# 完美国际/完美世界客户端程序编码规范

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 时间 | 参加者（按拼音） | 内容 |
| 1.0 | 2014-4-3 | 韩光辉 王永东 徐文彬 | 初始版本 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**一、通用规范**

1. **命名的通用规则**
2. 除了一眼能看出含义的单词，尽量不用缩写，如src、dst等缩写接受度较高，可以采用，但newRotateAngle不要使用nra、newRotAng等缩写，直接写更具可读性，名称稍长些没有关系。需要了解的是，代码的阅读次数，远远比代码的书写次数多。
3. 尽量使用口语化的单词和口语化的顺序，m\_nMaterialSlot比m\_nSoltMaterail读起来更顺口、也更易理解。
4. 尽量描述具体用途，避免如m\_canDo这样的名称而使用m\_canFinishAnimation
5. **类、结构、枚举、常量命名**

|  |  |  |
| --- | --- | --- |
| 类别 | 命名方法 | 举例 |
| 类名、结构名 | 对**主要类**使用字母’C’+前缀+大小写混合；  非主要类（如cpp中或函数中定义），不需加前缀，使用大小写混合即可 | CDlgWorldMap  CECGameRun |
| 枚举名 | enum+大小写混合  或者不加enum前缀、直接大小写混合 | enumRandMallShoppingState |
| 枚举值 | 应全部大写，单词间以下划线相连，最好采用前缀，以避免定义重复 | RMSS\_NO\_CONFIG（其中，RMSS前缀是枚举名enumRandMallShoppingState里的4个大写字母 |
| 常量变量 | 全部大写，单词间以下划线相连 | NUM\_PROFESSION |

1. **类成员变量、类方法、局部变量、静态变量、全局变量等命名**

|  |  |  |
| --- | --- | --- |
| 类别 | 命名方法 | 举例 |
| 类成员变量 | m\_开头，首字符小写，后面大小写混合； | m\_pvpStatusReady |
| 结构成员变量 | 复杂结构与类成员变量命名一样  （应尽量使用class，避免将成员公开）  简单结构不需要m\_前缀，即只要首字符小写，后面大小写混合 | 复杂结构变量：m\_minePos  简单结构变量：minePos |
| 类方法 | 大小写混合 | RegisterObserver |
| 局部变量 | 首字符小写，后面大小写混合 | newRotateAngle |
| 静态变量 | s\_开头，首字符小写，后面大小写混合 | s\_bFirstTime |
| 全局变量 | g\_开头，首字符小写，后面大小写混合 | g\_pGame |

1. **头文件**

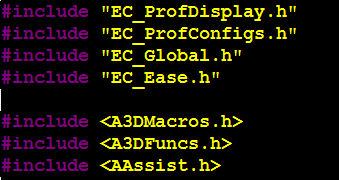
禁止使用#pragma once，使用#ifndef、#define、#endif，以同服务器、ISO等平台保持一致、并方便重用。#ifndef使用的宏命名规则为“\_目录名+文件名\_H\_”，如CElementClient目录下的DlgAction.h，使用 \_CELEMENTCLIENT\_DLGACTION\_H\_。

头文件中引用的类型，需要在头文件中前置声明或#include，即头文件对自己引入的类型要具有自解释的功能。如头文件中引入了A3DVECTOR，则由头文件前置声明或包含对应文件。

头文件应多使用前置声明，减少对其它文件的包含。

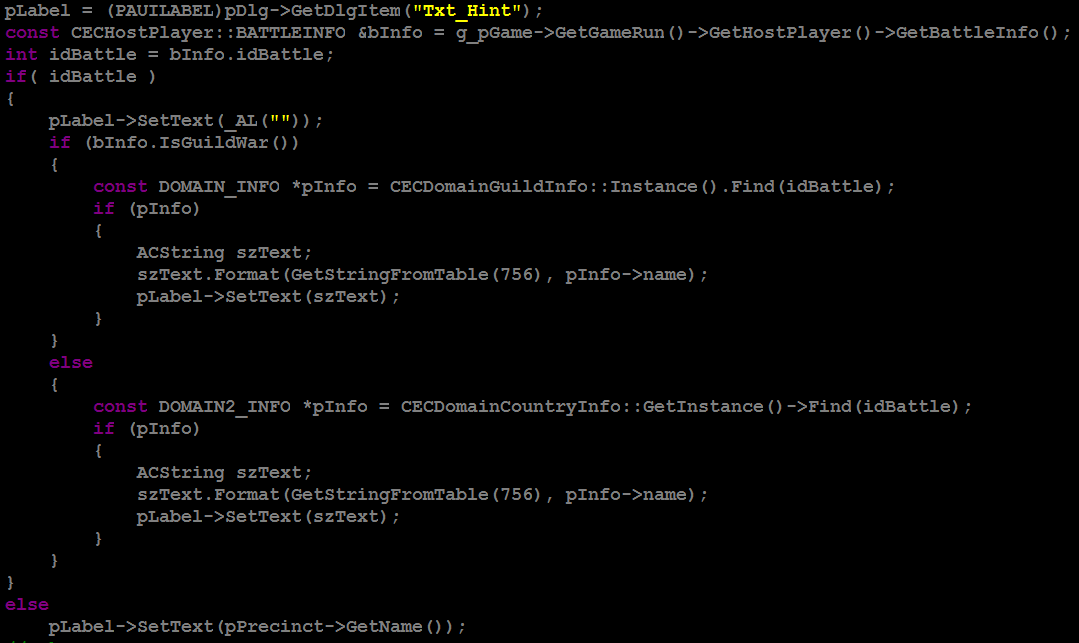
1. **include顺序**

包含项目中用到的文件、包含第三方库及引擎用到的文件、包含C++、windows系统头文件。除项目自身添加的文件，其它的使用#include <””>引入。

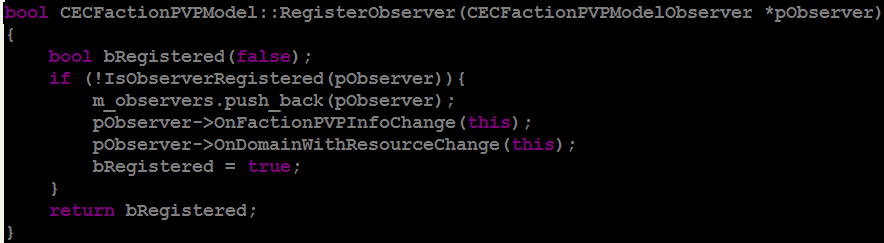


1. **if、while、for 等语句块中{}的使用**

推荐使用{放在末尾的方法，并始终使用{}。这样的好处是，同样的屏幕空间可以查看更多的代码行数，代码更加紧凑美观，如下：



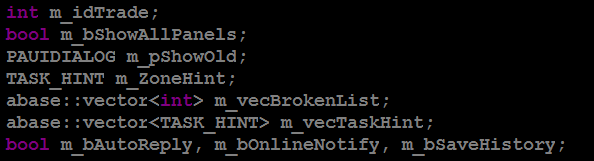
上图是没有使用这种规范的情况，屏幕空间利用率较低；下面使用了这种规范：



始终使用{}的原因是，保证块内单起一行，调试时比较方便，另外方便后续添加更多代码。

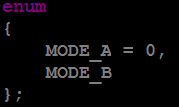
1. **缩进**

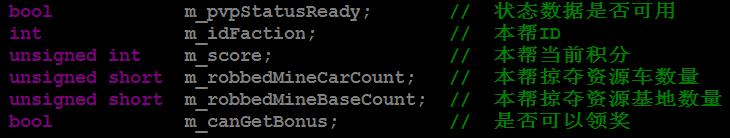
类变量、类成员方法定义时要缩进并对齐，杂乱的排序影响代码的阅读：



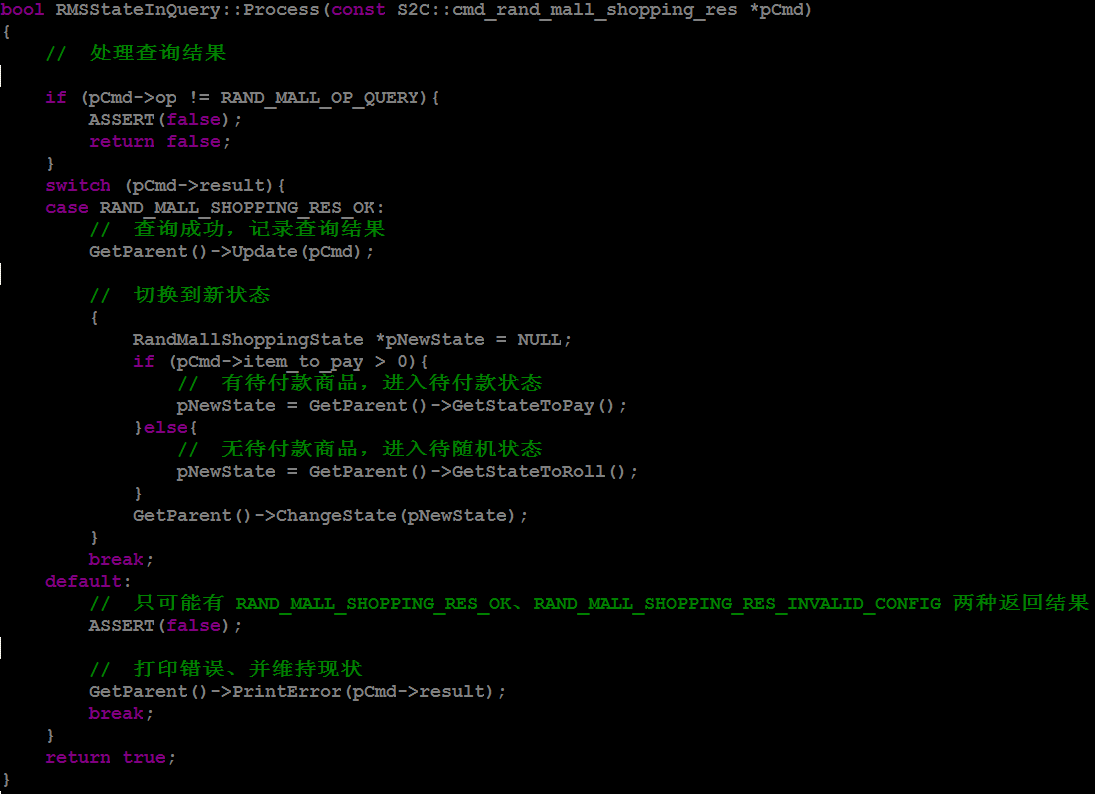
1. **添加必要的注释并使用中文**

程序中在必要的地方应添加注释，以增强可读性。

1. 变量名称后添加注释，解释变量的具体含义，尤其是当变量的命名理解有难度时，如：，不看代码不容易知道模式的具体含义：



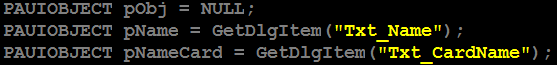
1. 代码按功能分块添加注释，以体现作者的思路，方便理解

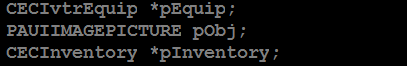


另外，用大家很容易理解的中文，有助于增加可读性。

1. **变量定义时请初始化**

可以减少BUG发生的机会：





需要使用第一种截图的方法，不能使用第二种。

1. **使用括号表示计算顺序**

多个运算符（逻辑与、逻辑或、位运算等）在一个表达式里运算时，尽量用括号表达其计算的先后顺序。不然计算顺序很容易搞混。



1. **控制好函数或方法的代码行数**

太长的实现，一般应用于复杂的文件读写操作。太长的实现，往往是函数功能不够单一。项目中逻辑功能实现中，如果发现超过100行，则需要考虑提取出公用的函数、分化函数的功能，提取出的函数，有可能在后续过程中为其它函数所用。

1. **促进合理的代码重用**

重复出现的代码，如果能够重用，则有以下好处：

1. 后续使用时，是经过一定测试的，减少测试工作量；
2. 修改BUG时，一处修改，多处受益，而且不会有遗忘；
3. 越是通用的可复用代码，在新的项目中可以直接使用；

代码的重用，尽量不要使用宏，而且有完整且合理的逻辑意义，仅仅是代码看起来一样就构建一个函数，后续用到的机会也就很少，重用机会也就很少，而且理解这个函数时也需要放在特定的环境中、理解起来有难度，不建议使用。

1. **使用常量变量代替数字**

比如游戏内的总职业数目NUM\_PROFESSION（【完美世界国际版】），之前的代码中到处使用8这个数字，添加新职业时，直接搜索8，搜到2522个，使用合适的正则表达式，还有243个，需要逐个查验修改，除此之外，还需要搜索9(PQ任务排行榜有8个按职业的排行榜+1个总排行榜)，还有10(NUM\_PROFESSION+2)、4(NUM\_PROFESSION/2)，甚至有没有指定数组大小的情况，导致添加新职业时，仅这方面的修改量就很大，还要细心和多次确认。修改后，程序中能搜索到NUM\_PROFESSION的有65处，以后直接按这个搜索逐个修改即可。

1. **为数组指定明确大小**

不使用明确大小带来的主要好处是少写一个数，如果这个数组的大小永远不变也无所谓，但往往这个数组的大小对应的类型在后续维护中是会扩充的（比如职业数目相关的）：

static int s\_profOrder[] = {0, 1, 7, 2, 3, 6, 4, 5};

定义了帮派成员列表中各职业的显示优先级。

又如：

WORD idFaceTex[][2] = {{44, 0},{0, 5},{855, 860},{0, 0},{0, 0},{845, 850},{0, 0},{0, 142}};

定义了脸部贴图纹理的默认ID，涉及每个职业男女两个性别。

再如：

int a\_nRaceProf[] = { 0, 0, 3, 1, 1, 3, 2, 2 };

定义了哪些职业ID是属于同一个种族。

还有其它类型一些情况，像各处使用的资源数目，和符合某些要求的物品ID配置表，像后面这种情况，如果要扩充，只能按ID查找，或者需要熟悉代码。指定了明确大小、并使用常量变量的情况下，情况将不一样。

1. **局部变量到使用时再定义**

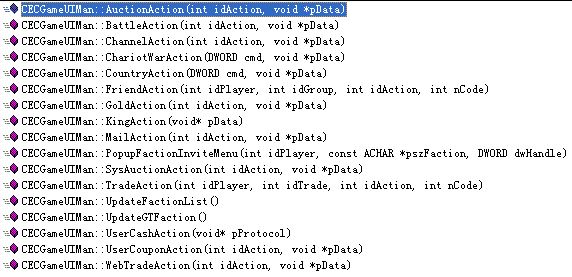
这是C++推崇的方法。传统的C语言都是将所有需要定义的变量定义在函数的开头部分，而阅读相关代码时，了解变量的定义和初始值，还需要跳很远，不方便阅读。另外，定义时才使用，有利于减少堆栈内存的使用，因为不使用的局部变量空间可以复用。

1. **可在客户端推广使用的通用逻辑**

在客户端代码中使用以下结构，提高代码的可维护性：

1. 根据服务器协议数据、客户端状态数据等构建数据模型，界面向数据模型注册接受通知，协议由数据模型处理，数据模型改变后，通知界面更新。

传统的方法以界面类为中心，如【完美世界国际版】项目中管理所有游戏界面的类CECGameUIMan：



里面定义了很多Action方法处理协议内容，在收到服务器协议时时，向各功能界面类推送数据更新（数据也可能分散存储在主玩家类CECHostPlayer、界面管理类CECGameUIMan中）、并在各功能界面类中根据数据构建显示内容，数据自身的逻辑和状态在界面中维护，还需要处理界面自身的变化（如动画），导致界面的功能负担过重，如果有多个界面使用同一套数据，还需要共享数据（将导致界面间相互引用）。因此，这种做法不利于维护和后续修改，甚至在初次撰写代码时也比较头痛。客户端的采用这种结构的代码比比皆是。

新方法的特点是界面逻辑和数据逻辑分开、并由界面逻辑向数据逻辑注册更新。两种逻辑分开的好处是可以分开演化，数据逻辑可以有代码重用和内部数据状态转变，而界面逻辑可以添加动画等自有逻辑，两部分功能分开后类的责任更加集中。而界面逻辑向数据逻辑注册的好处是，数据逻辑独立于界面逻辑、为多个注册类提供服务，不用再理会界面的各种繁琐更新状态，而界面逻辑每次收到更新时，只需要根据数据状态修改自身即可，不需要在界面里再有复杂判断何时再显示何值。

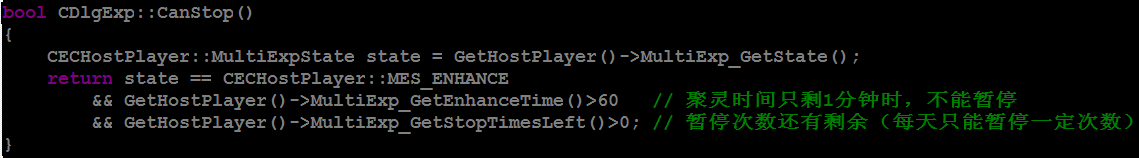
此方法在【完美世界国际版】中的典型应用是随机商城逻辑（RandMallShopping类、CDlgGeneralCardBuy类等）。

1. 单个游戏功能如果涉及多条服务器协议（如随机商城中有状态查询、随机、购买等过程），应使用状态模式构建客户端数据模型。状态模式的具体介绍，请参照《Head First 设计模式》中文版第385页。状态模式将各状态的逻辑分开，逻辑清晰，方便构建更加鲁棒的代码，也更易维护。此方法在【完美世界国际版】中的典型应用是随机商城逻辑的RandMallShopping类。
2. 客户端逻辑如果涉及复杂的状态切换，请使用状态模式。例如【完美世界国际版】中DisplayActionTrigger类，用于处理角色创建界面的人物旋转动画。
3. **不要用模拟替代服务器的真实状态**

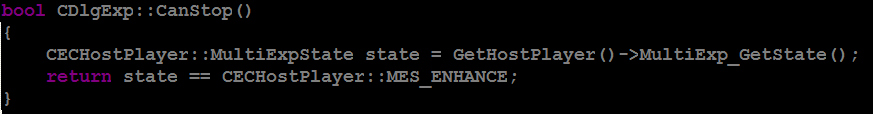
典型例子是【完美世界国际版】项目的聚灵系统：



此系统会在聚灵时间、缓冲时间、散灵时间3种状态间切换，其中，切换到聚灵时间修炼会加速，暂停聚灵则会切换到缓冲时间，如果缓冲时间已经用尽，则切换到散灵时间，聚灵时间自身用完后，也会强制切换到后面两个状态。其中两个按钮“暂停聚灵”、“开启聚灵”为玩家切换选项。此系统曾出过一个BUG，关于“暂停聚灵”按钮，以前的逻辑是这么写的：



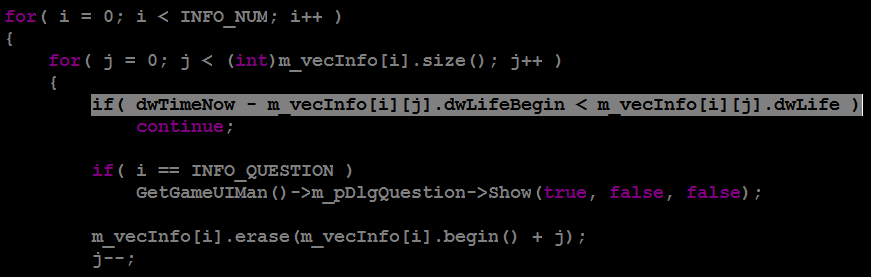
由于客户端与服务器的时间有误差，即GetHostPlayer()->MultiExp\_GetEnhanceTime()返回的客户端模拟值比服务器实际时间快，导致提前不能暂停聚灵。误差从几秒钟到几分钟甚至更多，导致玩家无法提前暂停聚灵而到关键时刻再使用。修改后，客户端只使用服务端同步来的状态判断：



客户端代码编写过程中，要做到以显示为主轴，涉及功能的逻辑状态判断要依赖服务器提供；特别注意的是，对于依赖到期时间的逻辑判断，客户端只在界面上提供显示，哪怕时间已经过了，那么在界面上显示剩余时间为0好了，但是只要服务器还没有说不可以，仍旧可以干还有剩余时间的事，即向服务器发送相关功能协议；如果服务器状态切换时没有提供通知，则应联系服务器同学补上。

1. **谨慎使用DWORD GetTickCount()计算时间差**

GetTickCount()是Windows API，之前很多代码记录某时刻结果，下一时刻再调用此函数与之计算时间差：



实际中发现，后调用的GetTickCount()在有的机器上有几率比之前的结果还要小，结果导致更大值（因为GetTickCount()返回结果是无符号数）。因此，相减前应做大小判断。【完美国际】提供CECTimeSafeChecker类解决此问题。

**二、完美世界国际版（完美国际）、完美世界经典版（收费完美）项目附加规范**

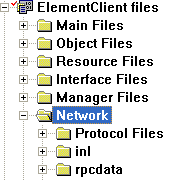
1. **类、对应文件名称**

界面对话框命名使用CDlg开头，如CDlgMiniMap、CDlgWorldMap，对应文件名称为Dlg替换CDlg，如DlgWorldMap.h、DlgWorldMap.cpp；

重要逻辑类采用CEC开头，如CECGameRun、CECUIConfig，对应文件名称为EC\_替换CEC，如EC\_GameRun.h、EC\_GameRun.cpp；

1. **协议要添加到项目中**

新增加的协议需要添加到项目Network子目录下（下图中Protocol Files、inl、rpcdata）:



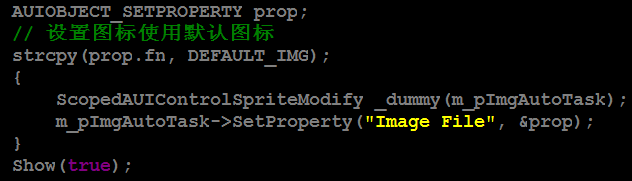
如accountloginrecord.hpp添加到Protocol Files项目目录中，而accountloginrecord添加到inl项目目录中。

不做上述处理也能通过编译，但是不便于从Visual Assist X（VC IDE插件） 中查找文件、影响协议查看等。

1. **图片路径格式、及使用 SetProperty 修改界面图片**

程序引用路径时，统一使用’\’。因引擎底层A2DSprite类区分对待’/’和’\’，因此，同一份资源，应只使用同一种分隔符。又因界面xml文件及ecm等文件格式中都使用了’\’，为最大程度共享图片，同它们保持一致。

针对**完美国际**，使用SetProperty修改界面图片时，使用ScopedAUIControlSpriteModify（对应单个控件所属资源）及ScopedDialogSpriteModify（对应对话框所属资源），以正确处理界面内存优化，具体原因及正确使用这两个类的方法见完美国际源码服务器，路径为”$/CElement/Doc/完美国际界面内存优化.docx”，以下为使用范例（摘自CDlgAutoTask类）：



1. **程序控制对话框、控件位置及大小时测试流程**

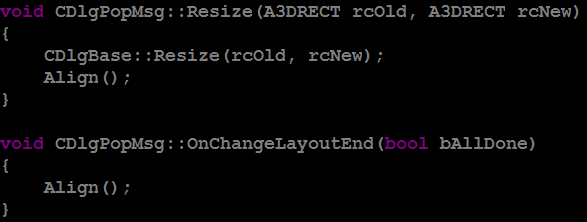
如果在程序中修改了对话框、或者控件的位置、大小、对齐等，则需要额外在以下3个条件下自行测试：

(1) 宽屏

(2) 随意拖拽游戏窗口大小

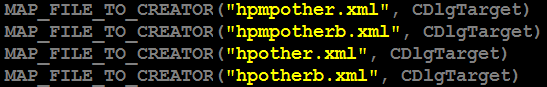
(3) 切换界面风格。

如CDlgPopMsg类在屏幕中间上方居中显示文字喊话，为处理（2），实现了AUIDialog::Resize(A3DRECT, A3DRECT)，为处理（3），实现了祖先类的函数 CDlgTheme::OnChangeLayoutEnd(bool)：



1. **不要直接使用同一个对话框类（或派生）处理多个相近对话框**

如CDlgTarget类，实现了4个类的功能（选中怪物时2种界面、选中玩家时2种界面）：



运行时使用其中的一个界面（hpother.xml）作为入口，更新4个界面的内容（当前4个界面不会同时出现），即在CDlgTarget::RefreshTargetStat()中，有时候同时处理4个界面、有时候通过if根据当前选中的对象类型（玩家或NPC）或某控件是否存在从而对4个中的某个界面执行某段逻辑，导致这个函数长达323行，非常难于查看和维护。

关于派生的典型例子是CDlgTask及从其派生的CDlgTaskList及CDlgWikiTaskDetail类。CDlgTask类是频繁改动的类，刚修复了一个崩溃BUG，CDlgTask类中对某控件的假定、没考虑到子类CDlgWikiTaskDetail，从而导致后者崩溃。

对话框类的功能应相互独立、从而减少某个修改的影响范围、减少维护代价。子类应扩展父类，而不应该在父类修改的时候还要重复测试子类、成为父类的累赘。

1. **不要在对话框及其控件中通过 SetData、SetDataPtr 保存各包裹物品指针**

NPC的许多服务功能，都需要从包裹中拖拽物品到某AUIDialog界面上（有些界面有右键快捷功能，本质是一样的）。在处理时，经常会有通过AUIDialog::SetData(DWORD,AString)、AUIDialog::SetDataPtr(void \*, AString)保存通过CECInventory::GetItem(int iSlot, bool bRemove=false)获取的物品的指针CECIvtrItem\*，然后在某个时候再取出来更新显示或发送协议。显然，这种做法是不安全的，因为包裹在少数情况下是可能变动的，原先的指针会成为悬留指针。

比较正确的用法是保留物品在包裹中的下标iSlot及物品id等，并在使用时验证。对于新项目而言，应有其它机制（典型如包裹变化时对界面的通知机制），从根本上解决这个问题。

1. **拼凑字符串时，请考虑本地化（应用于完美国际项目）**

由于完美国际是国际化项目，应用于英语、法语、俄语等，应尽量减少拼凑，尤其不要强行拼凑，仅按照中文的分词习惯，达到共享代码，必要的重复是推荐的，尽量提供完整的语句。