物品刷取脚本，顾名思义，就是可以借此脚本凭空刷取任何想要的物品并且不附带任何额外的代价。看上去这很不可思议，但对于消逝的光芒2这种偏单机的游戏来说，这完全能够实现。下面就来详细分析脚本的制作，这里以1.24.1版本为例。

**I.设法找出游戏中能够实现这一功能的函数**

程序是由一个个函数组成的，通过调用不同的函数，实现不同的功能。那我们可以假想游戏中有一个函数，游戏程序通过调用这个函数，实现往背包仓库中添加物品的功能。如果确实存在这么一个函数，我们就可以自行调用这个函数，往背包仓库里加入不同的物品，来达成刷取物品的目的。

那怎么找出这个函数呢？

首先需要知道函数是在什么时候被调用的，然后根据调用后引发的现象来反向寻找这个函数。如，调用后引起相应的数值变化。而在这里，显而易见，当我们在游戏中拾取地上的物品、购买商店的物品、搜寻尸体的物品等时，都会往背包中添加对应的物品。所以，往背包中加入物品的函数肯定是在此过程中被调用的，而调用后引发的现象之一就是背包中物品数量的增加。这样，我们就可以通过找出是什么代码改变了背包物品数量来间接找出函数。

在这里以配件物品为例（当然其他物品的数量也可以），通过搜寻配件物品数量的地址来监视何处代码改变此数值来找出函数。这里通过丢弃手雷和拾取手雷，不断地过滤地址，最终可以得到储存手雷配件数量的地址。



图1.1-配件自制手雷

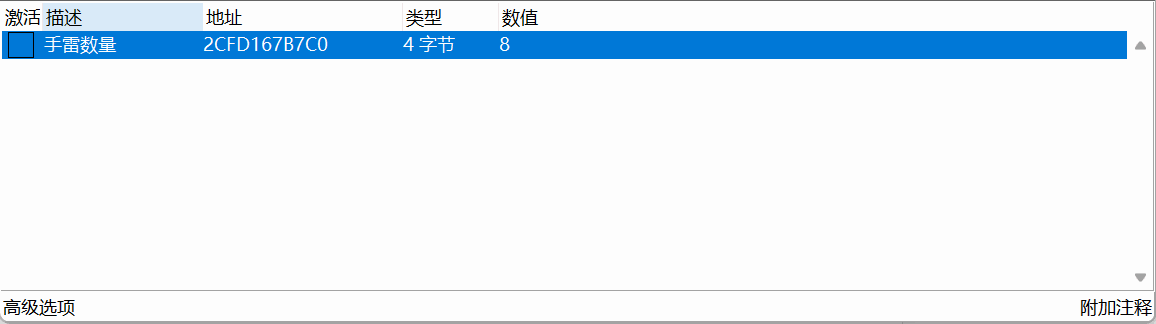


图1.2-获得的储存手雷数量的地址

接下来，就来找出往背包中添加物品的函数。右键点击此地址，使用“找出是什么改写了这个地址”功能，监视何处代码在起作用。接着拾起丢弃的配件道具手雷，获得改写背包中手雷数量的代码。

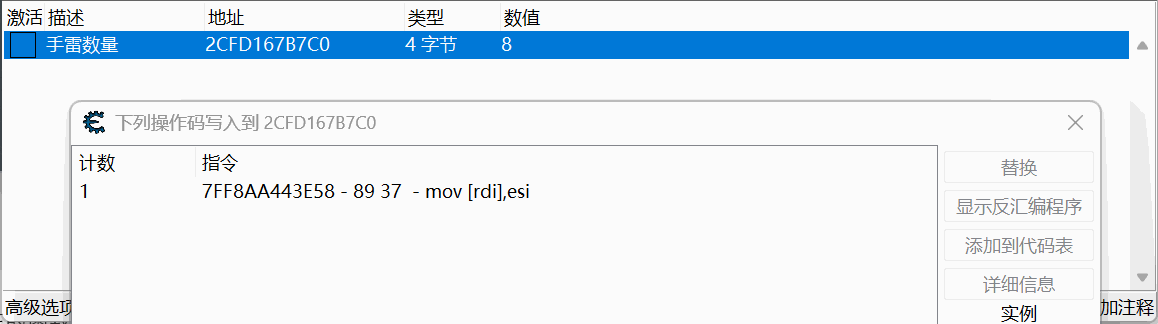


图1.3-获得的改写手雷数量的代码

那么，此处代码所在的函数就是我们想要的目标函数吗？事实上，此处改写数值的代码所在的函数不一定是我们所想要的函数。因为往背包中添加物品涉及多个过程，可能会调用多个函数来完成此过程，而此处所在的函数很可能只是其中的一部分。我们知道，这里的代码是改写配件物品数量的，那么此处的函数可能就是被封装为一个管理背包配件物品数量的功能而已，是整个添加物品这个功能中的一部分，远不是真正的目标函数。

来到这里，我们需要做的就是查找调用方。何为查找调用方？就是通过堆栈保留的函数调用信息来向前寻找调用此函数的调用方。因为调用方会调用更多的函数，实现更多的功能，所以调用方函数的功能会更加完善，进而更靠近我们的目标函数。上面我们分析到，此时找到的代码所在函数可能是实现功能的一小部分，所以我们需要查找此函数的调用方。接下来，我们就可以利用堆栈来寻找往背包中添加物品的函数。

在寻找之前，我们需要做一个小测试。我们的目标是找出往背包中添加物品的函数，那么与之相对应的，我们在寻找过程中的行为也应该是和“添加”这个行为相关的，比如以分解物品、丢弃物品、售卖物品等等的行为得来的结果也就不可用了。同样的，我们可以借此来排除无关的代码。我们上面得到的代码，是通过拾取手雷这一配件而得来的，同时分析得到这代码所在函数可能只是用于管理配件物品数量。既然是管理配件物品数量，那就必然存在增加和减少这两个操作，而这两个操作，会不会同时在这个函数里面实现呢？如果是，那我们要查找的调用方对应的就是添加物品时的调用方，那这样我们又如何去查找呢？所以我们需要在这里进行一个小测试，就直接把手雷配件给丢弃掉，观察刚才获得的代码的计数次数有无变化。如果计数次数增加了，表明此代码所在的函数是一个兼具增加和减少物品数量功能的函数，需要我们想出别的方式去排除移除物品的调用方。

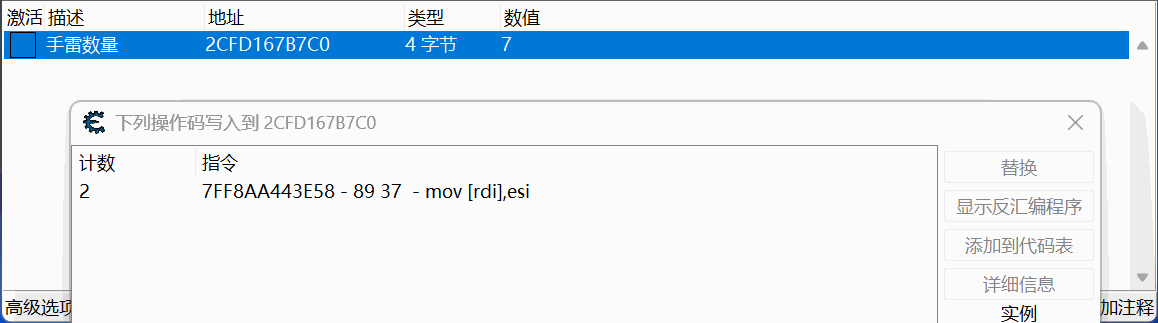


图1.4-功能兼具测试结果

从上图1.4的结果可以看到，计数次数增加了，证明了我们刚才的猜测，这需要我们想方法去排除移除物品的调用方。但其实这样很简单，上文我们说到目的与行为相对应，我们就可以用“添加”这一行为进行排除。我们来到反汇编界面，在此代码上下断点，接着直接拾起地上的配件手雷就可以了。因为拾起属于“添加”的行为，伴随的调用方也必然是添加物品的。这时可能会有矛盾，如果我们的目标函数也是像上面所说的兼具增添和移除的功能怎么办？这里先不用着急，后面会进行处理。就算真的是，由于在一次的调用中不是增添就是删除，而能控制这一行为的只有调用时传递的参数，所以只要我们调用时控制参数也同样能够实现我们脚本的功能。

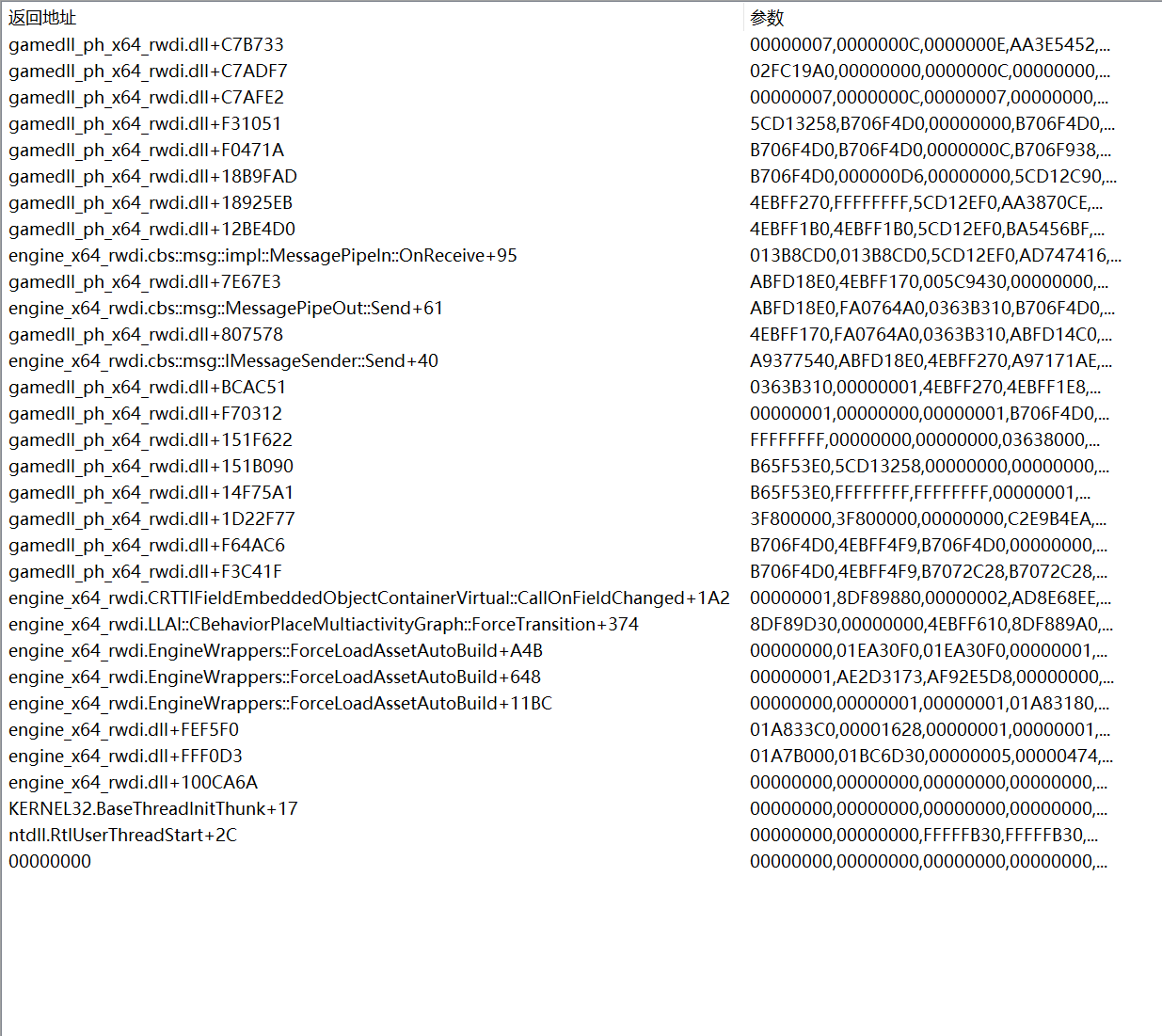


图1.5-得到的增添配件物品的调用方

如上图1.5所示，这里，我们下断点后得到了调用方。问题是，这里有那么多的调用方，究竟哪一个才是我们想要的呢？这时候，我们要想办法去筛选了。由于上图的调用方是通过拾取配件手雷得来的，所以这里我们并不确保这些调用方中有能添加主武器的，有可能只是专门添加配件物品的。来到这里，问题就变得复杂起来，我们希望找到一个函数，而这个函数能添加任何的物品，而不单单只是添加某一种或某一类物品。当然以上只是一种猜测，也有可能调用方中存在多态的使用，可以实现一个函数中添加不同种类的物品。但究竟是哪一种可能呢？所以我们在这里也要做一个测试，判定是上述情况的哪一种。当然测试过程也很简单，就是把我们刚才所得到的调用方地址给记录下来，然后从前往后，依次地在调用方所在函数的头部下断点，接着去游戏中拾起不同种类的物品，观察是否有断点处成功断下来。如果有断点断下来，那就说明我们所希望的函数是存在的，反之，则不存在这样理想的函数，需要我们之后找到各个对应的函数来制作脚本。现在，我们从背包的武器栏中丢弃一样物品，然后再从地上拾取。

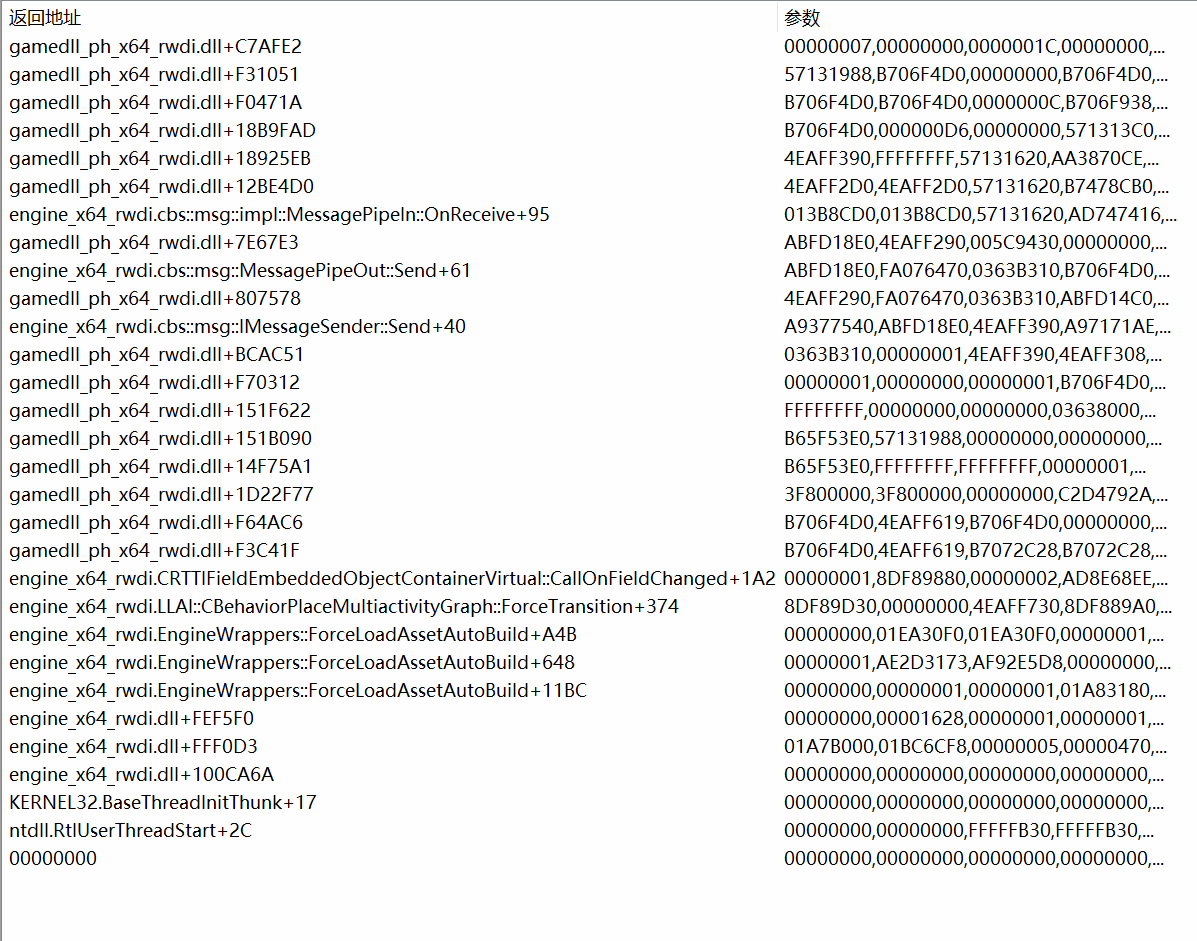


图1.6-得到了添加武器物品的调用方

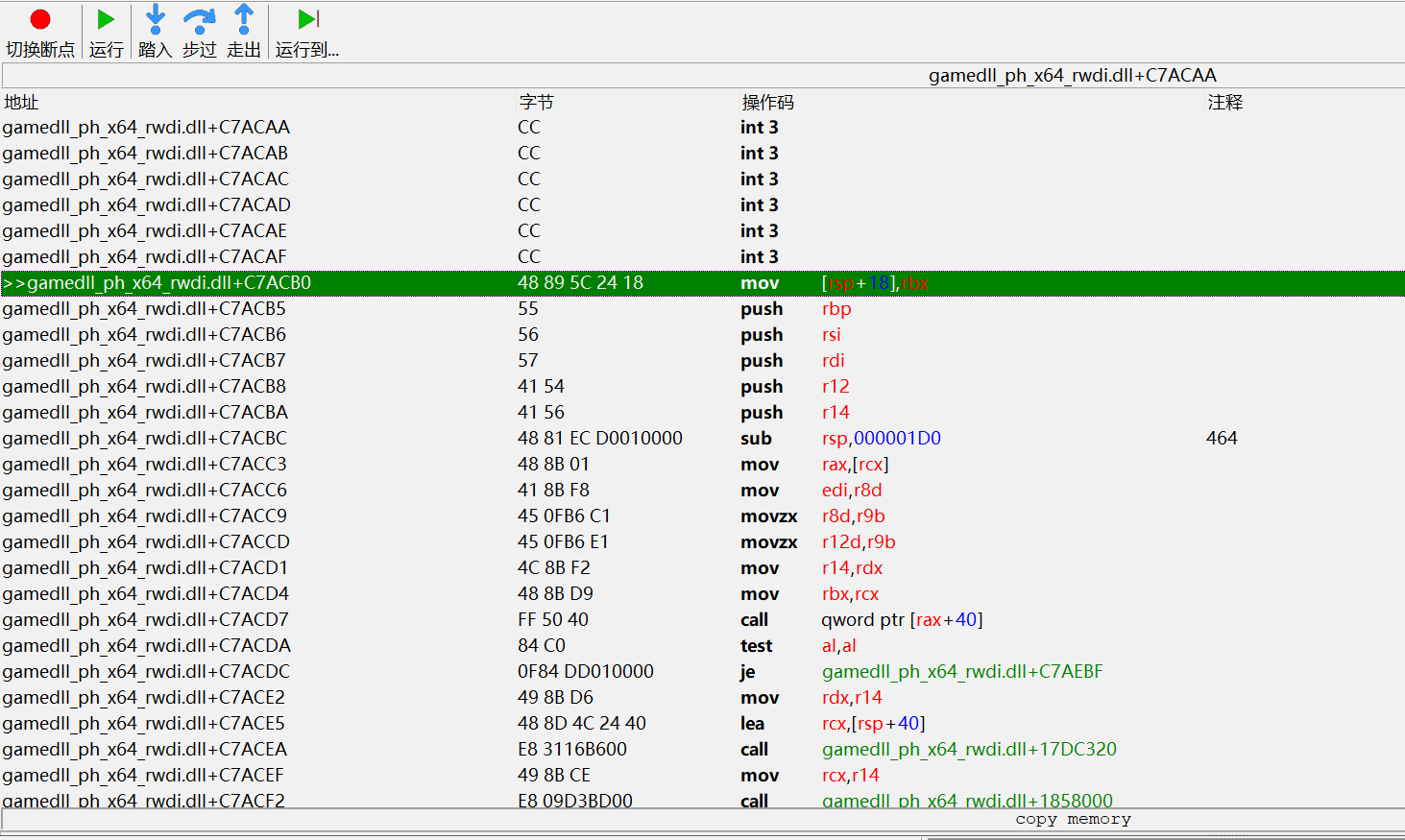


图1.7-调用方处存在理想函数

如图1.7，我们成功地在调用方地址gamedll\_ph\_x64\_rwdi.dll+C7ADF7函数的头部地址gamedll\_ph\_x64\_rwdi.dll+C7ACB0处断下来。这表明我们要找的函数是一个理想的函数，能够处理任何物品的添加（可以对每个种类的物品进行验证，这里不过多展示）。接下来我们就可以通过调用方来寻找我们希望找到的函数了。比对这里得到的调用方和之前通过拾取配件手雷得到的调用方，我们不难发现这次得到的调用方除了没有之前得到的调用方的最近的一个之外，从后面一个开始就是完全相同的。这就很好办了，现在我们可以不用管物品的种类，因为即使是不同种类的物品，也具有相同的函数调用。现在，可以直接从调用方所在的函数去分析是否为我们所想要的函数。要确定调用方所在函数是否符合我们的要求也很简单，就是去调用方函数的头部下断点，然后拾取物品，观察传递进来的参数对所拾取的物品是否有全面的描述，如：数量、耐久、伤害等等。接着就是尝试去修改参数，然后释放断点，观察物品进入背包之后是否变为了我们修改参数之后的，如果是，那么这个函数就能作为我们的目标函数，进而能够进一步研究其参数然后使用。反之，接着研究下一个调用方所在的函数，直到找到符合我们要求的函数。下面就来一个一个地在调用方所在函数进行断点，观察参数，修改参数，考察修改后的运行结果。

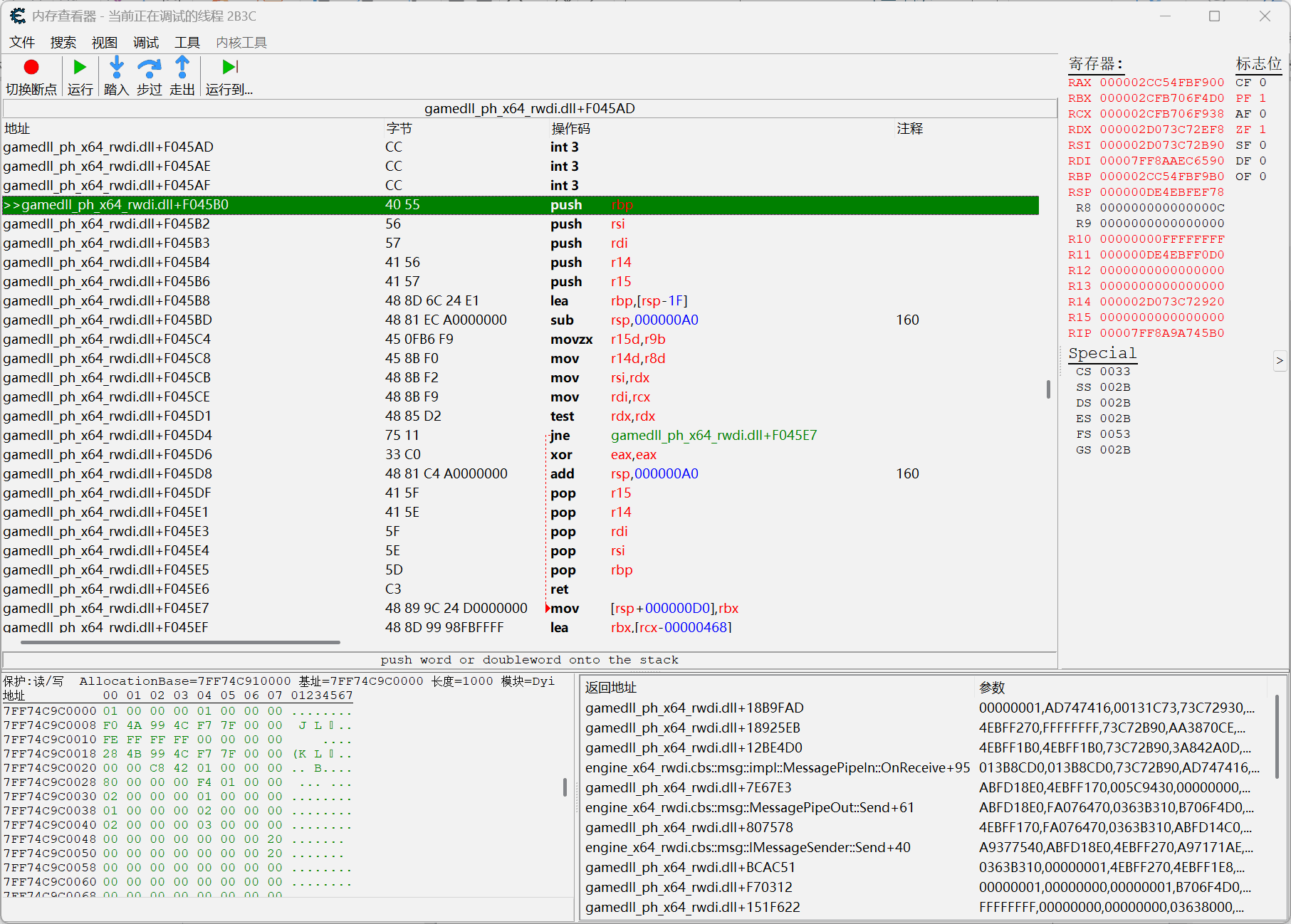


图1.8-得到符合要求的函数以及各个寄存器中的参数（rcx、rdx、r8、r9）

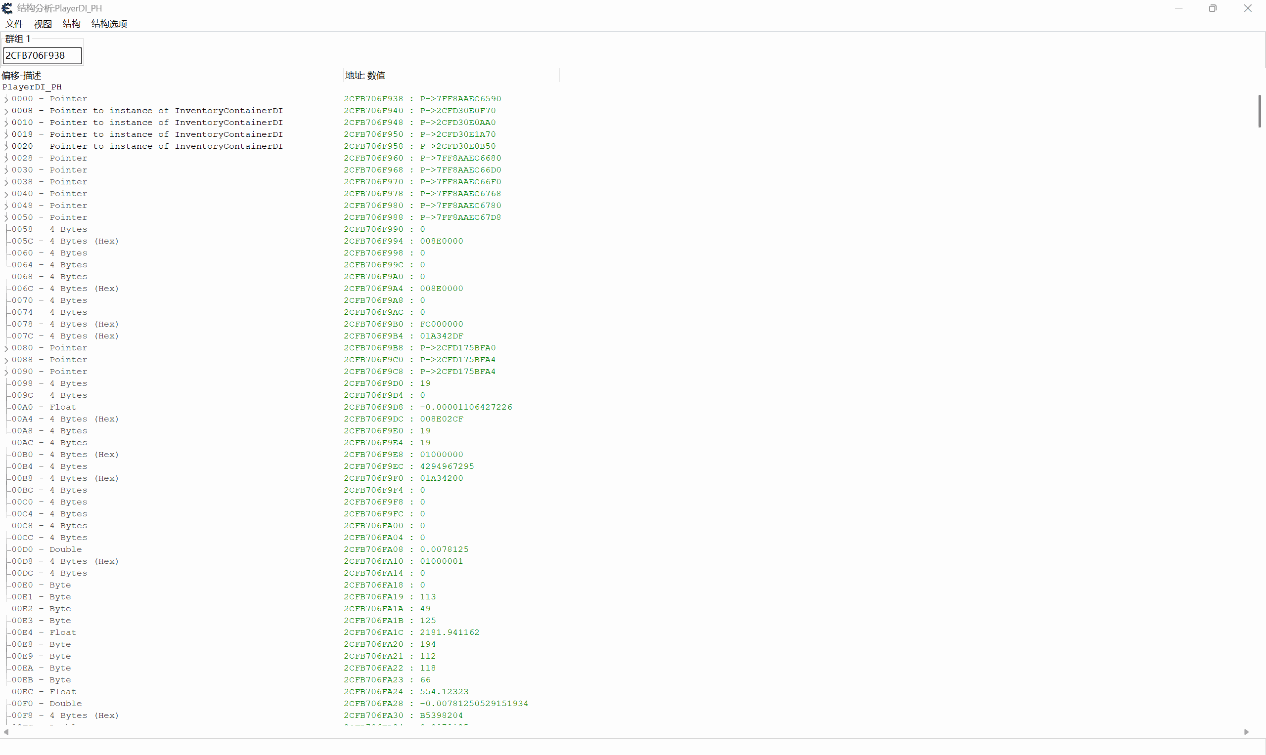


图1.9-符合要求的函数中的寄存器rcx中参数的结构

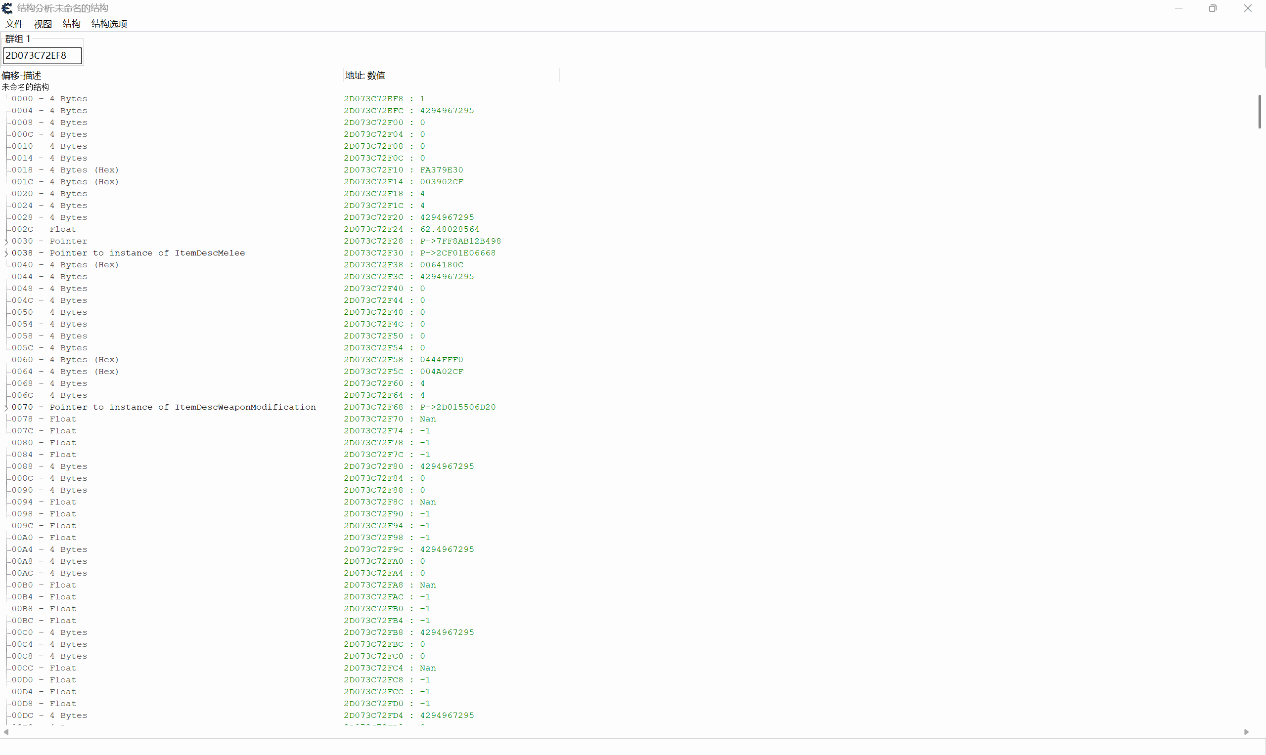


图1.A-符合要求的函数中的寄存器rdx中参数的结构

如图1.8所示，我们在函数gamedll\_ph\_x64\_rwdi.dll+F045B0处断下，发现此函数预期符合我们的要求。如图1.9所示，参数结构中包含有inventory的字眼，表明这是与背包仓库有关的参数，而添加物品必然要关联到背包仓库，这符合我们的预期。加上如图1.A所示，其中的结构名称含有ItemDescMelee（近战武器项目描述）、ItemDescWeaponModification（武器项目描述修订）的关键字眼，表明了此参数包含了对物品的描述。且在位置0x0处是1，刚好匹配了一件武器的数量（虽然有点牵强，后续会进行验证）。接着，我们就来尝试修改以下武器数量，直接改成99。然后我们释放断点，让武器添加到背包中，观察背包中刚才捡起的武器数量是否变为99。



图1.B-修改参数，武器数量成功变为99

如图1.B所示，参数的修改生效了，这表明在寄存器rdx处的参数结构0x0处正为物品的数量。这可初步判定函数gamedll\_ph\_x64\_rwdi.dll+F045B0为我们的目标函数，可往背包中增添任意物品。但要完全确定，就要进一步研究这个函数。游玩过游戏都知道，从地上拾取物品后，伴随的就是地上物品的消失、右侧出现获取物品的UI提示、背包物品的增添展示，这是完整的拾取后添加物品的过程。如果这个函数真的是完全地处理（可以不包括地上物品的消失，因为我们要的是凭空刷取，函数可以不包含处理地面的物品的行为），我们就可以先屏蔽这个函数，然后再从地上拾取物品。如果拾取后地上物品不消失（也可以消失），右侧无UI提示出现，背包中无拾取的物品，那么这个函数就是真正的我们所希望获取的函数，它是真正的往背包中增添物品的函数。现在，我们来到反汇编界面，在此函数的头部将汇编代码改为ret，让此函数被调用后直接返回，不执行函数里面的内容，观察是否出现上述的结果，从而验证这个函数。

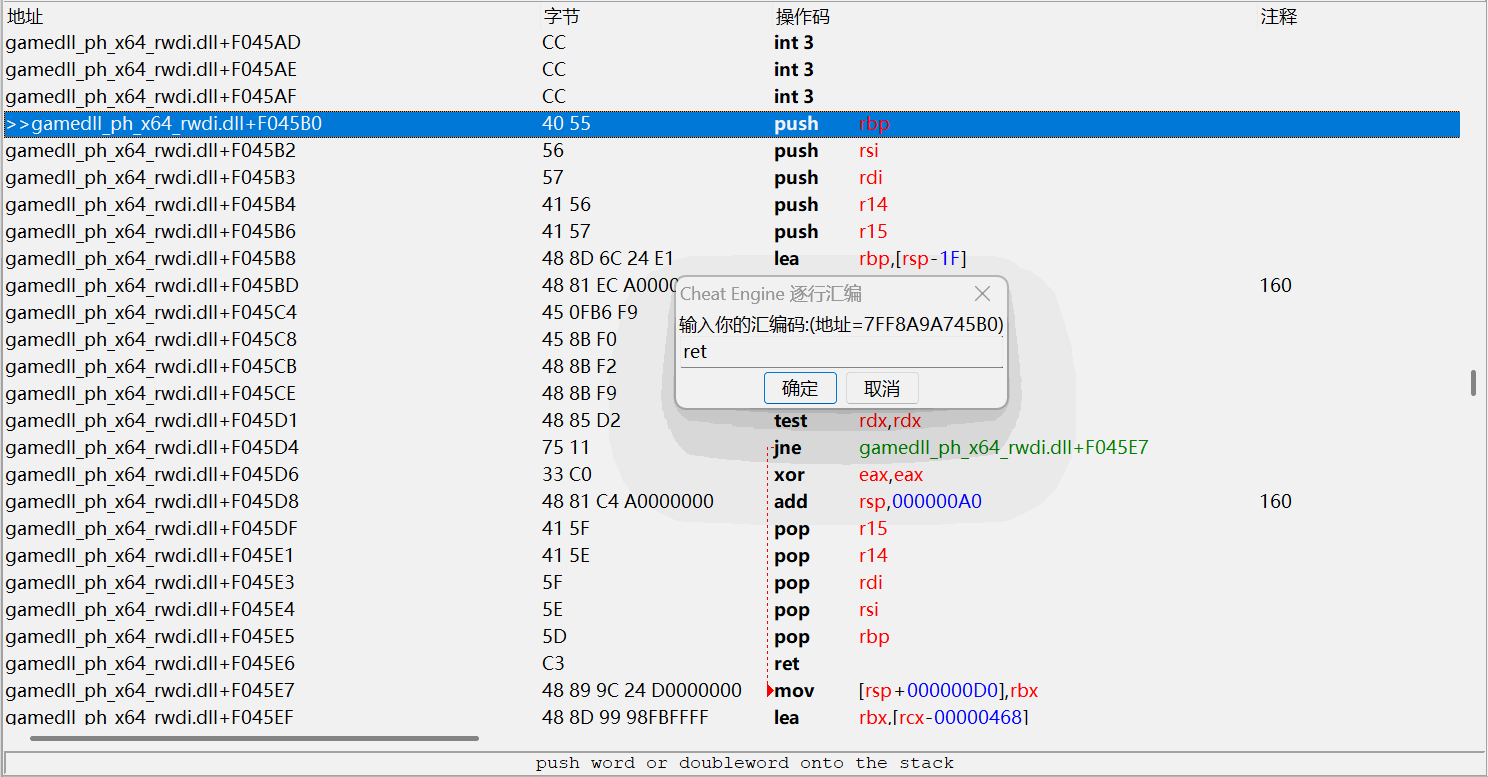


图1.C-将函数头部代码修改为ret，使函数直接返回

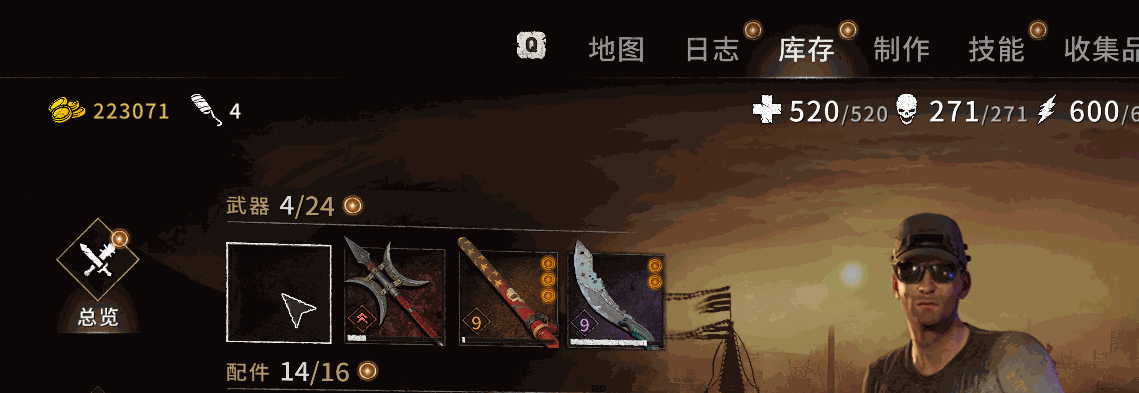


图1.D-修改后再次拾取地上的武器，地上物品消失，右侧无UI提升，背包中无物品

如图1.D所示，在修改后，拾取地上物品，会出现上述所描述的现象，这就完美证明了函数gamedll\_ph\_x64\_rwdi.dll+F045B0正是我们所要的函数。加上屏蔽了这个函数后出现的现象是地上物品消失、右侧无UI提示出现、背包中无拾取的物品，表明了这个函数不参与处理地面上的物品，也就是说这个函数的参数不应包含掉落在地面上的物品的参数，这使得我们可以自定义参数，凭空刷取我们任何想要的物品。至此，我们已经找到了往背包中增添物品的函数，其中它的相对地址是gamedll\_ph\_x64\_rwdi.dll+F045B0。接下来，我们的目标就是搜寻和分析这个函数的参数并且研究它的返回值。

直接屏蔽函数验证其实是一种比较粗暴简单而有效的方式，但同时也伴随着风险。我们都知道，函数是有其返回值的（也可以没有返回值），而返回值又是由参数加上函数内部共同决定的。现在我们屏蔽了函数，那么这个函数的调用方就得不到正确的返回值，可能就无法正确地执行代码（比如调用方需要用到返回值的指针，结果因为屏蔽而返回了空指针，就能导致内存读取的异常），从而导致游戏程序的异常或是崩溃。除了返回值的异常之外，游戏内部的数据也可能出现异常。比如函数内部有修改内部数据的代码，而我们现在屏蔽了函数使得游戏程序无法执行它们，就会导致内部数据的异常。如果游戏程序的其他地方有需要调用内部数据，也可能同样地会发生程序异常甚至崩溃。而我们在这里采用了这种方法并且游戏没有出现异常或崩溃，其实也是运气所带来的。面对冷冰冰的汇编代码，没有边修改边验证这种处理方式，我们实在想不到这些代码能够体现出怎样的实际含义。都搞逆向写脚本了，没有点运气的加成怎么行呢？

**Ⅱ.寻找和分析函数的参数，研究其返回值**

虽然我们已经找到了我们想要的函数，可是功能的实现就得调用函数，而函数的调用又必需对应的参数。只有正确的参数传递才能让函数调用成功，而错误的参数可能会导致游戏的异常乃至是游戏程序的崩溃。

在正式寻找和分析参数之前，我们应该先了解这个函数的参数到底有多少个，函数的参数个数直接决定了我们研究的工作量和研究方向。如果函数的参数只有1-4个，那么我们只要追踪寄存器rcx、rdx、r8、r9就可以了，四个参数依次保存在它们之中并传递给函数。但如果参数超过4个以上，从第五个开始参数就保留在堆栈中传递给函数了，如保存在rsp+0x20、rsp+0x28等等，这时候就要去追踪堆栈的内存，比起追踪寄存器是要麻烦得多。这样的参数传递方式是Windowsx64调用约定俗成的。

要知道函数的参数数量，就得到调用方去查看调用方的调用过程。调用方调用函数之前，一定会将相应的参数写入对应的寄存器和堆栈，便于参数的传递。现在我们在函数头部下断点，拾起地上物品，得到调用方。

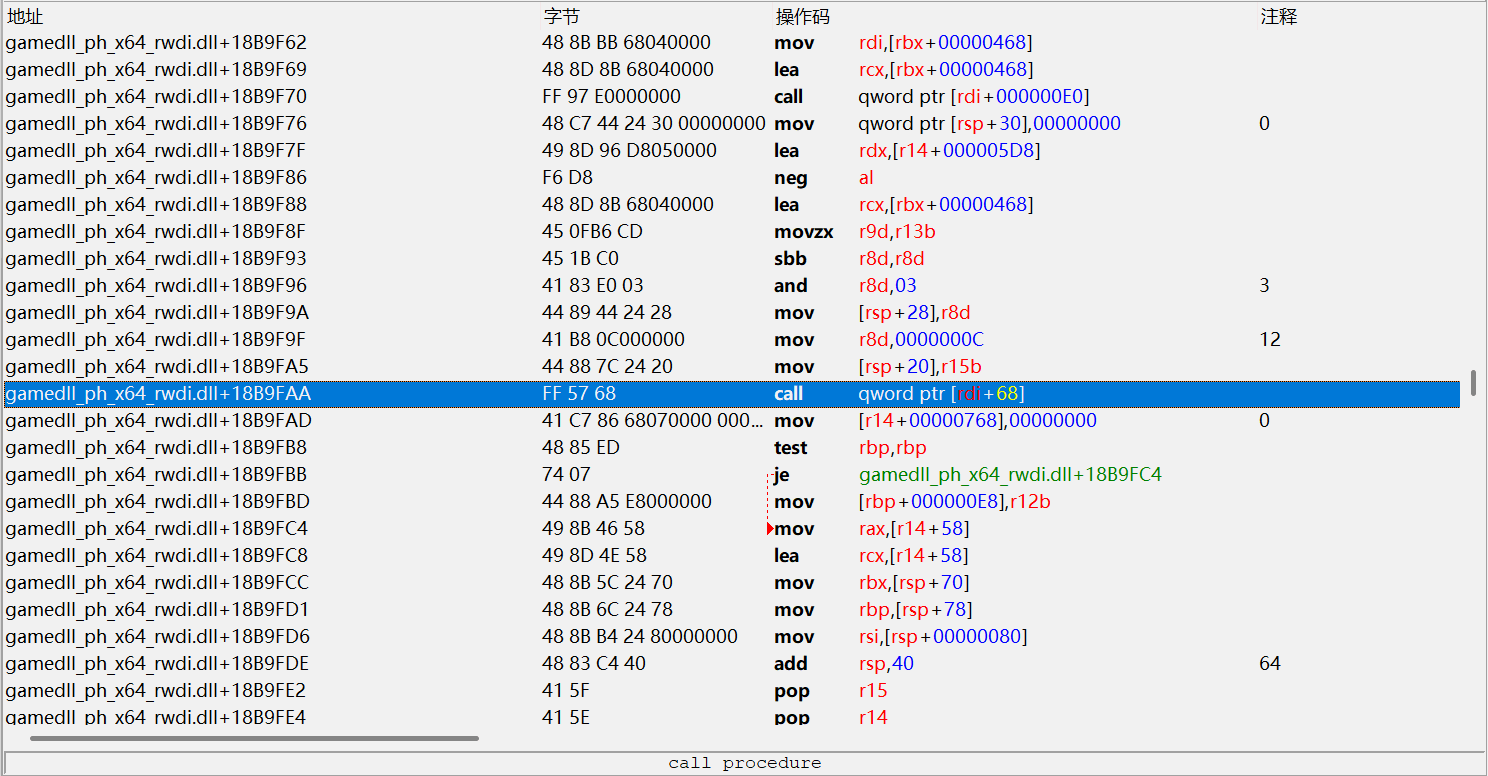


图2.1-断点后得到的函数调用方代码

如图2.1所示，地址gamedll\_ph\_x64\_rwdi.dll+18B9FAA处就是调用函数的代码，我们将在此地址处向上查找参数的数量，因为调用都必须是要写入了参数才调用的。在这里，我们要考察的其实是两个call之间的代码，在这里就是地址gamedll\_ph\_x64\_rwdi.dll+18B9F70和地址gamedll\_ph\_x64\_rwdi.dll+18B9FAA之间。为什么要这样子呢？因为当程序调用完一个函数之后，想要再调用一个函数，就必然要重新写入参数以确保参数传递的正确性。在这段代码中，有以下的关键字：lea rcx,[rbx+468]、lea rdx,[r14+5D8]、mov r8d,0C、movzx r9d,r13b、mov [rsp+20],r15b、mov [rsp+28],r8d、mov [rsp+30],0。结合上面的参数写入分析，就可以直接推测这个函数包含了7个参数。这样的方法也可以应用于其他的函数上面，确定函数的参数量，非常简单。

但我们需要注意的是，以上的方法并非是通用的。假如有三个函数，分别假设其为函数1、函数2、函数3。函数1调用了函数2，其中函数2有三个参数。然后函数2又调用了函数3，其中函数3有4个参数，其中第4个参数需要在函数2中处理获得。现在，函数1将3个参数写入，正确地调用了函数2。函数2接收到参数后就直接调用了函数3（中间除了参数传递之外没有其他任何的代码）。现在，如果你是一名逆向者，目的是寻找函数3并调用它。你看到了我的方法，找到了函数3的调用方函数2，发现函数2中在执行call指令之前只存在一条这样的参数写入指令：mov rcx,xxx，之后你可能就会觉得函数3只有一个参数。但很显然，函数3有4个参数，这样的方法失效了！为什么会这样？因为函数2直接调用了函数3，说明了函数3与函数2有共同的参数，而函数2的参数早在函数1调用之前就已经写入了，编译器会认为既然是相同的参数，就干脆把函数2的参数用作函数3的参数传递了进去，从而省略了对函数3的参数写入过程，除了一个参数不同需要写入之外，其他的照旧，写入完成后直接调用，这就是为什么你看不到包含rdx、r8、r9寄存器的指令，而只能看到有关写入rcx的指令，导致你误认为函数3只有一个参数。那为什么在上面就肯定这个函数有7个参数？因为这个函数的调用前还有一个函数调用，且在此之前程序没有任何的直接跳转跳到此处，导致程序在调用完前一个函数后，就必须要重新写入参数确保参数的正确。在调用函数程序进入这个函数执行指令时，程序通常是没有义务保存参数寄存器和参数堆栈的。因为寄存器就那么几个，程序为提高效率，会复用寄存器，包括参数寄存器、参数堆栈空间。这就导致参数寄存器和参数堆栈的内容可能会被更改，在调用完函数后要调用下一个函数时就必须要重新写入参数，即使这两个函数的参数是相同的。当然也没有绝对的说法，当函数比较简单时，有些编译器确实也会在两个参数相同的函数调用之前忽略掉参数的重新写入进而继续调用下一个函数。不过这些都要依据实际情况去查看。但是我们所找的函数显然不是简单的函数，就直接按照上述方法查找即可。但到底来说，任何事都没有绝对的说法，需要我们进入汇编代码细细地去分析。

取得参数的数量之后，我们现在来分析各个参数的意义和获取方式。接下来我们将按照参数的顺序进行分析。

我们首先分析寄存器rcx的参数，也就是第一个参数。在上面我们获取此函数的调用方时，我们注意到调用方的调用方式是这条指令：call qword ptr [rdi+68]。有没有觉得这很熟悉？它没有具体的函数地址，而是用[rdi+68]替代了。事实上，这是一个虚函数，因为虚函数是类和结构才拥有的，表明这个函数的调用是一个类或结构里面的调用。既然是类或结构的调用，那么就必然会传递一个隐含参数this指针。作为隐含参数，且在Windowsx64平台上，那么它就应该保存在寄存器rcx上。还有可以证明这一点的就是，如图1.9所示，在寄存器rcx储存的参数的结构中，开头0x0处储存的是高地址（0x00007fxxxxxxxxxx），也就是储存着虚表。而虚表的储存位置通常为其对应的类或结构的首位，这同样也验证了rcx就是this指针。 这样就很好办了，既然是类的指针，那么它就不用我们过多地去分析，只要我们找到它的基址以及偏移就可以了，每次调用时通过指针读取把它读出来传递给函数就好了。因为它作为隐含参数，在程序中是自动传递的，在函数需要时才会调用、读取和修改里面的内容，无需我们过多关注里面的结构，而我们需要重点关注的是显式参数，也就是rdx、r8、r9等剩下的参数，这些才是函数所必需要的。这里指针的基址和偏移的寻找过程不过多展示，手动查看汇编寻找也好，指针扫描器寻找也好，都能找到。这里直接给出指针：[gamedll\_ph\_x64\_rwdi.dll+034F62E0]+0x468。这里的指针就能直接作为第一个参数在之后的脚本中写入rcx传递给函数。

现在来分析第二个参数，rdx寄存器中的参数。在之前我们已经知道rdx储存的其实是一个结构，且这个结构的0x0偏移处储存着物品的数量。那剩下的物品属性，如耐久、伤害就需要我们往下一个个地去尝试修改。我们使用分析数据/遍历功能来查看这个参数的内部结构。如图1.A所示，在偏移0x30处有个很醒目的值：0x7FF8AB12B498。不难看出，这里的值是一个模块内的绝对地址，并且此处（0x1788B48C9E8）是作为了指针指向这个绝对地址，表明了这是一个虚表。既然是一个虚表，就表明了从0x30处开始，以下的是一个类或结构。我们的目标是要自定义参数，然后调用增添物品函数，得到想要的物品。要自定义参数，就需要我们得到这个类或结构的结构，了解里面的结构是怎样的，以便于构造出一个一样的结构，这样在传递参数给函数使用时，函数才能正确地识别我们的自定义参数，否则，可能就会导致游戏程序的异常和崩溃。那么怎样才能了解其结构呢？难道是要一个个修改数据去验证分析吗？我们不知道这个类或结构的长度，需要我们修改至何处？其实这一切的解决方法只是需要我们找到这个类或结构的构造函数。因为构造函数目的就是为了初始化类或结构，里面包含了对各个成员的初始化操作。同时，也隐含了每个成员的数据长度、数据类型甚至是整个类或结构的长度。但是，也有另外的一种可能。我们拾取地上的物品，而地上的物品也应含有保存其各种数据的类或结构。所以，我们拾取物品时游戏程序可能只是直接把其对应的类或结构的指针指向的地址直接复制到背包中而不调用构造函数。但是，如果真的是这种情况，也不妨碍我们去寻找构造函数。因为在上面我们分析到，我们所找到的增添物品的函数是不参与处理地面上的物品的，所以我们不必因为参数是指针传递进来而不是通过新的类构造进来从而加多我们的自定义参数同时也要变为地面上的物品来处理结果地面上却没有这个物品这个忧虑而烦恼。因此，接下来我们的一个目的就是去寻找这个类或结构的构造函数。

我们知道，虚表的储存位置在类或结构的首位。但能不能知道为什么虚表会在类或结构的首位？它是怎么来的？类或结构的那段内存空间也是通过调用申请内存的函数得来的，不可能在申请内存的过程中就把虚表给写入进去吧？所以虚表的写入一定存在于构造函数中，以便程序在使用这个类或结构前把它给初始化，使其有完整的结构内容。既然写入虚表的操作存在于构造函数中，那么构造函数中就必定存在对虚表地址的引用，否则怎么可能将虚表写入其中？既然存在对虚表的引用，猜想构造函数中会存在以下的指令完成写入操作：

mov [this],虚表地址——第一种操作

lea xxx,[虚表地址] mov [this],xxx——第二种操作

认识到有以上的操作之后，我们就可以利用搜索中的查找汇编码来直接定位构造函数了（当然也不止只有以上的两种操作，比如还有mov xxx,虚表地址 mov [this],xxx，可能性很多）。我们分开两种操作进行搜索，其中this要用寄存器替代，因为构造函数接收的是一段内存空间的首地址以及其他的附加参数，不可能是一个固定的地址。由于我们不知道会用到哪个寄存器，所以我们用通配符\*替代，毕竟这个搜索功能也支持通配符。下面我们开始搜索。搜索的时候注意右侧的搜索范围限制，我们搜索的应在模块gamedll\_ph\_x64\_rwdi.dll中，否则会导致搜索的时间过长。

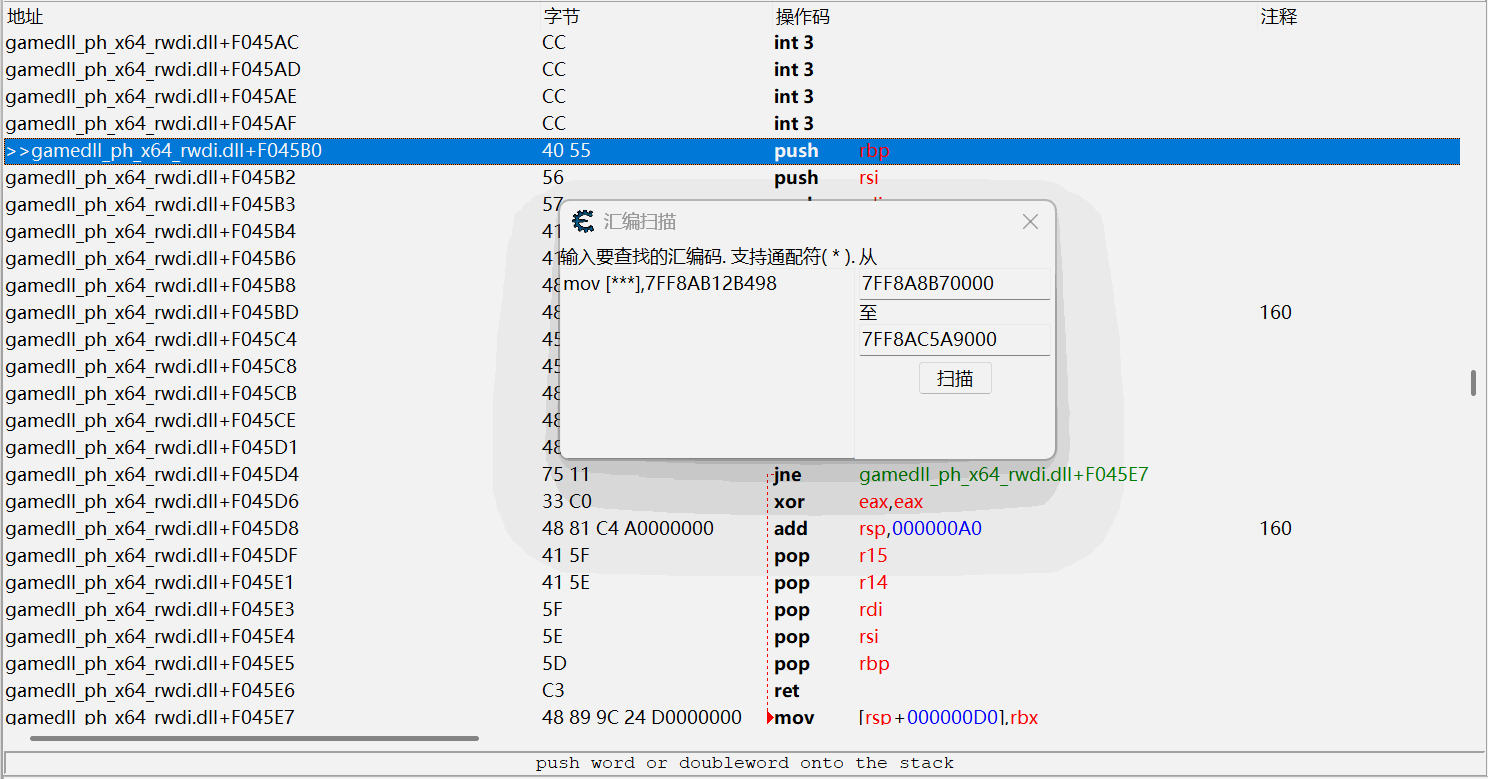


图2.2-使用第一种操作的指令进行通配符搜索

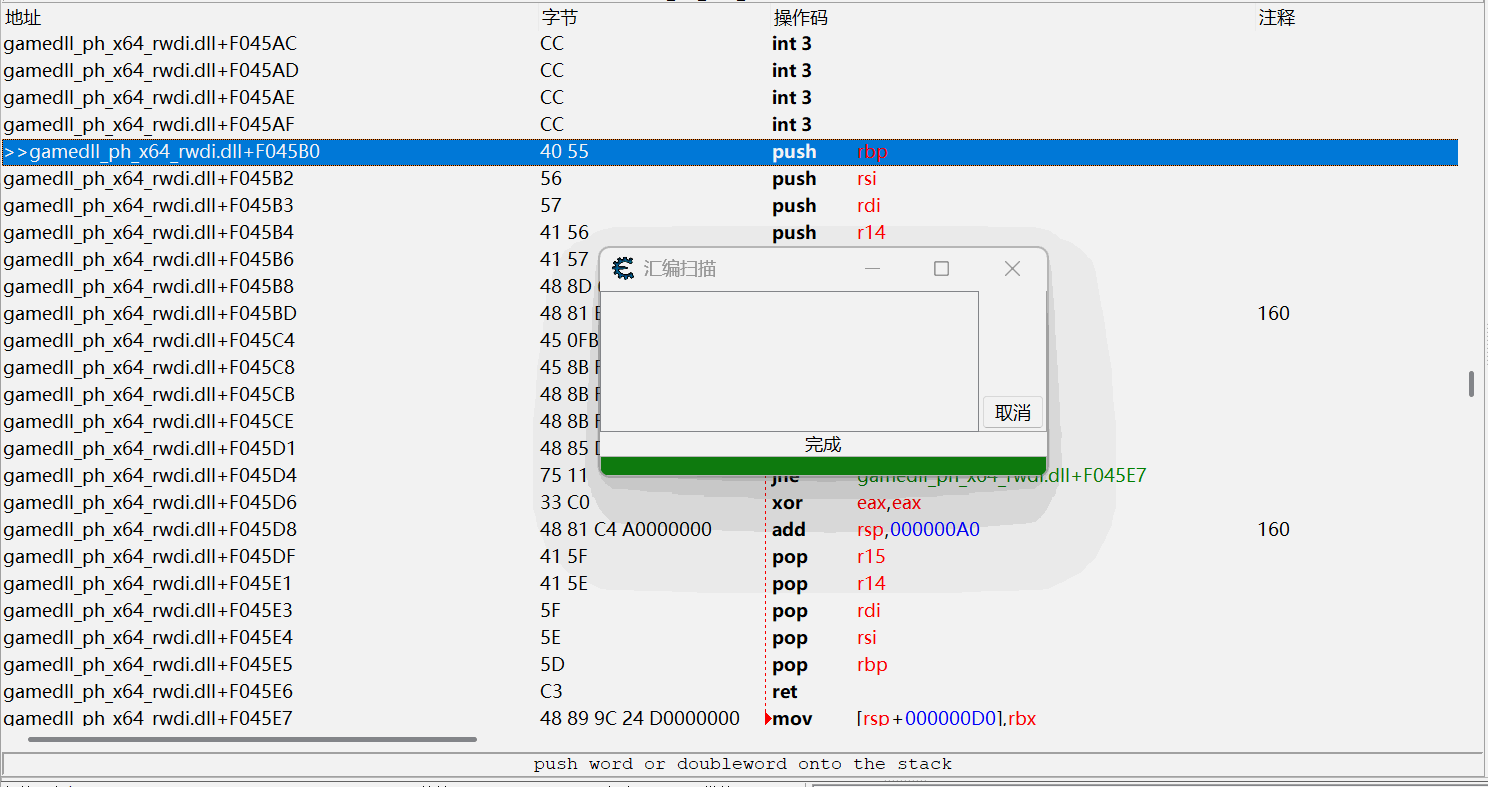


图2.3-使用第一种操作的指令进行通配符搜索的搜索结果

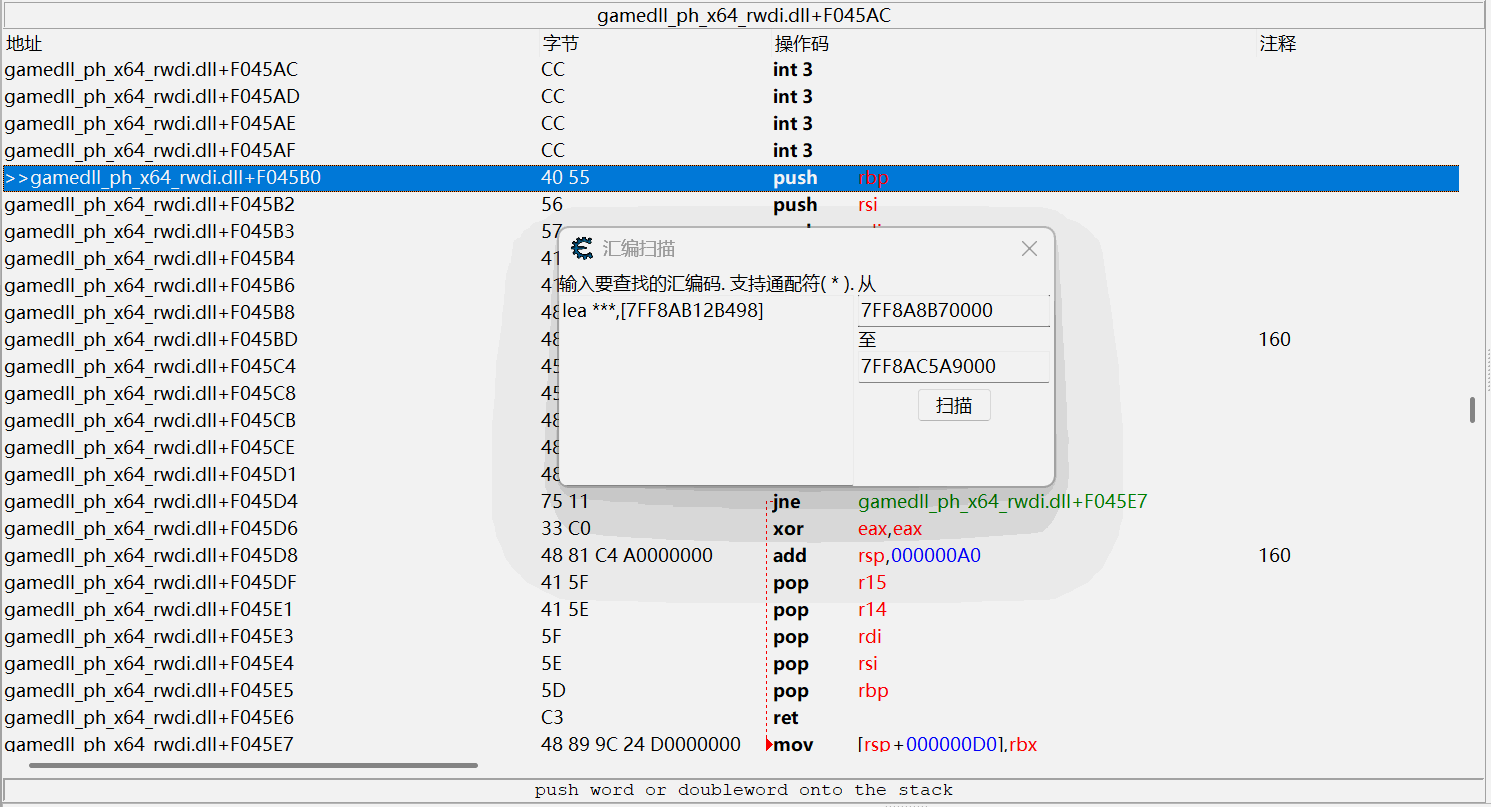


图2.4-使用第二种操作的指令进行通配符搜索

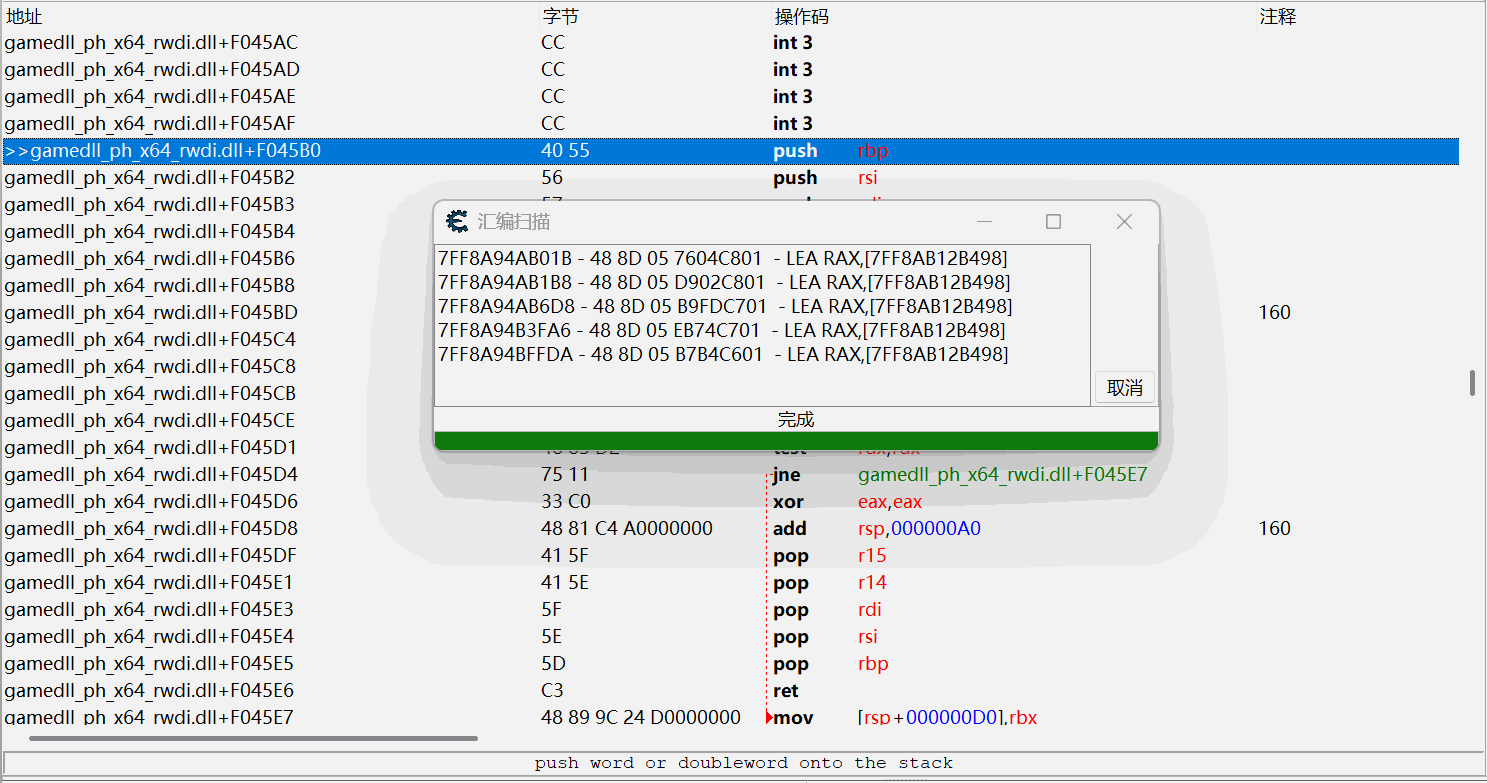


图2.5-使用第二种操作的指令进行通配符搜索的搜索结果

如图2.5所示，在使用第二种操作指令时搜索成功，得出如图所示的结果。在这里，有这么多的结果，哪一个才是呢？还是说全部都可以用？我们知道，在游戏程序调用增添物品函数时，出于参数的需要，肯定会调用对应的构造函数。只要我们到各个搜索结果对应的函数头部下断点，然后拾起地上的物品让游戏调用增添物品函数，哪里断下来哪里就是我们应要的构造函数，反正按照游戏程序自身的调用来行使肯定错不了。

但很不走运的是，在实际的测试当中，只要地上有物品，并且我们下了断点之后，不管有没有拾取物品，都会触发断点断下来，并且每一个搜索结果所在的函数都是如此，这就导致我们根本无法通过此方法确定。这时候就要我们另辟蹊径了。

我们知道在调用构造函数对类或结构进行初始化操作时，构造函数会对给予的内存空间进行写入的操作。基于此原理，我们就可以在内存空间上做标记来识别。想到拾取物品过程会调用对应的构造函数，而调用完成后可能会对初始化后的类或结构进行内容的写入。为防止写入覆盖的偏差，导致我们在构造函数中做的标记被覆盖掉，我们可以对虚表进行标记识别，因为正常的程序不可能会主动去修改虚表。那具体怎么操作呢？我们知道，虚表其实就是储存了一堆函数地址的固定的内存空间，而构造函数会将虚表的首地址写入类或结构的首位地址中。我们的目的是在类或结构中制造不同点，以便区分构造函数的调用。那我们就可以在构造函数中将lea xxx,[虚表地址]中的虚表地址进行修改，替换为我们自定义的地址。这样如果游戏在进行添加物品这个操作的过程调用到了这个构造函数，那么调用完后它对应的类或结构中写入的虚表地址也是我们自定义的地址。

为防止过程中游戏程序会调用其中的虚函数而导致异常，我们应复制一些虚函数到我们自定义的虚表中以便游戏的调用。比如复制前1000个函数地址到我们的自定义虚表中。在这里，需要注意的是，我们的自定义虚表所在的内存空间应与要替代的虚表所在的模块的内存空间相接近。下面就来构造我们的自定义虚表。

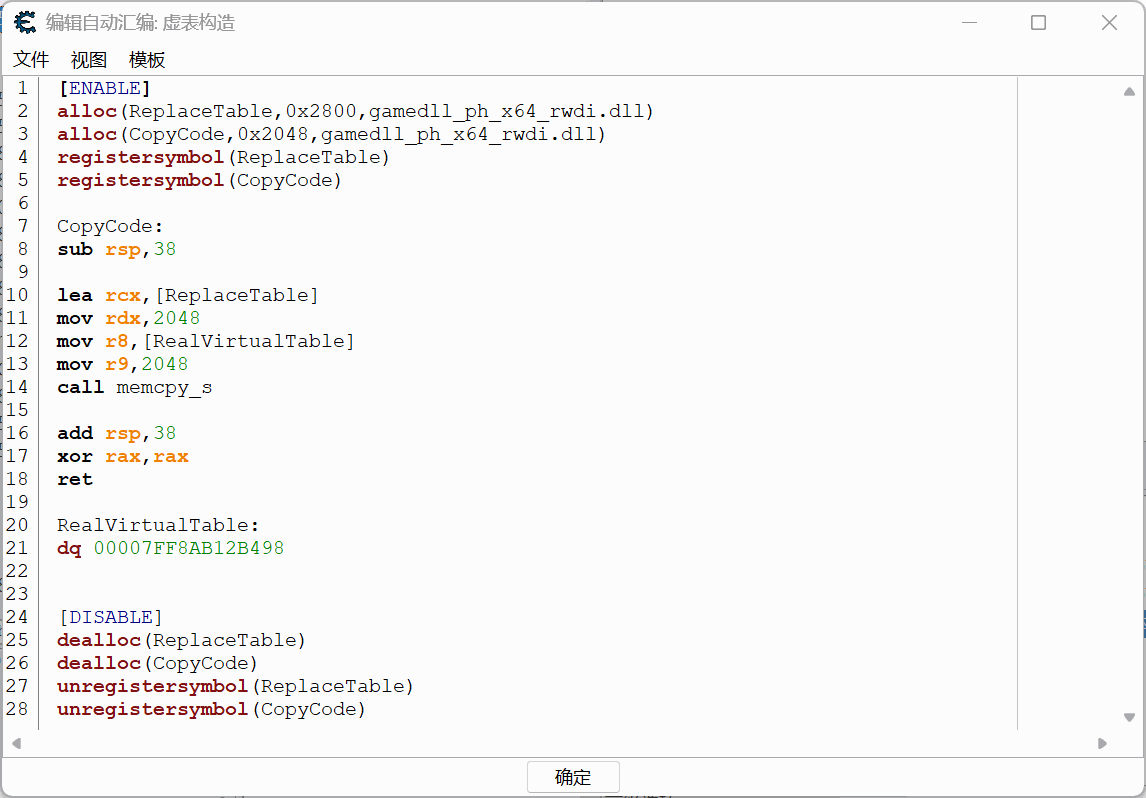


图2.6-运用汇编脚本辅助构造虚表

由于手动复制1000个函数地址需要花费我们大量的时间，因此在这里我们使用汇编脚本辅助我们进行地址的复制。如图2.6所示，在开始我们先申请两段内存，一段是我们的自定义虚表（ReplaceTable），一段用于储存我们复制虚函数地址的代码（CopyCode），并将它们的首地址进行全局符号注册。在CopyCode部分，我们调用memcpy\_s函数，将虚表中的虚函数地址复制到我们自主申请的一段内存ReplaceTable中。其中，memcpy\_s这个符号会被CE自动识别并转换为相应的地址，无需我们自行查找。memcpy\_s函数的参数依次是目标复制内存地址、目标空间大小、被复制内存地址、要复制的长度。这里我们将参数依次写入rcx、rdx、r8、r9中，然后直接使用call指令调用函数完成我们的函数地址复制操作。这里第二个参数的大小写成0x2048目的是为了限制写入的大小，同时申请时是申请了0x2800大小的空间，目的就是预留多余的空间以防意外的过度写入。为了给调用的函数预留寄存器保存空间，我们需要在调用call之前进行抬栈的操作，也就是开头的sub rsp,38。这里设置抬0x38个字节除了预留多的空间给被调用函数保存之外，还处理掉被调用函数中有要求内存十六进制对齐的指令，比如movaps指令。我们主动抬栈的0x38字节加上执行call指令时自动抬栈的0x8字节，恰好使得栈指针寄存器rsp的值的十六进制最后一位为0，能被16整除，贴合了被调用函数中有用栈空间作为变量的同时要求其十六字节对齐的要求。通常来讲被调用函数开头最多有4个寄存器保存操作。接着，在调用完函数之后，需要我们把栈指针寄存器恢复到原来的位置，也就是对应的反操作add rsp,38。最后，再设置返回值为0表示线程的正常退出。在这里我们直接对rax进行异或操作将其设置为0。之后我们激活脚本，来到反汇编界面，检查代码是否编译正确。检查无误之后，我们在我们写的代码头部地址使用工具中的创建线程，执行我们的代码，完成自定义虚表的构建。当然，也可以直接在脚本中使用CreateThread()函数在激活脚本的同时执行脚本的代码，前提是有充足的信心保证编译的代码不会出错。

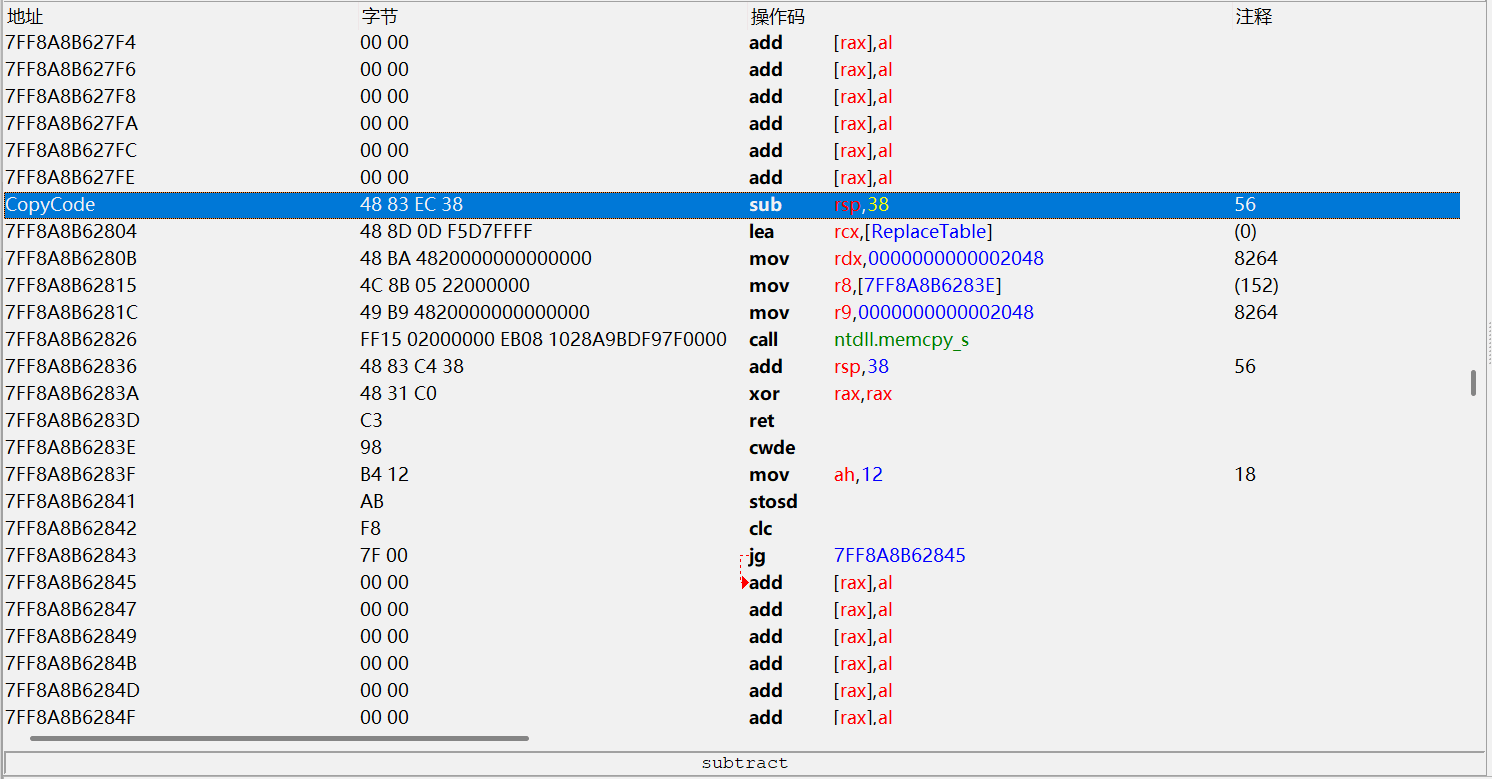


图2.7-检查编译后的代码，这是执行前的重要一步

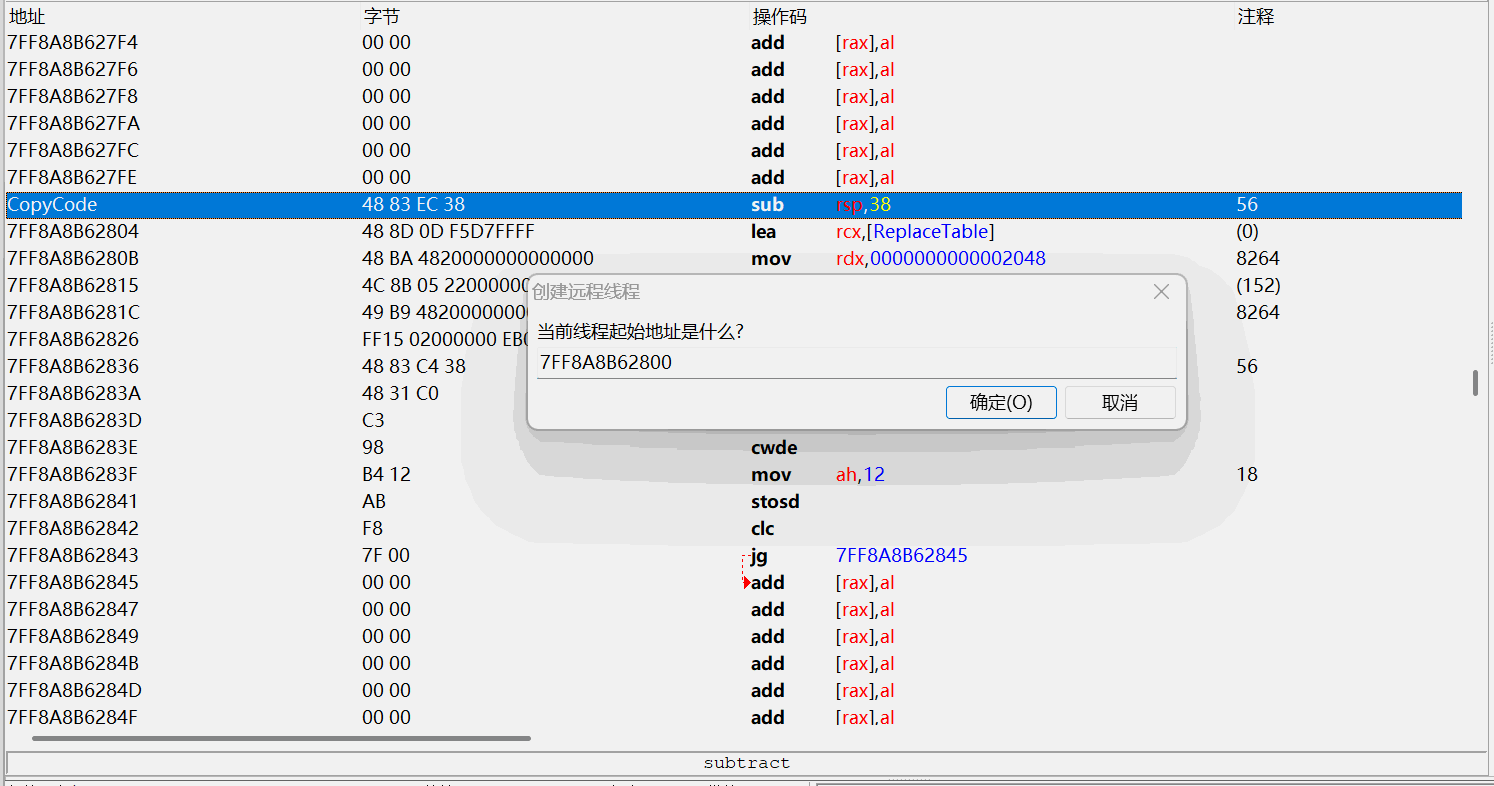


图2.8-在代码首地址创建线程执行代码

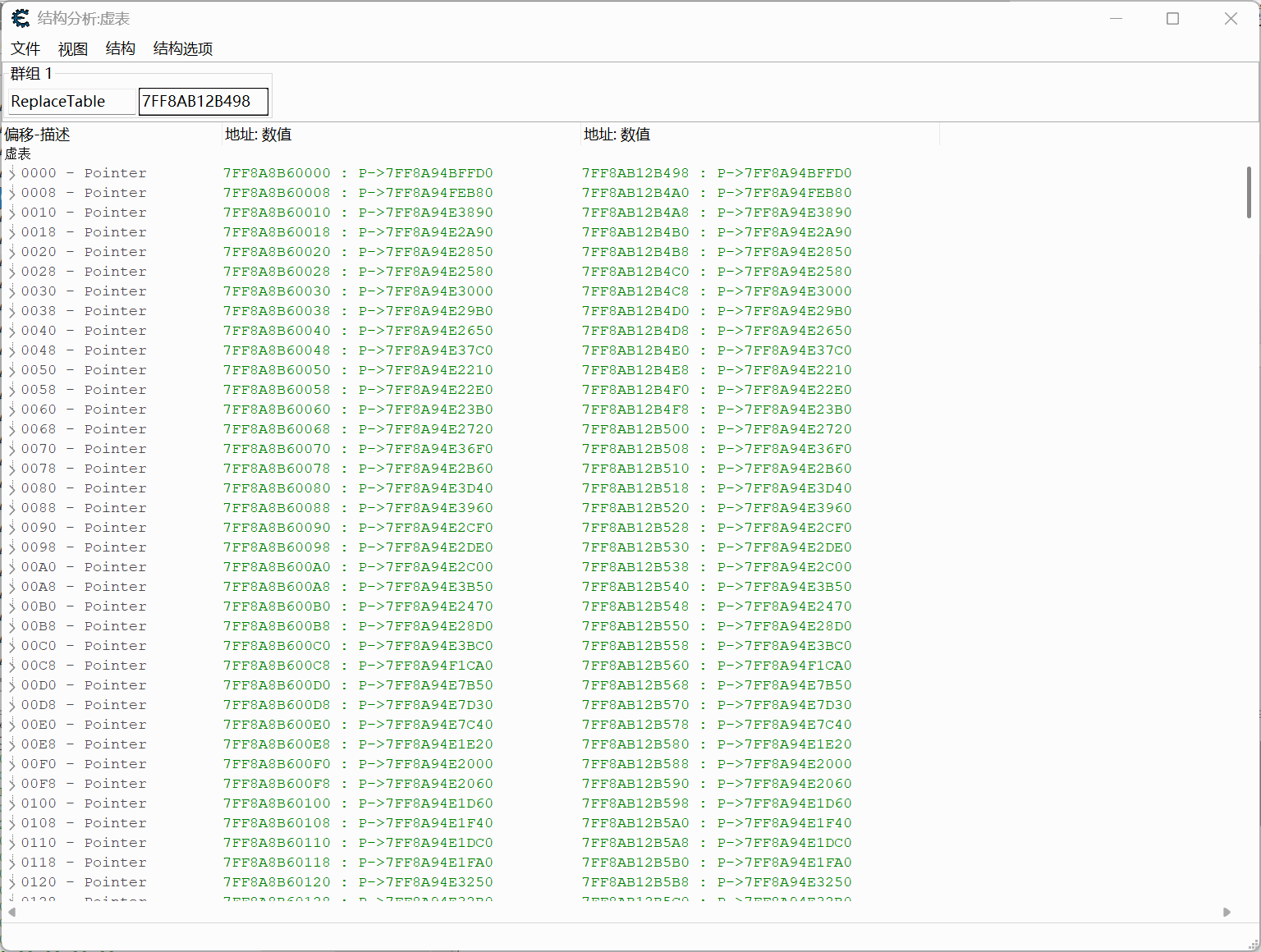


图2.9-执行完成后对构造虚表与原本虚表内容的比较

如图2.9所示，代码执行完毕之后，我们需要对写入内容正确与否进行检查。通过结构分析可以看出，一片的绿色，表明我们自主申请的内存ReplaceTable中的内容与原先虚表中的内容完全相同。接下来，我们就可以一个一个地到我们搜索到的构造函数中去把虚表给替换进行验证了。注意接下来我们不要取消激活脚本，否则我们构造的虚表会被释放掉，导致已经被改虚表的类或结构无法正常访问虚函数，进而导致游戏程序的异常或崩溃。

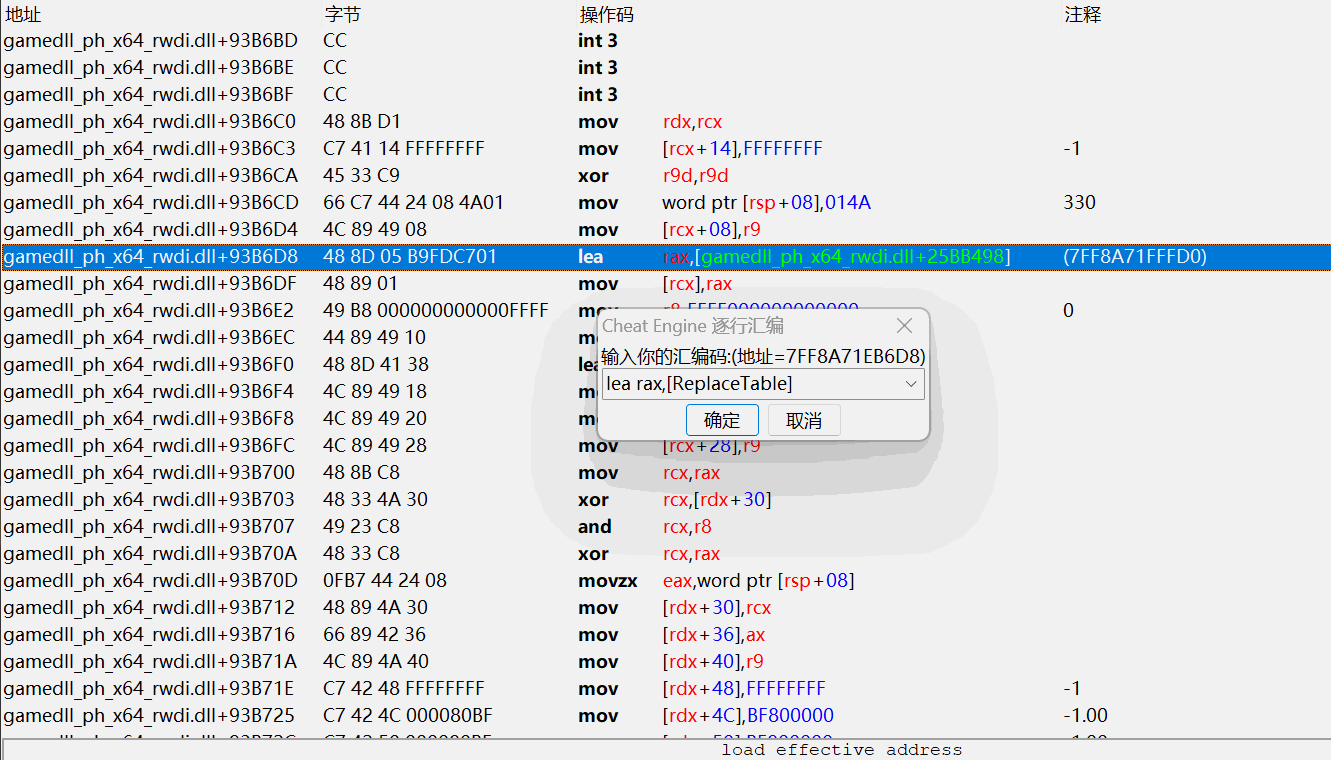


图2.A-修改在构造函数gamedll\_ph\_x64\_rwdi.dll+93B6C0中成功生效

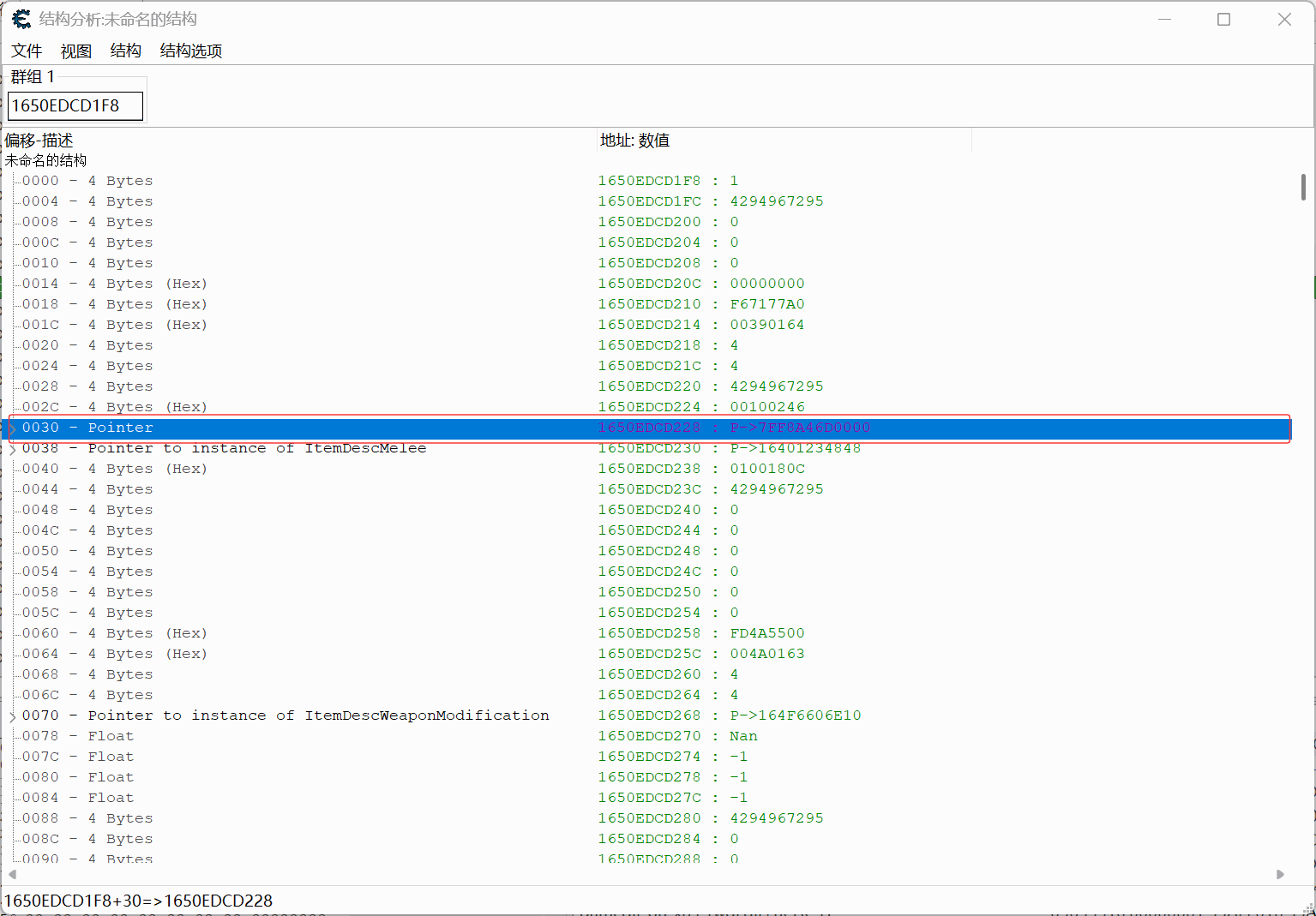


图2.B-虚表成功被替换为我们的自定义虚表

如图2.A和2.B所示，经过前两轮修改后，最终在第三处的构造函数gamedll\_ph\_x64\_rwdi.dll+93B6C0中修改生效。到这里，参数中类或结构对应的构造函数就被我们找出来了，最终确定为：gamedll\_ph\_x64\_rwdi.dll+93B6C0。

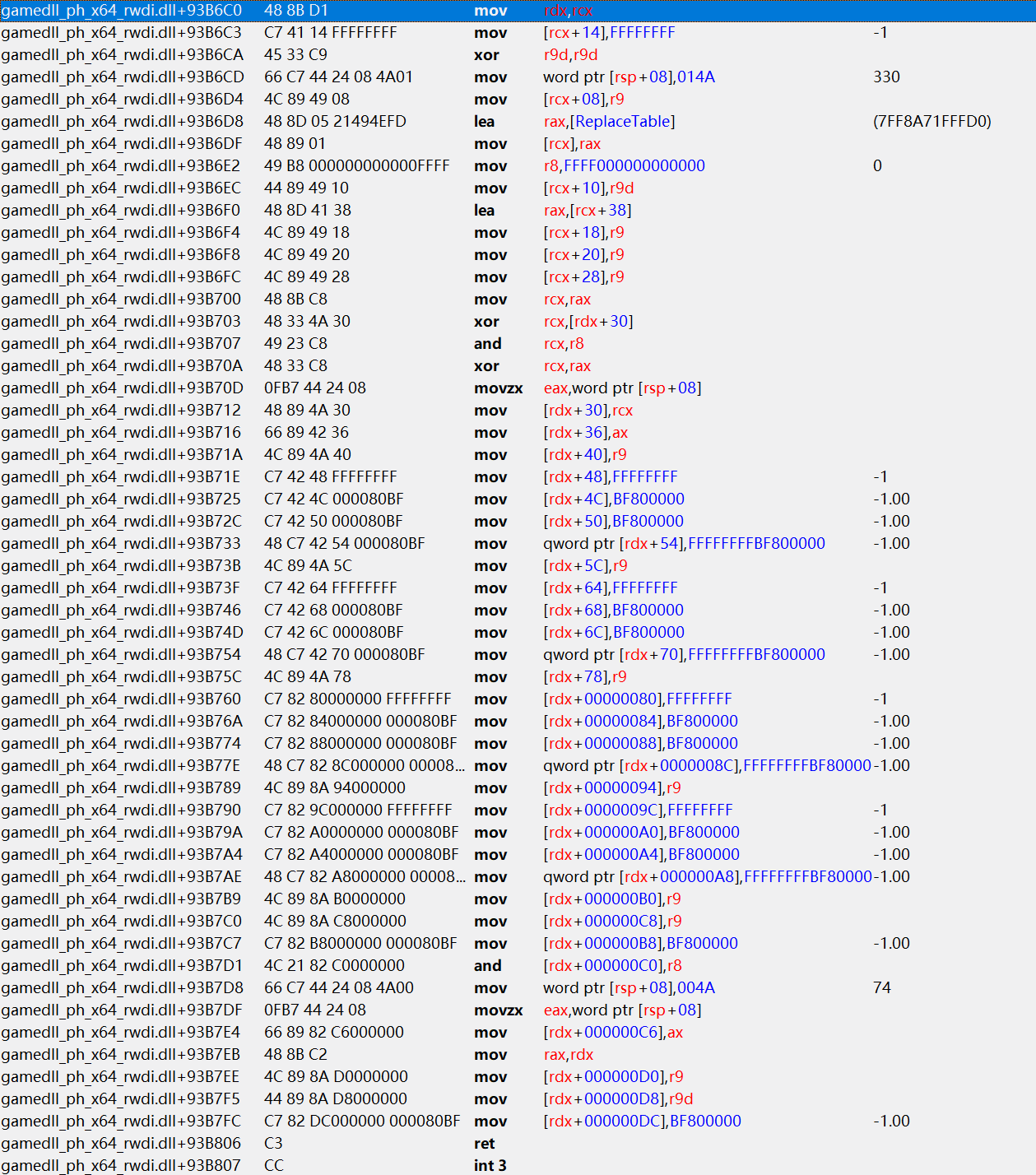


图2.C-构造函数中的全部指令

我们找到一个函数，就要分析其参数量和参数内容。即使是找到了目标的构造函数，我们也要找到其参数量和参数内容才能正常使用，就像我们分析我们正在分析的增添物品函数一样。如图2.C所示，函数一开头就有这样的指令：mov rdx,rcx、xor r9d,r9d，同时在执行这两条指令之前没有进行对rdx和r9的保存，即使这个函数有第二、第四个参数，也不可能使用，因为会直接被这两条指令擦写。这就表明了这个构造函数只有一个参数，也就是我们的类或结构的首地址，也同时表明其是一个默认构造函数。这样一来，就直接把构造函数的参数量和参数弄明白了。只要我们有需要，就直接拿一段内存空间出来，将其首地址作为参数直接调用就完成了对类或结构的初始化。由于构造函数没有返回值，这里直接忽略其返回值。

我们是找到构造函数了，但是这个构造函数里构造出来的内容只是我们整个参数内容的一部分而已，我们更希望找出一个能够构造整个参数结构的构造函数出来。我们了解到，上述所找的构造函数中构造出来的内容只是整个参数内容中的一部分。既然是这样，那么能够构造整个参数结构的构造函数中就应该会调用此构造函数对整个参数内容进行构造。这样一来，我们就可以去查看调用方，看调用方调用完我们上述所找的构造函数后其内容结构是否与我们参数的一样。



图2.C（继）-能构造整个参数内容的构造函数的调用方

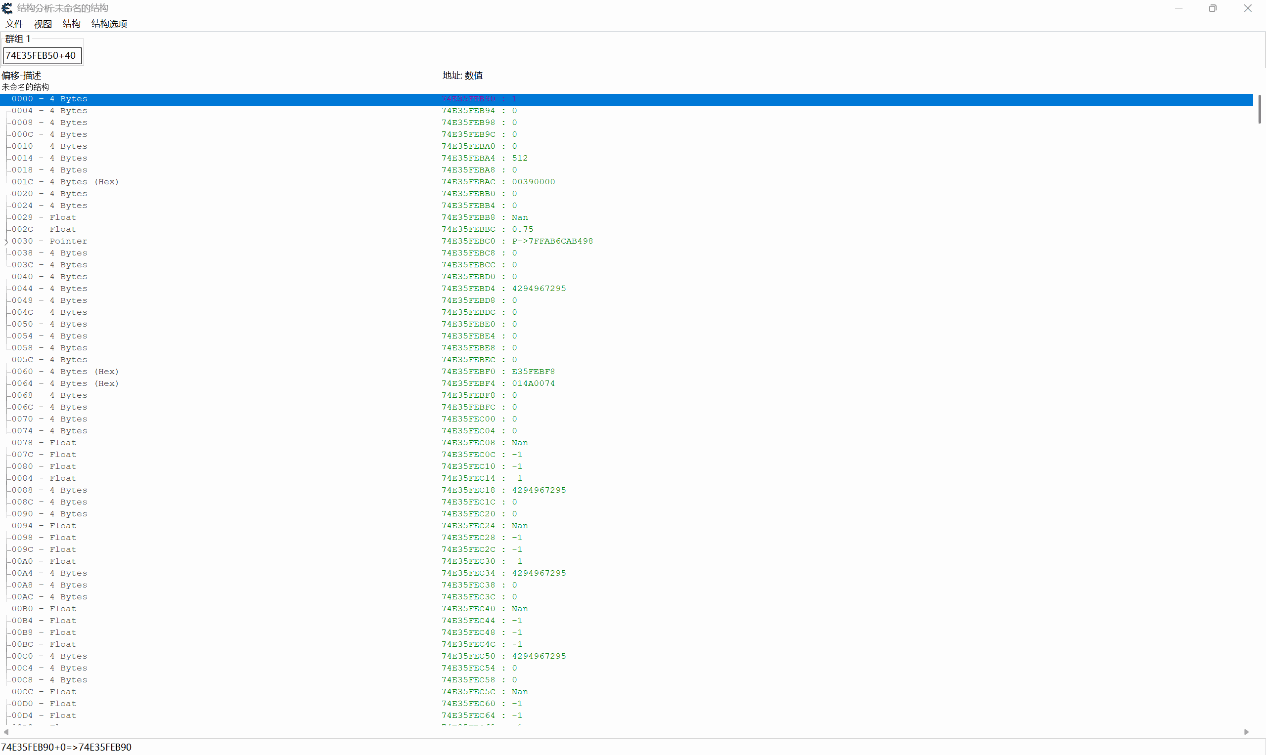


图2.C（续）-能构造整个参数的构造函数构造的内容

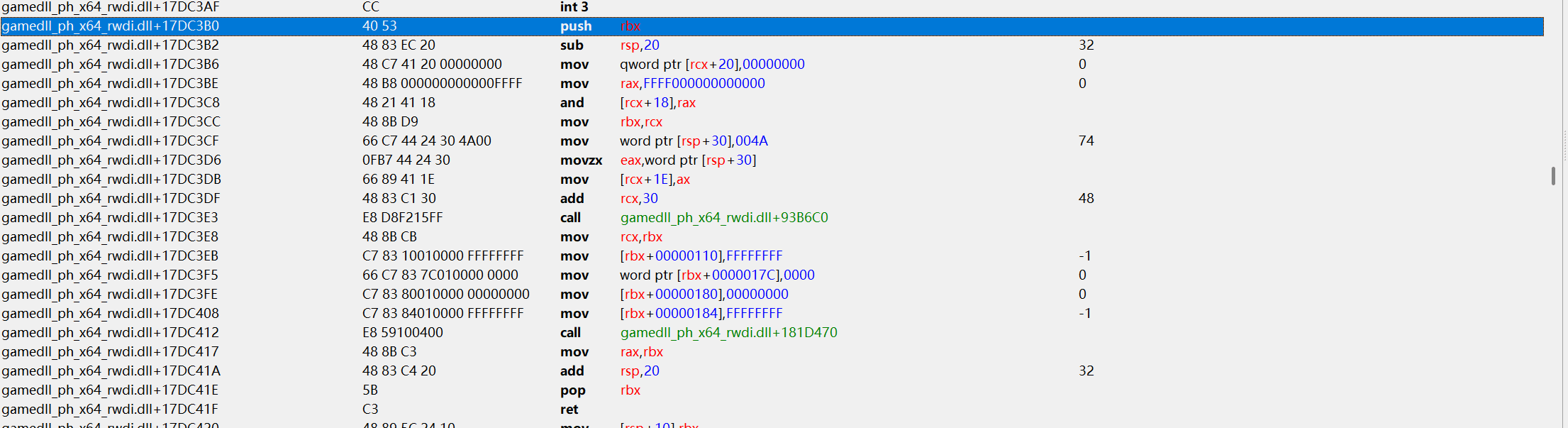


图2.C（续）-能构造整个参数内容的构造函数的全代码

如图2.C（续）所示，我们在函数gamedll\_ph\_x64\_rwdi.dll+17DC3B0处断下。可以发现，这个函数所构造出的内容与参数的内容一模一样，同样都是首位为物品数量，偏移0x30处为一个类或结构，且是同样的类或结构。同时，如全代码图所示，里面同样也调用到了我们上面所找的构造函数。这里，我们就可以将函数gamedll\_ph\_x64\_rwdi.dll+17DC3B0定为我们真正能构造参数结构的构造函数。同样地，在这个构造函数的调用方处可以发现，其只有一个参数rcx，也就是用于构造的内存空间。一般构造函数没有返回值，在这里我们也同样忽略其返回值。

解决了对类或结构的初始化操作后，现在就要去了解类或结构里面的内容，以便我们自定义里面的内容。如图2.C所示，一眼望去，全是对结构内容的写入操作。就像我们上面所说的一样，从写入的数据类型和大小就可以看出其对应的成员的类型和大小。如右侧注释中表明的数据，有的写入-1，有的写入-1.00，这就表明了其对应的数据类型是4字节整形和4字节单浮点型数据。又如直接将rcx、r8、r9写入进去的，这就表明其对应的数据长度是8个字节。由于存在大量这种操作，且图中有所体现，这里就不一一细讲出来。从最后的写入操作mov [rdx+DC],BF800000中就能看出来，这个类或结构的实际长度是0xDC+0x4=0xE0（十进制为224）字节。因为写入的偏移最大值是0xDC，表明了这里是最后一个成员所处的位置，同时又向此处写入了4字节数据，因此整个类或结构的长度就是224个字节。当然，这个长度只是参数中一个类或结构的长度，完整的参数的内容长度可以看到图2.C（续）所示，最后的写入指令中，也就是mov[rbx+184],FFFFFFFF，也可以代表了整个参数的长度，也就是0x184个字节的长度。由于写入的数据除了虚表以外，基本就只有0、-1、-1.00，根本看不出其位置的成员代表了什么。所以，接下来我们就要在这个类或结构中去修改，在游戏中体现出来并确定其代表的含义。

这里我们先重启游戏，让被我们替换了虚表的类或结构恢复正常，继续按照游戏程序提供的数据进行分析。我们回到增添物品函数的头部，断点，继续分析偏移0x30以下的内容。这里我们捡起的是方天画戟。



图2.D-方天画戟武器属性图

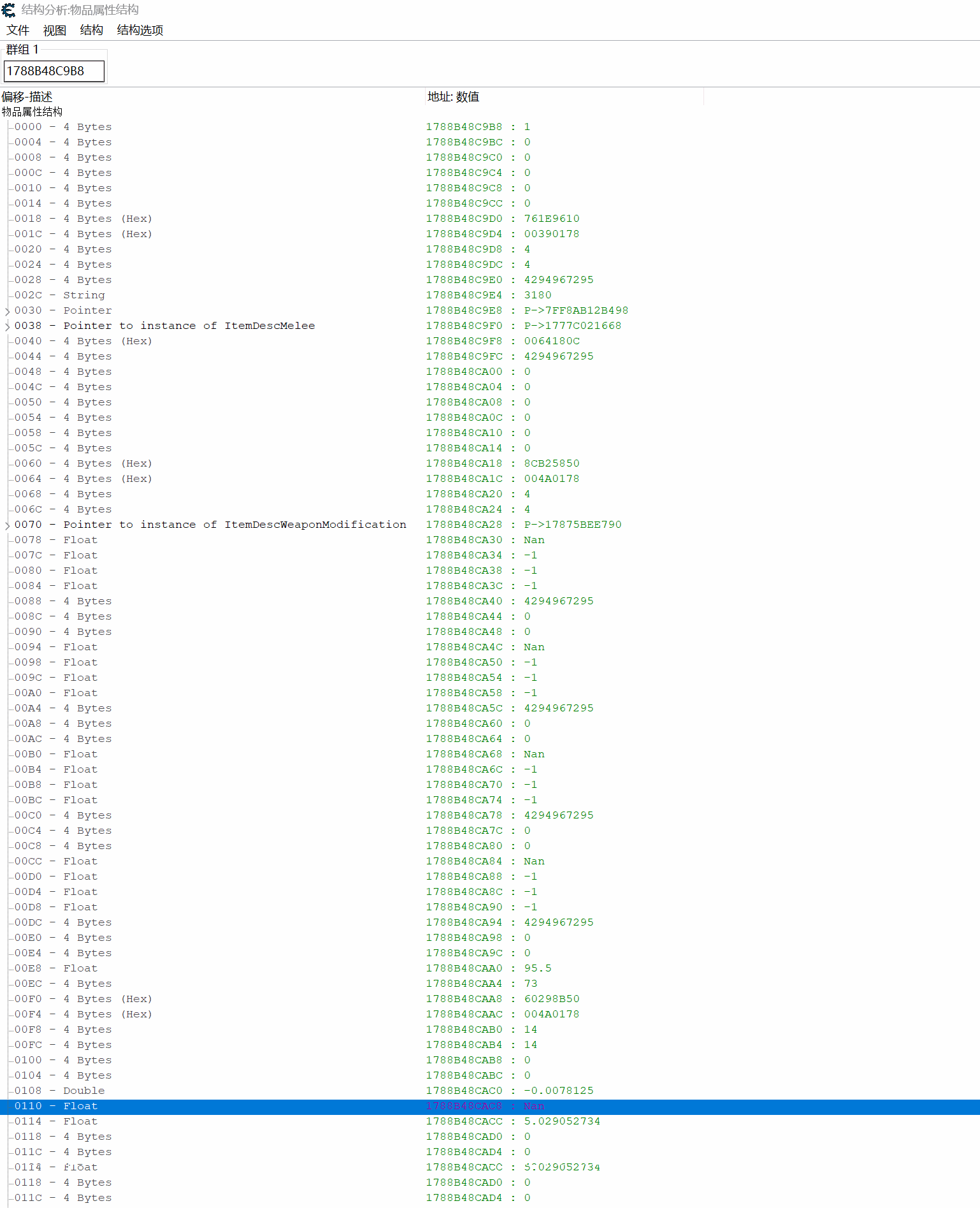


图2.E-方天画戟属性结构内容（rdx参数内容）

上面我们分析到，从偏移0x30开始的类或结构的长度是224个字节（0xE0），那我们就应该止步分析于0xE0+0x30=0x110偏移处，如图2.E所示。我们来看到偏移0x38处，这里是一个指针，指向一个实例。我们先展开其内部结构，观察内部有些什么东西。

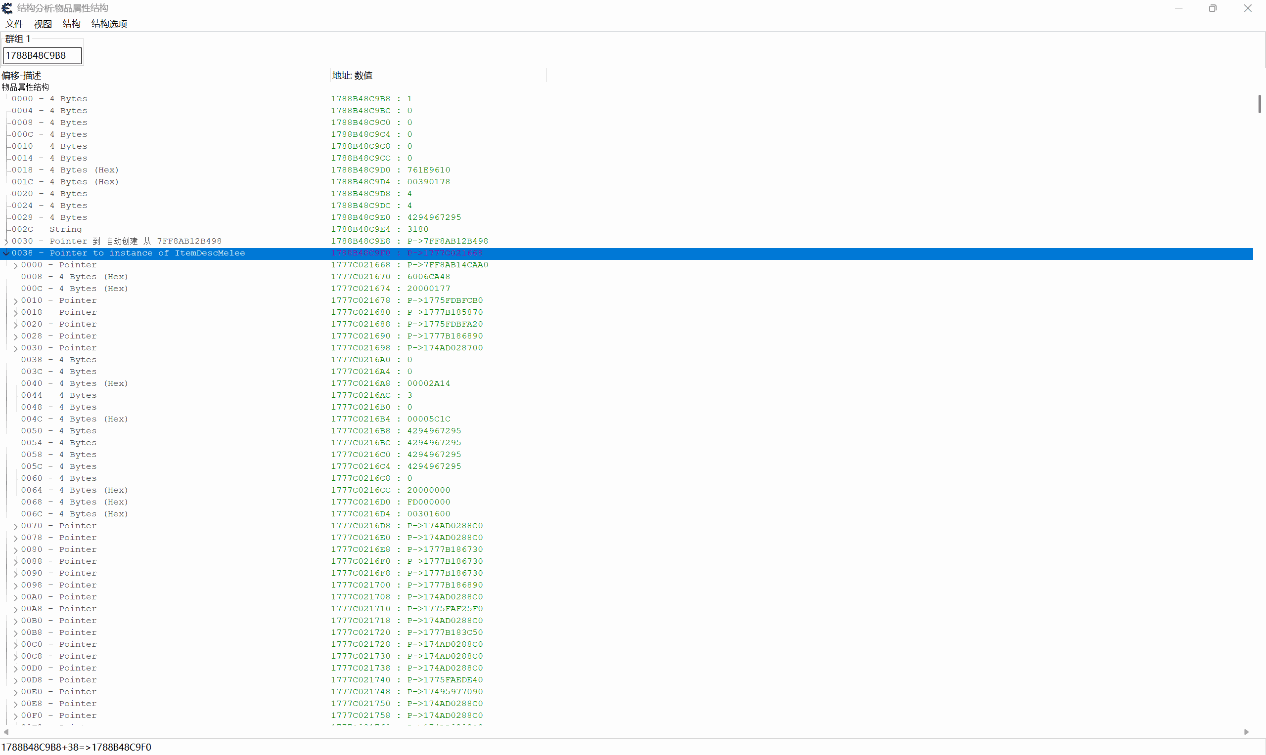


图2.F-在偏移0x38处的结构展开

如图2.F所示，在展开结构中的偏移0x0处，又出现了虚表，表明了此处又是一个类或结构。我们的第一反应可能是按照上面所说的方法去找到这个类或结构，然后想着就可以通过构造函数了解其中的内容。但是，如果这里面又有别的类或结构呢？那岂不是又要重复上述的操作去执行？什么时候才会有尽头呢？因此，我们在这里应先暂停下来思考一下有没有别的处理方法。我们可以想到，在地面上的物品的属性和已经被我们拾取进入背包的物品属性别无二致。同时，在我们进入游戏的存档后，物品的属性就已经定下来我们无法修改（除非自行对物品进行改造升级之类的）。那么，由此推测，游戏中的物品属性结构会不会早在游戏启动或进入存档时就已经创建好并保存在某处了呢？如果真是这样，就像这里的类或结构，早已经在游戏程序中创立完成，那我们就可以去寻找这些创立好的储存着有关物品属性的类或结构，接着将它们当作参数写入结构中即可。但换另一个角度想一下，我们是要刷取游戏中的物品，是游戏中存在的，而不是要去创造一个新的物品出来，这就要求我们必须找出有关储存物品属性的内容，也就是向上面所说的一样要我们找出相关的类或结构以此来当作参数调用增添物品函数（当然你也可以想着要创造一个游戏中不存在的物品，接着去寻找构造函数，真正地构造一个自定义物品的类）。但猜测始终是猜测，需要我们去验证。这里我们尝试去寻找它们储存的位置。

在实际的编程中，如果我们遇到很多的对象，我们通常会将它们放置到一个列表中去，这样便于管理。同样地，这游戏中有很多的物品，意味着有很多的对象，那它们也应被安排在一个列表中。既然是被安排在一个列表中，那我们的目标就是找到这个列表，而且是列表的首地址。我们现在有列表中的一个成员，也就是图2.F中偏移0x38中的类或结构。那我们就可以直接按8字节16进制搜索这个类或结构的地址，得到储存这个类或结构首地址的内存地址。在这些内存地址中，也一定包含了列表的地址，但不一定是其首地址。

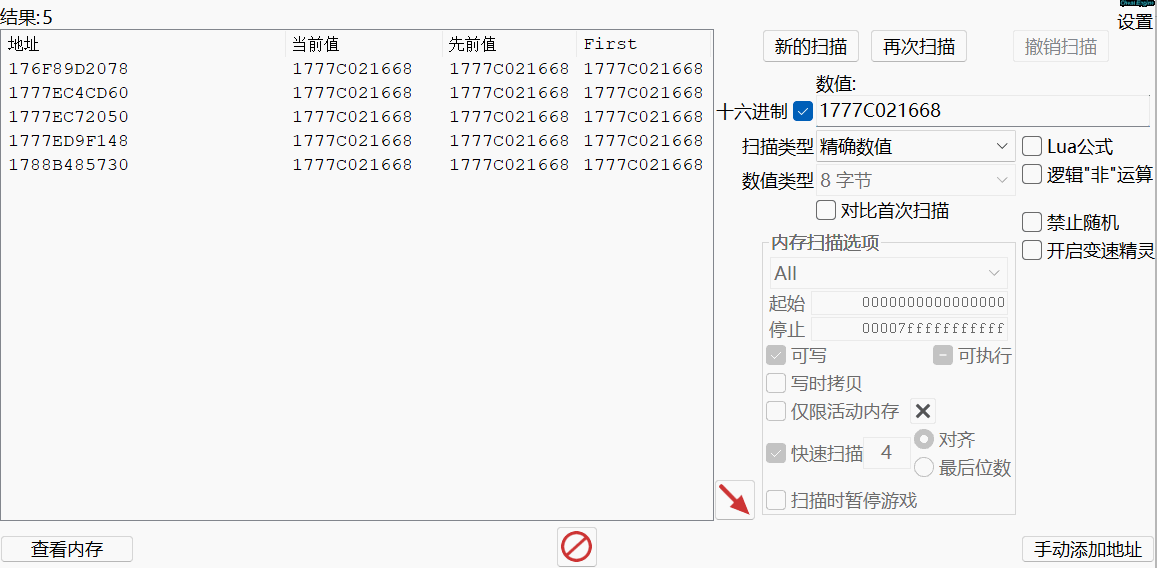


图2.F.1-搜索储存物品属性的类或结构的列表地址

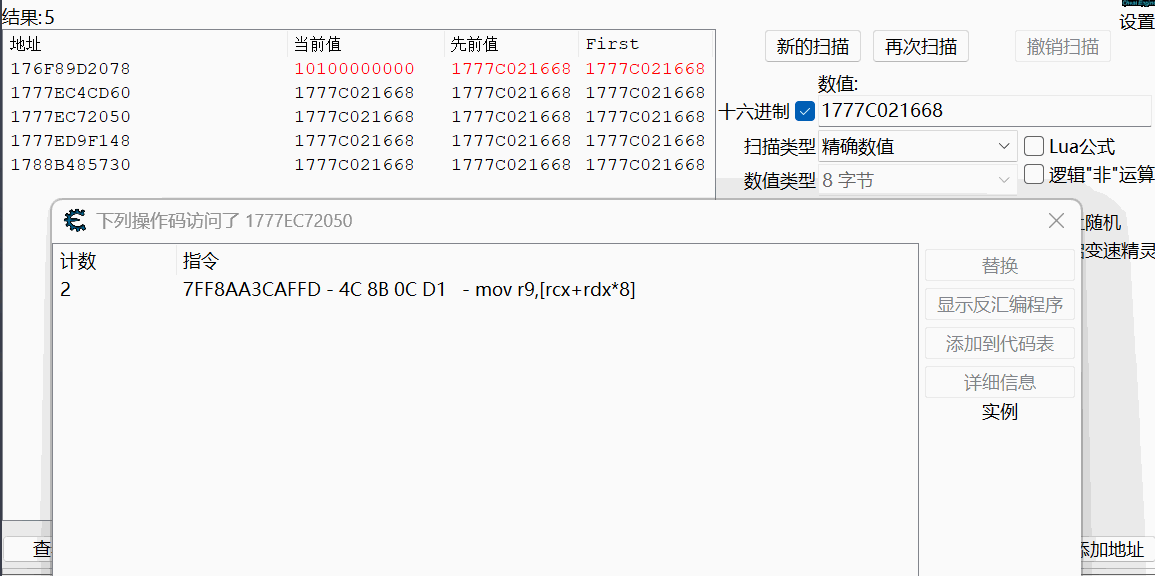


图2.F.2-在访问地址0x1777EC72050代码处发现类似数组查找的指令

如图2.F.1所示，我们寻找到了多个保存方天画戟物品属性地址的指针。为寻找列表地址，我们可以现在游戏中把方天画戟给丢弃在地上，然后从第一个地址开始寻找是什么代码访问了此地址，之后拾取地上的物品，留下有类似遍历和数组访问的访问代码。因为在拾取的过程中，游戏程序可能会访问到这个列表同时访问到储存方天画戟属性所在的列表地址，那我们就可以照上述方法找到访问列表地址的代码，从而进入代码所在的函数中找到列表的首地址。正因为在拾取过程中游戏程序可能会访问列表地址，所以也能推出其可能会遍历列表，遍历列表必然存在类似遍历或数组访问的代码，比如 mov xxx,[寄存器1+寄存器2\*常量]、lea xxx,[寄存器1+寄存器2\*常量] 等等，所以我们需要将访问代码中含有以上类似指令的给保留下来并进入到其所在的函数中进行分析。如图2.F.2，在拾取方天画戟并监视搜索到的第三个地址0x1777EC72050中，出现了这样的代码：mov r9,[rcx+rdx\*8]，表明了其可能是遍历或者是数组访问，那我们就可以进入其所在的函数中去分析。同时，在监视其他的4个地址中并无出现类似的访问代码，即保留此处的代码即可。

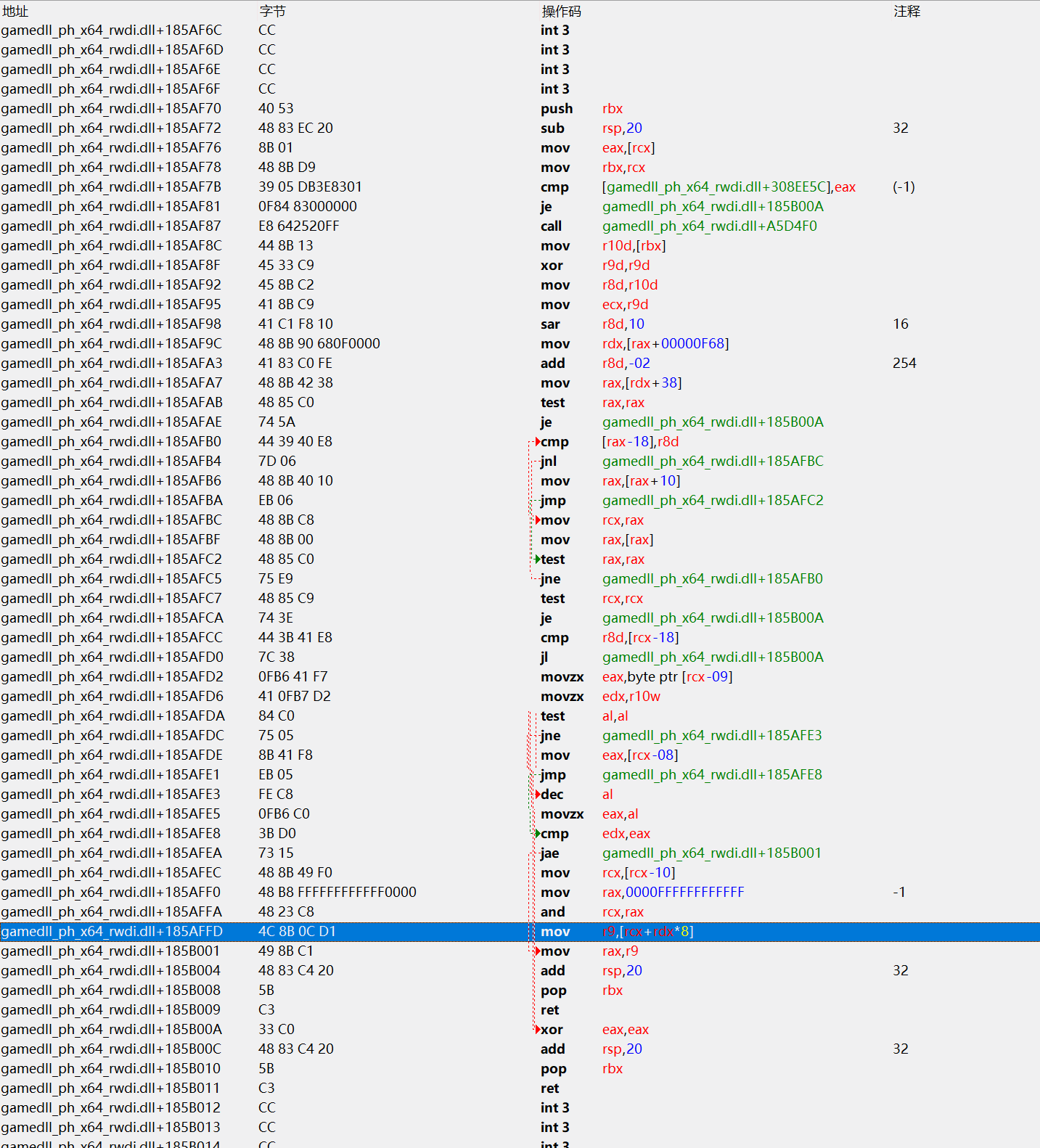


图2.F.3-遍历或数组访问代码所在的函数

如图2.F.3所示，通过指令mov r9,[rcx+rdx\*8]可知，列表的首地址就保存在寄存器rcx中，因为一个指针占据了8个字节，正好对应了指令中的乘以8的操作。只要通过改变rdx的值，就可实现对列表的特定位置访问和遍历。我们把方天画戟丢弃在地上，直接在此处下断点，再拾取方天画戟，把rcx的值取出来，对其进行结构分析。

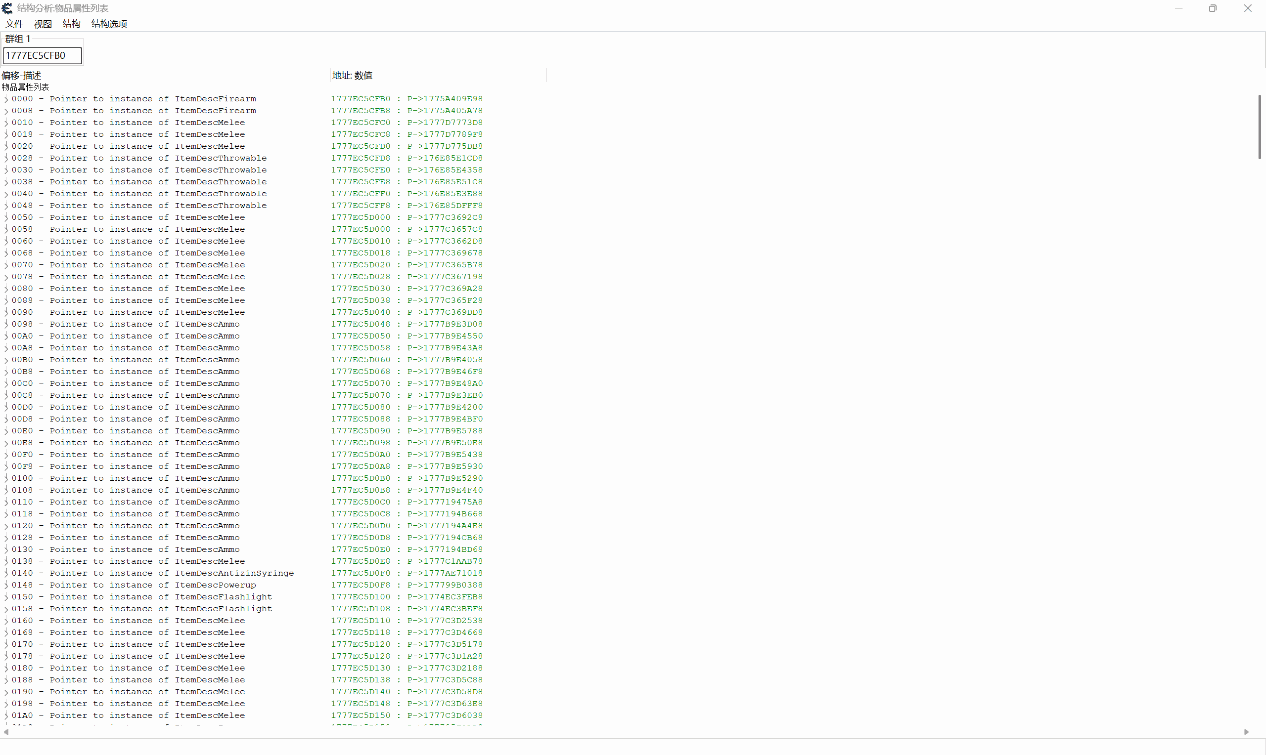


图2.F.4-断点取得rcx的值后对其进行分析得到的结构

如图2.F.4所示，列表中整整齐齐地从上往下排列了指向各种物品属性的指针。同时，从解析出来的名称中可以了解到，里面包含了近战武器、投掷物、火器、子弹等等的物品，表明了其覆盖范围的广泛，可以说其覆盖了游戏中所有的物品。那现在我们的目标就是找到获得列表首地址和物品数量的方式。如图2.F.3所示，我们在地址gamedll\_ph\_x64\_rwdi.dll+185AFFD往上查看，观察有什么指令对寄存器rcx进行了修改操作。首先就是有指令and rcx,rax，同时rax的值是0x0000FFFFFFFFFFFF，在往上就是mov rcx,[rcx-10]。为什么这里要对rcx进行and操作？为什么不在执行完mov指令后就直接使用而是要进行and操作才去使用？这个and操作的意义是什么？这个很有可能是一个带有标记的指针，从and操作来看，会将mov操作完后取出来的地址进行高16位的清零操作，然后才得到正确的地址进行访问。这样的操作很大可能是游戏程序故意对指针进行这样的操作：往指针指向地址的高16位中写入一些东西，完成对指针的加密操作，使得我们不能通过内存搜索的功能搜索到储存此地址的指针，同时也无法直接从基址加上偏移直接读取到。要引用这个指针，就要对其指向地址的高16位进行清零操作，也就是要执行上面的and操作。我们可以直接搜索此地址来验证，结果很大概率是搜索不出任何的结果。

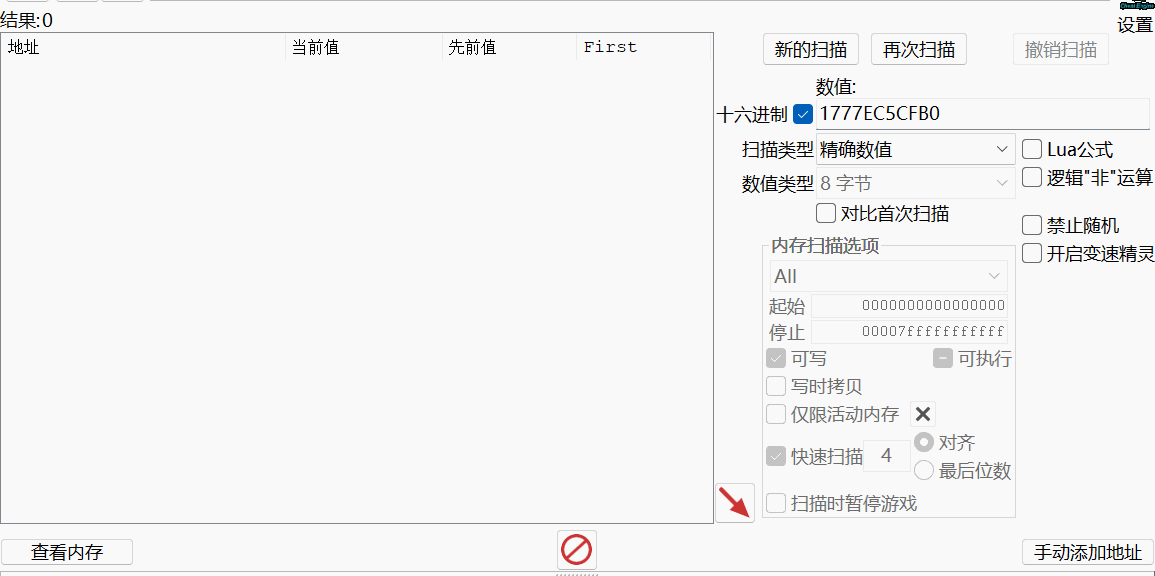


图2.F.5-直接搜索列表首地址无法搜索出任何结果

如图2.F.5所示，直接搜索是搜索不出任何的结果的，也同时表明了找基址和偏移的困难，更别想利用指针扫描工具找出来了。但另外一种可能就是这是一个被标记的指针，而不是加密。游戏程序可能认为普通的内存申请不可能申请到48位长这么大的内存空间，从而在高16位上对指针指向的地址写入别的东西进行标记以便用来识别指针的类型，需要引用的时候再进行处理。但在处理代码的附近看不到任何对标记进行读取和识别的操作，这种可能性很低。

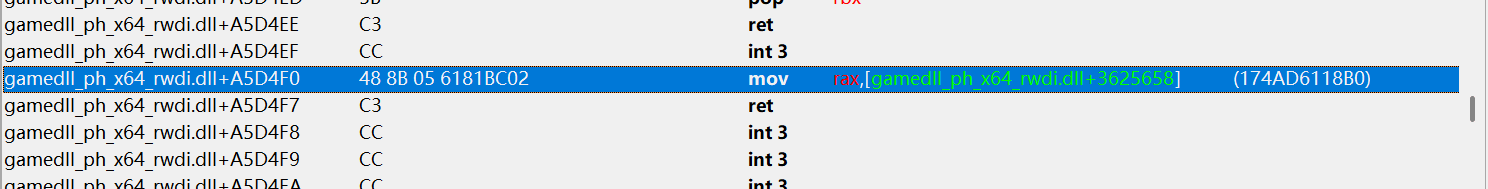


图2.F.6-地址gamedll\_ph\_x64\_rwdi.dll+185AF87出调用的函数

到这里，获取列表首地址的方法就只有进入到这个函数中去继续分析了。上面我们分析到，只要追踪修改rcx的指令就可以了。如图2.F.3，往上看，修改rcx的还有地址gamedll\_ph\_x64\_rwdi.dll+185AFBC的mov rcx,rax。这里又要追踪rax，看有无修改rax的指令，往上就有地址gamedll\_ph\_x64\_rwdi.dll+185AFA7的 mov rax,[rdx+38]，同时继续往上追踪rdx，寻找修改rdx的指令，就有地址gamedll\_ph\_x64\_rwdi.dll+185AF9C的mov rdx,[rax+F68]，再往上追踪，就没有修改rax的指令，只剩一个call指令调用函数了。而rax作为保存返回值的寄存器，且在call之上再也没有被修改的指令，必定在上面的地址gamedll\_ph\_x64\_rwdi.dll+185AF87执行的call指令中被修改。如图2.F.6所示，这里直接把基址中的指针读出来给rax当作返回值。到这里，我们的列表首地址寻找就结束了，只要我们像函数中的代码那样操作（可以说直接把代码直接抄下来），就能直接得到列表首地址。至于列表成员的数量，只要我们在最后取得列表地址的代码附近找找就能得到。因为通常来说，列表成员的数量应保存在保存列表首地址地址的附近。在这里，就比如地址gamedll\_ph\_x64\_rwdi.dll+185AFEC和地址gamedll\_ph\_x64\_rwdi.dll+185AFDE，它们所在的指令分别是mov rcx,[rcx-10]和mov eax,[rcx-8]，它们的读取偏移只差了8个字节，并且地址gamedll\_ph\_x64\_rwdi.dll+185AFDE处读取出来保存的寄存器是eax，为四字节读取，正好对应常用的整形。要验证此处，我们只要在此处监视其访问的地址并将其地址中的数值乘以8当作偏移加到列表首地址上，看加了偏移的列表后面是否还有储存指向物品属性的指针。如果没有，且刚好对应最后一个指向物品属性的指针，那这里就是储存着物品列表中的成员数量。

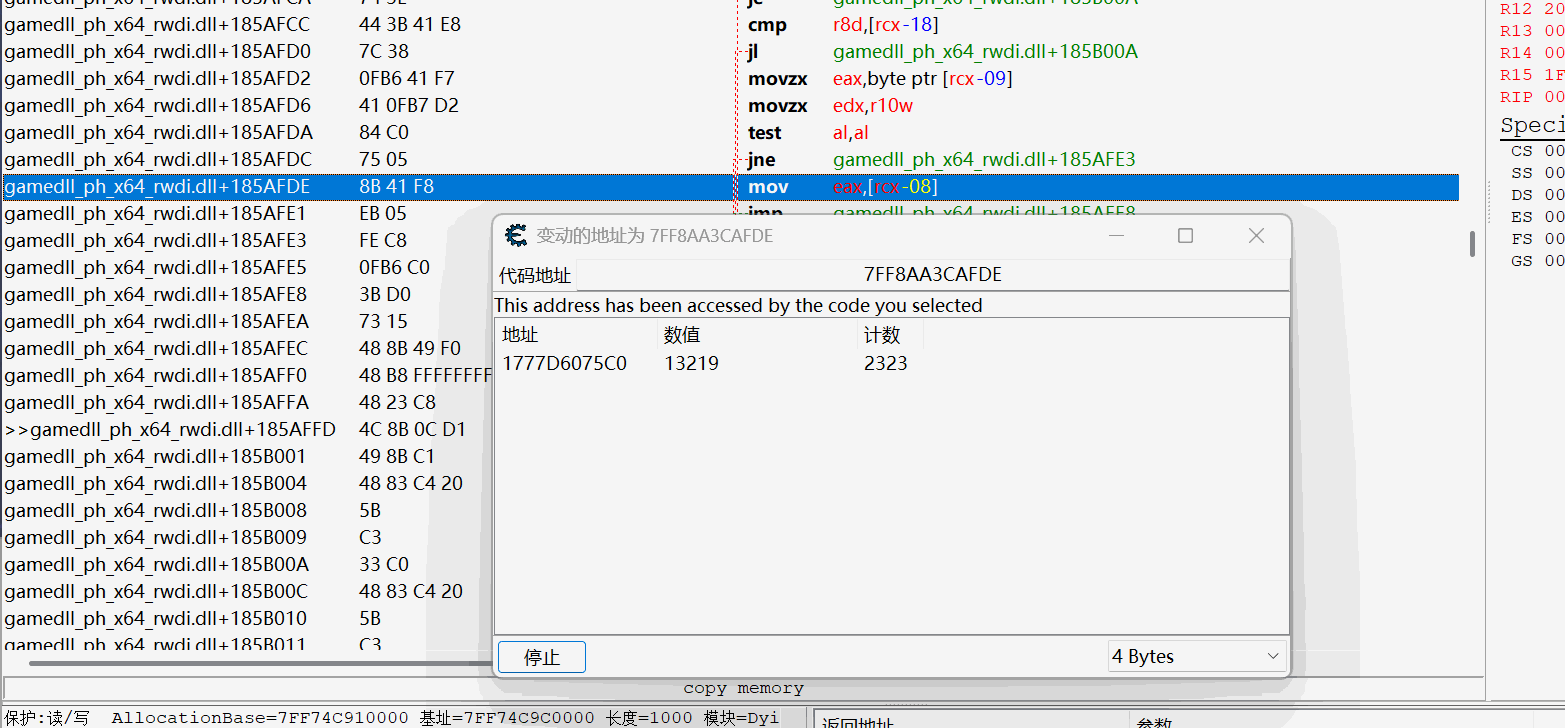


图2.F.7-监视疑似访问列表成员数量的代码处

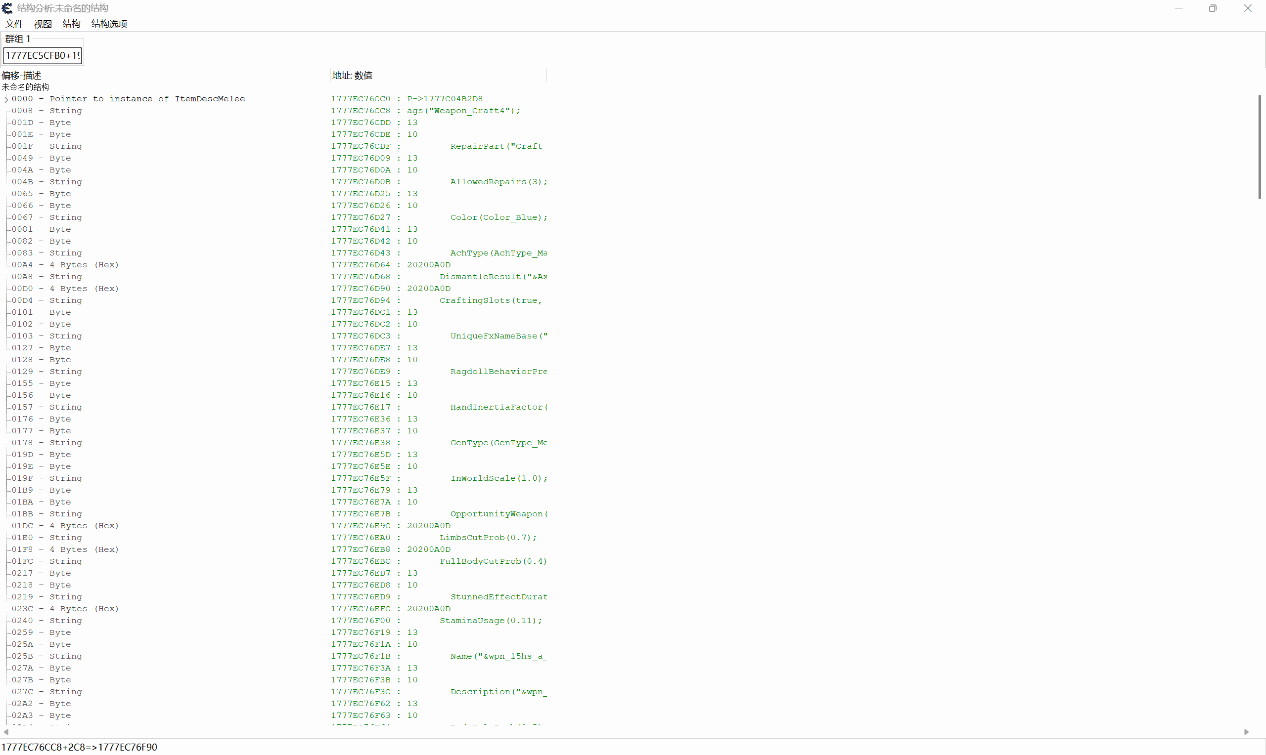


图2.F.8-得到成员数量乘以8加上列表首地址，刚好到列表尾部

如图2.F.8所示，在取得有13129个成员数量后，我们将其乘以8，得到十六进制偏移0x19D18，为能看到最后一个成员，我们将其减去0x8，得到用于验证的偏移量：0x19D10，同时加上列表首地址0x1777EC5CFB0得到0x1777EC76CC0，恰好能看到最后一个成员，再往后就没有了。在这里，我们就得到了列表成员数量及其获取方式。至于其获取方式，就跟获取列表首地址是一样的，因为它们两个之间的读取位置就只差了8个字节而已。至此，我们就得到了获取储存所有物品属性的列表首地址及其成员数量的方式。在后面正式写脚本的时候，只要我们按照如图2.F.3中的函数中的指令来获取即可。在完成这一切之后，回过头来验证我们最初的猜想：这一切是否早在我们进入存档之前就已经创建完毕？现在，我们重启游戏，来到主界面，按照代码的方式去获取列表地址及其成员数量。

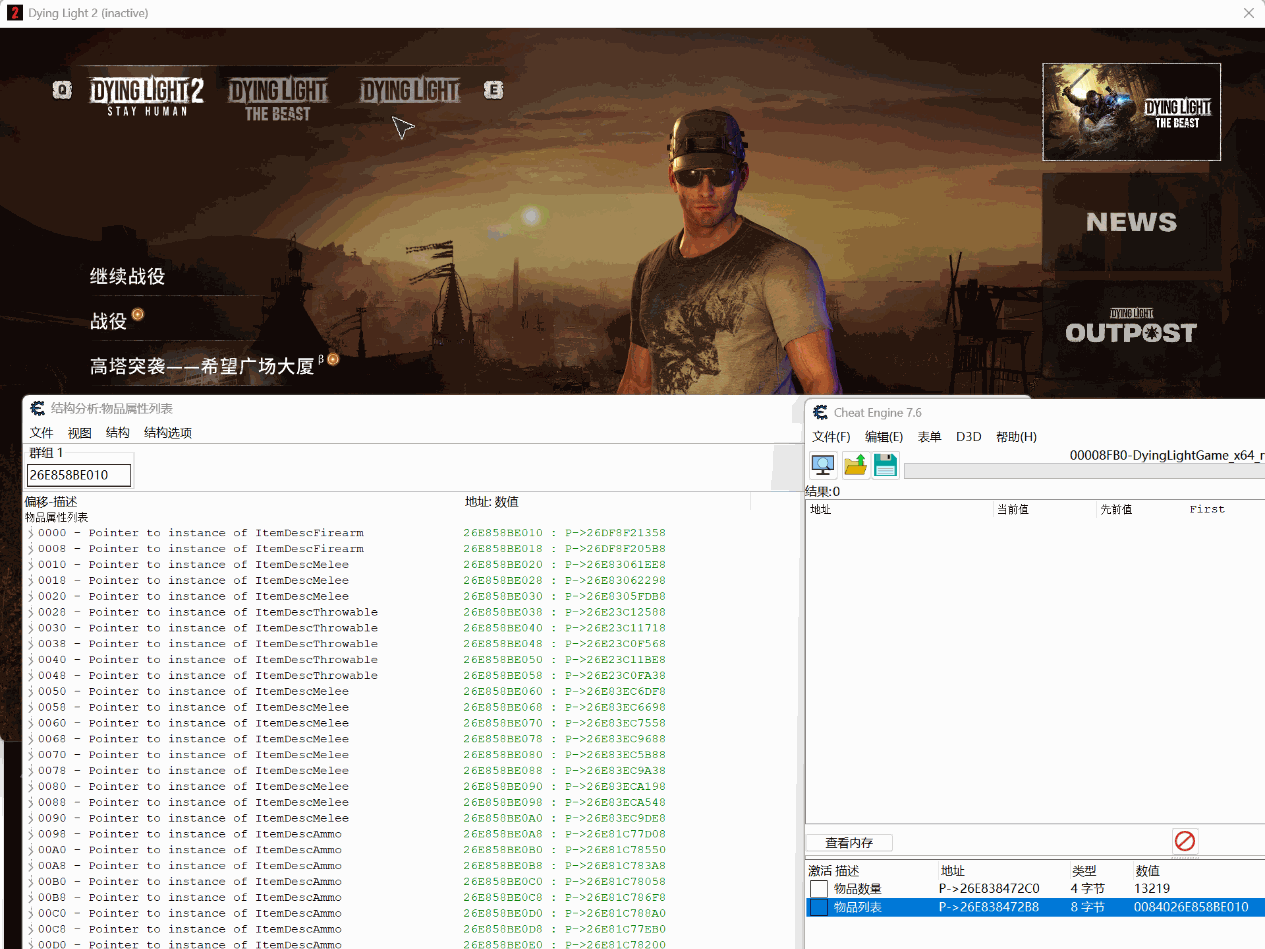


图2.F.9-重启游戏在游戏主界面进行验证

如图2.F.9所示，我们重启游戏后只在游戏主界面而不进入游戏，仍然能获取到物品属性列表并且其中储存着各种物品属性指针，同时获取到的成员数量仍然是13129。到这里，我们的猜想已取得验证，此0x38偏移处参数的获得方式也已得到。接下来，我们就进入物品属性的结构中，观察其中有什么可利用的物品信息，比如物品名称，同时这也是最重要的信息。它能够提供识别不同位置储存了什么物品的依据，也就是我们可以根据这个信息在列表中选择物品，为我们往后的物品刷取提供更高的自主性。

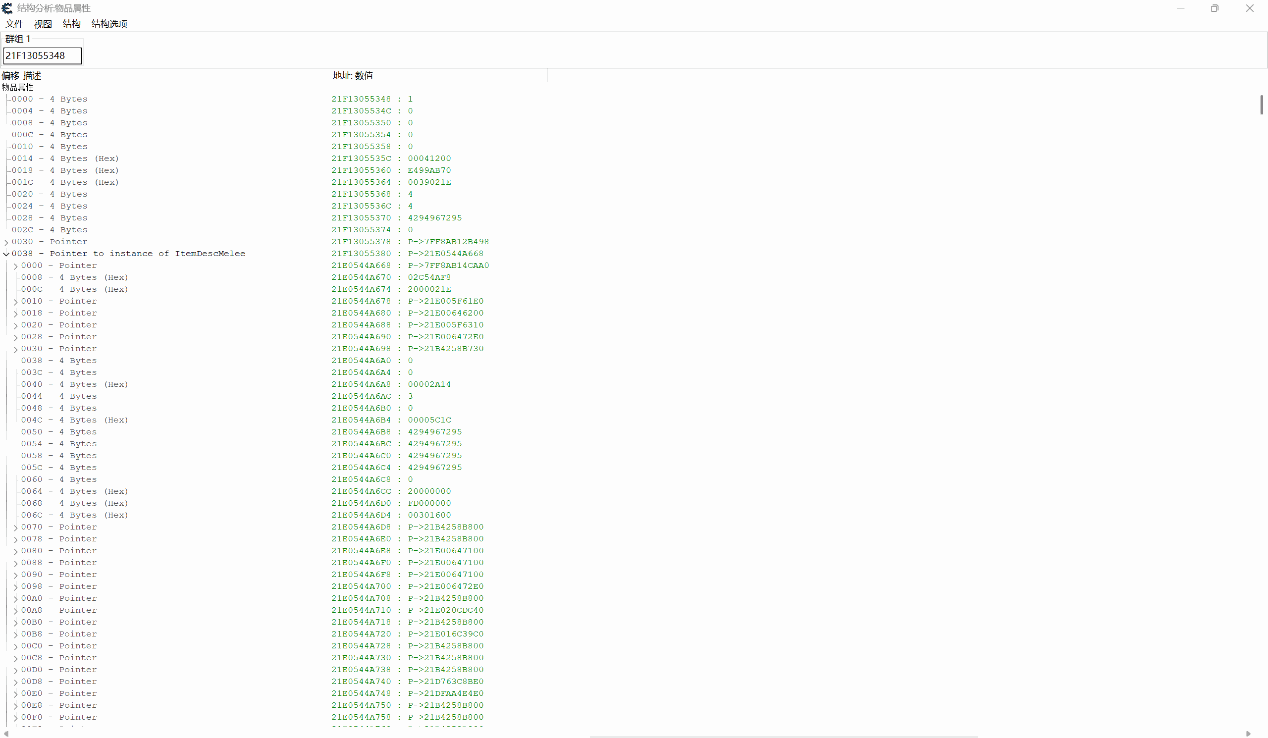


图2.F.A-参数在偏移0x38处的结构展开

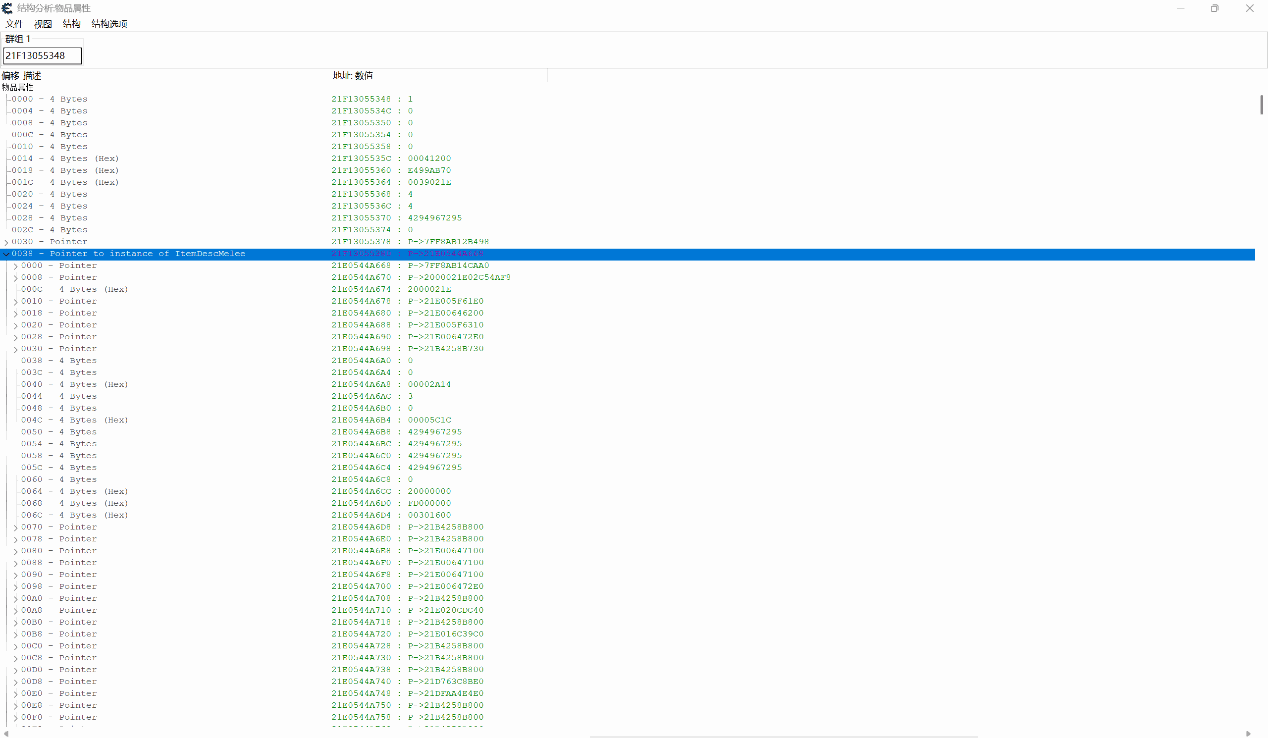


图2.F.B-将展开处的偏移0x8处的数据类型改为指针类型

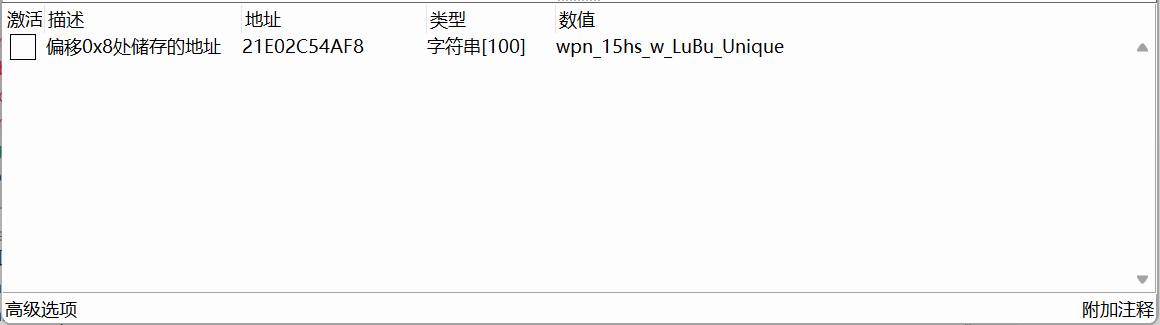


图2.F.C-将高位清零之后验证其是否为地址

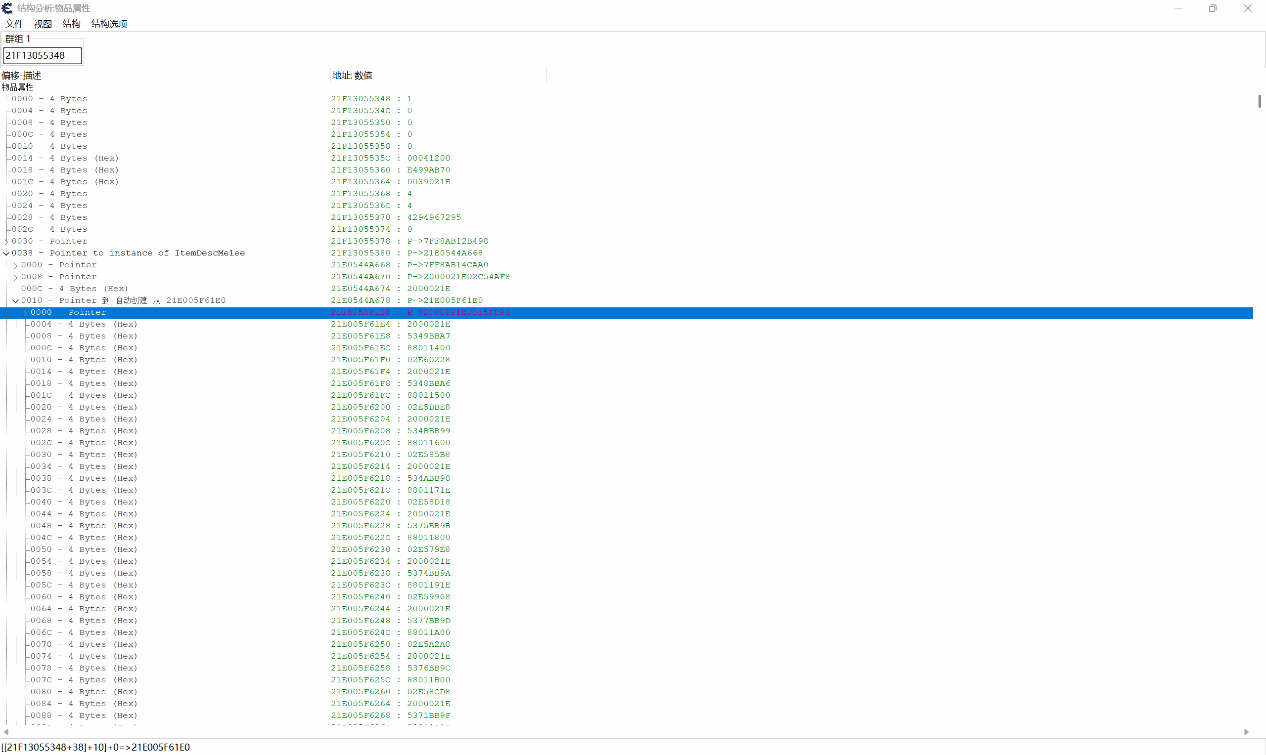
如图2.F.A所示，在展开处的偏移0x8和0xC处，我们可以发现这样的值：0x02C54AF8和0x2000021E。有没有觉得，它其实很像一个内存的地址？将它们倒着拼接起来，就是：0x2000021E02C54AF8，如图2.F.B所示。乍一看，这确实很像一处内存地址，只是高位处多了个2。如图2.F.C，我们将其高位清零之后，进行地址查看，发现其确实是存在的地址，且将其读取类型换成文本之后，得到存有类似代码的文本。来到这里，有没有发现这里的情况与上面要通过and操作清除高16位后才能引用列表地址的情况非常相似？表明游戏程序也可能对此处指针指向的地址也做了加密的处理，不能直接通过内存搜索得到。这里就给到我们一个提示，只要觉得某处地址里面储存的像一个地址，那就把它当作是指针类型来看，且把它指向的地址转化为正确的能引用的地址，看看里面都有些啥。在这里获得的文本中，有这样的字眼：LuBu，正好对应的吕布手拿方天画戟，而我们断点时恰好就是拾取方天画戟。那么，此处的文本可能就是代表了此物品的代码，能让我们用于识别物品，知道此物品属性结构储存的是什么物品的属性。但我们更希望获取的是这个物品的名称，也就是游戏UI中显示出来的“方天画戟”这字样。因此，我们继续往下寻找，尽量找到这一参量，那么我们就可以识别是哪一个物品了。

图2.F.D-在展开偏移0x10处展开，并将其偏移0x0处类型改为指针类型



图2.F.E-将其修改为正确地址后的读取结果

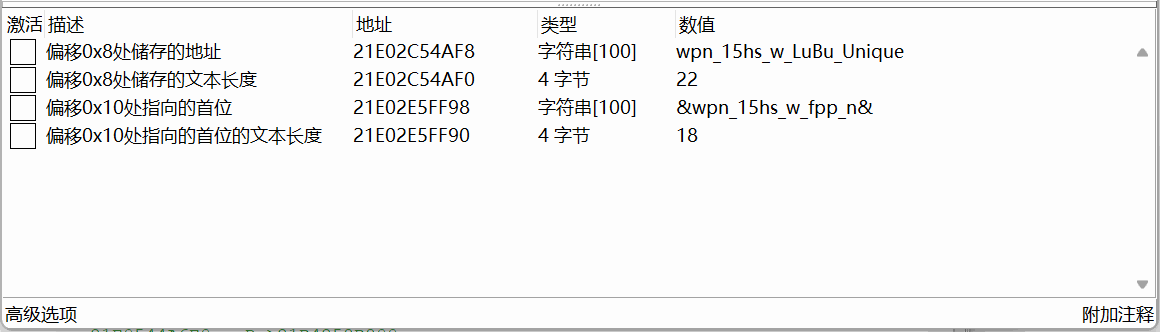


图2.F.F-获取文本长度

如图2.F.D所示，我们来到偏移0x10处，发现此处直接就是一个指针，我们将其指向的地址展开，得到一堆修改了高16位的地址。我们首先来看偏移0x0处的，将其高16位清零之后，同样去读取其内容，得到如图2.F.E所示的文本。这里的文本有些许奇怪，它前后用“&”把中间的文本给夹了起来。事实上，如果有过相关的了解，你就会知道这其实是用作文本查找的。其类似“&XXXXXX&”的格式是本地化查找键，因为这个游戏是支持多语言的，也就是说需要将文本对应玩家所在国家的语言。为了方便起见，游戏程序会设置一个统一的用于查找对应国家文本的文本（比如这里是中国，查找到的对应的本地化语言文本就是“方天画戟”），而这个文本也刚好对应着这个物品。并且其后缀的“\_n”也能对应名称的意思，n即是name的缩写，这在编程是很常见的一种缩写方式。也就是说，我们是能够通过这段文本来获取物品的名称的。到这里，我们还忽略了一件事，也就是我们需要去获取文本的长度。你可能会说我们直接用相关的函数获取不久行了么，比如用strlen\_s()或wcsnlen\_s()函数就好了。但通常来说，编程时设置了一段文本，肯定会在这段文本附近添加文本长度的信息，便于往后的使用。如图2.F.D和图2.F.B，这里指针的附近没有有关文本长度的信息，那我们就尝试直接在文本的地址上减去0x8，猜测文本长度就在文本的首部之上，之后才是往下文本的写入。如图2.F.F，我们直接获取到了文本长度，且其长度刚好对应着真实的文本长度。为验证上面游戏程序是用此文本来查找真实名称的猜测，我们可以用其他物品的本地化查找键来替代此处的文本，接着释放断点，观察方天画戟的名称是否变为另一个物品的名称。

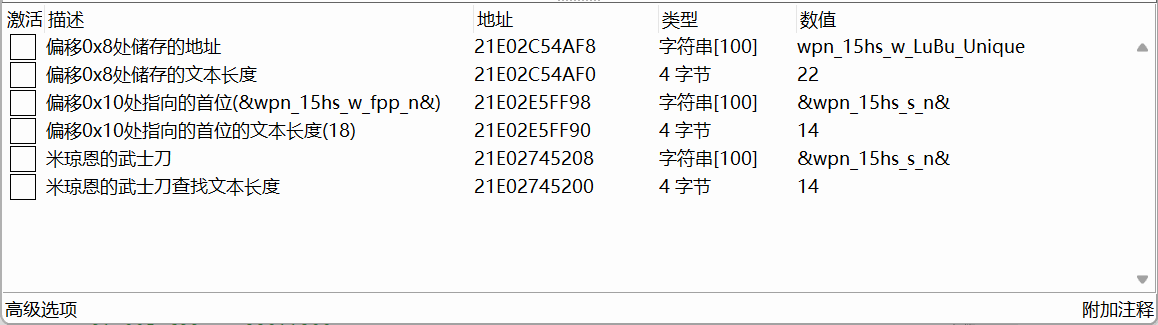


图2.F.10-将方天画戟的本地化查找键修改为米琼恩的武士刀的本地化查找键



图2.F.11-方天画戟物品名称修改成功

如图2.F.11所示，我们用相同的方法获得物品米琼恩的武士刀的本地化查找键，然后将其替换给方天画戟，物品方天画戟的名称成功被修改为了“米琼恩的武士刀”，从而验证了游戏程序确实是通过此文本来查找名称的。既然是通过本地化查找键查找对应的物品名称，那就一定有相关的函数处理此过程，接下我们的目的就是去寻找这个函数，实现对各个物品名称的获取。

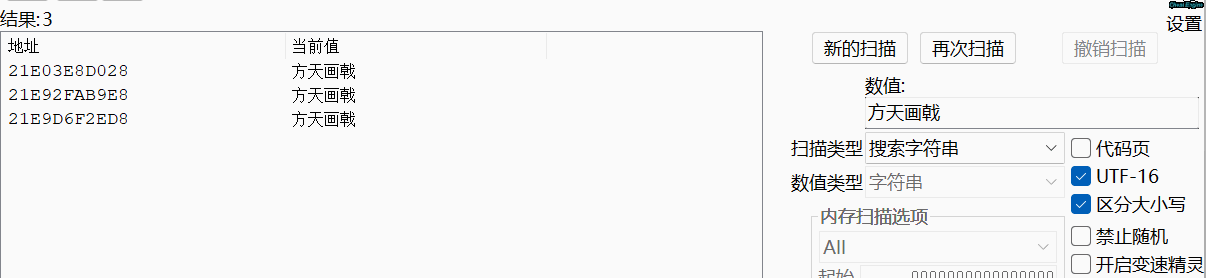


图2.F.12-使用宽字符设置搜索“方天画戟”

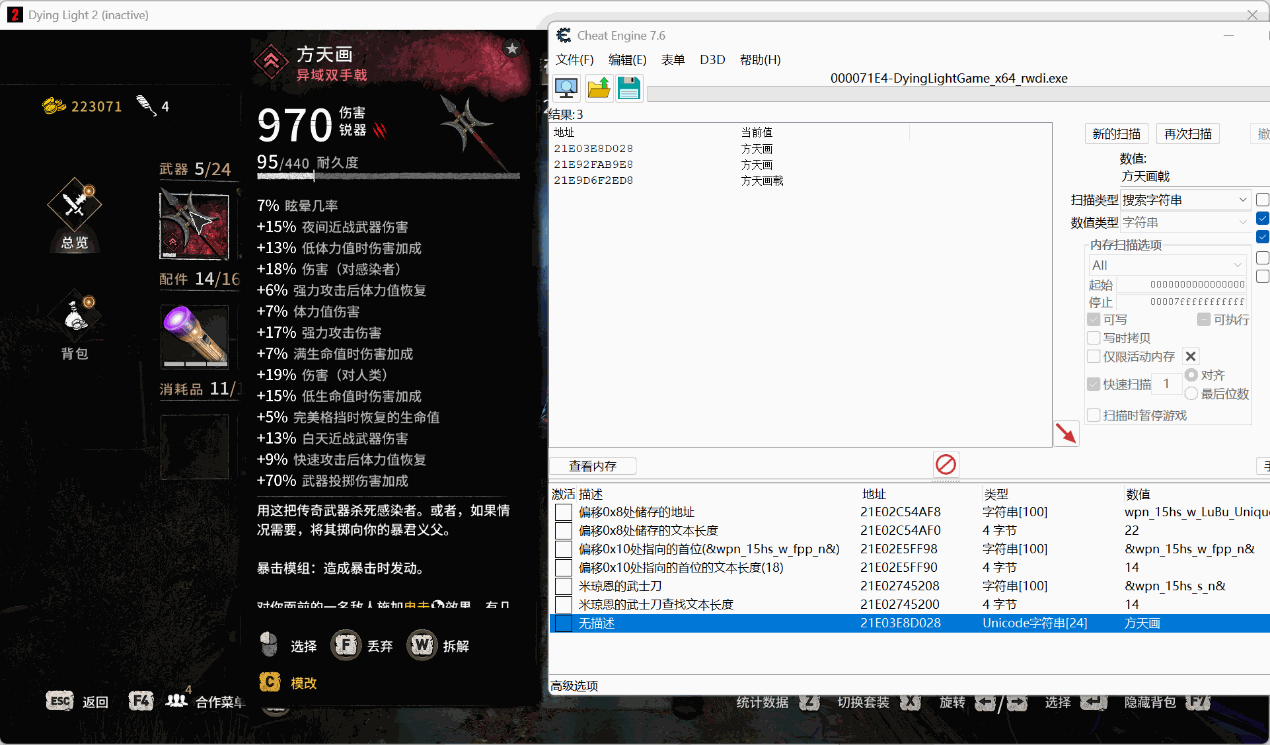


图2.F.13-修改文本，筛选真正的储存地址

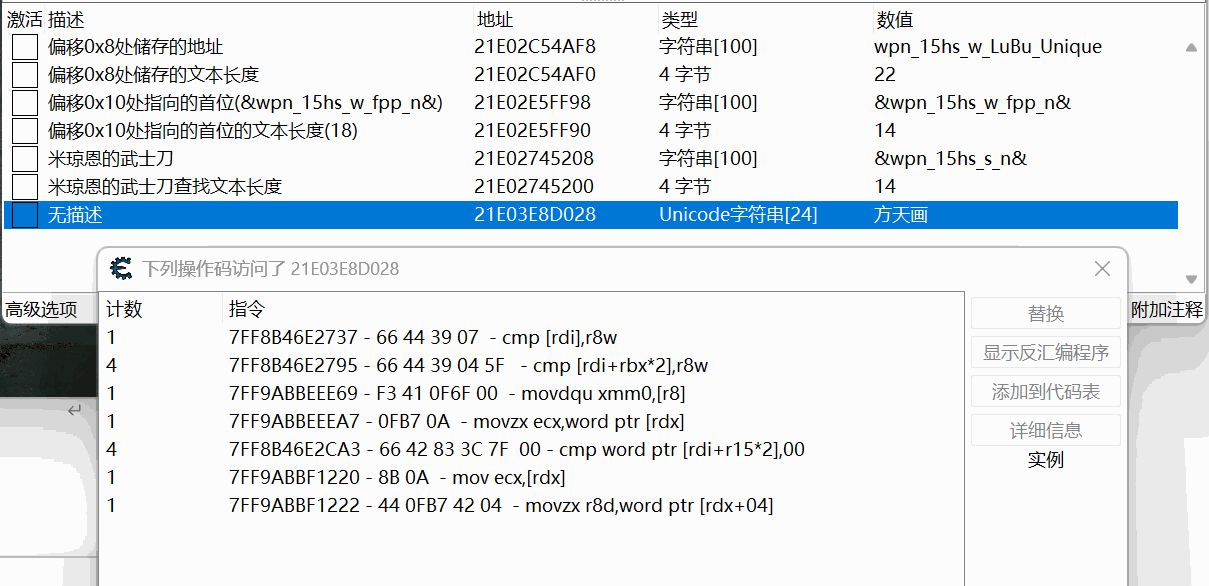


图2.F.14-监视访问储存”方天画戟”名称的代码

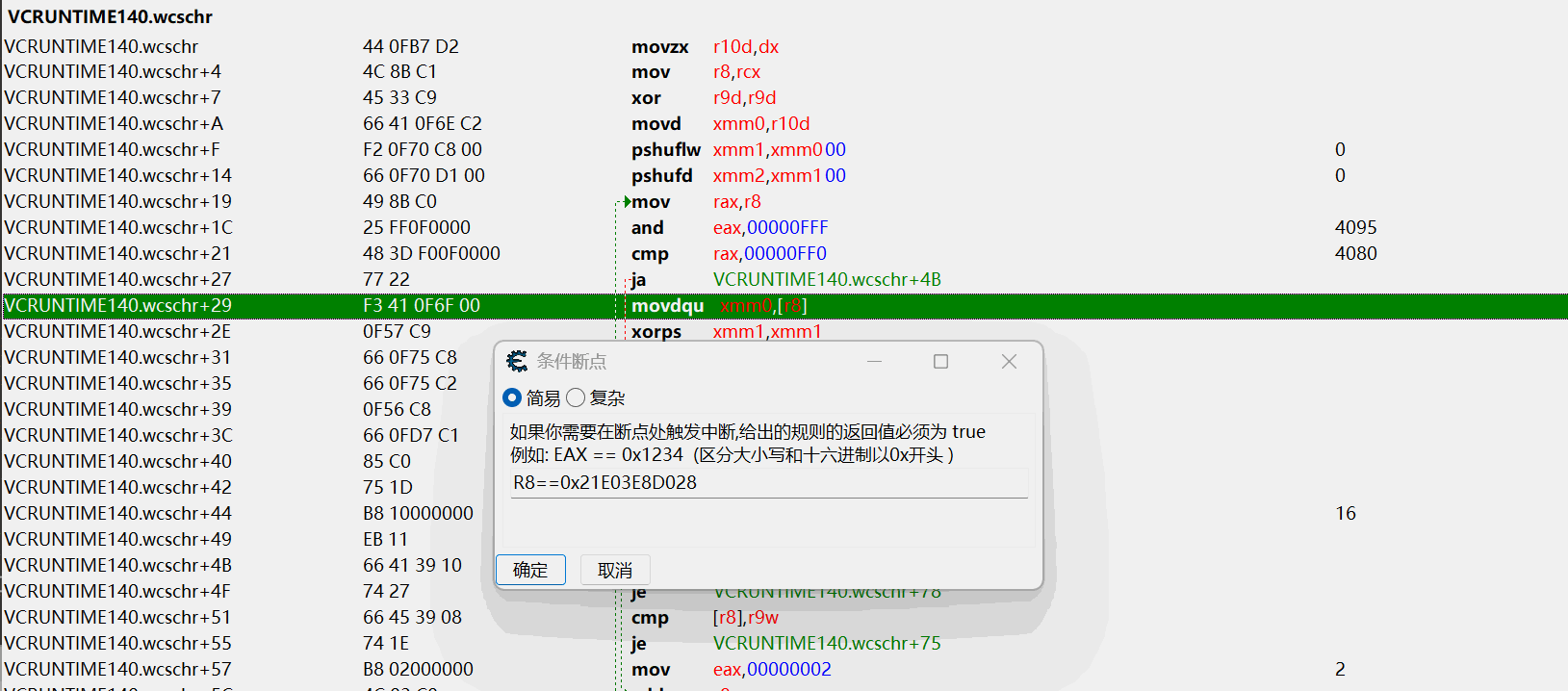


图2.F.15-设置条件断点

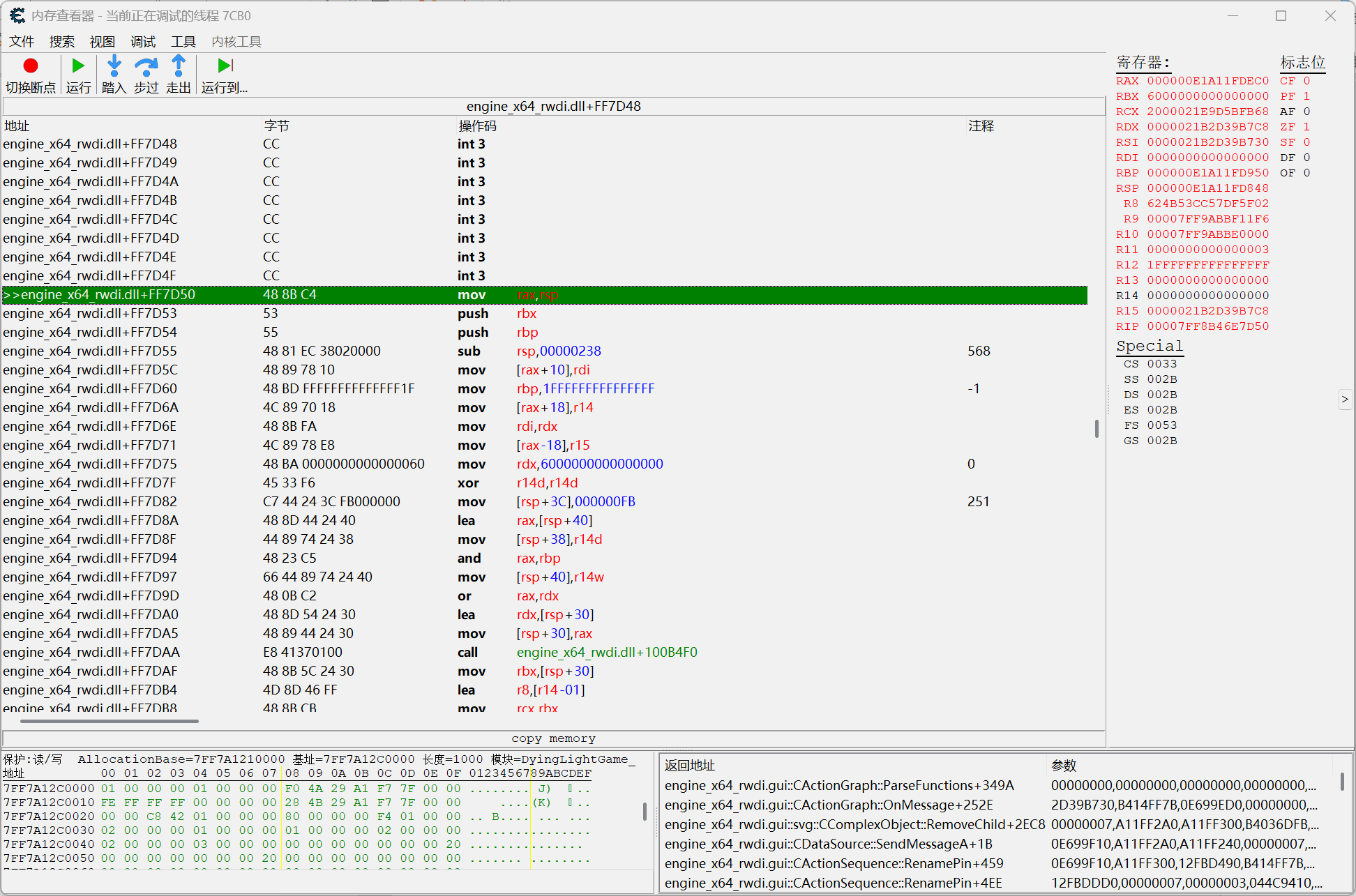


图2.F.16-找到符合要求能获取名称的函数

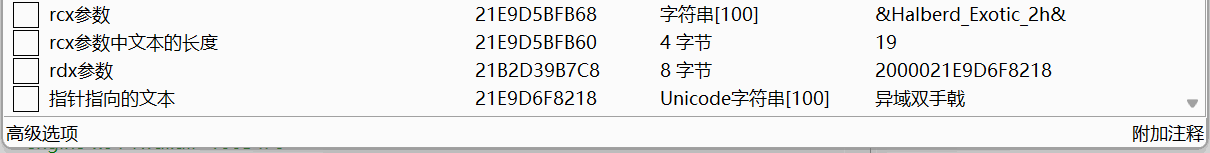


图2.F.17-参数内容(此时还未释放断点)

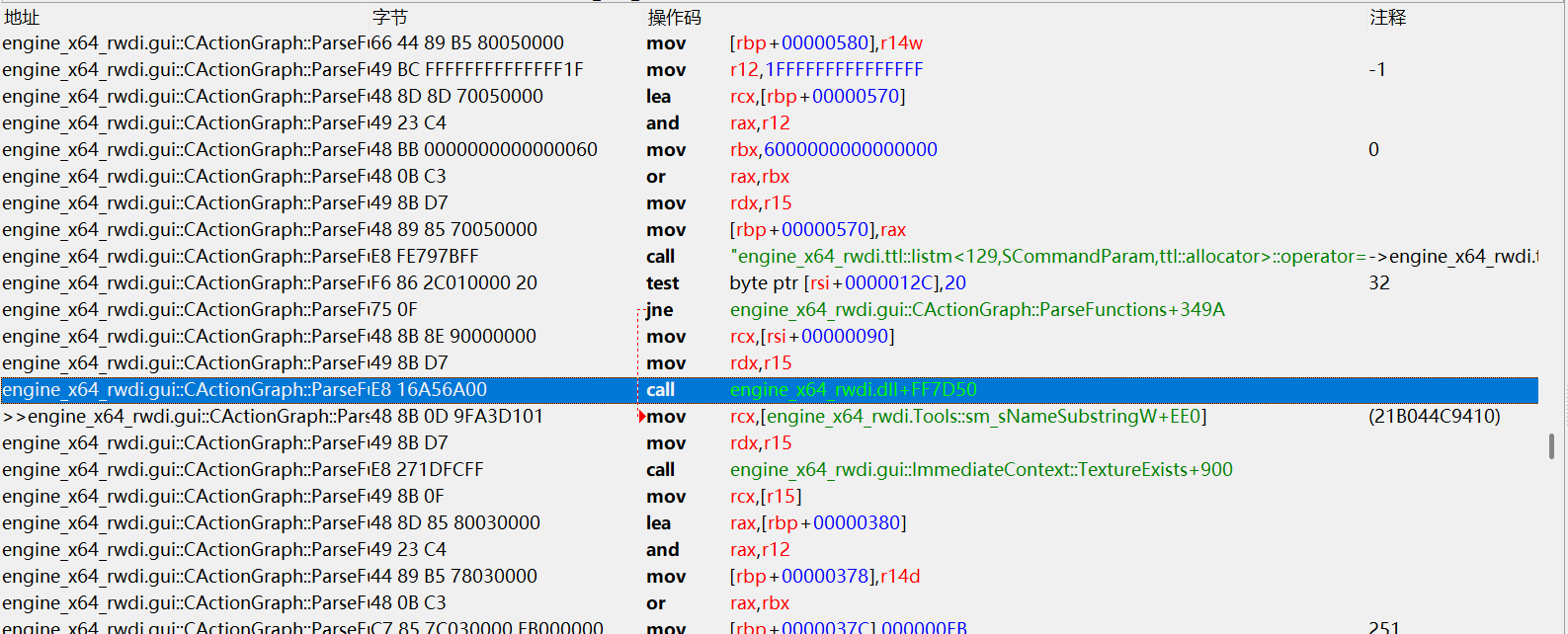


图2.F.18-函数调用方

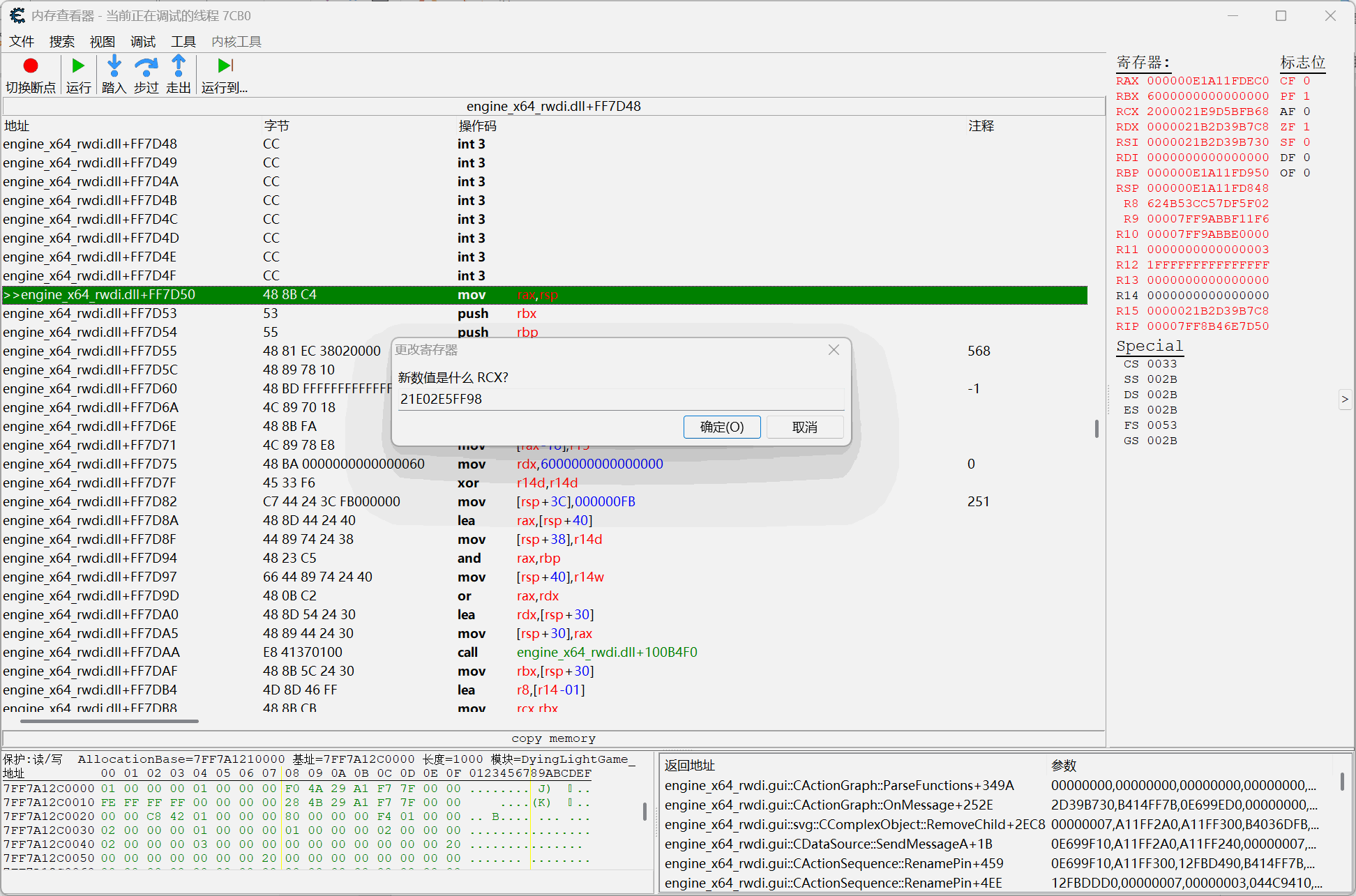


图2.F.19-修改rcx寄存器的参数，使其变为搜索方天画戟的本地化查找键

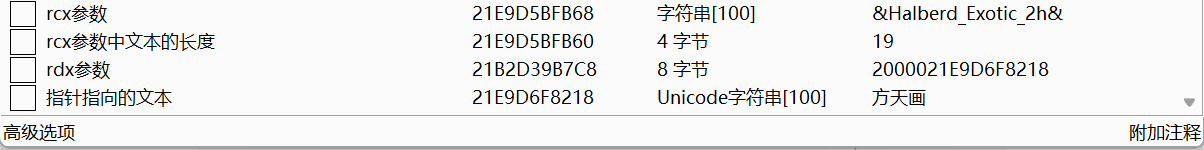


图2.F.1A-修改完成并释放断点后，获取到的文本同样变成了“方天画”

考虑到游戏程序要显示出物品名称必定会引用到真正的本地化语言文本，因此我们可以直接搜索“方天画戟”文本，然后监视什么代码访问了这个文本，然后间接地，像寻找增添物品函数那样，找出其相关的函数。

如图2.F.12和图2.F.13，考虑到游戏需要适应多国的语言文字，所以可以推测其应使用宽字符来保存文本，这样能够表示更多的字符。这里就把UTF-16给勾选上来搜索文本。随后我们修改文本，然后进入游戏丢弃方天画戟再捡起来刷新文本的显示从而测试并筛选出准确的地址。如图2.F.13，我们把名称的最后一个字“戟”给删去，随后游戏UI中显示出来的同样也是“方天画”，就筛选出了储存文本的地址。

接着，我们直接来监视上面得到的地址。如图2.F.14，这里得到了众多的访问代码。面对这么多的访问代码，就需要我们选择一个最好的去分析，否则就只会增加我们分析的工作量。结合上面的操作，我们可以发现，游戏中物品的名称并不是实时更新的，而是要结合特定的操作才会刷新，比如拾取物品。那就可以推测相关函数的处理过程可能存在文本复制的操作。因为既然不是实时更新的，而游戏UI又是时时刻刻在刷新绘制，这就说明了储存物品名称文本的地址不在游戏UI的代码中，而是在必要时，比如背包更新了物品，就会去查找物品名称，这时候就会涉及到复制文本的操作，把复制完成的文本再绘制到UI中显示出来。所以，在这里，我们选择分析的代码就应选择包含文本复制的操作。如图2.F.14所示，第三处访问代码的指令是：movdqu xmm0,[r8]，这个指令就是复制的操作，我们可以选择这里进行分析。进入到此代码所在的函数内部，如图2.F.15所示，由于这里是运行库的函数，基本上都是公共代码，游戏程序中有数不清的代码会调用此函数，所以不管我们有没有刷新文本，只要我们一在此处断点，就会被立刻断下来，因此我们需要使用条件断点来断下取得相应的调用方进行分析。之后，就像我们前面分析查找增添物品函数一样，到各个调用方所在的函数中去分析它们的参数，直到得到符合我们预期的函数。

如图2.F.16，我们在函数engine\_x64\_rwdi.dll+FF7D50处断下，发现其参数符合我们的要求。如图2.F.18所示，按照前面查找函数参数量的方式来看，结合图2.F.16推断其不是简单函数，就能得出其包括了两个参数。如图2.F.17所示，第一个参数，也就是rcx寄存器中的值，其地址中储存着本地化查找键的文本，同时第二个参数寄存器rdx中储存着接收查找结果文本的地址，因为其地址中储存“异域双手戟”的文本字样，刚好对应其本地化查找键中的意思。同时，我们可以得到第一个参数中的结构与我们上面找到的储存本地化查找键的结构是一致的。当我们把rcx参数的地址减去0x8后（当然，减去前要进行高16位清零的操作，这游戏惯用的伎俩了），我们就能得到参数中本地化查找键的文本长度了，如图2.F.17所示，19刚好对应其文本的长度。

如图2.F.19所示，为最终验证此函数的有效性，测试其是否能真正获取名称文本，我们先下断点，当被断下后，我们直接把第一个参数的指针替换为方天画戟的本地化查找键的指针，然后运行这个函数，观察第二个参数中是否得到了其对应的名称文本。看到图2.F.1A，第二个参数中确实得到了经过我们修改名称后的方天画戟名称“方天画”，从而验证了这个函数的有效性，能够通过本地化查找键获取其对应的本地化语言文本。随后，我们就可以结合我们找到的物品属性列表把每一个物品的本地化查找键给读取出来，然后调用这个函数将其名称给读取出来，不用我们根据物品代码去识别物品，给我们刷取物品提供更多的自主性。既然功能是这样，就把这个函数称为本地化语言文本查找函数，最终确定其函数地址为：engine\_x64\_rwdi.dll+FF7D50。同时，我们可以得到其参数的结构：

struct TextManagerA

{

int TextLen;

char Text[];

};

struct TextManagerW

{

int TextLen;

wchar\_t Text[];

};

第一个参数（rcx）：&TextManagerA.Text

第二个参数（rdx）：\*&TextManagerW.Text

至此，增添函数第二个参数中偏移0x38处的内容就已分析完毕。接下来继续往后面的偏移继续分析。这里需要注意的是，参数提供的本地化查找键是窄字节的，而写入回来的结果却是宽字节的。如图2.F.1A所示，本地化查找键的文本类型是窄字节的字符串，而查找的结果是Unicode宽字节的字符串。



图2.F.1B-方天画戟词条属性

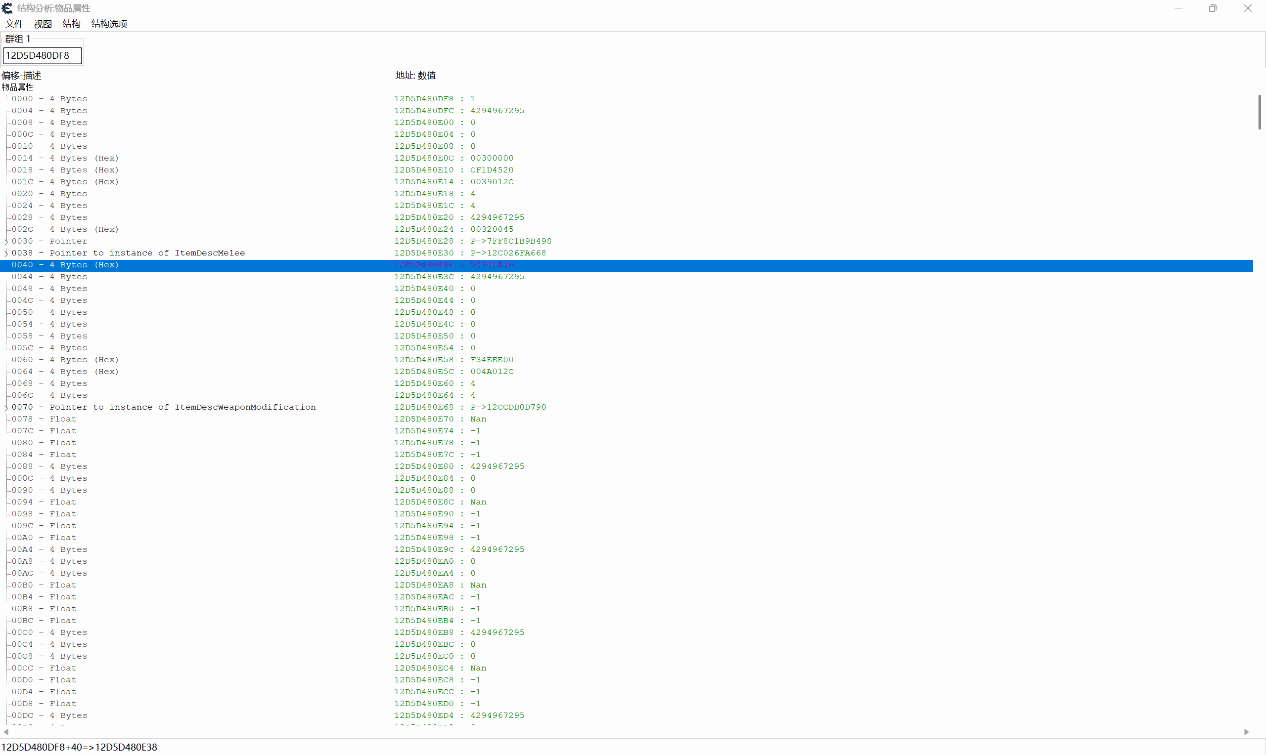


图2.F.1C-方天画戟物品属性偏移0x40处储存的数值

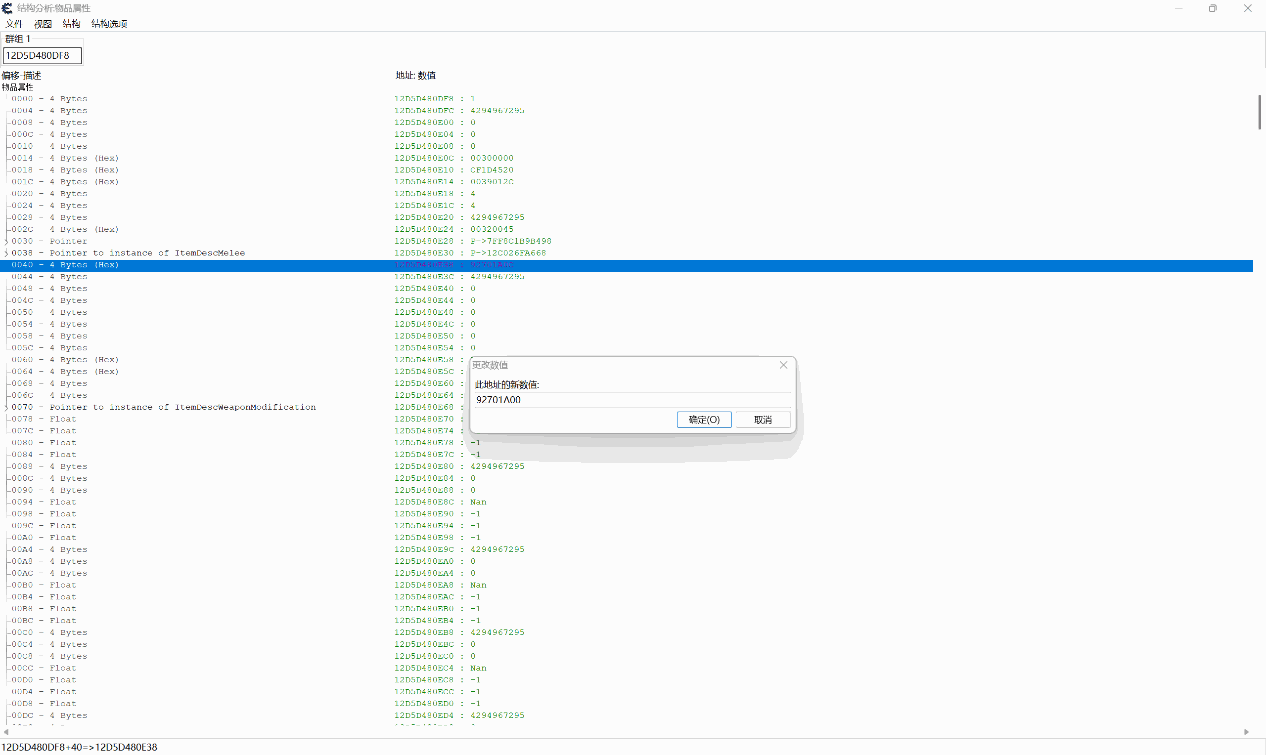


图2.F.1D-修改偏移0x40处的数值，将其末位改为0



图2.F.1E-修改了偏移0x40处的数值之后词条以及伤害发生变化

我们继续在增添物品函数头部断下点，继续往下分析。如图2.F.1C所示，这里我们来到偏移0x40处，可以发现此处是一个4字节数值。如图2.F.1D所示，将其数值的末位改为0释放断点，得到如图2.F.1E所示的物品词条。比对之前的词条，可以发现其词条的内容以及词条数量都发生了变化，同时伤害也发生了变化。表明偏移0x40处储存的数值可以控制武器的词条、伤害等的属性。下面就来细细探寻此处数值的规律。

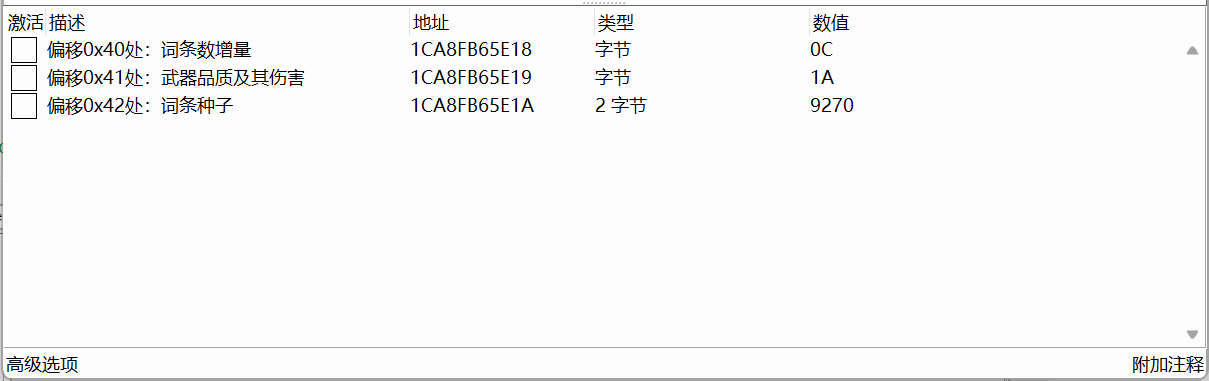


图2.F.1F-修改分析过后得到的偏移0x40处4字节数值的应用规律

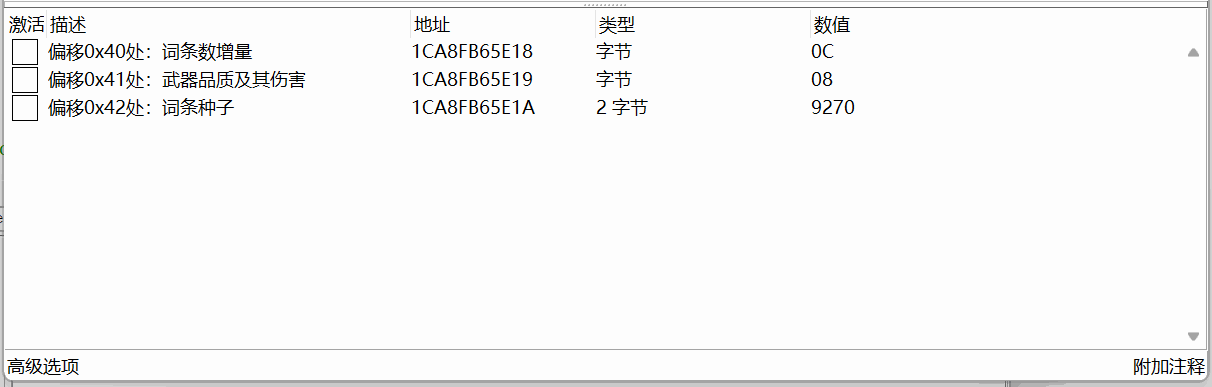


图2.F.20-修改方天画戟的武器品质



图2.F.21-方天画戟品质成功变为稀有品质

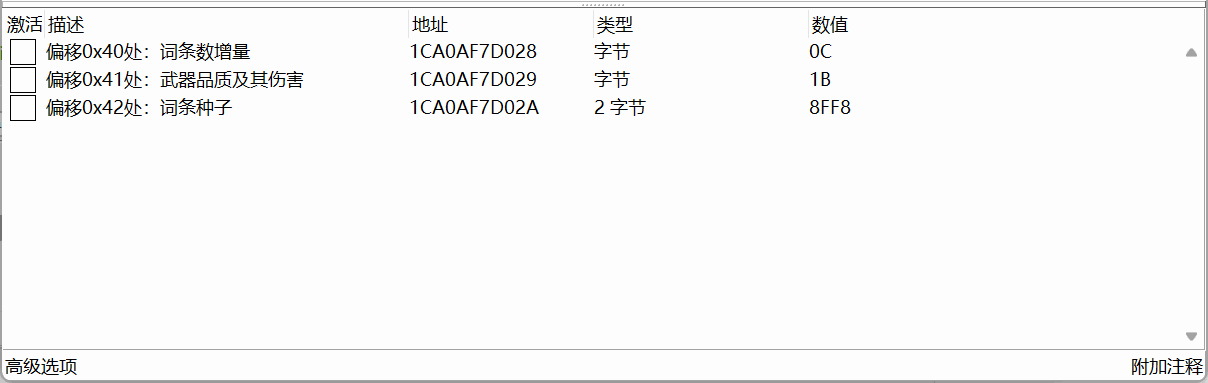


图2.F.22-将武器切切修改为无伤害



图2.F.23-成功将切切修改为无伤害

由于游戏重启过，因此此处的地址与上面的地址有所变化，但是并不影响继续分析。如图2.F.1F所示，通过我们的修改分析，可以将偏移0x40处的4字节数值分成3部分：0x40处为武器词条增量、0x41处为武器品质和伤害、0x42处为词条种子。如图2.F.20和图2.F.21所示，我们可以自定义武器的品质，不同的数值对应着不同的品质。再如图2.F.22和图2.F.23所示，修改成特定的数值会得到没有伤害这一特殊的属性。同时，这无伤害的属性修改只对大部分的武器有效，有些特别的武器，如方天画戟就不能修改出无伤害这一属性。同时我们可以得到以下的品质规律：

数值0x0-0x3：普通（其中0x2-0x3为无伤害，0x0为低伤害，0x1为高伤害）

数值0x4-0x7：罕见（其中0x6-0x7为无伤害，0x4为低伤害，0x5为高伤害）

数值0x8-0xB：稀有（其中0xA-0xB为无伤害，0x8为低伤害，0x9为高伤害）

数值0xC-0xF：独特（其中0xE-0F为无伤害，0xC为低伤害，0xD为高伤害）

数值0x10-0x13：精良（其中0x12-0x13为无伤害，0x10为低伤害，0x11为高伤害）

数值0x14-0x17：传奇（其中0x16-0x17为无伤害，0x14为低伤害，0x15为高伤害）

数值0x18-0x1B：异域（其中0x1A-0x1B为无伤害，0x18为低伤害，0x19为高伤害）

同时，词条数增量就表示了在基本的词条数量上会增加的数量。比如传奇品质的武器本来会有4个词条，现在把0x40处的数值改为2，就会增加多2个词条变为6个词条。增加的同时，词条及其数值也会发生变化。经过实测，词条增量并不是无上限的，根据武器的不同，相对应的有着不同的最大词条数量，超过这个数量之后只会改变词条的内容及其数值而不会继续增加词条的数量。而词条种子，顾名思义，就是决定了词条的生成，不同的数值可以生成不同的词条以及数值。

由于上面得出的数值规律是通过修改武器的得到的，但游戏中拥有词条的物品还有装备，那么装备此处的数值规律会不会和武器的一样？还是说有别的规律？这里需要我们继续探索。

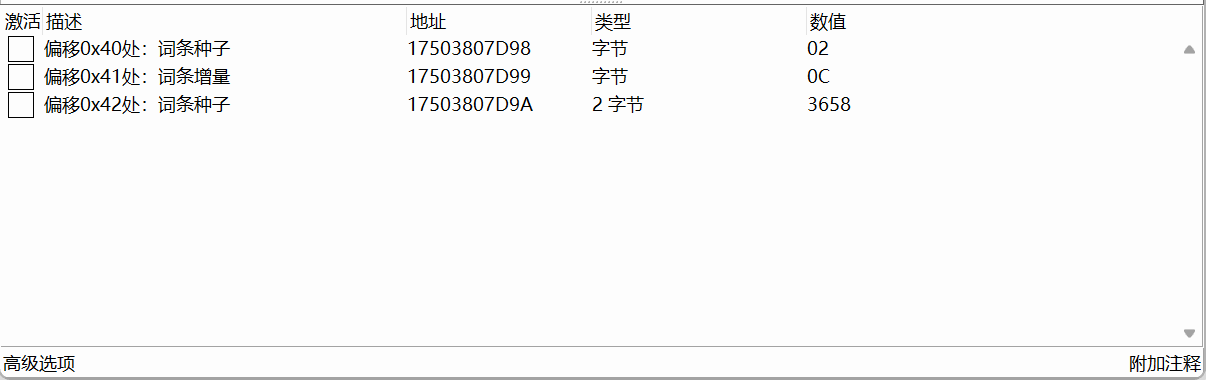


图2.F.24-装备偏移0x40处的四字节数值应用规律

如图2.F.24所示，装备的数值规律与武器的数值规律不一样。在偏移0x40处，这里不再是词条增量，而是变成了控制词条内容及其数值的词条种子，与偏移0x42处的情况一样。同时，偏移0x41处变成了词条增量，装备的品质不能修改，品质控制的功能消失了在此数值中。

由于其他物品没有词条这一说法，因此此数值也就失去了上述的各种功能。当然，这一结论也是通过修改相应的数值得到的，无论修改成什么，其属性都不会变化。到这里，就完成了对物品属性结构偏移0x40处字节数值规律的探索。虽然说上面的陈述很简单，只有简单的几句话，但这些规律都需要进行不停地修改得来，没有别的方法。因此，这里直接给出规律结果，没有别的过多的陈述。

最后在这里我们需要验证其有效性。它这个的有效性就是指修改过后的数值能不能保存下来，下次进游戏的时候能不能得到修改过后相同的词条及其数值？经过多次的测试，这里很幸运的是，修改过后的内容仍然会被保留下来，即使是重启游戏、重启电脑都会再现修改过后的内容。

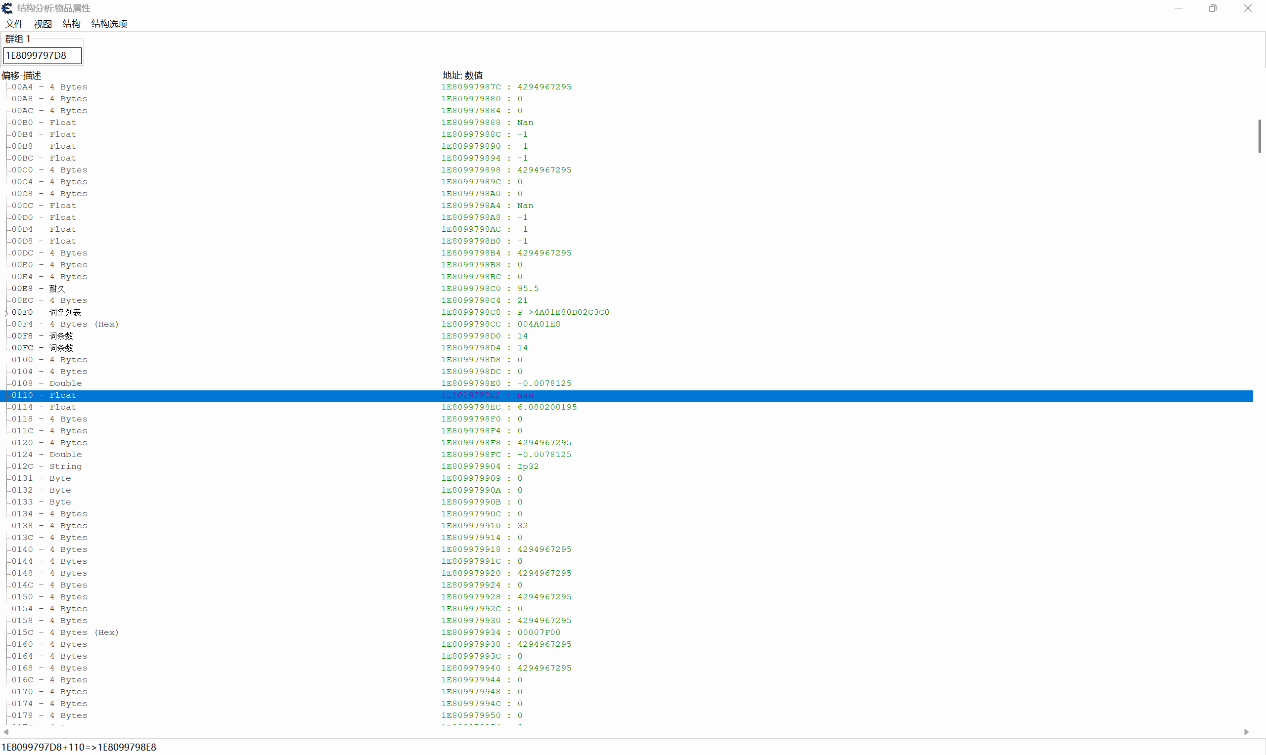


图2.F.25-耐久及词条列表

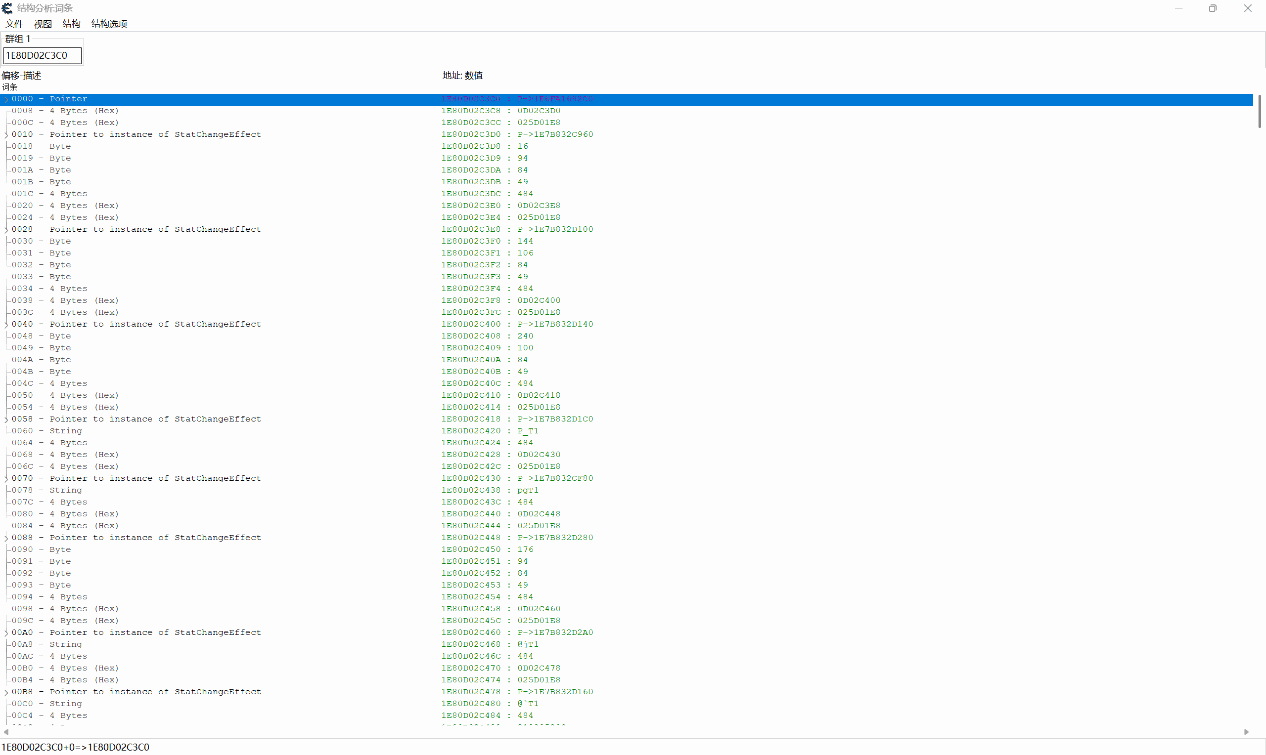


图2.F.26-词条列表内部

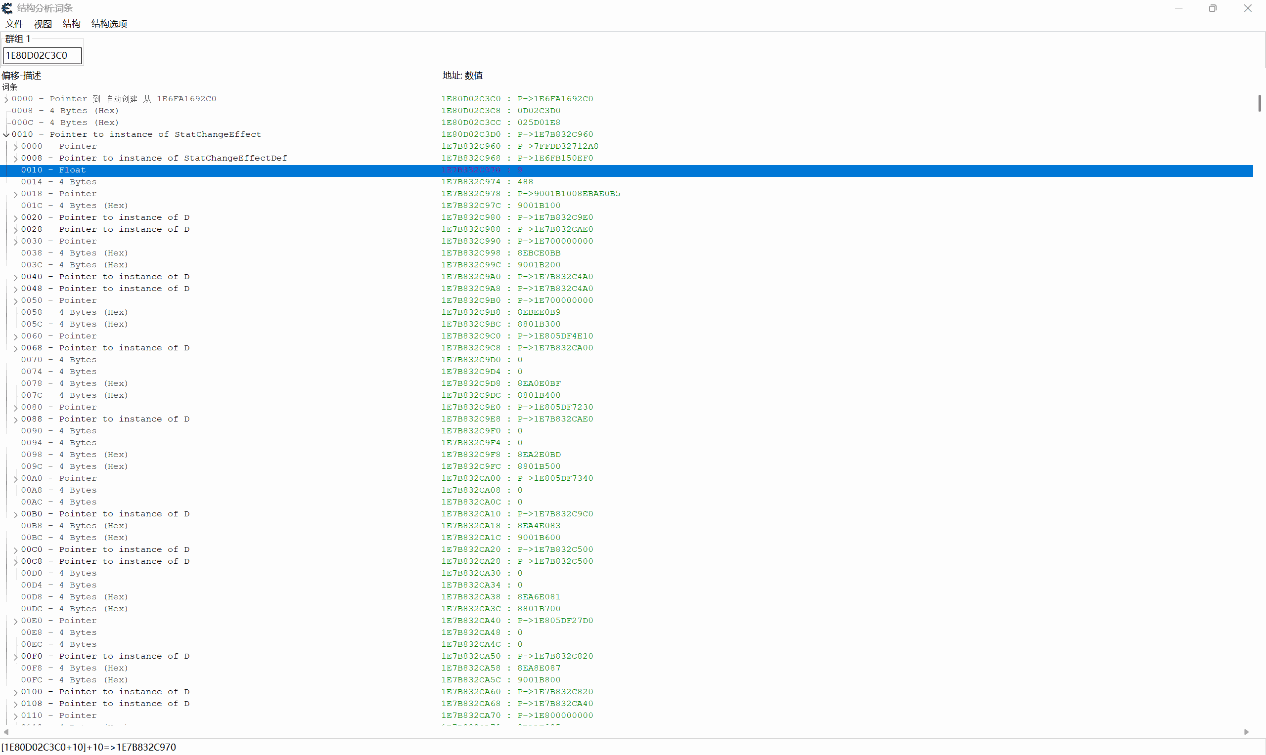


图2.F.27-词条偏移0x10处第一个词条结构展开

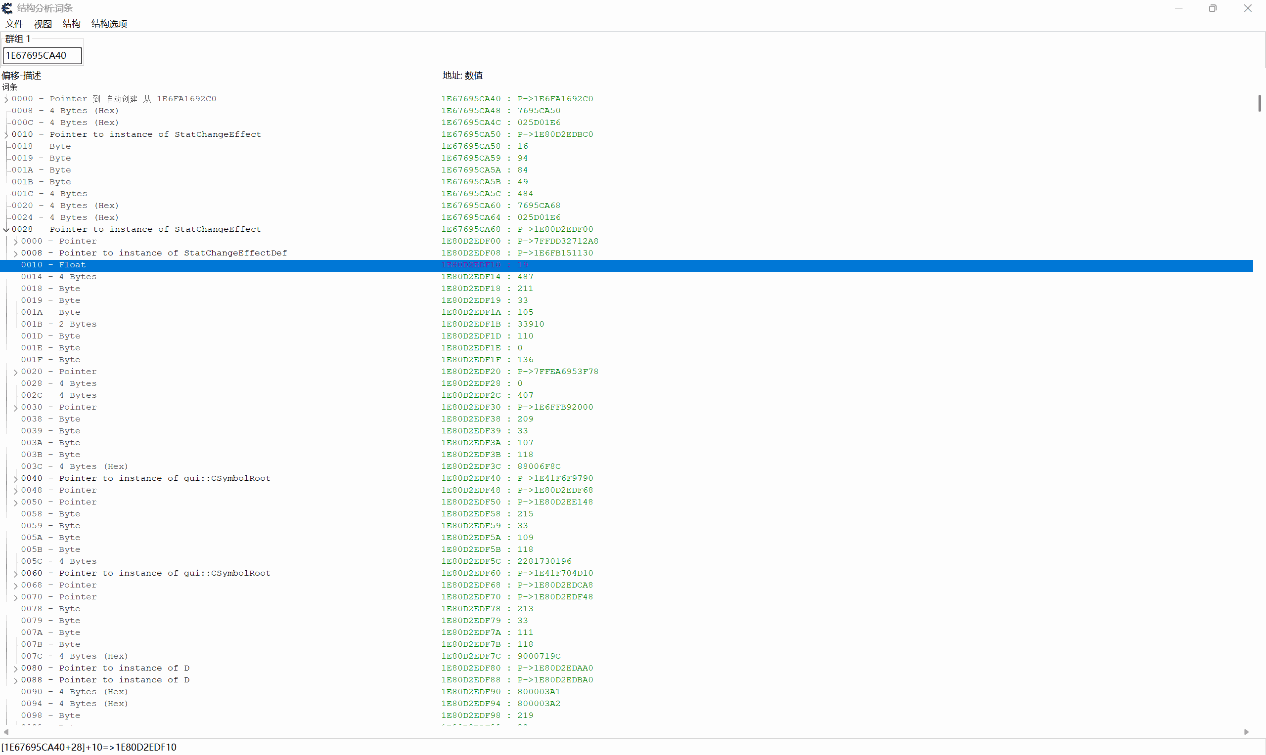


图2.F.28-词条偏移0x28处第二个词条结构展开



图2.F.29-方天画戟词条

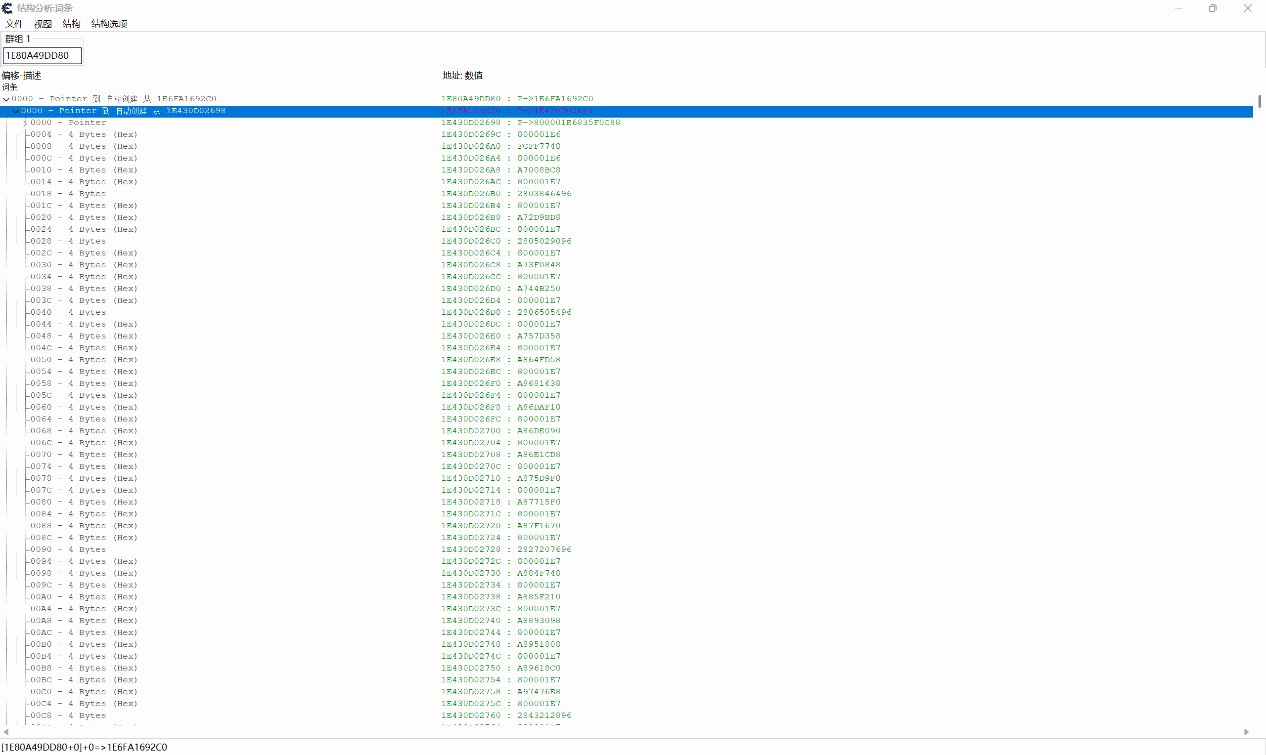


图2.F.2A-词条代码文本储存

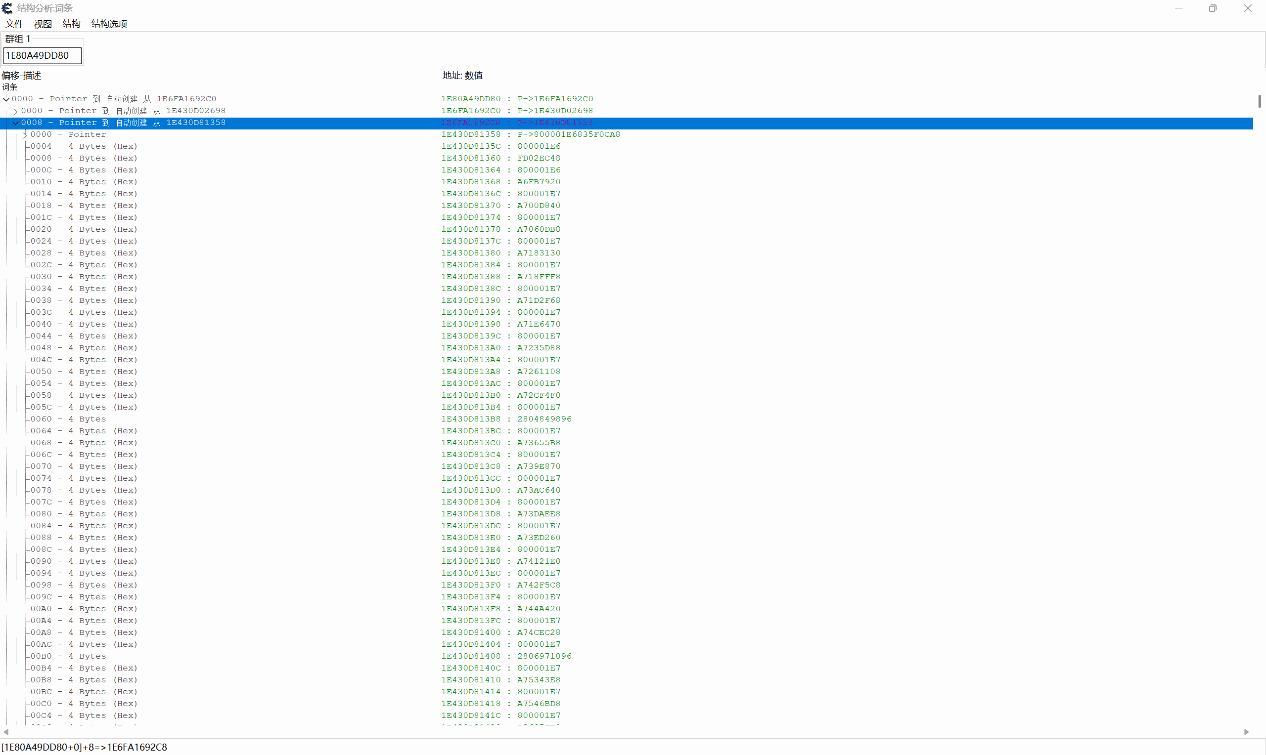


图2.F.2B-词条本地化语言查找键

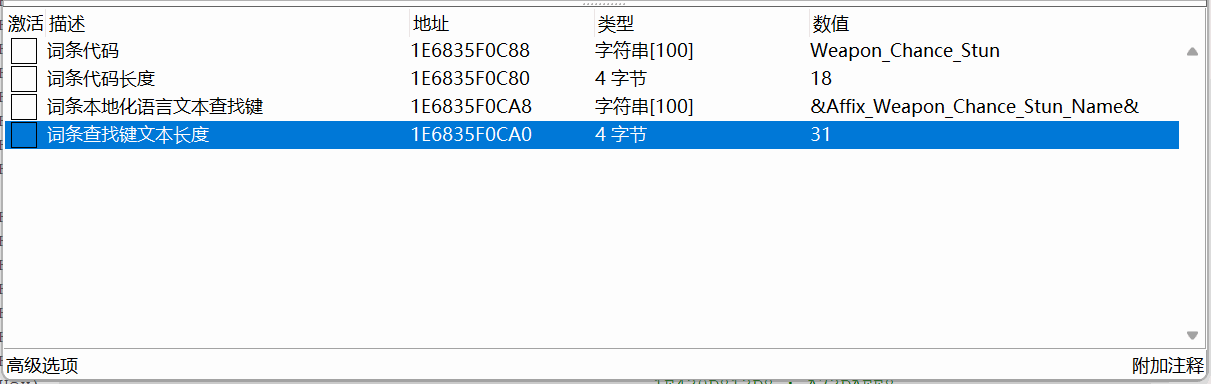


图2.F.2C-词条词条代码文本及其本地化语言查找键

我们继续往下看，如图2.F.25所示，在偏移0xE8处，这里储存着单浮点类型的耐久属性，修改成多少物品的当前耐久值就是多少，而不是物品的总耐久度，这里就不再展示其修改和验证的过程。

接着就来到偏移0xF0处，我们很容易发现，这里其实又是一个指向了修改了高16位地址的指针。我们将高16位清零，得到正确的地址后，我们进入其空间查看结构，如图2.F.26所示。通过其自动解析出来的结构名称StatChangeEffect，可以猜测这个列表储存的可能就是词条。再如图2.F.25的偏移0xF8和0xFC所示，这里的数值恰好对应着图2.F.29中武器的词条数量，可以进一步肯定里面储存着词条。继续进入其内部观察，如图2.F.27和图2.F.28所示，在其各自的偏移0x10处，其储存的单浮点类型数值恰好对应着图2.F.29中第一和第二个词条的数值。

我们继续在附近探索其结构，如图2.F.2A和图2.F.2B所示，这里面各自储存着词条词条代码文本和词条本地化语言查找键文本。将它们指针指向的地址的高16位清零后，查看其内容进行验证。如图2.F.2C所示，我们可以得到对应的文本，将其简单地翻译过后，其文本确实对应着图2.F.29中的第一个词条，验证了物品属性结构偏移0xF0处确实储存着指向词条列表的指针。然后偏移0xF8和0xFC处储存着列表中词条的数量。随后，我们可以访问装备的物品属性结构，来到相同的位置，可以发现也同样储存着其对应的词条，这里也不再展示图片。

到这里，我们可以有一个想法，既然这里的列表储存着词条的内容，而词条的内容又取决于物品属性结构中偏移0x40处4字节数值中的词条种子，那么我们就可以猜想游戏程序中有那么一个函数，给出用于保留词条的内存空间和相应的种子等内容，就可以生成出对应的词条供我们参考，让我们选择适合的种子，使得我们拥有更强的自主性。接下来，我们就尝试去找到这个函数。

考虑到游戏中词条就是那些词条，词条的内容及其效果是一样的，不同的只是词条对应的数值不同从而导致对应的作用效果的强烈度不同。既然是这样，我们就又可以推测出一些不变量，就如图2.F.27中结构StatChangeDef，从其名称末尾的Def中就可以得到这是某些东西的定义，就能联想到这是词条的定义，是游戏中已经定下来的，就像我们前面寻找物品列表一样，利用这个不变量来寻找函数。既然游戏要生成词条，而词条中又包含了此结构，同时此结构可能又是一个不变量，那么游戏程序在生成词条时就必然要引用到此结构的地址，其相关的代码就有可能会访问到此结构地址，那么我们就可以找到相关的代码从而找到我们的目标函数。

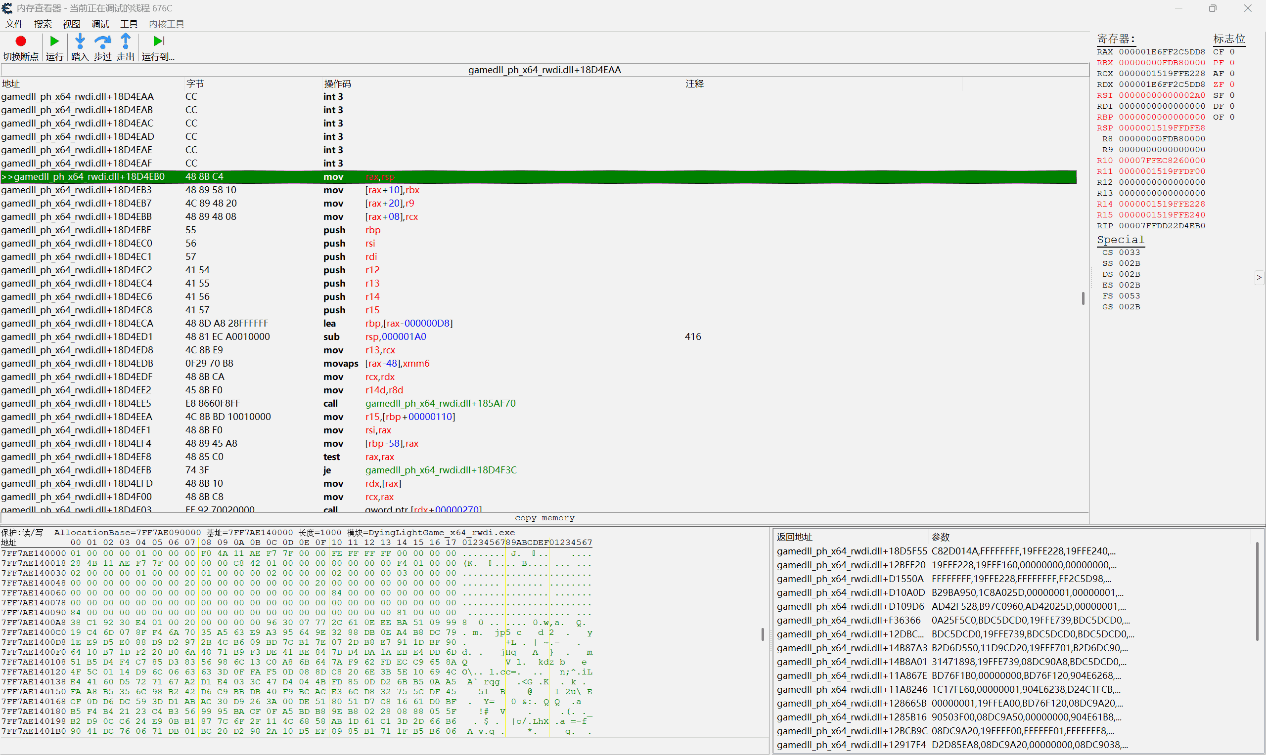


图2.F.2D-符合要求的词条生成函数

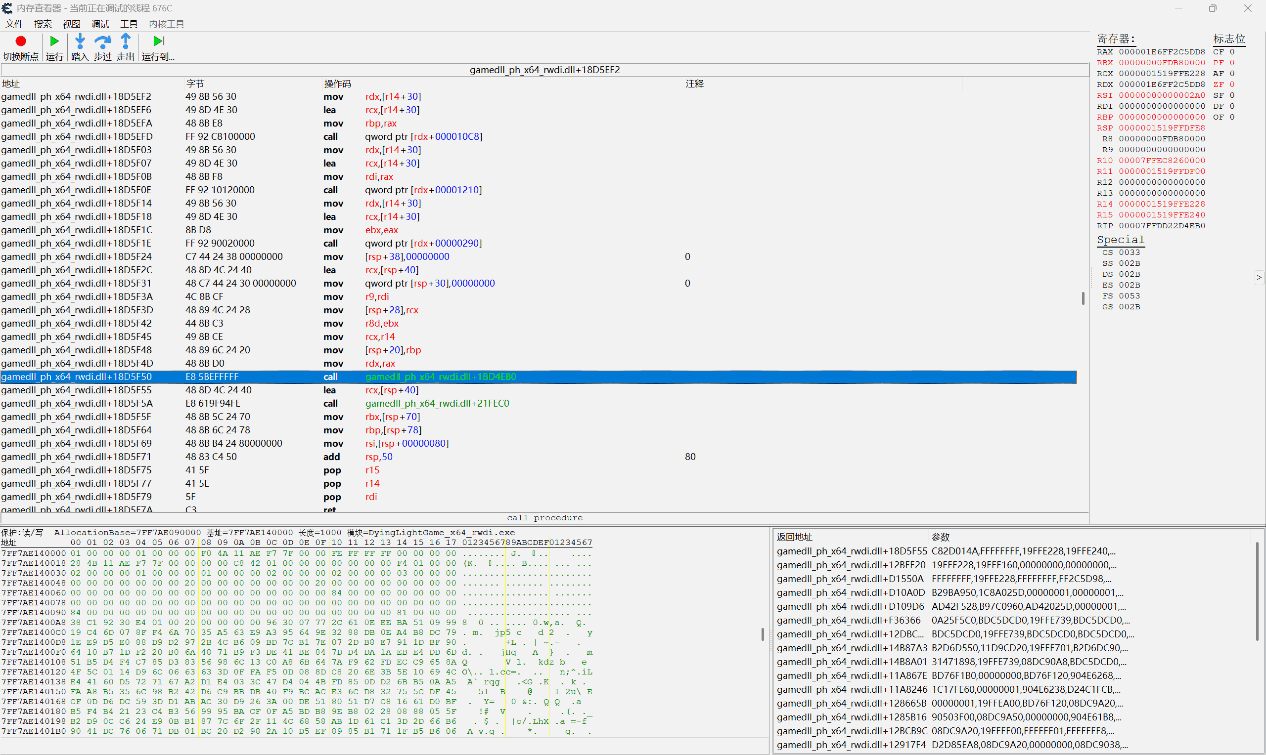


图2.F.2F-词条生成函数调用方



图2.F.30-词条生成函数rcx参数



图2.F.31-词条生成函数rdx参数

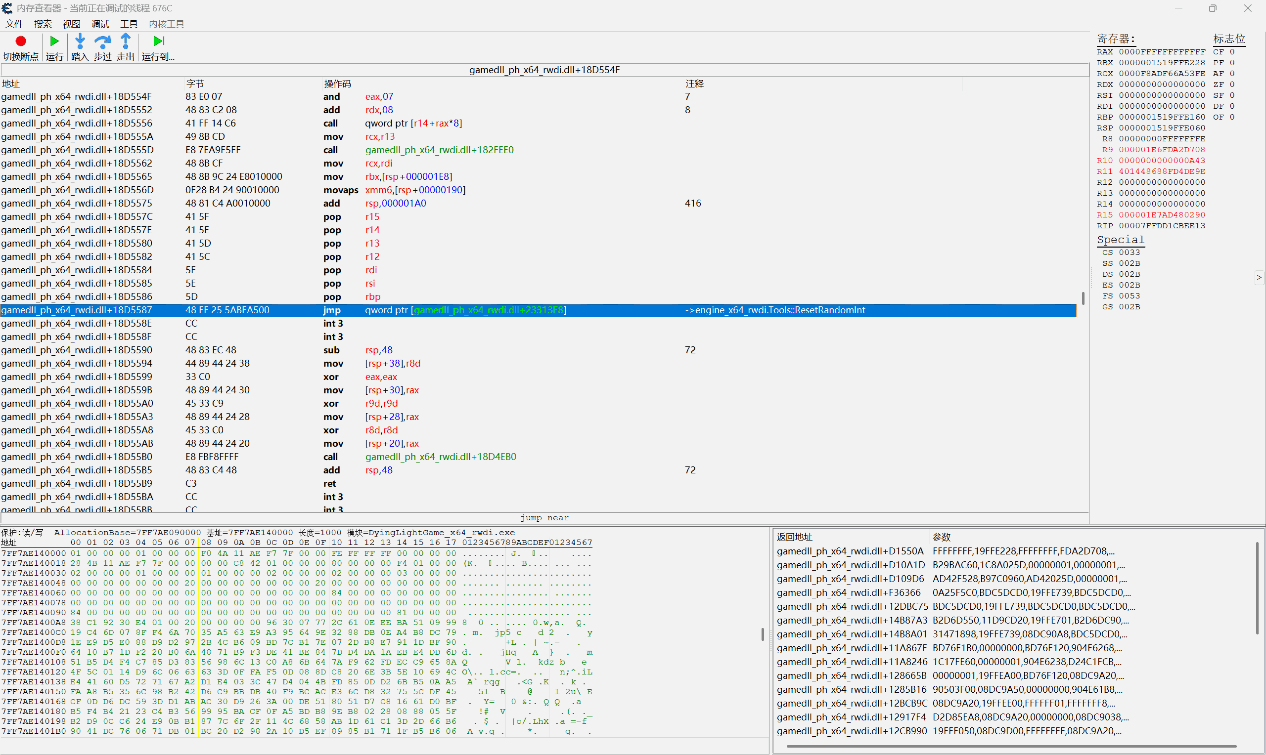


图2.F.32-词条生成函数尾

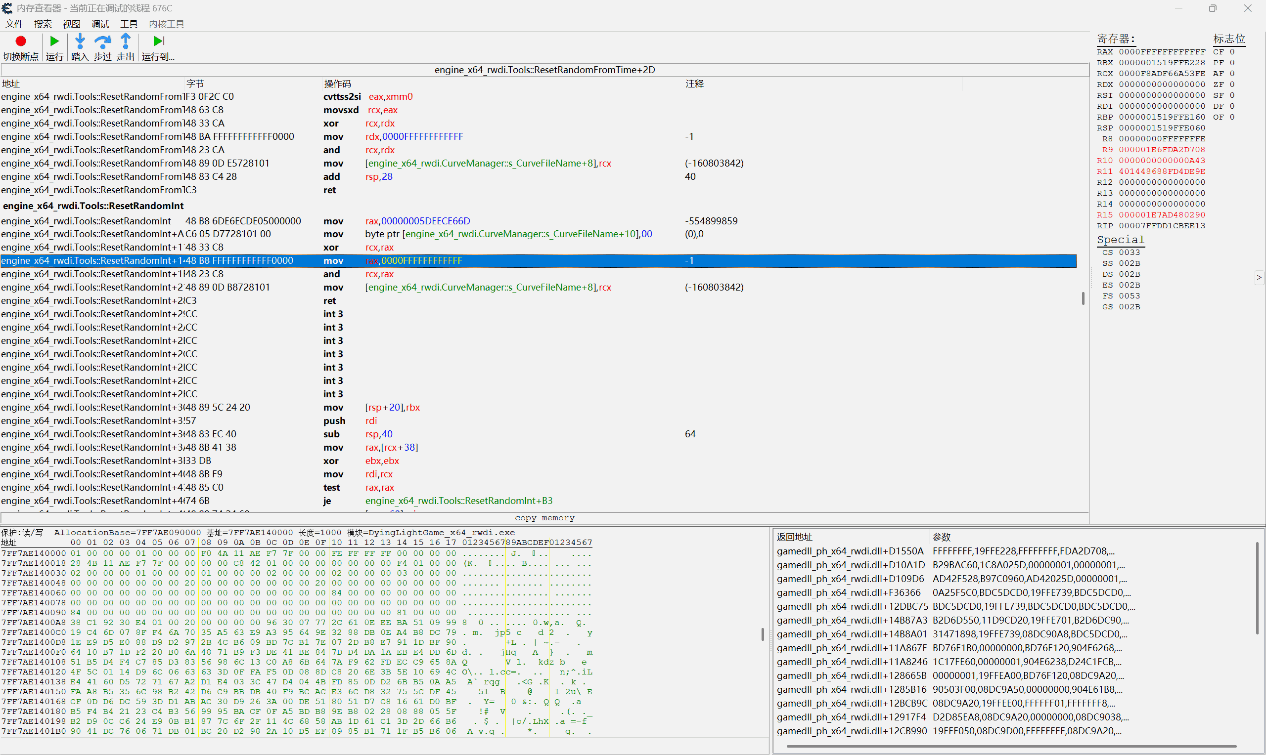


图2.F.33-词条生成函数尾部所进入的函数

像我们寻找增添物品函数、本地化语言查找函数那样，通过访问代码得到相关的函数，查看其调用方所在函数的参数以及运行结果来得到我们的目的函数。如图2.F.2D所示，我们定位到了地址gamedll\_ph\_x64\_rwdi.dll+18D4EB0处的函数。如图2.F.30所示，对于第一个参数的结构有没有觉得很眼熟？事实上，rcx参数的结构就是物品属性结构，简直就是一模一样！同时，rcx结构的偏移0x40处的词条种子与第三个参数r8寄存器中的数值一模一样，表明了函数第三个参数就是物品的词条种子，即要生成词条所提供的种子数值。至于第二个参数，这里的结构事实上就是物品属性结构偏移0x40处所在的结构而已，我们可以直接通过加0x40偏移获得此参数。同时，当我们释放断点之后，函数会在我们给第一个参数所提供的物品属性结构的偏移0xF0处，也就是储存词条列表的位置中写入相关的内容，同时也会在偏移0xF8和0xFC中写入列表中词条的数量。

来到函数的调用方，如图2.F.2F所示，从图中我们可以知道这个函数有8个参数。从多次的断点试验中可以得到，剩下的5个参数，r9、[rsp+0x20]、[rsp+0x28]、[rsp+0x30]、[rsp+0x38]中的值都是0，这也就意味着我们可以忽略分析此5个参数，直接将其填充为0即可，只需要我们正确填写前三个参数就能正常调用函数获得我们需要的结果，而其结果就保存在我们在第一个参数中提供的物品属性结构中。

最后，我们来分析此函数的返回值。如图2.F.32和图2.F.33所示，在函数返回时，函数直接jmp跳转进入到另一个函数中，而且此函数中的rax被固定写入了值0x0000FFFFFFFFFFFF。这种情况就表明了此函数是没有返回值的，因为无论怎样，都只会返回同一个值，这里我们就能忽略其返回值了。最后，我们可以得到词条生成函数各个参数的结构：

第一个参数（rcx）：物品属性结构地址（包含了耐久、数量、词条的结构，也就是增添函数中第二个参数rdx中的结构）

第二个参数（rdx）：物品属性结构偏移0x40处的地址（不同于第一个参数，仅是游戏一启动就创建完毕的属性结构）

第三个参数（r8d）：物品词条种子

第四个参数（r9）:0x0

第五个参数（[rsp+0x20]）：0x0

第六个参数（[rsp+0x28]）：0x0

第七个参数（[rsp+0x30]）：0x0

第八个参数（[rsp+0x38]）：0x0

至此，词条生成函数最终确定为地址处：gamedll\_ph\_x64\_rwdi.dll+18D4EB0，其参数以及返回值也已经分析完毕。最后，我们要解决的一个问题就是，词条生成函数中创建的词条列表我们该如何去释放它，否则我们在后续的调用中如果一直不释放列表，就可能会导致内存越积越多。或者说，游戏会有自己的内存管理，能够自行释放，无需我们手动操作。由于列表是在物品属性结构当中的，我们很容易想到我们可以调用这个结构的析构函数让其进行内存的释放，而析构函数通常会放在虚表函数中的第一个中，所以之后调用完成后我们只需要调用虚表中的第一个虚函数对结构进行析构即可。但是问题来了，我们这个参数中头部不是虚表啊，那我们怎么能获取虚表中第一个虚函数用作析构函数来析构我们的结构呢？其实要找到我们参数结构的析构函数也很简单，在前面我们找构造函数的时候提到过。我们一共找到了2个构造函数，一个只能构造一部分内容，一个则是全都能构造。其中，全都的构造的会调用一部分能构造的。我们以同样的角度来看，全都能析构的函数可能就会调用只能析构一部分内容的析构函数。同时，在我们所找到的第一个构造函数中构造出来的内容中是拥有虚表的，那我们就可以利用这个虚表中的保存在第一个虚函数中的析构函数来反向找到能析构整个参数结构的析构函数。

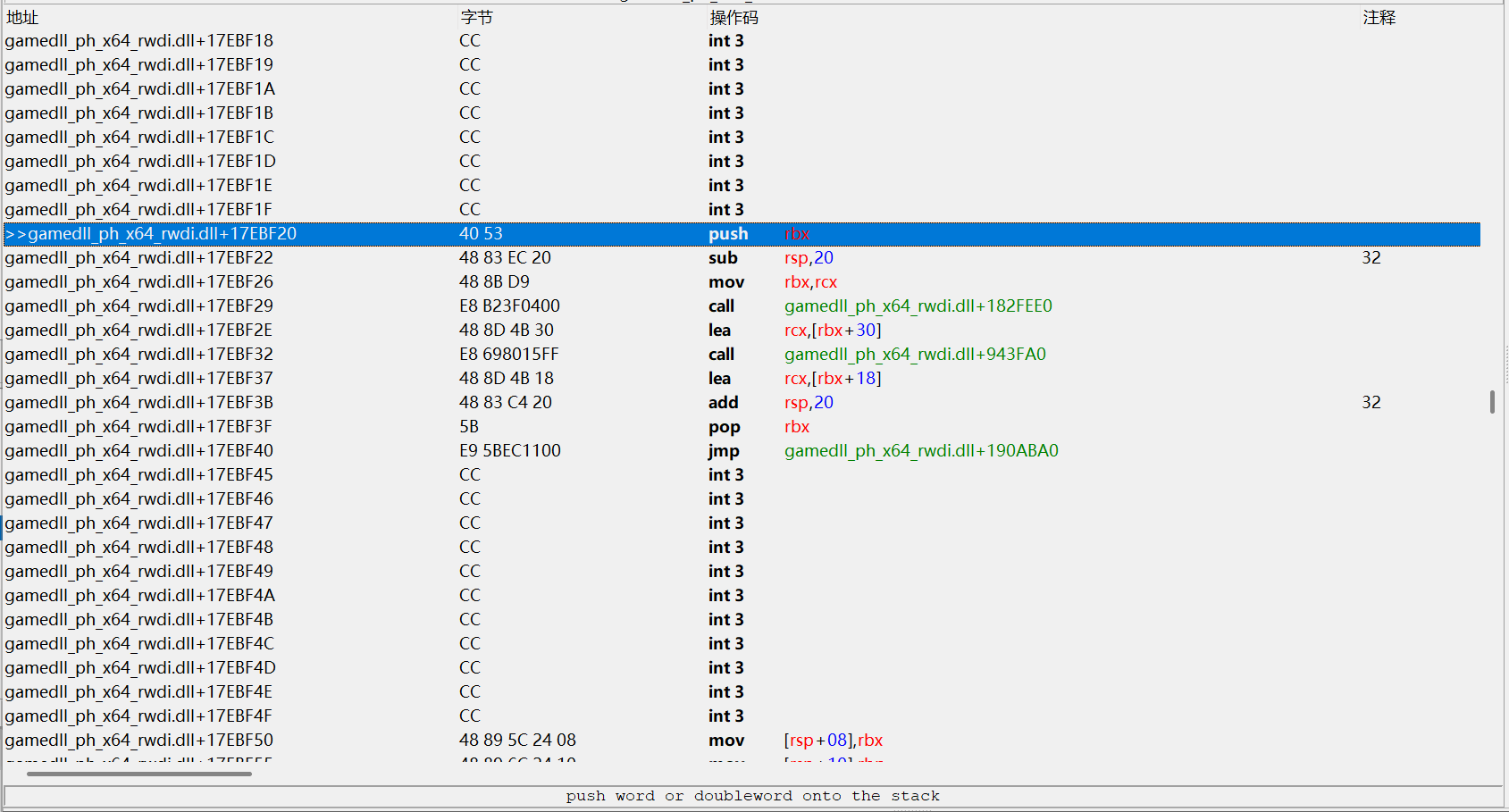


图2.F.33.1-能析构整个参数的析构函数

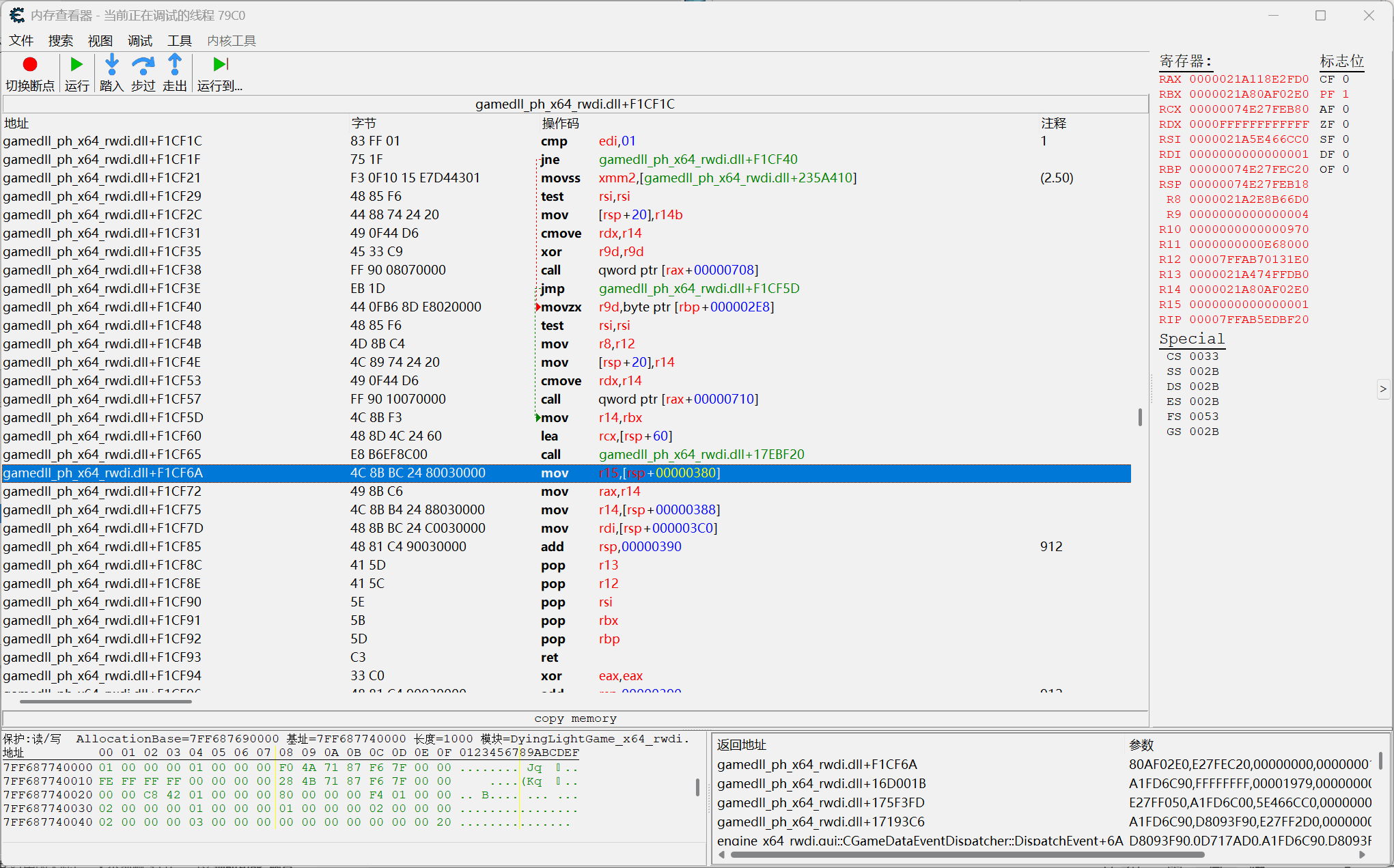


图2.F.33.2-能析构整个参数的析构函数的调用方

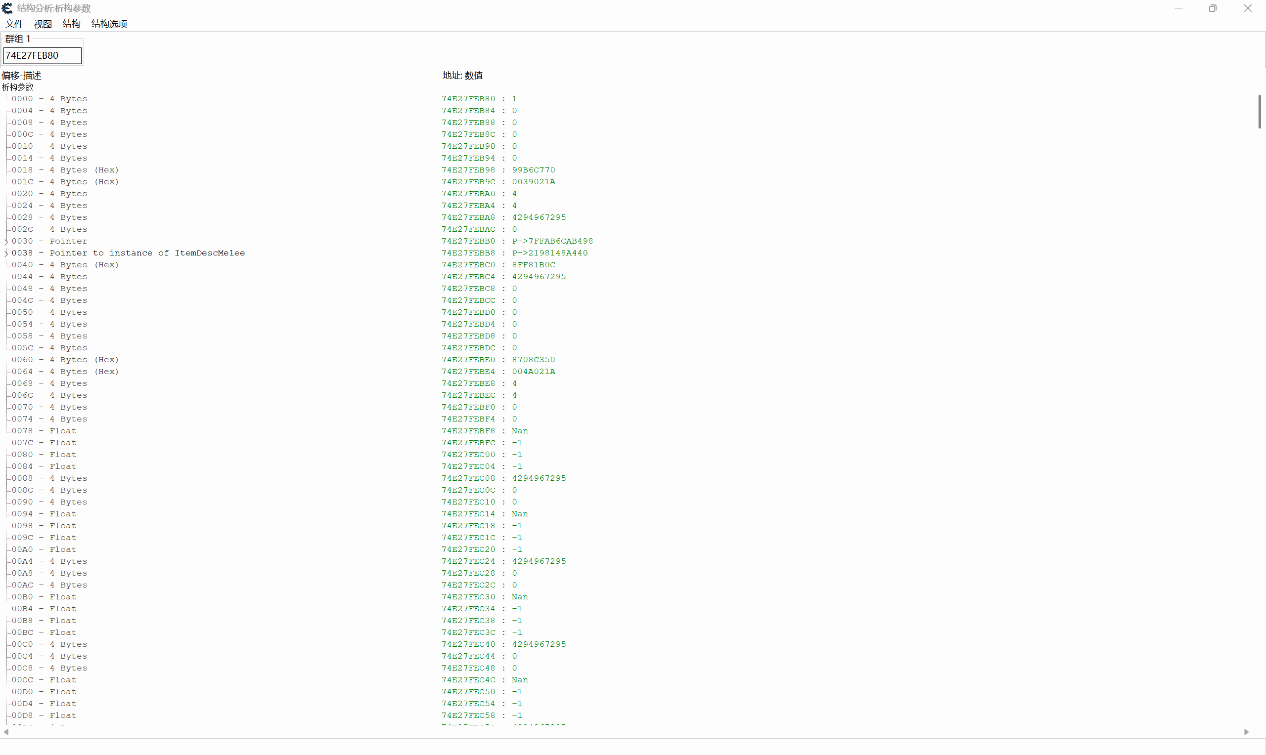


图2.F.33.3-能析构整个参数的析构函数的参数结构

如图2.F.33.1所示，我们从背包中丢弃一样武器，然后通过断点小析构函数（也就是只能析构一部分内容的析构函数）并查看其调用方，我们在函数gamedll\_ph\_x64\_rwdi.dll+17EBF20处发现其符合我们的要求，也就是能析构整个参数。如图2.F.33.3所示，其参数中的结构与我们所在分析的参数的结构完全相同，也表明了其拥有能够析构整个参数结构的能力。同时，我在其调用方，如图2.F.33.2所示，其参数依然只包括一个参数，也就是所要释放的类或结构的地址。虽然这里没有第二个参数，因为普通的在类或结构中的析构函数在编译完成后一般包含着2个参数，虽然在实际编程中我们不能给析构函数传递参数。第一个参数就是所要释放的类或结构的地址，第二个就是是或否的二值参数。第一个参数的意义我们能理解，毕竟需要释放，总要指明释放的对象，那也就肯定需要其地址了。到了第二个参数，其作用其实就是否在析构函数内部释放其内存空间，也就是这个参数可以控制析构函数是否在函数内部调用free函数以释放内存空间。第二个参数一般在调用delete或delete[]时会将其设置为真，以便直接释放内存空间。而在这里却没有指定寄存器rdx也就是第二个参数的值，这就表明这里的析构函数很可能就是类或结构中自行定义的析构函数，并不是传统意义上的定义的函数名前带波浪符号~的析构函数了，也就广义上的析构函数。同时，我们从图2.F.33.2中可以发现，其所给的参数结构的地址其实是在栈中的，这就表明这个析构函数内部不可能存在直接释放内存空间的相关代码，否则一调用栈空间直接没了。最终，我们确定了能析构整个参数结构的析构函数，其地址为：gamedll\_ph\_x64\_rwdi.dll+17EBF20。同时，析构函数一般没有返回值，我们同样忽略其返回值不作分析。

到这里，增添函数的第二个参数rdx中的结构可以说分析完毕了，结构中的其他位置中储存的不是0，就是-1，跟构造函数中的初始化操作赋予的值基本都一样，没有其他分析的价值。最后，我们可以总结出rdx的结构：

struct ItemDesc

{

long long\* VirtualTable;

TextManagerA.Text\* ItemName;

TextManagerA.Text\*\* LocalLanguageSearchText;

……

};

struct AffixText

{

TextManagerA.Text\*\* AffixName;

TextManagerA.Text\*\* AffixLocalLanguageSearchText;

};

struct StatChangeEffect

{

long long\* VirtualTable;

StatChangeEffectDef\* Def;

float AffixValue;

};

struct Affix

{

AffixText\* AffixItemText;

long long Fill;//补充字节所用

StatChangeEffect\* AffixItemValue;

};

struct ItemDescWithContext

{

long long\* VirtualTbale;

ItemDesc\* Item;

int AffixSeed;

char FILL[168];//补充字节所用

float Durability;

Affix\* AffixList;

int AffixAmount1;

int AffixAmount2;

char FILL2[16];//补充字节所用

};

struct Item

{

unsigned int Amount;//物品数量

char FILL[44];//补充字节所用

ItemDescWithContext Item;

Char FILL[116];//补充字节所用

};//0x184 bytes

其中，我们的rdx参数结构长度为338（0x184）个字节。

其中，参数结构中的各个内容的获取在前面也已经提及并解决。现在我们只要模仿上面我们自行构建的结构Item构建出来把它当作第二个参数传递给rdx就可以了。至此增添物品函数的第二个参数分析完毕。也许你会问到：还有其他位置的内容呢？这个参数的内容可是长达0x184个字节的。但我们的回答是，不可能每个位置的内容都分析完毕，这样的工作量将无比巨大，神仙来了都不一定搞得完。但我们尽量要把主要的内容给分析出来，把这些这分析出来就很好了，因为这些内容决定了一件物品的属性，同时游戏程序中也未必完全利用到其他的内容。再次，我们得到了能构造整个参数内容的构造函数，即使有些内容我们未能涉及到，但出于构造函数能帮我们初始化整个参数结构，即使游戏程序调用到了我们未涉及的内容，游戏程序照样能依据构造函数中构造的默认内容正常运行，我们不必为此担心。

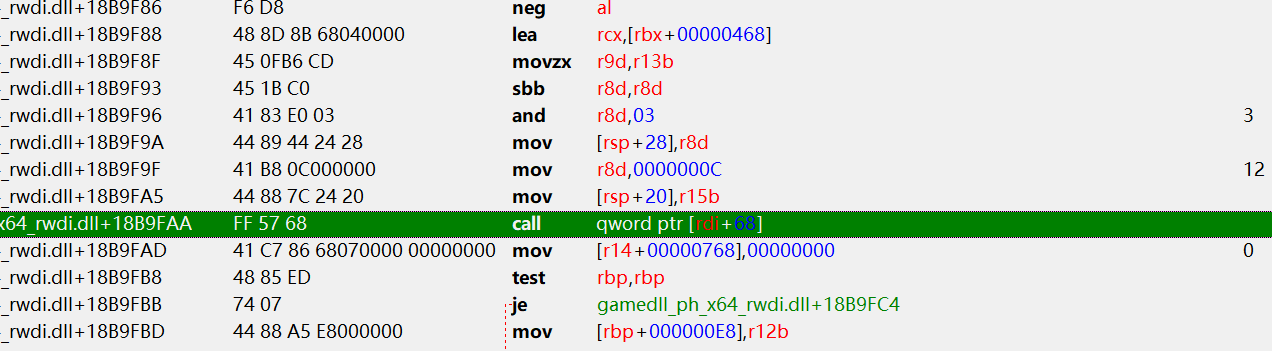


图2.F.34-通过拾取方式获得物品时的调用方

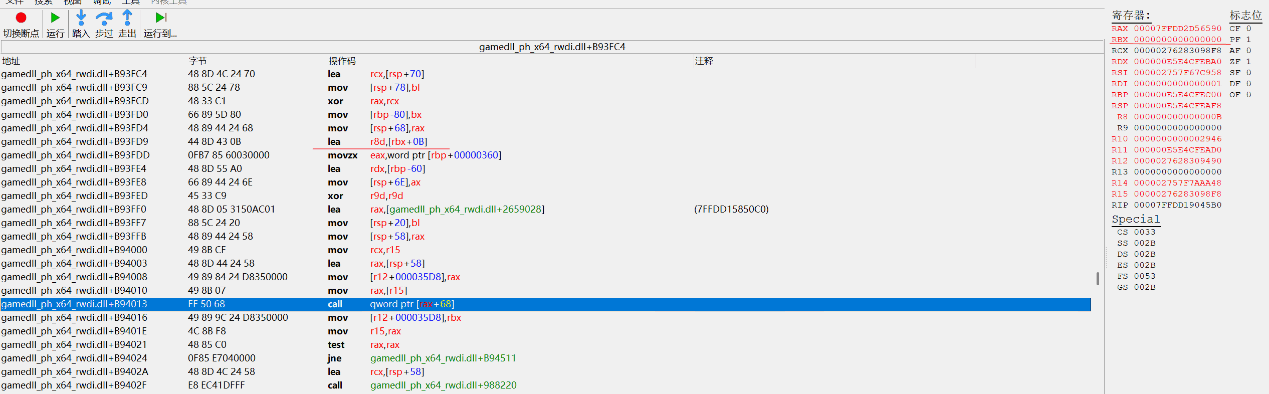


图2.F.35-通过制作方式获得物品时的调用方

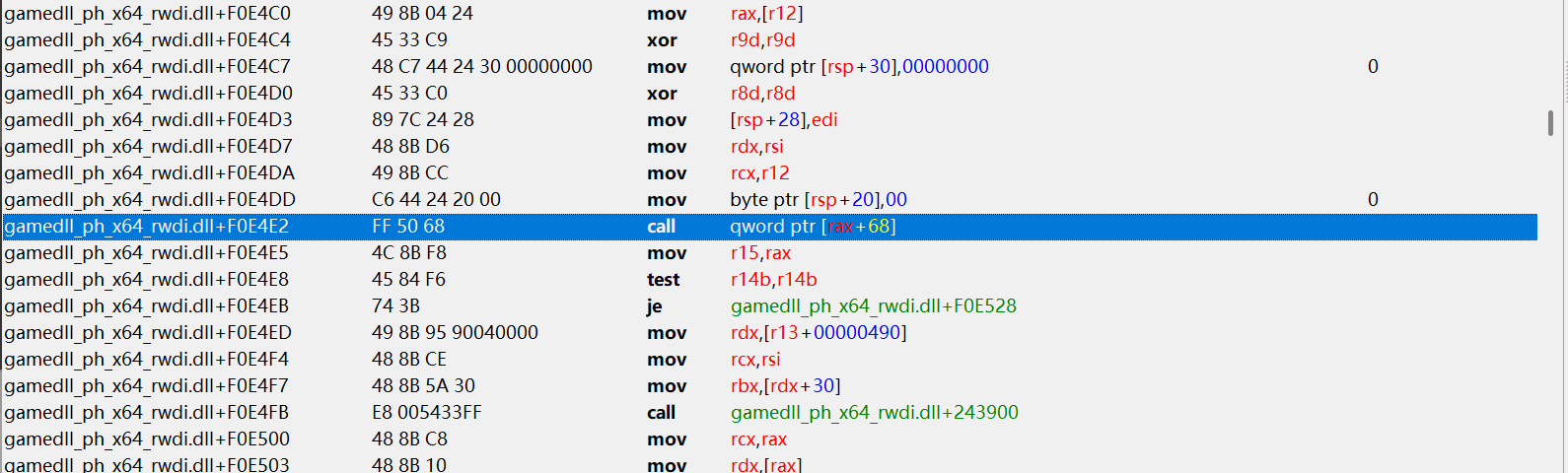


图2.F.36-通过购买方式获得物品时的调用方

现在我们来到第三个参数，分析r8寄存器中的内容。如图2.F.34-2.F.36所示，在我们通过不同方式来获取物品时，其有着相对应的数值。通过拾取的方式获得物品时，参数的数值就是0xC，而通过制作和购买的方式获得物品时，其数值分别就是0xB和0x0。这样一看，这个参数就很简单，就是一个四字节的数据，因为调用方写入时，使用的寄存器是r8d，很明显的四字节类型。这样一来，我们还可自定义获取物品的方式，想通过什么方式获取就定义什么方式来获取。

至此，增添函数的第三个参数r8寄存器中的数值就分析完毕，同时我们可以获得它的数据类型：四字节。下面是它的一些数值代表：

0x0：购买方式

0x2：搜刮方式

0xB：制作方式

0xC：拾取方式

0xE：宝箱开启方式

接下来看到剩下的4个参数。这剩下的4个参数，情况与词条生成函数中的后4个参数的情况一模一样。可以关注上面所有与增添物品函数调用有关的图，可以发现它们调用时基本都将后4个参数全部清零，即直接将它们赋予0x0。因此，在我们调用时只需要我们填充0x0即可，这里就不过多地去陈述分析了。

到这里，增添函数的全部参数就已经分析完毕，参数的获取方式也已经讨论并得到解决。最后我们要关注的就是这个函数的返回值。

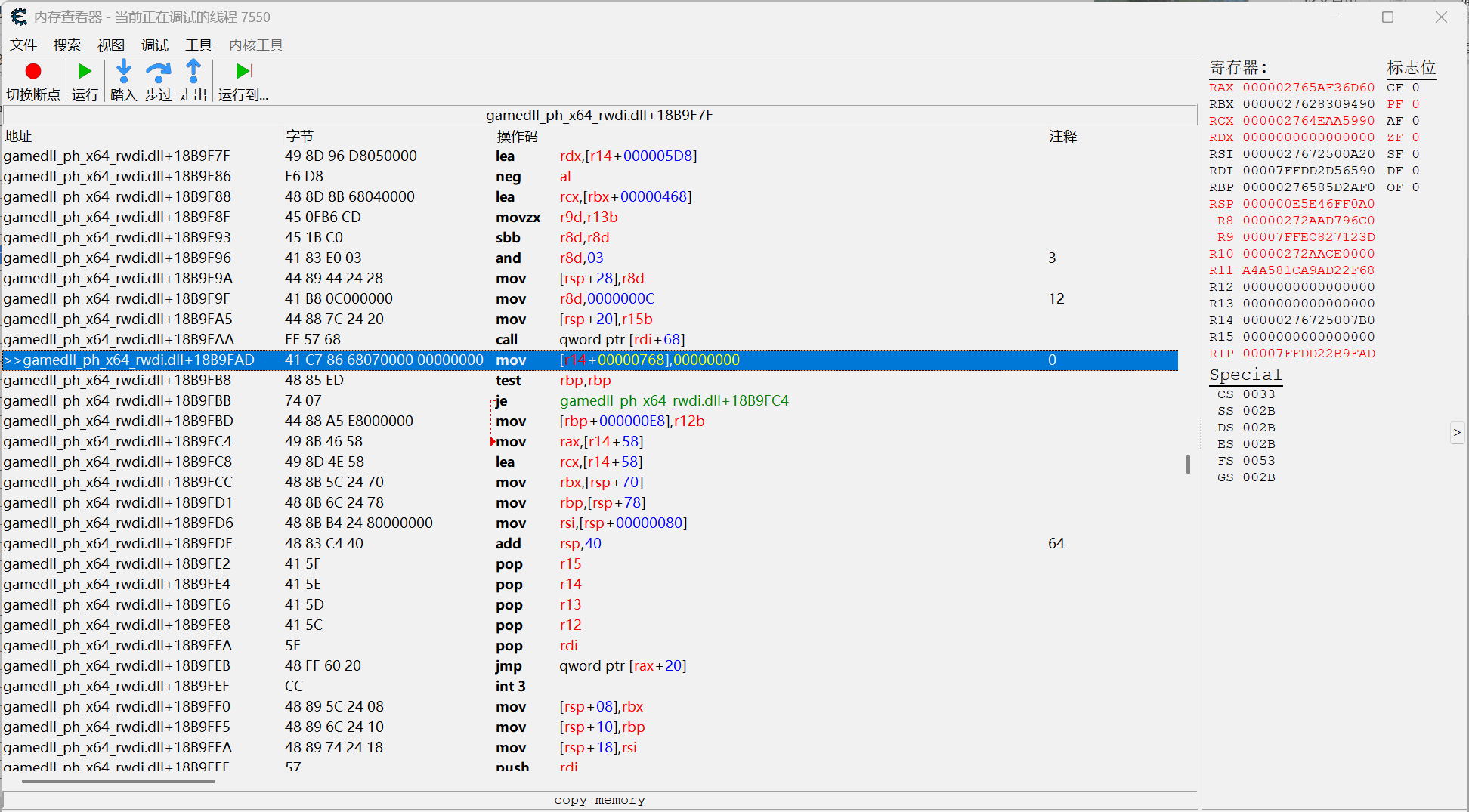


图2.F.37-断点获得返回值，见寄存器rax

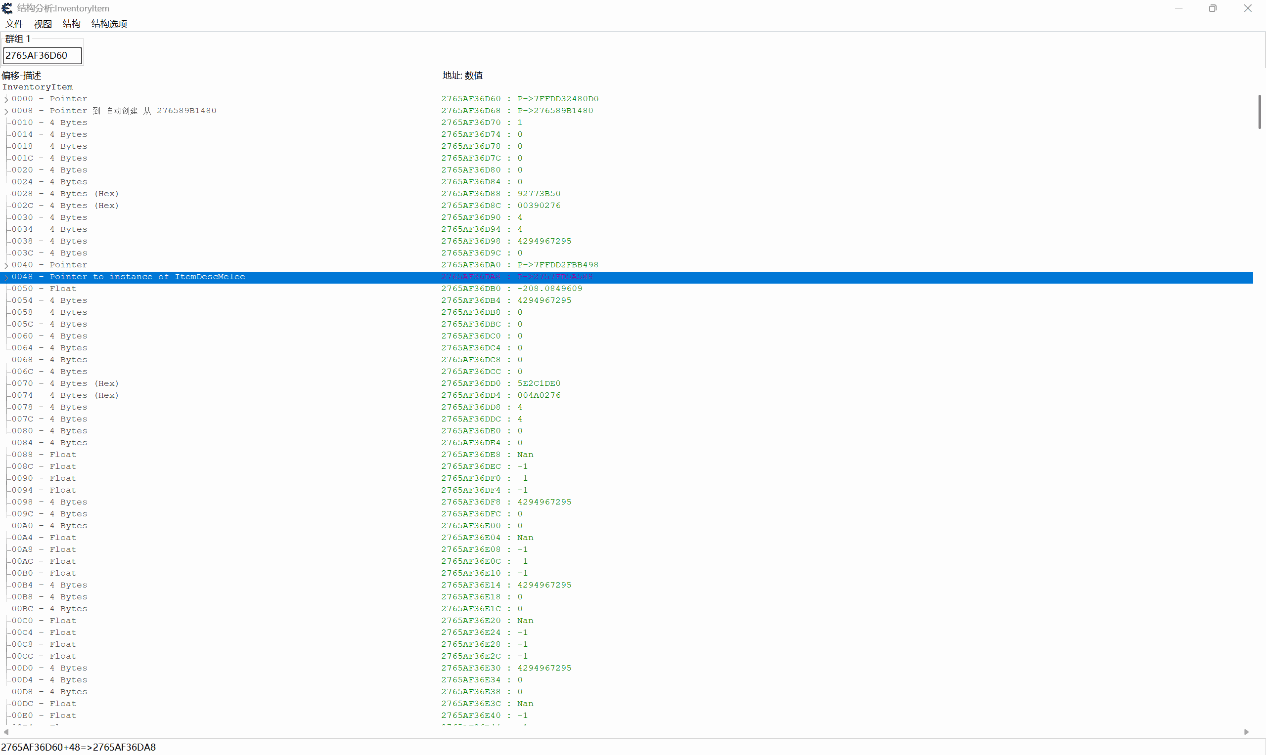


图2.F.38-返回值中的结构

如图2.F.38所示，返回值返回回来的是一个InventoryItem的结构。咋一看，其实这个结构很眼熟，从偏移0x10开始，其实就是第二个参数rdx所要提供的结构。

最后，整个增添物品函数到这里就分析完毕，其参数、返回值都已经被我们晓知，我们直接提供参数调用即可。

**III.编写相应代码制作脚本，完成对刷物品功能的构建**

在正式编写代码之前，我们需要思考的是我们应该怎样去调用增添物品函数。我们可能会想，直接创建线程调用就完事了呗，我们都已经取得函数的地址、函数的参数获取方式、函数的返回值内容了，我们只要正确传递参数正确调用就行了，我们只要想刷物品，就去调用就完事了。但是，我们要细想一下，我们真的只要创建线程直接调用函数就可以了吗？难道不用考虑其他的因素了吗？事实上，直接创建线程调用函数是一个非常危险的行为，其后果就是轻则导致游戏数据异常，变出一些奇奇怪怪的东西出来，重则会导致游戏程序的线程紊乱引发崩溃。

在平常的多线程编写过程中，我们通常会给线程加锁或用其他的方式管理线程从而避免程序资源的使用混乱，特别要涉及到指针的过程，更加要注意。而在这里，作为一款大型游戏，肯定会有相应的多线程管理。如果此时我们直接冒然创建线程调用函数，就肯定会打乱游戏中多线程的管理。因此，在这里，我们不能直接创建线程去调用函数。

那么，现在我们要探讨的就是我们应该如何去调用函数。上文我们提到游戏程序中会存在线程管理。我们的第一个想法可以能就是找到游戏程序中相关的代码，试着能不能自行把调用函数的任务加入进去从而达成目的，且不会影响游戏内本来就存在的线程。但这种想法的实现实际上非常困难，首先我们就不知道如何去找到相关的代码，这是完全没有想法的。再次就是这里是反汇编，要我们在反汇编中去管理线程，这将十分困难，这本来在平常的中编程就已经十分困难了，更别说是在反汇编中操作了。所以，我们需要的是一个简单而有效的想法和做法。

我们可以换个想法，既然我们很难甚至是无法在游戏程序的多线程管理中加入我们自己的任务，那我们可不可以将游戏程序中本来已经存在的任务替换为我们自己的任务呢？我们直接将调用增添物品函数所在的线程执行的某个函数给替换成我们自行编写的函数，然后在我们自己的函数内部先调用原本线程要执行的函数，接着有需要再调用增添物品函数实现刷物品功能。这样一来，我们就利用了游戏中原本就存在的线程，从而最大程度地避免了因外部创建线程而导致的线程紊乱问题。现在，我们就可以去寻找这样的函数，将它替换为我们自己的函数，最好是找到一个循环调用的，这样我们就可以在外部做出指令，等到我们需要刷物品时再进行调用。

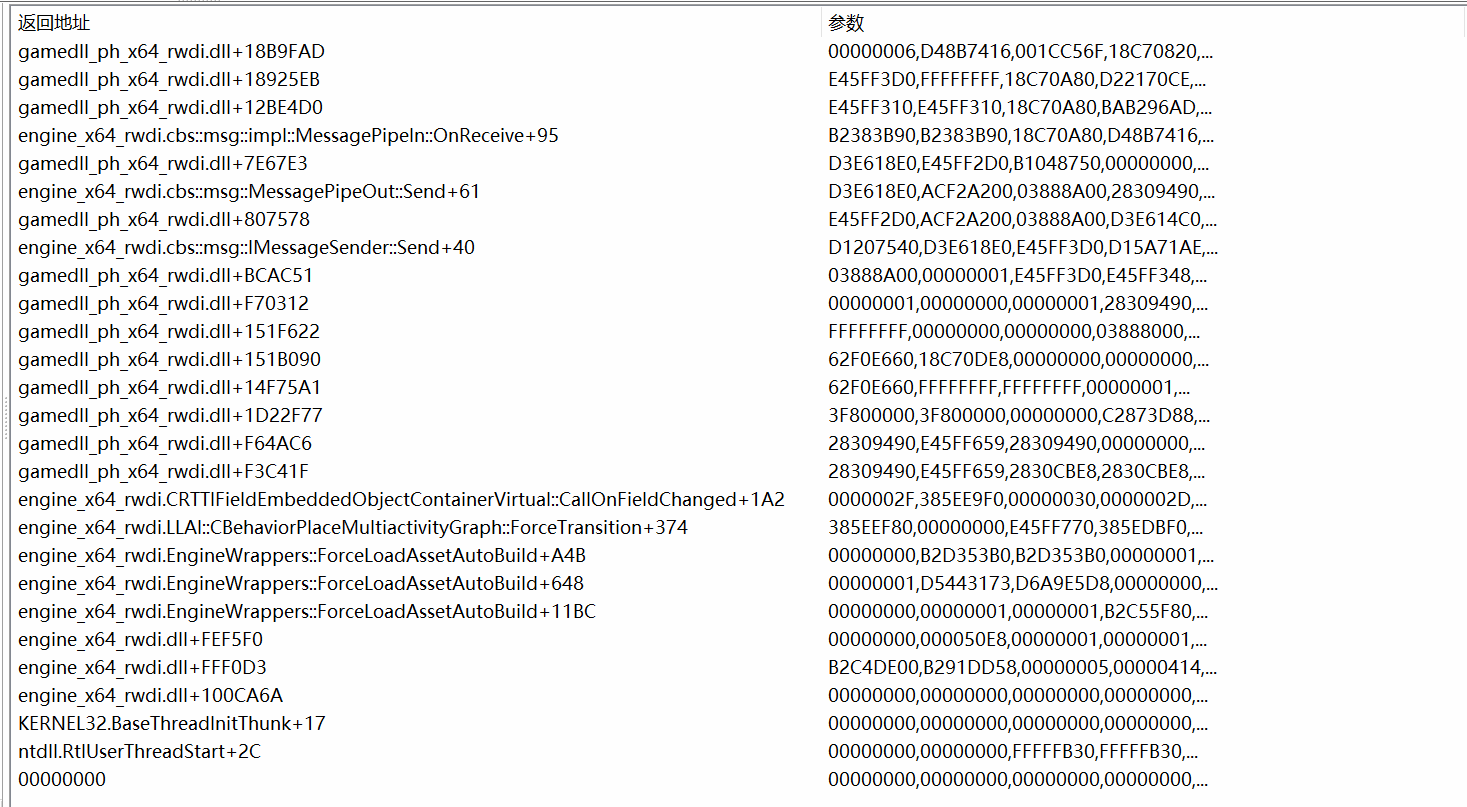


图3.1-在增添物品函数处下断点，拾取物品后得到同一线程的调用栈

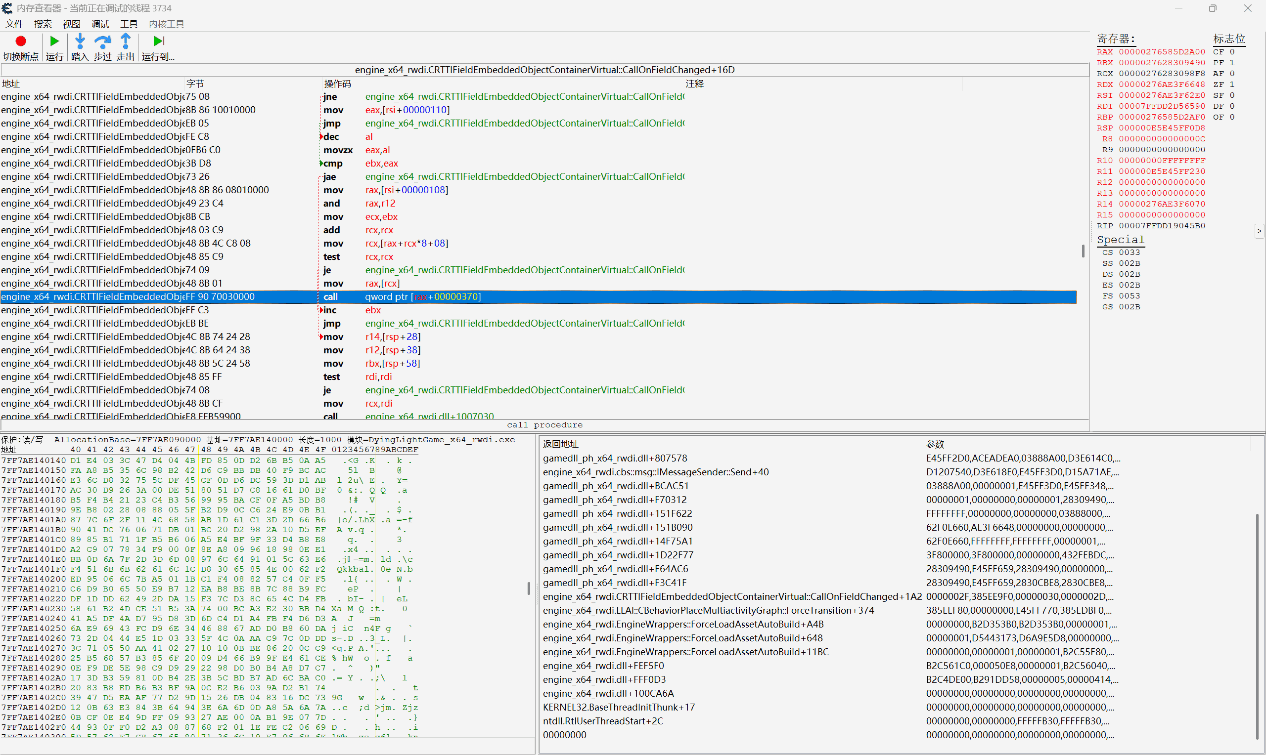


图3.2-切换处代码

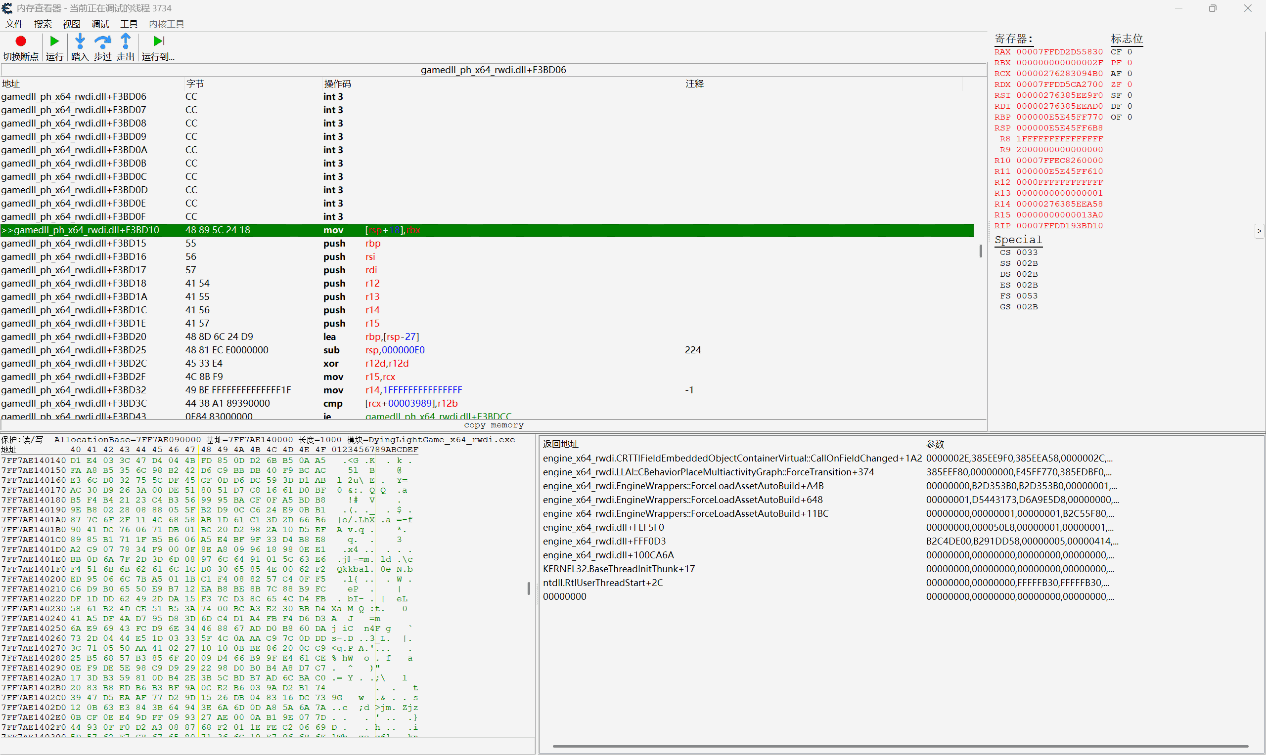


图3.3-在切换处调用增添物品函数的同一级函数头断点

如图3.1所示，我们得到增添物品函数的调用栈。看到模块engine与gamedll的切换处engine\_x64\_rwdi.CRTTIFieldEmbeddedObjectContainerVirtual::CallOnFieldChanged+1A2。在这里，游戏线程会一直循环调用此处，我们可以在此处将函数给替换。如图3.2所示，此处调用函数的方式还是虚函数的调用方式，这样就更加地方便了，只要我们把虚表中的函数地址换成我们自己的函数地址就能实现函数的替换。需要注意的是，我要替换的虚函数是要未来会调用增添物品函数的，也就是图3.2时所调用的虚函数。

我们在与增添物品函数同一级的函数头中断点，得到对应我们将要替换的虚表，也就是rax中的值：7FFDD2D55830，换算成相对地址也就是gamedll\_ph\_x64\_rwdi.dll+2355830。我们在虚表中替换的位置也就是要加上偏移0x370，也就是要替换gamedll\_ph\_x64\_rwdi.dll+2355BA0中的地址。下面我们就来编写一个虚函数替换函数，便于将我们的自定义函数给替换进去。

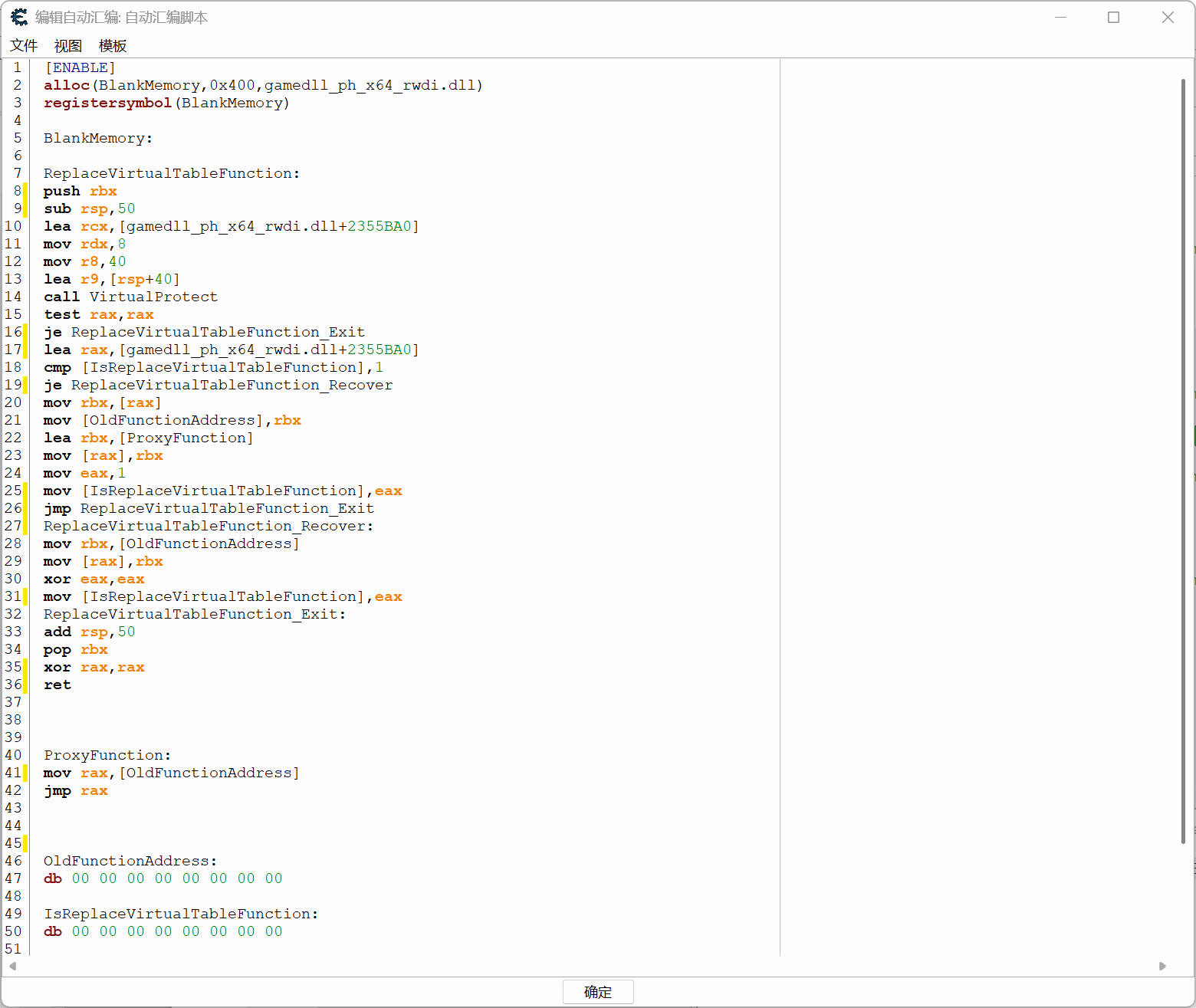


图3.4-编写虚表函数替换代码

如图3.4所示，我们编写一个名为ReplaceVirtualTableFunction的函数用于将虚表函数给替换成我们的自定义函数ProxyFunction。首先，还是像编写虚表复制函数那样，在开头先抬栈和保存我们需要用到的易失寄存器，栈的抬升需要迎合一切可能会遇到的16字节对齐的情况。之后我们直接引用相应的虚表地址，将其的访问权限改成可读、可写、可执行，便于我们将地址给替换掉。为防止意外的权限更改，因此在这里不管有没有更改虚表，是否恢复虚表，都先将权限维持可写的状态，因此这里将VirtualProtect的调用放在前面。之后函数返回，如果更改失败，就直接跳转到ReplaceVirtualTableFunction\_Exit处直接退出更改函数，如果更改成功，则进入是要恢复虚表还是要更改虚表的判断，判断的依据就储存在变量IsReplaceVirtualTableFunction中，如果变量的值是1，则表明需要恢复虚表，反之则要进行虚表函数的替换。判断完成之后，根据结果跳转到需要处理的代码当中，将相应的函数地址写入到虚表当中。我们把原来的函数地址给写入到变量OldFunctionAddress中，以便后面恢复虚表的函数地址时使用，同时也为了在我们的自定义函数中能够执行到原本的函数代码，不破坏游戏程序原本的代码执行。操作完成之后，统一恢复栈指针，随后返回数值0结果表示线程的正常退出。

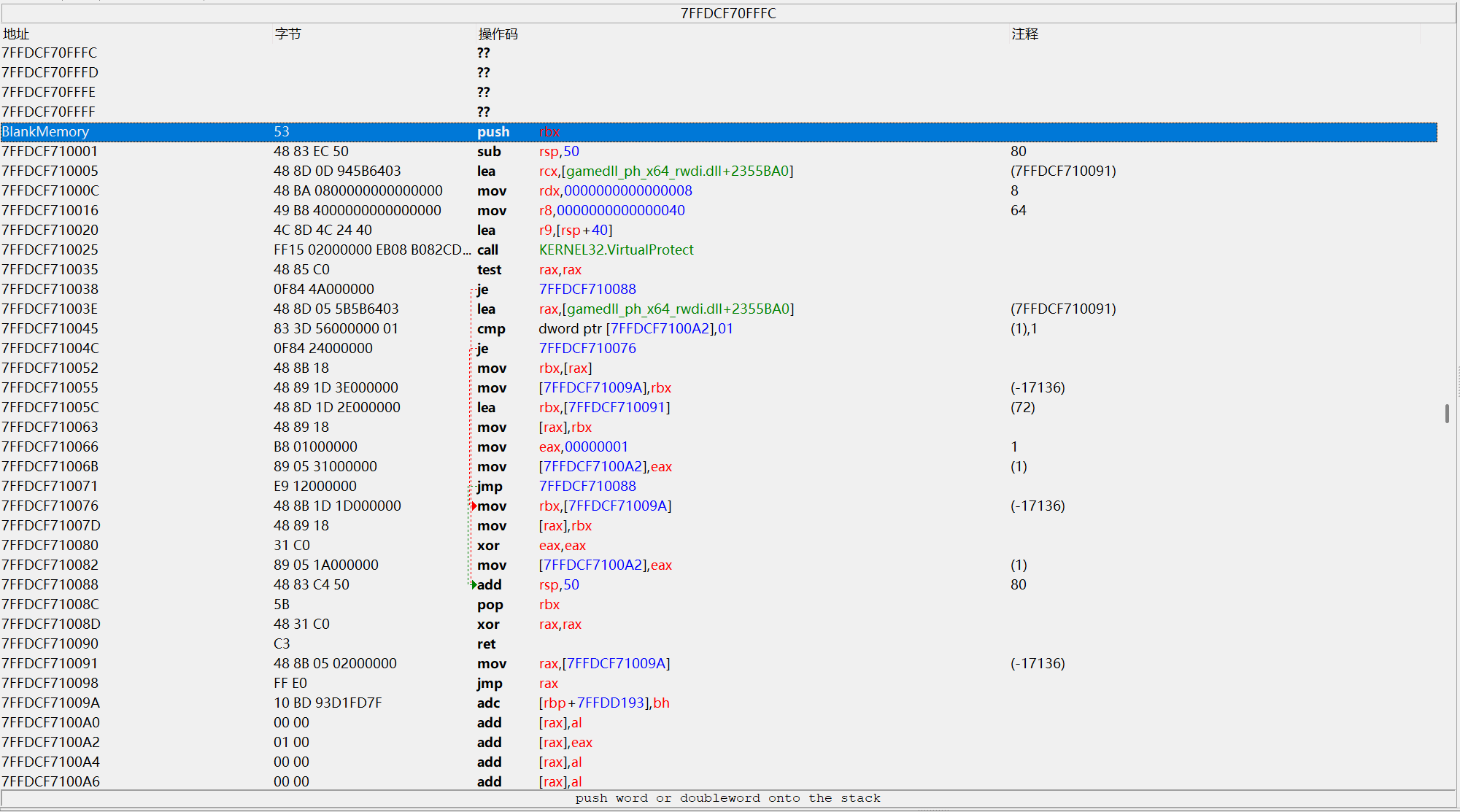


图3.5-检查编译后的汇编代码

我们激活脚本，确保我们的代码能够被正确地编译。检查编译的代码无误之后，在函数头部创建线程，测试我们的代码是否有效。在这一切完成之后，接下来我们就来构建脚本的各种功能。

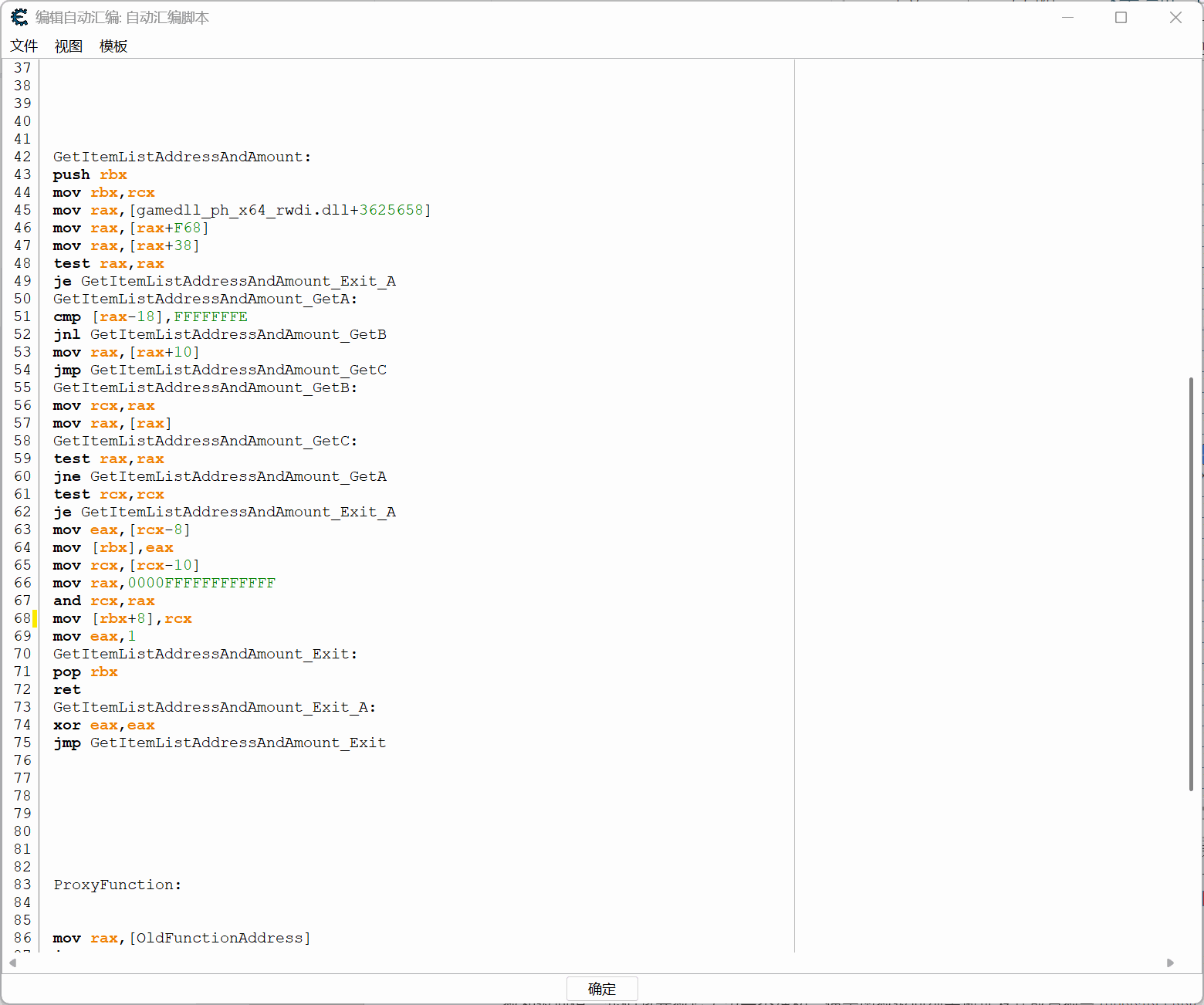


图3.6-编写物品属性结构列表获取函数

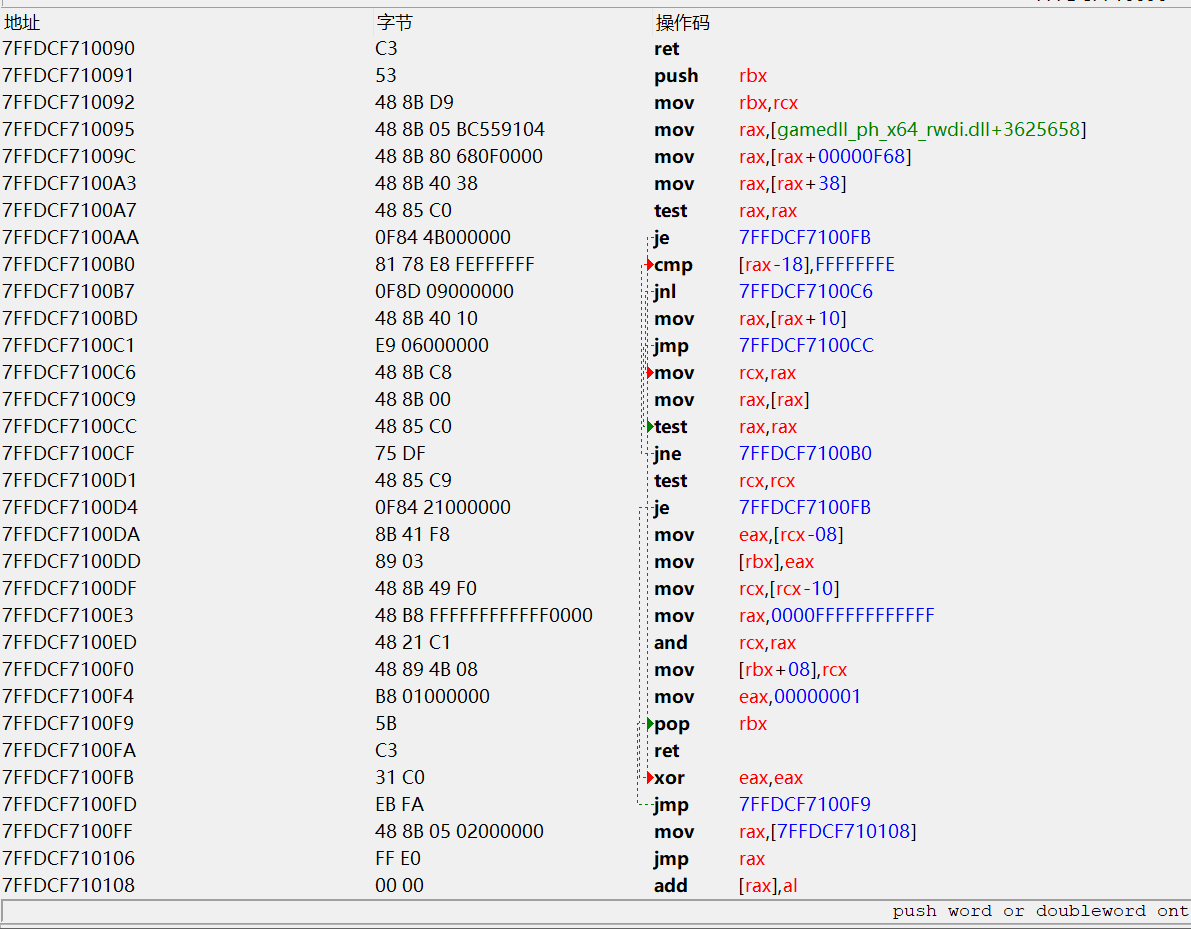


图3.7-列表及其成员数量获取函数编译代码检查

如图3.6所示，这里我们设置一个函数用于获取物品属性列表及其成员数量。我们将获取列表函数名称设置为GetItemListAddressAndAmount，并且将其设置一个参数和返回值。我们将参数设置为一个结构，便于函数返回列表地址及其成员数量。其参数和返回值如下：

struct ItemList

{

int Amount;

int FILL;//补足字节所用

ItemDesc\* ItemList;

};

第一个参数（rcx）：ItemList&

返回值（eax）：获取成功：0x1；获取失败：0x0

回到我们的代码中，我们按照如图2.F.3中的函数代码获取列表地址。由于这个函数中没有调用任何其他的函数，同时函数中没有任何需要16字节对齐的代码，因此我们在函数开头不再进行栈内存对齐的操作，只将我们需要利用的易失寄存器中的值给保留在栈中。在函数开头，我们先把通过rcx传递进来的参数保存在寄存器rbx中，便于之后的取用。随后，由于图2.F.3中的函数开头中的call的函数只有一句短短获取基址的指令，因此我们在这里直接读取而不调用相同的函数获取。获取完基本的东西之后，我们就一对一地按照图2.F.3中的代码进行模仿。接下来，在获得列表地址及其成员数量之后，我们把地址和成员数量分别对应地写入参数的结构当作完成对列表地址及其成员数量的获取，同时将返回值设置为1，表示获取的成功。反之，将返回值设置为0，表示获取的失败。

在函数编写完成后，我们同样激活脚本，到反汇编去检查此函数，考察其我们的代码是否被正确地编译在内存中，如图3.7所示。

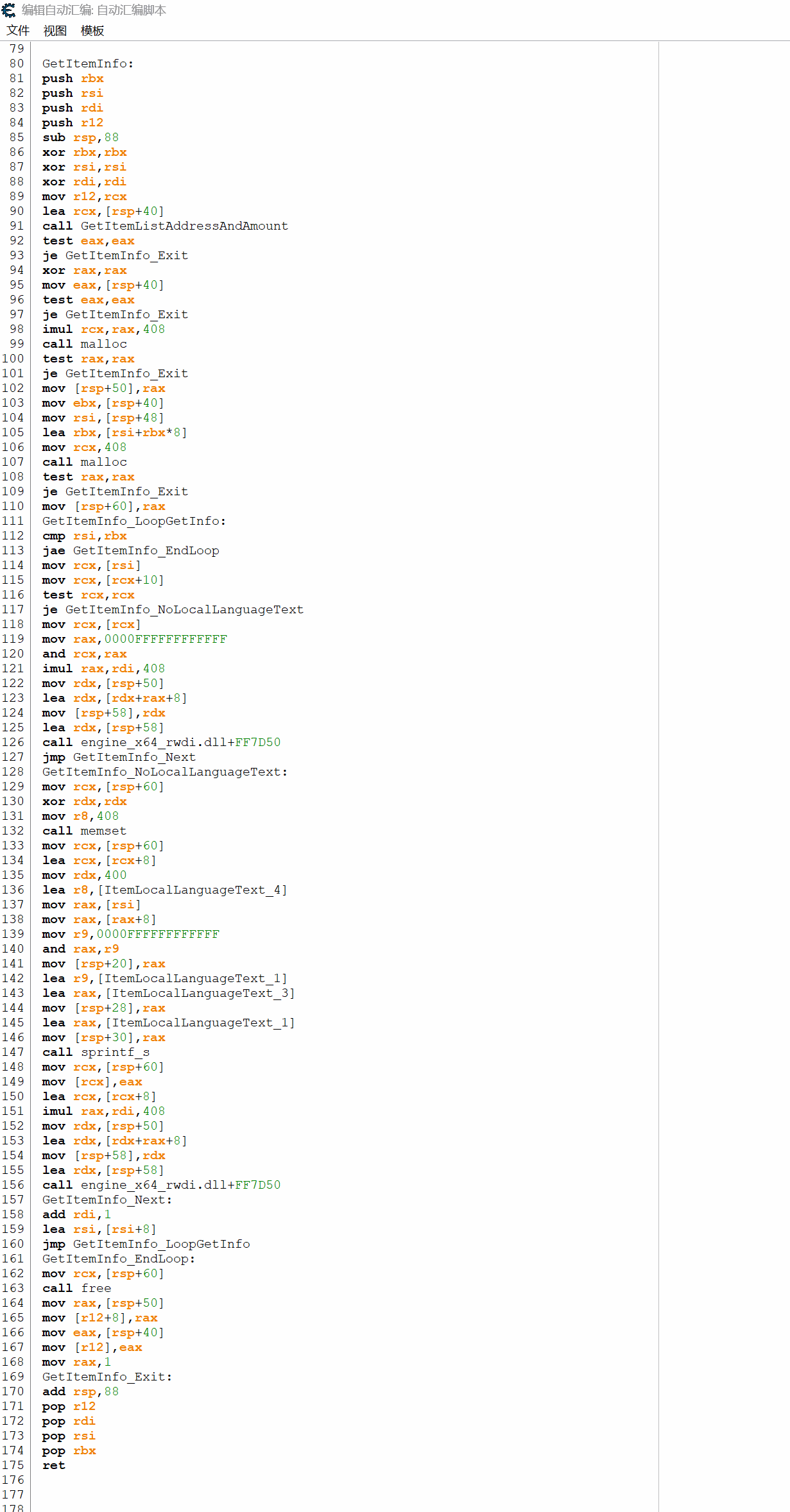


图3.8-编写物品名称获取函数

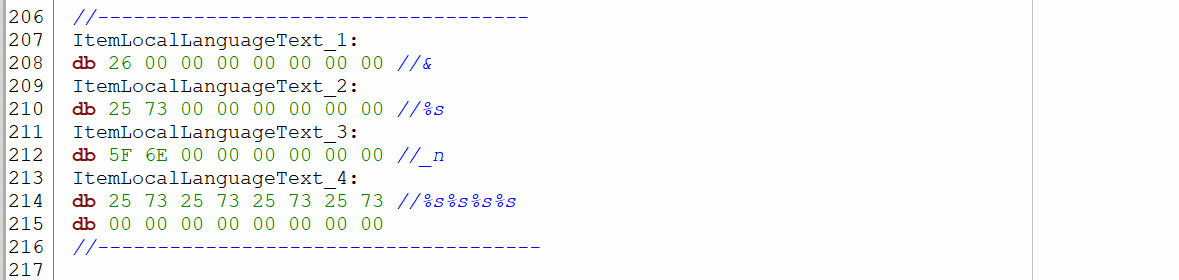


图3.9-物品名称获取函数中需要用到的文本内容

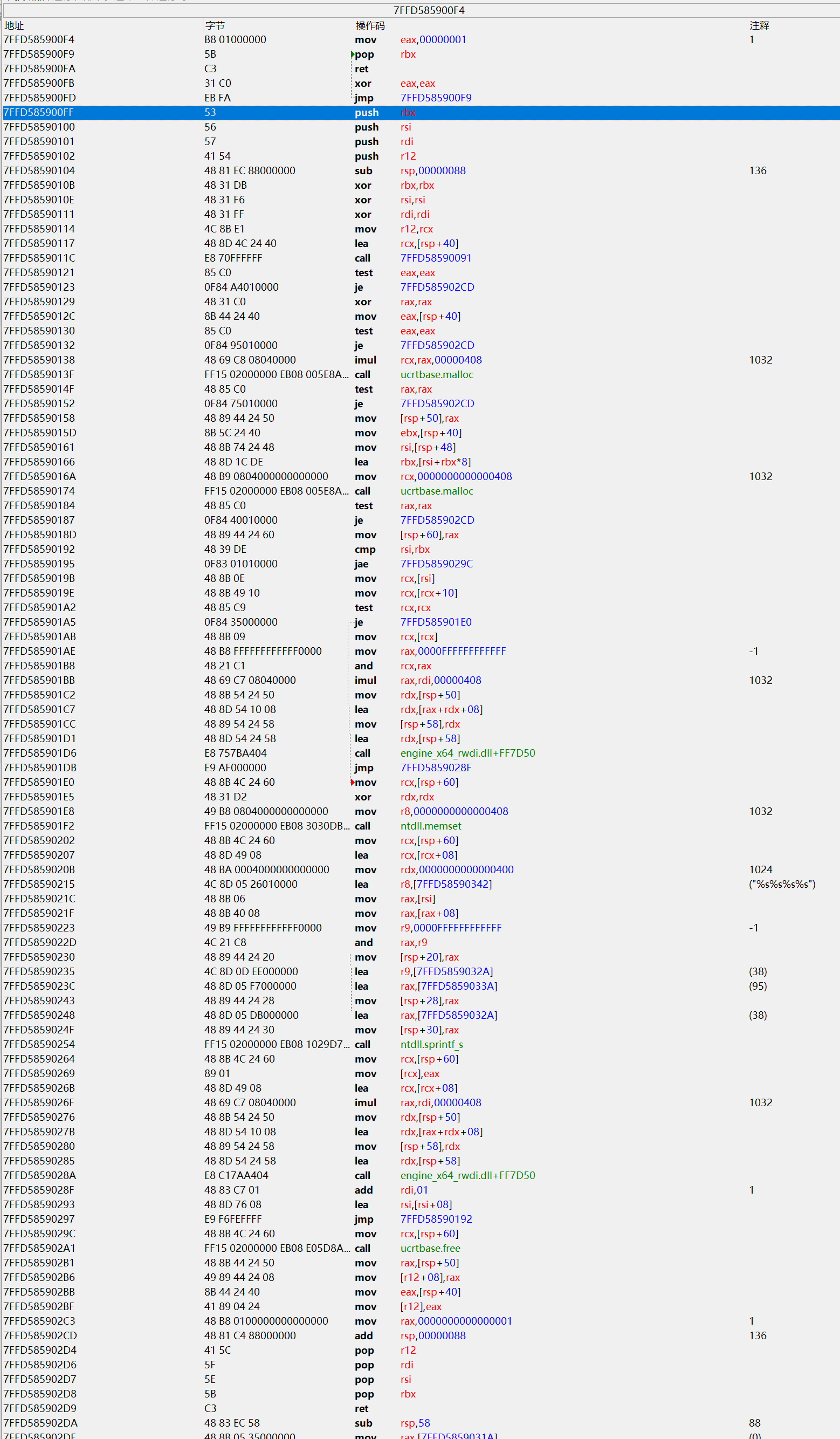


图3.A-检查获取名称函数是否被正确编译

如图3.8所示，我们需要编写一个物品名称获取函数，方便我们之后能够根据物品的名称来决定自行想要刷取的物品。我们将此函数的名称设置为GetItemInfo，意为获取物品信息，也就是获取物品名称。我们将此函数设置为一个参数以及一个返回值，便于我们保存获取到的名称和获取成功或失败的结果。我们将参数也同样地设置为一个结构，便于函数中保存名称和名称的数量，其参数和返回值如下：

struct ItemNameText

{

int NameLen;

int FILL;//补足字节所用

wchar\_t Name[512];

};

struct NameList

{

int Amount;

ItemNameText\* List;

};

第一个参数（rcx）：NameList&

返回值（rax）：获取成功：0x1；获取失败：0x0

回到我们的代码中。如图3.8所示，在开头，依旧老样子，保存我们要用到的易失寄存器和确保栈的16字节对齐，因为之后我们需要调用到其他的函数，如本地化语言查找函数和文本格式化函数。之后我们开始先将要使用到的寄存器给清零，避免数据的异常。然后我们利用前边所写的获取物品属性列表地址函数将列表地址及其成员数量获取到并将它们分别保存在临时变量[rsp+48]和[rsp+40]中，方便之后读取里面的本地化语言查找键获取名称。接下来，我们申请一段内存用于下面的本地化查找键的构建。由于一些物品的物品属性中是没有本地化语言查找键的，只有其物品代码文本，而这些物品利用格式 &+物品代码+\_n+&构建本地化语言查找键之后又能够查找出其名称，所以需要我们自行构建。内存申请操作完成后，我们判断其是否申请成功，如果不成功就直接退出函数表明获取的失败，成功则继续往下操作并将这段临时文本操作内存保存在临时变量[rsp+50]中以便往后的调用。再往下，我们读取保存在临时变量中的物品属性列表地址及其成员数量，计算出我们总共需要多少内存空间保存所有物品的名称文本。我们假设物品名称文本最长不到512个字符，就给每个物品申请出512个字符的内存空间，同时本地化语言查找函数返回的结果是宽字符的，那么实际上每个物品所要的名称文本保留空间就是512\*2=1024个字节，也就是0x400个字节，加上我们需要额外多个8字节空间来保存名称长度以及构建本地化语言查找函数所能接收的参数结构，最终加起来每个物品所要提供的内存空间大小就是0x400+0x8=0x408个字节。使用指令imul乘上物品的数量，就得到了总要申请内存的大小。接着，老样子，判断申请内存是否成功，如果不成功就直接退出函数，返回0x0值表示获取的失败，如果成功则把这段内存保存在临时变量[rsp+60]中以便往后的调取。同时，我们通过地址计算将列表尾的地址给计算出来并保存在寄存器rbx中并以此来判断列表访问的位置是否已经到达末尾，如图3.8中的第112行所示。再每一次正式执行名称获取的操作之前，我们都将进行位置的判断，如果列表已经访问完毕，我们就跳转到GetItemInfo\_EndLoop的位置退出循环，并将我们在循环中获得的一系列文本所储存在的内存的首地址及其数量给写入到我们传递进来的保存在寄存器r12中的参数中，并将返回值设置为1表示获取的成功。如果列表还没访问完毕，就接着进入循环，首先判断改位置中的物品属性中是否存在本地化语言查找键，如果不存在，则进入GetItemInfo\_NoLocalLanguageText的位置进行本地化语言查找键的构建。这里我们使用文本格式化函数sprintf\_s进行构建。构建完成后，我们根据保存在寄存器rdi中的访问位置来计算得出物品对应的文本储存位置，同时作为参数传递给本地化语言查找函数完成对没有本地化语言查找键的物品的名称获取。如果存在本地化语言查找键，我们就直接把本地化语言查找键文本所在的地址给读取出来，将其高16为清零后当作参数传递给本地化语言查找函数用以获取物品名称文本。在结束访问后即退出循环并完成获取结果的写入操作。最后将寄存器中的值恢复并将栈恢复到原来的位置并返回至调用方，同时结束获取的操作。

还是老样子，在结束了我们的代码编写后，我们需要激活脚本，来到反汇编区域，检查我们的代码是否被正确地编译，如图3.A所示。

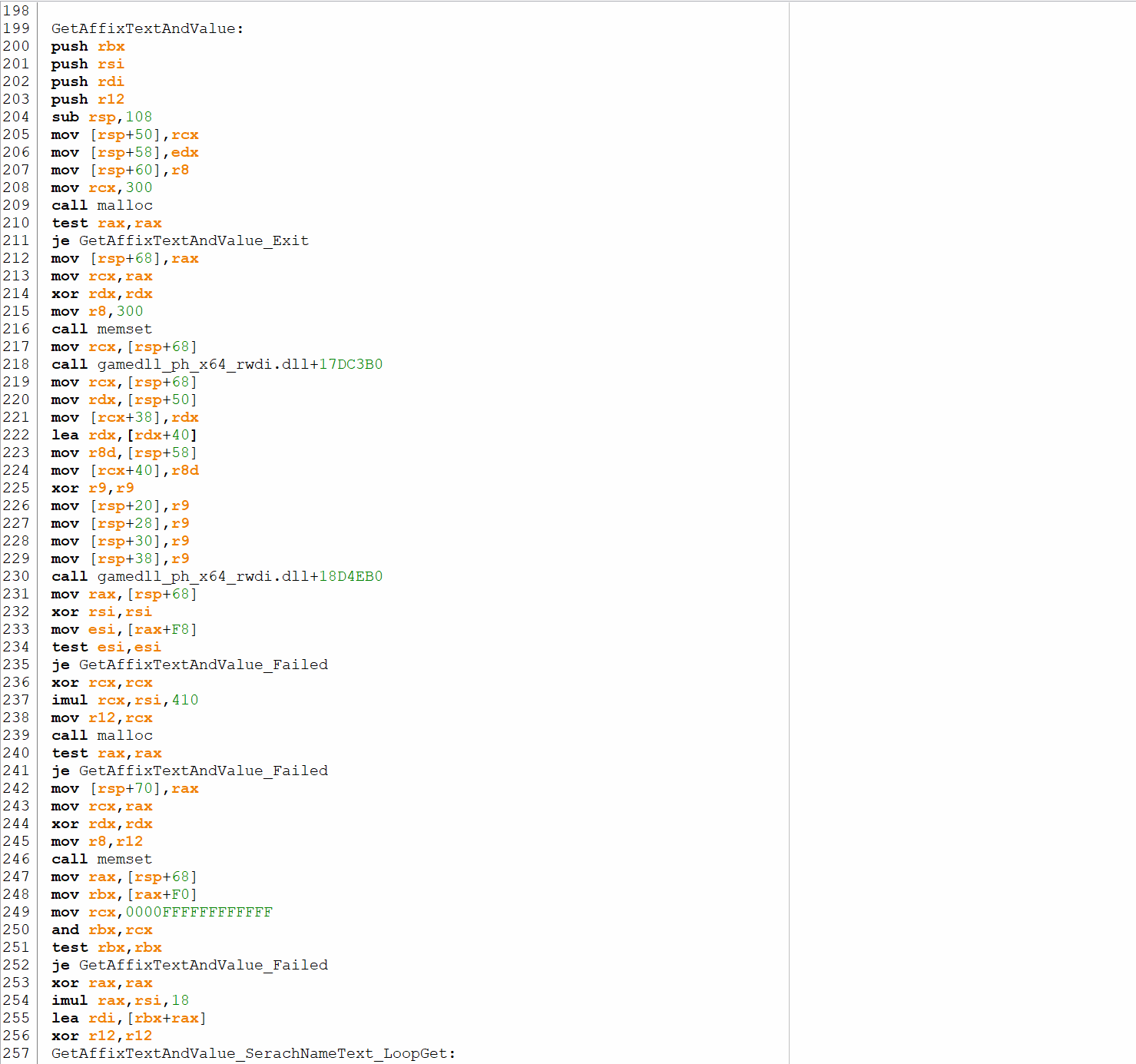


图3.B-词条及数值文本获取函数

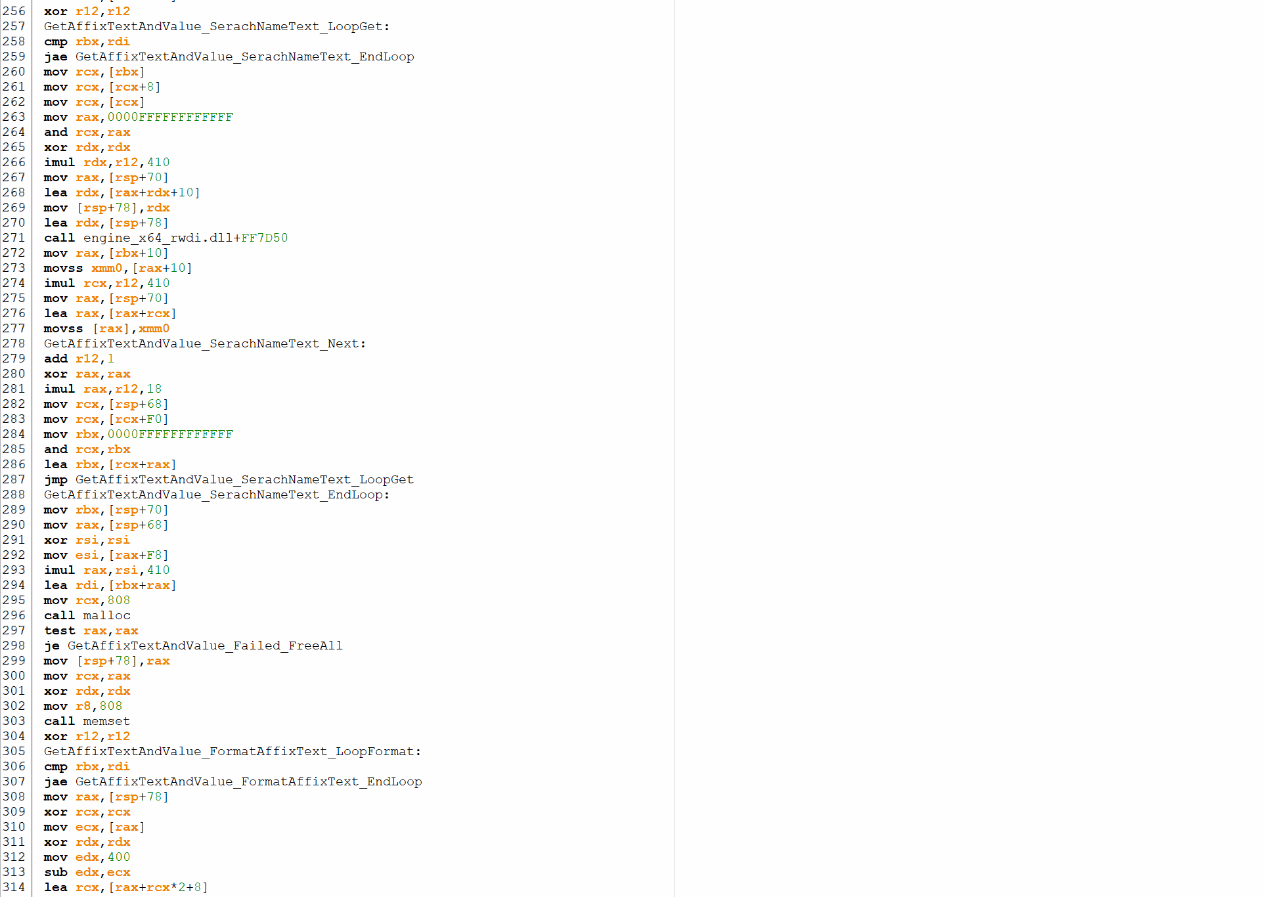


图3.C-词条及数值文本获取函数（续上图）

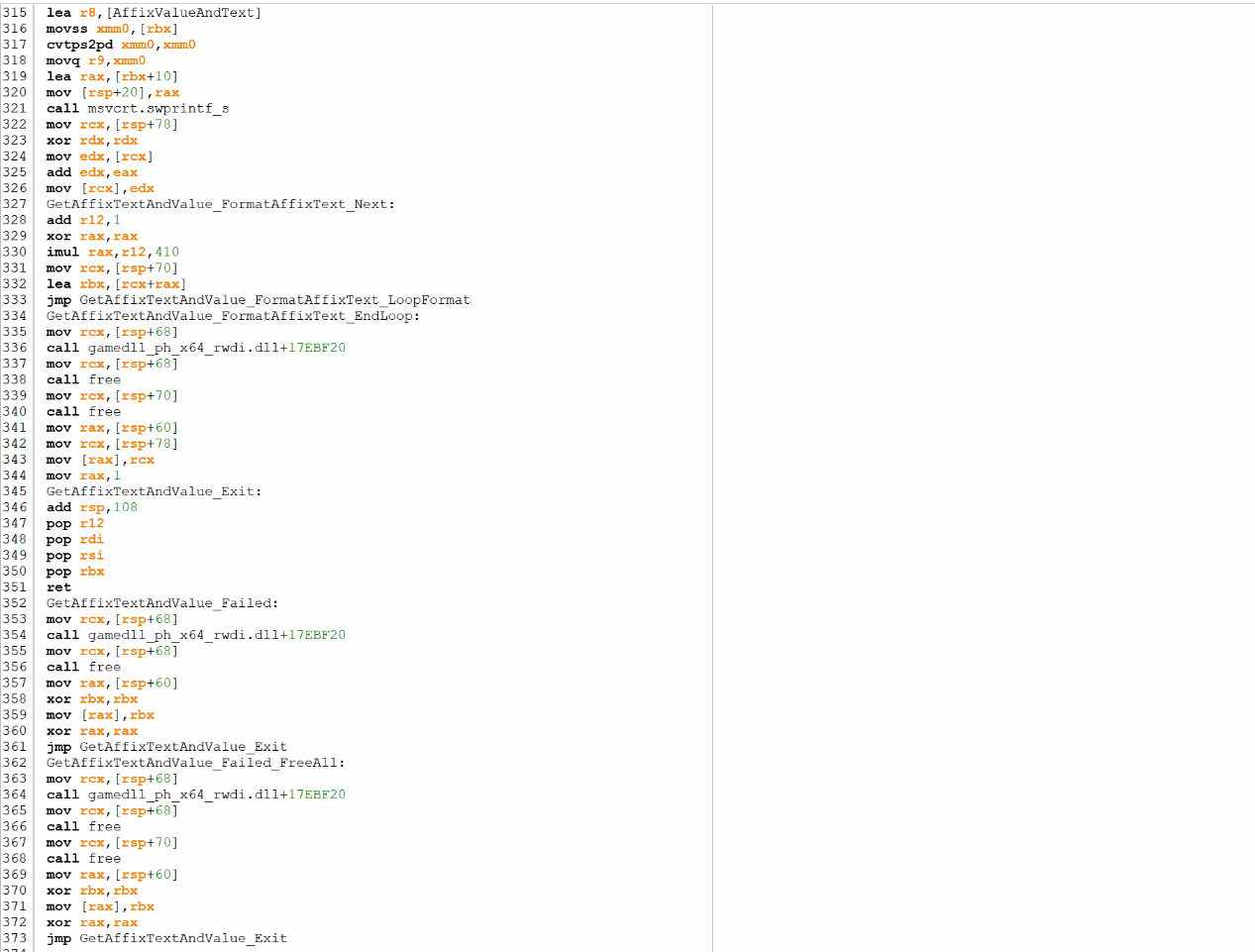


图3.D-词条及数值文本获取函数（续上图）

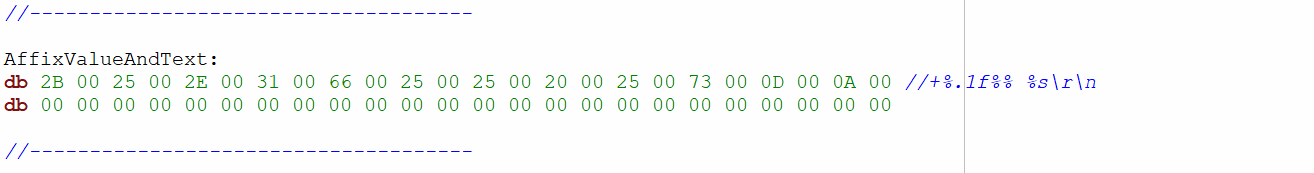


图3.E-词条及数值格式化文本

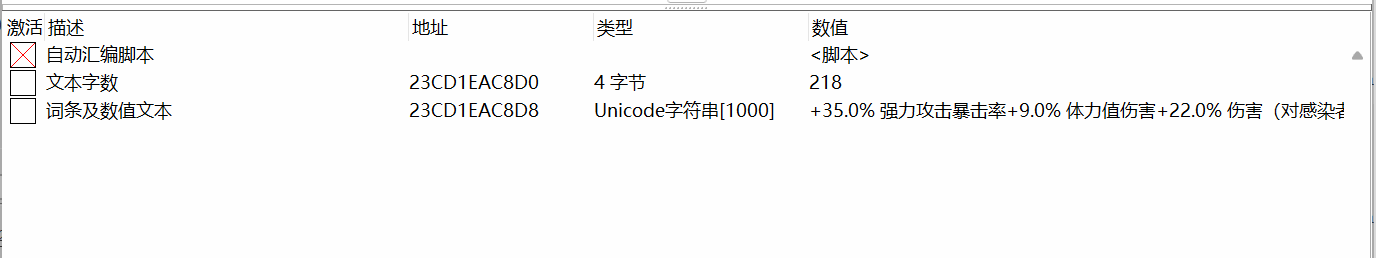


图3.F-词条及数值文本获取函数返回内容

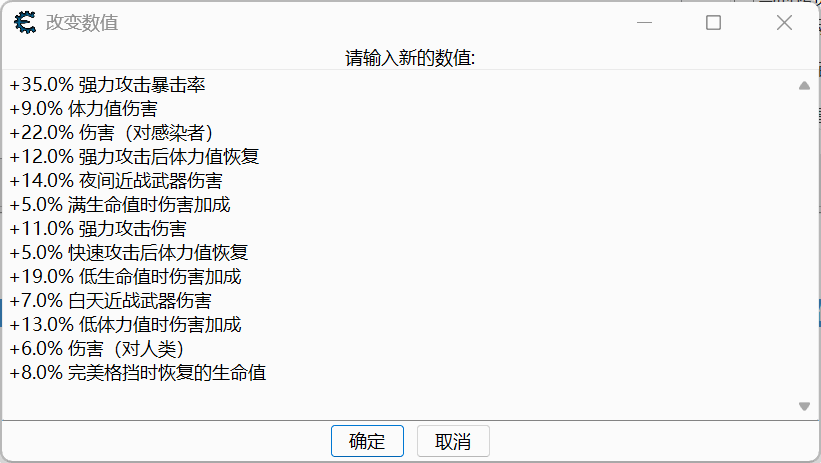


图3.F.1-词条及数值文本获取函数返回的文本效果

如图3.B-图3.E所示，这里我们需要编写一个词条文本及数值获取函数，目的就是为了方便我们在正式刷取物品之前可以根据自己提供的种子来提前预览将要生成的词条内容，方便我们自行刷取想要的词条内容。这里将函数名称设置为GetAffixTextAndValue，就名称上的意为获取词条文本和数值。同时，将函数设置为三个参数和一个返回值，其参数和返回值如下：

struct AffixText

{

int Len;

wchar\_t Text[];

};

第一个参数（rcx）：物品属性结构首地址（游戏一开始就已经创建好的）

第二个参数（edx）：物品词条种子

第三个参数（r8）：AffixText&

返回值（rax）：获取成功：0x1；获取失败：0x0

回到我们的代码中，开头部分就不多说了，依旧老样子，只是需要我们注意的是我要在函数中全代码中利用的寄存器不要选择函数不会保存的易失寄存器，比如r10。在调用完其他的函数之后，如果这个函数中有使用到r10寄存器，它是不会恢复原本的值再返回的，因此我们这里选择r12寄存器。下面，我们将三个参数依次保存在临时变量[rsp+50]、[rsp+58]和[rsp+60]中，便于之后的调用获取。之后，我们申请一段内存并调用构造函数用于创建我们的物品属性结构，将其保存在临时变量[rsp+68]中。之后，我们正确地填写生成词条函数的相应参数并调用它得到词条列表。往下，我们获取词条列表成员数量并根据数量申请对应大小的内存用于保存词条名称、名称文本长度以及词条数值，这里为每个词条申请0x410大小空间的内存。其中在0x0处保存词条的数值，在0x8处保存词条名称文本长度，在0x10开始处往后保存词条名称文本。申请完成后需手动将内存给清零，malloc函数不会自动给我们清空，特别是用于储存文本的内存，这一步显得尤为重要。在上面的所有申请内存中，只要我们发现申请失败，就会分别跳转到GetAffixTextAndValue\_Exit和GetAffixTextAndValue\_Failed，将内存清理完毕之后退出函数，并将返回值设置为0x0表示获取的失败。之后，我们将申请用于保存词条名称及数值的内存地址保存在临时变量[rsp+70]中。之后，我们从创建的结构中获取到列表地址及其成员数量并进入循环，如果成员数量为0，则会跳转到GetAffixTextAndValue\_Failed处，将我们前面申请的所有内存全部释放掉，其中我们创建的结构会调用其对应的析构函数用于释放在调用词条生成函数时在结构中创建的东西。之后，我们在循环中遍历列表中的每个词条，并取出每个词条的本地化语言查找文本用于调用本地化语言查找函数获得其对应的名称文本。在名称文本获取完毕时，我们接着再申请一段内存文本用于对词条名称及数值的文本构建，同时这也是最后写回进参数的内存，用于返回词条名称及数值文本所提供保存空间。我们默认构建的文本最大的长度只有0x400，由于本地化语言查找函数返回的是宽字节，因此我们在这里需要乘2变成0x800个字节要提供，再加上开头需要储存文本长度，所以我们申请0x808个字节用于最后的结果保存。同样地，在申请成功后我们要手动清零。接下来，我们以刚才保存词条名称和数值的内存为依据进入循环。在循环中，我们遍历每个词条名称及其数值，将它们作为参数用于调用宽字节的格式化文本函数，同时使用如图3.F的格式化规则构建出和游戏中一模一样的词条文本显示，其最终结果如图3.F.1所示。之后，在完成这一切之后，我们将保留了词条及数值文本的内存地址写回进我们所保存的参数中，同时将其他剩下的内存给释放掉，同时我们自行构建的结构调用其对应的析构函数用于释放其里面的内容，最后再将返回值设置为0x1表示词条名称文本及数值的获取成功并退出函数完成获取的操作。

别忘了，在完成对代码的构建之后需要我们到反汇编区域查看我们的代码是否被正确地编译。由于这里代码截图过长，就不再进行展示了。

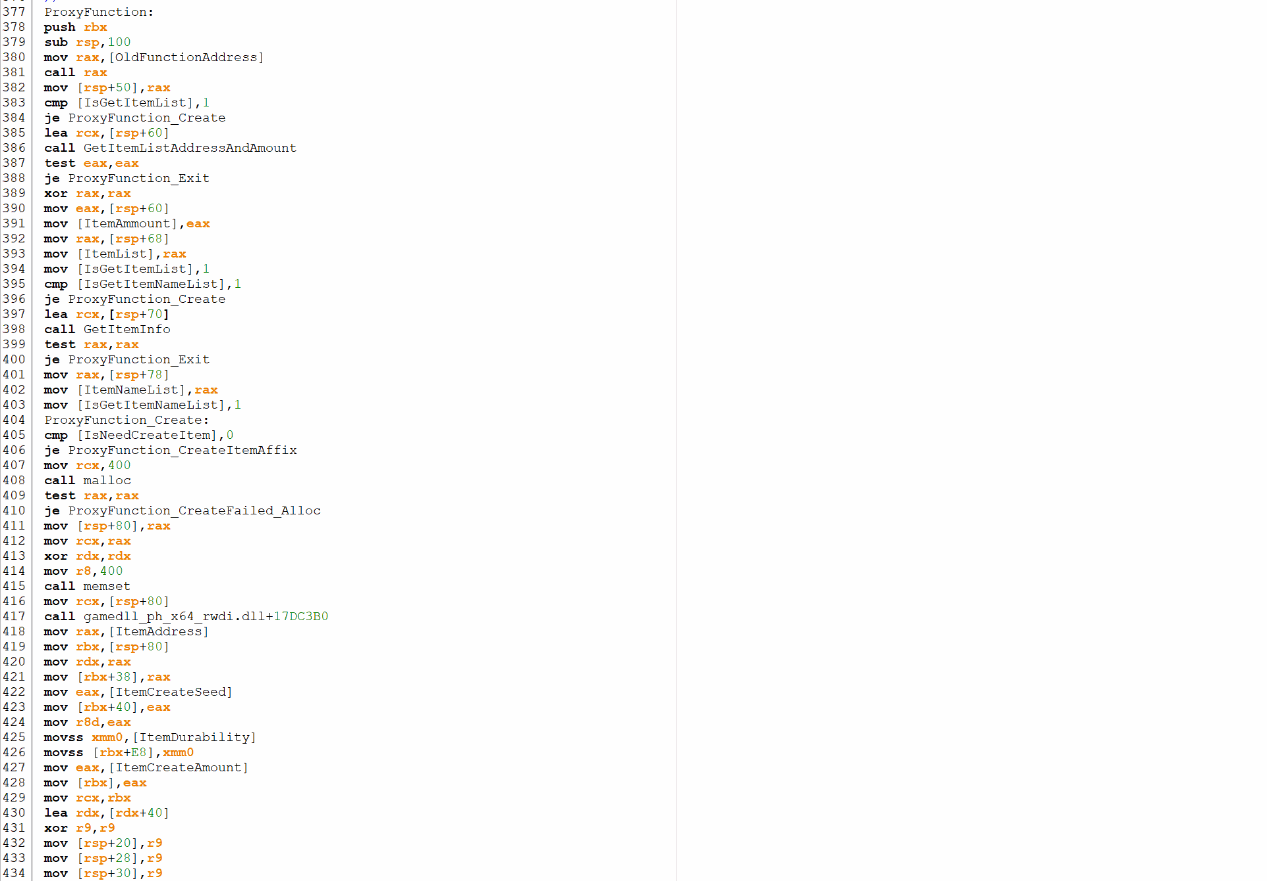


图3.F.2-代理函数

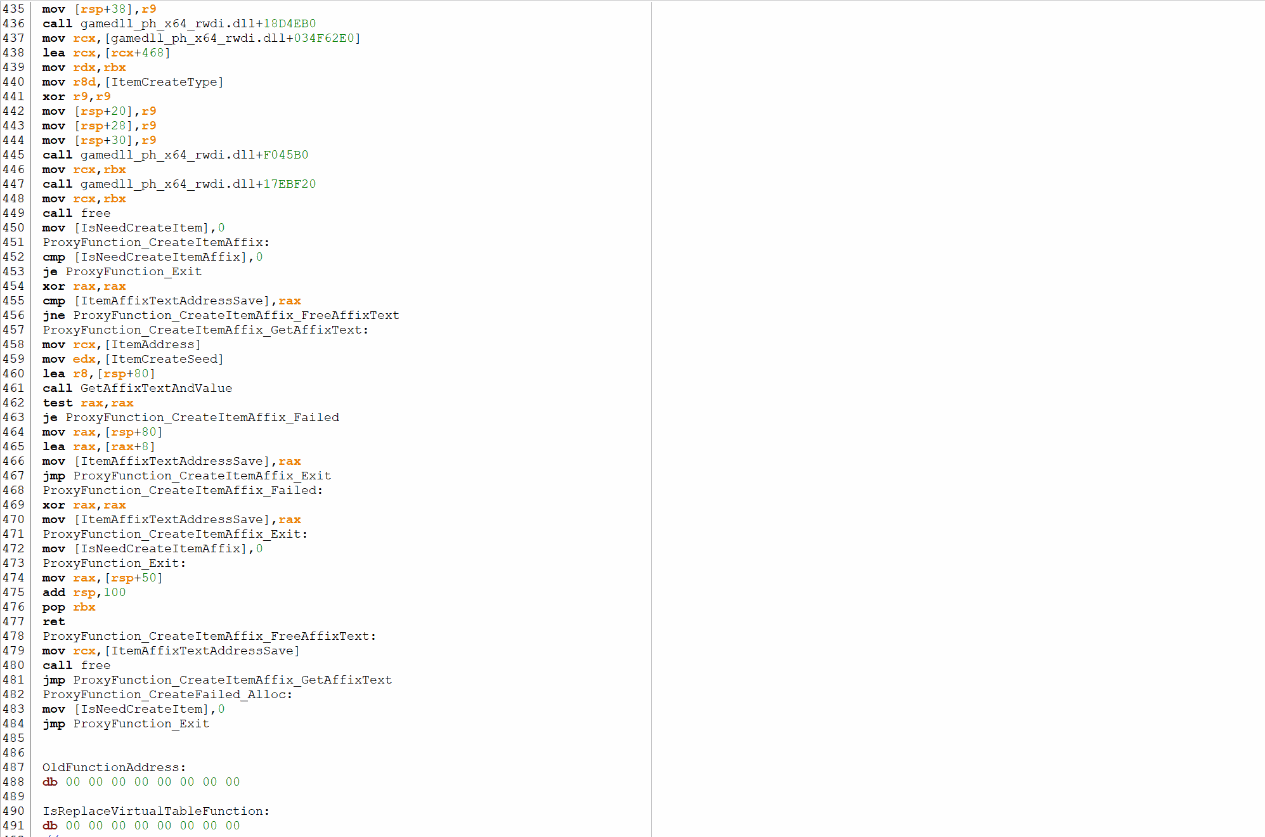


图3.F.3-代理函数（续上图）



图3.F.4-代理函数（续上图）

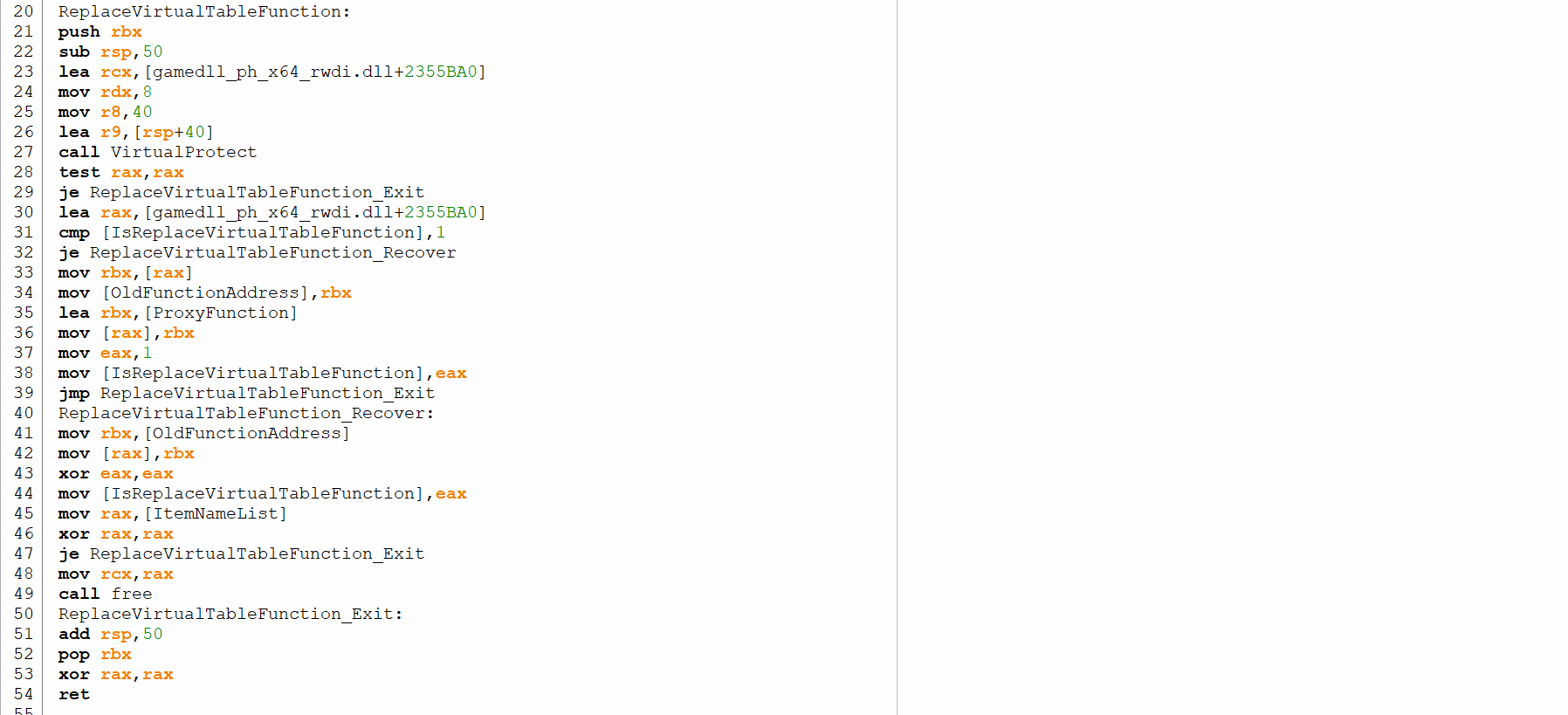


图3.F.5-虚表函数替换代码的更改（行45-行49）

在完成编写各个功能函数之后，我们就要去按照合理的时机去调用它们。在前面，我们编写了一个名为ProxyFunction的函数，用于代替游戏中本来要调用的虚函数，再从这个函数中调用回原本的虚函数进而实现多一个代理层出来，使得执行我们自己的代码成为可能。现在，我们重新回到ProxyFunction的函数中，在里面编写合适的代码使得我们可以在合理的时机实现刷物品脚本的不同功能。

我们的想法大致如下：我们要构建一个可视化的界面，然后我们可以自行选择不同的物品，填写对应物品的不同参数，同时可以预览将会生成的词条，然后可以自行刷取想要的物品。当然，要构建可视化的界面用自带的Lua脚本编写会更好，汇编这方面我们要提供可视化界面构成的基本数据，如物品的名称，词条生成的文本等等。当然了，如果你愿意，也是可以利用汇编脚本编写出可视化界面的，只不过很麻烦罢了，因此接下来我们将会使用Lua脚本构建可视化界面。

如图3.F.2所示，在代理函数ProxyFunction的开头，我们先调用原来的虚函数，然后将函数的返回值保存至临时变量[rsp+50]中用于最后的返回操作。这里你可能会有疑问，开头就直接sub rsp,100把栈抬这么高，会不会影响保存在栈中的参数的取用？这里我们已经分析过了，此虚函数的调用的参数不涉及栈中的参数，只有寄存器中的参数，因此这里不必担心因抬栈过高而导致函数无法读取栈中的参数。同时，此处的返回值保存在寄存器rax中，而不是浮点数寄存器xmm0中。接着，我们判断全局变量IsGetItemList中的值是否为1，如果不是，我们将调用上面我们所编写的函数GetItemListAddressAndAmount获取物品属性列表及其成员数量，并将它们分别保存在临时变量[rsp+60]和临时变量[rsp+68]中便于之后的调用。同时，我们把列表地址和成员数量分别保存在全局变量ItemList和ItemAmmount中，便于之后在Lua脚本中读取它们用于展现可视化数据供我们自行选择。在完成对列表地址及其成员数量的获取之后，我们将全局变量GetItemListAddressAndAmount设置为1，表示我们已经完成了获取的操作，且由于列表地址及其内容是不变的，除非重启游戏，否则我们只会获取一次，再下一次循环中，将会跳开此处的代码到ProxyFunction\_Create处判断我们是否要进行刷取或获取词条预览文本的操作。接下面，我们判断全局变量IsGetItemNameList中的值，如果非1，我们就进行物品本地化语言文本获取的操作。我们调用前边我们所写的函数GetItemInfo来获取储存每个物品的本地化语言文本的内存空间地址，并将其写入到全局变量ItemNameList中，供之后在Lua脚本中显示出来方便我们对着名称文本选择想要的物品。这前期基本数据获取完毕之后，我们会将对应的全局变量IsGetItemList和IsGetItemNameList设置为1表示我们已经完成获取的操作，也就是完成了初始化操作。接下来，我们判断全局变量IsNeedCreateItem中的值，如果是1，表示我们需要进行刷取物品的操作，这里全局变量的值将会由之后的Lua脚本写入。如果要刷取物品，我们首先创建一段空内容用于构建结构储存参数。在调用完构造函数后，我们将所有的参数写入到结构内存中，如全局变量ItemAddress（物品属性结构地址）、ItemCreateSeed（物品词条生成种子）、ItemDurability（物品耐久度）、ItemCreateAmount（物品刷取数量），这些全局变量中的值都会在之后的Lua脚本中写入。参数写入完毕后，我们分别调用词条生成函数和增添物品函数完成物品刷取的操作获得我们想要的物品。完成后，我们把刚才申请的用于构造参数结构的内存进行析构操作和内存释放操作，也就是调用析构函数和free函数。最后，我们将全局变量IsNeedCreateItem的值设置为0，表示刷取操作的完成，再有刷取操作时再将其值设置为1。之后来到我们生成预览词条文本的功能，也就是判断全局变量IsNeedCreateItemAffix处。老样子，如果其中的值为1，则需要我们进行生成预览词条文本的操作。我们首先判断全局变量ItemAffixTextAddressSave中的值是否为空值，也就是是否为空指针，这里储存着含有词条文本的内存空间地址，是给之后的Lua脚本中读取使用的。如果里面存在有效的地址，我们首先的操作就是先将他给释放掉，避免之后我们调用我们缩写的函数GetAffixTextAndValue时会不停地创建新的内存空间从而导致内存空间的浪费。然后，我们写入对应的参数，调用GetAffixTextAndValue函数来获得预览词条文本供Lua脚本读取并展示出来。完成上述的所有操作之后，我们将原本虚函数的返回值写入进寄存器rax中，并将栈恢复返回至调用方完成所有的操作。然后，还有一点就是，这里需要我们处理异常的操作，比如用于参数构造的内存申请失败，我们会跳转至ProxyFunction\_CreateFailed\_Alloc处理完成后退出函数；物品属性列表及其成员数量获取失败时会跳转到ProxyFunction\_Exit处退出函数等等，这些异常的处理在图中的代码中均有体现。最后，如果我们需要关闭脚本了，那必然也要把获取的保存物品本地化语言文本的内存空间地址给释放掉，而关闭脚本又需要将虚函数给替换回去。因此，如图3.F.5所示，我们在恢复虚函数的代码中插入了释放其内存的代码。

总之，整个代理函数的基本逻辑就是：我们先调用原本的虚函数并将其返回值保留用于最后的返回，然后我们进行初始化操作获取各种需要用到的数据。再下面我们就根据需要执行刷取物品的操作以及生成预览词条文本的操作。

还是老样子，代码编写完毕之后，需要到反汇编处查看代码被正确编译的与否。

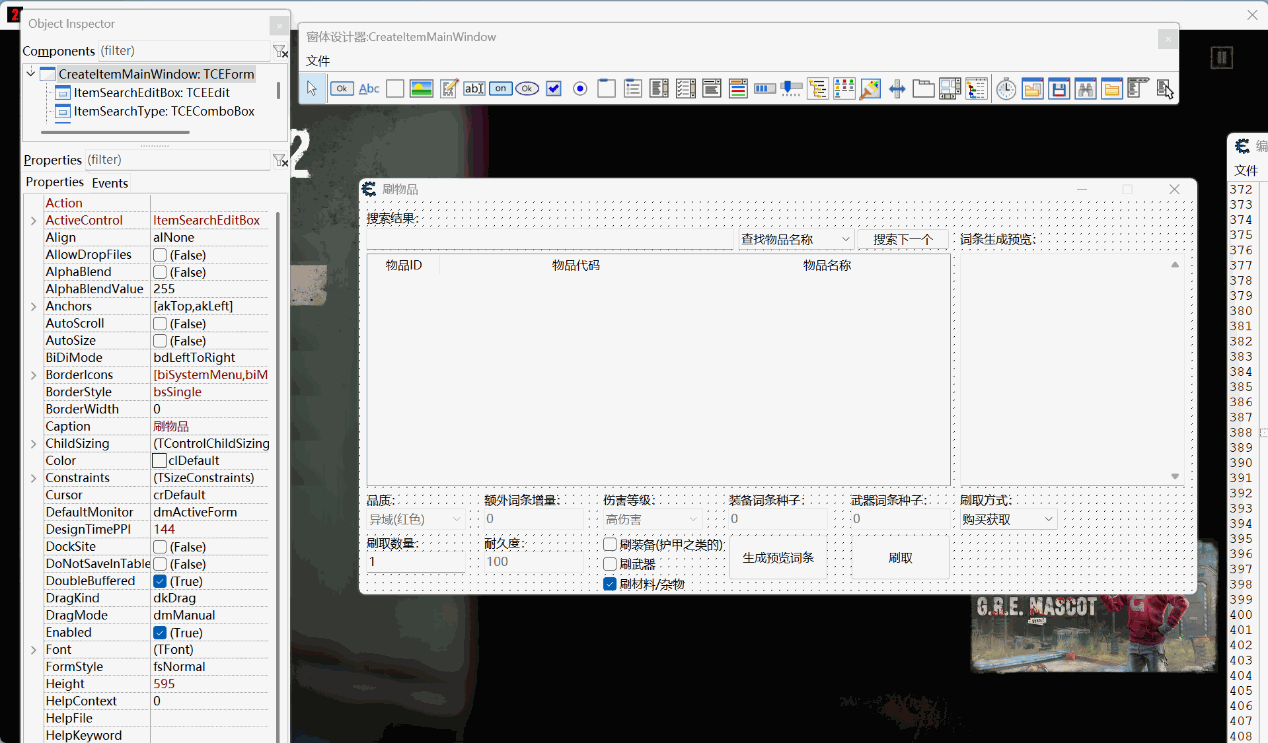


图3.F.6-构建可视化窗口界面

在完成了代理函数中代码的构建之后，我们就根据各个物品参数设计可视化刷取菜单界面。如图3.F.6所示，在上方我们设计了一个编辑框用于输入物品的ID、物品的代码、物品的名称用于在众多物品之中快速查找并定位到相应的位置中。同时在编辑框的右侧我们设置一个拉表用于选择以什么方式进行查找物品的操作，同时又在其右侧设置一个按钮以便我们确认搜索。由于物品众多，加上列表导入的效率不高，因此我们设计的搜索功能应为搜索至当前选中物品位置的最近的下一个物品的位置中，而不是将所用符合的物品统统罗列出来，这样导入效率太慢。同时，我们在其上方设置一个搜索结果提示标签让我们知道我们搜索的物品是否存在。之后，我们在中部的地方设置一个报表框，用于展示每个物品的各个数据，每行一个物品。在表头中我们设置三个栏目，分别是物品ID、物品代码、物品名称，恰好要对应着上方的搜索功能。在报表框下方，我们设置物品刷取的各个参数，方便我们自定义填写每个参数，就如图3.F.6所示，这里不在多说。同时，我们在下方设计两个按钮，它们的功能从左到右分别是生成预览词条和刷取物品。其中，要显示生成预览词条文本我们在左侧中设置一个文本框用于显示其文本内容。这就是窗口设计部分的全部内容，非常地简单。由于组件过多且每个组件都设置了不同的标记名字，这里可自行到脚本中去查找，不再全部将它们罗列出来。

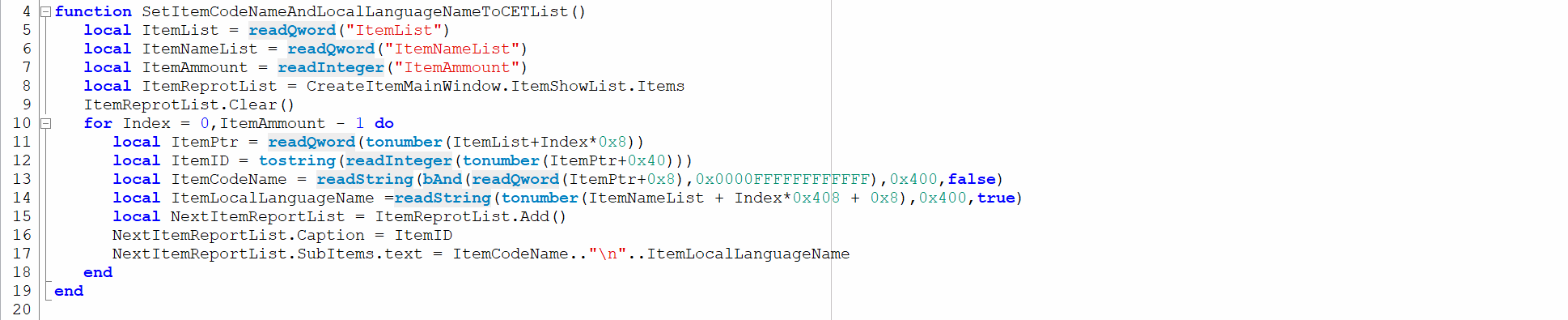


图3.F.7-Lua脚本中用于导入物品数据至报表框中的函数

首先，要实现可视化，我们就要把物品的每个数据导入到报表框中显示它们。如图3.F.7中的Lua函数SetItemCodeNameAndLocalLanguageNameToCETList所示，我们首先获取保存在全局变量中的各种物品数据，然后进入循环把它们每一样都读取出来并导入进列表中。

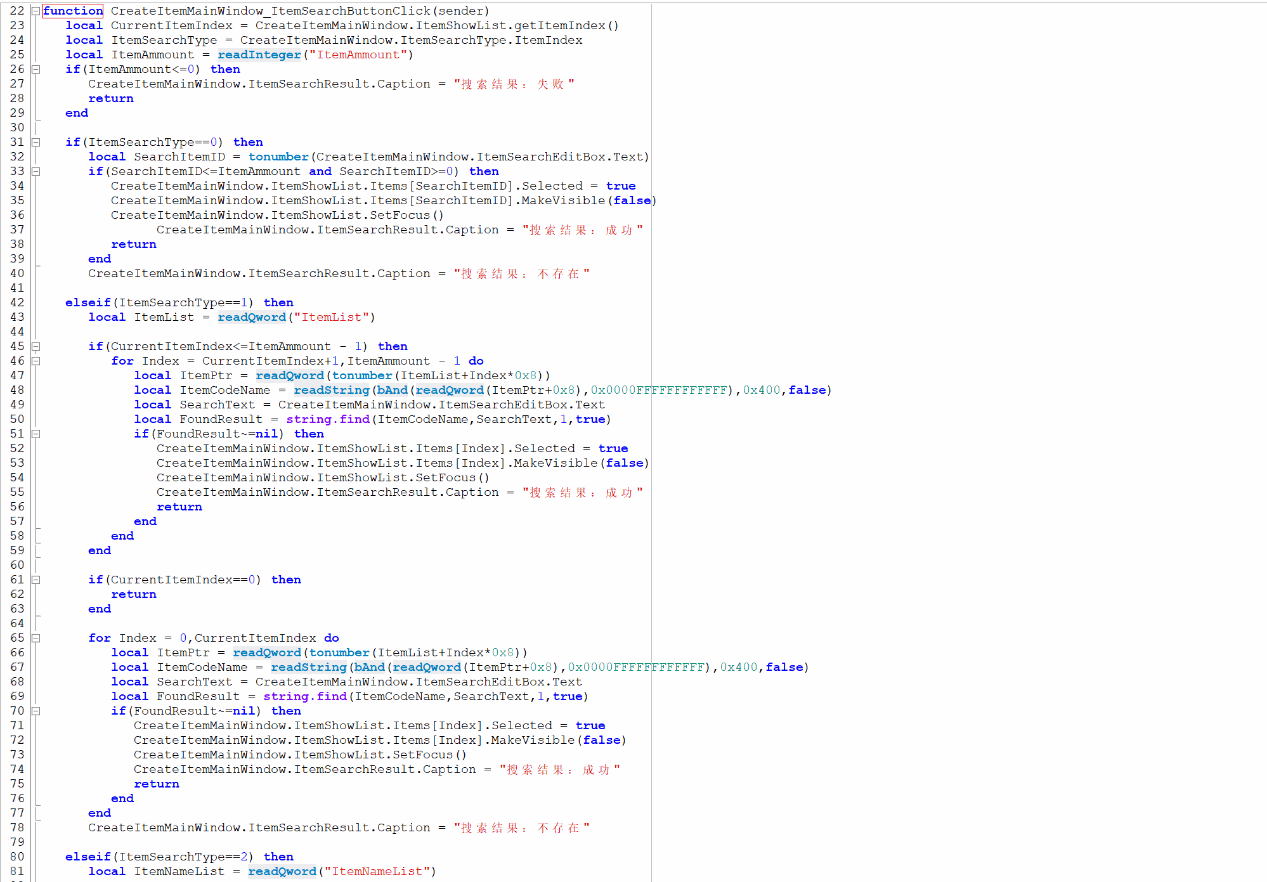


图3.F.8-Lua脚本中用于搜索并定位搜索物品的函数

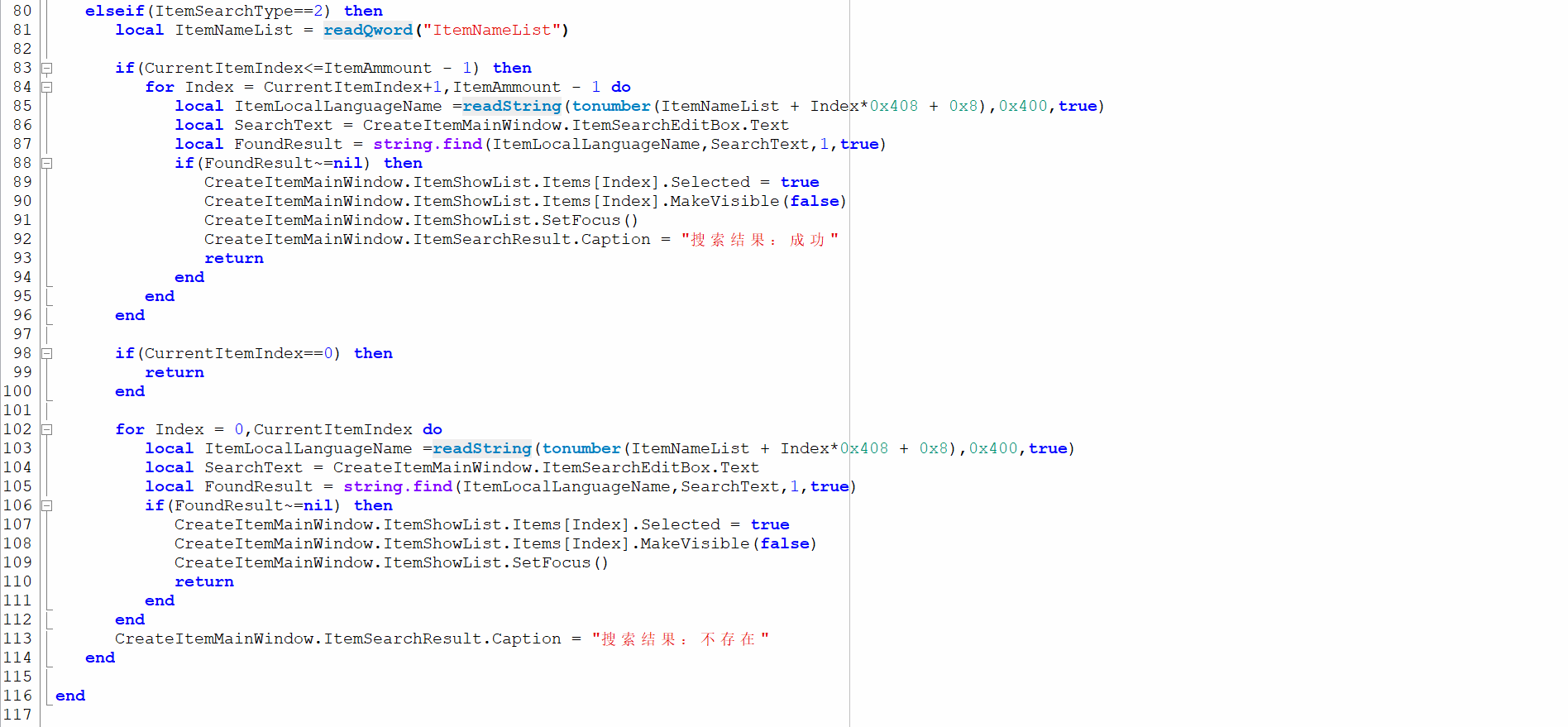


图3.F.9- Lua脚本中用于搜索并定位搜索物品的函数（续上图）

在显示完所有物品的数据之后，接下来我们就要构建搜索的代码让我们快速定位至物品在报表框中的位置。我们按照选择搜索方式的不同分别进行搜索。如果我们选择以物品ID的方式进行搜索，我们就直接按照我们所输入的ID数字在物品列表中定位其位置，因为每个物品ID是在列表中从上往下从小到大的连续的从0开始的整数下来的，因此这里我们直接定位即可。如果选择的是以物品代码搜索的方式，我们就在物品属性列表中逐个读取物品的代码，然后使用string.find()函数查找有无相关的字符，如果有，那么就直接定位到其位置并退出循环返回函数。如果选择的是以物品名称搜索的方式，方式就跟前一种方式基本一致，由于报表中显示的物品数据与我们在汇编脚本中获取的物品名称文本列表中的顺序是一致的，我们就逐个查找，如果有相关文本直接定位即可。最后，在各自搜索方式搜索完毕之后，我们设置搜索结果的文本，对应搜索的成功和失败。

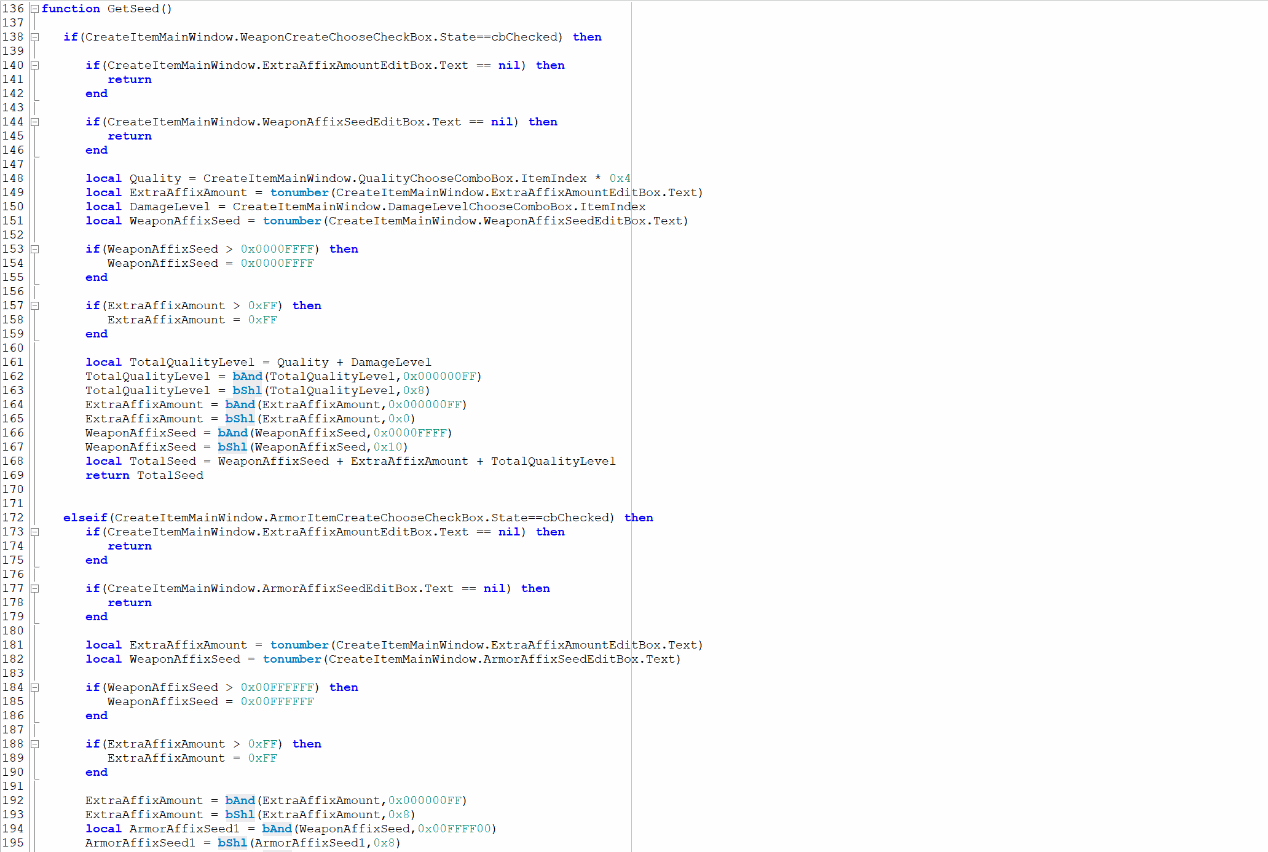


图3.F.A-设置词条种子函数

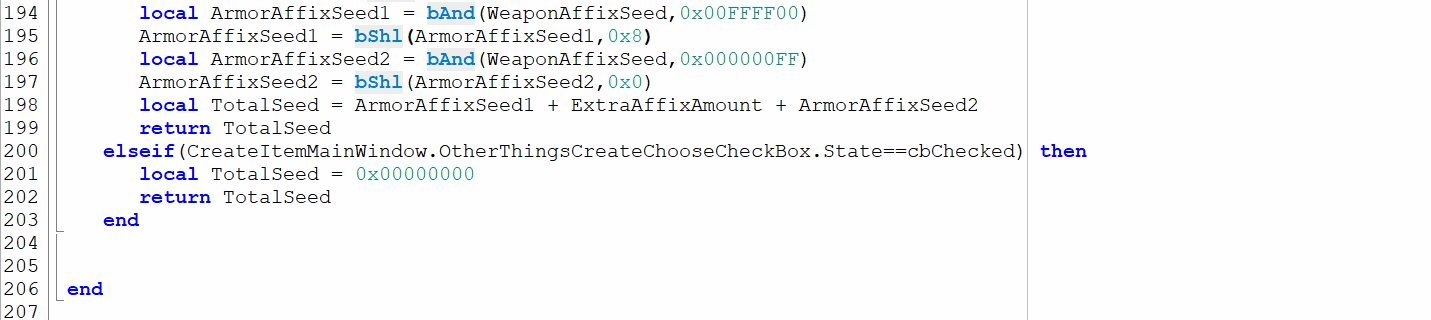


图3.F.B-设置种子函数（续上图）

在完成上述操作的代码构建后，我们就来到物品刷取中的最重要的参数构建：词条种子。如图3.F.A所示，我们根据我们所选刷取物品类型的不同来各自构造相应的种子。如图3.F.6所示，我们设置了3个复选框来选择要刷取物品的类型，因为不同类型的物品对应着不同的种子规律，最明显的就是物品和装备的种子规律。如果选择的是武器类型，我们就按照前边我们所发现的规律进行种子的构建，如图3.F.A中行138-行169所示，我们获取各个参数，将它们构建成一个十六进制的4字节长度的数值然后返回。如果选择的是装备类型，也是同样的操作，按照我们发现的规律进行构建，如图3.F.A和图3.F.B中的行172-行203所示。如果选择的是材料或杂物的类型，我们就直接返回0x00000000，因为此类物品不需要用到种子构建词条。

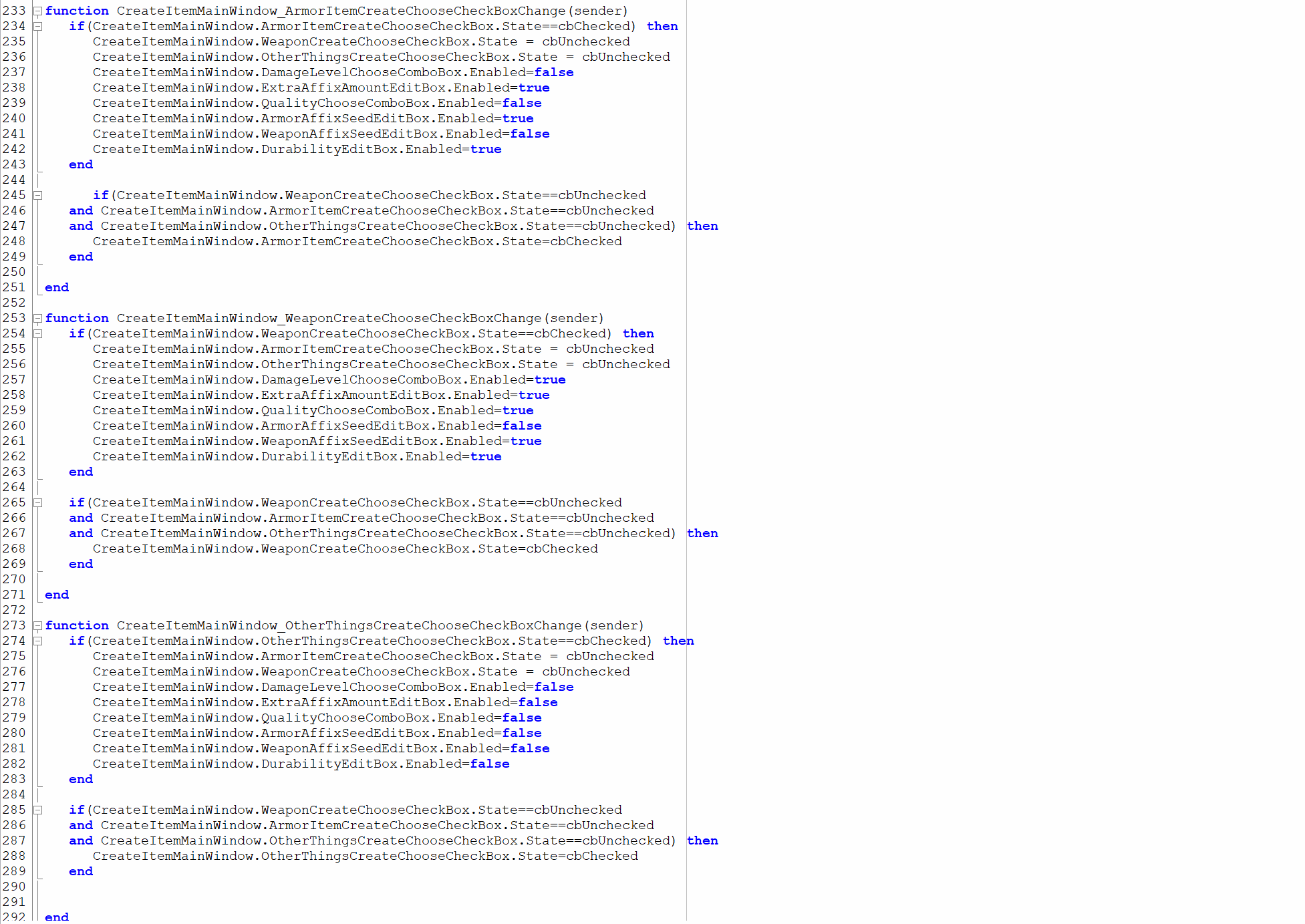


图3.F.C-三个复选框的逻辑

由于每个物品只对应着一种物品类型，因此三个复选框这里需要做出如图3.F.C所示的逻辑进去，只能同时选择一个复选框，同时其他参数设置为对应的禁止与使用状态。



图3.F.D-生成预览词条文本按钮逻辑

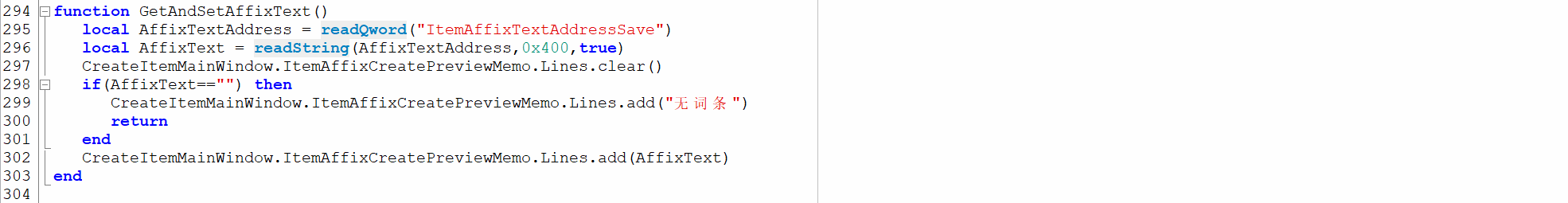


图3.F.E-获取和展示预览词条文本逻辑

接下来，我们要做的就是构建预览词条文本的相关逻辑。如图3.F.D所示，我们在生成按钮的逻辑中调用我们刚构建完毕的构建种子函数，然后从报表框中选择的物品获取其对应的物品属性结构。同时，将它们写入到对应的全局变量中提示汇编代码开始进行词条文本的获取操作。之后，我们暂停Sleep 50毫秒用于给汇编代码处留给充足的时间进行获取。在50毫秒后，我们调用GetAndSetAffixText函数将获取到的词条文本给显示出来。



图3.F.F-确认刷取物品按钮逻辑

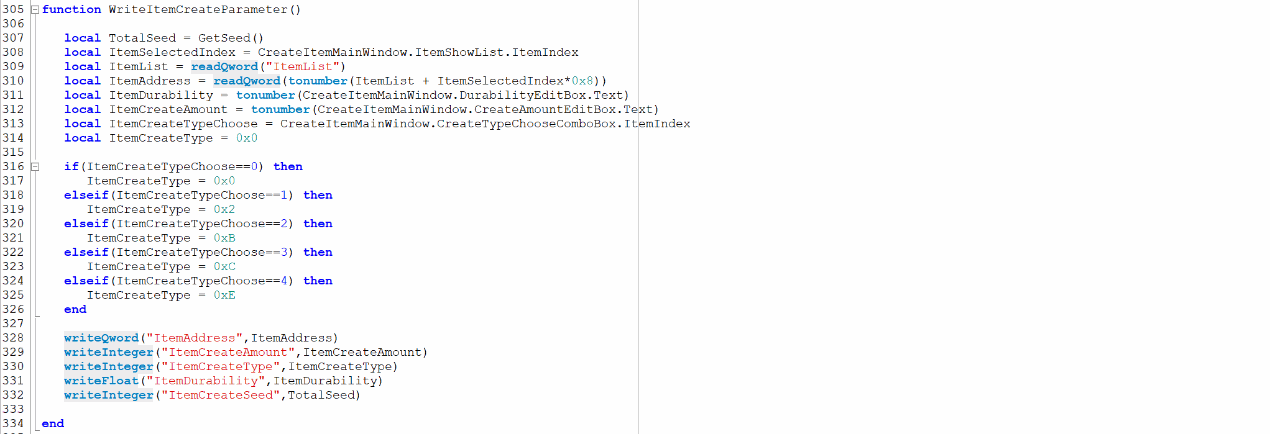


图3.F.10-物品刷取参数写入逻辑

下面，我们要完成另一个按钮，刷取按钮逻辑的构建。如图3.F.10所示，我们构建一个将物品参数写入对应的全局变量的函数，方便我们之后调用。在此函数中，我们读取窗口中的每个组件的参数，将它们全部写入到对应的全局变量中以便给汇编代码的刷取提供正确的参数。接着，如图3.F.F所示，我们来到按下按钮的处理函数中，我们先调用函数WriteItemCreateParameter将参数写入，然后将全局变量IsNeedCreateItem的值设置为1，提示汇编代码处需要进行刷取物品的操作。

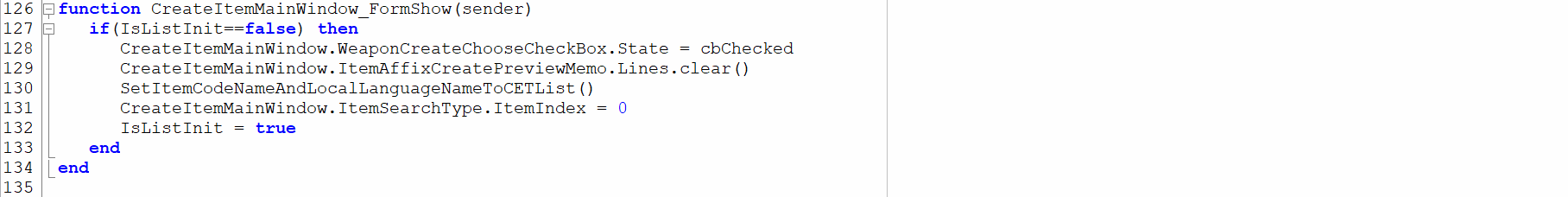


图3.F.11-窗体显示逻辑

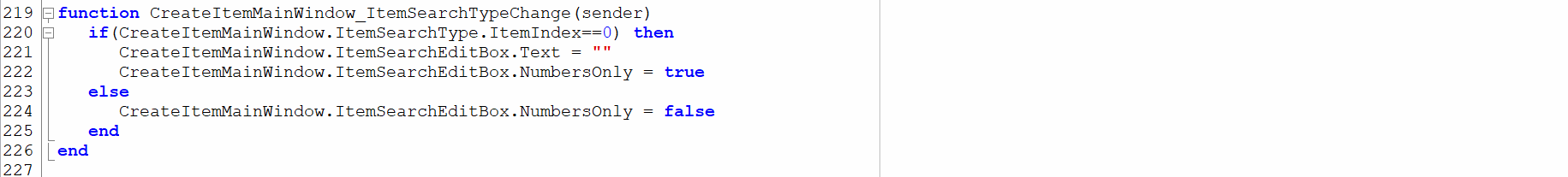


图3.F.12-搜索方式改变逻辑

在上述的所有逻辑构建完成后，我们的Lua脚本的构建也差不多要结束了，我们剩下的就是处理一些小细节。如图3.F.11所示，我们在显示窗体的逻辑中，构建清除词条预览文本框和将物品数据导入到报表框中的操作，同时将报表框中的选择项设置为第一个。同时，我们把Lua脚本中的全局变量IsListInit设置为true，表示初始化操作的完成。如图3.F.12，由于以物品ID搜索时只需要我们输入数字即可。因此，在这里我们将更改搜索方式的逻辑写为当选择以物品ID方式进行搜索时把编辑框的输入方式更改为仅数字输入，如果选择其他两种的方式就将其取消掉，防止用户输入其他字符导致搜索的失败。

到这里，我们的Lua脚本构建就算完成了。接下来我们要做的最后一步就是编写启动汇编脚本和Lua脚本的代码。

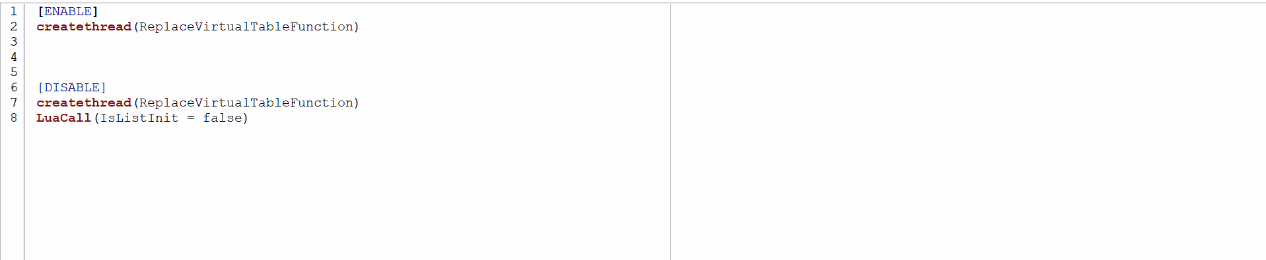


图3.F.13-启动代码

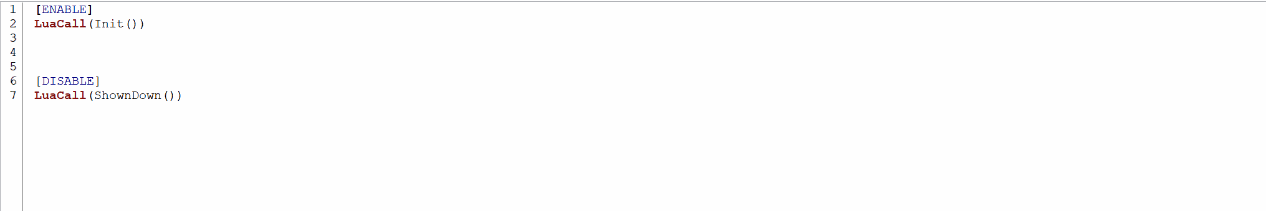


图3.F.14-显示物品刷取菜单代码



图3.F.15-Lua脚本中的启动代码

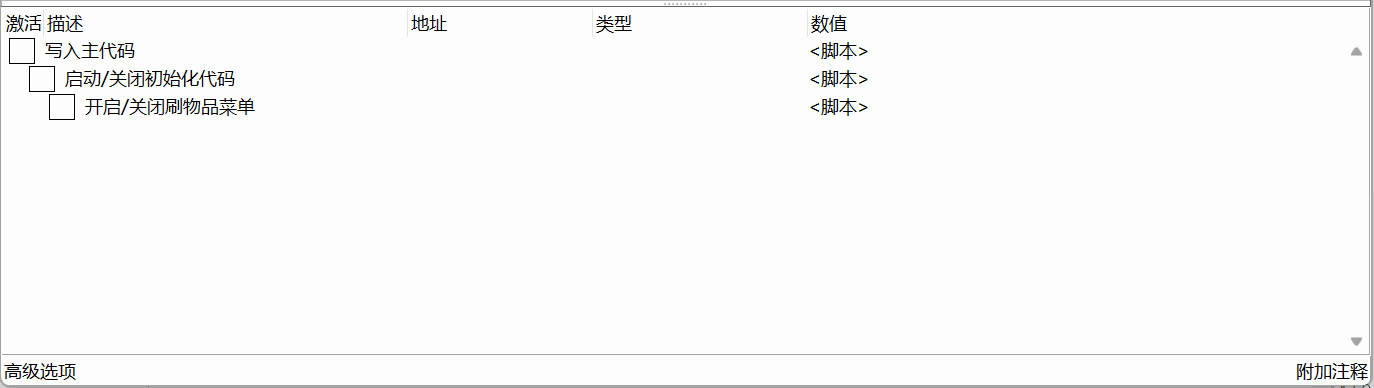


图3.F.16-脚本的排列

如图3.F.13所示，我们新开一个脚本，在[ENABLE]启动标签中对我们的虚表函数替换函数ReplaceVirtualTableFunction进行创建远程线程执行指令的操作使得相应的虚函数替换为我们自己的函数以便游戏程序会执行我们自己的相关的代码。而在关闭标签[DISABLE]处我们再次创建远程线程执行ReplaceVirtualTableFunction函数中的指令将虚函数替换回去同时释放相应的内存空间从而关闭脚本。在执行完启动代码的指令后，如图3.F.14所示，我们使用LuaCall指令执行Lua指令将物品刷取菜单给呼唤出来，要关闭时就执行另一个Lua指令将其关闭。它们对应的Lua脚本中的函数如图3.F.15所示。最后，我们按照它们的启动顺序，一个一个将它们排列为其的字脚本一个一个激活，如图3.F.16所示。要开启脚本时，从上往下，依次激活；要关闭脚本时，从下往上，依次取消激活。

到这里，我们的刷物品脚本就全部构建完毕了，下面就可以自行测试脚本的功能，查看有无异常。



图3.F.17-脚本功能测试

如图3.F.17所示，我们所构建的脚本的功能一切正常，能完美完成我们的刷取任务。

**最后，我们历经千辛万苦，终于完整地将物品刷取脚本构建完毕，其中的功能能满足我们对于物品刷取的需求，它大大降低了我们获取物品的难度和成本，做到了无代价即可获得任何我们想要获得的物品，大大增强了游戏的乐趣。物品刷取脚本的教程在这里就正式结束了。**

**以上的制作方法具有一般通用的性质，在九月份《消逝的光芒：困兽》即将发售，以上提到的方法或许还能在其中运用，能制作出一个新的刷物品脚本出来。**

**脚本下载地址（教程版）：https://wwmu.lanzoub.com/incBy35b27xc**

**脚本下载地址（应用版）：https://wwmu.lanzoub.com/id2cB35b28gb**

**感谢各位能够学习到这里，以上的内容仅是本人的一些拙见，有任何问题可向B站发私信提醒，有空闲时间将会回复：**

**UID：452370879**

**名称：--南海-**