第 3 章　低级错误大全

一些低级错误可能是阻碍一个初学者入门的最大障碍。低级错误可能会狠狠地打击初学者的信心和学习热情！入门篇的目的是为初学者扫清入门障碍，巩固基础，建立对 Cocos2d-x 的整体认识，降低初学过程的出错几率。

在使用 Cocos2d-x 时，特别是新手，不犯错误是不可能的，本章将告诉你，Cocos2d-x 的代码，不能怎样写！这里面有些是习惯问题，有些是新手比较容易犯的问题，也有些是老手都有可能不小心中招的问题。本章尽量将这些问题的症状和原因一一列出，以便中招的时候能够提供一些线索。

这里总结的一些问题是笔者从 Cocos2d-x 1.x 版本到现在的 Cocos2d-x 3.x 发现的问题，随着版本的更新，其中一些问题已经被修复，但大部分的修复只是从运行错误变为了编译错误。本章主要介绍以下内容：

- create 和 retain-release。
- 继承对象的 create。
- 多个对象执行同一个 Action。
- 忘记调用父类的虚函数。
- 隐藏在代码中的神秘杀手，节点中的节点操作。
- 普通对象和 new 出来的对象。
- 不要忘记 init。
- addChild 失败。
- 在 onEnter 中调用 parent 的 addChild。
- 忘记移除。
- 重载 draw 注意事项。
- 关于引用。
- 关于命名空间。
- 关于类之间的互相包含。
- 关于平台相关的 API。
- 关于 update 中写逻辑。
- 关于调试。

3.1　create 和 retain-release

这是一位自称有 8 年 C++经验的“大牛”犯下的低级错误，错误是这样产生的，在初始化时使用 create 方法创建了一个动画对象，在初始化函数中进行测试，这个动画是可以

正常播放的。

根据需求，需要在对象被攻击到时播放这个动画，于是将动画对象作为成员变量，在 init 中使用 create 方法创建动画对象，初始化后并赋值给成员变量。

将播放动画的代码迁移至 update 的逻辑中，在触发相应的事件时，播放动画，但运行后就崩溃了，原因是动画对象被释放了。

因为使用 create 方法创建出来的对象的 release 在这一帧会被 Cocos2d-x 调用，**如果不希望它被释放，请确保你执行了 retain 操作。执行之后调用 release 操作来释放，确保没有内存泄漏**。更多关于内存的问题可以参考第 7 章。

解决这个问题有两种方法，第一种是在每次需要使用一个对象的时候，才调用 create 方法进行创建，Cocos2d-x 中大部分情况下都是这样使用的。第二种是在初始化时创建一次，然后调用对象的 retain 方法，以避免该对象在外部被释放，后在不需要使用该对象时，调用对象的 release 方法，以避免内存泄露。

注意：文中 retain 指执行 retain（引用）操作，或调用 retain（引用）操作。rease 指执行 release（释放）操作，调用 release（释放）操作。后面为表述方便，直接用 retain 和 release 来表达。

3.2　继承对象的 create

这是笔者曾经犯过的一个小错误，笔者的对象 MySprite 继承于 Sprite，然后习惯性地使用 MySprite::create 来创建对象，结果发现 create 创建出来的对象并不是 MySprite 而是 Sprite。这会导致一些奇怪的表现，而原因正是在 MySprite 的定义中遗漏了一行代码——**CREAT_FUNC(MySprite)**，导致 create 调用的是父类 Sprite 的方法。

3.3　多个对象执行同一个 Action

当希望多个对象执行同一个 Action 时，并不能这样操作：

```
Action* act = MoveTo::create(...);
sprite1->runAction(act);
sprite2->runAction(act);
```

因为**每个 Action 只能被一个对象执行**，如果希望多个对象执行一个 Action，可以使用 clone 来创建一个新的实例。

3.4　忘记调用父类的虚函数

这是一个很经典，很容易犯的错误，当我第一次碰到这个错误的时候，浪费了很多时间，因此希望能帮大家节省一些时间。如果你继承了某个 Cocos2d-x 的类，又重写了它的

init 或者 onEnter 函数，那么一不小心就可能出现一些比较奇怪的问题，例如，对这个对象施加一个动作，而这个对象没有任何反应，那么首先你会从这个动作上来找问题，而很难想到是因为没有调用到父类的函数。

在很多情况下都需要用到继承、重写虚函数，当重写虚函数的时候，先看一下父类的这个虚函数做了什么。如果这些操作是必须的，就在函数中对父类的虚函数进行调用，如 CGameObject 继承了 Sprite 并重写了 Sprite 的 onEnter。

```
void CGameObject::onEnter()
{
    //切记回调父类
    Sprite::onEnter();
    scheduleUpdate();
}
```

3.5　隐藏在代码中的神秘杀手，节点中的节点操作

这是笔者 2013 年碰到的 BUG，归功于自动提示，它制造了一个很隐蔽的 BUG，笔者将一个节点添加到了另外一个节点中，希望每次单击它都能够向上移动 x 个单位，于是代码变成了这样：

```
node->setPositionY(getPositionY() + dis);
```

然后，这个对象就突然消失不见了，笔者以为是其他地方改变了其位置，或者被隐藏了或意外地删掉了，最后笔者跟踪到代码里面进行调试才发现，括号中的 **getPositionY 返回的并不是自己的 Y 坐标，而是 this 的 Y 坐标，this != node**，所以其就偏到屏幕外去了，在节点中操作其他节点时，代码不要输入太快！

3.6　普通对象和 new 出来的对象

如果读者刚接触 C++，那么很可能分不清楚普通对象和 new 出来的对象有什么不一样，在写代码的时候，会看到如图 3-1 所示对话框。

图 3-1　程序崩溃

当把一个 Sprite 添加到场景中时，弹出了图 3-1 所示对话框，仔细看一下代码，没错，要传入一个指针，这里确实传了一个指针进去，由于指针使用不当而引起的 BUG 太多太多了！对于下面这种情况，**sp 是在栈空间分配的，当这个函数执行完成返回的时候，就被释放了**，而 new 出来的对象是在堆空间分配的，只有当手动删掉的时候才会释放，而在栈空间分配的对象不能使用 delete 操作符来释放。在 Cocos2d-x 中，应该遵循 Cocos2d-x 的内存管理规则，使用 release 方法来释放 Cocos2d-x 的对象，而不是使用 delete 操作符。

```
Sprite sp;
sp.initWithFile("HelloWorld.png");
this->addChild(&sp);
```

这个问题在 3.0 之后就不会再遇到了，上面的代码会变成一个编译错误，因为 Sprite 只能使用 create 方式来创建，不能 new 也不能定义为普通的局部变量，但如果创建的是一个子类，那么还是会出现这个错误的。

3.7　不要忘记 init

在 Cocos2d-x 的诸多对象中，大部分都提供了 create 方法，它将返回一个 new 出来的对象，并且这个对象调用了 init 和 autorelease 函数。在 Cocos2d-x 的对象中，往往都**没有把初始化放在构造函数中**，而是放在 init 函数中，当手动 new 了一个对象而忘记调用它的 init 的时候，可能会出现各种错误，如图 3-2 所示。

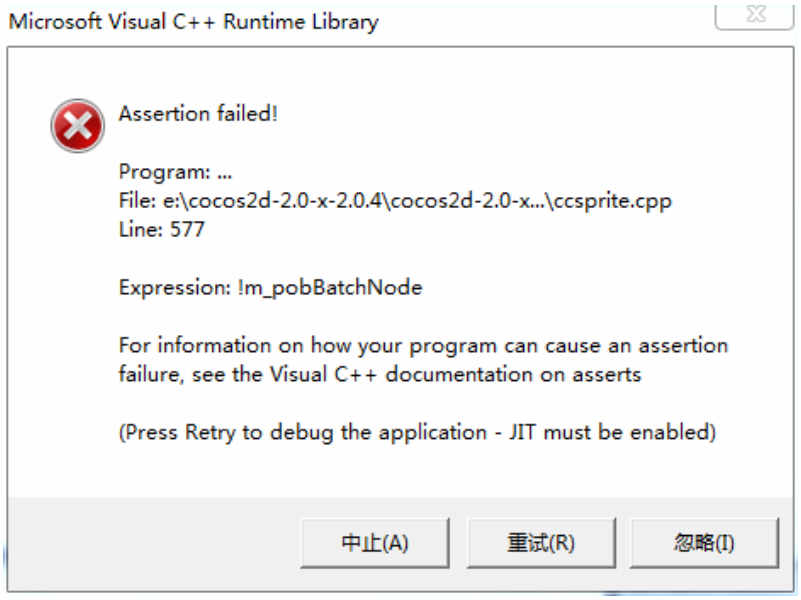

图 3-2　未初始化导致的崩溃

这种先构造再初始化的方法，被称为“**二阶段构造机制**”。而这个二阶段构造机制，简单地说，就是因为在构造函数中可能需要初始化很多东西，因此就有可能出现失败的情况，而我们知道，构造函数是没有返回值的，所以要捕获在构造函数中出现的错误，则比

较麻烦，而将要初始化的内容放到 init 回调函数中，就可以很直观地知道这个对象是否成功初始化，这比在对象中添加一个错误码，或者使用 try-catch，要方便、简单得多，所以在 new 一个对象之后，请随手调用它的 init 如果有的话。

由于 3.0 将很多对象的构造函数和析构函数取消了 public，所以多数情况下只能使用 create 来创建 Cocos2d-x 对象。

3.8　addChild 失败

其实该问题是一个陈年老 BUG 了，在这里提出来是因为其比较有意思，当时笔者曾试了一下将两个节点互为父子节点（死循环直到栈空间溢出），或者将一个节点 add 到两个节点上，则会有如图 3-3 所示的栈溢出。而现在的版本已经修正该错误，一个节点被限定为只能有一个父节点，否则会触发断言。

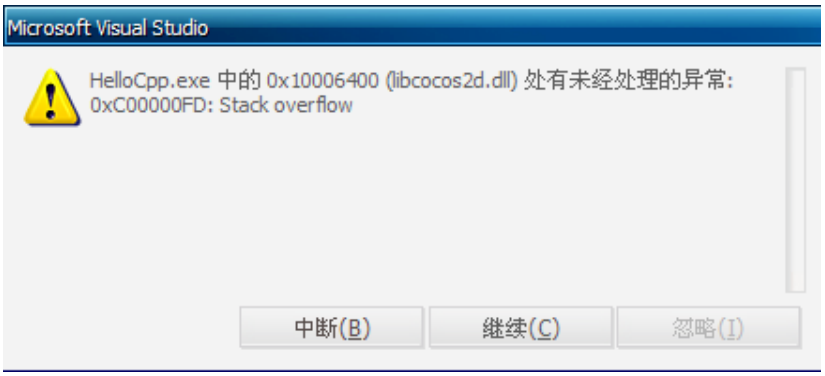

图 3-3　栈溢出

所谓内存溢出，是指内存不足，可用内存小于程序运行所需的内存。

此时查看堆栈会发现堆栈几乎被 onEnter 函数的调用填满了，嵌套了一层又一层的 onEnter 函数调用。A 调用了 B 的 onEnter 方法，B 又调 A 的 onEnter 方法，反复循环。这个问题读者了解即可。

3.9　在 onEnter 中调用 parent 的 addChild

addChild 和 removeChild 是常用的方法，经常在 onEnter、onExit、onEnterTransitionDidFinish 和 onExitTransitionDidStart 等方法中调用这些方法。

这里隐藏着一个问题，就是当此时调用父节点的 **addChild** 或 **removeChild** 时，程序可能会崩溃。这是因为这些方法都是**在父节点中遍历子节点调用的**，也就是说，如在数组遍历的过程中动态添加或删除了数组中的元素，那么这个操作就很危险，虽然 Cocos2d-x 自身允许这样操作，但这种操作存在危险！

3.10　忘 记 移 除

在使用 NotifierCenter 的时候，一个很容易使程序崩溃的问题就是，你的对象添加到了 NotifierCenter 中而又忘记移除，这样的结果是当对象从场景中移除时，NotifierCenter 还会调用对象的回调，此时程序可能会崩溃。

虽然 Cocos2d-x 3.0 之后废弃了 NotifierCenter，使用 EventDispatcher 来替代，事件的监听器关联到了 Node 对象上，在 Node 对象被释放时会自动将关联到该对象的监听器注销，但在做一些添加注册的时候，程序员应该习惯性地想到何时移除，是否 Cocos2d-x 自动帮你做了这个事情。

3.11　重载 draw 注意事项

想要按照自己的规则进行渲染，先继承于 Node 类或 Node 的子类，因为在这里只有节点才能被渲染，请不要破坏 Cocos2d-x 的封装，遵守其规则，之所以选择 Node，是因为比较简单，然后只需要在其 draw 函数里进行绘制就可以了。

假设希望绘制的东西可以挂在某个节点上面，在绘制之前，可调用 CC_NODE_DRAW_SETUP();，来初始化矩阵。

注意：Cocos2d-x 3.0 之后使用了 Render 来执行渲染，所以需要将渲染的逻辑实现在一个自定义的渲染命令中，在 Node 的 draw 方法中将该命令添加到 Cocos2d-x 的渲染器 Renderer 中。

3.12　关 于 引 用

如果需要将某个对象传入到一个函数中，然后在函数执行的过程中修改该对象的某些属性，那么可以使用指针，但出于安全性的考虑，还是建议使用引用。

另外，当在传递容器的时候，引用的效果也非常好。假设只是传递一个容器供函数查询，而不希望它修改，那么可以用 const 引用，普通的值传递将会重新创建一个容器，然后将用户传入容器里的内容复制到临时容器中，如果容器的内容比较多，对效率的影响也较大。

如果将引用作为返回值，则不要返回一个局部对象（在函数中定义的临时对象），否则会有意想不到的情况发生，如程序崩溃。

3.13　关于命名空间

在使用 Cocos2d-x 的时候，需要使用 using 关键字来导入一个命名空间，可能有的人

会在头文件中直接使用 using 关键字导入命名空间，这是一种非常不好的做法，这种做法虽然会让程序员少写一些代码，但是在头文件中这么写，命名空间就失去意义了，而且可能会引起命名空间混乱，在此之前可能一切运行良好，而当程序员去包含某个文件的时候，则会突然发现一堆重复定义的错误。

好的习惯应该是在头文件中，直接使用命名空间的规则，在源文件中，将 using 语句写在所有#include 的后面。

3.14　关于类之间的互相包含

在两个类 A 和 B 需要互相包含的时候，需要一个向前声明，即写一行 class A；在 B 的前面，写一行 class B；在 A 的前面，以通过编译。这说明一个问题：即这两个类直接互相依赖了，而我们应该尽量减少类之间的依赖关系，让每个类更加独立一些。

3.15　关于平台相关的 API

Cocos2d-x 是一个跨平台的移动端游戏引擎，一般在 Windows 上编写代码并调试，因此可能会有意无意地加入一些 Windows 的 API 调用，例如 MessageBox 之类的，一般情况下，不要使用与平台相关的 API，否则在移植到其他平台时，会有各种麻烦的问题，如有不得不加入的与平台相关的代码，则需要用预处理来区分开。

3.16　关于 update 中写逻辑

我们喜欢在 Update 里写游戏逻辑，在每个对象的 Update 里做自己的事情，例如，在 Update 里根据当前的状态使用不同的规则来更新位置，现在是往上走还是往下走？速度大概多少？输入 5 运行一下，速度太慢了，输入 50 运行一下，速度又太快了，输入 25 的话，看起来速度正合适，在 Win32 上感觉良好。根据这种感觉，调整其他 100 种怪物的移动速度，最后导到其他设备上，你会发现，速度怎么不对？而且有的快有的慢，一切都乱了。

在 Update 里更新一些数值的时候，请**乘以 dt 参数**，这样就不会因为不同设备的帧频不同，而导致结果不同，另外也可以让显示对象的更新更加平滑。将调整好的值乘以设定的 FPS（windows 下默认是 60），将这个值作为新的速度，然后在修改位置的地方，改成下面这样，dt 的单位为秒。

```
void CGameObject::update(float dt)
{
    setPositionX(getPositionX() + m_MoveSpeed * dt);
}
```

不要把逻辑全部写在 Update 里，尽量让 Update 的逻辑清晰简单。例如，要做一个比较复杂的动作，播放一个攻击动画，当这个动画播放到第 X 帧的时候，对这个怪物造成伤害，如果这个怪物“挂掉”了，则切换回移动状态，并且播放移动的动画，假设它没“挂

掉”，那么再攻击一次，将这个逻辑写在 Update 中，会比较复杂。

通过使用 Action 可以大大简化代码，更容易维护，也更稳定。在这里完全可以利用 Cocos2d-x 的 Action 来做这些简单的事情，如下面的代码：

```
void CGameObject::AttackByAnimate()
{
  //播放攻击动画，并在动画播放完成的时候调用 onAttackAnimateFinish 函数，该函数只判
  断目标是否还活着，如活着的就再调用一次 AttackByAnimate
  CCFiniteTimeAction* seq = CCSequence::create(m_AttackAniAction,
      CCCallFunc::create(this, callfunc_selector(CGameObject::onAttackAn
      imateFinish)),
      NULL);

  //在 m_AttackTick 帧的时候调用 onAttack 对目标造成伤害
  CCFiniteTimeAction* seqattack = CCSequence::create(
      CCDelayTime::create(m_AttackTick*m_Duration),
      CCCallFuncO::create(this, callfuncO_selector(CGameObject::onAttack
      ), m_Target),
      NULL);

  //将这两个动画同步执行
  CCFiniteTimeAction* spawn = CCSpawn::create(seq, seqattack, NULL);
  spawn->setTag(GAME_CURACTION);
  runAction(spawn);
}
```

3.17　关 于 调 试

如果将程序的开发分为思考与设计、编码、调试 3 个阶段的话，那么它们所占时间的比例大概为 30%、10%和 60%。这个比例并不准确，只是想说明程序员会有大部分时间花在调试上。调试能力是一个程序员应该掌握的基本技能，所以读者需要掌握好（这句话是对没有编程经验的初学者说的）。

CCLOG 打印日志是 Cocos2d-x 中最初级的调试方法，并且在 Android 和 iOS 下都能很好地工作。

断点调试是非常高效的调试手段，通过设置断点可以暂停游戏的流程，调试流程，观察变量的数值，这里详细介绍一下。

在代码左侧的空行单击可以添加一个断点（XCode 也是如此），再次单击可以删除该断点，或者右击，在弹出的断点菜单项中选择“插入断点”或“删除断点”命令。在调试菜单中可以选择删除所有断点，如图 3-4 所示。

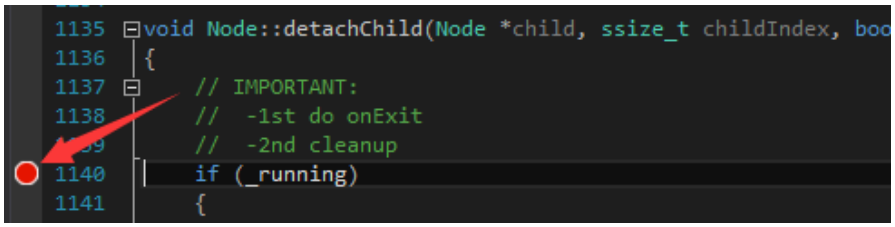

图 3-4　添加断点

设置断点之后，选择调试运行，在执行到断点处会停住，并弹出断点所在位置，坐标的空行会出现一个黄色的小箭头，表示当前所执行到的位置，如图 3-5 所示。按 F5、F10 和 F11 等键可以进行调试（Xcode 是其他快捷键）。

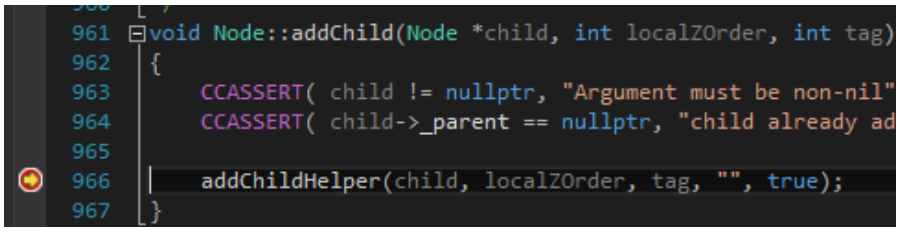

图 3-5　当前执行位置

拖曳黄色的小箭头到其他代码行，可以**在函数中强行改变程序的执行流程**。将鼠标光标移动到变量上，可观察变量当前的数值。也可以在变量上右击添加一个监视，这样可以在监视窗口查看该变量，如图 3-6 所示。

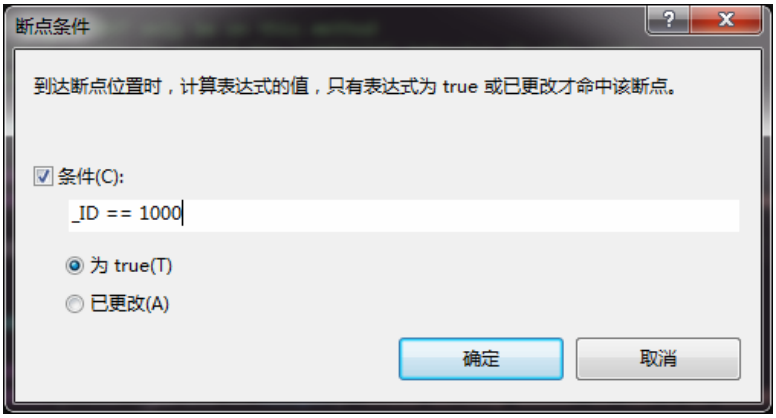

图 3-6　条件断点

在断点上右击选择条件断点，会在指定的条件成立时才停住断点，过滤掉不满足条件的断点，以减少被打断的时间，提高调试效率，条件断点相当于在当前断点行添加一个条件语句包围该代码，如图 3-7 所示。

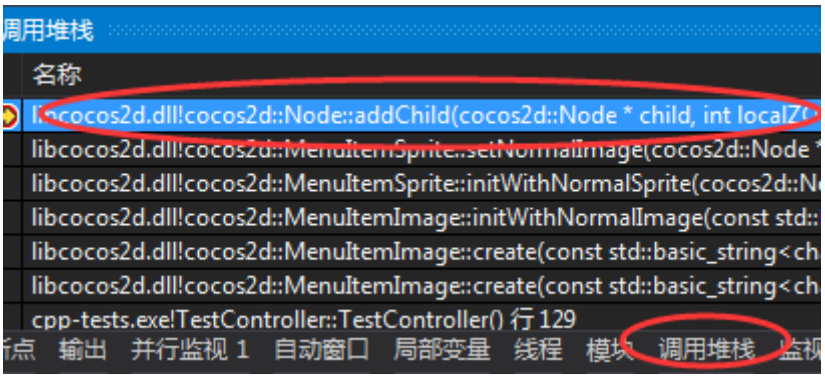

图 3-7　调用堆栈

在程序崩溃时，如果崩溃处有大量外包调用，查看调用堆栈可以分析崩溃函数的入口，帮助分析程序崩溃原因。如果无法调试到 Cocos2d-x 的代码中，可把项目添加到 Cocos2d-x 的 sln 解决方案中，在里面调试；如果还是解决不了问题，可拿着堆栈和崩溃处代码去请教其他人，看上去会更有诚意一些（一些新手问问题的时候，总喜欢说一大堆与问题无关的话题，所以每次笔者先回复一句：说重点，堆栈和崩溃处的代码就是重点）。

3.18　小　　结

本章中的问题大多是 C++的一些基础问题，如果读者对 C++的基础不是很扎实的话，推荐看**《Effective C++》**这本书，如果完全没有 C++基础，不妨先看看**《21 天精通 C++》**这样的书。了解一点简单的语法之后，再看**《C++Primer》**和**《Effective C++》**这样的书会较好一点。中间说到了 onEnter 和 Action，读者听起来可能感觉云里雾里一样不明白，没有关系，带着疑问继续看下去，所有的疑问都会得到解决。