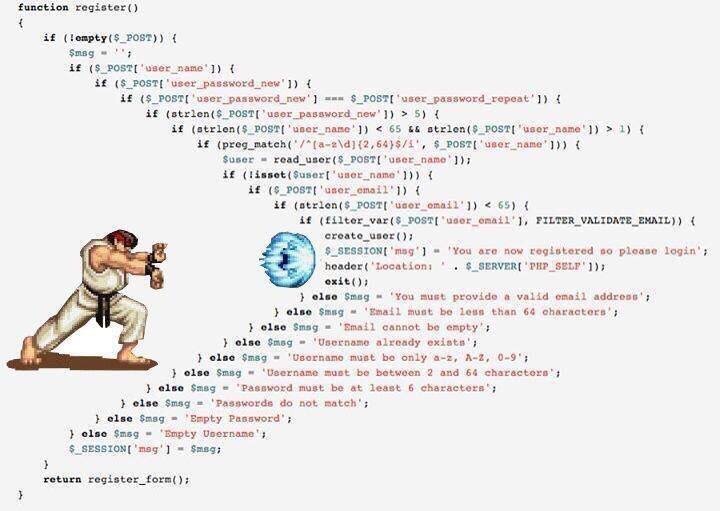
**如何重构“箭头型”代码**

[**2017年04月05日**](http://coolshell.cn/articles/17757.html)[**陈皓**](http://coolshell.cn/articles/author/haoel)**评论**[**28 条评论**](http://coolshell.cn/articles/17757.html#comments)**11,734 人阅读**

本文主要起因是，一次在微博上和朋友关于嵌套好几层的if-else语句的代码重构的讨论（[微博原文](http://weibo.com/1401880315/ECmCW0oy2)），在微博上大家有各式各样的问题和想法。按道理来说这些都是编程的基本功，似乎不太值得写一篇文章，不过我觉得很多东西可以从一个简单的东西出发，到达本质，所以，我觉得有必要在这里写一篇的文章。不一定全对，只希望得到更多的讨论，因为有了更深入的讨论才能进步。

文章有点长，我在文章最后会给出相关的思考和总结陈词，你可以跳到结尾。

所谓箭头型代码，基本上来说就是下面这个图片所示的情况。



那么，这样“箭头型”的代码有什么问题呢？看上去也挺好看的，有对称美。但是……

关于箭头型代码的问题有如下几个：

1）我的显示器不够宽，箭头型代码缩进太狠了，需要我来回拉水平滚动条，这让我在读代码的时候，相当的不舒服。

2）除了宽度外还有长度，有的代码的if-else里的if-else里的if-else的代码太多，读到中间你都不知道中间的代码是经过了什么样的层层检查才来到这里的。

总而言之，**“箭头型代码”如果嵌套太多，代码太长的话，会相当容易让维护代码的人（包括自己）迷失在代码中，因为看到最内层的代码时，你已经不知道前面的那一层一层的条件判断是什么样的，代码是怎么运行到这里的，所以，箭头型代码是非常难以维护和Debug的**。

**微博上的案例 与 Guard Clauses**

OK，我们先来看一下微博上的那个示例，代码量如果再大一点，嵌套再多一点，你很容易会在条件中迷失掉（下面这个示例只是那个“大箭头”下的一个小箭头）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | FOREACH(Ptr<WfExpression>, argument, node->arguments) {      int index = manager->expressionResolvings.Keys().IndexOf(argument.Obj());      if (index != -1) {          auto type = manager->expressionResolvings.Values()[index].type;          if (! types.Contains(type.Obj())) {              types.Add(type.Obj());              if (auto group = type->GetTypeDescriptor()->GetMethodGroupByName(L"CastResult", true)) {                  int count = group->GetMethodCount();                  for (int i = 0; i < count; i++) { auto method = group->GetMethod(i);                      if (method->IsStatic()) {                          if (method->GetParameterCount() == 1 &&                              method->GetParameter(0)->GetType()->GetTypeDescriptor() == description::GetTypeDescriptor<DescriptableObject>() &&                              method->GetReturn()->GetTypeDescriptor() != description::GetTypeDescriptor<void>() ) {                              symbol->typeInfo = CopyTypeInfo(method->GetReturn());                              break;                          }                      }                  }              }          }      }  } |

上面这段代码，可以把条件反过来写，然后就可以把箭头型的代码解掉了，重构的代码如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | FOREACH(Ptr<WfExpression>, argument, node->arguments) {      int index = manager->expressionResolvings.Keys().IndexOf(argument.Obj());      if (index == -1)  continue;        auto type = manager->expressionResolvings.Values()[index].type;      if ( types.Contains(type.Obj()))  continue;        types.Add(type.Obj());        auto group = type->GetTypeDescriptor()->GetMethodGroupByName(L"CastResult", true);      if  ( ! group ) continue;        int count = group->GetMethodCount();      for (int i = 0; i < count; i++) { auto method = group->GetMethod(i);          if (! method->IsStatic()) continue;            if ( method->GetParameterCount() == 1 &&                 method->GetParameter(0)->GetType()->GetTypeDescriptor() == description::GetTypeDescriptor<DescriptableObject>() &&                 method->GetReturn()->GetTypeDescriptor() != description::GetTypeDescriptor<void>() ) {              symbol->typeInfo = CopyTypeInfo(method->GetReturn());              break;          }      }  } |

这种代码的重构方式叫 **Guard Clauses**

* [Martin Fowler](https://martinfowler.com/) 的 Refactoring 的网站上有相应的说明《[Replace Nested Conditional with Guard Clauses](https://refactoring.com/catalog/replaceNestedConditionalWithGuardClauses.html)》。
* [Coding Horror](https://blog.codinghorror.com/) 上也有一篇文章讲了这种重构的方式 —— 《[Flattening Arrow Code](https://blog.codinghorror.com/flattening-arrow-code/)》
* [StackOverflow](http://stackoverflow.com/) 上也有相关的问题说了这种方式 —— 《[Refactor nested IF statement for clarity](http://stackoverflow.com/questions/356121/refactor-nested-if-statement-for-clarity)》

这里的思路其实就是，**让出错的代码先返回，前面把所有的错误判断全判断掉，然后就剩下的就是正常的代码了**。

**抽取成函数**

微博上有些人说，continue 语句破坏了阅读代码的通畅，我觉得他们一定没有好好读这里面的代码，其实，我们可以看到，所有的 if 语句都是在判断是否出错的情况，所以，在维护代码的时候，你可以完全不理会这些 if 语句，因为都是出错处理的，而剩下的代码都是正常的功能代码，反而更容易阅读了。当然，一定有不是上面代码里的这种情况，那么，不用continue ，我们还能不能重构呢？

当然可以，抽成函数：

|  |  |
| --- | --- |
| 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  31  32  33  34  35  36  37  38  39 | bool CopyMethodTypeInfo(auto &method, auto &group, auto &symbol)  {      if (! method->IsStatic()) {          return true;      }      if ( method->GetParameterCount() == 1 &&             method->GetParameter(0)->GetType()->GetTypeDescriptor() == description::GetTypeDescriptor<DescriptableObject>() &&             method->GetReturn()->GetTypeDescriptor() != description::GetTypeDescriptor<void>() ) {          symbol->typeInfo = CopyTypeInfo(method->GetReturn());          return false;      }      return true;  }    void ExpressionResolvings(auto &manager, auto &argument, auto &symbol)  {      int index = manager->expressionResolvings.Keys().IndexOf(argument.Obj());      if (index == -1) return;        auto type = manager->expressionResolvings.Values()[index].type;      if ( types.Contains(type.Obj())) return;        types.Add(type.Obj());      auto group = type->GetTypeDescriptor()->GetMethodGroupByName(L"CastResult", true);      if  ( ! group ) return;        int count = group->GetMethodCount();      for (int i = 0; i < count; i++) { auto method = group->GetMethod(i);          if ( ! CopyMethodTypeInfo(method, group, symbol) ) break;      }  }    ...  ...  FOREACH(Ptr<WfExpression>, argument, node->arguments) {      ExpressionResolvings(manager, arguments, symbol)  }  ...  ... |

你发出现，抽成函数后，代码比之前变得更容易读和更容易维护了。不是吗？

有人说：“如果代码不共享，就不要抽取成函数！”，持有这个观点的人太死读书了。函数是代码的封装或是抽象，并不一定用来作代码共享使用，函数用于屏蔽细节，让其它代码耦合于接口而不是细节实现，这会让我们的代码更为简单，简单的东西都能让人易读也易维护。这才是函数的作用。

**嵌套的 if 外的代码**

微博上还有人问，原来的代码如果在各个 if 语句后还有要执行的代码，那么应该如何重构。比如下面这样的代码。

|  |  |
| --- | --- |
| 原版 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | for(....) {      do\_before\_cond1()      if (cond1) {          do\_before\_cond2();          if (cond2) {              do\_before\_cond3();              if (cond3) {                  do\_something();              }              do\_after\_cond3();          }          do\_after\_cond2();      }      do\_after\_cond1();  } |

上面这段代码中的那些 do\_after\_condX() 是无论条件成功与否都要执行的。所以，我们拉平后的代码如下所示：

|  |  |
| --- | --- |
| 重构第一版 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | for(....) {      do\_before\_cond1();      if ( !cond1 ) {          do\_after\_cond1();          continue      }      do\_after\_cond1();        do\_before\_cond2();      if ( !cond2 ) {          do\_after\_cond2();          continue;      }      do\_after\_cond2();        do\_before\_cond3();      if ( !cond3 ) {          do\_after\_cond3();          continue;      }      do\_after\_cond3();        do\_something();  } |

你会发现，上面的 do\_after\_condX 出现了两份。**如果 if 语句块中的代码改变了某些do\_after\_condX依赖的状态，那么这是最终版本。**

但是，如果它们之前没有依赖关系的话，根据 DRY 原则，我们就可以只保留一份，那么直接掉到 if 条件前就好了，如下所示：

|  |  |
| --- | --- |
| 重构第二版 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | for(....) {      do\_before\_cond1();      do\_after\_cond1();      if ( !cond1 ) continue;        do\_before\_cond2();      do\_after\_cond2();      if ( !cond2 ) continue;        do\_before\_cond3();      do\_after\_cond3();      if ( !cond3 ) continue;        do\_something();  } |

此时，你会说，我靠，居然，改变了执行的顺序，把条件放到 do\_after\_condX() 后面去了。这会不会有问题啊？

其实，你再分析一下之前的代码，你会发现，本来，cond1 是判断 do\_before\_cond1() 是否出错的，如果有成功了，才会往下执行。而 do\_after\_cond1() 是无论如何都要执行的。从逻辑上来说，do\_after\_cond1()其实和do\_before\_cond1()的执行结果无关，而 cond1 却和是否去执行 do\_before\_cond2() 相关了。如果我把断行变成下面这样，反而代码逻辑更清楚了。

|  |  |
| --- | --- |
| 重构第三版 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | for(....) {        do\_before\_cond1();      do\_after\_cond1();          if ( !cond1 ) continue;  // <-- cond1 成了是否做第二个语句块的条件      do\_before\_cond2();      do\_after\_cond2();        if ( !cond2 ) continue; // <-- cond2 成了是否做第三个语句块的条件      do\_before\_cond3();      do\_after\_cond3();        if ( !cond3 ) continue; //<-- cond3 成了是否做第四个语句块的条件      do\_something();    } |

于是乎，在未来维护代码的时候，维护人一眼看上去就明白，代码在什么时候会执行到哪里。 这个时候，你会发现，把这些语句块抽成函数，代码会干净的更多，再重构一版：

|  |  |
| --- | --- |
| 重构第四版 | |
| 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  31  32  33 | bool do\_func3() {     do\_before\_cond2();     do\_after\_cond2();     return cond3;  }    bool do\_func2() {     do\_before\_cond2();     do\_after\_cond2();     return cond2;  }    bool do\_func1() {     do\_before\_cond1();     do\_after\_cond1();     return cond1;  }    // for-loop 你可以重构成这样  for (...) {      bool cond = do\_func1();      if (cond) cond = do\_func2();      if (cond) cond = do\_func3();      if (cond) do\_something();  }    // for-loop 也可以重构成这样  for (...) {      if ( ! do\_func1() ) continue;      if ( ! do\_func2() ) continue;      if ( ! do\_func3() ) continue;      do\_something();  } |

上面，我给出了两个版本的for-loop，你喜欢哪个？我喜欢第二个。这个时候，因为for-loop里的代码非常简单，就算你不喜欢 continue ，这样的代码阅读成本已经很低了。

**状态检查嵌套**

接下来，我们再来看另一个示例。下面的代码的伪造了一个场景——把两个人拉到一个一对一的聊天室中，因为要检查双方的状态，所以，代码可能会写成了“箭头型”。

|  |  |
| --- | --- |
| 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  31  32 | int ConnectPeer2Peer(Conn \*pA, Conn\* pB, Manager \*manager)  {      if ( pA->isConnected() ) {          manager->Prepare(pA);          if ( pB->isConnected() ) {              manager->Prepare(pB);              if ( manager->ConnectTogther(pA, pB) ) {                  pA->Write("connected");                  pB->Write("connected");                  return S\_OK;              }else{                  return S\_ERROR;              }            }else {              pA->Write("Peer is not Ready, waiting...");              return S\_RETRY;          }      }else{          if ( pB->isConnected() ) {              manager->Prepare();              pB->Write("Peer is not Ready, waiting...");              return S\_RETRY;          }else{              pA->Close();              pB->Close();              return S\_ERROR;          }      }      //Shouldn't be here!      return S\_ERROR;  } |

重构上面的代码，我们可以先分析一下上面的代码，说明了，上面的代码就是对 PeerA 和 PeerB 的两个状态 “连上”， “未连上” 做组合 “状态” （注：实际中的状态应该比这个还要复杂，可能还会有“断开”、“错误”……等等状态）， 于是，我们可以把代码写成下面这样，合并上面的嵌套条件，对于每一种组合都做出判断。这样一来，逻辑就会非常的干净和清楚。

|  |  |
| --- | --- |
| 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  31  32  33 | int ConnectPeer2Peer(Conn \*pA, Conn\* pB, Manager \*manager)  {      if ( pA->isConnected() ) {          manager->Prepare(pA);      }        if ( pB->isConnected() ) {          manager->Prepare(pB);      }        // pA = YES && pB = NO      if (pA->isConnected() && ! pB->isConnected()  ) {          pA->Write("Peer is not Ready, waiting");          return S\_RETRY;      // pA = NO && pB = YES      }else if ( !pA->isConnected() && pB->isConnected() ) {          pB->Write("Peer is not Ready, waiting");          return S\_RETRY;      // pA = YES && pB = YES      }else if (pA->isConnected() && pB->isConnected()  ) {          if ( ! manager->ConnectTogther(pA, pB) ) {              return S\_ERROR;          }          pA->Write("connected");          pB->Write("connected");          return S\_OK;      }        // pA = NO, pB = NO      pA->Close();      pB->Close();      return S\_ERROR;  } |

**延伸思考**

对于 if-else 语句来说，一般来说，就是检查两件事：**错误** 和 **状态**。

**检查错误**

对于检查错误来说，使用 Guard Clauses 会是一种标准解，但我们还需要注意下面几件事：

1）当然，出现错误的时候，还会出现需要释放资源的情况。你可以使用 goto fail;这样的方式，但是最优雅的方式应该是C++面向对象式的 RAII 方式。

2）以错误码返回是一种比较简单的方式，这种方式有很一些问题，比如，如果错误码太多，判断出错的代码会非常复杂，另外，正常的代码和错误的代码会混在一起，影响可读性。所以，在更为高组的语言中，使用 try-catch 异常捕捉的方式，会让代码更为易读一些。

**检查状态**

对于检查状态来说，实际中一定有更为复杂的情况，比如下面几种情况：

1）像TCP协议中的两端的状态变化。

2）像shell各个命令的命令选项的各种组合。

3）像游戏中的状态变化（一棵非常复杂的状态树）。

4）像语法分析那样的状态变化。

对于这些复杂的状态变化，其本上来说，你需要先定义一个状态机，或是一个子状态的组合状态的查询表，或是一个状态查询分析树。

**写代码时，代码的运行中的控制状态或业务状态是会让你的代码流程变得混乱的一个重要原因，重构“箭头型”代码的一个很重要的工作就是重新梳理和描述这些状态的变迁关系**。

**总结**

好了，下面总结一下，把“箭头型”代码重构掉的几个手段如下：

1）**使用 Guard Clauses**。 尽可能的让出错的先返回， 这样后面就会得到干净的代码。

2）**把条件中的语句块抽取成函数**。 有人说：“如果代码不共享，就不要抽取成函数！”，持有这个观点的人太死读书了。函数是代码的封装或是抽象，并不一定用来作代码共享使用，函数用于屏蔽细节，让其它代码耦合于接口而不是细节实现，这会让我们的代码更为简单，简单的东西都能让人易读也易维护，**写出让人易读易维护的代码才是重构代码的初衷**！

3）**对于出错处理，使用try-catch异常处理和**[**RAII机制**](http://stackoverflow.com/questions/2321511/what-is-meant-by-resource-acquisition-is-initialization-raii)。返回码的出错处理有很多问题，比如：A) 返回码可以被忽略，B) 出错处理的代码和正常处理的代码混在一起，C) 造成函数接口污染，比如像atoi()这种错误码和返回值共用的糟糕的函数。

4）**对于多个状态的判断和组合，如果复杂了，可以使用“组合状态表”，或是状态机加Observer的状态订阅的设计模式**。这样的代码即解了耦，也干净简单，同样有很强的扩展性。

5） **重构“箭头型”代码其实是在帮你重新梳理所有的代码和逻辑，这个过程非常值得为之付出**。重新整思路去想尽一切办法简化代码的过程本身就可以让人成长。

（全文完）