转载请注，9ria.com，General\_Clarke著。  
发此帖时已在论坛见过多条希望解释为什么要用MVC框架、为什么要用接口、接口和父类继承有什么区别的帖子  
使用上述办法其实目的只有一个——脱耦。  
兹专门拿出一篇帖子来，深入说一说耦合与脱耦。  
文章不只针对AS，本文对各种三代编程语言都适用  
  
目录：  
一、几个基本概念：模块、耦合与脱耦  
二、脱耦的误区  
三、耦合的7个级别  
四、怎样设计才能使耦合程度适当  
  
  
**一、几个基本概念：模块、耦合与脱耦**  
模块是一组功能的集合体，一个模块能处理一种特定事务。  
模块不一定指swf。为了完成同一事务需要几个不同的类，这几个类每个有自己的子业务要做，那么每个类其实可以考虑成一个“小模块”。同类内负责完全不同事务的代码也可以分别被考虑成“小小模块”。（但根据面向对象的单一职责原则，应避免一个类负责多项事务的情况存在）  
  
即使是初窥门径的新手也多少听到过“脱耦”这个词  
“耦合”的意思是模块之间的依赖  
对一个完整的程序来讲，模块之间总是有耦合的。  
耦合复杂的模块互相依赖程度高，牵一发动全身  
即使最好的框架也不能在模块间完全不互相发送信息的情况下实现模块的联动。  
我们能做到的“脱耦”其实并不是“消除模块间耦合”而是“降低模块间耦合”  
  
  
**二、脱耦的误区**  
“脱耦即是好的”其实是一个误区  
脱耦的同时将使项目修改的复杂程度提高，  
某些脱耦的办法还会增加代码量、降低执行效率。  
pureMVC是一个用来强力脱耦的框架  
其效率本身不是很高，函数调用层次较深，而有时根本不清楚消息发到了哪里，这被广大程序员在调试时所诟病。  
  
下文在谈到脱耦优点的同时也会提出脱耦的代价，  
读者设计程序或对已有程序修改时，可参考下文，确定这个耦合究竟脱到什么程度才合适。  
  
  
**三、耦合的7个级别**  
这是本文重点。  
耦合一共有7个等级，下文以lv7最高，lv1最低。  
高等级下，能轻易对功能进行修改，但对程序进行修改要考虑的元素越多、风险越大。  
而低等级下，虽然代码灵活可修改复用，反会使代码松散，层次增多，降低效率和修改定位的速度。  
恰当把握程序耦合度能扬长避短，事半功倍。  
  
lv7：内容耦合：A块直接修改B块的数据  
【举例】在A中有如下代码  
var b:B = new B();  
b.somePro = 1000;  
这是一种最强的耦合。  
【优点】在编译环境中直接按住Ctrl点击somePro就能找到正确的属性位置，最容易修改，逻辑简单。  
【缺点】耦合程度太高，B的这个属性修改后A的代码也要改，而B修改代码时始终要想这个属性是否还被其它块调用，在模块增多的时候导致程序员要想的东西越来越多，bug防不胜防，修改也是牵一发动全身。  
【适用】A和B密切相关、甚至可以写成一个模块的时候。要求A和B一个（或一组）被创建另一个（或一组）也被创建（或在适当情况下由已有对象创建），一个被销毁时另一个也被销毁。在这种“捆绑”式的块之间搞脱耦是多此一举。  
另适用于一次性抢时间的代码、小例子、打补丁等不是正常开发项目的情况  
【吐槽】面向对象的继承关系是内容耦合。面向对象的子类继承父类时，对protected修饰符以上的属性访问、对方法的override均属内容耦合的特征，修改父类的方法时，也会体现在子类身上。  
这种耦合可以接受，子类当然要完全依赖于父类，没有父类子类也就不能存在。  
面向对象保留了必须的内容耦合，并将这种耦合限定在有限的继承关系上，被依赖的父类较少，主动依赖的子类较多，藉此降低最容易出错部分的代码难度，这是面向对象推荐用继承而不使用直接修改的原因之一  
  
  
lv6：公共耦合：A和B共同访问某全局数据结构C  
【举例】在Flash中，Stage显示树只有一个，有多个模块共享显示树。  
【优点】公共数据结构C是A和B分别已声明的部分，不用找就能使用。编译环境中同样可以Ctrl+点击直接跟踪属性、方法位置  
【缺点】耦合度仅次于内容耦合，A改了自己的显示树也就相当于改了B的显示树。不妥善管理的话，A可能把B还要用的剪辑给removeChild掉。  
【适用】情况1：频繁调用的公共资源，类似Stage、控制台、对后台访问接口等。虽然可以将上述这些例子只交给一个A模块维护，其它模块需要功能时从A模块间接调用，但这些资源用处太多、调用过于频繁，这样要走一个比较麻烦的路线。没有非常高的安全要求时，有限制地使用公共耦合还是可以维护的。  
情况2：几个模块共享一块资源，但只有一个模块是写者，其它模块对这些资源只读，比如资源加载器。对这种情况使用公共耦合危险性小，且是最方便的。  
【吐槽】我所在项目已有模块将显示树分成若干层，并给每层添加了一个空影片剪辑来占位，但在写新模块时发现原来的分层根本不合理，为防止影响原有功能不敢修改了，只好拿原来的显示树凑合用。  
  
  
  
lv5：外部耦合：A和B共同引用一个全局项。外部耦合在很多情况下与公共耦合相似，可以当成是“较简单的公共耦合”  
【举例】A和B都在访问读写同一个全局变量，那么它们外部耦合  
【优点】被访问的变量是全局的，任何地方都能找到这个变量，始终有代码提示，也能使用Ctrl加点击跟踪到变量  
【缺点】随着程序加大，全局变量越来越多，程序员可能忘掉一些全局变量，或者忘记这个变量影响哪些模块。随着全局变量的增多，代码提示冗余度加大，代码提示第一行的命中率降低。。AS3和其它高级语言中，全局变量大多使用static属性实现，import关键字能通过屏蔽不被import的类来一定程度地降低外部耦合的危害，但这不应成为我们声明和使用大量静态变量的原因。  
【适用】请问你能不能吃下1个鸡蛋？2个呢？3个呢？100个呢？结论是较少、可控制的全局变量使程序能节约开发者的时间，降低代码维护难度，但几百几千个全局变量和对象同样会将程序弄乱，很容易忘记几个或者让几个变量负责了相同的作用。  
【吐槽】作为一种设计模式，不加控制的单例模式（getInstance)是lv5的外部耦合，部分单例模式甚至上升到lv6。目前很多外文博客攻击单例模式（特指无计划、不加管理、随时想到随时创建的单例模式）不无道理。这种单例模式（仍指无计划、不加管理、随时想到随时创建单例的模式）在项目最初会很有用（毕竟lv5<lv7）但无节制地写起来，便会在项目后期出现上文说的缺点而较难维护。  
  
  
  
lv4：控制耦合：A向B传递一条命令，由B解释执行这条命令。到这个级别时，脱耦已经做得比较好了。  
【举例】面向对象的绝大部分的函数、方法调用是lv4控制耦合。  
【优点】绝大部分情况都适用A和B都有一定的自由度，A可以决定命令发与不发、什么时候发，而收到命令怎么处理的权限交给了B，B可以改动自己的代码来针对一个命令做出适当判断。使用Ctrl+点击可以跟踪到B开始处理这条命令的位置，比较方便。  
【缺点】虽然A和B本身对代码有自由度，但命令名称和命令传递参数的格式一旦约定，尽量不要更改，否则修改代价较大。FlashBuilder提供了Ctrl+Alt+R批量更换类名、方法名的办法，使我们使用控制耦合的代价大大降低，我们可以更放心地使用控制耦合。  
【适用】大部分情况。不会被更换或较少更换的模块；有明确、稳定协议的前后对接  
【吐槽】在面向对象中，最普通的方法调用及getter/setter满足控制耦合的定义，而接口、抽象基类属于lv3的标记耦合。lv4控制耦合和lv3标记耦合、lv2数据耦合的区别是，在lv4控制耦合下，A明确知道自己发送的命令是定向发给B的且B会执行，甚至A了解B会怎么响应自己；而lv3标记耦合下A只管按一定格式输出数据，lv2数据耦合下A不按格式输出数据，而根本不关心哪个模块会收这个数据或者这个数据之后会被怎么处理。  
  
  
lv3：标记耦合：A向B传递了一个格式固定的数据结构  
【举例】前台向后台发的URLRequest请求要求返回值也属于此类；在一个即时通讯软件的Socket链接下，A和B已经连接上了。A和B约定，A向B发的数据格式是利用某算法加密后的格式，一次发100字节。B了解这个加密算法并自己有解密的办法。这样相当于前后台约定了固定的数据结构。  
【优点】需要模块脱耦时，这应是最被推荐的脱耦等级，耦合度相当低，A和B的逻辑非常自由、不依赖。A的输出是B的输入。A输出数据时候完全不必想谁收到了，也不必关心这个数据是否会被处理，只管做好自己的工作。  
【缺点】发出一条信息不知道信息发给了谁，Ctrl+点击完全没用，在多个模块可能处理同一消息时，哪怕查看调用层次结构，甚至全工作空间搜索也无法确定这个信息到底被谁收了、怎么处理了。一定要确定消息如何发送时，需要在运行时卡断点而难以在编译时精确定位。另外每个模块发出的信息不一定被处理，可能有些信息成为谁也不处理但客观存在的冗余信息，影响效率。  
【适用】被传递的数据结构非常确定和稳定，不易更改。模块间功能独立，有些模块经常被拆卸、替换。特别适用于网页游戏的一些功能模块，比如一款页游经常3天出一个新的活动，每出一个新的活动时老活动会被撤下，这种情况适用于lv3标记耦合或者lv2数据耦合。  
【吐槽】这里要吐槽的很多  
1，在上文说出“发出一条信息不知道信息发给了谁”时，应该已有聪明的读者发现pureMVC在用lv3标记耦合。在pureMVC中约定使用固定的数据结构Notification来发送信息，Notification（INotification)基本不会被更改。使用pureMVC是很强的lv3脱耦，A和B几乎不相干，很容易被拆卸、替换，修改单一功能模块时也无需考虑其它模块而感受不到压力。pureMVC还有部分脱耦是lv1最强的非直接耦合，详见下文。  
2，Flash自带的dispatchEvent-addEventListener机制是lv3标记耦合。同样有管发不管谁收的优点和问题。  
3，接口/抽象基类编程是标记耦合，接口或抽象基类约定了其它模块能调用自己哪些功能以及参数表。脱耦的代价，我们Ctrl+点击跟踪一个方法时会跟踪到接口而不是具体类。不脑补过程的话，只有运行时卡断点才能知道调用的是哪个具体类的对象。  
4，一个类向另一个类发的信息是某静态常量时是标记耦合，如在A中有如下代码:  B.handleNotice(Enum.SOME\_THING)，B.handleNotice中有多个switch分支，根据参数作不同处理。这种情况下，Ctrl+点击无法直接找到B真正处理Enum.SOME\_THING的部分。  
  
  
  
lv2：数据耦合：A向B传递了一些基本、通用类型的数据  
因lv1非直接耦合已经不太能被称作耦合了，lv2数据耦合是实际意义上的最低程度的耦合。  
和lv3标记耦合的区别是这里发的是基本类型数据，比如int、String，或者约定俗成的PNG、XML  
注意：这个等级是糟糕的脱耦，脱到这里会导致程序松散，维护性下降  
【举例】  
1，A是个JSON格式化工具，加载一个XML格式化成JSON后发给B，B是一个纯文本显示工具，它不关心自己收到的是HTML、XML或者别的格式文本，只是收到数据后显示出来。  
2，this["abc"]["gotoAndStop"](100)，这里将abc和gotoAndStop写成了字符串，绕过编译器检查  
3，A向B传递了一个动态类的对象，比如mc:MovieClip，b调用了mc上的动态添加属性，比如mc.target  
【优点】同lv3标记耦合，并在lv3的基础上，因为没有约定的数据结构，所以不用担心数据结构更改对通信双方造成的影响  
【缺点】同lv3标记耦合，使用跳过编辑器检查技巧或者传动态类对象时脱耦最大，难以维护。在实际运行程序之前，我们并不知道this有没有一个叫abc的属性，也不知道mc有没有一个target属性，更不能直接享用Ctrl+Alt+R的快速替换方法名的一次性替换，而必须通过查找字符串，字符处有大量重名时候。。哭吧。。。  
【适用】同lv3标记耦合  
【吐槽】lv2数据耦合毕竟比lv3标记耦合低了一级，A模块如果向B模块发了一个字符串"myEventA"，那么这是数据耦合，但一旦发的是MyEvent.MY\_EVENT\_A，那么会上升到标记耦合。后者依赖于一个常量MY\_EVENT\_A，这个常量名或值发生变化时则所有依赖其的部分会有修改。MVC携带的数据都是弱类型、基本数据类型的话，虽然约定使用notification传递，我们可以不深究原理地将其看做数据耦合。  
  
  
  
lv1：非直接耦合：A和B不直接发数据  
【举例】A发了数据给C，C处理后发数据给B。这种情况下A对C、B对C有耦合，而AB之间谈不上什么耦合  
【优点】A和B相互独立程度最高，在lv2数据耦合时，还需要A的输出作为B的输入；而这种情况下，A有没有输出、B是否需要输入都不再是被关心的问题  
【缺点】A访问到B需要经过至少一道中间工序，A要发给B的消息在中间的若干道工序上如何变化跟踪程序多，需要多次大于lv1的耦合的跟踪。  
此外，在上面的举例中，如果程序员有目的地让A向B发消息，需要程序员完全了解C会做出什么中间功能，脑补量大。  
【适用】完全不相干的功能，或者某些非常不想扯上关系的模块。譬如pureMVC就强制视图到数据是非直接耦合，详见下文吐槽部分。  
【吐槽】  
pureMVC中Mediator到Proxy不能直接发消息而必须经过Command，这是非直接耦合  
pureMVC之所以设置了Command就是不让显示部分直接能访问到数据，否则Command干脆做成Proxy的方法就好了  
非直接耦合意味着一次发送消息至少经过一道工序才能真正到达目的地，对pureMVC来讲工序更多，项目尽可能多地使用pureMVC来降低耦合纯属自找麻烦。随便处理一点小事也要从显示Comp发到Mediator，从Mediator发给Command，Command才有权查Proxy，Proxy才能访问具体的数据对象，访问到之后Proxy还要发信息才能返回给Mediator，Mediator最后显示到Comp，笔者认为这种上茅厕也要先背一遍毛主席语录的行为不是先进，而是迂腐。只在恰当的时候使用MVC，而不要强制所有功能都MVC  
  
  
  
  
**四、怎样设计才能使耦合程度适当**  
7种耦合是从理论上的分析，下面将从实践的角度上，说一说设计程序的快速方法  
假设类的功能单一，风格良好，以类作为最小模块，  
从Flash最容易实现的脱耦办法来说，我们最常用的其实一共只有5种:  
  
1，A类直接访问B类的public属性，这是lv7的内容耦合。  
2，造一个全局变量或者单例模式，所有访问这个全局变量的类之间是lv5的外部耦合。  
3，直接调用目标类的方法或者直接引用目标类对象，这是lv4的控制耦合。  
4，使用接口或基类引用目标类对象，这是lv3的标记耦合。addEventListener之流也是lv3标记耦合  
5，使用绕过编译器检查技巧，或者直接传字符串、动态类对象实现的耦合是lv2数据耦合  
  
  
笔者兹提供一些实用的设计技巧：  
1，使用getter/setter的目的是将高级的耦合降低到lv4控制耦合。  
  
2，除了特殊说明的情况外，使用lv4控制耦合或lv3标记耦合能应付绝大部分情况。  
将lv4当成“可接受的耦合”(直接调用方法,getter/setter但不操作public属性，不创造无理的继承)  
将lv3当成“脱耦”(接口，addEventListener或写一套Notice机制)  
这样就不必用7个等级逐个比对了。  
  
3，认为目标模块可能被替换时，  
--使用接口或基类编程，目标模块可以被替换成实现同接口的模块或者同基类的派生类  
--使用addEventListener相当于EventDispatcher的抽象基类编程，模块可以被替换成其它能收发同种事件的模块。  
--使用pureMVC的senNotification也相当于接口编程，不过，能收发Notification的对象太多了，每个能收发Notification的对象都能被替换成其它能收Notification的对象。  
笔者接触过项目中，继承同一个基类的派生类最多只有20余个，而一个项目中能收Notification的Command和Mediator少说有几百个，虽然在同一个耦合等级，但不是一个数量级，因此脱耦程度看似更高，问题也更加明显。  
对可能被替换掉的模块，使用基类引用编程维护最快，但可换的模块限定为基类的派生类；使用接口中庸；使用addEventListener或pureMVC编程和维护最慢但可换的模块更自由  
  
4，极限效率优化时使用lv7内容耦合的代码执行效率最高，模块之间访问的速度高出lv4控制耦合的4倍左右，某些情况下高出lv3标记耦合的10倍不止；另外，AB密切绑定，一个创建另一个也创建、一个销毁另一个也销毁时，直接访问public没有问题。除此情况以外不要随意将属性设置为public。  
  
5，单例模式过于万能，也是高耦合写法，依赖import关键字而大量创建单例并不明智。  
使用单例模式时尽量使用树形结构来管理，而不要每个单例类都使用static的getInstance方法得到。单例大多是一些“xx管理者”，在使用它们前先创建一个“管理者的管理者”（类似pureMVC的Facade），从“管理者的管理者”获得“管理者”，使不是谁都能访问到单例，以及自己能清楚一共有哪些单例、这些单例都在什么状态等。  
另一种减少单例模式副作用的办法时，约定一些单例“只读不写”“多写一读”或者“一写多读”  
  
6，如果你的程序中出现了addEventListener("click",xxxx)，将其改成addEventListener(MouseEvent.CLICK,xxxx)。将裸露的数值或字符串写成常量，可以将耦合程度从最低的lv2数据耦合提高到lv3标记耦合，避免耦合过于松散反倒难以维护。PS:有人问这两种写法貌似没什么区别，笔者在这里举个例子。同样使用FlashBuilder，要找出项目中所有监听CLICK的监听器，那么对"click"需要使用Ctrl+H的全局文本搜索，会将一些不是监听器的"click"字符串也搜出来。而使用MouseEvent.CLICK，搜索时使用Ctrl+Alt+H，能精确找出应用这个常量的位置  
  
7，需要传动态类对象时，改为使用ValueObject(部分语言或框架称其为Entity)可以使耦合从lv2提升到lv3，避免耦合过于松散难以维护  
譬如A类中有如下代码  
var unitParam:Object = {atk:150,def:220,xxx:30,mag:55};  
b.setData(unitParam);  
编程到后期即易忘记unitParam中到底有哪些数据。  
可将上述写法改写，定义一个新类UnitParam，该类有4个public属性，分别对应上面4个键  
之后将这个对象发给b  
这样可以通过UnitParam类的代码提示来确定其一共有几个参数要传递。