

# xquant: CTP开发+系统设计+序列切片运行设计

---

CTP提供的库文件:

CTP调用方法

主程序设计

参考 clickhouse的server设计 Daemon设计

## 【切片语言】

LLVM编译

## 1. 【策略程序设计】

### 1.1. 交易模式

#### 1.1.1. 目前支持的方式:

### 1.2.

### 1.3. 调用 RSV EMA SMA 要注意的事项:

### 1.4. 策略程序逻辑设计

### 1.5. base/base.cpp功能

## 2. 【策略代码规范】

### 2.1. 策略程序逻辑

#### 2.1.1. 定义策略所需合约及周期 table.json

#### 2.1.2. base对象初始化

#### 2.1.3. 每个合约每个周期绑定base中的相对应的unit

### 2.2. 策略代码规范

#### 2.2.1. 标准序列的定义:

#### 2.2.2. 其它序列的定义:

#### 2.2.3. 不存在的搞法:

#### 2.2.4. 错误做法

#### 2.2.5. class EMA\_t 解析

序列对象不可直接赋值给double。

double可以赋值给序列对象。

#### 2.2.6. 【第一类写法】

写法一: EMA\_T 即: #define EMA\_T static EMA\_t

写法二：和写法一实际上一样。

写法三：【推荐写法】

2.2.7. 【废弃】 【第二类写法】 针对入参有值的情况，比如 用OHLCV作为入参

2.2.8. 【第三类写法】 显式调用一次【可以使用，必须使用显示调用】

第四类写法（废弃）

定义OHLCV（对象）

定义double序列（对象）

定义 int 序列（对象）

定义bool序列（对象）

定义其它（对象）

double序列对象的运算

复杂的计算

序列对象的其它数学运算

初始化过程

数据类型及运算（切片运行）

存在的问题

2.3. main中初始化 future

写法注意事项

服务进程







数据级别时间同步策略

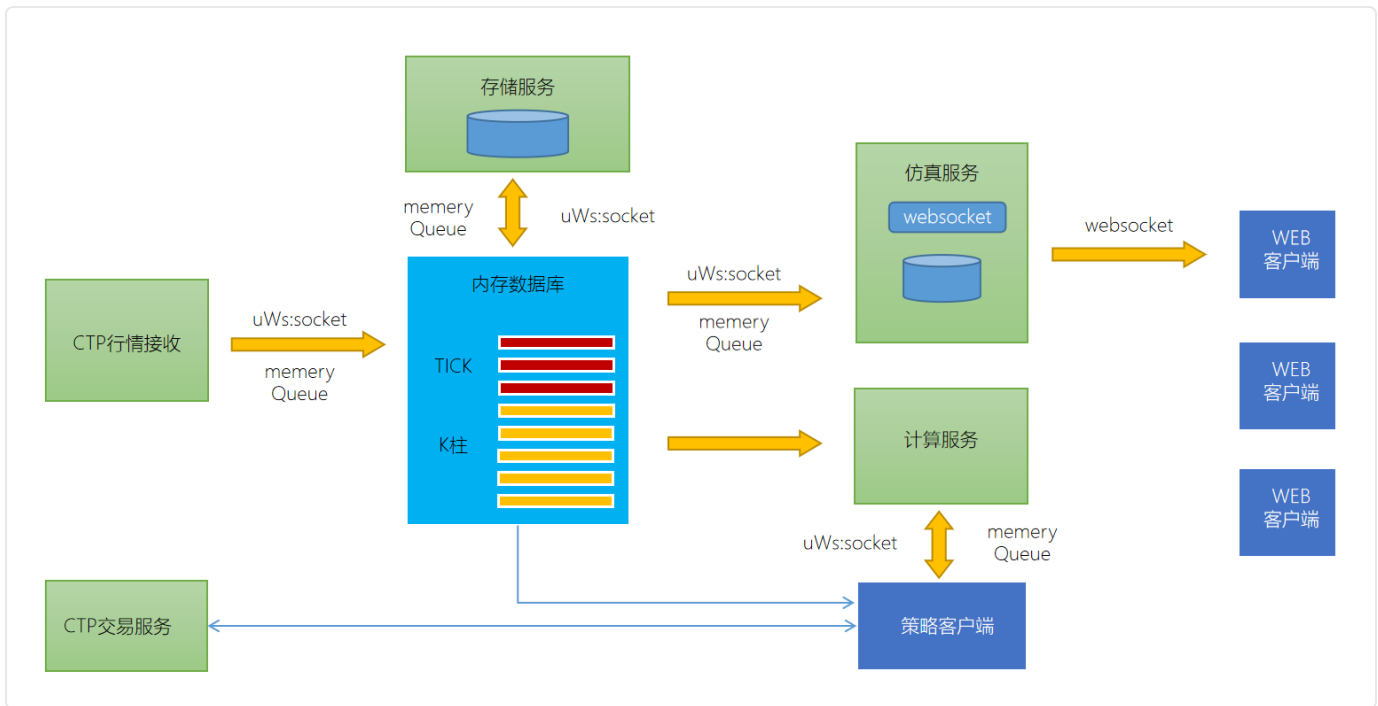
一：只提供最小级别，由策略自行合并成其它级别

二：由服务器提供不同级别的数据，服务器进行时间同步。

增加

## CTP提供的库文件：

	error.dtd	2021/7/19 14:56	Document Type ...	1 KB
	error.xml	2022/1/7 15:38	XML 文档	22 KB
	ThostFtdcMdApi.h	2022/6/13 15:01	C++ Header file	6 KB
	ThostFtdcTraderApi.h	2022/6/13 15:00	C++ Header file	38 KB
	ThostFtdcUserApiDataType.h	2022/6/13 15:01	C++ Header file	259 KB
	ThostFtdcUserApiStruct.h	2022/6/13 15:01	C++ Header file	271 KB
	thostmduserapi_se.dll	2022/6/13 15:03	应用程序扩展	2,836 KB
	thostmduserapi_se.lib	2022/6/13 15:03	Object File Library	4 KB
	thosttraderapi_se.dll	2022/6/13 15:01	应用程序扩展	3,212 KB
	thosttraderapi_se.lib	2022/6/13 15:01	Object File Library	4 KB



内存数据库的需求：

- 1、单列处理。
- 2、每列长度固定，写满以后，将以前的数据存储到硬盘上，类似循环队列。
- 3、每列均可以用作循环队列，供其它进程或线程读写。无锁无等待。
- 4、CTP行情接收线程可以直接写tick队列，计算线程无锁读取，进行K柱等计算，并将结果存储于内存数据库中。
- 5、仿真服务可以读取内存数据库，并push给前端用户。
- 6、本地策略进程，可以直接读取内存数据库。
- 7、存储服务进程，可以直接读取数据库。

# CTP调用方法

```
C Bars.h 1243
M CMakeLists.txt 1244
C ctp_DataCollect.h 1245
C ctp_make_mduser.cpp M 1246
C ctp_make_mduser.h M 1247
C ctp_make_trader.cpp U 1248
C ctp_make_trader.h U 1249
C ctp_mduser_spi.h M 1250
C ctp_trader_spi.h M 1251
C ctp_traderApi.cpp 1252
C ctp_traderApi.h 1253
C ctp_traderSpi.cpp 1254
C ctp_traderSpi.h 1255
C File.cpp M 1256

// 请求查询合约
void ReqQryInstrument() {
    CThostFtdcQryInstrumentField a = {0};
    strcpy(a.ExchangeID, g_chExchangeID);
    // strcpy(a.InstrumentID, g_chInstrumentID);
    // strcpy(a.ExchangeInstID, "");
    // strcpy(a.ProductID, "a");
    int b = m_pUserApi->ReqQryInstrument(&a, nRequestID++);
    printf((b == 0) ? "请求查询合约.....发送成功\n"
        : "请求查询合约.....发送失败, 错误序号=%d\n",
        b);
}

/// 请求查询合约响应
virtual void OnRspQryInstrument(CThostFtdcInstrumentField *pInstrument,
    CThostFtdcRspInfoField *pRspInfo, int nRequestID,
```

```
C ctp_DataCollect.h 1255
C ctp_make_mduser.cpp M 1256
C ctp_make_mduser.h M 1257
C ctp_make_trader.cpp U 1258
C ctp_make_trader.h U 1259
C ctp_mduser_spi.h M 1260
C ctp_trader_spi.h M 1261
C ctp_traderApi.cpp 1262
C ctp_traderApi.h 1263
C ctp_traderSpi.cpp 1264
C ctp_traderSpi.h 1265
C File.cpp M 1266
C File.h 1267
C global.cpp M 1268
C global.h M 1269
C quant_server.cpp M 1270
C queue_read.cpp M 1271

C ctp_traderSpi.h 1272
C ctp_traderSpi.cpp 1273
C ctp_traderSpi.h 1274
C ctp_traderSpi.cpp 1275
C ctp_traderSpi.h 1276

/// 请求查询合约响应
virtual void OnRspQryInstrument(CThostFtdcInstrumentField *pInstrument,
    CThostFtdcRspInfoField *pRspInfo, int nRequestID,
    bool bIsLast) {
    CTraderSpi::OnRspQryInstrument(pInstrument, pRspInfo, nRequestID, bIsLast);
    if (pInstrument) {
        if (strlen(pInstrument->InstrumentID) < 7) {
            if (pInstrument->InstrumentID[2] < '9' && pInstrument->InstrumentID[2] > '0') {
                md_InstrumentID.push_back(pInstrument->InstrumentID);
                std::cout << pInstrument->InstrumentID << std::endl;
            }
        }
    }
    if (bIsLast) {
        // SetEvent(xinhao);
        std::cout << " last !!!!!!!!!!!!!!!!!!!" << std::endl;
        g_count = 1;
        usleep(2000000);
    }
}
```

```
extern
> extern
> src
  > Common
    > Bars.cpp
    > Bars.h
    > CMakeLists.txt
    > ctp_DataCollect.h
    > ctp_make_mduser.cpp
    > ctp_make_mduser.h
    > ctp_make_trader.cpp
    > ctp_make_trader.h
    > ctp_mduser_spi.h
    > ctp_trader_spi.h
    > ctp_traderApi.cpp
    > ctp_traderApi.h
    > ctp_traderSpi.cpp
    > ctp_traderSpi.h
    > File.cpp
    > File.h
    > global.cpp
    > global.h
    > quant_server.cpp
    > queue_read.cpp
    > queue_read.h

231 void SubscribeMarketData() // 收行情
232 {
233     int md_num = 0;
234     char **ppInstrumentID = new char *[5000];
235     for (int count1 = 0; count1 <= md_InstrumentID.size() / 500; count1++) {
236         if (count1 < md_InstrumentID.size() / 500) {
237             int a = 0;
238             for (a = 0; a < 500; a++) {
239                 ppInstrumentID[a] = const_cast<char *>(md_InstrumentID.at(md_num).c_str());
240                 md_num++;
241             }
242             int result = m_pUserMdApi->SubscribeMarketData(ppInstrumentID, a);
243             printf((result == 0) ? "订阅行情请求1.....发送成功\n"
244                 : "订阅行情请求1.....发送失败, 错误序号=%d\n",
245                 result);
246         } else if (count1 == md_InstrumentID.size() / 500) {
247             int count2 = 0;
248             for (count2 = 0; count2 < md_InstrumentID.size() % 500; count2++) {
249                 ppInstrumentID[count2] = const_cast<char *>(md_InstrumentID.at(md_num).c_str());
250                 md_num++;
251             }
252             int result = m_pUserMdApi->SubscribeMarketData(ppInstrumentID, count2);
253             printf((result == 0) ? "订阅行情请求2.....发送成功\n"
254                 : "订阅行情请求2.....发送失败, 错误序号=%d\n",
255                 result);
256         }
257     }
258 }
```

```
> ctp668
> ctplib
> extern
> src
  > Common
    > Bars.cpp
    > Bars.h
    > CMakeLists.txt
    > ctp_DataCollect.h
    > ctp_make_mduser.cpp
    > ctp_make_mduser.h
    > ctp_make_trader.cpp
    > ctp_make_trader.h
    > ctp_mduser_spi.h
    > ctp_trader_spi.h
    > ctp_traderApi.cpp
    > ctp_traderApi.h
    > ctp_traderSpi.cpp
    > ctp_traderSpi.h
    > File.cpp
    > File.h
    > global.cpp

280 // 深度行情通知
281 virtual void OnRtnDepthMarketData(CHostFtdcDepthMarketDataField *pDepthMarketData) {
282     // 获取系统时间
283     // SYSTEMTIME sys;
284     // GetLocalTime(&sys);
285     // printf("%02d:%02d:%02d.%03d\t", sys.wHour, sys.wMinute, sys.wSecond,
286     // sys.wMilliseconds);
287     // printf("<OnRtnDepthMarketData> before !");
288
289     // std::thread::id this_id = std::this_thread::get_id();
290     // std::cout << " CHostFtdcMdSpi thread OnRtnDepthMarketData :< this_id << "\n";
291
292     if (pDepthMarketData) {
293         g_tick_queue->push([&pDepthMarketData](CHostFtdcDepthMarketDataField &msg) {
294             memcpy(&msg, pDepthMarketData, sizeof(CHostFtdcDepthMarketDataField));
295         });
296         // SaveTick(pDepthMarketData); ...
297         /* ...
298         /* 处理k_bar !!
299         std::map<std::string, uBEE::FuBo *>::iterator iter;
300         iter = M_CtpFuBo.find(pDepthMarketData->InstrumentID);
301         if (iter == M_CtpFuBo.end())
302             std::cout << "we do not find :< pDepthMarketData->InstrumentID << std::endl;
303         else {
304             HandleTick(iter->second, pDepthMarketData, SEND_SAVE_ALL);
305         }
```

# 主程序设计

参考 clickhouse的server设计 Daemon设计

## 【切片语言】

<https://developer.aliyun.com/article/59983>

<https://ld246.com/article/1569932084075>

基于Illum开发新的编程语言-掘金

Illum开发新语言-掘金

<https://www.sqlite.org/arch.html>

<https://www.sqlite.org/lemon.html>

pine script

[https://cn.tradingview.com/pine-script-reference/v5/#op\\_=%3E](https://cn.tradingview.com/pine-script-reference/v5/#op_=%3E)

<https://www.tradingview.com/pine-script-docs/en/v5/index.html>

[https://unknown-marketwizards.github.io/pine\\_script\\_docs\\_zh/#/](https://unknown-marketwizards.github.io/pine_script_docs_zh/#/)

LLVM编译

```
1 [rabbit @ moztart ~/llvm/llvm-project] $ cmake -S llvm -B build -G Ninja -DCMAKE_BUILD_TYPE=Release
2 [rabbit @ moztart ~/llvm/llvm-project] $ ninja -C build check-llvm
```

1. 【策略程序设计】

1.1. 交易模式

方式 Mode::BACK_TEST	方式 Mode::REAL_CTP;	方式 Mode::SIMULATION;	方式 Mode:: CHOOSING;
remote: websocket 自己指定日期， 开始、结束日期	remote: websocket --> 远端CTP  无须确定日期	remote: websocket 获得Kbar 开机从远程取数据 无须确定日期	选股模式 remote: websocket 获取数据 无须确定日期

👉local: 本地文件 指定日期 指定目录	local: 本地直接建立 CTP连接 无法确定日期	无须支持	👉local:本地文件 指定日期 指定目录
local: 本地数据库 指定日期	无须支持	无须支持	local:本地数据库 指定日期

```

1 enum class Mode {
2     BACK_TEST, // 回测          // remote : 数据从websocket来    local : 本地数
   据, 文件或数据库
3     REAL_CTP,   // CTP实盘      // remote : 数据从websocket来    local : 本地CT
   P连接
4     SIMULATION, // 模拟交易      // remote : 数据从websocket来    local : 本地数
   据, 文件或数据库
5     CHOOSING    // 选股          // remote : 数据从websocket来    local : 本地数
   据, 文件或数据库
6 };

```

### 1.1.1. 目前支持的方式:

#### 1. 本地选股

```

bb.mode_ = Mode::CHOOSING;
bb.remote_ = false;
bb.kbarfile_ = kbarfile

```

本方式不读取trade.json，直接分配内存。读取本地文件。

分配的空间有 <8,9> <8,12>

请查看 `int Base::FutureInit(const std::string &tradejson)`

- 需要循环读取文件，并设置 `bb.kbarfile_`

## 1.2.

### 1.3. 调用 RSV EMA SMA 要注意的事项:

- double 要初始化 成 NAN !

```
1         this->data = new double[this->unit->ohlcw_len];
2         for (int i = 0; i < this->unit->ohlcw_len; i++) {
3             this->data[i] = NAN;
4         }
```

- RSV的调用 要初始化:

```
1     RSV_t() {
2         preH = NAN;
3         preL = NAN;
4         preF = 0;
5     }
```

## 1.4. 策略程序逻辑设计

一个策略有三种数据来源,

- 某个合约的某个周期: 与 `future<>` 对应
- 某个合约的多个周期: 与 `future<N>` 对应
- 多个合约的多个周期: 与 `future<M,N>` 对应

## 1.5. base/base.cpp功能

1. 为交易策略的每个合约每个周期初始化一个unit, 分配相应的 ohlcw空间。
2. 连接服务器, 获得相应的数据。
3. 进行策略循环。base.run();

为数据源定义了一个二维指针数组: `Unit* unit_array_[FutureNum][PeriodNum];`

`Unit` 是一个数据单元, 是一组 `OHLCV` 及相关数据, 每个 `Unit` 是独立的。



下面的这个写法，可能会移到bb的构造函数中去。

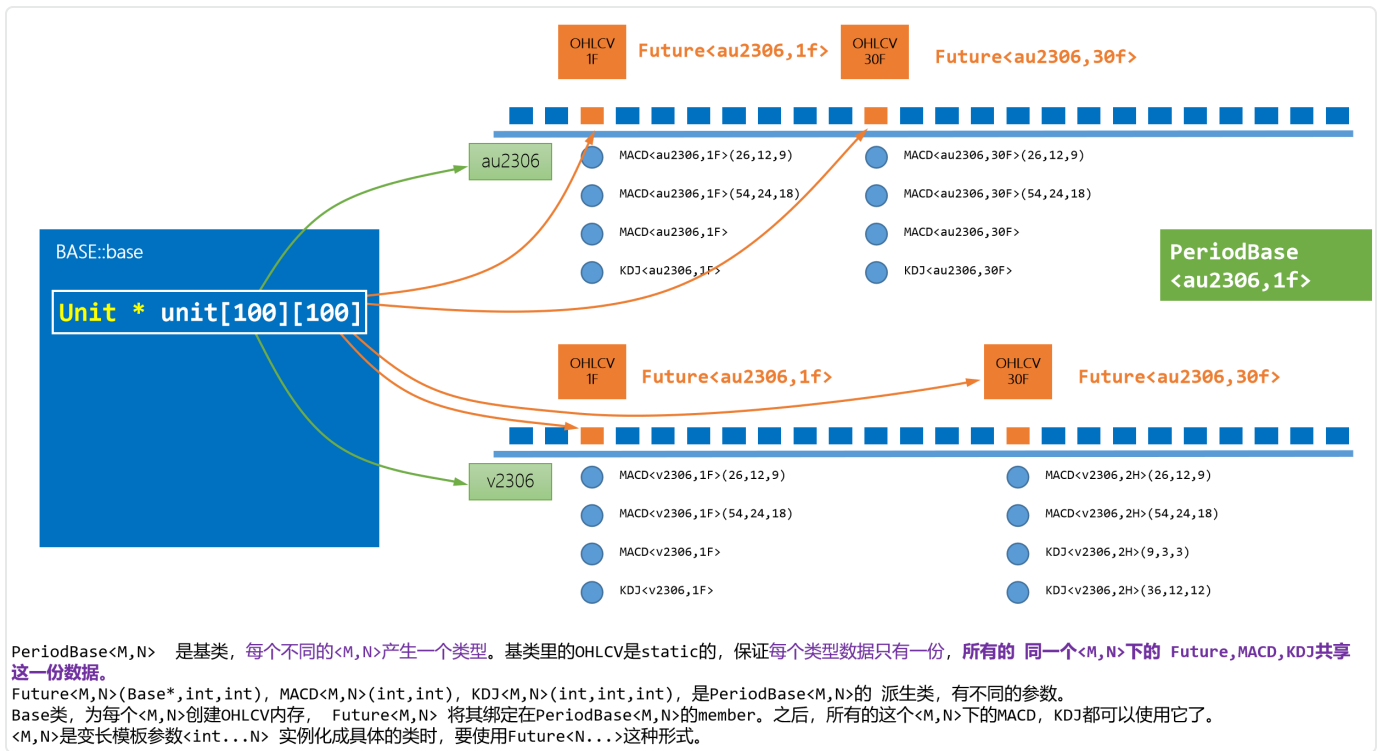
```
bb.FutureInit("./tbl/trade.json", LOCAL_DATA);
```

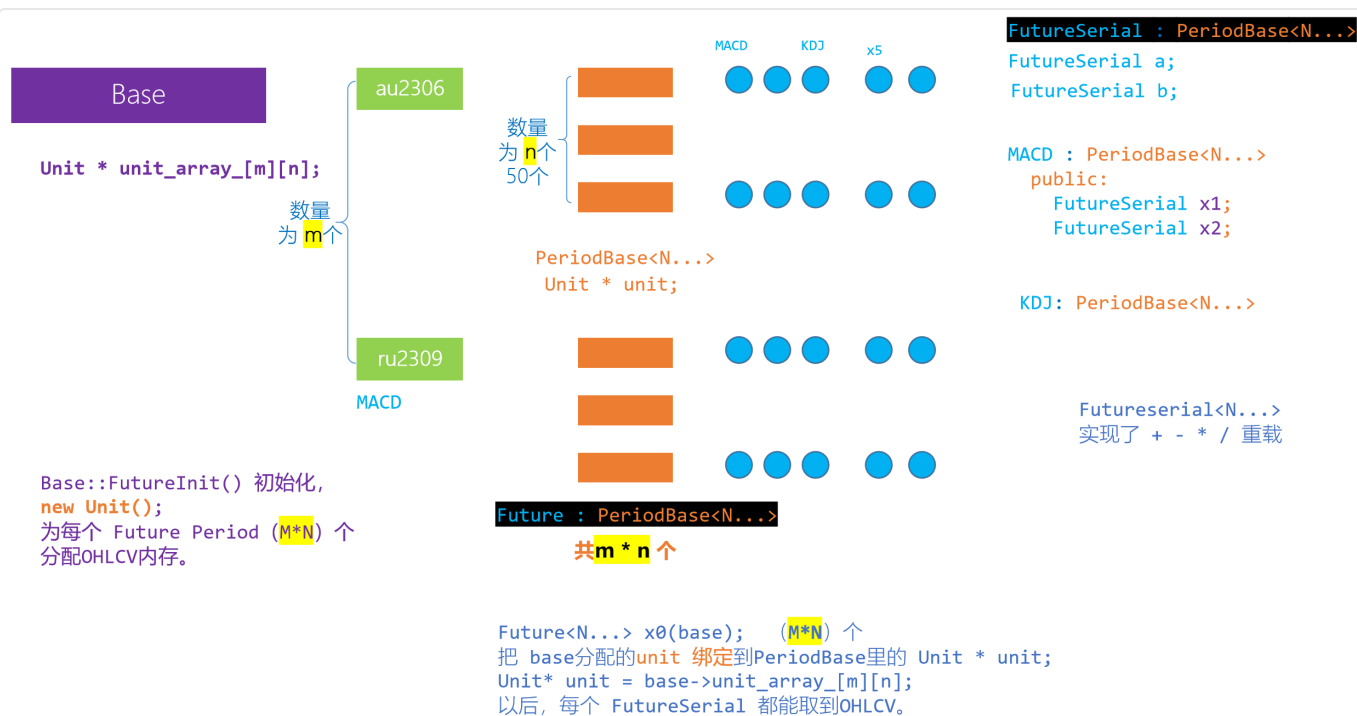
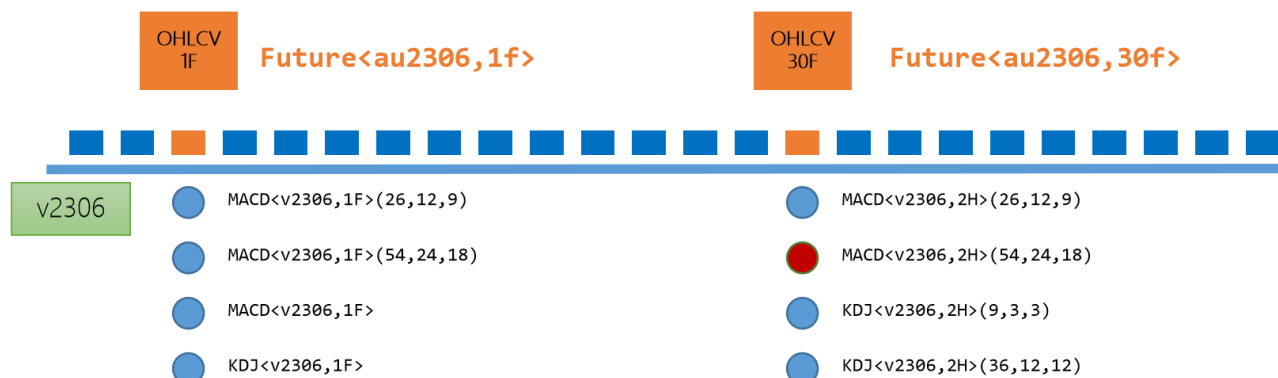
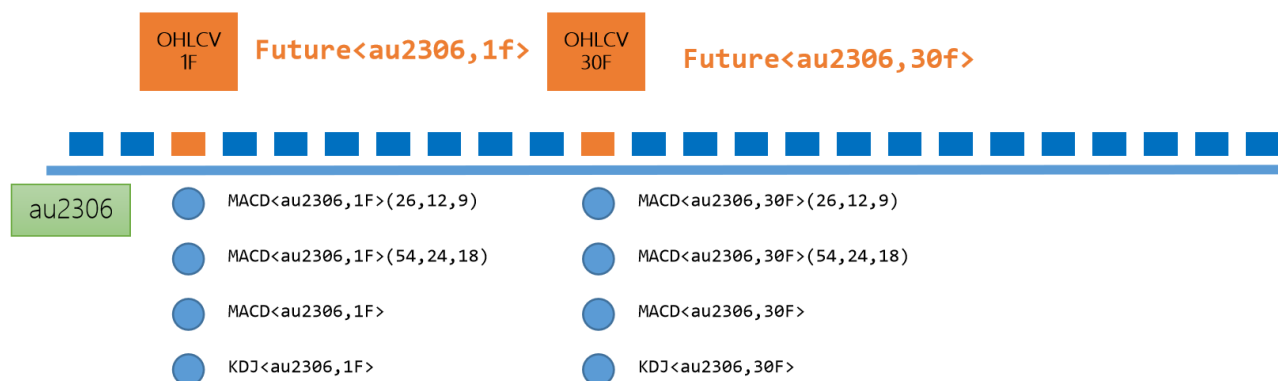
base/base.h 定义了 struct Unit

```

1 struct Unit {
2     char instrumentID[81];
3
4     // 用于是否是第一次获取数据, 如果是断线连, 则需要重新设定 start_day finish_day s
    tart_time finish_time ....
5     bool is_first_get{true};
6     char start_day[9];
7     char finish_day[9];
8     char start_time[9];
9     char finish_time[9];
10
11     int ohlcv_len;
12     int index;
13     int future_index; // Future<M,N> 的 M => 数组下标
14     int period_value; // 用秒计的period
15     int period_index; // Future<M,N> 的 N => 数组下标
16     char period_name[16];
17     double* O;
18     double* H;
19     double* L;
20     double* C;
21     double* V;
22     double* out; // 初始化时要和OHLCV一样分配空间, 用于talib计算时, 做为出参。
23                 // 由于 new_talib.h中的D_EMA()函数的调用方式是 FutureSeries<
    8,4> x =
24                 // D_EMA(K3,100); 这里的DEMA()返回值是double, 它没有办法使
    用 x序列的 data[]作为
25                 // talib中的 SEE_EMA()的 out[]入参。 所以需要在这里先申请一个
    double数组。
26                 // 在onBar里的计算都是串行的, 所以, 所有的D_MA, D_EMA, D_KDJ可
    以用它。
27                 // 没有并行计算的要求, 所以整个 Future<8,4> 均使用这一个out
28
29     // todo 按照 <M,N>对 来说的话, 这里只需要一个 Signal<void()>就可以了,
30     // 因为它可以连接所有与M这个合约N这个周期的所有 function。
31     // 一个 sig 可以 connect 多个functions。
32     // sig(); 这样调用, 与之相联的所有functions都会执行一遍。
33     // Signal<void()> future_fun_list[1]; // 这里初始化了50个 Signal<void()
    >, 每个都可以connect
34     // 多个function。
35     Signal<void()> func_signal;
36 };

```





```

// -- 初始化 BaBo[50] ; BarBlock -----
/*
// 不同的周期，相当于不同的 frequency : fr      fr= 1 表示周期为1秒      fr= 60 表示周期为60秒
// 默认周期有以下30个：
// 0  1  2  3  4  5  6  7  8  9  10 11 12 13  14  15  16  17 18 19 20 21 22 23 24  25  26 27 28 29 30
// T   1S 2S 3S 5S 10S 15S 20S 30S 1F 2F 3F 5F 10F 15F 20F 30F 1H 2H 3H 4H 5H 6H 8H 10H 12H 1D 1W 1M 1J 1Y
//
// 用户自定义周期，可以有19个。
// pBaBo[50] : pBaBo[0-30] pBaBo[30-49]
*/
// 默认的周期
// 初始化----- 每个 FuBo
// --> future block 有 50个不同的周期 每个周期一个 BaBo ( bar block )

```

程序为每个future定义了一个BarBlock的指针数组，pBaBo[50]： pBaBo[0-30] 固定的，pBaBo[30-49] 用户可以自己定义。

## 2. 【策略代码规范】

### 2.1. 策略程序逻辑

1. 初始化一个 `base` 对象。
2. `base.FutureInit()`，为每个合约每个周期初始化一个 `unit`，分配相应的 `OHLCV` 空间。
3. 每个合约每个周期调用 `Future<M,N> x(&bb);` 绑定 `base` 对象中相应的 `unit`。
4. 编码切版代码 `Onbar()`，`Ontick()`。
5. `base.run()`进入循环从服务器获到数据。

#### 2.1.1. 定义策略所需合约及周期 table.json

定义一个交易策略所需的 json文件，需要包含合约名及周期

```

1  {
2      "-----": "define begin",
3      "###_trading_mode": "BACK_TEST, REAL_CTP, SIMULATION ",
4      "###_period0": " period_TK 0",
5      "###_period1": " period_1S 1",
6      "###_period2": " period_2S 2",
7      "###_period3": " period_3S 3",
8      "###_period4": " period_5S 4",
9      "###_period5": " period_10S 5",
10     "###_period6": " period_15S 6",
11     "###_period7": " period_20S 7",
12     "###_period8": " period_30S 8",

```

### 2.1.2. base对象初始化

`bb.FutureInit` 会为每个合约的每个周期 分配 OHLCV空间。

初始化了N个unit, 每个unit对应一个OHLCV, 每个合约的每个周期, 对应一个unit.

```

1  static Base bb;
2  bb.FutureInit("./tbl/trade.json", LOCAL_DATA);

```

### 2.1.3. 每个合约每个周期绑定base中的相对应的unit

```

1  static Future<8, period_1F> x(&bb); // Future<8, 9> x(&bb);
2  static Future<8, period_5F> y(&bb); // Future<8, 12> y(&bb);
3  static Future<3, period_1F> x(&bb); // Future<8, 9> x(&bb);
4  static Future<3, period_5F> y(&bb); // Future<8, 12> y(&bb);

```

## 2.2. 策略代码规范

- `onBar()` 函数是切片运行, 所以在里面定义的对象, 绝大部情况必须是静态变量, 只初始化一次。
- 所有 `EMA_T`、`RSV_T` 这种类型的构造函数, 目前均不支持copy构造等。
- 所有 `EMA_T`、`RSV_T` 这种类型的对象, 可以进行赋值, 采用 `EMA_T<8, 9> XXX4 = EE1;`, 均是对其所包含的序列的当前 `data[index]` 进行赋值。或者先定义对象, 再赋值。 `EMA_T<8, 9> XXX4;` `EXXX4 = EE1;` `EXXX4 = 3.1415;`

- 这些序列均是 `double` 序列。
- 有两种方式对 `EMA_T`、`RSV_T` 这一类对象进行操作：

#### 第一种操作方式：

- 先申明一个空对象 `static EMA<8, 4> k5;`
- 显示调用 `k5.EMA(r3,12);`

#### 第二种方式：仅对 OHLCV有作用

对于OHLCV这些有基础数据的操作，可以使用以下方式：

- `static EMA<8,4> X5(OHLCV:C,34);`
- 这种方式，信号槽内置了EMA的计算函数，不需要再显示调用 `X5.ema();`

#### 第三种方式：【这一种方式已经弃用】

- 定义对象时，~~直接初始化 `EMA_T<8, 4> X2(dif2, 2);`。~~
- 再显示调用一次 ~~`X2.ema();`。~~

### 2.2.1. 标准序列的定义：

OK1

```
1      static FutureSerial<8, 9> TEST2;
2      TEST2 = 33;
3      TEST2 = 55.123;
4      TEST2 = TEST1;
5      TEST2 = TEST1 + 1;
6      TEST2 = TEST1 + 2.5;
7      TEST2 = TEST1 + TEST2;
8      TEST2 = (TEST1 + TEST2) / 2;
9      TEST2 = TEST2 / 2;
```

```

1  static FutureSerial<8, 5> dif55;           // 分配内存
2  static FutureSerial<8, 5> dif55();         // 分配内存
3  static FutureSerial<8, 5> dif55(5.0);      // 分配内存
4
5  static FutureSerial<8, 9> O('o');
6  static FutureSerial<8, 9> H('h');
7  static FutureSerial<8, 9> L('l');
8  static FutureSerial<8, 9> C('c');
9  static FutureSerial<8, 9> V('v');         // 目前还是double类型
10
11  dif55 = 3.0;
12

```

OK2

```

1      EMA_T<8, 9> EE1;
2      EMA_T<8, 9> XXX4 = EE1;
3      EMA_T<8, 9> XXX5 = 3.0;
4      EMA_T<8, 9> XXX6 = TEST2 + 3.0;
5
6      EMA_T<8, 9> d0tt0;
7      EMA_T<8, 9> d0tt1 = 3;
8      EMA_T<8, 9> d0tt2 = 3.0;
9      EMA_T<8, 9> d0tt3 = EE1;
10     EMA_T<8, 9> d0tt4 = TEST1 + TEST2;
11     EMA_T<8, 9> d0tt5 = TEST1 + TEST2 + 100.5;
12
13     RSV_T<8, 9> RR00;
14     EMA_T<8, 9> dtt1(3);
15     EMA_T<8, 9> dtt2(3.0);
16     EMA_T<8, 9> dtt3(EE1);
17     EMA_T<8, 9> dtt4(TEST1 + EE1);
18     EMA_T<8, 9> dtt5(TEST1 + EE1 + 100.5);
19
20     // EMA_T<8, 9> dtt3(RR00);
21
22     EMA_T<8, 9> d2tt0{};
23     EMA_T<8, 9> d2tt1{3};
24     EMA_T<8, 9> d2tt2{3.0};
25     EMA_T<8, 9> d2tt3{EE1};
26     EMA_T<8, 9> d2tt4{TEST1 + EE1};
27     EMA_T<8, 9> d2tt5{TEST1 + EE1 + 100.5};

```

### 2.2.2. 其它序列的定义：



```

1
2  EMA_T<8, 9> DEA3;
3  EMA_T<8, 9> DEA3();
4  EMA_T<8, 9> DEA3(5.0);
5
6  static FutureSerial<8, 9> 0('o');
7  EMA_T<8, 9> DEA3(0);
8
9  EMA_T<8, 9> DEA3();
10 EMA_T<8, 9> DEA4(DEA3);
11
12 static FutureSerial<8, 9> 0('o');
13 EMA_T<8, 12> EMA_812(C, 10);
14
15 EMA_T<8, 9> DEA3();
16 EMA_T<8, 12> EMA_812(DEA3, 10);
17

```

### 2.2.3. 不存在的搞法：

```

1      static FutureSerial<8, 5> XXX1{} = 3.0;
2      static FutureSerial<8, 9> XXX2{} = TEST2; // 不是c++规范 大括号后面应
   该是 ";"
3      static FutureSerial<8, 5> XXX1() = 3.0;
4      static FutureSerial<8, 9> XXX2() = TEST2; // 未能初始化

```

不同类型不能做参数

```

1      RSV_T<8, 9> RR00;
2
3      EMA_T<8, 9> ES9;
4      ES9.EMA(RR00, 12);    // OK
5
6      EMA_T<8, 9> ES91(RR00, 35);    // OK
7      ES91.EMA();
8
9      EMA_T<8, 4> ES4;
10     // ES4.EMA(RR00, 12);    // error
11
12     // EMA_T<8, 4> ES41(RR00, 35); // error
13     // ES41.EMA();
14
15     static FutureSerial<8, 9> XrX1;
16     XrX1 = ES4;    // error 重载=号 类型不同
17     XrX1 = ES9;    // ok 重载=号
18     XrX1 = ES4 + 0; // ok 这个相当于 是 XrX1 = 3.14 ; ES4 + 0 之后是一个double类型。

```

## 2.2.4. 错误做法

1. 在onBar内部对初始化的变量不能直接赋值： 以下这种写法只会运行一次，不会每次都运行。

```

1      static FutureSerial<8, 9> XXX1 = 3;
2      static FutureSerial<8, 9> XXX2 = 3.0;
3      static FutureSerial<8, 9> XXX3 = 3.15 + XXX2;

```

正确的写法是：一，将变量定义到onBar之外，二，先定义一个空变量，然后做运算：

【可以将上面的情况，用python转成下面的写法再编译】 todo!!!

```

1      static FutureSerial<8, 9> XXX1;
2      static FutureSerial<8, 9> XXX2;
3      static FutureSerial<8, 9> XXX3;
4      XXX1 = 3;
5      XXX2 = 3.0;
6      XXX3 = 3.15;

```

# 测试:

1. 这种不带参数的构造函数，会不会分配内存？两种写法：`EMA_T<8, 9> DEA3();` `EMA_T<8, 9> DEA3;`

```
1  EMA_t() {
2      std::cout << "B:constructor !! EMA() -----*****" <<
    std::endl;
3      // PeriodBase<N...>::UnitCheck();
4      std::cout << "E:constructor !! EMA() -----*****" <<
    std::endl;
5  };
```

### 2.2.5. `class EMA_t` 解析

`class EMA_t` 因为继承了 `class FutureSerial` 所以在定义时，会分配内存。

序列对象不可直接赋值给double。

```
static FutureSerial<8, 5> dif55;
```

```
double d = dif55; // 不可以。
```

```
d = dif55 + 0; // 可以。
```

double可以赋值给序列对象。

```
double d = 101
```

```
static FutureSerial<8, 5> dif55;
```

```
dif55 = d; // 可以。
```

### 2.2.6. 【第一类写法】

定义一个静态对象。 `static EMA<8, 4> r5();`

用对象方法调用 `r5.EMA(rsv5, 55);` ，加入参。

这种写法， `EMA()` 的调用是在切片 `onBar([&](){...})` 内部，执行顺序可以保证。

写法一：EMA\_T 即：#define EMA\_T static EMA\_t

```
1
2     #define EMA_T static EMA_t // 在别的.h文件中定义
3
4     onBar([]){
5         EMA_T<8, 4> r5();    // 定义 r5 为static
6         r5.EMA(rsv5, 55);    // r5:=EMA(rsv5,55);
7     };
8
```

写法二：和写法一实际上一样。

```
1     onBar([]){
2         static EMA_t<8, 4> r5();    // 定义 r5 为static
3         r5.EMA(rsv5, 55);    // r5:=EMA(rsv5,55);
4     };
5
```

写法三：【推荐写法】

- 先申明一个空对象 static EMA<8, 4> k5;
- 显示调用 k5.EMA(r3,12);

把r5的申明，放在onBar外面。放在里面是不是也没有问题？要测试！！

r5的成员函数 r5.EMA(rsv5, 55) 调用，加参数。这种方式，EMA 可以直接存取 r5 的 data 。

static SMA<8, 4> k52; // 1 static EMA<8, 4> d52(); // 2 这两种写法一样吗？

```

1 // static EMA<8, 4> r5(); // 定义 r5 为static
2 static EMA<8, 4> r5; // 定义 r5 为static
3 static RSV<8, 4> rsv5;
4 static EMA<8, 4> r5;
5 static SMA<8, 4> k5;
6 static EMA<8, 4> d5;
7 static SMA<8, 4> k52; // 1 这两种写法一样吗?
8 static EMA<8, 4> d52(); // 2
9 static SMA<8, 4> k53();
10 static EMA<8, 4> d53();
11 onBar([&](){
12     // KDJ 5
13     rsv5.RSV(A5);
14     r5.EMA(rsv5, 55); // r5:=EMA(rsv5,55);
15     k5.SMA(r5, B5, 1);
16     d5.EMA(k5, B5 / 2);
17     k52.SMA(r5, B4 * 2, 1);
18     d52.EMA(k52, B4);
19     k53.SMA(r5, B4, 1);
20     d53.EMA(k53, B4 / 2);
21 });

```

### 2.2.7. 【废弃】【第二类写法】针对入参有值的情况，比如 用OHLCV作为入参

- 这种写法最好只针对 OHLCV ，其它的序列最好不要用。【不能用】
- 目前打算针对非OHLCV的使用，要废弃信号槽？

定义变量时，加入参 `static EMA<8, 4> r5(rsv5, 55);` 。

这种写法，是使用**信号槽**，将ema()的执行放在后台，先运行。然后再执行 onBar()切片。

这种写法，要求参数里的值必须先存在。如果这个参数值是必须在切片函数onBar()中运行才能得到，那将会造成后台执行ema()时，入参值不存在的情况。

场景是，针对只有OHLCV 作为入参的情形，可以采用这种写法。因为后台 运行这些ema(), kdj(), 时 ohlcv是已经有了数据。

这种写法可不可以编译期，提示一下？至少运行时提示一下？

写法一：

```

1  #define EMA_T static EMA_t // 在别的.h文件中定义
2
3  onBar([]){
4      // KDJ 5
5      RSV_T<8, 4> rsv5(A5);
6      EMA_T<8, 4> r5(rsv5, 55); // 定义 r5 为static
7      SMA_T<8, 4> k5(r5, B5, 1);
8      EMA_T<8, 4> d5(k5, B5 / 2);
9      SMA_T<8, 4> k52(r5, B4 * 2, 1);
10     EMA_T<8, 4> d52(k52, B4);
11     SMA_T<8, 4> k53(r5, B4, 1);
12     EMA_T<8, 4> d53(k53, B4 / 2);
13 };

```

本质上是：

```

1  onBar([]){
2      // KDJ 5
3      static RSV_t<8, 4> rsv5(A5);
4      static EMA_t<8, 4> r5(rsv5, 55); // 定义 r5 为static
5      static SMA_t<8, 4> k5(r5, B5, 1);
6      static EMA_t<8, 4> d5(k5, B5 / 2);
7      static SMA_t<8, 4> k52(r5, B4 * 2, 1);
8      static EMA_t<8, 4> d52(k52, B4);
9      static SMA_t<8, 4> k53(r5, B4, 1);
10     static EMA_t<8, 4> d53(k53, B4 / 2);
11 };

```

## 2.2.8. 【第三类写法】显式调用一次【可以使用，必须使用显示调用】

- 定义对象时，直接初始化 `EMA_T<8, 4> X2(dif2, 2);`。
- 再显示调用一次 `X2.ema();`。

是针对第二类写法的补充。

入参是在onBar()中运行产生，需要显示调用 `X2.ema();`

如下面的 `dif2` ,必须在onBar中运行了 `dif2 = ema_s2 - ema_l2;` 以后才能有值。

但是 `EMA_T<8, 4> X2(dif2, 2);` 这是对X2的定义，所以在 `onBar` 中不会被执行，在 `X2` 的构造函数中，将其挂接到信号槽中了，这个信号是在 `ohlcv` 数据产生后，由系统先行调用，然后再执行 `onBar` 函数。但在执行槽函数时，`dif2` 的值还没有计算出来，因为 `dif2` 的值必须在 `onBar` 中产生。

所以在后面的 `onBar()` 切片中，还必须再等 `dif2` 的值出来后，再显式地调用一次：`X2.ema();`

注意：要解决后台隐式调用无效的问题。如何禁后台调用。

```
1  onBar([]){
2      dif2 = ema_s2 - ema_l2; // 这个在前!!!
3      EMA_T<8, 4> X2(dif2, 2); // 这个在后!!!
4      double kx = X2.cur();
5      double ky = D_EMA(dif55, 12) + 200000;
6      EMA<8, 4> DEA2(X2, M2);
7  };
```

【目前能想到的解决方案：是用python将上述代码生成为：】 `todo!!`

```
1  onBar([]){
2      dif2 = ema_s2 - ema_l2;
3
4      EMA_T<8, 4> X2(dif2, 2);
5      X2.ema(); // python 自动生成。
6
7      double kx = X2.cur();
8      double ky = D_EMA(dif55, 12) + 200000;
9
10     EMA<8, 4> DEA2(X2, M2);
11     DEA2.ema(); // python 自动生成。
12 };
```

## 第四类写法（废弃）

于2023年11月14日废弃

废弃的原因有两点：

1. 输出要用到外部的`out[]`数组。
2. 类型会混用？ 会不会有这种需求？

采用直接函数调用。

函数返回的值是 `double`。

序列接受函数的返回值 `double`，并将其填写回其 `data[this->unit->index]` 即数组最当前 `index`。

这类写法，最大的问题是：如下面一句 `dif33 = D_EMA(ema_s2, 10);`，在 `D_EMA` 内部调用 `SEE_EMA()` 的入参 `out[]` 无法传递进去，因为 `out[]` 其实应该是 `dif33` 本身的 `data[]`。它只能根据返回值来回填。

需要额外一个临时的数组作为 `D_EMA` 内部调用 `SEE_EMA()` 的入参 `out[]`。

但这种写法有个好处是，可以在不同类型之间调用：比如 `dif44 = D_EMA(ema_855, 10);` `dif44` 是 `<8,4>` 类型，入参 `ema_855` 可以是 `<8,5>` 类型。但也容易造成错误，不同类型是不能来回窜，没有这种场景。

```
C++ |
1      static FutureSerial<8, 4> dif22;
2      static FutureSerial<8, 4> dif33;
3      EMA<8, 5> ema_855(ohlcv::C, 10);
4      static FutureSerial<8, 4> dif44;
5      static FutureSerial<8, 5> dif55;
6
7      dif22 = ema_s2 - ema_l2;
8      dif33 = D_EMA(ema_s2, 10);
9      dif44 = D_EMA(ema_855, 10);
10     dif55 = D_EMA(dif44, 10);
11     dif55 = D_EMA(0, 10);
```

## 定义OHLCV（对象）

OHLCV在每个序列中均存在。比如 `dif22.unit->0;`

但下面可以更加直观地定义。供后面直接使用。

```
C++ |
1      FutureSerial<8, 4> O('o');
2      // FutureSerial<> O('o');
3      FutureSerial<8, 4> H('h');
4      FutureSerial<8, 4> L('l');
5      FutureSerial<8, 4> C('c');
6      FutureSerial<8, 4> V('v');
```



## 定义double序列（对象）

```
1 static FutureSerial<8, 4> dif22;  
2 static FutureSerial<8, 4> dif33;  
3 static FutureSerial<8, 4> dif44;  
4 static FutureSerial<8, 5> dif55;
```

## 定义 int 序列（对象）

暂无。

## 定义bool序列（对象）

定义链表

## 定义其它（对象）

主要采用先定义一个对象。 `static bars_count xx1;`

再调用这个对象的相应的方法。 `int xa1 = xx1.bars_count(b001);`

下面的

```
1 bool b001 = ema_s2 > ema_l2;  
2 static bars_count xx1;  
3 int xa1 = xx1.bars_count(b001);  
4 static bars_count xx2;  
5 int xa2 = xx2.bars_count(!b001);  
6 int xa2 = xx2.bars_count(ema_s2 > ema_l2); // 可以  
7 int xa2 = xx2.bars_count(!(ema_s2 > ema_l2));
```

```
int xa2 = xx2.bars_count(ema_s2 > ema_l2); // 可以。
```

## double序列对象的运算

所有 `double`序列 的运算（+ - \* / 等）后的结果，均为 `double`，

遵循结合律之类。

```
1      dif22 = ema_s2 - ema_l2;
2      double kx = X2.cur();
3      double ky = D_EMA(dif55, 12) + 200000;
4      std::cout << "kx --- X2:  " << kx << " " << ky << " " << X2.ref(0)
    << std::endl;
```

## 复杂的计算

1、对象自引用。

对象当前index的值，要根据它自身前面的值来确定。相当于MA， EMA 之类。比如：  
待定。

2、递归、循环等。

待定。

## 序列对象的其它数学运算

要重写相应的函数。

`MAX()` `MIN()` 用法：

```

1
2     EMA<8, 4> d53(k53, B4 / 2);
3     int t = std::max(double(3), 4.1);
4     double dbt = MIN(d53, double(1000));
5     printf("printf MIN MIN == : %g  \n", dbt);
6

```

`double dbt = MIN(d53, 1000);` 这种写法没有实现。还需要 `double(1000)` ;

## 初始化过程

// 以上代码，针对每一个品种的每个级别，做两件事情：

// 【一】：生成对应 `O,H,L,C,V` ==> 注意，这里应该是一个对象列表，用户会向里面注入

// MA5之类的托管数据。

// `serial_list[] = { O,H,L,C,V, MA5,MA30,RSI, .... }` 里面会包括用户注入的部分。

// 这里，`OHLCV` 会自动挂接到系统的内存，对于象 `MA5, MA30,` 这样的，如果系统有，则挂接，

// 否则可能需要用户自己维护内存。用户也可以自行维护 `OHLCV` 等所有数据。

// 【二】：发出订阅信息。

```

1  let x1("ru2301",1F);
2  let x2("ru2301",1S);
3  let x3("fu2305",1H);
4
5  // 以上代码，针对每一个品种级别的每个级别，做两件事情：
6  // 一：生成对应 O,H,L,C,V ==> 注意，这里应该是一个对象列表，用户可能会向里面注入MA5之
   // 类的托管数据。
7  //      serial_list[] = { O,H,L,C,V, MA5,MA30,RSI, .... } 里面会包括用户注入的
   // 部分。
8  //      这里，OHLCV会自动挂接到系统的内存，对于象MA5，MA30，这样的，如果系统有，则挂
   // 接，
9  //      否则可能需要用户自己维护内存。
10 // 二：发出订阅信息。
11
12 let m5 = MA<注入>(C.x1, 5);
13 let m30 = MA<注入>(C.x1, 30);
14
15

```

## 数据类型及运算（切片运行）

**序列类型**，与之相关联的信息有：**计数器**，**futureid**，**5F等级别**。当然还有一个**数组**。比如 **OHLCV**，**RSI**，**EMA** 的返回值等。其中，计数器应该是同一品种同一级别，计数器的值是相同的，也就是当前数据在数组中的位置，理论上是自增1的。

这个计数器，应该与 **对象列表**？相关联。对象列表：**【OHLCV,MA5, x1,x2, .....】** 针对一个品种，一个周期级别，有一个这样的对象列表。

**链表类型**：链表，用于记录 cross点这样的信息。即，在某位置出现了5日线上穿10日线。

[illegible]

所有有初始化序列，均只是初始化 序列的当前index。

所有的 = 后面, 均是 等于 当前index的 `this->data[this->unit->index]`

所有序列变量，如果进行运算，均是对当前的 `this->data[this->unit->index]` double进行运算，并返回相应的double。

## 存在的问题

如下：

```

1      DIF2 = ema_s2 - ema_l2; // 这个在
前!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
2
3      EMA<8, 4> X2(DIF2, 2); // 这个在
后!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
4      X2.ema();
5
6      EMA<8, 4> DEA2(X2, M2); // 这
7      DEA2.ema();

```

X2的 `ema()` 需要显示地写在这里。因为 它的参数 `Dif2` 的数据是在切片 `(onBar)` 中产生的新的。所以X2的 `ema()` 不能在后台运行。

主程序的设计是：新的 OHLCV 到来，后台运行 `kdj, macd, rsi ema` 等，然后切片运行 `onBar()`。但在切片中运行的顺序不能被打断。

上面的代码，如果不调用 `X2.ema()`，则不会产生新数据。因为后台运行运行 `X2.ema()` 时，DIF2 的数据还没有产生，因为切片还没有运行。

## 2.3. main中初始化 future

1. `future<M,N>` `M` 代表 `future`，`N` 代表 `period`，是 `period_index`。 `M,N` 都用作了数组的下标，所以 `M` 必须从0开始，顺序递增，不得有间隔。
  - // 【注意】 `vector` 的第一个值，必须是 `0,1,2,3,4,5...` 从0开始递增，不得有间隔。
  - // 即，每个 `future` 有一个 `index`，这个 `index` 是 `unit` 数组下标，所以：从0开始递增，不得有间隔。
2. 有 `future<> fu(&base)`；【只有一个 `future`，只有一个 `period_index`】
3. 有 `future<period_1S> fu(&base); future<period_5S> fu(&base);`；【只有一个 `future`，有多个 `period_index`】
4. `future<future_index, period_index>` 【多个 `future`，多个 `period_index`】

```

1
2    // 【注意】 vector的第一个值，必须是 0,1,2,3,4,5... 从0开始递增，不得有间隔。
3    // 即，每个future有一个index，这个index是unit数组下标，所以：从0开始递增，不得
    有间隔。
4    #define xu1801 0
5    #define tu1801 1
6    #define ru1805 2
7    #define ru1809 3
8    #define xu1807 4
9    #define zz1805 5
10   #define ag1912 6
11   #define fu2305 7
12   #define au2306 8
13   #define LOCAL_DATA false // 是否使用本地数据
14
15   // 只有一个品种一个周期
16   // 可以写成 Future<> f(&base);
17   //      MACD<> ;
18   //      KDJ<> ;
19   std::map<std::string, std::vector<int>> future_map0;
20   future_map0["xu1801"] = {xu1801, period_1F};
21
22   // 只有一个品种多个周期
23   // 可以写成 Future<10> f1(&base);
24   //      Future<20> f2(&base);
25   //      MACD<10> ;
26   //      KDJ<10> ;
27   //      MACD<20> ;
28   //      KDJ<20> ;
29   std::map<std::string, std::vector<int>> future_map1;
30   future_map0["xu1801"] = {xu1801, period_1F, period_10F};
31
32   // 多个品种，多个周期
33   // 可以写成 Future<0,10> f0_10(&base);
34   // 可以写成 Future<0,20> f0_20(&base);
35   // 可以写成 Future<1,10> f1_10(&base);
36   // 可以写成 Future<1,20> f1_20(&base);
37   //      MACD<0,10> ;
38   //      KDJ<0,10> ;
39   //      MACD<0,20> ;
40   //      KDJ<0,20> ;
41   //      MACD<1,10> ;
42   //      KDJ<1,10> ;
43   //      MACD<1,20> ;
44   //      KDJ<1,20> ;

```

```

45
46 // 【注意】 vector的第一个值，必须是 0,1,2,3,4,5... 从0开始递增，不得有间隔。
47 // 即，每个future有一个index，这个index是unit数组下标，所以：从0开始递增，不得
48 有间隔。
49 std::map<std::string, std::vector<int>> future_map2;
50 future_map2["xu1801"] = {xu1801, period_TK, period_1S, period_1F, peri
51 od_2H};
52 future_map2["tu1801"] = {tu1801, period_TK, period_30F, period_2H};
53 future_map2["ru1805"] = {ru1805, period_1S, period_5F, period_2H};
54 future_map2["ru1809"] = {3, period_TK, period_15S, period_1F, period_5
55 F};
56 future_map2["xu1807"] = {4, period_TK, period_1S, period_1F, period_2H
57 };
58 future_map2["zz1805"] = {5, period_TK, period_1S, period_1F, period_2H
59 };
60 future_map2["ag1912"] = {6, period_TK, period_1S, period_1F, period_2H
61 };
62 future_map2["fu2305"] = {7, period_TK, period_1S, period_1F, period_2H
63 };
64 future_map2["au2306"] = {8, period_TK, period_1S, period_1F, period_2H
65 , period_5S};
66
67 static Base bb;
68 bb.FutureInit(future_map2, LOCAL_DATA);

```

## 写法注意事项

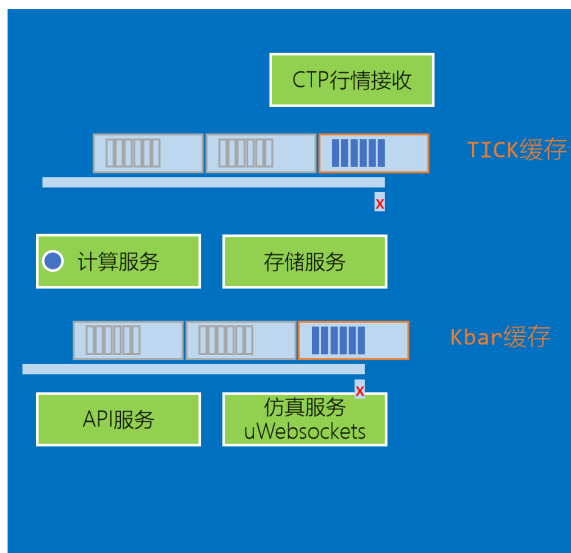


```

1  /*
2  // 要求:
3  // 1、 静态变量最好定义在onBar()外面。
4  //      包括 FutureSerial<m,n> x;
5  // 2、 在onBar()里, 不能 FutureSerial<m,n> x = 100; 要改成两句:
6  //      static FutureSerial<m,n> x;
7  //      x = 100;
8  // 3、 D_EMA(serial, n); 第一个参数必须为 FutureSerial<m,n>类型。
9  //      【如何实现 y = D_EMA(abs(close - close.ref(1), n) ; 】
10 //      以上目前只支持:
11 //      static FutureSerial x;
12 //      static FutureSerial y;
13 //      x = x.close - x.close.ref(1) ;
14 //      y = D_EMA(x, n);
15 //
16 //
17 */

```

## 服务进程



服务进程：指以上蓝色框里的服务。

CTP接收的数据，存放在“缓存”区。缓存区的长度为1024，如果存储满了，则再从0开始存储，类似于环形队列。

序号X，每天开机从0开始，CTP接收一条TICK，则自增1，X会mod 1024，当存储服务线程读到X=1024时，会再从“缓存”0开始取数据。当X=2048时，又会再次从“缓存”0开始取数据。

K线计算线程，也同样读取这个X值，做类似的处理。

策略进程：

- 1、向仿真服务器发出数据请求，from 20180808, to today。
- 2、仿真服务器，发送历史数据。
- 3、仿真服务器，追赶X值，发送TICK，Kbar实时数据。

策略进程和服务进程断线：

当策略进程断线重连上，后，首先发出新的数据请求，前面已经有的数据，可以不用再发请求了，从丢失的数据开始请求。仿真服务器重新追赶X值。

服务进程出了问题重启之后：

- 1、策略进程重新连接
- 2、再次发出数据请求，前面已经有的数据，可以不用再发请求了，从丢失的数据开始请求。
- 3、仿真服务器重新追赶X值。【服务进程要保证数据的完整性】。

服务进程，可以部署独立两套，或多套。1、从不同的期货公司接口接入，2、不部署在不同的云服务器上。

针对某个合约，设置单独的队列，动态从CTP或者其它进程获得tick或者是kbar，并记录写入序号（每天或一定的时间，重新设置），队列的长度要保证当天的消息全部能够保存。

当有策略要使用这个合约时，可以根据情况，让仿真服务从某个序号开始，从这个合约的队列读取数据传送给这个策略，然后这个序号一直自增，直到追上，上面队列的写序号。

策略进程通过 `getData("1998-08-08","today");` `getData("1998-08-08","today-10:45:37");` 这种调用，告知 仿真服务进程，从哪个序号开始。仿真服务根据 `("1998-08-08","today-10:45:37")` 计算出当天这个时间点的序号。

如果有多个级别，怎么办？ 仿真服务，只传送某一个级别的数据，策略服务进程，自己合成？

## 数据级别时间同步策略

### 一：只提供最小级别，由策略自行合并成其它级别

- 如果一个合约需要多个级别的数据，只传输最小级别的数据，由策略自己合成其它级别。
- 如果有多个合约，统一按最小级别传输，并进行同步。
- 策略需要的最小级别数据如果服务器不提供的话，可以请求更小级别的数据，进行合成。
- 要求：策略需要的最小级别，必须比服务器能够提供的最小级别大。

### 二：由服务器提供不同级别的数据，服务器进行时间同步。

## 增加