对C#开发两个基本原则的深入讨论 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/245/2021_2022__E5_AF_B9C_23_E5_BC_80_E5_8F_c97_245957.htm 使用属性，避免将数据成员直接暴露给外界 学习研究.NET的早期，经常碰到一些学习C#/.NET的朋友问，要属性这种华而不实的东西做什么？后来做项目时也时常接到team里的人的抱怨反馈，为什么不直接放一个public字段？如： class Card { public string Name. } 而要做一个private字段 public属性 class Card { private string name. public string Name { get { return this.name.} set { this.name=value.} } } 我记得在早期的一个项目里，team中的一个朋友甚至厌烦了写private字段 public属性，尤其是碰到一大堆臃肿的data object class的时候，索性自己写了一个小工具，来提供一个类的字段名和类型，然后自动为该类生成相应的private字段 public属性。 我在编程的时候是个彻底的实用主义者，用稍微高雅一点的话说叫“ 不喜欢过度的设计 ”。如果真的像上面那样写Card，而且在将来没有什么改变的需求，我也不喜欢像上面第2段程序那样把事情故意搞得复杂。但如果从component的角度来讲，总有一些class是要供外部长久地使用，也潜在地在将来有被改变的需求。这时候，提供属性就很有必要了。 这就是这个Item试图要归纳的使用属性的理由： 1.可以对赋值做校验、或者额外的处理 2.可以做线程同步 3.可以使用虚属性、或者抽象属性 4.可以将属性置于interface中 5.可以提供get-only或者set-only版本，甚至可以给读、写以不同的访问权限（C# 2.0支持） 个人感觉3、4条是属性最大的优点，可以填补没有“ 虚字段 ”或“ 抽象字段 ”

的缺憾，在设计组件的时候非常有用，也体现了C#这样的component-oriented语言的精神内涵。 但如果没有上述理由，而且日后对程序做大的改动可能性比较小时，我想也大可不必非要把每个public字段都要变成属性。比如在设计一些轻型的struct，用于互操作的时候，直接使用public字段没什么不好。所以，感觉本条目Bill Wagner先生使用“ Always Use Properties Instead of Accessible Data Members ”显得太过强硬。其实，这里的讨论也表明阅读《Effective C#》一书时需要注意的地方，即Effective原则并不是放之四海而皆准的。不同的项目（组件化、复用程度较高的项目？还是“ 一次编写、N年都run ”的项目），不同的角色（类库/组件开发人员？还是应用程序开发人员？），有着不同的Effective准则。事实上，书中很多Items都是从类库/组件开发人员的角度来考虑的。关于属性的性能问题需要谈一点，如果仅仅是简单地以存取模式来使用属性，在相当程度上是没有性能损失的。因为在JIT编译过程中已经做了inline的处理。不过inline处理还是有一些基本的条件，有些情况下JIT编译器不会inline，比如虚调用，方法的IL代码长度过长（目前CLR的规定是超过32bytes为代码长度过长），有复杂的控制流逻辑，有异常处理等。这些条件都是要么根本不能使用inline（比如虚属性），要么inline的代价太大，容易导致代码的bloat，要么是inline起来很费时间已经丧失了inline的意义，因为.NET的inline机制发生在JIT过程中。使用属性有个别让人感觉不舒服的地方，比如它影响开发人员的开发效率，但对代码运行的效率不产生影响。 明辨值类型和引用类型的使用场合 这个条款讨论的是类型设计时候的tradeoff是将类型设计为结构还是类。Bill Wagner

先生给出了一个原则“ 值类型用于存储数据，引用类型用于定义行为（value types store values and reference types define behavior） ”。 如何判断这个原则的适用性，Bill Wagner也给出了一个方法，那就是首先回答下面几个问题： 1.该类型的主要职责是否用于数据存储？ 2.该类型的公有接口是否都是一些存取属性？ 3.是否确信该类型永远不可能有子类？ 4.是否确信该类型永远不可能具有多态行为？ 如果所有问题的答案都是yes，那么就应该采用值类型。这样的判断确实有很好的理由支撑，但是我个人认为“ 将这4个问题回答为yes ”还不足以构成采用值类型的全部理由。因为在很多项目实践中，我发现值类型带来的性能问题不可小视。值类型带来的性能问题主要有两个： 1.由于值类型实例在栈和托管堆之间的转换而导致的box/unbox，以及由此带来的托管堆上的垃圾。 2.值类型默认情况下采用的是值拷贝语义，如果是比较大的值类型，在传递参数和函数返回值时，同样会带来性能问题。 关于第1条，Bill Wagner在本条款中提到了“ 引用类型会给垃圾收集器带来负担 ”这个表面看似正确的判断。但是由于box/unbox的效应，有些情况下，反倒是值类型给垃圾收集器带来了更多的负担。比如将一些值类型放到一个集合中，然后又频繁地对其进行读写操作。如果碰到这种情况，我想“ 放弃结构而采用类 ”未尝不是一种更好的做法。事实上，将一个用作数据存储的值类型（比如System.Drawing.Point）添加到一个集合（System.Collections.ArrayList）中是一个太常见不过的操作。不过，C# 2.0中新引入的泛型技术对box/unbox的问题有极大的改善。 关于第2条，Scott Meyers先生在Effective C 的第22条“ 尽量使用pass-by-reference（传址

），少用pass-by-value（传值）”中讲的比较清楚。虽然由于C#中的结构类型具有默认的深拷贝语义，没有拷贝构造器的调用。而且结构类型也没有子类，因此在某种程度上来讲不具有多态性，也就没有C 对象传值时可能出现的切割（slicing）效应。但是值拷贝的成本仍然不小。尤其是在这个值类型比较大的情况下，问题就比较严重。实际上，在.NET框架的Design Guidelines for Class Library Developers文档中，在说明什么时候应该使用结构类型的时候，其中提到了一项原则（还有其他一些并行原则）类型实例数据的大小要小于16个字节。该文档主要是从类型的运行效率层面来考虑的，而Bill Wagner先生这里的条款主要是从类型的设计层面来考虑的。 从上述两条讨论来看，我个人倾向于对结构类型采取更为保守的设计策略。而对于类则可以积极大胆地使用。因为“将结构类型不适当地设计为类”带来的不良后果要远远小于“将类不适当地设计为结构类型”所带来的不良后果。就目前的经验来看，我甚至认为只有和非托管互操作打交道的情况才是使用结构类型最充足的理由，其他情况都要“三思而后行”。当然，在C# 2.0中引入泛型技术之后，box/unbox将不再是一个沉重的负担，应付一些非常轻量级的场合，结构类型依然有自己的一席之地。 