# 头文件

## 按功能划分头文件

将宏定义按功能来细分，写在不同的头文件中，增加可读性和方便快速明白的定位代码，如：

* item.hrl：则只包含与物品相关的宏定义，如颜色品质的定义等
* task.hrl：则只包含与任务相关的宏定义
* scene.hrl：则只包含与场景相关的宏定义

## 2．需要时尽可能使用宏定义

察觉需要宏定义时就尽可能的定义宏定义，比如各种分类，并将这些相关的宏定义放在一起，以便让代码阅读者知道这是要干啥以及有多少种分类会处理到。

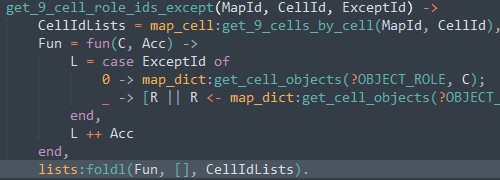
# 缩进

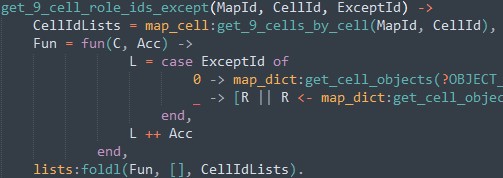
## 缩进统一为4个字符宽度的tab

## 不整体缩进

（这个不强求，看代码布局，主要目的是不要让代码难以阅读）

编写代码时存在如下两种缩进方式：





我们采用第一种方式，主要是考虑代码不会右移很多而造成阅读不便

# 命名约定

命名风格最好一至，并且要**可读性强**，不然别人不知道那个变量或者函数到底是个干啥的，增加了阅读的痛苦程度和出bug的可能性，不利于项目的维护。具体规则列于下：

* 尽量是使用英文命令，且是常用的英文单词最好
* 变量或者函数的名称不要过度缩写，明明白白最好，如：

Num = 1; %% 好的方式

SceneType = city, %% 好的方式

get\_all\_monsters() %% 好的方式

N = 1; %% 不好的方式

ST = city, %% 不好的方式

get\_ml() %% 不好的方式

* 变量命名一律开头都大写，函数命令都是小写并且单词之间用下划线，如：

ItemType

UserId

add\_monster()

* 宏定义一律大写，单词之间用下划线隔开，如：

?ITEM\_COLOR\_WHITE

?SEX\_MAN

* 函数的命名最好应该能简单的描述这个函数做的事情，如：

set\_flag %% 设置一个flag

get\_role\_lv %% 获取玩家的等级

handle\_role\_enter\_scene\_event %% 处理玩家进入场景的事件

* 事件命名，我们的代码中很多地方会抛出一个事件代码来方便别人插入代码的，定义一个统一的命名规则，能很方便的让人知道在哪里插入自己的代码，规则如下：

on\_role\_enter\_scene\_event(RoleId) ->

fun\_copy:handle\_role\_enter\_scene\_event(RoleId),

ok.

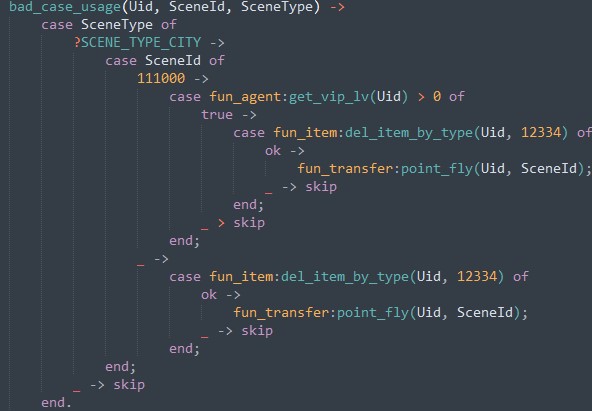
on\_equip\_strengthen(EquipRec, OldLv, NewLv) ->

fun\_task:process\_event(?TASK\_EVENT\_EQUIP\_STRENGTHEN,{EquipRec,OldLv,NewLv }),

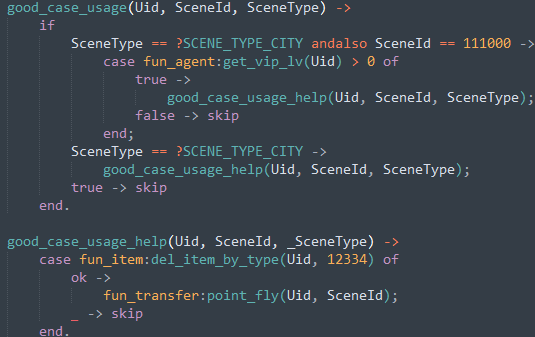
ok.

# case/if嵌套

case/if嵌套最好别超过3个，原则上不能超过5个，否则这样的代码必定会难看的，一行的字符数也会超过标准的80个的。当改代码发现嵌套要超过3个时就该考虑如何重构下这个代码了。如下面的代码：



有4个case嵌套了，而且还有重复的代码，以后的需求变得复杂点，在多个条件判定肯定会更加难看了，重构如下：



如此便简单明了很多了，也去除了重复代码。

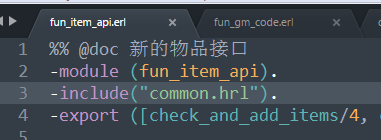
# 注释

最好的注释就是代码本身，只要代码写的可读性很强，基本上也不用啥注释了。

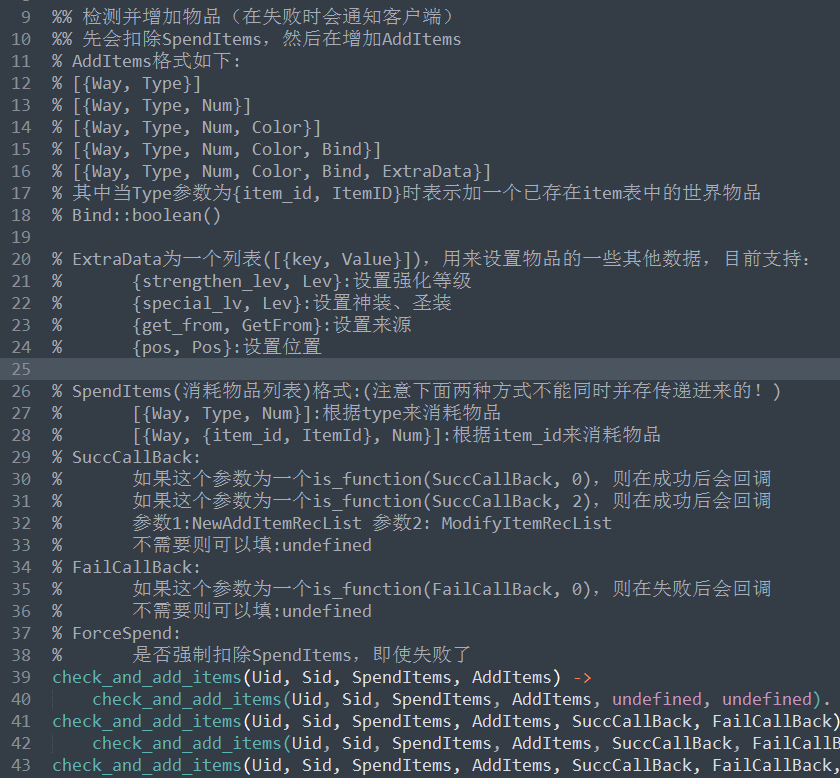
但是有些复杂而且容易让人误会的地方就要加上注释提示后来的维护者。

还有很多时候代码写完了，更多的是后面的维护者如何能找到和大致知道这个代码是做啥的，这个时候必要的简短注释还是很有必要的，如模块开头等，具体注释的地方如下：

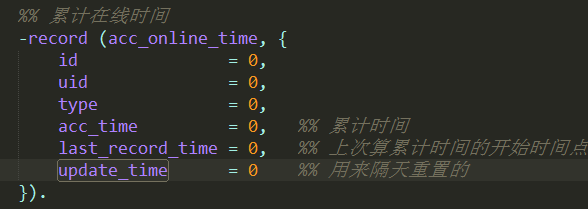
* 一个模块的开头增加如下注释，以简要说明该模块是干啥的：

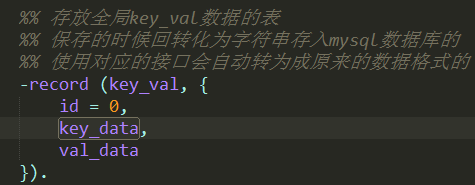


* 方法参数比较复杂或api公共接口之类的函数，明白说明下告知别人如何使用和传递参数：

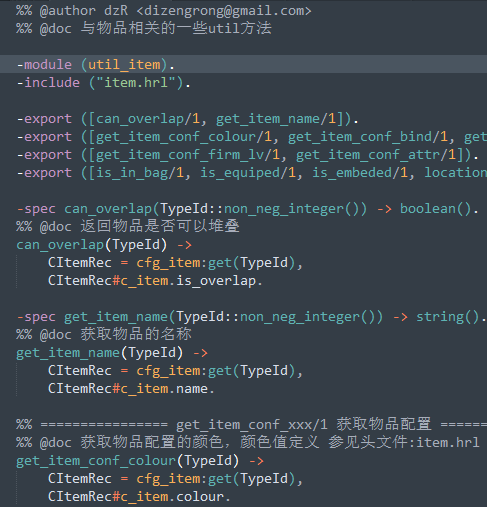


* 对record定义加上注释，让后来的人大概知道这个record是干什么的：





注释示例：

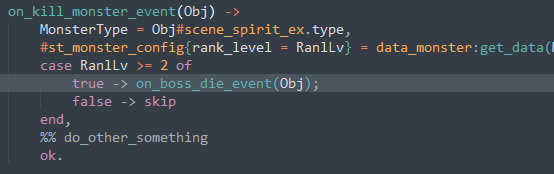


# 尽量重构代码以消除重复代码

重复的代码不仅让代码显得臃肿难看，而且还是bug的温床，以后维护的代价也是很高的。

# 尽量重构以提供公共的方法

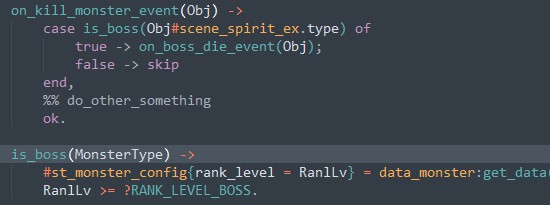
对一些公共的东西，尽量去编写一个方法来获取，而不是本次编写用了就不管别人了，如果都这样做的话那代码里就到处充斥着重复编写的代码了，如下面的代码，是为了判定一个怪物是否为boss的代码，由于没有人去封装一个方法来判定一个怪物是不是boss，各自写各自的，如果需求变更那改动的地方将会很多，而这就是维护成本！



这里有3个可以修正的点：

1. 代码可读性不强，要直到读到on\_boss\_die\_event方法才知道RanlLv >= 2是判定是否为boss的。
2. 数字2没有宏定义，这将导致以后2变为3后，所有判定为boss的代码都要找出来一一修改！如果定义了宏定义则只需要改一个宏定义就可以搞定了！
3. 没有封装代码，就意味着每次都要拷贝这么多代码来做事情，明显写起来会感觉不顺手，会有一种这么基础的东西居然还要自己再一个来做。

重构后的代码如下：



清爽明了，而且可读性很强

# 一些好的风格代码



