目　录

- order_volume - 按指定量委托
- order_value - 按指定价值委托
- order_percent - 按总资产指定比例委托
- order_target_volume - 调仓到目标持仓量
- order_target_value - 调仓到目标持仓额
- order_target_percent - 调仓到目标持仓比例（总资产的比例）
- order_close_all - 平当前所有可平持仓
- order_cancel - 委托撤单
- order_close_all - 平当前所有可平持仓
- order_cancel_all - 撤销所有委托
- place_order - 按指定量委托
- get_orders - 查询所有委托
- get_unfinished_orders - 查询未结委托
- get_execution_reports - 查询成交
- get_cash - 查询资金
- get_position - 查询持仓

两融业务成员函数

- credit_buying_on_margin - 融资买入
- credit_short_selling - 融券卖出
- credit_repay_share_by_buying_share - 买券还券
- credit_repay_cash_by_selling_share - 卖券还款
- credit_buying_on_collateral - 担保品买入
- credit_selling_on_collateral - 担保品卖出
- credit_repay_share_directly - 直接还券
- credit_repay_cash_directly - 直接还款
- credit_collateral_in - 担保品转入
- credit_collateral_out - 担保品转出
- credit_get_collateral_instruments - 查询担保证券
- credit_get_borrowable_instruments - 查询融券标的证券
- credit_get_borrowable_instruments_positions - 查询融券账户头寸
- credit_get_contracts - 查询融资融券合约
- credit_get_cash - 查询融资融券资金

算法交易成员函数

- order_algo - 委托算法单
- algo_order_cancel - 撤单算法委托
- algo_order_pause - 暂停/恢复算法单
- get_algo_orders - 查询算法委托
- get_algo_child_orders - 查询算法子委托

指引

快速新建策略

- 打开终端后，登陆掘金账号点击研究策略，新建策略
  或者点击右上角新建策略
  
- 新建一个典型默认账号交易策略
  新建C++的默认账号交易策略
  

编译策略

- 打开新建策略文件目录
  策略文件目录内容可以拷贝到本地其他盘符也可以进行编译生成
  
- 策略文件说明:
  `gmskd:  sdk目录`
  `Stretegy: 策略源码目录`
  `readme.txt 说明文件`
  
- 打开工程文件 `sln` 文件
  需要用`visual studio`打开工程文件
  
- 编写策略
  打开`main.c`文件，可进行策略编辑
  

  编译并运行策略
  
- 查看运行结果
  掘金客户端中关闭新建策略窗口并打开回测结果列表
  

  查看回测结果

回测相关数据指标

策略框架应该是这样的

- 继承策略基类
- 重改关注事件
- 在on_init里订阅行情，初始化
- 在main里实例化一个派生类对像
- 设置token,策略id,和mode
- 开始运行

继承策略基类

```
class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}
private:
};
```

重改关注事件

```
class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

    //重写on_init事件，进行策略开发
    void on_init()
    {
        cout << "on_init" << endl;

        return;
    }
private:
};
```

在on_init里订阅行情，初始化

```
class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

    //重写on_init事件，进行策略开发
```

```
    void on_init()
    {
        cout << "on_init" << endl;
        subscribe("SHSE.600000", "tick");
        return;
    }
private:
};
```

在main里实例化一个派生类对像

```
MyStrategy s;
```

设置token,策略id,和mode

1. 获取token: 打开客户端->点击右上角用户头像 -> 系统设置 -> 复制token
2. 获取策略id: 打开客户端->策略研究->右上角新建策略->新建C/C++策略->复制策略ID
3. 策略模式:
   MODE_LIVE(实时)=1
   MODE_BACKTEST(回测)=2

```
//设置策略id
s.set_strategy_id("strategy_id");
//设置token
s.set_token("token");
//设置回测模式
s.set_mode(MODE_BACKTEST);
//回测模式相关设置
s.set_backtest_config("2016-07-11 17:20:00", "2017-07-11 17:30:00", 1000000, 1, 0, 0, 0, 1);
```

开始运行

```
s.run();
```

订阅行情策略示例

源文件

```
#include <iostream>
#include "strategy.h"

using namespace std;

class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

```

```
    //重写on_init事件, 进行策略开发
    void on_init()
    {
        cout << "on_init" << endl;
        //订阅行情数据
        subscribe("SHSE.600000", "tick");

        return;
    }

   //接收tick行情事件
   void on_tick(Tick *tick)
   {
      cout<< "代码                                " << tick->symbol << endl
         << "utc时间,精确到毫秒                  " << tick->created_at << endl
         << "最新价                              " << tick->price << endl
         << "开盘价                              " << tick->open << endl
         << "最高价                              " << tick->high << endl
         << "最低价                              " << tick->low << endl
         << "成交总量                            " << tick->cum_volume << endl
         << "成交总金额 / 最新成交额, 累计值     " << tick->cum_amount << endl
         << "合约持仓量(期), 累计值              " << tick->cum_position << endl
         << "瞬时成交额                          " << tick->last_amount << endl
         << "瞬时成交量                          " << tick->last_volume << endl
         << "保留)交易类型, 对应多开, 多平等类型 " << tick->trade_type << endl
         << "报价                                " << tick->quotes << endl;
   }

private:
};

int main(int argc, char *argv[])
{
    MyStrategy s;
    s.set_strategy_id("07ea5d21-59ab-11e8-83bf-94c69161828a");
    s.set_token("39624b0f1916ae0b2a4cb1f2d13704368badf576");
    s.set_mode(MODE_BACKTEST);
    s.set_backtest_config("2017-07-11 14:20:00", "2017-07-11 15:30:00",
        1000000, 1, 0, 0, 0, 1);
    s.run();
    cout << "回测完成！" << endl;
    getchar();
    return 0;
}
```

典型场景

空策略

```
////////////////////////////////////////////////////////////////////////////
//空策略

#include <iostream>
#include "strategy.h"

using namespace std;

class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

    //重写on_init事件，进行策略开发
    void on_init()
    {
        cout << "on_init" << endl;
        //订阅行情数据
        subscribe("SHSE.600000", "tick");

        return;
    }

    //接收tick行情事件
   void on_tick(Tick *tick)
   {
      cout << "代码                                  " << tick->symbol << endl
         << "utc时间，精确到毫秒                  " << tick->created_at << endl
         << "最新价                                " << tick->price << endl
         << "开盘价                                " << tick->open << endl
         << "最高价                                " << tick->high << endl
         << "最低价                                " << tick->low << endl
         << "成交总量                              " << tick->cum_volume << endl
         << "成交总金额 / 最新成交额，累计值       " << tick->cum_amount << endl
         << "合约持仓量(期)，累计值                " << tick->cum_position << endl
         << "瞬时成交额                            " << tick->last_amount << endl
         << "瞬时成交量                            " << tick->last_volume << endl
         << "保留)交易类型，对应多开，多平等类型   " << tick->trade_type << endl
         << "报价                                  " << tick->quotes << endl;
```

```
    }

private:
};

int main(int argc, char *argv[])
{
    MyStrategy s;
    s.set_strategy_id("07ea5d21-59ab-11e8-83bf-94c69161828a");
    s.set_token("39624b0f1916ae0b2a4cb1f2d13704368badf576");
    s.set_mode(MODE_BACKTEST);
    s.set_backtest_config("2017-07-11 14:20:00", "2017-07-11 15:30:00",1000000, 1, 0, 0, 0, 1);
    s.run();
    cout << "回测完成！" << endl;
    getchar();
    return 0;
}
```

定时任务

```
////////////////////////////////////////////////////////////////////////////
//定时任务
//策略描述：
//典型如选股交易。比如，策略每日收盘前10分钟执行：选股->决策逻辑->交易->退出。可能无需订阅实时数据。

#include <iostream>
#include "strategy.h"

using namespace std;

class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

    //重写on_init事件，进行策略开发
    void on_init()
    {
        cout << "on_init" << endl;

        //设置定时任务
        schedule("1d","13:24:00");
        return;
    }

    //定时任务触发事件
    void on_schedule(const char *data_rule, const char *time_rule)
    {
        //购买200股浦发银行股票
        Order o = order_volume("SHSE.600000", 200, 1, 2, 1, 0);
    }
```

```
33.
34.     //回测完成事件
35.     void on_backtest_finished()
36.     {
37.         cout << "on_backtest_finished" << endl;
38.     }
39.
40.     //回测完成后收到绩效报告
41.     void on_indicator(Indicator *indicator)
42.     {
43.         cout << "on_indicator" << endl
44.             << "账号ID:        "  << indicator->account_id        << endl
45.             << "累计收益率:    "  << indicator->pnl_ratio         << endl
46.             << "年化收益率:    "  << indicator->pnl_ratio_annual  << endl
47.             << "夏普比率:      "  << indicator->sharp_ratio       << endl
48.             << "最大回撤:      "  << indicator->max_drawdown      << endl
49.             << "风险比率:      "  << indicator->risk_ratio        << endl
50.             << "开仓次数:      "  << indicator->open_count        << endl
51.             << "平仓次数:      "  << indicator->close_count       << endl
52.             << "盈利次数:      "  << indicator->win_count         << endl
53.             << "亏损次数:      "  << indicator->lose_count        << endl
54.             << "胜率:          "  << indicator->win_ratio         << endl
55.             << "指标创建时间:  "  << indicator->created_at        << endl
56.             << "指标变更时间:  "  << indicator->updated_at        << endl;
57.     }
58.
59. private:
60. };
61.
62. int main(int argc, char *argv[])
63. {
64.     MyStrategy s;
65.     s.set_strategy_id("4727c864-84da-11e8-81b2-7085c223669d");
66.     s.set_token("39624b0f1916ae0b2a4cb1f2d13704368badf576");
67.     s.set_mode(MODE_BACKTEST);
68.     s.set_backtest_config("2016-07-11 17:20:00", "2017-07-11 17:30:00",1000000, 1, 0,
    0, 0, 1);
69.     s.run();
70.
71.     return 0;
72. }
```

数据事件驱动

```
1. ////////////////////////////////////////////////////////////////////////
2. //数据事件驱动
3. //策略描述：
4. //典型如选股交易策略。比如，策略每日收盘前10分钟执行：选股->决策逻辑->交易->退出。可能无需订阅实时数据
5.
6.
7. #include <iostream>
8. #include "strategy.h"
9.
```

```cpp
using namespace std;

class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

    //重写on_init事件, 进行策略开发
    void on_init()
    {
        cout << "on_init" << endl;

        //订阅浦发银行, bar频率为一天
        subscribe("SHSE.600000", "1d");

        return;
    }

    void on_bar(Bar *bar)
    {
        cout << "代码:            "   <<   bar->symbol       << endl
            << "bar的开始时间:    "   <<   bar->bob          << endl
            << "bar的结束时间:    "   <<   bar->eob          << endl
            << "开盘价:           "   <<   bar->open         << endl
            << "收盘价:           "   <<   bar->close        << endl
            << "最高价:           "   <<   bar->high         << endl
            << "最低价:           "   <<   bar->low          << endl
            << "成交量:           "   <<   bar->volume       << endl
            << "成交金额:         "   <<   bar->amount       << endl
            << "前收盘价:         "   <<   bar->pre_close    << endl
            << "持仓量:           "   <<   bar->position     << endl
            << "bar频度:          "   <<   bar->frequency    << endl;
    }
private:
};

int main(int argc, char *argv[])
{
    MyStrategy s;
    s.set_strategy_id("07ea5d21-59ab-11e8-83bf-94c69161828a");
    s.set_token("39624b0f1916ae0b2a4cb1f2d13704368badf576");
    s.set_mode(MODE_BACKTEST);
    s.set_backtest_config("2016-07-11 17:20:00", "2017-07-11 17:30:00",1000000, 1, 0, 0, 0, 1);
    s.run();
    return 0;
}
```

默认交易账号

```cpp
////////////////////////////////////////////////////////////////////////////
//默认账号交易
```

```
//策略描述：
//默认账号进行交易，下单时不指定account

#include <iostream>
#include "strategy.h"

using namespace std;

class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

    //重写on_init事件，进行策略开发
    void on_init()
    {
        cout << "on_init" << endl;
        subscribe("SHSE.600000,SZSE.000001", "1d");

        return;
    }

    void on_bar(Bar *bar)
    {
        //不指定account 使用默认账户下单
        order_volume(bar->symbol, 200, 1, 2, 1, 0);
    }

    //回测完成事件
    void on_backtest_finished()
    {
        cout << "on_backtest_finished" << endl;
    }

    //回测完成后收到绩效报告
    void on_indicator(Indicator *indicator)
    {
        cout << "on_indicator" << endl
            << "账号ID:       " << indicator->account_id << endl
            << "累计收益率:   " << indicator->pnl_ratio << endl
            << "年化收益率:   " << indicator->pnl_ratio_annual << endl
            << "夏普比率:     " << indicator->sharp_ratio << endl
            << "最大回撤:     " << indicator->max_drawdown << endl
            << "风险比率:     " << indicator->risk_ratio << endl
            << "开仓次数:     " << indicator->open_count << endl
            << "平仓次数:     " << indicator->close_count << endl
            << "盈利次数:     " << indicator->win_count << endl
            << "亏损次数:     " << indicator->lose_count << endl
            << "胜率:         " << indicator->win_ratio << endl
            << "指标创建时间:  " << indicator->created_at << endl
            << "指标变更时间:  " << indicator->updated_at << endl;
    }

private:
```

```
};

int main(int argc, char *argv[])
{
    MyStrategy s;
    s.set_strategy_id("ba8785aa-8641-11e8-98cb-7085c223669d");
    s.set_token("39624b0f1916ae0b2a4cb1f2d13704368badf576");
    s.set_mode(MODE_BACKTEST);
    s.set_backtest_config("2016-07-11 17:20:00", "2017-07-11 17:30:00",1000000, 1, 0, 0, 0, 1);
    s.run();
    return 0;
}
```

显示指定交易账号

```
////////////////////////////////////////////////////////////////////////////
//显示指定交易账号
//策略描述：
//下单时指定交易账号，account参数传账号id或者账号标题

#include <iostream>
#include "strategy.h"

using namespace std;

class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

    //重写on_init事件，进行策略开发
    void on_init()
    {
        cout << "on_init" << endl;
        subscribe("SHSE.600000,SZSE.000001", "1d");

        return;
    }

    void on_bar(Bar *bar)
    {
        //不指定account 使用默认账户下单
        order_volume(bar->symbol, 200, 1, 2, 1, 0, "ba8785aa-8641-11e8-98cb-7085c223669d");
    }

    //回测完成事件
    void on_backtest_finished()
    {
        cout << "on_backtest_finished" << endl;
    }
```

```
37.
38.     //回测完成后收到绩效报告
39.     void on_indicator(Indicator *indicator)
40.     {
41.         cout << "on_indicator" << endl
42.             << "账号ID:        " << indicator->account_id << endl
43.             << "累计收益率:    " << indicator->pnl_ratio << endl
44.             << "年化收益率:    " << indicator->pnl_ratio_annual << endl
45.             << "夏普比率:      " << indicator->sharp_ratio << endl
46.             << "最大回撤:      " << indicator->max_drawdown << endl
47.             << "风险比率:      " << indicator->risk_ratio << endl
48.             << "开仓次数:      " << indicator->open_count << endl
49.             << "平仓次数:      " << indicator->close_count << endl
50.             << "盈利次数:      " << indicator->win_count << endl
51.             << "亏损次数:      " << indicator->lose_count << endl
52.             << "胜率:          " << indicator->win_ratio << endl
53.             << "指标创建时间: " << indicator->created_at << endl
54.             << "指标变更时间: " << indicator->updated_at << endl;
55.     }
56.
57. private:
58. };
59.
60. int main(int argc, char *argv[])
61. {
62.     MyStrategy s;
63.     s.set_strategy_id("ba8785aa-8641-11e8-98cb-7085c223669d");
64.     s.set_token("39624b0f1916ae0b2a4cb1f2d13704368badf576");
65.     s.set_mode(MODE_BACKTEST);
66.     s.set_backtest_config("2016-07-11 17:20:00", "2017-07-11 17:30:00",1000000, 1, 0,
    0, 0, 1);
67.     s.run();
68.     return 0;
69. }
```

模式选择

```
1. ////////////////////////////////////////////////////////////////////////////
2. //模式选择
3. //策略描述：
4. //策略支持两种运行模式，实时模式和回测模式，用户需要在运行策略时选择模式，执行run函数时mode=1 表示回测
   模式，mode=0表示实时模式
5.
6. #include <iostream>
7. #include "strategy.h"
8.
9. using namespace std;
10.
11. class MyStrategy :public Strategy
12. {
13. public:
14.     MyStrategy() {}
15.     ~MyStrategy(){}
```

```
16.
17.     //重写on_init事件，进行策略开发
18.     void on_init()
19.     {
20.         cout << "on_init" << endl;
21.         subscribe("SHSE.600000", "tick");
22.
23.         return;
24.     }
25.
26.     void on_tick(Tick *tick)
27.     {
28.         cout << "代码:                          " << tick->symbol << endl
29.             << "utc时间，精确到毫秒:           " << tick->created_at << endl
30.             << "最新价:                        " << tick->price << endl
31.             << "开盘价:                        " << tick->open << endl
32.             << "最高价:                        " << tick->high << endl
33.             << "最低价:                        " << tick->low << endl
34.             << "成交总量                        " << tick->cum_volume << endl
35.             << "成交总金额/最新成交额,累计值:  " << tick->cum_amount << endl
36.             << "合约持仓量(期),累计值:         " << tick->cum_position << endl
37.             << "瞬时成交额:                    " << tick->last_amount << endl
38.             << "瞬时成交量:                    " << tick->last_volume << endl
39.             << "交易类型,对应多开,多平等类型:  " << tick->trade_type << endl;
40.     }
41.
42. private:
43. };
44.
45. int main(int argc, char *argv[])
46. {
47.     MyStrategy s;
48.     s.set_strategy_id("ba8785aa-8641-11e8-98cb-7085c223669d");
49.     s.set_token("39624b0f1916ae0b2a4cb1f2d13704368badf576");
50.     // mode = MODE_LIVE 实时模式
51.     // mode = MODE_BACKTEST  回测模式，指定回测开始时间backtest_start_time和结束时间
    backtest_end_time
52.     //s.set_backtest_config("2016-07-11 17:20:00", "2017-07-11 17:30:00",1000000, 1, 0,
    0, 0, 1);
53.
54.     s.set_mode(MODE_LIVE);
55.
56.     s.run();
57.     return 0;
58. }
```

数据研究

```
1. ////////////////////////////////////////////////////////////////////////
2. //数据研究
3. //策略描述：
4. //无需实时数据驱动策略，无需交易下单，只是取数据的场景
5.
```

```cpp
#include <iostream>
#include "strategy.h"

using namespace std;

class MyStrategy :public Strategy
{
public:
    MyStrategy() {}
    ~MyStrategy(){}

    //重写on_init事件, 进行策略开发
    void on_init()
    {
        cout << "on_init" << endl;

        DataArray<Tick>* ht = history_ticks("SZSE.000002", "2017-07-11 10:20:00", "2017-07-11 10:30:00");
        if (ht->status() == 0)
        {
            for (int i = 0; i < ht->count(); i++)
            {
                cout << "代码:                                " << ht->at(i).symbol << endl
                    << "utc时间, 精确到毫秒:                 " << ht->at(i).created_at << endl
                    << "最新价:                               " << ht->at(i).price << endl
                    << "开盘价:                               " << ht->at(i).open << endl
                    << "最高价:                               " << ht->at(i).high << endl
                    << "最低价:                               " << ht->at(i).low << endl
                    << "成交总量:                             " << ht->at(i).cum_volume << endl
                    << "成交总金额 / 最新成交额, 累计值:      " << ht->at(i).cum_amount << endl
                    << "合约持仓量(期), 累计值:               " << ht->at(i).cum_position << endl
                    << "瞬时成交额:                           " << ht->at(i).last_amount << endl
                    << "瞬时成交量:                           " << ht->at(i).last_volume << endl
                    << "保留)交易类型, 对应多开, 多平等类型:  " << ht->at(i).trade_type << endl
                    << "报价:                                 " << ht->at(i).quotes << endl;
            }
        }

        return;
    }

private:
};

```

```
int main(int argc, char *argv[])
{
    MyStrategy s;
    s.set_strategy_id("ba8785aa-8641-11e8-98cb-7085c223669d");
    s.set_token("39624b0f1916ae0b2a4cb1f2d13704368badf576");
    s.set_mode(MODE_BACKTEST);
    s.set_backtest_config("2017-07-11 10:20:00", "2017-07-11 10:30:00",1000000, 1, 0, 0, 0, 1);

    s.run();
    return 0;
}
```

重要概念

symbol - 代码标识

掘金代码(**symbol**)是掘金平台用于唯一标识交易标的代码,

格式为: 交易所代码.交易标代码, 比如深圳平安的**symbol** 示例: `SZSE.000001`

交易所代码

目前掘金支持国内的7个交易所, 各交易所的代码缩写如下:

市场中文名	市场代码
上交所	SHSE
深交所	SZSE
中金所	CFFEX
上期所	SHFE
大商所	DCE
郑商所	CZCE
上海国际能源交易中心	INE

交易标的代码

交易表代码是指交易所给出的交易标的代码, 包括股票, 期货, 期权, 指数, 基金等代码。

具体的代码请参考交易所的给出的证券代码定义

策略运行模式

策略支持两种运行模式, 实时模式和回测模式,用户需要在运行策略时选择模式.

实时模式

订阅行情服务器推送的实时行情，也就是交易所的实时行情，只在交易时段提供。

回测模式

订阅指定时段、指定交易代码、指定数据类型的行情，行情服务器将按指定条件全速回放对应的行情数据。适用的场景是策略回测阶段，快速验证策略的绩效是否符合预期。

基类原型

策略类简介

策略类集成了行情、交易和事件的接口，用户的策略都从此类继承实现自己的业务逻辑。每个进程只能实例化一个策略类对象。

策略类定义

```
1.
2. class GM_CLASS Strategy
3. {
4. public:
5.     Strategy(const char *token, const char *strategy_id, int mode);
6.     Strategy();
7.     virtual ~Strategy();
8.
9. public: //基础函数
10.
11.     //运行策略
12.     int run();
13.
14.     //停止策略
15.     void stop();
16.
17.     //设置策略ID
18.     void set_strategy_id(const char *strategy_id);
19.
20.     //设置用户token
21.     void set_token(const char *token);
22.
23.     //设置策略运行模式
24.     void set_mode(int mode);
25.
26.     //定时任务
27.     int schedule(const char *data_rule, const char *time_rule);
28.
29.     //当前时间
30.     double now();
31.
32.     //设置回测参数
33.     int set_backtest_config(
34.         const char  *start_time,
35.         const char  *end_time,
36.         double initial_cash = 1000000,
37.         double transaction_ratio = 1,
38.         double commission_ratio = 0,
39.         double slippage_ratio = 0,
40.         int    adjust = 0,
```

```
        int     check_cache = 1
    );

public: //数据函数

        // 订阅行情
    int subscribe(const char *symbols, const char * frequency, bool
unsubscribe_previous = false);

    // 退订行情
    int unsubscribe(const char *symbols, const char * frequency);


public: //交易函数

    //查询交易账号
    DataArray<Account>* get_accounts();

    //查询指定交易账号状态
    int get_account_status(const char *account, AccountStatus &as);

    //查询所有交易账号状态
    DataArray<AccountStatus>* get_all_account_status();

    //按指定量委托
    Order order_volume(const char *symbol, int volume, int side, int order_type, int
position_effect, double price = 0, const char *account = NULL);

    //按指定价值委托
    Order order_value(const char *symbol, double value, int side, int order_type, int
position_effect, double price = 0, const char *account = NULL);

    //按总资产指定比例委托
    Order order_percent(const char *symbol, double percent, int side, int order_type,
int position_effect, double price = 0, const char *account = NULL);

    //调仓到目标持仓量
    Order order_target_volume(const char *symbol, int volume, int position_side, int
order_type, double price = 0, const char *account = NULL);

    //调仓到目标持仓额
    Order order_target_value(const char *symbol, double value, int position_side, int
order_type, double price = 0, const char *account = NULL);

    //调仓到目标持仓比例（总资产的比例）
    Order order_target_percent(const char *symbol, double percent, int position_side,
int order_type, double price = 0, const char *account = NULL);

    //平当前所有可平持仓
    DataArray<Order>* order_close_all();

    //委托撤单
    int order_cancel(const char *cl_ord_id, const char *account = NULL);

    //撤销所有委托
    int order_cancel_all();

```

```
91.     //委托下单
92.     Order place_order(const char *symbol, int volume, int side, int order_type, int position_effect, double price = 0, int order_duration = 0, int order_qualifier = 0, double stop_price = 0, int order_business = 0, const char *account = NULL);
93. 
94.     //盘后定价交易
95.     Order order_after_hour(const char *symbol, int volume, int side, double price, const char *account = NULL);
96. 
97.     //查询委托
98.     DataArray<Order>* get_orders(const char *account = NULL);
99. 
100.    //查询未结委托
101.    DataArray<Order>* get_unfinished_orders(const char *account = NULL);
102. 
103.    //查询成交
104.    DataArray<ExecRpt>* get_execution_reports(const char *account = NULL);
105. 
106.    //查询资金
107.    DataArray<Cash>* get_cash(const char *accounts = NULL);
108. 
109.    //查询持仓
110.    DataArray<Position>* get_position(const char *account = NULL);
111. 
112.    //委托算法单
113.    AlgoOrder order_algo(const char *symbol, int volume, int position_effect, int side, int order_type, double price, AlgoParam &algo_param, const char *account = NULL);
114. 
115.    //撤单算法委托
116.    int algo_order_cancel(const char *cl_ord_id, const char *account = NULL);
117. 
118.    //暂停/恢复算法单
119.    int algo_order_pause(const char *cl_ord_id, int status, const char *account = NULL);
120. 
121.    //查询算法委托
122.    DataArray<AlgoOrder>* get_algo_orders(const char *account = NULL);
123. 
124.    //查询算法子委托
125.    DataArray<Order>* get_algo_child_orders(const char *cl_ord_id, const char *account = NULL);
126. 
127.    //功能号调用
128.    int raw_func(const char *account, const char *func_id, const char *func_args, char*&rsp);
129. 
130. 
131.    /* 两融业务 */
132. 
133.    //融资买入
134.    Order credit_buying_on_margin(int position_src, const char *symbol, int volume, double price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
135.    //融券卖出
136.    Order credit_short_selling(int position_src, const char *symbol, int volume, double
```

```
    price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown,
    int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
137.    //买券还券
138.    Order credit_repay_share_by_buying_share(const char *symbol, int volume, double
    price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown,
    int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
139.    //卖券还款
140.    Order credit_repay_cash_by_selling_share(const char *symbol, int volume, double
    price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown,
    int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
141.    //担保品买入
142.    Order credit_buying_on_collateral(const char *symbol, int volume, double price, int
    order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int
    order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
143.    //担保品卖出
144.    Order credit_selling_on_collateral(const char *symbol, int volume, double price,
    int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int
    order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
145.    //直接还券
146.    Order credit_repay_share_directly(const char *symbol, int volume, const char
    *account = NULL);
147.    //直接还款
148.    int credit_repay_cash_directly(double amount, const char *account = NULL, double
    *actual_repay_amount = NULL, char *error_msg_buf = NULL, int buf_len = 0);
149.    //担保品转入
150.    Order credit_collateral_in(const char *symbol, int volume, const char *account =
    NULL);
151.    //担保品转出
152.    Order credit_collateral_out(const char *symbol, int volume, const char *account =
    NULL);
153.    //查询担保证券
154.    DataArray<CollateralInstrument>* credit_get_collateral_instruments(const char
    *account = NULL);
155.    //查询标的证券, 可做融券标的股票列表
156.    DataArray<BorrowableInstrument>* credit_get_borrowable_instruments(int
    position_src, const char *account = NULL);
157.    //查询券商融券账户头寸, 可用融券的数量
158.    DataArray<BorrowableInstrumentPosition>*
    credit_get_borrowable_instruments_positions(int position_src, const char *account =
    NULL);
159.    //查询融资融券合约
160.    DataArray<CreditContract>* credit_get_contracts(int position_src, const char
    *account = NULL);
161.    //查询融资融券资金
162.    int credit_get_cash(CreditCash &cash, const char *account = NULL);
163.
164.    /* 新股业务 */
165.
166.    //新股申购
167.    Order ipo_buy(const char *symbol, int volume, double price, const char *account =
    NULL);
168.    //查询客户新股申购额度
169.    int ipo_get_quota(double &quota, const char *account = NULL);
```

```
170.     //查询当日新股清单
171.     DataArray<IPOInstruments>* ipo_get_instruments(const char *account = NULL);
172.     //配号查询
173.     DataArray<IPOMatchNumber>* ipo_get_match_number(const char *account = NULL);
174.     //中签查询
175.     DataArray<IPOLotInfo>* ipo_get_lot_info(const char *account = NULL);
176.
177.     /* 基金业务 */
178.
179.     //ETF申购
180.     Order fund_etf_buy(const char *symbol, int volume, double price, const char
    *account = NULL);
181.     //ETF赎回
182.     Order fund_etf_redemption(const char *symbol, int volume, double price, const char
    *account = NULL);
183.     //基金认购
184.     Order fund_subscribing(const char *symbol, int volume, const char *account = NULL);
185.     //基金申购
186.     Order fund_buy(const char *symbol, int volume, const char *account = NULL);
187.     //基金赎回
188.     Order fund_redemption(const char *symbol, int volume, const char *account = NULL);
189.
190.     /* 债券业务 */
191.
192.     //国债逆回购
193.     Order bond_reverse_repurchase_agreement(const char *symbol, int volume, double
    price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown,
    int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
194.
195. public: //策略参数类函数
196.
197.     //添加参数
198.     int add_parameters(Parameter *params, int count);
199.
200.     //删除参数
201.     int del_parameters(const char *keys);
202.
203.     //设置参数
204.     int set_parameters(Parameter *params, int count);
205.
206.     //获取参数
207.     DataArray<Parameter>* get_parameters();
208.
209.     //设置标的
210.     int set_symbols(const char *symbols);
211.
212.     //获取标的
213.     DataArray<Symbol>* get_symbols();
214.
215.
216. public: //事件函数
217.
218.         //初始化完成
219.         virtual void on_init();
220.         //收到Tick行情
221.         virtual void on_tick(Tick *tick);
```

```
222.         //收到bar行情
223.         virtual void on_bar(Bar *bar);
224.         //收到逐笔成交（L2行情时有效）
225.         virtual void on_l2transaction(L2Transaction *l2transaction);
226.         //收到逐笔委托（深交所L2行情时有效）
227.         virtual void on_l2order(L2Order *l2order);
228.         //收到委托队列（上交所L2行情时有效）
229.         virtual void on_l2order_queue(L2OrderQueue *l2queue);
230.         //委托变化
231.         virtual void on_order_status(Order *order);
232.         //执行回报
233.         virtual void on_execution_report(ExecRpt *rpt);
234.         //算法委托变化
235.         virtual void on_algo_order_status(AlgoOrder *order);
236.         //参数变化
237.         virtual void on_parameter(Parameter *param);
238.         //定时任务触发
239.         virtual void on_schedule(const char *data_rule, const char *time_rule);
240.         //回测完成后收到绩效报告
241.         virtual void on_backtest_finished(Indicator *indicator);
242.         //实盘账号状态变化
243.         virtual void on_account_status(AccountStatus *account_status);
244.         //错误产生
245.         virtual void on_error(int error_code, const char *error_msg);
246.         //收到策略停止信号
247.         virtual void on_stop();
248.         //数据已经连接上
249.         virtual void on_market_data_connected();
250.         //交易已经连接上
251.         virtual void on_trade_data_connected();
252.         //数据连接断开了
253.         virtual void on_market_data_disconnected();
254.         //交易连接断开了
255.         virtual void on_trade_data_disconnected();
256.
257. };
```

基本成员函数

Strategy - 构造函数

构造策略对象。

函数原型：

```
    //带参数的构造函数
    Strategy(const char *token, const char *strategy_id, int mode);

    //不带参数的构造函数
    Strategy();
```

参数：

参数名	类型	说明
token	const char *	系统权限密钥,可在终端系统设置-密钥管理中生成
strategy_id	const char *	策略ID,在终端中获取
mode	int	实时模式:MODE_LIVE, 回测模式:MODE_BACKTEST

注意事项：

- 一个进程只能构造一个策略对象。
- 如用 Strategy() 构造对象时，`token`，`strategy_id`，`mode`三个参数可以从`set_token`，`set_strategy_id`，`set_mode`成员函数传入。

run - 运行策略

运行策略。只有调用run后，才会驱动所有的事件，如行情接入与交易事件。

函数原型：

```
int run();
```

参数：

参数名	类型	说明

返回值	int	如果策略正常退出返回0， 非正常退出返回错误码

注意事项：

调用run会阻塞线程，策略进入事件驱动状态，所以所有初始操作（如读配置文件，分配缓冲区等）都应该在run之前完成，如果run退出，意味着策略运行结束，整个进程应该就此退出。

stop - 停止策略

用于停止策略， 也就是如果调用run()之后， 在某个事件响应中调用stop, 这是run就是退出，并返回0。

函数原型：

```
1. void stop();
```

set_strategy_id - 设置策略ID

函数原型：

```
1. void set_strategy_id(const char *strategy_id);
```

参数：

参数名	类型	说明
strategy_id	const char *	策略ID,在终端中获取

注意事项：

不管是从构造函数传入还成员函数传入， `token` ， `strategy_id` ， `mode` 都是必须要设置的参数。

set_token - 设置用户token

函数原型：

```
1. void set_token(const char *token);
```

参数：

参数名	类型	说明
token	const char *	系统权限密钥,可在终端系统设置-密钥管理中生成

注意事项：

不管是从构造函数传入还成员函数传入， `token` ， `strategy_id` ， `mode` 都是必须要设置的参数。

set_mode - 设置策略运行模式

函数原型：

```
void set_mode(int mode);
```

参数:

参数名	类型	说明
mode	int	实时模式:MODE_LIVE, 回测模式:MODE_BACKTEST

注意事项:

不管是从构造函数传入还成员函数传入, `token` , `strategy_id` , `mode` 都是必须要设置的参数。

schedule - 预设定时任务

在指定时间自动执行策略算法, 通常用于选股类型策略。schedule一般在on_init中调用。如果schedule预设成功, 那么达成预设时间条件时, on_schedule会被调用, 并在on_schedule的参数中返回设置的 `data_rule` 和 `time_rule` 。schedule可以调用多次, 设置多个不同定时任务。

函数原型:

```
int schedule(const char *data_rule, const char *time_rule);
```

参数:

参数名	类型	说明
data_rule	const char *	n + 时间单位, 可选'd/w/m' 表示n天/n周/n月
time_rule	const char *	执行算法的具体时间 (%H:%M:%S 格式)
返回值	int	预设成功返回0, 预设失败返回错误码

事例:

```

    #每天的19:06:20执行
    schedule(date_rule='1d', time_rule='19:06:20')

    #每月的第一个交易日的09:40:00执行
    schedule(date_rule='1m', time_rule='9:40:00')
```

注意事项:

- 现在 `data_rule` 暂只支持 1d,1w,1m, 任意n后续会支持。
- 1w,1m 只在回测中支持, 实盘模式中不支持。

now - 获取当前时间

实时模式下, 返回当前的系统时间。回测模式下, 返回当前的回测时间点。格式是utc时间戳。

函数原型:

```
double now();
```

参数:

参数名	类型	说明
返回值	double	当前utc时间戳

set_backtest_config - 设置回测参数

如果mode设置为回测模式，则在调用run之前，需要先设置本函数设置回测参数。在实时模式下，该调用被忽略。

函数原型:

```
int set_backtest_config(
    const char  *start_time,
    const char  *end_time,
    double initial_cash = 1000000,
    double transaction_ratio = 1,
    double commission_ratio = 0,
    double slippage_ratio = 0,
    int    adjust = 0,
    int    check_cache = 1
);
```

参数:

参数名	类型	说明
start_time	const char *	回测开始时间 (%Y-%m-%d %H:%M:%S格式)
end_time	const char *	回测结束时间 (%Y-%m-%d %H:%M:%S格式)
initial_cash	double	回测初始资金，默认1000000
transaction_ratio	double	回测成交比例，默认1.0，即下单100%成交
commission_ratio	double	回测佣金比例，默认0
slippage_ratio	double	回测滑点比例，默认0
adjust	int	回测复权方式(默认不复权) ADJUST_NONE(不复权)=0 ADJUST_PREV(前复权)=1 ADJUST_POST(后复权)=2
check_cache	int	回测是否使用缓存: 1 - 使用， 0 - 不使用; 默认使用

注意:
start_time和end_time中月,日,时,分,秒均可以只输入个位数，例: `'2016-6-7 9:55:0'` 或 `'2017-8-1 14:6:0'` ,但若对应位置为零，则0不可被省略,比如不能输入 `"2017-8-1 14:6: "`

行情成员函数

subscribe - 订阅行情

订阅行情推送，实时模式下订阅实时行情推送，回测模式下订阅历史行情推送。订阅tick会触发on_tick回调，订阅bar则触发on_bar回调。

函数原型：

```
int subscribe(const char *symbols, const char * frequency, bool unsubscribe_previous = false);
```

参数：

参数名	类型	说明
symbols	const char *	订阅标的代码列表，字符串格式，如有多个代码，中间用 , (英文逗号) 隔开
frequency	const char *	频率，支持 ‘tick’, ‘1d’, ‘15s’, ‘30s’ 等
unsubscribe_previous	bool	是否取消过去订阅的symbols，默认false不取消，输入true则取消所有原来的订阅。
返回值	int	订阅成功返回0， 订阅失败返回错误码

示例：

```
//订阅 SHSE.600000和 SZSE.000001 两个标的的tick行情
subscribe(symbols="SHSE.600000,SHSE.600004", frequency="tick");

//订阅 SHSE.600000和 SZSE.000001 两个标的的1分钟bar
subscribe(symbols="SHSE.600000,SHSE.600004", frequency="60s");
```

unsubscribe - 退订行情

退订已经订阅行情推送， 与subscribe作用相返。

函数原型：

```
int unsubscribe(const char *symbols, const char * frequency);
```

参数：

参数名	类型	说明
symbols	const char *	退订标的代码列表，字符串格式，如有多个代码，中间用 , (英文逗号) 隔开
frequency	const char *	频率，支持 ‘tick’, ‘1d’, ‘15s’, ‘30s’ 等

返回值	int	退订成功返回0， 退订失败返回错误码

示例：

```
//退订 SHSE.600000和 SZSE.000001 两个标的的tick行情
unsubscribe(symbols="SHSE.600000,SHSE.600004", frequency="tick");
```

普通交易成员函数

get_accounts - 查询交易账号

用于查询交易账号配置信息。多用于实盘时，策略同时关联多个交易账号的时候，获取所有交易账号的信息，所返回的账号id(`account_id`)用于后续各个交易api的入参， 即指定操作某个交易账户。
如果关联的交易账号只有一个， 一般用不到此函数。

函数原型：

```
1. DataArray<Account>* get_accounts();
```

参数：

参数名	类型	说明
返回值	`DataArray<Account>*`	一个Account结构数组

order_volume - 按指定量委托

按指定量委托， 如果调用成功，后续委托单状态变化将会触发on_order_status回调。

函数原型：

```
1. Order order_volume(const char *symbol, int volume, int side, int order_type, int
   position_effect, double price = 0, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的

volume	int	委托数量
side	int	委托方向 参见 `enum OrderSide`
order_type	int	委托类型 参见 `enum OrderType`
position_effect	int	开平类型 参见 `enum PositionSide`
price	double	委托价格
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例:

```
//以11块的价格限价买入10000股浦发银行
Order o = order_volume("SHSE.600000", 10000, OrderSide_Buy, OrderType_Limit,
PositionEffect_Open, 11);
```

注意:

1.仅支持一个标的代码，若交易代码输入有误，终端会拒绝此单，并显示 `委托代码不正确` 。

2.若下单数量输入有误，终端会拒绝此单，并显示 `委托量不正确` 。股票买入最小单位为 `100` ，卖出最小单位为 `1` ，如存在不足100股的持仓一次性卖出;期货买卖最小单位为 `1` ， `向下取整` 。

3.若仓位不足，终端会拒绝此单，显示 `仓位不足` 。平仓时股票默认 `平昨仓` ，期货默认 `平今仓` 。应研究需要， `股票也支持卖空操作` 。

4.Order_type优先级高于price,若指定OrderTpye_Market下市价单，使用价格为最新一个tick中的最新价，price参数失效。则price参数失效。若OrderTpye_Limit限价单，仿真模式价格错误，终端拒绝此单，显示委托价格错误， `回测模式下对价格无限制` 。

5.函数调用成功并不意味着委托已经成功，只是意味委托单已经成功发出去， 委托是否成功根据on_order_status，或get_order来判断。

order_value - 按指定价值委托

按指定价值委托， 如果调用成功，后续委托单状态变化将会触发on_order_status回调。

函数原型:

```
Order order_value(const char *symbol, double value, int side, int order_type, int
position_effect, double price = 0, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
value	int	股票价值

side	int	委托方向 参见 `enum OrderSide`
order_type	int	委托类型 参见 `enum OrderType`
position_effect	int	开平类型 参见 `enum PositionSide`
price	double	委托价格
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例:

```
//下限价单，以11元每股的价格买入价值为100000元的SHSE.600000，根据volume = value / price,计算并取整得到volume = 9000
Order o = order_value("SHSE.600000", 100000, OrderSide_Buy, OrderType_Limit, PositionEffect_Open, 11);
```

注意:

1.仅支持一个标的代码，若交易代码输入有误，终端会拒绝此单，并显示 `委托代码不正确` 。

2.根据指定价值计算购买标的数量，即 `value/price` 。股票买卖最小单位为 `100` ，不足100部分 `向下取整` ，如存在不足100的持仓一次性卖出;期货买卖最小单位为 `1` ， `向下取整` 。

3.若仓位不足，终端会拒绝此单，显示 `仓位不足` 。平仓时股票默认 `平昨仓` ，期货默认 `平今仓` 。应研究需要， `股票也支持卖空操作` 。

4.Order_type优先级高于price,若指定OrderTpye_Market下市价单，使用价格为最新一个tick中的最新价，price参数失效。则price参数失效。若OrderTpye_Limit限价单，仿真模式价格错误，终端拒绝此单，显示委托价格错误， `回测模式下对价格无限制` 。

5.函数调用成功并不意味着委托已经成功，只是意味委托单已经成功发出去， 委托是否成功根据on_order_status，或get_order来判断。

order_percent - 按总资产指定比例委托

按总资产指定比例委托， 如果调用成功，后续委托单状态变化将会触发on_order_status回调。

函数原型:

```
Order order_percent(const char *symbol, double percent, int side, int order_type, int position_effect, double price = 0, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
percent	double	委托占总资产比例

side	int	委托方向 参见 `enum OrderSide`
order_type	int	委托类型 参见 `enum OrderType`
position_effect	int	开平类型 参见 `enum PositionSide`
price	double	委托价格
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例:

```
//当前总资产为1000000。下限价单，以11元每股的价格买入SHSE.600000,期望买入比例占总资产的10%，根据volume = nav * precent / price 计算取整得出volume = 9000

Order o = order_percent("SHSE.600000", 0.1, OrderSide_Buy, OrderType_Limit, PositionEffect_Open, 11);
```

注意:

1.仅支持一个标的代码，若交易代码输入有误，终端会拒绝此单，并显示 `委托代码不正确` 。

2.根据指定比例计算购买标的数量,即 `(nav*precent)/price` ,股票买卖最小单位为 `100` ，不足100部分 `向下取整` ，如存在不足100的持仓一次性卖出;期货买卖最小单位为 `1` ， `向下取整` 。

3.若仓位不足，终端会拒绝此单，显示 `仓位不足` 。平仓时股票默认 `平昨仓` ，期货默认 `平今仓` 。应研究需要， `股票也支持卖空操作` 。

4.Order_type优先级高于price,若指定OrderTpye_Market下市价单，使用价格为最新一个tick中的最新价，price参数失效。则price参数失效。若OrderTpye_Limit限价单，仿真模式价格错误，终端拒绝此单，显示委托价格错误， `回测模式下对价格无限制` 。

5.函数调用成功并不意味着委托已经成功，只是意味委托单已经成功发出去， 委托是否成功根据on_order_status，或get_order来判断。

order_target_volume - 调仓到目标持仓量

调仓到目标持仓量， 如果调用成功，后续委托单状态变化将会触发on_order_status回调。

函数原型:

```
Order order_target_volume(const char *symbol, int volume, int position_side, int order_type, double price = 0, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	期望的最终数量

position_side	int	持仓方向 参见 `enum PositionSide`)
order_type	int	委托类型 参见 `enum OrderType`
price	double	委托价格
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例:

```
//当前SHSE.600000多方向持仓量为0，期望持仓量为10000，下单量为期望持仓量 - 当前持仓量 = 10000

Order o = order_target_volume("SHSE.600000", 10000, PositionSide_Long, OrderType_Limit,
11);
```

注意:

1.仅支持一个标的代码，若交易代码输入有误，终端会拒绝此单，并显示 `委托代码不正确` 。

2.根据目标数量计算下单数量，系统判断开平仓类型。若下单数量有误，终端拒绝此单，并显示 `委托量不正确` 。若实际需要买入数量为0，则订单会被拒绝， `终端无显示，无回报` 。股票买卖最小单位为 `100` ，不足100部分 `向下取整` ，如存在不足100的持仓一次性卖出;期货买卖最小单位为 `1` ， `向下取整` 。

3.若仓位不足，终端会拒绝此单，显示 `仓位不足` 。平仓时股票默认 `平昨仓` ，期货默认 `平今仓` 。应研究需要， `股票也支持卖空操作` 。

4.Order_type优先级高于price,若指定OrderTpye_Market下市价单，使用价格为最新一个tick中的最新价，price参数失效。则price参数失效。若OrderTpye_Limit限价单，仿真模式价格错误，终端拒绝此单，显示委托价格错误， `回测模式下对价格无限制` 。

5.函数调用成功并不意味着委托已经成功，只是意味委托单已经成功发出去， 委托是否成功根据on_order_status，或get_order来判断。

order_target_value - 调仓到目标持仓额

调仓到目标持仓额， 如果调用成功，后续委托单状态变化将会触发on_order_status回调。

函数原型:

```
Order order_target_value(const char *symbol, double value, int position_side, int
order_type, double price = 0, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
value	int	期望的股票最终价值
position_side	int	持仓方向 参见 `enum PositionSide`)

order_type	int	委托类型 参见 `enum OrderType`
price	double	委托价格
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//当前SHSE.600000多方向当前持仓量为0，目标持有价值为100000的该股票，根据value / price 计算取整得出目标持仓量volume为9000，目标持仓量 - 当前持仓量 = 下单量为9000

Order o = order_target_value("SHSE.600000", 100000, PositionSide_Long, OrderType_Limit, 11);
```

注意：

1.仅支持一个标的代码，若交易代码输入有误，终端会拒绝此单，并显示 `委托代码不正确` 。

2.根据目标数量计算下单数量，系统判断开平仓类型。若下单数量有误，终端拒绝此单，并显示 `委托量不正确` 。若实际需要买入数量为0，则订单会被拒绝， `终端无显示，无回报` 。股票买卖最小单位为 `100` ，不足100部分 `向下取整` ，如存在不足100的持仓一次性卖出;期货买卖最小单位为 `1` ， `向下取整` 。

3.若仓位不足，终端会拒绝此单，显示 `仓位不足` 。平仓时股票默认 `平昨仓` ，期货默认 `平今仓` 。应研究需要， `股票也支持卖空操作` 。

4.Order_type优先级高于price,若指定OrderTpye_Market下市价单，使用价格为最新一个tick中的最新价，price参数失效。则price参数失效。若OrderTpye_Limit限价单，仿真模式价格错误，终端拒绝此单，显示委托价格错误， `回测模式下对价格无限制` 。

5.函数调用成功并不意味着委托已经成功，只是意味委托单已经成功发出去， 委托是否成功根据on_order_status，或get_order来判断。

order_target_percent - 调仓到目标持仓比例（总资产的比例）

调仓到目标持仓比例（总资产的比例）， 如果调用成功，后续委托单状态变化将会触发on_order_status回调。

函数原型：

```
Order order_target_percent(const char *symbol, double percent, int position_side, int order_type, double price = 0, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
percent	double	期望的最终占总资产比例
position_side	int	持仓方向 参见 `enum PositionSide`)

order_type	int	委托类型 参见 `enum OrderType`
price	double	委托价格
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` ，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例:

```
//当前总资产价值为1000000，目标为以11元每股的价格买入SHSE.600000的价值占总资产的10%，根据volume = nav * percent / price 计算取整得出应持有9000股。当前该股持仓量为零，因此买入量为9000

Order o = order_target_percent("SHSE.600000", 0.1, PositionSide_Long, OrderType_Limit, 11);
```

注意:

1.仅支持一个标的代码，若交易代码输入有误，终端会拒绝此单，并显示 `委托代码不正确` 。

2.根据目标比例计算下单数量，为占 `总资产(nav)` 比例，系统判断开平仓类型。若下单数量有误，终端拒绝此单，并显示 `委托量不正确` 。若实际需要买入数量为0，则本地拒绝此单， `终端无显示，无回报` 。股票买卖最小单位为 `100` ，不足100部分 `向下取整` ，如存在不足100的持仓一次性卖出;期货买卖最小单位为 `1` ， `向下取整` 。

3.若仓位不足，终端会拒绝此单，显示 `仓位不足` 。平仓时股票默认 `平昨仓` ，期货默认 `平今仓` 。应研究需要， `股票也支持卖空操作` 。

4.Order_type优先级高于price,若指定OrderTpye_Market下市价单，使用价格为最新一个tick中的最新价，price参数失效。则price参数失效。若OrderTpye_Limit限价单，仿真模式价格错误，终端拒绝此单，显示委托价格错误， `回测模式下对价格无限制` 。

5.函数调用成功并不意味着委托已经成功，只是意味委托单已经成功发出去， 委托是否成功根据on_order_status，或get_order来判断。

order_close_all - 平当前所有可平持仓

平当前所有可平持仓， 如果调用成功，后续委托单状态变化将会触发on_order_status回调

函数原型:

```
DataArray<Order>* order_close_all();
```

参数:

参数名	类型	说明
返回值	`DataArray<Order>*`	一个Order结构数组

order_cancel - 委托撤单

撤销单个委托单， 如果调用成功，后续委托单状态变化将会触发on_order_status回调

函数原型：

```
int order_cancel(const char *cl_ord_id, const char *account = NULL);
```

参数：

参数名	类型	说明
cl_ord_id	const char *	委托单的客户id，可以在下单或查单时获得
account	const char *	实盘账号id，关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	int	成功返回0， 失败返回错误码

order_close_all - 平当前所有可平持仓

平当前所有可平持仓，如果调用成功，后续委托单状态变化将会触发on_order_status回调

函数原型：

```
DataArray<Order>* order_close_all();
```

参数：

参数名	类型	说明
返回值	`DataArray<Order>*`	一个Order结构数组

order_cancel_all - 撤销所有委托

撤销所有委托，如果调用成功，后续委托单状态变化将会触发on_order_status回调

函数原型：

```
int order_cancel_all();
```

参数：

参数名	类型	说明
返回值	int	成功返回0， 失败返回错误码

place_order - 按指定量委托

按指定量委托，如果调用成功，后续委托单状态变化将会触发on_order_status回调。

函数原型：

```
Order place_order(const char *symbol, int volume, int side, int order_type, int
position_effect, double price = 0, int order_duration = 0, int order_qualifier = 0,
```

```
    double stop_price = 0, int order_business = 0, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
side	int	委托方向 参见 `enum OrderSide`
order_type	int	委托类型 参见 `enum OrderType`
position_effect	int	开平类型 参见 `enum PositionSide`
price	double	委托价格
order_duration	int	委托时间属性 参见 `enum OrderDuration`
order_qualifier	int	委托成交属性 参见 `enum OrderQualifier`
stop_price	double	止损价
order_business	int	委托业务类型 参见 `enum OrderBusiness`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例:

```
1. //以市价类型“五档即成剩撤”买入10000股浦发银行
2. Order o = place_order("SHSE.600000", 10000, OrderSide_Buy, OrderType_Market,
   PositionEffect_Open, 0, OrderDuration_Unknown, OrderQualifier_B5TL);
```

注意:

1.仅支持一个标的代码，若交易代码输入有误，终端会拒绝此单，并显示 `委托代码不正确` 。

2.若下单数量输入有误，终端会拒绝此单，并显示 `委托量不正确` 。股票买入最小单位为 `100` ，卖出最小单位为 `1` ，如存在不足100股的持仓一次性卖出;期货买卖最小单位为 `1` ， `向下取整` 。

3.若仓位不足，终端会拒绝此单，显示 `仓位不足` 。平仓时股票默认 `平昨仓` ，期货默认 `平今仓` 。应研究需要， `股票也支持卖空操作` 。

4.Order_type优先级高于price,若指定OrderTpye_Market下市价单，使用价格为最新一个tick中的最新价，price参数失效。则price参数失效。若OrderTpye_Limit限价单，仿真模式价格错误，终端拒绝此单，显示委托价格错误， `回测模式下对价格无限制` 。

5.函数调用成功并不意味着委托已经成功，只是意味委托单已经成功发出去， 委托是否成功根据on_order_status，或get_order来判断。

get_orders - 查询所有委托

查询所有委托单

函数原型:

```
1. DataArray<Order>* get_orders(const char *account = NULL);
```

参数:

参数名	类型	说明
account	const char *	账号ID `account_id`, 如果输入为NULL, 则返回所有账号的委托
返回值	`DataArray<Order>*`	一个Order结构数组

get_unfinished_orders - 查询未结委托

查询所有未结委托

函数原型:

```
1. DataArray<Order>* get_unfinished_orders(const char *account = NULL);
```

参数:

参数名	类型	说明
account	const char *	账号ID `account_id`, 如果输入为NULL, 则返回所有账号的委托
返回值	`DataArray<Order>*`	一个Order结构数组

get_execution_reports - 查询成交

查询所有成交

函数原型:

```
1. DataArray<ExecRpt>* get_execution_reports(const char *account = NULL);
```

参数:

参数名	类型	说明
account	const char *	账号ID `account_id`, 如果输入为NULL, 则返回所有账号的成交
返回值	`DataArray<ExecRpt>*`	一个ExecRpt结构数组

get_cash - 查询资金

查询资金

函数原型:

```
1. DataArray<Cash>* get_cash(const char *accounts = NULL);
```

参数:

参数名	类型	说明
account	const char *	账号ID `account_id`, 如果输入为NULL, 则返回所有账号的资金
返回值	`DataArray<Cash>*`	一个Cash结构数组

get_position - 查询持仓

查询所有持仓

函数原型:

```
1. DataArray<Position>* get_position(const char *account = NULL);
```

参数:

参数名	类型	说明
account	const char *	账号ID `account_id`, 如果输入为NULL, 则返回所有账号的持仓
返回值	`DataArray<Position>*`	一个Position结构数组

两融业务成员函数

credit_buying_on_margin - 融资买入

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
1.     Order credit_buying_on_margin(const char *symbol, int volume, double price, int
   order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int
   order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	委托价格
order_type	int	委托类型 参见 `enum OrderType`
order_duration	int	委托时间属性 参见 `enum OrderDuration`
order_qualifier	int	委托成交属性 参见 `enum OrderQualifier`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述， 其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
1. //以价格11.9融资买入10000股浦发银行
```

```
Order o = credit_buying_on_margin("SHSE.600000", 10000, 11.9);
```

credit_short_selling - 融券卖出

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
    Order credit_short_selling(const char *symbol, int volume, double price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	委托价格
order_type	int	委托类型 参见 `enum OrderType`
order_duration	int	委托时间属性 参见 `enum OrderDuration`
order_qualifier	int	委托成交属性 参见 `enum OrderQualifier`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//以价格11.9融券卖出10000股浦发银行
Order o = credit_short_selling("SHSE.600000", 10000, 11.9);
```

注意：
融券卖出一般不支持市价单，以柜台为准

credit_repay_share_by_buying_share - 买券还券

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
    Order credit_repay_share_by_buying_share(const char *symbol, int volume, double price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	委托价格
order_type	int	委托类型 参见 `enum OrderType`
order_duration	int	委托时间属性 参见 `enum OrderDuration`
order_qualifier	int	委托成交属性 参见 `enum OrderQualifier`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//以价格11.9买入10000股浦发银行还券
Order o = credit_repay_share_by_buying_share("SHSE.600000", 10000, 11.9);
```

credit_repay_cash_by_selling_share - 卖券还款

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
    Order credit_repay_cash_by_selling_share(const char *symbol, int volume, double price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	委托价格
order_type	int	委托类型 参见 `enum OrderType`
order_duration	int	委托时间属性 参见 `enum OrderDuration`
order_qualifier	int	委托成交属性 参见 `enum OrderQualifier`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//以价格11.9卖出10000股浦发银行还款
Order o = credit_repay_share_by_buying_share("SHSE.600000", 10000, 11.9);
```

credit_buying_on_collateral - 担保品买入

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
    Order credit_buying_on_collateral(const char *symbol, int volume, double price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	委托价格
order_type	int	委托类型 参见 `enum OrderType`
order_duration	int	委托时间属性 参见 `enum OrderDuration`
order_qualifier	int	委托成交属性 参见 `enum OrderQualifier`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述， 其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//以价格11.9买入10000股浦发银行
Order o = credit_buying_on_collateral("SHSE.600000", 10000, 11.9);
```

credit_selling_on_collateral - 担保品卖出

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
    Order credit_selling_on_collateral(const char *symbol, int volume, double price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	委托价格
order_type	int	委托类型 参见 `enum OrderType`
order_duration	int	委托时间属性 参见 `enum OrderDuration`
order_qualifier	int	委托成交属性 参见 `enum OrderQualifier`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例:

```
//以价格11.9卖出10000股浦发银行
Order o = credit_selling_on_collateral("SHSE.600000", 10000, 11.9);
```

credit_repay_share_directly - 直接还券

注: 融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型:

```
    Order credit_repay_share_directly(const char *symbol, int volume, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例:

```
//还券10000股浦发银行
Order o = credit_repay_share_directly("SHSE.600000", 10000);
```

credit_repay_cash_directly - 直接还款

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
1.     int credit_repay_cash_directly(double amount, const char *account = NULL, double
   *actual_repay_amount = NULL, char *error_msg_buf = NULL, int buf_len = 0);
```

参数：

参数名	类型	说明
amount	double	还款金额
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
actual_repay_amount	double*	出参，返回值，如果还款成功，返回实际的还款金额
error_msg_buf	char *	出参，返回值，如果还款失败，返回错误信息，内存需要用户分配
buf_len	int	指定error_msg_buf 空间大小
返回值	int	成功返回0，否则返回非0错误

示例：

```
1. //还款 100000块
2. double actual_repay_amount;
3. char error_msg_buf[256] = {0};
4. int ret = credit_repay_cash_directly("SHSE.600000", 100000, &actual_repay_amount,
   error_msg_buf, sizeof(error_msg_buf));
```

credit_collateral_in - 担保品转入

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
1.     Order credit_collateral_in(const char *symbol, int volume, const char *account =
   NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
account	const char	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL

	*	置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` ，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//担保品转入10000股浦发银行
Order o = credit_collateral_in("SHSE.600000", 10000);
```

credit_collateral_out - 担保品转出

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
    Order credit_collateral_out(const char *symbol, int volume, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` ，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//担保品转出10000股浦发银行
Order o = credit_collateral_out("SHSE.600000", 10000);
```

credit_get_collateral_instruments - 查询担保证券

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
DataArray<CollateralInstrument>* credit_get_collateral_instruments(const char *account = NULL);
```

参数：

参数名	类型	说明
account	const char *	账号ID `account_id`，如果输入为NULL，则返回所有账号的委托
返回值	`DataArray<CollateralInstrument>*`	一个CollateralInstrument结构数组

credit_get_borrowable_instruments - 查询融券标的证券

注：融资融券暂时仅支持实盘委托，不支持仿真交易

查询标的证券，可做融券标的股票列表

函数原型：

```
1. DataArray<BorrowableInstrument>* credit_get_borrowable_instruments(const char *account
   = NULL);
```

参数：

参数名	类型	说明
account	const char *	账号ID `account_id`，如果输入为NULL，则返回所有账号的委托
返回值	`DataArray<BorrowableInstrument>*`	一个BorrowableInstrument结构数组

credit_get_borrowable_instruments_positions - 查询融券账户头寸

注：融资融券暂时仅支持实盘委托，不支持仿真交易

查询券商融券账户头寸，可用融券的数量

函数原型：

```
1. DataArray<BorrowableInstrumentPosition>*
   credit_get_borrowable_instruments_positions(const char *account = NULL);
```

参数：

参数名	类型	说明
account	const char *	账号ID `account_id`，如果输入为NULL，则返回所有账号的委托
返回值	`DataArray<BorrowableInstrumentPosition>*`	一个BorrowableInstrumentPosition结构数组

credit_get_contracts - 查询融资融券合约

注：融资融券暂时仅支持实盘委托，不支持仿真交易

查询融资融券合约，负债

函数原型：

```
DataArray<CreditContract>* credit_get_contracts(const char *account = NULL);
```

参数：

参数名	类型	说明
account	const char *	账号ID `account_id`，如果输入为NULL，则返回所有账号的委托
返回值	`DataArray<CreditContract>*`	一个CreditContract结构数组

credit_get_cash - 查询融资融券资金

注：融资融券暂时仅支持实盘委托，不支持仿真交易

函数原型：

```
int credit_get_cash(CreditCash &cash, const char *account = NULL);
```

参数：

参数名	类型	说明
cash	CreditCash	出参，返回资金信息
account	const char *	账号ID `account_id`，如果输入为NULL，则返回所有账号的委托
返回值	int	成功返回0，否则返回非0错误

示例：

```
CreditCash cash;
int ret = credit_get_cash(cash);
```

算法交易成员函数

order_algo - 委托算法单

注：仅支持实时模式，部分券商版本可用

下算法单

函数原型：

```
AlgoOrder order_algo(const char *symbol, int volume, int position_effect, int side, int order_type, double price, AlgoParam &algo_param, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
position_effect	int	开平类型 参见 `enum PositionSide`
side	int	委托方向 参见 `enum OrderSide`
order_type	int	委托类型 参见 `enum OrderType`
price	double	委托价格
algo_param	struct	算法参数 参见 `struct AlgoParam`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	AlgoOrder	AlgoOrder, 如果函数调用失败， AlgoOrder.status 值为 `OrderStatus_Rejected`, Order.ord_rej_reason_detail 为错误原因描述, 其它情况表示函数调用成功，AlgoOrder.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//用算法 `TWAP` 委托买入10000股浦发银行

AlgoParam p{ "TWAP", "2019-1-18 9:30:00", "2019-1-18 11:30:00", 1, 100 };
AlgoOrder o = order_algo("SHSE.600000", 10000, PositionEffect_Open, OrderSide_Buy, OrderType_Market, 0, p);
```

algo_order_cancel - 撤单算法委托

注：仅支持实时模式，部分券商版本可用

撤销算法单

函数原型：

```
1. int algo_order_cancel(const char *cl_ord_id, const char *account = NULL);
```

参数：

参数名	类型	说明
cl_ord_id	const char *	委托单的客户id，可以在下单或查单时获得
account	const char *	实盘账号id，关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	int	成功返回0， 失败返回错误码

algo_order_pause - 暂停/恢复算法单

注：仅支持实时模式，部分券商版本可用

暂停/恢复算法单

函数原型：

```
1. int algo_order_pause(const char *cl_ord_id, int status, const char *account = NULL);
```

参数：

参数名	类型	说明
cl_ord_id	const char *	委托单的客户id，可以在下单或查单时获得
status	int	参考 `AlgoOrderStatus`
account	const char *	实盘账号id，关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	int	成功返回0， 失败返回错误码

get_algo_orders - 查询算法委托

注：仅支持实时模式，部分券商版本可用

查询所有算法委托单

函数原型：

```
1. DataArray<AlgoOrder>* get_algo_orders(const char *account = NULL);
```

参数：

参数名	类型	说明
account	const char *	账号ID `account_id`，如果输入为NULL，则返回所有账号的委托
返回值	`DataArray<AlgoOrder>*`	一个AlgoOrder结构数组

get_algo_child_orders - 查询算法子委托

注：仅支持实时模式，部分券商版本可用

查询子单

函数原型：

```
DataArray<Order>* get_algo_child_orders(const char *cl_ord_id, const char *account = NULL);
```

参数：

参数名	类型	说明
cl_ord_id	const char *	母单ID
account	const char *	账号ID `account_id`，如果输入为NULL，则返回所有账号的委托
返回值	`DataArray<Order>*`	一个AlgoOrder结构数组

新股业务成员函数

ipo_buy - 新股新债申购

注：仅在实盘中可以使用

函数原型：

```
1.     Order ipo_buy(const char *symbol, int volume, double price, const char *account =
   NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	申购价
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 OrderStatus_Rejected，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
1. //申购1000股的SHSE.688001
2. Order o = ipo_buy("SHSE.688001", 1000, 42.0);
```

ipo_get_quota - 查询客户新股新债申购额度

注：仅在实盘中可以使用

函数原型：

```
1.     DataArray<IPOQI>* ipo_get_quota(const char *account = NULL);
```

参数：

参数名	类型	说明
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL

返回值	`DataArray<IPOQI>*`	返回每个板块的申购额度

示例：

```
DataArray<IPOQI>*da = ipo_get_quota();
```

ipo_get_instruments - 查询当日新股新债清单

注：仅在实盘中可以使用

函数原型：

```
DataArray<IPOInstruments>* ipo_get_instruments(int security_type, const char *account = NULL);
```

参数：

参数名	类型	说明
security_type	int	指定要查询的品种，参见枚举 `SecurityType`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	`DataArray<IPOInstruments>*`	一个IPOInstruments结构数组

ipo_get_match_number - 配号查询

注：仅在实盘中可以使用

函数原型：

```
DataArray<IPOMatchNumber>* ipo_get_match_number(const char *account = NULL);
```

参数：

参数名	类型	说明
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts 获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	`DataArray<IPOMatchNumber>*`	一个IPOMatchNumber结构数组

ipo_get_lot_info - 中签查询

注：仅在实盘中可以使用

函数原型：

```
DataArray<IPOLotInfo>* ipo_get_lot_info(const char *account = NULL);
```

参数:

参数名	类型	说明
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	`DataArray<IPOLotInfo>*`	一个IPOLotInfo结构数组

基金业务成员函数

fund_etf_buy - ETF申购

注：仅在实盘中可以使用

函数原型：

```
1.     Order fund_etf_buy(const char *symbol, int volume, double price, const char
   *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	申购份额
price	double	申购价
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 OrderStatus_Rejected，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

fund_etf_redemption - ETF赎回

注：仅在实盘中可以使用

函数原型：

```
1.     Order fund_etf_redemption(const char *symbol, int volume, double price, const char
   *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	赎回份额
price	double	赎回价
account	const	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配

account	char *	置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 OrderStatus_Rejected ，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

fund_subscribing - 基金认购

注：仅在实盘中可以使用

函数原型：

```
1.    Order fund_subscribing(const char *symbol, int volume, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	认购份额
price	double	认购价
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 OrderStatus_Rejected ，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

fund_buy - 基金申购

注：仅在实盘中可以使用

函数原型：

```
1.    Order fund_buy(const char *symbol, int volume, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	申购份额
price	double	申购价
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 OrderStatus_Rejected ，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

注：仅在实盘中可以使用

函数原型：

```
Order fund_redemption(const char *symbol, int volume, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	赎回份额
price	double	赎回价
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected`，Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

债券业务成员函数

bond_reverse_repurchase_agreement - 国债逆回购

注：仅在实盘中可以使用

函数原型：

```
Order bond_reverse_repurchase_agreement(const char *symbol, int volume, double price, int order_type = OrderType_Limit, int order_duration = OrderDuration_Unknown, int order_qualifier = OrderQualifier_Unknown, const char *account = NULL);
```

参数：

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	委托价格
order_type	int	委托类型 参见 `enum OrderType`
order_duration	int	委托时间属性 参见 `enum OrderDuration`
order_qualifier	int	委托成交属性 参见 `enum OrderQualifier`
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

示例：

```
//下单1000张1天期的上海国债逆回购
Order o = bond_reverse_repurchase_agreement("SHSE.204001", 1000, 3.00);
```

bond_convertible_call - 可转债转股

注：仅在实盘中可以使用

函数原型：

```
Order bond_convertible_call(const char *symbol, int volume, double price, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	转股价（大部分柜台忽略，可填0）
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 OrderStatus_Rejected , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

bond_convertible_put - 可转债回售

注: 仅在实盘中可以使用

函数原型:

```
Order bond_convertible_put(const char *symbol, int volume, double price, const char *account = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码，只能单个标的
volume	int	委托数量
price	double	回售价（大部分柜台忽略，可填0）
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 OrderStatus_Rejected , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功，Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

bond_convertible_put_cancel - 可转债回售撤销

注: 仅在实盘中可以使用

函数原型:

```
Order bond_convertible_put_cancel(const char *symbol, int volume, const char *account = NULL);
```

参数:

参数名	类型	说明
	const	

symbol	char *	标的代码，只能单个标的
volume	int	委托数量
account	const char *	实盘账号id,关联多实盘账号时填写，可以从 get_accounts获取，也可以从终端实盘账号配置里拷贝。如果策略只关联一个账号，可以设置为NULL
返回值	Order	一个Order结构，如果函数调用失败， Order.status 值为 `OrderStatus_Rejected` , Order.ord_rej_reason_detail 为错误原因描述，其它情况表示函数调用成功， Order.cl_ord_id 为本次委托的标识，可用于追溯订单状态或撤单

动态参数成员函数

add_parameters - 添加参数

注：动态参数仅在仿真交易和实盘交易下生效

添加动态参数， 添加成功后， 参数将在终端上显示。

函数原型：

```
1. int add_parameters(Parameter *params, int count);
```

参数：

参数名	类型	说明
params	Parameter *	指向一个Parameter结构数据的指针
count	int	数组长度
返回值	int	成功返回0， 失败返回错误码

del_parameters - 删除参数

注：动态参数仅在仿真交易和实盘交易下生效

删除动态参数

函数原型：

```
1. int del_parameters(const char *keys);
```

参数：

参数名	类型	说明
keys	const char *	对应参数的键值
返回值	int	成功返回0， 失败返回错误码

set_parameters - 设置参数

注：动态参数仅在仿真交易和实盘交易下生效

设置参数值

函数原型：

```
int set_parameters(Parameter *params, int count);
```

参数：

参数名	类型	说明
params	Parameter *	指向一个Parameter结构数据的指针
count	int	数组长度
返回值	int	成功返回0， 失败返回错误码

get_parameters - 获取参数

注：动态参数仅在仿真交易和实盘交易下生效

获取参数值

函数原型：

```
DataArray<Parameter>* get_parameters();
```

参数：

参数名	类型	说明
返回值	DataArray*	一个Parameter数组

set_symbols - 设置标的

设置交易标的， 设置成功后， 标的将在终端上显示。

函数原型：

```
int set_symbols(const char *symbols);
```

参数：

参数名	类型	说明
symbols	const char *	symbol列表，逗号分隔
返回值	int	成功返回0， 失败返回错误码

get_symbols - 获取标的

获取交易标的

函数原型：

```
DataArray<Symbol>* get_symbols();
```

参数:

参数名	类型	说明
返回值	DataArray*	返加一个 Symbol数组

事件成员函数

on_init - 初始化完成

sdk初始化完成时触发，用户可以改写此成员函数，在些订阅行情，提取历史数据等初始化操作。

函数原型：

```
virtual void on_init();
```

on_tick - 收到Tick行情

收到Tick行情时触发

函数原型：

```
virtual void on_tick(Tick *tick);
```

参数：

参数名	类型	说明
tick	Tick *	收到的Tick行情

on_bar - 收到bar行情

收到bar行情时触发

函数原型：

```
1. virtual void on_bar(Bar *bar);
```

参数:

参数名	类型	说明
bar	Bar*	收到的Bar行情

on_l2transaction - 收到逐笔成交

收到逐笔成交（L2行情时有效）

函数原型:

```
1. virtual void on_l2transaction(L2Transaction *l2transaction);
```

参数:

参数名	类型	说明
l2transaction	L2Transaction*	收到的逐笔成交行情

on_l2order - 收到逐笔委托

收到逐笔委托（深交所L2行情时有效）

函数原型:

```
1. virtual void on_l2order(L2Order *l2order);
```

参数:

参数名	类型	说明
l2order	L2Order *	收到的逐笔委托行情

on_l2order_queue - 收到委托队列

收到委托队列, L2行情时有效, 最优价最大50笔委托量

函数原型:

```
1. virtual void on_l2order_queue(L2OrderQueue *l2queue);
```

参数:

参数名	类型	说明
l2queue	L2OrderQueue *	收到的委托队列行情

on_order_status - 委托变化

响应委托状态更新事情，下单后及委托状态更新时被触发。
注意：交易账户重连后，会重新推送一遍交易账户登录成功后查询回来的所有委托

函数原型：

```
virtual void on_order_status(Order *order);
```

参数：

参数名	类型	说明
order	Order*	发生变化的委托

注意：

1. 交易服务连接断开重连后，会自动重新推送一次所有委托(包含近期委托).
2. 交易账号错误断开到”已登陆”状态后，会自动重新推送一次所有委托(包含近期委托).
3. 交易服务连接断开重连事件通过on_trade_data_connected()回调通知.
4. 交易账号错误断开到”已登陆”事件通过on_account_status()回调通知.
5. 主动查询日内全部委托记录和未结委托的方式为get_orders()和get_unfinished_orders()函数.

on_execution_report - 执行回报

响应委托被执行事件，委托成交或者撤单拒绝后被触发。
注意：交易账户重连后，会重新推送一遍交易账户登录成功后查询回来的所有执行回报

函数原型：

```
virtual void on_execution_report(ExecRpt *rpt);
```

参数：

参数名	类型	说明
rpt	ExecRpt*	收到的回报

注意：

1. 交易服务连接断开重连后，会自动重新推送一次所有成交(包含近期成交).
2. 交易账号错误断开到”已登陆”状态后，会自动重新推送一次所有成交(包含近期成交).
3. 交易服务连接断开重连事件通过on_trade_data_connected()回调通知.
4. 交易账号错误断开到”已登陆”事件通过on_account_status()回调通知.
5. 主动查询日内全部执行回报的方式为get_execution_reports()函数.

on_parameter - 参数变化

参数变化时触发，一般是终端修了动态参数

函数原型：

```
virtual void on_parameter(Parameter *param);
```

参数:

参数名	类型	说明
param	Parameter*	变化的参数

on_schedule - 定时任务触发

预设任务时间条件符合时触发

函数原型:

```
virtual void on_schedule(const char *data_rule, const char *time_rule);
```

参数:

参数名	类型	说明
data_rule	const char *	设置的 data_rule
time_rule	const char *	设置的 time_rule

on_backtest_finished - 回测完成

回测完成时触发

函数原型:

```
virtual void on_backtest_finished();
```

on_indicator - 回测完成后收到绩效报告

回测完成后收到绩效报告时触发

函数原型:

```
virtual void on_indicator(Indicator *indicator);
```

参数:

参数名	类型	说明
data_rule	Indicator *	设置的 data_rule

on_account_status - 实盘账号状态变化

实盘账号状态变化时触发, 比如实盘账号登录, 退出登录等

函数原型：

```
virtual void on_account_status(AccountStatus *account_status);
```

参数：

参数名	类型	说明
account_status	AccountStatus *	对应变化的账号

on_error - 错误产生

有错误产生时触发，比如网络断开。

函数原型：

```
virtual void on_error(int error_code, const char *error_msg);
```

参数：

参数名	类型	说明
error_code	int	错误码
error_msg	const char *	错误信息

on_stop - 收到策略停止信号

终端点击停止策略时触发

函数原型：

```
virtual void on_stop();
```

on_market_data_connected - 数据服务已经连接上

数据服务已经连接时触发

函数原型：

```
virtual void on_market_data_connected();
```

on_trade_data_connected - 交易已经连接上

交易已经连接时触发

函数原型：

```
virtual void on_trade_data_connected();
```

on_market_data_disconnected - 数据连接断开了

数据连接断开时触发

函数原型:

```
1. virtual void on_market_data_disconnected();
```

on_trade_data_disconnected - 交易连接断开了

交易连接断开时触发

函数原型:

```
1. virtual void on_trade_data_disconnected();
```

数据查询函数

current - 查询当前行情快照

查询当前行情快照，返回tick数据。回测时，返回回测时间点的tick数据

函数原型：

```
DataArray<Tick>* current(const char *symbols);
```

参数：

参数名	类型	说明
symbols	const char *	查询代码，如有多个代码，中间用 `,` (英文逗号) 隔开
返回值	DataArray*	Tick数组

示例：

```
DataArray<Tick>* current_data = current("SZSE.000001,SZSE.000002");
```

注意：

1.start_time和end_time中月,日,时,分,秒均可以只输入个位数,例: `"2010-7-8 9:40:0"` 或 `"2017-7-30 12:3:0"` ,但若对应位置为零，则 `0` 不可被省略,如不可输入 `"2017-7-30 12:3: "`

history_ticks - 查询历史Tick行情

函数原型：

```
DataArray<Tick>* history_ticks(const char *symbols, const char *start_time, const char *end_time, int adjust = 0, const char *adjust_end_time = NULL, bool skip_suspended = true, const char *fill_missing = NULL);
```

参数：

参数名	类型	说明
symbols	const char *	标的代码,若要获取多个标的的历史数据，可以采用中间用 `,` （英文逗号）隔开的方法
start_time	const char *	开始时间 (%Y-%m-%d %H:%M:%S 格式),其中日线包含start_time数据,非日线不包含start_time数据
end_time	const char *	结束时间 (%Y-%m-%d %H:%M:%S 格式),
adjust	int	`ADJUST_NONE or 0: 不复权` , `ADJUST_PREV or 1: 前复权` , `ADJUST_POST or 2: 后复权` 默认不复权
adjust_end_time	const char *	复权基点时间, 默认当前时间
skip_suspended	bool	是否跳过停牌, 默认跳过
fill_missing	const char *	填充方式, `None` - 不填充, `'NaN'` - 用空值填充, `'Last'` - 用上一个值填充, 默认None
返回值	DataArray*	一个Tick数组

示例：

```
DataArray<Tick>* history_tick = history_ticks(symbol="SHSE.000300", start_time="2010-07-28",  end_time="2017-07-30")
```

注意：

1.start_time和end_time中月,日,时,分,秒均可以只输入个位数,例: `"2010-7-8 9:40:0"` 或 `"2017-7-30 12:3:0"` ,但若对应位置为零，则 `0` 不可被省略,如不可输入 `"2017-7-30 12:3: "`

2. skip_suspended 和 fill_missing 参数暂不支持

3. 单次返回数据量最大返回33000, 超出部分不返回

history_bars - 查询历史Bar行情

函数原型：

```
DataArray<Bar>* history_bars(const char *symbols, const char *frequency, const char *start_time, const char *end_time, int adjust = 0, const char *adjust_end_time = NULL, bool skip_suspended = true, const char *fill_missing = NULL);
```

参数：

参数名	类型	说明

symbols	const char *	标的代码,若要获取多个标的的历史数据，可以采用中间用 `,` （英文逗号）隔开的方法
frequency	const char *	频率，支持 ‘1d’, ‘60s’, ‘300s’ , ‘1800s’ , ‘3600s’ 等
start_time	const char *	开始时间 (%Y-%m-%d %H:%M:%S 格式),其中日线包含start_time数据,非日线不包含start_time数据
end_time	const char *	结束时间 (%Y-%m-%d %H:%M:%S 格式),
adjust	int	`ADJUST_NONE or 0: 不复权` , `ADJUST_PREV or 1: 前复权` , `ADJUST_POST or 2: 后复权` 默认不复权
adjust_end_time	const char *	复权基点时间, 默认当前时间
skip_suspended	bool	是否跳过停牌, 默认跳过
fill_missing	const char *	填充方式, `None` - 不填充, `'NaN'` - 用空值填充, `'Last'` - 用上一个值填充, 默认None
返回值	DataArray*	一个Bar数组

示例:

```
//获取1分钟的bar
DataArray<Bar>* history_bar = history_bars(symbol="SHSE.000300", "60s", start_time="2010-07-28",  end_time="2017-07-30");

//获取60分钟的bar
DataArray<Bar>* history_bar = history_bars(symbol="SHSE.000300", "3600s", start_time="2010-07-28",  end_time="2017-07-30");

//获取日线
DataArray<Bar>* history_bar = history_bars(symbol="SHSE.000300", "1d", start_time="2010-07-28",  end_time="2018-07-30");
```

注意:

1.start_time和end_time中月,日,时,分,秒均可以只输入个位数,例: `"2010-7-8 9:40:0"` 或 `"2017-7-30 12:3:0"` ,但若对应位置为零，则 `0` 不可被省略,如不可输入 `"2017-7-30 12:3: "`

2. skip_suspended 和 fill_missing 参数暂不支持

3. 单次返回数据量最大返回33000, 超出部分不返回

history_ticks_n - 查询最新n条Tick行情

函数原型:

```
DataArray<Tick>* history_ticks_n(const char *symbol, int n, const char *end_time = NULL, int adjust = 0, const char *adjust_end_time = NULL, bool skip_suspended = true, const char *fill_missing = NULL);
```

参数:

参数名	类型	说明

symbol	const char *	标的代码,只支持单个标的
n	int	获取行情的数量
end_time	const char *	从此时间开始，往前取行情， 如果为NULL， 那么为当前时间开始
adjust	int	`ADJUST_NONE or 0: 不复权` , `ADJUST_PREV or 1: 前复权` , `ADJUST_POST or 2: 后复权` 默认不复权
adjust_end_time	const char *	复权基点时间， 默认当前时间
skip_suspended	bool	是否跳过停牌， 默认跳过
fill_missing	const char *	填充方式， `None` - 不填充， `'NaN'` - 用空值填充， `'Last'` - 用上一个值填充， 默认None
返回值	DataArray*	一个Tick数组

示例:

```
//获取沪深300最新10条tick
DataArray<Tick>* history_tick_n = history_ticks_n(symbol="SHSE.000300", 10);
```

注意:

1.end_time中月,日,时,分,秒均可以只输入个位数,例: `"2010-7-8 9:40:0"` 或 `"2017-7-30 12:3:0"` ,但若对应位置为零，则 `0` 不可被省略,如不可输入 `"2017-7-30 12:3:"`

2. skip_suspended 和 fill_missing 参数暂不支持

3. 单次返回数据量最大返回33000， 超出部分不返回

history_bars_n - 查询最新n条Bar行情

函数原型:

```
DataArray<Bar>* history_bars_n(const char *symbol, const char *frequency, int n, const char *end_time = NULL, int adjust = 0, const char *adjust_end_time = NULL, bool skip_suspended = true, const char *fill_missing = NULL);
```

参数:

参数名	类型	说明
symbol	const char *	标的代码,只支持单个标的
frequency	const char *	频率， 支持 ‘1d’, ‘60s’, ‘300s’ , ‘1800s’ , ‘3600s’ 等
n	int	获取行情的数量
end_time	const char *	从此时间开始，往前取行情， 如果为NULL， 那么为当前时间开始
adjust	int	`ADJUST_NONE or 0: 不复权` , `ADJUST_PREV or 1: 前复权` , `ADJUST_POST or 2: 后复权` 默认不复权

adjust_end_time	const char *	复权基点时间, 默认当前时间
skip_suspended	bool	是否跳过停牌, 默认跳过
fill_missing	const char *	填充方式, `None` - 不填充, `'NaN'` - 用空值填充, `'Last'` - 用上一个值填充, 默认None
返回值	DataArray*	一个Bar数组

示例:

```
//获取沪深300最新10条1分钟bar
DataArray<Bar>* history_bars_n = history_bars_n(symbol="SHSE.000300", "60s", 10);
```

注意:

1.end_time中月,日,时,分,秒均可以只输入个位数,例: `"2010-7-8 9:40:0"` 或 `"2017-7-30 12:3:0"` ,但若对应位置为零, 则”0”不可被省略,如不可输入 `"2017-7-30 12:3: "`

2. skip_suspended 和 fill_missing 参数暂不支持

3. 单次返回数据量最大返回33000, 超出部分不返回

history_l2ticks - 查询历史L2 Tick行情

函数原型:

```
DataArray<Tick>* history_l2ticks(const char *symbols, const char *start_time, const char *end_time, int adjust = 0, const char *adjust_end_time = NULL, bool skip_suspended = true, const char *fill_missing = NULL);
```

参数:

参数名	类型	说明
symbols	const char *	标的代码,若要获取多个标的的历史数据, 可以采用中间用 `,` (英文逗号) 隔开的方法
start_time	const char *	开始时间 (%Y-%m-%d %H:%M:%S 格式),其中日线包含start_time数据, 非日线不包含start_time数据
end_time	const char *	结束时间 (%Y-%m-%d %H:%M:%S 格式),
adjust	int	`ADJUST_NONE or 0: 不复权`, `ADJUST_PREV or 1: 前复权`, `ADJUST_POST or 2: 后复权` 默认不复权
adjust_end_time	const char *	复权基点时间, 默认当前时间
skip_suspended	bool	是否跳过停牌, 默认跳过
fill_missing	const char *	填充方式, `None` - 不填充, `'NaN'` - 用空值填充, `'Last'` - 用上一个值填充, 默认None
返回值	DataArray*	一个Tick数组

示例:

```
DataArray<Tick>* history_tick = history_l2ticks(symbol="SHSE.600000", start_time="2020-03-28",  end_time="2020-03-29")
```

history_l2bars - 查询历史L2 Bar行情

函数原型：

```
DataArray<Bar>* history_l2bars(const char *symbols, const char *frequency, const char *start_time, const char *end_time, int adjust = 0, const char *adjust_end_time = NULL, bool skip_suspended = true, const char *fill_missing = NULL);
```

参数：

参数名	类型	说明
symbols	const char *	标的代码,若要获取多个标的的历史数据，可以采用中间用 `,` （英文逗号）隔开的方法
frequency	const char *	频率，支持 ‘1d’, ‘60s’, ‘300s’ , ‘1800s’ , ‘3600s’ 等
start_time	const char *	开始时间 (%Y-%m-%d %H:%M:%S 格式),其中日线包含start_time数据,非日线不包含start_time数据
end_time	const char *	结束时间 (%Y-%m-%d %H:%M:%S 格式),
adjust	int	`ADJUST_NONE or 0: 不复权` , `ADJUST_PREV or 1: 前复权` , `ADJUST_POST or 2: 后复权` 默认不复权
adjust_end_time	const char *	复权基点时间, 默认当前时间
skip_suspended	bool	是否跳过停牌, 默认跳过
fill_missing	const char *	填充方式, `None` - 不填充, `'NaN'` - 用空值填充, `'Last'` - 用上一个值填充, 默认None
返回值	DataArray*	一个Bar数组

示例：

```
//获取1分钟的bar
DataArray<Bar>* history_bar = history_l2bars(symbol="SHSE.000300", "60s", start_time="2010-07-28",  end_time="2017-07-30");

//获取60分钟的bar
DataArray<Bar>* history_bar = history_l2bars(symbol="SHSE.000300", "3600s", start_time="2010-07-28",  end_time="2017-07-30");
```

history_l2transactions - 查询历史L2 逐笔成交

函数原型：

```
DataArray<L2Transaction>* history_l2transactions(const char *symbols, const char *start_time, const char *end_time);
```

参数:

参数名	类型	说明
symbols	const char *	标的代码,若要获取多个标的的历史数据，可以采用中间用 `,` (英文逗号) 隔开的方法
start_time	const char *	开始时间 (%Y-%m-%d %H:%M:%S 格式),其中日线包含start_time数据, 非日线不包含start_time数据
end_time	const char *	结束时间 (%Y-%m-%d %H:%M:%S 格式),
返回值	DataArray*	一个L2Transaction数组

示例:

```
DataArray<L2Transaction>* data = history_l2transactions(symbol="SHSE.600000", start_time="2020-03-28",  end_time="2020-03-29");
```

history_l2orders -查询历史L2 逐笔委托

函数原型:

```
DataArray<L2Order>* history_l2orders(const char *symbols, const char *start_time, const char *end_time);
```

参数:

参数名	类型	说明
symbols	const char *	标的代码,若要获取多个标的的历史数据，可以采用中间用 `,` (英文逗号) 隔开的方法
start_time	const char *	开始时间 (%Y-%m-%d %H:%M:%S 格式),其中日线包含start_time数据, 非日线不包含start_time数据
end_time	const char *	结束时间 (%Y-%m-%d %H:%M:%S 格式),
返回值	DataArray*	一个L2Order数组

示例:

```
DataArray<L2Order>* data = history_l2orders(symbol="SHSE.600000", start_time="2020-03-28",  end_time="2020-03-29");
```

history_l2orders_queue -查询历史L2 委托队列

函数原型:

```
DataArray<L2OrderQueue>* history_l2orders_queue(const char *symbols, const char *start_time, const char *end_time);
```

参数:

参数名	类型	说明
symbols	const char *	标的代码,若要获取多个标的的历史数据，可以采用中间用 `,` (英文逗号) 隔开的方法
start_time	const char *	开始时间 (%Y-%m-%d %H:%M:%S 格式),其中日线包含start_time数据, 非日线不包含start_time数据
end_time	const char *	结束时间 (%Y-%m-%d %H:%M:%S 格式),
返回值	DataArray*	一个L2OrderQueue数组

示例:

```
DataArray<L2OrderQueue>* data = history_l2orders_queue(symbol="SHSE.600000",
start_time="2020-03-28",  end_time="2020-03-29");
```

get_fundamentals - 查询基本面数据

函数原型:

```
DataSet* get_fundamentals(const char *table, const char *symbols, const char
*start_date, const char *end_date, const char *fields, const char *filter = NULL, const
char *order_by = NULL, int limit = 1000);
```

参数:

参数名	类型	说明
table	const char*	表名，只支持单表查询. 具体表名及fields字段名及filter可过滤的字段参考 财务数据文档
symbols	const char*	标的代码, 多个代码可用 `,` (英文逗号)分割
start_date	const char*	开始时间, (%Y-%m-%d 格式)
end_date	const char*	结束时间, (%Y-%m-%d 格式)
fields	const char*	查询字段 (必填)
filter	const char*	查询过滤,使用方法参考例3、例4
order_by	const char*	排序方式, 默认 `None`. `TCLOSE` 表示按 `TCLOSE` 升序排序. `-TCLOSE` 表示按 `TCLOSE` 降序排序. `TCLOSE, -NEGOTIABLEMV` 表示按 `TCLOSE` 升序, `NEGOTIABLEMV` 降序综合排序
limit	int	数量. 为保护服务器, 一次查询最多返回 `40000` 条记录
返回值	DataSet*	一个结果集

示例:

例1: 取股票代码 `SHSE.600000, SZSE.000001` , 离 `2017-01-01` 最近一个交易日的 股票交易财务衍生表 的 `TCLOSE,NEGOTIABLEMV,TOTMKTCAP,TURNRATE,PELFY,PETTM,PEMRQ,PELFYNPAAEI,PETTMNPAAEI` 字段的值

```
DataSet* ds = get_fundamentals("trading_derivative_indicator",
"SHSE.600000,SZSE.000001", "2017-01-01", "2017-01-01",
"TCLOSE,NEGOTIABLEMV,TOTMKTCAP,TURNRATE,PELFY,PETTM,PEMRQ,PELFYNPAAEI,PETTMNPAAEI")
```

例2: 取股票代码 `SHSE.600000, SZSE.000001` , 指定时间段 `2016-01-01 -- 2017-01-01` 股票交易财务衍生表 的 `TCLOSE,NEGOTIABLEMV,TOTMKTCAP,TURNRATE,PELFY,PETTM,PEMRQ,PELFYNPAAEI,PETTMNPAAEI` 字段的值

```
DataSet* ds = get_fundamentals("trading_derivative_indicator",
"SHSE.600000,SZSE.000001", "2016-01-01", "2017-01-01",

"TCLOSE,NEGOTIABLEMV,TOTMKTCAP,TURNRATE,PELFY,PETTM,PEMRQ,PELFYNPAAEI,PETTMNPAAEI")
```

例3: 取指定股票 `SHSE.600000, SHSE.600001, SHSE.600002` 离 `2017-01-01` 最近一个交易日, 且满足条件 `PCTTM > 0 and PCTTM < 10` 股票交易财务衍生表 的 `TCLOSE,NEGOTIABLEMV,TOTMKTCAP,TURNRATE,PELFY,PETTM,PEMRQ,PELFYNPAAEI,PETTMNPAAEI` 字段的值

```
DataSet* ds = get_fundamentals("trading_derivative_indicator",
"SHSE.600000,SHSE.600001,SHSE.600002", "2017-01-01", "2017-01-01", "PCTTM &gt; 0 and
PCTTM &lt; 10",

"TCLOSE,NEGOTIABLEMV,TOTMKTCAP,TURNRATE,PELFY,PETTM,PEMRQ,PELFYNPAAEI,PETTMNPAAEI")
```

例4: 取指定股票 `SHSE.600000,SZSE.000001` 离 `2016-01-20` 最近一个财报, 同时满足条件 `CURFDS > 0 and TOTLIABSHAREQUI > 0` 的 资产负债 的数据

```
DataSet* ds = get_fundamentals("balance_sheet", "2016-01-20", "2016-01-20",
                "CURFDS, SETTRESEDEPO, PLAC, TRADFINASSET, ',
                symbols="SHSE.600000, SZSE.000001",
                "CURFDS > 0 and TOTLIABSHAREQUI > 0"
                )
```

注意:

1.当 `start_date` == `end_date` 时, 取所举每个股票离 `end_date` 最近业务日期(交易日期或报告日期)一条数据,当 `start_date` < `end_date` 时, 取指定时间段的数据,当 `start_date` > `end_date` 时, 返回 `空`

2.start_date和end_date中月,日均可以只输入个位数,例: `"2010-7-8"` 或 `"2017-7-30"`

3.在该函数中, table参数只支持输入一个表名, 若表名输入错误或以 `"table1,table2"` 方式输入多个表名, 函数返回空结果集

4.若表名输入正确, 但查询字段中出现非指定字符串, 则返回错误

get_fundamentals_n - 查询基本面数据最新n条

取指定股票的最近 `end_date` 的 `n` 条记录

函数原型:

```
DataSet* get_fundamentals_n(const char *table, const char *symbols, const char *end_date, const char *fields, int n = 1, const char *filter = NULL, const char *order_by = NULL);
```

参数:

参数名	类型	说明
table	const char *	表名. 具体表名及fields字段名及filter可过滤的字段参考 财务数据文档
symbols	const char *	标的代码, 多个代码可用 `,` (英文逗号)分割
end_date	const char *	结束时间, (%Y-%m-%d 格式)
fields	const char *	查询字段 (必填)
filter	const char *	查询过滤,,使用方法参考 `get_fundamentals` 的例3、例4
n	int	每个股票取最近的数量(正整数)
返回值	DataSet*	一个结果集

示例:

例: 取股票代码 `SHSE.600000, SZSE.000001` , 离 `2017-01-01` 最近3条(每个股票都有3条) 股票交易财务衍生表 的 `TCLOSE,NEGOTIABLEMV,TOTMKTCAP,TURNRATE,PELFY,PETTM,PEMRQ,PELFYNPAAEI,PETTMNPAAEI` 字段的值

```
DataSet* ds = get_fundamentals_n("trading_derivative_indicator", "SHSE.600000, SZSE.000001", "2017-01-01", 3,
"TCLOSE,NEGOTIABLEMV,TOTMKTCAP,TURNRATE,PELFY,PETTM,PEMRQ,PELFYNPAAEI,PETTMNPAAEI")
```

注意:

1.end_date中月,日均可以只输入个位数,例: `'2010-7-8'` 或 `'2017-7-30'`

2.在该函数中, table参数只支持输入一个表名, 若表名输入错误或以 `'table1,table2'` 方式输入多个表名, 函数返回空结果集

3.若表名输入正确, 但查询字段中出现非指定字符串, 则返回错误

get_instruments - 查询最新交易标的信息

查询最新交易标的信息,有基本数据及最新日频数据

函数原型:

```
DataSet* get_instruments(const char *exchanges = NULL, const char *sec_types = NULL, const char* fields = NULL);
```

参数:

参数名	类型	说明
exchanges	const	交易所代码, 多个交易所代码可用 `,` (英文逗号)分割, NULL表示所有

exchanges	char *	交易所代码，多个交易所代码可用 `,` (英文逗号)分割，NULL表示所有
sec_types	const char *	指定类别，品种类型，stock：股票，fund：基金，index：指数，future：期货，bond：债券，bond_convertible：可转债，option：期权，confuture：虚拟合约，多个品种可用 `,` (英文逗号)分割，NULL表示所有品种
fields	const char *	查询字段 默认NULL 表示所有
返回值	DataSet*	一个结果集

字段：

字段名	类型	说明
symbol	string	标的代码
sec_level	int	1-正常,2-ST 股票,3-*ST 股票,4-股份转让,5-处于退市整理期的证券,6-上市开放基金LOF,7-交易型开放式指数基金(ETF),8-非交易型开放式基金(暂不交易,仅揭示基金净值及开放申购赎回业务),9-仅提供净值揭示服务的开放式基金;,10-仅在协议交易平台挂牌交易的证券,11-仅在固定收益平台挂牌交易的证券,12-风险警示产品,13-退市整理产品,99-其它
is_suspended	int	是否停牌. 1: 是, 0: 否
multiplier	double	合约乘数
margin_ratio	double	保证金比率
settle_price	double	结算价
position	int	持仓量
pre_close	double	昨收价
pre_settle	double	昨结算价
upper_limit	double	涨停价
lower_limit	double	跌停价
adj_factor	double	复权因子.基金跟股票才有
created_at	longlong(utc)	交易日期

示例：

```
//取深交所所有代码
DataSet* ds = get_instruments("SZSE");

//取深交所所有股票和基金
DataSet* ds = get_instruments("SZSE", "stock,fund");
```

注意：

1.停牌时且股票发生除权除息，涨停价和跌停价可能有误差

2.预上市股票以1900-01-01为虚拟发布日期，未退市股票以2038-01-01为虚拟退市日期。

get_history_instruments - 查询交易标的历史数据

返回指定symbols的标的日频历史数据

```
DataSet* get_history_instruments(const char *symbols, const char *start_date, const char *end_date, const char *fields = NULL);
```

参数:

参数名	类型	说明
symbols	const char *	标的代码, 多个代码可用 `,`(英文逗号)分割
start_date	const char *	开始时间. (%Y-%m-%d 格式)
end_date	const char *	结束时间. (%Y-%m-%d 格式)
fields	const char *	查询字段. NULL表示所有字段
返回值	DataSet*	一个结果集

字段:

字段名	类型	说明
symbol	string	标的代码
sec_level	int	1-正常,2-ST 股票,3-*ST 股票,4-股份转让,5-处于退市整理期的证券,6-上市开放基金LOF,7-交易型开放式指数基金(ETF),8-非交易型开放式基金(暂不交易,仅揭示基金净值及开放申购赎回业务),9-仅提供净值揭示服务的开放式基金;,10-仅在协议交易平台挂牌交易的证券,11-仅在固定收益平台挂牌交易的证券,12-风险警示产品,13-退市整理产品,99-其它
is_suspended	int	是否停牌. 1: 是, 0: 否
multiplier	double	合约乘数
margin_ratio	double	保证金比率
settle_price	double	结算价
position	int	持仓量
pre_close	double	昨收价
pre_settle	double	昨结算价
upper_limit	double	涨停价
lower_limit	double	跌停价
adj_factor	double	复权因子.基金跟股票才有
created_at	longlong(utc)	交易日期

示例:

```
get_history_instruments(symbols="SZSE.000001,SZSE.000002", start_date="2017-09-19", end_date="2017-09-19");
```

注意:

1.停牌时且股票发生除权除息, 涨停价和跌停价可能有误差

2.为保护服务器, 单次查询最多返回 **3300** 条记录

3.start_date和end_date中月,日均可以只输入个位数,例: `'2010-7-8'` 或 `'2017-7-30'`

3.start_date和end_date中月,日均可以只输入个位数,例: '2010-7-8' 或 '2017-7-30'

get_instrumentinfos - 查询交易标的基本信息

获取到交易标的基本信息, 与时间无关.

函数原型:

```
DataSet* get_instrumentinfos(const char *symbols = NULL, const char *exchanges = NULL,
const char * sec_types = NULL, const char *names = NULL, const char *fields = NULL);
```

参数:

参数名	类型	说明
symbols	const char *	标的代码, 多个代码可用 , (英文逗号)分割, NULL 表示所有
exchanges	const char *	交易市场代码, 多个交易所代码可用 , (英文逗号)分割, NULL 表示所有市场
sec_types	const char *	指定类别, 品种类型, stock: 股票, fund: 基金, index: 指数, future: 期货, option: 期权, confuture: 虚拟合约, 多个品种可用 , (英文逗号)分割, NULL表示所有品种
names	const char *	查询代码, NULL 表示所有
fields	const char *	查询字段 NULL 表示所有
返回值	DataSet*	一个结果集

字段:

字段名	类型	说明
symbol	string	标的代码
sec_type	int	1: 股票, 2: 基金, 3: 指数, 4: 期货, 5: 期权, 10: 虚拟合约
exchange	string	见交易所代码
sec_id	string	代码
sec_name	string	名称
price_tick	double	最小变动单位
listed_date	longlong(utc)	上市日期
delisted_date	longlong(utc)	退市日期

示例:

```
DataSet* ds = get_instrumentinfos("SHSE.000001,SHSE.000002");
```

注意:

1.对于检索所需标的信息的4种手段 symbols,exchanges,sec_types,names ,若输入参数之间出现任何矛盾(换句话说, 所有的参数限制出满足要求的交集为空),则返回空结果集

get_constituents - 查询指数成份股

函数原型:

```
1. DataSet* get_constituents(const char *index, const char *trade_date = NULL);
```

参数:

参数名	类型	说明
index	const char *	表示指数的symbol,比如上证指数SHSE.000001, 不支持多个 symbol
trade_date	const char *	交易日 (%Y-%m-%d 格式) NULL 表示最新日期
返回值	DataSet*	一个结果集

字段:

字段名	类型	说明
symbol	string	标的代码
weight	double	权重

示例:

```
1. DataSet* ds = get_constituents("SHSE.000001", "2017-07-10");
```

注意:

1.trade_date 中月,日均可以只输入个位数,例: `"2010-7-8"` 或 `"2017-7-30"` ,但若对应位置为零, 则 `0` 不可被省略

get_industry - 查询行业股票列表

函数原型:

```
1. DataArray<Symbol>* get_industry(const char *code);
```

参数:

参数名	类型	说明
code	const char *	行业代码
返回值	DataArray*	一个symbol数组

示例:

```
1. #返回所有以J6开头的行业代码对应的成分股(包括:J66,J67,J68,J69的成分股)
2. DataArray<Symbol>* da = get_industry("j6");
```

注意:

注意：

1.在该函数中，code参数只支持输入一个行业代码

get_trading_dates - 查询交易日历

函数原型：

```
DataArray<TradingDate>* get_trading_dates(const char *exchange, const char *start_date,
const char *end_date);
```

参数：

参数名	类型	说明
exchange	const char *	交易市场代码
start_date	const char *	开始时间 (%Y-%m-%d 格式)
end_date	const char *	结束时间 (%Y-%m-%d 格式)
返回值	DataArray*	一个交易日数组

示例：

```
DataArray<TradingDate>* da = get_trading_dates("SZSE", "2017-01-01", "2017-01-30");
```

注意：

1. `exchange` 参数仅支持输入单个交易所代码

2. `start_date` 和 `end_date` 中月,日均可以只输入个位数,
例: `'2010-7-8'` 或 `'2017-7-30'`

get_previous_trading_date - 返回指定日期的上一个交易日

函数原型：

```
int get_previous_trading_date(const char *exchange, const char *date, char
*output_date);
```

参数：

参数名	类型	说明
exchange	const char *	交易市场代码
date	const char *	时间 (%Y-%m-%d 格式)
output_date	char *	出参，返回值，上一个交易日，字符串格式， 由用户分配buf， 大于32字节即可
返回值	int	0表示成功，非0表示失败

示例：

```
char trading_date[32] = {0};
get_previous_trading_date("SZSE", "2017-05-01", trading_date);
```

注意:

1. `exchange` 参数仅支持输入单个交易所代码

2. `date` 中月,日均可以只输入个位数,
例: `"2010-7-8"` 或 `"2017-7-30"`

get_next_trading_date - 返回指定日期的下一个交易日

函数原型:

```
int get_next_trading_date(const char *exchange, const char *date, char *output_date);
```

参数:

参数名	类型	说明
exchange	const char *	交易市场代码
date	const char *	时间 (%Y-%m-%d 格式)
output_date	char *	出参，返回值，下一个交易日，字符串格式， 由用户分配buf， 大于32字节即可
返回值	int	0表示成功，非0表示失败

示例:

```
char trading_date[32] = {0};
get_next_trading_date(exchange="SZSE", date="2017-05-01", trading_date);
```

注意:

1. `exchange` 参数仅支持输入单个交易所代码

2. `date` 中月,日均可以只输入个位数,
例: `"2010-7-8"` 或 `"2017-7-30"`

get_dividend - 查询分红送配

函数原型:

```
DataSet* get_dividend(const char *symbol, const char *start_date, const char
*end_date);
```

参数:

参数名	类型	说明

symbol	const char *	标的代码, (必填)
start_date	const char *	开始时间 (%Y-%m-%d 格式)
end_date	const char *	结束时间 (%Y-%m-%d 格式)
返回值	DataSet*	一个结果集

字段:

字段名	类型	说明
symbol	string	标的代码
cash_div	double	每股派现
allotment_ratio	double	每股配股比例
allotment_price	double	配股价
share_div_ratio	double	每股送股比例
share_trans_ratio	double	每股转增比例

示例:

```
DataSet* ds = get_dividend("SHSE.600000", "2000-01-01", "2017-12-31");
```

注意:

1.在该函数中, symbol参数只支持输入一个标的代码

2. `start_date` 和 `end_date` 中月,日均可以只输入个位数,
例: `'2010-7-8'` 或 `'2017-7-30'`

get_continuous_contracts - 获取连续合约

函数原型:

```
DataSet* get_continuous_contracts(const char *csymbol, const char *start_date, const char *end_date);
```

参数:

参数名	类型	说明
csymbol	const char *	虚拟合约代码, 比如获取主力合约, 输入SHFE.AG;获取连续合约, 输入SHFE.AG01
start_date	const char *	开始时间 (%Y-%m-%d 格式)
end_date	const char *	结束时间 (%Y-%m-%d 格式)
返回值	DataSet*	一个结果集

字段:

字段名	类型	说明

symbol	string	真实合约
created_at	longlong(utc)	交易日

示例:

```
1. DataSet* get_continuous_contracts"SHFE.AG", "2017-07-01", "2017-08-01");
```

注意:

1.在该函数中，csymbol参数只支持输入一个标的代码

2. `start_date` 和 `end_date` 中月,日均可以只输入个位数,
例: `'2017-7-1'` 或 `'2017-8-1'`

类定义

DataSet 结果集

DataSet类是基本面数据查询结果的标准返回， 表示一个二维表数据存储。类声明如下：

```
class DataSet
{
public:
    //获取api调用结果, 0: 成功,  非0: 错误码
    virtual int status() = 0;

    //判断是否已经是到达结果集末尾
    virtual bool is_end() = 0;

    //移到下一条记录
    virtual void next() = 0;

    //获取整型值
    virtual int get_integer(const char *key) = 0;

    //获取长整型值
    virtual long long get_long_integer(const char *key) = 0;

    //获取浮点型值
    virtual double get_real(const char *key) = 0;

    //获取字符串型值
    virtual const char* get_string(const char *key) = 0;

    //释放数据集合
    virtual void release() = 0;

    //打印数据
    virtual const char* debug_string() = 0;
};
```

典型的使用场景如下：

1. 调用数据查询函数返回一个DataSet对象指针 DataSet *ds;
2. 调用 `ds->status()` 判断函数调用是否成功，0表示成功，非0表示错误码，调用失败，结果集为空。
3. 如果 `ds->status()` 返回成功，调用 `ds->is_end()` 与 `ds->next()` 遍历结果集取值。
4. 调用 `ds->release()` 释放结果集。

`debug_string` 用于返回整个结果集内容， 包含字段和值， 一般用于开发调试， 快速知晓结果集的表结构。

使用举例

```
//获取深交所最新的代码信息
DataSet* jy = get_instruments("SZSE");

if (jy->status() == 0)
{
    //调用get_instruments成功,  以下遍历结果集

    while (!jy->is_end())  //先要判断是否已经到达结果集末尾
    {

        cout << jy->get_string("symbol") << endl;        //取字符串值
        cout << jy->get_integer("sec_level") << endl;    //取整型值
        cout << jy->get_real("pre_close") << endl;       //取浮点值

        jy->next(); //移动到下一条记录
    }
}
else
{
    // 调用get_instruments 失败,   jy->status() 为错误码
    cout << "get_instruments error: " << jy->status() << endl;
}

// 使用完结果集要释放
jy->release();
```

成员函数

status 获取函数调用结果

获得结果集之后， 第一步都应该先调用本成员函数判断查询数据是否成功。

函数原型：

```
1. int status()
```

参数：

参数名	类型	说明
返回值	int	0：成功， 非0：错误码

注意：

1. 如果status 断定为失败时， 不应该再遍历结果集， 这时直接释放结果集即可。

is_end 判断是否到达结果集末尾

获得结果集之后， 默认指向结果集的第一条记录。如果 is_end返回true， 则表示没有指向有效的记录了， 遍历应该就此结束。

函数原型：

```
1. bool is_end()
```

参数：

参数名	类型	说明
返回值	bool	true：已到结果集末尾，当前记录无效， false：未到结果集末尾，当前记录有效

next 移到下一条记录

移到下一条记录， 先用 is_end 判断记录是否有效，再取值。

函数原型：

```
void next()
```

get_integer 获取整型值

如果当前记录有效， 则可取值

函数原型：

```
int get_integer(const char *key)
```

参数：

参数名	类型	说明
key	const char *	字段名
返回值	int	所取的字段值

get_long_integer 获取长整型值

如果当前记录有效， 则可取值

函数原型：

```
long long get_long_integer(const char *key)
```

参数：

参数名	类型	说明
key	const char *	字段名
返回值	long long	所取的字段值

get_real 获取浮点型值

如果当前记录有效， 则可取值

函数原型：

```
double get_real(const char *key)
```

参数：

参数名	类型	说明
key	const char *	字段名
返回值	double	所取的字段值

get_string 获取字符串值

如果当前记录有效， 则可取值

函数原型：

```
const char* get_string(const char *key);
```

参数：

参数名	类型	说明
key	const char *	字段名
返回值	const char*	所取的字段值

release 释放数据集合

获取DataSet指针之后，最后都应该释放数据集合(不管status是什么状态)， 不然会造成内存泄露。调用release之后，不能再调用结果集任何成员函数。

函数原型：

```
void release()
```

debug_string 返回整个结果集信息

使用调试，快速知道结果集的表结构

函数原型：

```
const char* debug_string();
```

参数：

参数名	类型	说明
返回值	const char*	整个结果集信息

类定义

DataArray 数组

DataArray类模块是行情与交易数据查询的标准返回， 表示一个结构体数组。类声明如下：

```
template <typename T>
class DataArray
{
public:
    //获取api调用结果, 0: 成功,  非0: 错误码
    virtual int status() = 0;

    //返回结构数组的指针
    virtual T* data() = 0;

    //返回数据的长度
    virtual int count() = 0;

    //返回下标为i的结构引用, 从0开始
    virtual T& at(int i) = 0;

    //释放数据集合
    virtual void release() = 0;
};
```

典型的使用场景如下：

1. 调用数据查询函数返回一个DataArray对象指针 DataArray *da;
2. 调用 `da->status()` 判断函数调用是否成功，0表示成功，非0表示错误码，调用失败，数组长度为0
3. 如果 `da->status()` 返回成功，则可以遍历数组。
4. 调用 `da->release()` 释放结果集。

使用举例

```

//查询一段tick行情
DataArray<Tick> *da = history_ticks("SHSE.600000", "2018-07-16 09:30:00", "2018-07-16 10:30:00");

if (da->status() == 0) //判断查询是否成功
{
    //遍历行情数组
    for (int i = 0; i < da->count(); i++)
    {
        cout << da->at(i).symbol << " " << da->at(i).price << endl;
    }
```

```
}

//释放数组
da->release();
```

另一种遍历方式

```

//查询一段tick行情
DataArray<Tick> *da = history_ticks("SHSE.600000", "2018-07-16 09:30:00", "2018-07-16 10:30:00");

if (da->status() == 0) //判断查询是否成功
{
    //获得原始数组指针
    Tick *ticks = da->data();

    //遍历行情数组
    for (int i = 0; i < da->count(); i++)
    {
        cout << ticks[i].symbol << " " << ticks[i].price << endl;
    }
}

//释放数组
da->release();
```

成员函数

status 获取函数调用结果

获得结果集之后， 第一步都应该先调用本成员函数判断查询数据是否成功。

函数原型：

```
1. int status()
```

参数：

参数名	类型	说明
返回值	int	0：成功， 非0：错误码

注意：

1. 如果status 断定为失败时， 不应该再遍历数组， 这时直接释放数组即可。

data 返回结构数组的指针

返回数组的原始指针， 可以用于遍历和拷贝数据。

函数原型：

```
1. T* data()
```

参数：

参数名	类型	说明
返回值	结构指针	具体取决于实例化类时的模板参数

注意：

1. 如果status 断定为失败时， 不应该再遍历数组， 这时直接释放数组即可。

count 返回数组长度

指的是元素个数

函数原型：

```
int count()
```

参数:

参数名	类型	说明
返回值	int	数据元素个数

注意:

1. 如果status 断定为失败时， 不应该再遍历数组， 这时直接释放数组即可。

at 返回元素值

返回下标为i的结构引用，从0开始

函数原型:

```
T& at(int i)
```

参数:

参数名	类型	说明
i	int	数组下标，从0开始
返回值	T&	返回数据元素的引用，具体取决于实例化类时的模板参数

注意:

1. 如果status 断定为失败时， 不应该再遍历数组， 这时直接释放数组即可。

release 释放数组

获取DataArray指针之后，最后都应该释放(不管status是什么状态)， 不然会造成内存泄露。调用release之后，不能再调用任何成员函数。

函数原型:

```
void release()
```

数据类

Tick - Tick结构

逐笔行情数据

```
struct Tick
{
    char                symbol[LEN_SYMBOL];
    double              created_at;             ///<utc时间，精确到毫秒
    float               price;                  ///<最新价
    float               open;                   ///<开盘价
    float               high;                   ///<最高价
    float               low;                    ///<最低价
    double              cum_volume;             ///<成交总量
    double              cum_amount;             ///<成交总金额/最新成交额,累计值
    long long           cum_position;           ///<合约持仓量(期),累计值
    double              last_amount;            ///<瞬时成交额
    int                 last_volume;            ///<瞬时成交量
    int                 trade_type;             ///(保留)交易类型,对应多开,多平等类型
    Quote               quotes[DEPTH_OF_QUOTE];    ///报价，下标从0开始，0-表示第一档，1-表示第二档，依次类推

};
```

报价 `Quote`

```
struct Quote
{
    float      bid_price;                       ///本档委买价
    long long  bid_volume;                      ///本档委买量
    float      ask_price;                       ///本档委卖价
    long long  ask_volume;                      ///本档委卖量
};
```

Bar - Bar结构

bar数据是指各种频率的行情数据

```

struct Bar
{
```

```
    char                    symbol[LEN_SYMBOL];
    double                  bob;                        ///bar的开始时间
    double                  eob;                        ///bar的结束时间
    float                   open;                       ///<开盘价
    float                   close;                      ///<收盘价
    float                   high;                       ///<最高价
    float                   low;                        ///<最低价
    double                  volume;                     ///<成交量
    double                  amount;                     ///<成交金额
    float                   pre_close;                  ///昨收盘价，只有日频数据赋值

    long long               position;                   ///<持仓量
    char                    frequency[LEN_FREQUENCY];   ///bar频度
};
```

L2Transaction - L2Transaction结构

L2行情的逐笔成交

```

struct L2Transaction
{
    char                    symbol[LEN_SYMBOL];
    double                  created_at;             ///成交时间，utc时间
    float                   price;                  ///成交价
    long long               volume;                   ///成交量
    char                    side;                   ///内外盘标记
    char                    exec_type;              ///成交类型
};
```

L2Order - L2Order结构

L2行情的逐笔委托

```

struct L2Order
{
    char                    symbol[LEN_SYMBOL];
    double                  created_at;             ///委托时间，utc时间
    float                   price;                  ///委托价
    long long               volume;                   ///委托量
    char                    side;                   ///买卖方向
    char                    order_type;             ///委托类型
};
```

L2OrderQueue - L2OrderQueue结构

L2行情的委托队列

```
1.
2.  struct L2OrderQueue
3.  {
4.      char                    symbol[LEN_SYMBOL];
5.      double                  created_at;                                 ///行情时间, utc
    时间
6.      float                   price;                                      ///最优委托价
7.      long long               volume;                                       ///委托量
8.      char                    side;                                       ///买卖方向
9.      int                     queue_orders;                               ///委托量队列中元
    素个数(最多50)
10.     int                     queue_volumes[LEN_MAX_ORDER_QUEUE];         ///委托量队列(最
    多50个, 有可能小于50, 有效数据长度取决于queue_orders)
11. };
```

交易类

Account - 账户结构

```
struct Account
{
    char            account_id[LEN_ID];         //账号ID
    char            account_name[LEN_NAME];     //账户登录名
    char            intro[LEN_INFO];            //账号描述
    char            comment[LEN_INFO];          //账号备注
};
```

AccountStatus - 账户状态结构

```
struct AccountStatus
{
    char            account_id[LEN_ID];         //账号ID
    char            account_name[LEN_NAME];     //账户登录名
    int             state;                      //账户状态
    int             error_code;                 //错误码
    char            error_msg[LEN_INFO];        //错误信息
};
```

Order - 委托结构

```

struct Order
{
```

```
    char          strategy_id[LEN_ID];                  //策略ID
    char          account_id[LEN_ID];                   //账号ID
    char          account_name[LEN_NAME];               //账户登录名

    char          cl_ord_id[LEN_ID];                    //委托客户端ID
    char          order_id[LEN_ID];                     //委托柜台ID
    char          ex_ord_id[LEN_ID];                    //委托交易所ID
    char          algo_order_id[LEN_ID];                //算法母单ID
    int           order_business;                       //业务类型

    char          symbol[LEN_SYMBOL];                   //symbol
    int           side;                                 //买卖方向，取值参考enum OrderSide
    int           position_effect;                      //开平标志，取值参考enum PositionEffect
    int           position_side;                        //持仓方向，取值参考enum PositionSide

    int           order_type;                           //委托类型，取值参考enum OrderType
    int           order_duration;                       //委托时间属性，取值参考enum OrderDuration
    int           order_qualifier;                      //委托成交属性，取值参考enum OrderQualifier
    int           order_src;                            //委托来源，取值参考enum OrderSrc
    int           position_src;                         //头寸来源（仅适用融资融券），取值参考 enum PositionSrc

    int           status;                               //委托状态，取值参考enum OrderStatus
    int           ord_rej_reason;                       //委托拒绝原因，取值参考enum OrderRejectReason
    char          ord_rej_reason_detail[LEN_INFO];      //委托拒绝原因描述

    double        price;                                //委托价格

    int           order_style;                          //委托风格，取值参考 enum OrderStyle
    long long     volume;                               //委托量
    double        value;                                //委托额
    double        percent;                              //委托百分比
    long long     target_volume;                        //委托目标量
    double        target_value;                         //委托目标额
    double        target_percent;                       //委托目标百分比

    long long     filled_volume;                        //已成量
    double        filled_vwap;                          //已成均价（股票实盘支持，期货实盘不支持）
    double        filled_amount;                        //已成金额（股票实盘支持，期货实盘不支持）

    long long     created_at;                           //委托创建时间
    long long     updated_at;                           //委托更新时间
};
```

AlgoOrder - 算法委托结构

```
1.
2.  struct AlgoOrder
3.  {
4.      char         strategy_id[LEN_ID];                    //策略ID
5.      char         account_id[LEN_ID];                     //账号ID
6.      char         account_name[LEN_NAME];                 //账户登录名
7.
8.      char         cl_ord_id[LEN_ID];                      //委托客户端ID
9.      char         order_id[LEN_ID];                       //委托柜台ID
10.     char         ex_ord_id[LEN_ID];                      //委托交易所ID
11.     int          order_business;                         //业务类型
12.
13.     char         symbol[LEN_SYMBOL];                     //symbol
14.     int          side;                                   //买卖方向，取值参考enum
    OrderSide
15.     int          position_effect;                        //开平标志，取值参考enum
    PositionEffect
16.     int          position_side;                          //持仓方向，取值参考enum
    PositionSide
17.
18.     int          order_type;                             //委托类型，取值参考enum
    OrderType
19.     int          order_duration;                         //委托时间属性，取值参考enum
    OrderDuration
20.     int          order_qualifier;                        //委托成交属性，取值参考enum
    OrderQualifier
21.     int          order_src;                              //委托来源，取值参考enum
    OrderSrc
22.     int          position_src;                           //头寸来源（仅适用融资融券），取值
    参考 enum PositionSrc
23.
24.     int          status;                                 //委托状态，取值参考enum
    OrderStatus
25.     int          ord_rej_reason;                         //委托拒绝原因，取值参考enum
    OrderRejectReason
26.     char         ord_rej_reason_detail[LEN_INFO];        //委托拒绝原因描述
27.
28.     double       price;                                  //委托价格
29.
30.     int          order_style;                            //委托风格，取值参考 enum
    OrderStyle
31.     long long    volume;                                 //委托量
32.     double       value;                                  //委托额
33.     double       percent;                                //委托百分比
34.     long long    target_volume;                          //委托目标量
35.     double       target_value;                           //委托目标额
36.     double       target_percent;                         //委托目标百分比
37.
38.     long long    filled_volume;                          //已成量
39.     double       filled_vwap;                            //已成均价
```

```
    double        filled_amount;                    //已成金额
    char          algo_name[LEN_NAME];              //算法策略名
    char          algo_param[LEN_PARAM];            //算法策略参数
    int           algo_status;                      //算法策略状态,仅作为AlgoOrderPause请求入参, 取值参考 enum AlgoOrderStatus
    char          algo_comment[LEN_COMMENT];        //算法单备注

    long long     created_at;                       //委托创建时间
    long long     updated_at;                       //委托更新时间
};
```

AlgoParam - 算法参数结构

```

struct AlgoParam
{
    char              algo_name[LEN_NAME];          //算法名称
    char              time_start[LEN_ISO_DATATIME]; //开始时间
    char              time_end[LEN_ISO_DATATIME];   //结束时间
    double            part_rate;                    //量比比例
    int               min_amount;                   //最小委托金额
};
```

ExecRpt - 回报结构

```
struct ExecRpt
{
    char              strategy_id[LEN_ID];          //策略ID
    char              account_id[LEN_ID];           //账号ID
    char              account_name[LEN_NAME];       //账户登录名

    char              cl_ord_id[LEN_ID];            //委托客户端ID
    char              order_id[LEN_ID];             //委托柜台ID
    char              exec_id[LEN_ID];              //委托回报ID

    char              symbol[LEN_SYMBOL];           //symbol

    int               position_effect;              //开平标志, 取值参考enum PositionEffect
    int               side;                         //买卖方向, 取值参考enum OrderSide
    int               ord_rej_reason;               //委托拒绝原因, 取值参考enum OrderRejectReason
    char              ord_rej_reason_detail[LEN_INFO];  //委托拒绝原因描述
    int               exec_type;                    //执行回报类型, 取值参考enum ExecType

    double            price;                        //委托成交价格
    long long         volume;                       //委托成交量
    double            amount;                       //委托成交金额
    double            cost;                         //委托成交成本金额（期货实盘支
```

```
持，股票实盘不支持）
    long long           created_at;                         //回报创建时间

};
```

Cash - 资金结构

```
struct Cash
{
    char            account_id[LEN_ID];                     //账号ID
    char            account_name[LEN_NAME];                 //账户登录名

    int             currency;                               //币种

    double          nav;                                     //总资产(cum_inout + cum_pnl + fpnl - cum_commission)
    double          fpnl;                                    //浮动盈亏(sum(each position fpnl))
    double          frozen;                                  //持仓占用资金（期货实盘支持，股票实盘不支持）
    double          order_frozen;                            //冻结资金
    double          available;                               //可用资金

    double          balance;                                 //资金余额
    double          market_value;                            //市值（股票实盘支持，期货实盘不支持）

    long long       created_at;                              //资金初始时间
    long long       updated_at;                              //资金变更时间

};
```

Position - 持仓结构

```
struct Position
{
    char                account_id[LEN_ID];             //账号ID
    char                account_name[LEN_NAME];         //账户登录名

    char                symbol[LEN_SYMBOL];             //symbol
    int                 side;                           //持仓方向，取值参考enum PositionSide
    long long           volume;                         //总持仓量; 昨持仓量(volume-volume_today)
    long long           volume_today;                   //今日持仓量
    double              vwap;                           //持仓均价(股票为基于开仓价的持仓均价，期货为基于结算价的持仓均价)
    double              vwap_diluted;                   //摊薄成本价
    double              vwap_open;                      //基于开仓价的持仓均价(期货)
    double              amount;                         //持仓额 (volume*vwap*multiplier)
```

```
    double                  price;                          //当前行情价格
    double                  fpnl;                           //持仓浮动盈亏((price-vwap)*volume*multiplier)
    double                  fpnl_open;                      //持仓浮动盈亏,基于开仓均价，适用于期货((price-vwap_open)*volume*multiplier)
    double                  cost;                           //持仓成本(vwap*volume*multiplier*margin_ratio)
    long long               order_frozen;                   //挂单冻结仓位
    long long               order_frozen_today;             //挂单冻结今仓仓位（仅上期所和上海能源交易中心支持）
    long long               available;                      //可用总仓位(volume-order_frozen); 可用昨仓位(available-available_today)
    long long               available_today;                //可用今仓位(volume_today-order_frozen_today)（仅上期所和上海能源交易中心支持）
    long long               available_now;                  //当前可平仓位
    double                  market_value;                   //持仓市值

    long long               created_at;                     //建仓时间
    long long               updated_at;                     //仓位变更时间

};
```

Indicator - 绩效指标结构

```
struct Indicator
{
    char      account_id[LEN_ID];           //账号ID
    double    pnl_ratio;                    //累计收益率(pnl/cum_inout)
    double    pnl_ratio_annual;             //年化收益率
    double    sharp_ratio;                  //夏普比率
    double    max_drawdown;                 //最大回撤
    double    risk_ratio;                   //风险比率
    int       open_count;                   //开仓次数
    int       close_count;                  //平仓次数
    int       win_count;                    //盈利次数
    int       lose_count;                   //亏损次数
    double    win_ratio;                    //胜率

    long long  created_at;                  //指标创建时间
    long long  updated_at;                  //指标变更时间
};
```

Parameter - 动态参数结构

```
struct Parameter
{
    char   key[LEN_ID];          //参数键
    double value;                //参数值
```

```
    double min;                 //可设置的最小值
    double max;                 //可设置的最大值
    char   name[LEN_NAME];      //参数名
    char   intro[LEN_INFO];     //参数说明
    char   group[LEN_NAME];     //组名
    bool   readonly;            //是否只读
};
```

CollateralInstrument - 担保品标的结构

```
struct CollateralInstrument
{
    char                symbol[LEN_SYMBOL];  //担保证券标的
    char                name[LEN_NAME];      //名称
    double              pledge_rate;         //折算率
};
```

BorrowableInstrument - 可做融券标的结构

```
struct BorrowableInstrument
{
    char                symbol[LEN_SYMBOL];         //可融证券标的
    char                name[LEN_NAME];             //名称
    double              margin_rate_for_cash;       //融资保证金比率
    double              margin_rate_for_security;   //融券保证金比率
};
```

BorrowableInstrumentPosition - 可做融券标的持仓结构

```
struct BorrowableInstrumentPosition
{
    char                symbol[LEN_SYMBOL];         //可融证券标的
    char                name[LEN_NAME];             //名称
    double              balance;                    //证券余额
    double              available;                  //证券可用
};
```

CreditContract - 融资融券合约结构

```
struct CreditContract
{
    char symbol[LEN_SYMBOL];            //证券代码 stkcode
    char name[LEN_NAME];                //名称
    int orderdate;                      //委托日期
    char ordersno[LEN_ID];              //委 托 号
    char creditdirect;                  //融资融券方向
```

```
    double orderqty;                    //委托数量
    double matchqty;                    //成交数量
    double orderamt;                    //委托金额
    double orderfrzamt;                 //委托冻结金额
    double matchamt;                    //成交金额
    double clearamt;                    //清算金额
    char lifestatus;                    //合约状态
    int enddate;                        //负债截止日期
    int oldenddate;                     //原始的负债截止日期
    double creditrepay;                 //T日之前归还金额
    double creditrepayunfrz;            //T日归还金额
    double fundremain;                  //应还金额
    double stkrepay;                    //T日之前归还数量
    double stkrepayunfrz;               //T日归还数量
    double stkremain;                   //应还证券数量
    double stkremainvalue;              //应还证券市值
    double fee;                         //融资融券息、费
    double overduefee;                  //逾期未偿还息、费
    double fee_repay;                   //已偿还息、费
    double punifee;                     //利息产生的罚息
    double punifee_repay;               //已偿还罚息
    double rights;                      //未偿还权益金额
    double overduerights;               //逾期未偿还权益
    double rights_repay;                //已偿还权益
    double lastprice;                   //最新价
    double profitcost;                  //浮动盈亏
    int sysdate;                        //系统日期
    char sno[LEN_ID];                   //合约编号
    int lastdate;                       //最后一次计算息费日期
    int closedate;                      //合约全部偿还日期
    double punidebts;                   //逾期本金罚息
    double punidebts_repay;             //本金罚息偿还
    double punidebtsunfrz;              //逾期本金罚息
    double punifeeunfrz;                //逾期息费罚息
    double punirights;                  //逾期权益罚息
    double punirights_repay;            //权益罚息偿还
    double punirightsunfrz;             //逾期权益罚息
    double feeunfrz;                    //实时偿还利息
    double overduefeeunfrz;             //实时偿还逾期利息
    double rightsqty;                   //未偿还权益数量
    double overduerightsqty;            //逾期未偿还权益数量
};
```

CreditCash - 融资融券资金信息结构

```
struct CreditCash
{
    double fundintrrate;                    //融资利率
    double stkintrrate;                     //融券利率
    double punishintrrate;                  //罚息利率
    char   creditstatus;                    //信用状态
```

```
    double marginrates;                 //维持担保比例
    double realrate;                    //实时担保比例
    double asset;                       //总资产
    double liability;                   //总负债
    double marginavl;                   //保证金可用数
    double fundbal;                     //资金余额
    double fundavl;                     //资金可用数
    double dsaleamtbal;                 //融券卖出所得资金
    double guaranteeout;                //可转出担保资产
    double gagemktavl;                  //担保证券市值
    double fdealavl;                    //融资本金
    double ffee;                        //融资息费
    double ftotaldebts;                 //融资负债合计
    double dealfmktavl;                 //应付融券市值
    double dfee;                        //融券息费
    double dtotaldebts;                 //融券负债合计
    double fcreditbal;                  //融资授信额度
    double fcreditavl;                  //融资可用额度
    double fcreditfrz;                  //融资额度冻结
    double dcreditbal;                  //融券授信额度
    double dcreditavl;                  //融券可用额度
    double dcreditfrz;                  //融券额度冻结
    double rights;                      //红利权益
    double serviceuncomerqrights;       //红利权益(在途)
    double rightsqty;                   //红股权益
    double serviceuncomerqrightsqty;    //红股权益(在途)
    double acreditbal;                  //总额度
    double acreditavl;                  //总可用额度
    double acashcapital;                //所有现金资产（所有资产、包括融券卖出）
    double astkmktvalue;                //所有证券市值（包含融资买入、非担保品）
    double withdrawable;                //可取资金
    double netcapital;                  //净资产
    double fcreditpnl;                  //融资盈亏
    double dcreditpnl;                  //融券盈亏
    double fcreditmarginoccupied;       //融资占用保证金
    double dcreditmarginoccupied;       //融券占用保证金
    double collateralbuyableamt;        //可买担保品资金
    double repayableamt;                //可还款金额
    double dcreditcashavl;              //融券可用资金
};
```

IPOQI -新股申购额度

```
struct IPOQI
{
    char exchange[LEN_TYPE];        //市场代码
    double quota;                   //市场配额
    double sse_star_quota;          //上海科创板配额
};
```

IPOInstruments - 新股标的结构

```
struct IPOInstruments
{
    char     symbol[LEN_SYMBOL];     //申购新股symbol
    double   price;                  //申购价格
    int      min_vol;                //申购最小数量
    int      max_vol;                //申购最大数量
};
```

IPOMatchNumber - 配号结构

```
struct IPOMatchNumber
{
    char order_id[LEN_ID];           //委托号
    char symbol[LEN_SYMBOL];         //新股symbol
    int   volume;                    //成交数量
    char match_number[LEN_ID];       //申购配号
    int  order_at;                   //委托日期
    int  match_at;                    //配号日期
};
```

IPOLotInfo - 中签结构

```
struct IPOLotInfo
{
    char symbol[LEN_SYMBOL];   //新股symbol
    int order_at;              //委托日期
    int lot_at;                //中签日期
    int lot_volume;            //中签数量
    int give_up_volume;        //放弃数量
    double price;              //中签价格
    double amount;             //中签金额
    double pay_volume;         //已缴款数量
    double pay_amount;         //已缴款金额

};
```

枚举常量

OrderStatus - 委托状态

```
enum OrderStatus
{
    OrderStatus_Unknown = 0,
    OrderStatus_New = 1,                    //已报
    OrderStatus_PartiallyFilled = 2,        //部成
    OrderStatus_Filled = 3,                 //已成
    OrderStatus_Canceled = 5,               //已撤
    OrderStatus_PendingCancel = 6,          //待撤
    OrderStatus_Rejected = 8,               //已拒绝
    OrderStatus_Suspended = 9,              //挂起
    OrderStatus_PendingNew = 10,            //待报
    OrderStatus_Expired = 12,               //已过期

};
```

OrderSide - 委托方向

```
enum OrderSide
{
    OrderSide_Unknown = 0,
    OrderSide_Buy = 1,    //买入
    OrderSide_Sell = 2,    //卖出
};
```

OrderType - 委托类型

```
enum OrderType
{
```

```
    OrderType_Unknown = 0,
    OrderType_Limit = 1,     //限价委托
    OrderType_Market = 2,     //市价委托
    OrderType_Stop = 3,     //止损止盈委托
};
```

OrderDuration - 委托时间属性

```
enum OrderDuration
{
    OrderDuration_Unknown = 0,
    OrderDuration_FAK = 1,   //即时成交剩余撤销(fill and kill)
    OrderDuration_FOK = 2,   //即时全额成交或撤销(fill or kill)
    OrderDuration_GFD = 3,   //当日有效(good for day)
    OrderDuration_GFS = 4,   //本节有效(good for section)
    OrderDuration_GTD = 5,   //指定日期前有效(goodl till date)
    OrderDuration_GTC = 6,   //撤销前有效(good till cancel)
    OrderDuration_GFA = 7,   //集合竞价前有效(good for auction)
};
```

OrderQualifier - 委托成交属性

```
enum OrderQualifier
{
    OrderQualifier_Unknown = 0,
    OrderQualifier_BOC = 1,   //对方最优价格(best of counterparty)
    OrderQualifier_BOP = 2,   //己方最优价格(best of party)
    OrderQualifier_B5TC = 3,   //最优五档剩余撤销(best 5 then cancel)
    OrderQualifier_B5TL = 4,   //最优五档剩余转限价(best 5 then limit)
};
```

ExecType - 执行回报类型

```
enum ExecType
{
    ExecType_Unknown = 0,
    ExecType_New = 1,                    //已报
    ExecType_Canceled = 5,               //已撤销
    ExecType_PendingCancel = 6,          //待撤销
    ExecType_Rejected = 8,               //已拒绝
    ExecType_Suspended = 9,              //挂起
    ExecType_PendingNew = 10,            //待报
    ExecType_Expired = 12,               //过期
    ExecType_Trade = 15,                 //成交
    ExecType_OrderStatus = 18,           //委托状态
    ExecType_CancelRejected = 19,        //撤单被拒绝
};
```

PositionEffect - 开平仓类型

```
enum PositionEffect
{
    PositionEffect_Unknown = 0,
    PositionEffect_Open = 1,      //开仓
    PositionEffect_Close = 2,      //平仓,具体语义取决于对应的交易所
    PositionEffect_CloseToday = 3,      //平今仓
    PositionEffect_CloseYesterday = 4,      //平昨仓
};
```

PositionSide - 持仓方向

```
enum PositionSide
{
    PositionSide_Unknown = 0,
    PositionSide_Long = 1,    //多方向
    PositionSide_Short = 2,    //空方向
};
```

OrderRejectReason - 订单拒绝原因

```
enum OrderRejectReason
{
    OrderRejectReason_Unknown = 0,                          //未知原因
    OrderRejectReason_RiskRuleCheckFailed = 1,              //不符合风控规则
    OrderRejectReason_NoEnoughCash = 2,                     //资金不足
    OrderRejectReason_NoEnoughPosition = 3,                 //仓位不足
    OrderRejectReason_IllegalAccountId = 4,                 //非法账户ID
    OrderRejectReason_IllegalStrategyId = 5,                //非法策略ID
    OrderRejectReason_IllegalSymbol = 6,                    //非法交易代码
    OrderRejectReason_IllegalVolume = 7,                    //非法委托量
    OrderRejectReason_IllegalPrice = 8,                     //非法委托价
    OrderRejectReason_AccountDisabled = 10,                 //交易账号被禁止交易
    OrderRejectReason_AccountDisconnected = 11,             //交易账号未连接
    OrderRejectReason_AccountLoggedout = 12,                //交易账号未登录
    OrderRejectReason_NotInTradingSession = 13,             //非交易时段
    OrderRejectReason_OrderTypeNotSupported = 14,           //委托类型不支持
    OrderRejectReason_Throttle = 15,                        //流控限制
    OrderRejectReason_SymbolSusppended = 16,                //交易代码停牌
    OrderRejectReason_Internal = 999,                       //内部错误

    CancelOrderRejectReason_OrderFinalized = 101,           //委托已完成
    CancelOrderRejectReason_UnknownOrder = 102,             //未知委托
    CancelOrderRejectReason_BrokerOption = 103,             //柜台设置
    CancelOrderRejectReason_AlreadyInPendingCancel = 104,   //委托撤销中
};
```

CashPositionChangeReason - 仓位变更原因

```
enum CashPositionChangeReason
{
    CashPositionChangeReason_Unknown = 0,
    CashPositionChangeReason_Trade = 1,  //交易
    CashPositionChangeReason_Inout = 2,  //出入金/出入持仓
    CashPositionChangeReason_Dividend = 3,  //分红送股
};
```

AccountState - 交易账户状态

```

enum AccountState
{
    State_UNKNOWN = 0,        //未知
    State_CONNECTING = 1,     //连接中
    State_CONNECTED = 2,      //已连接
    State_LOGGEDIN = 3,       //已登录
    State_DISCONNECTING = 4, //断开中
    State_DISCONNECTED = 5,  //已断开
    State_ERROR = 6           //错误
};
```

AlgoOrderStatus - 算法单状态,暂停/恢复算法单时有效

```
enum AlgoOrderStatus
{
    AlgoOrderStatus_Unknown = 0,
    AlgoOrderStatus_Resume = 1,                    //恢复母单
    AlgoOrderStatus_Pause = 2,                     //暂停母单
    AlgoOrderStatus_PauseAndCancelSubOrders = 3  //暂算母单并撤子单
};
```

PositionSrc - 头寸来源(仅适用融券融券)

```
enum PositionSrc
{
    PositionSrc_Unknown = 0,
    PositionSrc_L1 = 1,          //普通池
    PositionSrc_L2 = 2           //专项池
};
```

SecurityType - 证券类型

```
enum SecurityType
```

```
{
    SecurityType_Unknown = 0,
    SecurityType_Stock = 1,  //股票
    SecurityType_Fund = 2,   //基金
    SecurityType_Index = 3,  //指数
    SecurityType_Future = 4,  //期货
    SecurityType_Option = 5,  //期权
    SecurityType_Credit = 6,  //两融
    SecurityType_Bond = 7,    //债券
    SecurityType_Bond_Convertible = 8 //可债券
};
```

OrderBusiness - 业务类型

```
enum OrderBusiness
{
    OrderBusiness_NORMAL = 0,                           //普通交易
    OrderBusiness_IPO_BUY = 100,                        //新股申购
    OrderBusiness_CREDIT_BOM = 200,                     //融资买入(buying on margin)
    OrderBusiness_CREDIT_SS = 201,                      //融券卖出(short selling)
    OrderBusiness_CREDIT_RSBBS = 202,                   //买券还券(repay share by buying share)
    OrderBusiness_CREDIT_RCBSS = 203,                   //卖券还款(repay cash by selling share)
    OrderBusiness_CREDIT_DRS = 204,                     //直接还券(directly repay share)
    OrderBusiness_CREDIT_DRC = 211,                     //直接还款(directly repay cash)
    OrderBusiness_CREDIT_CPOM = 205,                    //融资平仓(close position on margin)
    OrderBusiness_CREDIT_CPOSS = 206,                   //融券平仓(close position on short selling)
    OrderBusiness_CREDIT_BOC = 207,                     //担保品买入(buying on collateral)
    OrderBusiness_CREDIT_SOC = 208,                     //担保品卖出(selling on collateral)
    OrderBusiness_CREDIT_CI = 209,                      //担保品转入(collateral in)
    OrderBusiness_CREDIT_CO = 210,                      //担保品转出(collateral out)
    OrderBusiness_ETF_BUY = 301,                        //ETF申购(purchase)
    OrderBusiness_ETF_RED = 302,                        //ETF赎回(redemption)
    OrderBusiness_FUND_SUB = 303,                       //基金认购(subscribing)
    OrderBusiness_FUND_BUY = 304,                       //基金申购(purchase)
    OrderBusiness_FUND_RED = 305,                       //基金赎回(redemption)
    OrderBusiness_FUND_CONVERT = 306,                   //基金转换(convert)
    OrderBusiness_FUND_SPLIT = 307,                     //基金分拆(split)
    OrderBusiness_FUND_MERGE = 308,                     //基金合并(merge)
    OrderBusiness_BOND_RRP = 400,                       //债券逆回购(reverse repurchase agreement (RRP) or reverse repo)
    OrderBusiness_BOND_CONVERTIBLE_BUY = 401,           //可转债申购(purchase)
    OrderBusiness_BOND_CONVERTIBLE_CALL = 402,          //可转债转股
    OrderBusiness_BOND_CONVERTIBLE_PUT = 403,           //可转债回售
    OrderBusiness_BOND_CONVERTIBLE_PUT_CANCEL = 404     //可转债回售撤销
};
```

错误码

错误码	描述
0	成功
1000	错误或无效的token
1001	无法连接到终端服务
1010	无法获取掘金服务器地址列表
1011	消息包解析错误
1012	网络消息包解析错误
1013	交易服务调用错误
1014	历史行情服务调用错误
1015	策略服务调用错误
1016	动态参数调用错误
1017	基本面数据服务调用错误
1018	回测服务调用错误
1019	交易网关服务调用错误
1020	无效的ACCOUNT_ID
1021	非法日期格式
1100	交易消息服务连接失败
1101	交易消息服务断开
1200	实时行情服务连接失败
1201	实时行情服务连接断开
1300	初始化回测失败，可能是终端未启动或无法连接到终端
1301	回测时间区间错误
1302	回测读取缓存数据错误
1303	回测写入缓存数据错误