

xquant 策略开发API文档

1. 序列index

2. BoolsegsList类

3. StopsegsList类

3.1. 新增加 set_segflag(100)

4. sections类设计

4.1. sections类的源码解析

4.2. 最终确定版本： 版本二，不记录默认section。【21年12月24日】。

4.3. 两个版本

4.3.1. 版本一：记录默认的section为INT_MIN类型。【系统不再有这个版本了!!! 原因如下】

4.3.2. 版本二：不记录默认的section【最终定型版本】

4.4. 2023-12-18修正

4.5. 2023-12-18修正

4.6. 2023-12-16修正

4.6.1. onbar中的处理

4.6.2. 后置update中的处理

4.7. 前期设计

5. double序列在 ref() 参数异常时，返回当前值。

1. 序列index

任何序列 有以下值：

S1.x == 100 ? .x表示index值，从0开始，最大为10万。

2. BoolsegsList类

- BoolsegsList和StopsegsList是姊妹类。
- **BoolsegsList** 会对没有赋值的 onbar ，自动**延续**。
- **StopsegsList** 会对没有赋值的 onbar ，自动**停止**。
- 对于类似下面的代码：如果是 **BoolsegsList** 类，如果在某个onbar条件不成立时，则不会执行

H_KPB3 = true; 也没有重新将其赋值为false。但BoolsegsList会自动为H_KPB3延续记录值为true的段。即后面只要不明确 执行 H_KPB3 = false; 系统将自动延续 H_KPB3 一直为 true 。

```
1      if (K33 > K3 || K32 > K3 || X3 > REF(X3, 1)) {
2          H_KPB3 = true;
3      }
4      if (K33 < K3 || K32 < K3 || X3 < REF(X3, 1)) {
5          H_KPS3 = true;
6      }
7      if (H_KPB3 && H_KPS3) {
8          H_KPP3 = true;
9      };
```

- 上面的代码，如果是 **BoolsegsList**，如果某个onbar条件不成立时，则不会执行 H_KPB3 = true; 也没有重新将其赋值为false，从这个onbar开始，系统将自动将 H_KPB3 变为 false，系统将自动终止为H_KPB3记录的值为true的段，并为H_KPB3 重新记录一个为false的默认段，这个默认段不能设置flag。

BoolsegsList 之所以设计成自动延续前值的原因，是因为正常情况，在前面onbar对某对象赋值之后，应该在后面的onbar中，这个值是保留原值，不应该被改变的。如同普通的bool，int， double之类一样。

StopsegsList 为什么为设计成不再赋值后，自动终止，设置成反向值，是因为在实际使用中，程序员可能只想记录某种情况，不需要考虑其它的情况。一旦不再赋值，即表示这种要记录的段的结束。万一有人写成了上面的代码呢， H_KPB3 岂不永无为 true 了？虽然可能性很小。

如果双向都记录，则两个类的使用没有区别，如下代码。这种写法，H_KPB4不是true,就是false，在每个onbar中都会明确赋值。这种情况，使用 **BoolsegsList** **StopsegsList** 没有区别，但建议使用 **BoolsegsList**，因为细节上 BoolsegsList更简单，性能更好一点点。

```
1      H_KPB4 = (K43 > K4 || K42 > K4 || X4 > REF(X4, 1));
2      H_KPS4 = (K43 < K4 || K42 < K4 || X4 < REF(X4, 1));
3      H_KPP4 = H_KPB4 && H_KPS4;
```

两个类中，都会有 名为 segment_ 的 vector。它的很多更新工作是在 onbar 结束后，由系统调用 update() 完成的，对于策略程序，是在 onbar 中执行，所以在 onbar 中，不要直接读取系统

记录的 `segment_` 这个 `vector`，它的内容很可能不是实时的，并不及时，因此，如果要使用 `segment` 相关的内容，请使用系统函数和方法，大部的方法，是以 `ref_` 和 `cur_` 开头。

3. StopsegsList类

- StopsegsList类 用于定义一个 bool类型的对象。 `StopsegsList<8, 9> S1;`
- 这个bool对象，可以作为bool进行相应的 `&&` `||` `!` 等运算。也可以赋值给一个bool变量。也可以 if 条件判断。
- 在onbar中，对S1一直赋值true，将会记录一个value为true的段。
- 这个为true段，可以设置flag，用于对这个段进行标记。
- 当在某一个onbar中停止赋值时，这个为true的段结束。开始记录一个value为false的默认段。这个为false的默认段，不能设置flag。
- 反之，如果在onbar中，一直 `S1 = false;` 则会记录一个value为 false的段。这个段可以设置flag。
- 当在某个onbar（假设为 `t0`）停止赋值时，这个为false的段结束。开始记录一个value为true的默认段。这个为true的默认段，不能设置flag。
- 当这个为true的默认段在之后的某个onbar(假设为 `t1`)做了 `S1 = true;`的赋值操作，将这个默认段变成实际正常的段，从此时开始，可以标记flag了。这个段的起始值，是false结束的那一个onbar index,即 `t0` ,而不是 `t1`，而不是后面重新给 `S1= true;` 赋值的那个onbar的index `t1`。这个 `S1 = true;` 的赋值操作会带来另外一个作用，就是后继的每个onbar都要 `S1= true;` 否则一旦停止赋值true，这个为true的段，就结束了，一个新的为false的默认段开始了。见后面的方式一。
- 正常情况，是true false两者都要记录的。见方式二，这也是建议的方式。
- StopsegsList类是用来记录 bool段的类。它被设计成对称的：即，从某个onbar调用了 `set_segment()`开始，它就开始既记录true的段，又记录false的段。（对这个序列进行赋值 true or false，在内部就会调用 `set_segment()` 进行处理）。
- 当反向赋值时，前一个段会结束，新的段将开始。但系统会在onbar结束时才调用 `update`，将新的 `segment` push到vector中。
- 在使用时，序列必须明确赋值，如果在某些onbar中，没有对StopsegsList对象进行true 和false赋值，那么segment将结束，新的反向的段会自动开始记录。即：如果停止赋值，segment将结束，并反向做记录。
- 对于没有明确赋值的段，不能设置flag标记。
- 方式 一：只针对 true进行赋值。当不赋值时，自动变成false的默认段开始记录，但对默认的段不能设置flag。方式一从本质上来说，用户只需要为true的段，默认不赋值的部分表示用户并不需要。不提倡这种写法。

```

1      if (K33 > K3 || K32 > K3 || X3 > REF(X3, 1)) {
2          H_KPB3 = true;
3      }
4      if (K33 < K3 || K32 < K3 || X3 < REF(X3, 1)) {
5          H_KPS3 = true;
6      }
7      if (H_KPB3 && H_KPS3) {
8          H_KPP3 = true;
9      };

```

- **建议写法：** 方式二： 正反都赋值，都可以设置flag。表示用户正反段都需要使用。

```

1      H_KPB4 = (K43 > K4 || K42 > K4 || X4 > REF(X4, 1));
2      H_KPS4 = (K43 < K4 || K42 < K4 || X4 < REF(X4, 1));
3      H_KPP4 = H_KPB4 && H_KPS4;

```

- 如果只定义，在后面一直没有任何赋值操作，则不会记录任何vector。
- bool类型
- 包含一个bool类型序列
- 包含对上涨和下跌（true or false）段的记录的vector。

要求：~~每个onbar都必须给变量赋值，要么true,要么false。~~

前置：

```

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

```

定义：

```

1      StopsegsList<8, 9> S2;

```

暂时不支持 `StopsegsList<8, 9> S2 = true;`

不支持： `StopsegsList<8, 9> S2{false};`

使用：

```

1      S2 = true;
2      S2 = false;
3      S2 = S3 && S5;

```

API:

```
1 typedef struct BoolSeg {
2     bool torf;           // true or false; // 2023-12-09 增加。 用于记录此段是
    上涨还是下跌
3     int B;               // begin index; 开始index值
4     int E;               // end index; 结束index值
5     int len;             // 本 BoolSeg 长度
6     int flag;            // 标识
7     bool boolarray[8];  // 标识位
8     // int distance;     // 距离前一个 BoolSeg的长度。 2023-12-09: 不用了, 因为
    同时记录上涨段和下跌段, 所以: 距离前一个上涨段的长度, 实际上是前面下跌段的长度
9 } BoolSeg;
```

```
1     int tofalse = S2.tofalse_last(); // 最近一次由true to false时到现在的
    周期数。
2     int totrue = S2.tottrue_last(); // 最近一次由false to true时到现在的周
    期数。 从0开始计算, 例如: ref() = false, this->value_ = true, 则 ftot_last
    () = 0 ;
3
4     // 是否改成设置为 int型?
5     S2.set_segmark(5,true);
6     S2.set_segmark(1,true);
7     bool mark5 = S2.get_segmark(5); // 针对当前段的标志 0-7号, 此例为读
    取 第5号 标志
8     bool mark1 = S2.get_segmark(1);
9
10    bool ref4 = S2.ref(4); // bool序列 ref(4)
11    bool cur = S2.cur();   // bool序列 当前值
12
13    // 对segment_的vector的操作。// BoolSeg 类型
14    int size = S2.segments_.size();
15    int B = S2.segments_.back().B;
16    int E = S2.segments_.back().E;
17    bool torf = S2.segments_.back().torf;
18    int len = S2.segments_.back().len;
```

3.1. 新增加 `set_segflag(100)`

针对当前整个段的标识设置。

`H_KPP3.set_segflag(100);` 设置一个整形值, 不能用 `INT_MIN` 。

`H_KPP3.get_segflag();`

- 这个设置，对整个segment都有效。即：在本segment内，如果在当前切片中，设置了这个flag，则在这个segment后面所有的切片中，不需要再设定，并且有效。
- 对于segment而言，每次的赋值为true或false，会决定是在同一个segment还是再新生成一个segment，取决于所赋的值与前一个切片中所赋的值是否相同，如果相同，比如都为true，则表示这个segment在延续。如果不同，则新生成一个segment。
- 如果这个段是表示值为true的段（S1 = true;）除了记录为true的段，还记录反向段，实际这上这个类，也记录了为false的segment.
- 注意有set_section的区别。set_section默认不设置type的那些onbar，vector不作记录，也不能设置flag。

```

1  if (H_KPP3 && H_KPB3 and !(H_KPB3.ref(1))) {
2      // H_KPP3.set_segmark(1,true);    // 这个标志，对整个段起作用。
3      H_KPP3.set_segflag(100); // 这个标志，对整个段起作用。
4  }
5      // H_KB3 = (H_KPB3 && (!H_KPP3)) || (H_KPP3 && H_KPP3.get_segmark
(1));
6      H_KB3 = (H_KPB3 && (!H_KPP3)) || (H_KPP3 && (H_KPP3.get_segflag()
== 100));
7
8  if (H_KPP3 && H_KPS3 and !(H_KPS3.ref(1))) {
9      // H_KPP3.set_segmark(2,true);
10     H_KPP3.set_segflag(200);
11 }
12     // H_KS3 = (H_KPS3 && (!H_KPP3)) || (H_KPP3 && H_KPP3.get_segmark
(2));
13     H_KS3 = (H_KPS3 && (!H_KPP3)) || (H_KPP3 && (H_KPP3.get_segflag()
== 200));

```

4. sections类设计

4.1. sections类的源码解析

section的整个设计，是根据 onbar 的切片执行原理进行。即每个index都会执行一次onbar。

section因此形成了两个循环处理：

- 一个是在同一个onbar内部，可能会多次调用set_section(type)，形成一个循环，可能反复设置成同一个type，或者设置成type a后又改成type b，最终又改回type a。在这个循环中，以最后一个type为准，如果某一次将type又设置回了前一个onbar同一个type，则要恢复其flag，因为表示这个

类型的section仍在延续，并没有改变成其它的type。但如果改成了新的type，则flag将设置成为 `INT_MIN`。

- 另一个是在多个 `onbar` 从 `t0` 一直到 `tn` 的切片执行过程中，type从某个a，切换成b，切换成c，d，又切换回a的大循环，在这个循环过程中，主要由update完成工作，如果type发生变化，update将会结束前一个section，重新push一个新的section。

`section` 的很多更新工作是在 `onbar` 结束后，由系统调用 `update()` 完成的，对于策略程序，是在onbar中执行，所以在 `onbar` 中，不要直接读取系统记录的 `section_` 这个 `vector`，它的内容很可能不是实时的，并不及时，因此，如果要使用section相关的内容，请使用系统函数和方法，大部的方法，是以 `ref_` 和 `cur_` 开头。

4.2. 最终确定版本： 版本二，不记录默认section。【21年12月24日】。

4.3. 两个版本

2021-12-21 采用版本二：不记录默认section。原因是不太容易处理默认段的flag。即，如果某种段有用，在程序中就必须为其设置一个type。

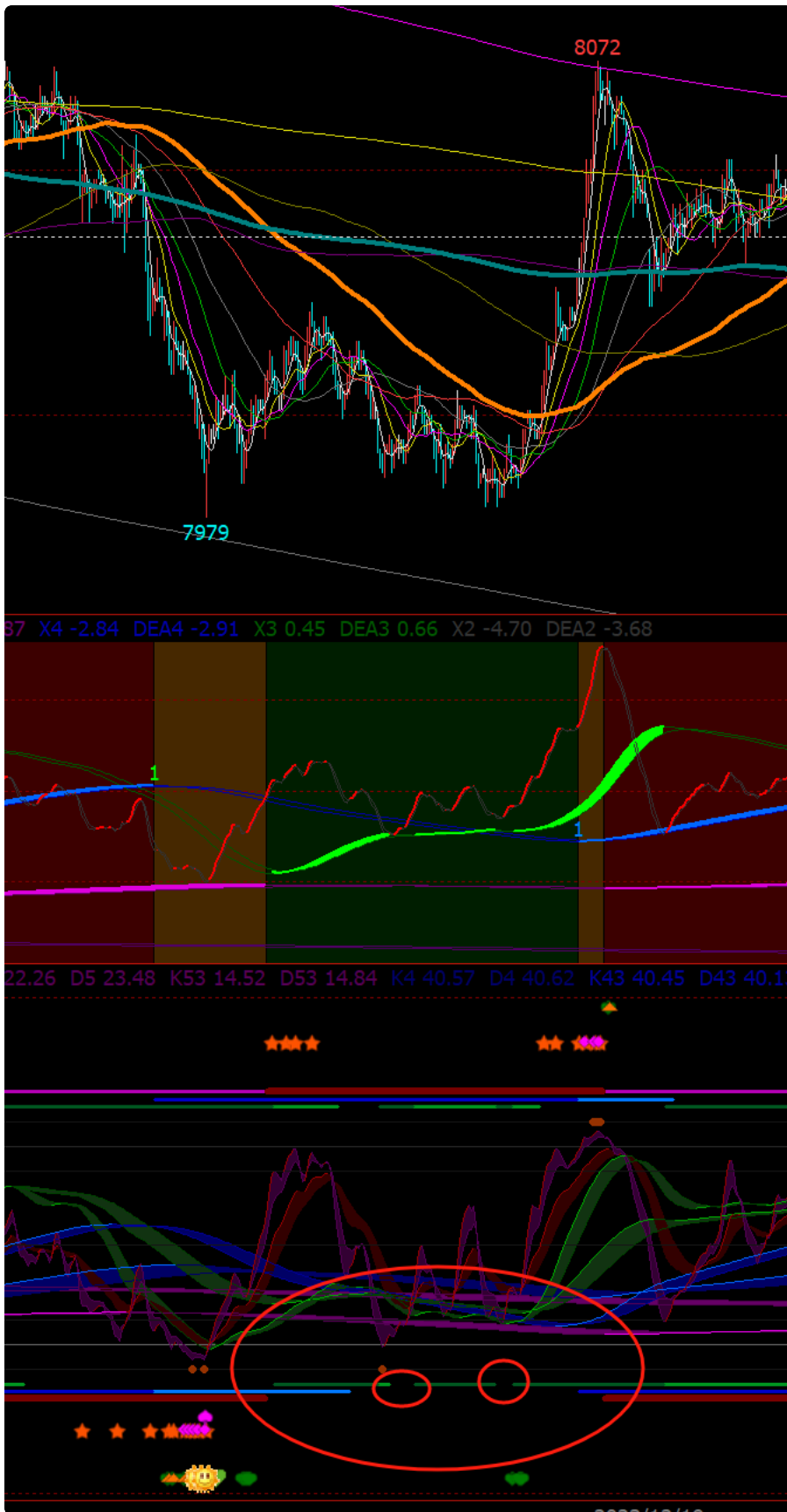
- 在连续的onbar调用过程中，一直设置 `set_action(type a)`。这个type为a的section就会一直增长。直到某一时刻的onbar不再 `set_action(type a)`了。这时，有两种情况，一种是因为重新设置type为b了，这种情况下，为b的section将开始记录。还有一种情况是不再调用 `set_action()`，不再设置任何type了，这时，不再记录任何section了，不能再设置flag了。
- 对某个序列设置section，要连续设置，这个section的每个onbar都要设置。
- section type 的设置，必须在每个onbar都设置，一旦不设置，将会被结束掉。
- 系统不记录默认section段。根本不会记录到vector中。和segment_不同。
- 消除段的设置，采用直接设置为 `set_section(INT_MIN)`;
- 为某一个段设置flag，不必在每个onbar都设置。
- 一个新的段，如果不设置flag，它的flag的值一直是INT_MIN。
- 不要对一个默认段设置flag（已改成，设置无效）
- 在一个onbar中，如果在没有确定这个onbar在哪个段之前，使用flag是不合理的。也就是说，在使用flag前，必须为这个序列在这个onbar时刻设置成为某个type的section。然后再使用这个section flag才有意义。比如：有一种情况，是用section记录 MACD的0轴之上的上涨的段，如果这个段的长度小于一个值，比如是60，就设置成flag 1，如果大于某个长度，就设置成flag 2。这种情况下，要先设置这段 `S2.set_sction(100)`; 然后再判断是否大于60，再为其设置 flag。不能在设置section的情况，设置flag。

- 在针对 section 使用flag之前，必须要设置 section的type，否则，不管是set_section_flag 还是 cur_section_flag均无效。

4.3.1. 版本一：记录默认的section为INT_MIN类型。【系统不再有这个版本了!!! 原因如下】

2021-12-24 下图中如果想要处理红圈中的空白的情况，如果不采用版本一，而采用版本二，只需要再增加一个新的type，来专门记录这个空白段即可，没必要使用默认section。所以也完全可以不需要版本一。

2023-12-19 采用版本一的原因是，策略有可能会根据是否为默认section来做事情。比如当上涨过程中，出现默认section，表示这一段为空，是一种极端行情。如下图，在绿线之间，蓝线之间，红线之间，均出现空段。可以用section 类型为 INT_MIN 表示，并记录在 vector中。



future.h

```

1  inline void _update() { // 这个方法是在onbar结束后进行的，但在send之前调用。
2      if (is_set == false) {
3          if (cur_section_.type == INT_MIN) {
4              return;
5          } else {
6              cur_section_.type = INT_MIN;
7              cur_section_.flag = INT_MIN;
8          }
9      }
10     if (cur_section_.type != ref_type_) {
11         if (idx >= 0) { // 最后才更新长度和结束 .E    // 改成这个判断 idx >
= 0 , 可以记录 type=INT_MIN.    // ref_type_ != INT_MIN
12             sections_[idx].E = this->x - 1;
13             sections_[idx].len = this->x - sections_[idx].B;
14         }
15         // if (cur_section_.type != INT_MIN) { // 去掉这个判断，就会记录 t
ype=INT_MIN.
16             cur_section_.B = this->x;
17             cur_section_.E = -1;
18             sections_.push_back(cur_section_);
19             idx++;
20             ref_type_ = cur_section_.type;
21             ref_flag_ = cur_section_.flag;
22         // }
23     } else {
24         if (ref_flag_ != cur_section_.flag) {
25             sections_[idx].flag = cur_section_.flag;
26             ref_flag_ = cur_section_.flag;
27         }
28     }
29     is_set = false;
30 }

```

base.cpp

```

1      // sections output to web -----
2      for (auto tp : this->unit_array_[8][9]->sectionslist_output) {
3          memset(temp, '\0', 1024);
4          std::string name;          // 1、名称
5          std::vector<Section> *sections; // 2、是一个vector<int> 里面存储是index位置信息
6          int color;                // 3、颜色 ??
7          auto tp2 = std::make_tuple(std::ref(name), std::ref(sections), std::ref(color)) = tp;
8          if (!sections->empty()) {
9              // if (sections->back().type != -1) {
10             // -1是自定义section_id的默认ID。不必上传
11             if (sections->back().type != INT_MIN) { // INT_MIN 是自定义section_id的默认ID。不必上传
12                 // if (sections->back().E < 0) {
13                 ret = sprintf(temp, 1024, "\'%s_section\':\'%d\'", //
14                     name.c_str(), //
15                     sections->back().type); //
16                 std::string temps(temp);
17                 happy = happy + temps;
18             }
19         }
20     }

```

4.3.2. 版本二：不记录默认的section【最终定型版本】

- 在连续的onbar调用过程中，一直设置 `set_action(type a)`。这个type为a的section就会一直增长。直到某一时刻的onbar不再 `set_action(type a)`了。这时，有两种情况，一种是因为重新设置type为b了，这种情况下，为b的section将开始记录。还有一种情况是不再调用 `set_action()`，不再设置任何type了，这时，不再记录任何section了，不能再设置flag了。

future.h不再push type为 INT_MIN的section。

```

1     private:
2         // 2020-12-18 多个不同的onbar执行, _update()和set_section() 一起形成一个大
        的循环。
3     inline void _update() { // 这个方法是在onbar结束后进行的, 但在send之前调用。
4         if (is_set == false) {
5             if (cur_section_.type == INT_MIN) {
6                 return;
7             } else {
8                 cur_section_.type = INT_MIN;
9                 cur_section_.flag = INT_MIN;
10            }
11        }
12        if (cur_section_.type != ref_type_) {
13            if (ref_type_ != INT_MIN) { // 最后才更新长度和结束 .E // 改成这
        个判断 idx >= 0 , 可以记录 type=INT_MIN. // ref_type_ != INT_MIN
14                sections_[idx].E = this->x - 1;
15                sections_[idx].len = this->x - sections_[idx].B;
16            }
17            if (cur_section_.type != INT_MIN) { // 去掉这个判断, 就会记录 typ
        e=INT_MIN.
18                cur_section_.B = this->x;
19                cur_section_.E = -1;
20                sections_.push_back(cur_section_);
21                idx++;
22                ref_type_ = cur_section_.type;
23                ref_flag_ = cur_section_.flag;
24            }
25        } else {
26            if (ref_flag_ != cur_section_.flag) {
27                sections_[idx].flag = cur_section_.flag;
28                ref_flag_ = cur_section_.flag;
29            }
30        }
31        is_set = false;
32    }

```

base.h 中, 因为每个 sections_里的每个section都是用户标记的section。当用户不再标记时, 最后一个section停留下来。因为不再记录默认section, 所以上传到web时, 要判断最后一个section是不是活动的section。采用的方式是判断最后一个section的 E 是否由-1更新了。 <0 表示没有更新, 即仍然是活动的section, 所以要上传。

```

1      // sections output to web -----
2      for (auto tp : this->unit_array_[8][9]->sectionslist_output) {
3          memset(temp, '\0', 1024);
4          std::string name;           // 1、名称
5          std::vector<Section> *sections; // 2、是一个vector<int> 里面存储是index位置信息
6          int color;                  // 3、颜色 ??
7          auto tp2 = std::make_tuple(std::ref(name), std::ref(sections), std::ref(color)) = tp;
8          if (!sections->empty()) {
9              // if (sections->back().type != -1) {
10             // -1是自定义section_id的默认ID。不必上传
11             // if (sections->back().type != INT_MIN) { // INT_MIN 是自定义section_id的默认ID。不必上传
12             if (sections->back().E < 0) {
13                 ret = snprintf(temp, 1024, "\\\"%s_section\\\":\\\"%d\\\"\"", //
14                                name.c_str(), //
15                                sections->back().type); //
16                 std::string temps(temp);
17                 happy = happy + temps;
18             }
19         }
20     }

```

4.4. 2023-12-18修正

- 不再记录默认段。
- 活动的section.E index一直为 `INT_MIN`，只到最后它结束时，修改成`this->x`，即当前onbar的index。

4.5. 2023-12-18修正

只有INT_MIN，没有 INT_MAX了。

sections被设计成bool序列和double序列的附属的段的记录者。

与StopsegsList的记录方法不同的是。section只是针对调用过的set_section()方法的段进行记录，一旦停止set_section()调用，则这个段自动结束，后继的段被记录成 type默认为 `INT_MIN` 的段，并且这个段设置的任何flag将视为无效，其flag永远为INT_MIN。

4.6. 2023-12-16修正

- `set_section(INT_MIN)`; 表示不再记录。从调用它的切片开始，不再记录，直到有新的`set_section()`调用为止。type 为 `INT_MIN`的section 不会记录 flag标志。

```
1      // set_section() 是幂等的，在同一个切片中，可以多次执行，结果相同。
2      // 在同一onbar中，可以多次执行 set_section
3      // type :      5      | 5      = 6      = 6      = 700      = 6      = 5
= 5      = MIN      = MIN = 5      |
4      // flag :      100     | 100     = MAX     = 105     = MAX     = MAX     = 100
= 109 = MIN      = MIN = 100     |
5      // set_flag:                                105                                109
          99
6      // 1. 设置了type后，flag要重置。
7      // 2. 如果 type 和 前一个bar相同，则flag与前一个flag相同，除非重新设置flag
8      // 3. 在同一个onbar中，当 type 一直变化时，每次变化会导致flag清除，清除的方式
      是：A、新type的flag为 INT_MAX，B、type和前一个onbar一样时，flag也和前一个flag一
      样。
9      // 4. 当在一个onbar中，type经过一些变化，稳定后，最好重置flag，否则，会因为
      上面第3条规则，导致flag异常。
```

以下设思路：

4.6.1. onbar中的处理

1. 在onbar过程中，只修改 `cur_section_` 的 type 和 flag，其它全部移到 后置 update 中处理。
2. 在onbar中，只有 type有变化时，才更新。
3. 更新的方式是：
 - a. 修改`cur_section_type`为当前设定的type.
 - b. 如果发现 type改回了和前一个bar的type(`ref_type_`) 一样，则需要把其flag也修改回来。
 - c. 如果是新的type，则flag同时设置为 `INT_MAX`.
 - d. 如果是 `INT_MIN`，则表明，从此onbar之后，不再记录section，flag设置为 `INT_MIN`.

有可能存在在同一个onbar中，出现多次调用`set_section`的情况，所以，有可能会重新设置回前一个onbar的type，这样，需要恢复其flag。对于新设定的type，意味着一个新的section开始，可以为其设置flag。但在同一个onbar中，再次设置一个更新的type，则前一个次新的type连同其flag全部消失，即使重新设置为前面次新的type，其flag也不会被恢复（被设置成了 `INT_MAX` ），所以要重新为其设置flag。

也就是说，flag是跟随type的。需要为新的type设置与其相应的flag。

```

1 void set_section(int type) {
2     if (cur_section.type != type) { // type有变化才更新 cur_section;
3         cur_section.type = type;
4         if (type == ref_type_) { // 如果 type和上一个onbar的type相同, 则f
lag 要改回。// 改回以前的flag ! 在 seg_section 中, 可以重设这个值。
5             cur_section.flag = ref_flag_;
6         } else if (type == INT_MIN) { // 表示不再使用
7             cur_section.flag = INT_MIN;
8         } else { // 其它所有type的更改, 都将flag初始化为 INT_MAX。
9             cur_section.flag = INT_MAX;
10        }
11    }
12 }

```

4.6.2. 后置update中的处理

1. 当onbar结束后, 调用update() 进行后置处理。
2. 当type为INT_MIN时, 初始化为开始状态。
3. 当type为新的类型时, 需要push_back一个新的Section。
4. 当type和上一个bar的type一样时, 更新它。
5. 之后, 备份当前type和flag (将ref_type设置为当前type。 ref_flag设置为当前flag.)
6. 之后, 如果type不是 INT_MIN , 将预设置下一个onbar的type为 INT_MAX 。

```

1  inline void _update() { // 这个方法是在onbar结束后进行的，但在send之前调用。
2      if (cur_section.type == INT_MIN) {
3          if (ref_type_ != INT_MIN) {
4              cur_section.type = INT_MIN;
5              cur_section.flag = INT_MIN;
6              ref_type_ = INT_MIN;
7              ref_flag_ = INT_MIN;
8          }
9      } else {
10         if (cur_section.type != ref_type_) {
11             cur_section.B = this->x;
12             cur_section.E = this->x;
13             sections_.push_back(cur_section);
14             idx++;
15             ref_type_ = cur_section.type;
16             ref_flag_ = cur_section.flag;
17         } else {
18             sections_[idx].E = this->x;
19             sections_[idx].len += 1;
20             sections_[idx].flag = cur_section.flag;
21             ref_flag_ = cur_section.flag;
22         }
23         cur_section.type = INT_MAX;
24         // cur_section.flag = INT_MAX;
25     }
26 }

```

4.7. 前期设计

- 和segment不同，section是针对所有序列的，是隶属于某个序列的，包括double序列，也包括bool序列，所以也可以为segment来生成section。
- 通过在这个序列的每个切片上 调用 set_section(int) 来产生section，原理相当于segment的赋值操作。比如：在每个切片中，一直set_section(a)时，这些连续的切片将成为一个类型为a的section。如果在接下来的某一切片中，set_section(b)，不同于a，则从这个切片开始，生成一个新的类型为b的section。
- 注意set_section() **必须保证在每个切片中都调用**，否则因为这个section因为没有连续赋值而中断，因为在没有进行人为调用set_section的切片中，系统会自动调用 set_section(-1),将这些切片设置为默认类型为-1的section。当然，如果只想记录部分的section，不想记录的部分，不必调用set_section()，系统自动把这个序列中，不需要的切片记录到 section(-1)中去。
- 比较有用的例子是，针对MACD黄白线序列，可以分成4种section，0轴之上上涨，为轴之上下跌，0轴之下上涨，0轴之下下跌。可以针对MACD序列，做如下section的记录：

- - `if(MACD>0 && MACD > MACD.ref(1)) { MACD.set_section(a); }`
- - `if(MACD>0 && MACD < MACD.ref(1)) { MACD.set_section(b); }`
- - `if(MACD<0 && MACD > MACD.ref(1)) { MACD.set_section(c); }`
- - `if(MACD<0 && MACD < MACD.ref(1)) { MACD.set_section(d); }`

- 一个只记录部分section的例子是：有一个segment用来记录250均线和89均线同时上涨的部分，针对这个segment，可以用section来记录 ma10线上涨的部分，其它的不用记录，系统将自动记录到 section(-1) 中。

- `StopsegsList<8,9> Seg89_250 ;`
- `Seg89_250 = Ma89 && Ma250;`
- `if(Seg89_250 && Ma10) { Seg_89_250.set_section(100);} // 这里的100
可以自行定义。`

- section这种记录，对于输出显示是比较有用的，比如，前端如果要针对某个序列上涨下跌显示不同的颜色，则可以为这个序列将上涨与下跌设置为不同的section。

- **todo 为每个section增加 flag 增加mark.**

可以用S.set_section(100); 来设定相应的section. 但不能是-1。

// sections 记录 sections是用来记录某个序列根据某些条件形成的分段。

// 比如MACD指标上涨下跌，在0轴之上之下，共有4种类型的sections，可以用这个序列 `S.set_section(100); S.set_section(200); S.set_section(300); S.set_section(400);...` 来记录它。

- 对于S序列来说，`S.set_section()` 必须在每个切片都调用，否则，下一个切片自动调用 `S.set_section(-1)` ;
- 如果S从来没有调用过 `S.set_section()` , 则 S序列将会有一个初始长度为0的 section .

```

1          // MACD 3
2      {
3          //   EMA_S3.ema();
4          //   EMA_L3.ema();
5          DIF3 = (EMA_S3.ema() - EMA_L3.ema()) / 1.5; // todo 这种方式, e
ma()是否执再执行了一次?
6          X3.ema(DIF3, 55);
7          DEA3.ema(X3, 13);
8
9      if (X3 > X3.ref(1)) {
10         X3.set_section(100);
11     } else {
12         X3.set_section(200);
13     }
14 }

```

MA_t EMA_t 增加对 double的调用

```

1          // DIF3 = (EMA_S3.ema() - EMA_L3.ema()) / 1.5; // todo 这种方
式, ema()是否执再执行了一次?
2          // X3.ema(DIF3, 55);
3          X3.ema((EMA_S3.ema() - EMA_L3.ema()) / 1.5, 55);
4          double kk = DIF3.cur();
5          X3.ema(kk, 55);
6          X3.ema(DIF3.cur(), 55);
7
8          DEA3.ema(X3, 13);

```

5. double序列在 ref() 参数异常时，返回当前值。

- 当使用 C-C.ref(1)时，如果这时候index. this->x ==0，那么 ref返回 NAN时，导致 C-C.ref(1) 为NAN。sma不能给出正确结果，出错。例如 RSI的计算如下：

```
rsi = sma_max.sma(std::max((C - C.ref(1)), 0.0), N1, 1) / sma_abs.sma(st
d::abs(C - C.ref(1)), N1, 1) * 100;
```

- 当然也可以在RSI指标中，先判断ref(1) 是否为NAN。或者是否存在 ref(-1)。但这样会导致在写指标时的完整性。由系统来处理比较好。

```
1 double ref(int x) {  
2     if (x >= 0 && x <= this->x) {  
3         return this->data[this->x - x];  
4     } else {  
5         // return NAN;  
6         // return 0; // 这里不能返回 NAN!  
7         return this->data[this->x]; // 返回当前值。 这样最合理。  
8     }  
9 }
```