,参数见表 3-19。

表 3-19 欧易获取深度信息 API 参数

参数名称	类型	是否是必需的	描 述
op	String	是	操作,subscribe 为订阅; unsubscribe 为取消
args	Array	是	频道列表,可以包含若干频道

续表

参数名称	类型	是否是必需的	描　　述
>channel	String	是	频道 1,例如 books：深度数据；index-candle：K 线数据
>instType	String	是	产品 1 类型,spot：币币；swap：永续合约

第 1 步,导入 SDK 的 WebSocket 公共数据模块,代码如下：

```
from okx.websocket.WsPublicAsync import WsPublicAsync
```

第 2 步,导入 Python 的异步 IO 框架和 json 模块,代码如下：

```
import asyncio
import json
```

第 3 步,创建公共数据的 WebSocket 连接,如果实盘和模拟盘的 URL 网址不同,则需要根据实际情况进行切换,我们的代码使用模拟盘的 URL,代码如下：

```
#模拟盘的链接
url = "wss://wspap.okex.com:8443/ws/v5/public?brokerId=9999"
#实盘的链接
#url = "wss://ws.okx.com:8443/ws/v5/business"
ws = WsPublicAsync(url=url)
await ws.start()
```

第 4 步,新建一个频道列表,然后创建一个频道名为 books 的深度数据频道,加入频道列表中,进行订阅操作,指定 publicCallback 函数来接收交易的深度数据的推送消息,代码如下：

```
#频道列表
args = []
#第 1 个频道:深度信息
arg1 = {"channel": "books", "instType": "SPOT", "instId": "BTC-USDT"}
#将第 1 个频道加到频道列表里
args.append(arg1)
#订阅,指定 publicCallback 函数来接收推送数据
await ws.subscribe(args, publicCallback)
```

第 5 步,创建接收推送数据的函数,并处理推送过来的数据,代码如下：

```
def publicCallback(message):
    msg = json.loads(message)
    print(msg)
```

接收的深度数据推送结果如图 3-16 所示。

第 6 步,交易所推送的数据是 JSON 字符串格式,需要使用 json 包的 json.loads 方法转换为字典数据,方便提取特定键值,代码如下：

```
def publicCallback(message):
    msg = json.loads(message)
```

```
        print("交易对",arg["instId"])
        data = msg.get("data")
        print("卖出订单",data["ask"])
        print("卖出订单",data["ask"])
```

图 3-16　接收的深度数据推送结果

第 7 步，启动异步函数的方法，代码如下：

```
asyncio.run(main())
```

第 8 步，main 函数前面要加上 async 关键词，代码如下：

```
async def main():
```

第 9 步，在将 JOSN 数据转换为字典数据时，经常会出现找不到键值的错误，原因是推送数据是递增的，有时会缺少一些键值，从而导致程序出现异常错误，因此可以在代码中加入 try except 方式来捕捉异常，这样程序更加健壮，代码如下：

```
try:
        msg = json.loads(message)
        arg = msg.get("arg")
        #判断键值是否存在
        if arg is not None and "instId" in arg:
            print(arg["instId"])
        data = msg.get("data")
        if data is not None:
```

```
            if "ask" in data:
                print(data["ask"])
            if "bid" in data:
                print(data["bid"])
    except json.jsonDecodeError as e:
        print("JSON 解码错误:", e)
    except KeyError as e:
        print(f"键值错误: {e} - the key is not in the JSON structure")
```

完整的获取深度信息的代码如下：

```
# 文件名:okDepth.py
import asyncio
import json
from okx.websocket.WsPublicAsync import WsPublicAsync


def publicCallback(message):
    try:
        msg = json.loads(message)
        arg = msg.get("arg")
        if arg is not None and "instId" in arg:
            print("产品 ID", arg["instId"])

        data = msg.get("data")
        if data is not None:
            if "ask" in data:
                print("卖出订单", data["ask"])
            if "bid" in data:
                print("买入订单", data["bid"])
    except json.jsonDecodeError as e:
        print("JSON 解码错误:", e)
    except KeyError as e:
        print(f"键值错误: {e} - the key is not in the JSON structure")


async def main():
    # 模拟盘 URL
    url = "wss://wspap.okex.com:8443/ws/v5/public?brokerId=9999"
    # 实盘 URL
    # url = "wss://ws.okx.com:8443/ws/v5/business"
    ws = WsPublicAsync(url=url)
    await ws.start()
    args = []
    arg1 = {"channel": "books", "instType": "SPOT", "instId": "BTC-USDT"}
    args.append(arg1)
    await ws.subscribe(args, publicCallback)
    while True:
        await asyncio.sleep(1)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

3.3.5 获取 K 线数据 API

欧易的 K 线数据获取方式和获取深度数据的方式完全一样，不同之处是推送的数据不同。订阅 1 个 K 线频道，代码如下：

```
url = "wss://wsaws.okx.com:8443/ws/v5/business"
ws = WsPublicAsync(url=url)
await ws.start()
args = []
#产品 ID:BTC-USDT
arg1 = {"channel": "index-candle1m", "instType": "SPOT", "instId": "BTC-USDT"}
args.append(arg1)
await ws.subscribe(args, publicCallback)
```

K 线产品名为 index-candle，后面加上时间粒度 1m，表示推送时间的间隔是 1min，更多的时间粒度有[1m/3m/5m/15m/30m/1h/2h/4h]，m 是分钟，h 是小时。

推送的 K 线数据如图 3-17 所示。

图 3-17 推送的 K 线数据

完整的获取 K 线数据的代码如下：

```
#文件名:okTicker.py
import asyncio
import json
from okx.websocket.WsPublicAsync import WsPublicAsync


def publicCallback(message):
    try:
        msg = json.loads(message)
        print("数据", msg)
        print("产品 ID",msg["arg"]["instId"])
```

```
            print("开盘价",msg["data"][0][1])
            print("最高价",msg["data"][0][2])
            print("最低价",msg["data"][0][3])
            print("收盘价",msg["data"][0][4])
        except json.jsonDecodeError as e:
            print("JSON 解码错误:", e)
        except KeyError as e:
            print(f"键值错误: {e} - the key is not in the JSON structure")


    async def main():
        #url = "wss://wspap.okx.com:8443/ws/v5/business?brokerId=9999"
        url = "wss://wsaws.okx.com:8443/ws/v5/business"
        ws = WsPublicAsync(url=url)
        await ws.start()
        args = []
        arg1 = {"channel": "index-candle1m", "instType": "SPOT", "instId": "BTC-USDT"}
        args.append(arg1)
        arg2 = {"channel": "index-candle1m", "instType": "SPOT", "instId": "ETH-BTC"}
        args.append(arg2)
        arg3 = {"channel": "index-candle1m", "instType": "SPOT", "instId": "ETH-USDT"}
        args.append(arg3)
        await ws.subscribe(args, publicCallback)
        while True:
            await asyncio.sleep(1)


    if __name__ == "__main__":
        asyncio.run(main())
```

3.3.6　币币市价下单 API

欧易的下单 API 只有一个，通过不同的参数组合，可以满足币币市价下单、币币限价下单、合约市价下单、合约限价下单等需求，下单 API 常用参数见表 3-20。

表 3-20　欧易下单 API 常用参数

参数名称	类型	是否是必需的	描　　述
instId	String	是	产品 ID，例如 BTC-USDT
tdMode	String	是	交易模式，isolated 为逐仓，cross 为全仓，cash 为非保证金
ccy	String	是	保证金币种
side	String	是	订单方向，buy 为买，sell 为卖
posSide	String	可选	持仓方向，long 为做多，short 为做空
ordType	String	是	订单类型，market 为市价单，limit 为限价单
sz	String	是	委托数量，单位是交易货币数量
px	String	是	委托价格

第 1 步，导入 SDK 的交易模块，代码如下：

```
from okx import Trade
```

第 2 步，用交易模块的 place_order 方法下币币市价单，币币下单的参数组合为 tdMode＝"cash"，ordType＝"market"，不用填 px 价格参数。

币币下单，交易方向是买入，代码如下：

```
result = tradeAPI.place_order(
    instId = "BTC - USDT",          #交易对
    tdMode = "cash",                #模式为币币交易
    side = "buy",                   #买卖方向为买入
    ordType = "market",             #订单类型为市价单
    sz = "20",                      #下单数量，单位是 USDT
)
print("币币市价下单结果", result)
```

下单后返回的结果，如果 code 为 0，则代表下单成功，如图 3-18 所示。

图 3-18　欧易币币下市价单结果

币币下单，交易方向是卖出，代码如下：

```
result2 = tradeAPI.place_order(
    instId = symbol,            #交易对
    tdMode = "cash",            #模式为币币交易
    side = "sell",              #交易方向为卖出
    ordType = "market",         #订单类型为市价单
    sz = fillSz,                #下单数量，单位为 BTC
)
print("币币市价卖出下单结果", result2)
```

注意：如果交易方向是买入，则用 USDT 换 BTC，下单数量为 USDT，如果交易方向是卖出，则用 BTC 换 USDT，下单数量为 BTC。

欧易币币下市价单的完整代码如下：

```
#文件名：okNewSpotMarketOrd.py
from okx import Trade
from okx import Account
```

```
from env import getOkApiKey

apiKey, apiSecretKey, passphrase = getOkApiKey(
    "okTestKey", "okTestSecret", "passphrase"
)


#币币市价下单
def main(symbol, sz):
    accountAPI = Account.AccountAPI(apiKey, apiSecretKey, passphrase, False, flag = "1")
    acc = accountAPI.get_account_balance(ccy = "USDT")
    usdtBalance = float(acc["data"][0]["details"][0]["availBal"])
    print("USDT余额:", usdtBalance)

    tradeAPI = Trade.TradeAPI(
        apiKey, apiSecretKey, passphrase, False, flag = "1"       #0为实盘,1为模拟盘
    )

    result = tradeAPI.place_order(
        instId = symbol,                                          #交易对
        tdMode = "cash",                                          #模式为币币交易
        side = "buy",                                             #买卖方向为买入
        ordType = "market",                                       #订单类型为市价单
        sz = sz,                                                  #下单数量:20个USDT
    )
    print("币币市价买入下单结果", result)
    fillPx, fillSz = getOrd(symbol, result["data"][0]["ordId"])
    print("买入成交价格", fillPx, "成交数量", fillSz)

    result2 = tradeAPI.place_order(
        instId = symbol,                                          #交易对
        tdMode = "cash",                                          #模式为币币交易
        side = "sell",                                            #买卖方向为卖出
        ordType = "market",                                       #订单类型为市价单
        sz = fillSz,                                              #下单数量,单位为BTC
    )
    print("币币市价卖出下单结果", result2)
    fillPx, fillSz = getOrd(symbol, result2["data"][0]["ordId"])
    print("卖出成交价格", fillPx, "成交数量", fillSz)
    #卖出成交价格为69789.4,成交数量为0.00028657

    acc2 = accountAPI.get_account_balance(ccy = "USDT")
    usdtBalance2 = float(acc2["data"][0]["details"][0]["availBal"])
    print("买卖后盈利:", usdtBalance2 - usdtBalance, "个USDT")
    #买卖后盈利: -0.02002820535835781个USDT


#获取订单信息
def getOrd(instId, orderId):
```

```
    tradeAPI = Trade.TradeAPI(
        apiKey, apiSecretKey, passphrase, False, flag = "1" #0 为实盘,1 为模拟盘
    )
    result = tradeAPI.get_order(instId, orderId)
    print("获取订单信息", result)
    fillPx = 0                                          #成交价
    fillSz = 0                                          #成交数量
    for i in range(len(result["data"])):
        fillPx += float(result["data"][i]["fillPx"])    #累计成交价
        fillSz += float(result["data"][i]["fillSz"])    #累计成交数量
    return fillPx / len(result["data"]), fillSz


if __name__ == "__main__":
    #下单 20 个 USDT
    main("BTC-USDT", 20)
```

3.3.7 币币限价下单 API

欧易币币限价下单的参数组合为 tdMode="cash",ordType="limit",px=委托价格。欧易币币下限价单的完整代码如下：

```
#文件名:okNewSpotLimitOrd.py
from okx import Trade
from env import getOkApiKey

apiKey, apiSecretKey, passphrase = getOkApiKey(
    "okTestKey", "okTestSecret", "passphrase"
)

#币币限价下单
def main():
    tradeAPI = Trade.TradeAPI(
        apiKey, apiSecretKey, passphrase, False, flag = "1"       #0 为实盘,1 为模拟盘
    )
    result = tradeAPI.place_order(
        instId = "BTC-USDT",                                      #交易对
        tdMode = "cash",                                          #币币交易
        side = "buy",                                             #买入
        ordType = "limit",                                        #限价
        sz = "0.01",                                              #数量
        px = "60000",                                             #委托价格
    )
    print("币币限价下单结果", result)


if __name__ == "__main__":
    main()
```

下单后返回的结果，如果 code 为 0，则代表下单成功，如图 3-19 所示。

图 3-19　欧易币币下限价单结果

3.3.8　合约市价开仓和平仓 API

欧易合约开仓下单的参数组合比较复杂，新手经常因不知如何正确地选择参数组合，而导致开仓或平仓出错，开仓的参数组合为：做多 side ="buy" 并且 posSide="long"；做空 side ="sell"并且 posSide="short"。

市价开仓参数：ordType="market"；sz 参数的单位是张数，1 张=0.001BTC；px 为委托价格，不填。

如果欧易交易对后面有 SWAP，则表示这是永续合约交易对，合约的市价开仓做多的代码如下：

```
result = tradeAPI.place_order(
    instId = "BTC - USDT - SWAP",          # 交易对
    ccy = "USDT",                          # 保证金币种
    tdMode = "isolated",                   # 模式为逐仓
    side = "buy",                          # 交易方向为买入
    posSide = "long",                      # 持仓方向,long 为做多,short 为做空
    ordType = "market",                    # 订单类型为市价单
    sz = qty,                              # 下单数量,单位是张数,1 张是 0.001 个 BTC
)
print("市价开仓做多结果", result)
```

合约的市价开仓做空的代码如下：

```
result = tradeAPI.place_order(
    instId = "BTC - USDT - SWAP",          # 交易对
    ccy = "USDT",                          # 保证金币种
    tdMode = "isolated",                   # 模式为逐仓
    side = "sell",                         # 交易方向为卖出
    posSide = "short",                     # 持仓方向,long 为做多,short 为做空
    ordType = "market",                    # 订单类型为市价单
    sz = qty,                              # 下单数量,单位为张数,1 张是 0.001 个 BTC
)
print("市价开仓做空结果", result)
```

刚看欧易 API 时会发现根本没有专门的平仓 API,后来才知道是用下单的 API 实现平仓,欧易合约平仓下单的参数组合为:平多 side ="sell",posSide="long";平空 side ="buy",posSide="short"。

合约的平多代码如下:

```
result = tradeAPI.place_order(
    instId = "BTC - USDT - SWAP",       #交易对
    ccy = "USDT",                       #保证金币种
    tdMode = "isolated",                #模式为逐仓
    side = "sell",                      #平仓多单
    posSide = "long",                   #持仓方向,long 为平多,short 为平空
    ordType = "market",                 #订单类型为市价单
    sz = qty,                           #平仓数量
)
print("市价开仓平多结果", result)
```

合约的平空代码如下:

```
result = tradeAPI.place_order(
    instId = "BTC - USDT - SWAP",       #交易对
    ccy = "USDT",                       #保证金币种
    tdMode = "isolated",                #模式为逐仓
    side = "sell",                      #平仓多单
    posSide = "long",                   #持仓方向,long 为平多,short 为平空
    ordType = "market",                 #订单类型为市价单
    sz = qty,                           #平仓数量
)
print("市价开仓平空结果", result)
```

欧易合约市价开仓平仓的完整代码如下:

```
#文件名:okNewSwapMarketOrd.py
from okx import Trade
from env import getOkApiKey
import time

apiKey, apiSecretKey, passphrase = getOkApiKey(
   "okTestKey", "okTestSecret", "passphrase"
)
tradeAPI = Trade.TradeAPI(
   apiKey, apiSecretKey, passphrase, False, flag = "1"   #0 为实盘,1 为模拟盘
)


#市价开仓平多
def NewOrd(qty):
   result = tradeAPI.place_order(
      instId = "BTC - USDT - SWAP",                      #交易对
      ccy = "USDT",                                      #保证金币种
```

```
        tdMode = "isolated",                    # 模式为逐仓
        side = "buy",                           # 买卖方向为买入
        posSide = "long",                       # 持仓方向,long 为平多,short 为平空
        ordType = "market",                     # 订单类型为市价单
        sz = qty,                               # 下单数量
    )
    print("市价开仓结果", result)


# 市价平仓平多
def CloseOrd(qty):
    result = tradeAPI.place_order(
        instId = "BTC - USDT - SWAP",           # 交易对
        ccy = "USDT",                           # 保证金币种
        tdMode = "isolated",                    # 模式为逐仓
        side = "sell",                          # 买卖方向为卖出
        posSide = "long",                       # 持仓方向,long 为做多,short 为做空
        ordType = "market",                     # 订单类型为市价单
        sz = qty,                               # 下单数量
    )
    print("市价开仓结果", result)


# 合约市价下单
def main():
    qty = "1"                                   # 下单数量为张数
    NewOrd(qty)                                 # 开仓
    time.sleep(20)                              # 休眠 20s
    CloseOrd(qty)                               # 平仓


if __name__ == "__main__":
    main()
```

3.3.9　合约限价开仓 API

限价开仓参数：ordType="limit",px 委托价格必填。需要注意的是,委托价格不能距离市场成交价过近,否则下单会失败。

合约的限价开仓做多的代码如下：

```
result = tradeAPI.place_order(
    instId = instId,            # 交易对
    ccy = "USDT",               # 保证金币种
    tdMode = "isolated",        # 模式为逐仓
    side = "buy",               # 买卖方向为买入
    posSide = "long",           # 持仓方向,long 为做多,short 为做空
    ordType = "limit",          # 订单类型为市价单
```

```
    px = "63000",              #委托价格
    sz = qty,                  #下单数量
)
print("限价做多结果", result)
```

3.3.10 合约止盈止损单 API

欧易的合约止盈止损单,需要在开仓后才可下单,合约的止盈止损单参数见表 3-21。

表 3-21 欧易合约的止盈止损单参数

参数名称	类型	是否是必需的	描述
tpTriggerPx	String	是	止盈触发价格
tpOrdPx	String	是	止盈价格
tpTriggerPxType	String	是	止盈触发价类型：last 为最新价格,index 为指数价格,mark 为标记价格
slTriggerPx	String	是	止损触发价格
slOrdPx	String	是	止损价格
slTriggerPxType	String	是	止损触发价类型：last 为最新价格,index 为指数价格,mark 为标记价格

合约的止盈止损单的代码如下：

```
result = tradeAPI.place_order(
    instId = instId,
    tdMode = "isolated",
    side = "buy",
    posSide = "long",
    ordType = "limit",
    sz = qty,                          #数量
    px = "63100.0",
    #attachAlgoOrds = arr
    tpTriggerPx = "63110.0",           #触发价格
    tpOrdPx = "64000.0",               #止盈价格
    tpTriggerPxType = "last",          #止盈触发价类型:last 为最新价格,index 为指数价格,mark 为标
                                       #记价格
    slTriggerPx = "63000.0",           #触发价格
    slOrdPx = "62000.0",               #止损价格
    slTriggerPxType = "last",          #止损触发价类型:last 为最新价格,index 为指数价格,mark 为标
                                       #记价格
)
print("止盈单结果", result)
```

3.3.11 查询订单信息 API

下单后并不能马上得到订单的完整数据,需要通过这个查询订单 API 再次查询,才可以获得开仓平均价格、订单成交状态等订单详细信息,查询订单信息的参数见表 3-22。