给大家分享我今天两个定位的小故事。

1. 支撑SNMP定位MMLbundle的问题。

SNMP 就是简单网络管理协议的缩写，然后我们有一个项目专门做这个，他们部门就叫做SNMP，MMLbundle就是MML的bundle，bundle理解为模块吧，MML是一个开发工具，主要是做命令的。这次SNMP就是新增两个维护命令，查询一些东西，简单地说就是查询一下一些信息的命令。比如你在windows下面做一个命令查询一下每个盘有多少个word文件，那我可以搞一个命令叫做 DSP WORDFILENUM (display word file number)。设置几个参数，如要查询的盘符（C盘or D盘等等，或者不选就全部一起查）。就类似这种命令。

然后他做这个命令做出来有问题，因为snmp是一个新的bundle，以前我们都是在旧的bundle添加命令，他是要新增一个bundle，再新增命令的框架。结果搞出来报错了，命令跑不动，一半代码还是平台的不可见，照着平台给的指导书搞也没有搞出来。这种就是啥也不知道，那时候已经下午4点过了，强哥让我去帮他们搞。我说这个没人了解，没有把握，尽力而为。

于是我接手了这个坑爹的东西，lian meng dai cai，与其他已有的bundle做对比。发现了他们的很多问题，逐一解决。后来基本被我把问题解决完了，想编个包上环境跑一把试一试，结果发现平台自动生成的代码（代码框架是平台的，我们注册一些函数，写一部分内容，平台根据我们注册的东西，自动生成剩余部分的内容。）总是多了一个文件夹（与之前的bundle对比发现）。于是我联系平台的人，问为什么多了，平台告诉我说新机制文件夹已经变了，并且以前我要自己手动添加的一些内容，平台会自动生成，让我不要用之前的文件夹，就用新生成的。

但是我早就研究过，新生成的文件夹内容少了很多函数的定义，并且没有在启动代码的框架中注册内容，根本调不到。（平台都是python的，python做完了才是我们C端的）。 我就随便找了一个例子，跟平台沟通说比如XXX定义就找不到，平台的人很没有耐心，说是我们坑爹，（内心想着产品都是菜B,平台做底层的，技术更高，工资更高，外包更是菜B），说我们肯定犯了什么低级错误，没有照着他们的指导书来搞。我就说是照着来的，没有问题，然后我把XXX定义，就是平台新增的文件夹没有这个定义，之前旧的文件夹有，我全局搜索给他看，我说你看确实找不到。这时候平台态度就变了，但是他不想接锅啊，想了下说自己也不是很清楚，不是自己弄得。我就一直问他谁弄得，我去找相关责任人，他也不回答，就在那儿纠结。然后我也懒得跟他纠结，把他给的指导书翻出来，给他看，说你看这里定义了啥，这里定义了啥，你们指导书的定义不全啊 ! 这时候他终于想起来了，他说这个指导书太旧了，我找个新的给你。 新的指导书就把我所有觉得要加的定义全部加上了。但是也有一些坑，被我及时搞定了，最后在6点完成了整个新bundle维护命令的添加。

后来snmp的同事，跟我比较熟那个，还请我喝了杯奶茶。还不错。。。

嗯，怎么说呢，一个新东西，里面有一半的定义不可见，各种猜测推理吧，加上平台也坑我们，最后还是及时弄出来，我觉得，还是不错吧。整个过程我比较有耐心吧，步步为营，把每一个地方都推敲了一遍，我说得最多的一句话就是，“别着急，看看，多对比下。”snmp的兄弟说得最多的一句话就是，“卧槽，好坑，好多坑。”然后一脸懵比地望着我。。。

还是要冷静吧，不管是时间上的压力，对未知的恐惧，还有对平台质疑我能力差的压力，各种吧。

1. 定义EAP指标请求次数减去成功次数小于失败次数的问题。

嗯，EAP是啥，EAP认证，是一种认证方式，其中我们家里面的路由器那个WAP2就是EAP认证的一种认证类型。

指标，就是统计某些我们需要的数值，比如在EAP认证的流程中，把用户EAP认证请求的次数，EAP认证成功的次数，因为各种原因而失败的次数都统计下来。这样可以观察到我们因为什么而失败，失败占比等等指标，可以由此来评估我们的性能。

EAP认证失败的原因有很多，七八种吧，比如认证超时了，去认证的时候就直接回你失败了，反正在EAP认证流程中，各种失败都做一个指标来统计一下。反正就是用户发个eap认证请求，然后等认证服务器回应他。他们两个互相交互多次，多次互相伤害，才确认没问题。简单画个图：

用户 认证服务器

我可以认证不？ \_——》

《—— 来啊

我真的要来咯？ ——》

<—— 草泥马赶紧上车别BB

好的，那我上车了？ \_——》

<—— 我日，快上！

要得要得 ——\_>

《-—— 别慌，你特么是用户吧？

草泥马……. -----》

。。。。。。。。

大概就是这种互相确认互相伤害的各种交互。

除了我们已知的失败，还有未知的失败，我们叫做因其他原因认证失败。那么公式就可以这样表示：

EAP认证已知失败原因次数（七八种）+ 未知失败原因次数 =

EAP认证请求次数 – EAP认证成功次数

好的，公式给出来了(其实eap还分全流程和半流程，漫游和非漫游，就不扯那么详细了)。那么问题是什么呢？就是让几百个用户去认证，最后统计出来的结果发现，失败次数 > 请求次数 – 成功次数

我擦，这不科学啊。。于是让我去定位，我首先去找做这一套指标的人，问他这个之前有遇到吗？他说遇到过，而且一直有，他也很好奇是为什么会这样，我简直是日了狗。。。你做这个的，你也很好奇，当时有问题，你也不去定位，不改。。。。然后看了下他最后的计算公式，怀疑一下计算公式有没有问题，现在我先用英文代替一下几个指标：

EAP认证成功次数： EAPSucc

EAP认证请求次数：EAPReq

EAP认证因已知原因失败次数：EAPFail

EAP认证因其他（未知）原因失败次数：EAPFailOther

我去理了一下代码，结论就是，在打点（打点的意思：走到对应的流程，将对应的全局变量结构体成员加一，比如代码跑到了EAPReq，那么g\_stEapCounter.ulEapReq++,Counter就是指标的意思）之前，代码的公式计算是正确的，也就是EAPReq – EAPSucc = EAPFail + EAPFailOther

好吧，只能继续研究了，看了下机制，是这样的。有两个全局变量存指标的值，一个叫做g\_stEapCounter，里面的成员就代码有哪些指标。Such as：

Typedef struct tagEAP\_AUTH\_COUNTER {

ULONG ulEapReq;

ULONG ulEapSucc;

ULONG ulEapFail；

ULONG ulEapFailOther；

}EAP\_COUNTER;

EAP\_COUNTER g\_stEapCounter = {0}；

只要跑到了一次对应的流程我就把对应的成员加一。

另外一个全局变量叫做g\_stEapCounterSave，他们是一个结构体，但是用法不同，这个是拿来保存上一次指标的值的。 啥意思呢？ 原来为了优化性能，减少操作次数，这里设置了一个定时器，每隔30秒把所有EAP指标统一打一次点。那么每次打点的值就是g\_stEapCounter – g\_stEapCounterSave，每次打点完了，就把g\_stEapCounter的值赋给g\_stEapCounterSave，让g\_stEapCounter继续去增加，等到30秒定时器超时，又来打点g\_stEapCounter - g\_stEapCounterSave，这样可以减少打点的次数，增加性能。

用代码简单写一下就是：

ProcAllEapCounter() /\* 处理所有的EAP指标 \*/

{

AddEapCounter(g\_stEapCounter.ulEapReq-g\_stEapCounterSave.ulEapReq);

g\_stEapCounterSave. ulEapReq = g\_stEapCounter. ulEapReq;

AddEapCunter (g\_stEapCounter.ulEapSucc-g\_stEapCounterSave.ulEapSucc);

g\_stEapCounterSave.ulEapSucc = g\_stEapCounter.ulEapSucc;

…… /\* 后面还有失败次数，和其他原因失败次数就不写了 \*/

}

而且，EAPFailOther是减出来的，也就是我们把知道的失败原因和成功的次数，用请求次数去减就是EAPFailOther。准确点说，就是用

g\_stEapCounter.(EapReq – EapSucc- EapFail) = g\_ stEapCounter.EapFailOther

看到这里，就特么反科学了啊，这特么公式都是减出来，还能有错么？30秒内，把所有现在的指标数量减一下，应该没有问题啊！！！

到底问题是什么呢？

于是我加了日志，把指标都打出来，发现一个诡异的现象。例举g\_stEapCounter的两组数据如下：

First

Req 100

Succ 90

Fail 0

Other 10

Second

Req 100

Succ 90

Fail 10

Other 0

根据这两组g\_stEapCounter成员的数据，我们最后的打点会是怎样呢？

指标名 变换状态 打点值

Req 100 -> 100 0

Succ 90 -> 90 0

Fail 0 -> 10 10

Other 10 -> 0 0 /\*减少自然不会增加打点数量\*/

看到这里就有意思了，有10次因为其他原因失败，隔了30秒到下次打点的时候，都变成了因为已知原因失败。这特么就尴尬了，你看，现在我们再来拿这个整理出来的打点表计算一下，

req（0） - Succ（0） - Fail（10） != Other（0）

这特么多出来的10次是什么鬼？原来特么的还带内耗的？

于是我马上去大量上下线用户，查看多组数据，发现这个情况是必先的，有多少次其他原因失败，在下一次定时器处理中都会变成已知原因。也就是，先是OtherFail 多少次，然后Fail 就会有多少次，那这会不会是重复打点了呢?

想睡觉了，不多扯了，直接揭晓答案:

用户在认证的时候，发起req（假设时间为25，30的时候定时器会超时，开始打点），那么用户等认证结果时，如果成功了就会快，一两秒，失败了就会一直转啊转，因为失败了会发起重认证流程，多尝试下。结果失败大概花费七八秒的样子，也就是失败的时候，时间点是32或者33，这个时候30的打点已经过了，只能等下一次打点，60的时候打点了。那么，问题就是这样。画个图（标红加粗为打点时间，假设刚开始全局变量都是0）：

Time g\_stEapCounter++ 打点值

…

25 req

**30**  req++**， other++**

33 fail

…

**60**  fail++

由上图，我们可以看到，30处打点定时器超时之前，只有一次req全局变量有增加，但是这个时候，由于用户还没有收到认证服务器的回应(成功或者失败)，30处，定时器超时，而根据公式 :

failother = req（1） – succ（0） – fail（0） = 1

也就是failother++，变为1 了。

30过了之后，我们在33才收到了认证服务器的回应，全局变量fail++，到60处，定时器超时，根据公式：

Failother = req(1) – succ(0) – fail(1) = 0

Failother又变回0了!！！

这里很明显地，因为时间错开的原因，导致多加了一次失败次数。而且全局变量本身应该一直增加的（有一次成功就加一次，有一次失败也加一次），居然还减少了。这特么还内耗的。

说了这么多，为啥我要深夜写这篇总结，其实是因为，我发现我的数学太垃圾了。说得没错，懂得越多越觉得自己无知。每次我进步了，都觉得自己是一个垃圾，基础还很差。

大家不要觉得，我们那个小游戏，感觉做了半天，其实也没弄到啥，而且坚持着很累。但是其实你已经获取了一些东西了，坚持下来，会有收获的，总比碌碌无为强多了。

以此总结，与大家共勉。