这篇文章，将《Effective C# Second Edition》一书中适用于Unity游戏引擎里使用C#的经验之谈进行了提炼，总结成为21条（一开始总结的是22条，后来发现第22条也是.NET的特性，Unity版本的mono并没有实现，所以严格意义上来说是21条）准则，供各位快速地掌握这本书的知识梗概，在Unity中写出更高质量的C#代码。

《Effective C# Second Edition》一书原本有50条原则，但这50条原则是针对C#语言本身以及.NET来写的，我在阅读过程中，发现是有些原则并不适用于Unity中mono版本的C#的使用。于是，在进行读书笔记总结的时候，将不适用的原则略去，同时将适用的原则进行提炼，总结出21条，构成本文的内容。

需要注意，因为是挑出了书中适用的准则，导致准则序号有些跳跃，为了阅读方便，本文对这些序号进行了重新排列。重排后，标题中与书中序号不一样的准则，都在该原则总结的末尾注明了对应的原书序号。

同样地，作为总结式文章，每一条的内容都高度概括，也许理解坡度比较陡，若有读到不太理解的地方，建议大家去阅读原书，英文版和中文版均可，看看原书中提供的各种代码与示例，这样掌握起来就会事半功倍。

本文内容思维导图式总结

以下是本文内容，提高Unity中C#代码质量的22条准则的总结式思维导图：

原则1

尽可能地使用属性而不是可直接访问的数据成员

● 属性(property)一直是C#语言中比较有特点的存在。属性允许将数据成员作为共有接口的一部分暴露出去，同时仍旧提供面向对象环境下所需的封装。属性这个语言元素可以让你像访问数据成员一样使用，但其底层依旧是使用方法实现的。

● 使用属性，可以非常轻松的在get和set代码段中加入检查机制。

需要注意，正因为属性是用方法实现的，所以它拥有方法所拥有的一切语言特性：

1）属性增加多线程的支持是非常方便的。你可以加强 get 和 set 访问器（accessors）的实现来提供数据访问的同步。

2）属性可以被定义为virtual。

3）可以把属性扩展为abstract。

4）可以使用泛型版本的属性类型。

5）属性也可以定义为接口。

6）因为实现实现访问的方法get与set是独立的两个方法，在C# 2.0之后，你可以给它们定义不同的访问权限，来更好的控制类成员的可见性。

7）而为了和多维数组保持一致，我们可以创建多维索引器，在不同的维度上使用相同或不同类型。

无论何时，需要在类型的公有或保护接口中暴露数据，都应该使用属性。如果可以也应该使用索引器来暴露序列或字典。现在多投入一点时间使用属性，换来的是今后维护时的更加游刃有余。

原则2

偏向于使用运行时常量而不是编译时常量

对于常量，C#里有两个不同的版本：运行时常量（readonly）和编译时常量（const）。

应该尽量使用运行时常量，而不是编译器常量。虽然编译器常量略快，但并没有运行时常量那么灵活。应仅仅在那些性能异常敏感，且常量的值在各个版本之间绝对不会变化时，再使用编译时常量。

编译时常量与运行时常量不同之处表现在于他们的访问方式不同，因为Readonly值是运行时解析的：

● 编译时常量（const）的值会被目标代码中的值直接取代。

● 运行时常量（readonly）的值是在运行时进行求值。● 引用运行时生成的IL将引用到readonly变量，而不是变量的值。

这个差别就带来了如下规则：

● 编译时常量（const）仅能用于数值和字符串。

● 运行时常量（readonly）可以为任意类型。运行时常量必须在构造函数或初始化器中初始化，因为在构造函数执行后不能再被修改。你可以让某个readonly值为一个DataTime结构，而不能指定某个const为DataTIme。

● 可以用readonly值保存实例常量，为类的每个实例存放不同的值。而编译时常量就是静态的常量。

● 有时候你需要让某个值在编译时才确定，就最好是使用运行时常量（readonly）。

● 标记版本号的值就应该使用运行时常量，因为它的值会随着每个不同版本的发布而改变。

● const优于readonly的地方仅仅是性能，使用已知的常量值要比访问readonly值略高一点，不过这其中的效率提升，可以说是微乎其微的。

综上，在编译器必须得到确定数值时，一定要使用const。例如特性（attribute）的参数和枚举的定义，还有那些在各个版本发布之间不会变化的值。除此之外的所有情况，都应尽量选择更加灵活的readonly常量。

原则3

推荐使用is 或as操作符而不是强制类型转换

● C#中，is和as操作符的用法概括如下：

is : 检查一个对象是否兼容于其他指定的类型,并返回一个Bool值,永远不会抛出异常。

as：作用与强制类型转换是一样,但是永远不会抛出异常,即如果转换不成功,会返回null。

● 尽可能的使用as操作符，因为相对于强制类型转换来说，as更加安全，也更加高效。

● as在转换失败时会返回null，在转换对象是null时也会返回null，所以使用as进行转换时，只需检查返回的引用是否为null即可。

● as和is操作符都不会执行任何用户自定义的转换，它们仅当运行时类型符合目标类型时才能转换成功，也不会在转换时创建新的对象。

● as运算符对值类型是无效，此时可以使用is，配合强制类型转换进行转换。

● 仅当不能使用as进行转换时，才应该使用is操作符。否则is就是多余的。

原则4

 推荐使用条件属性而不是#if条件编译

● 由于#if/#endif很容易被滥用，使得编写的代码难于理解且更难于调试。C#为此提供了一条件特性(Conditional attribute)。使用条件特性可以将函数拆分出来，让其只有在定义了某些环境变量或设置了某个值之后才能编译并成为类的一部分。Conditional特性最常用的地方就是将一段代码变成调试语句。

● Conditional特性只可应用在整个方法上，另外，任何一个使用Conditional特性的方法都只能返回void类型。不能再方法内的代码块上应用Conditional特性。也不可以在有返回值的方法上应用Conditional特性。但应用了Conditional特性的方法可以接受任意数目的引用类型参数。

● 使用Conditional特性生成的IL要比使用#if/#Eendif时更有效率。同时，将其限制在函数层面上可以更加清晰地将条件性的代码分离出来，以便进一步保证代码的良好结构。

原则5

理解几个等同性判断之间的关系

● C#中可以创建两种类型：值类型和引用类型。如果两个引用类型的变量指向的是同一个对象，它们将被认为是“引用相等”。如果两个值类型的变量类型相同，而且包含同样的内容，它们被认为是“值相等”。这也是等同性判断需要如此多方法的原因。

● 当我们创建自己的类型时（无论是类还是struct），应为类型定义“等同性”的含义。C#提供了4种不同的函数来判断两个对象是否“相等”。

1）public static bool ReferenceEquals (object left, object right);判断两个不同变量的对象标识（object identity）是否相等。无论比较的是引用类型还是值类型，该方法判断的依据都是对象标识，而不是对象内容。

2）public static bool Equals (object left, object right); 用于判断两个变量的运行时类型是否相等。

3）public virtual bool Equals(object right); 用于重载

4）public static bool operator ==(MyClass left, MyClass right); 用于重载

● 不应该覆写Object.referenceEquals（）静态方法和Object.Equals（）静态方法，因为它们已经完美的完成了所需要完成的工作，提供了正确的判断，并且该判断与运行时的具体类型无关。对于值类型，我们应该总是覆写Object.Equals（）实例方法和operatior==( ),以便为其提供效率更高的等同性判断。对于引用类型，仅当你认为相等的含义并非是对象标识相等时，才需要覆写Object.Equals( )实例方法。在覆写Equals( )时也要实现IEquatable<T>。

PS: 此原则对应于《EffectiveC# Second Edition》中原则6。

原则6

了解GetHashCode( )的一些坑

● GetHashCode( )方法在使用时会有不少坑，要谨慎使用。GetHashCode()函数仅会在一个地方用到，即为基于散列(hash)的集合定义键的散列值时，此类集合包括HashSet<T>和Dictionary<K,V>容器等。对引用类型来讲，索然可以正常工作，但是效率很低。对值类型来讲，基类中的实现有时甚至不正确。而且，编写的自己GetHashCode( )也不可能既有效率又正确。

● 在.NET中，每个对象都有一个散列码，其值由System.Object.GetHashCode()决定。

● 实现自己的GetHashCode( )时，要遵循上述三条原则：

1）如果两个对象相等（由operation==定义），那么他们必须生成相同的散列码。否则，这样的散列码将无法用来查找容器中的对象。

2）对于任何一个对象A，A.GetHashCode()必须保持不变。

3）对于所有的输入，散列函数应该在所有整数中按随机分别生成散列码。这样散列容器才能得到足够的效率提升。

PS: 此原则对应于《EffectiveC# Second Edition》中原则7。

原则7

理解短小方法的优势

将C#代码翻译成可执行的机器码需要两个步骤。

C#编译器将生成IL，并放在程序集中。随后，JIT将根据需要逐一为方法（或是一组方法，如果涉及内联）生成机器码。短小的方法让JIT编译器能够更好地平摊编译的代价。短小的方法也更适合内联。

除了短小之外，简化控制流程也很重要。控制分支越少，JIT编译器也会越容易地找到最适合放在寄存器中的变量。

所以，短小方法的优势，并不仅体现在代码的可读性上，还关系到程序运行时的效率。

PS:此原则对应于《EffectiveC# Second Edition》中原则11。

原则8

选择变量初始化而不是赋值语句

成员初始化器是保证类型中成员均被初始化的最简单的方法——无论调用的是哪一个构造函数。初始化器将在所有构造函数执行之前执行。使用这种语法也就保证了你不会再添加的新的构造函数时遗漏掉重要的初始化代码。

综上，若是所有的构造函数都要将某个成员变量初始化成同一个值，那么应该使用初始化器。

PS: 此原则对应于《Effective C# Second Edition》中原则12。

原则9

正确地初始化静态成员变量

● C#提供了有静态初始化器和静态构造函数来专门用于静态成员变量的初始化。

● 静态构造函数是一个特殊的函数，将在其他所有方法执行之前以及变量或属性被第一次访问之前执行。可以用这个函数来初始化静态变量，实现单例模式或执行类可用之前必须进行的任何操作。

● 和实例初始化一样，也可以使用初始化器语法来替代静态的构造函数。若只是需要为某个静态成员分配空间，那么不妨使用初始化器的语法。而若是要更复杂一些的逻辑来初始化静态成员变量，那么可以使用静态构造函数。

● 使用静态构造函数而不是静态初始化器最常见的理由就是处理异常。在使用静态初始化器时，我们无法自己捕获异常。而在静态构造函数中却可以做到。

PS: 此原则对应于《Effective C# Second Edition》中原则13。

原则10

使用构造函数链（减少重复的初始化逻辑）

● 编写构造函数很多时候是个重复性的劳动，如果你发现多个构造函数包含相同的逻辑，可以将这个逻辑提取到一个通用的构造函数中。这样既可以避免代码重复，也可以利用构造函数初始化器来生成更高效的目标代码。

● C#编译器将把构造函数初始化器看做是一种特殊的语法，并移除掉重复的变量初始化器以及重复的基类构造函数调用。这样使得最终的对象可以执行最少的代码来保证初始化的正确性。

● 构造函数初始化器允许一个构造函数去调用另一个构造函数。而C# 4.0添加了对默认参数的支持，这个功能也可以用来减少构造函数中的重复代码。你可以将某个类的所有构造函数统一成一个，并为所有的可选参数指定默认值。其他的几个构造函数调用某个构造函数，并提供不同的参数即可。

PS: 此原则对应于《EffectiveC# Second Edition》中原则14。

原则11

实现标准的销毁模式

● GC可以高效地管理应用程序使用的内存。不过创建和销毁堆上的对象仍旧需要时间。若是在某个方法中创建了太多的引用对象，将会对程序的性能产生严重的影响。

这里有一些规则，可以帮你尽量降低GC的工作量：

1）若某个引用类型（值类型无所谓）的局部变量用于被频繁调用的例程中，那么应该将其提升为成员变量。

2）为常用的类型实例提供静态对象。

3）创建不可变类型的最终值。比如string类的+=操作符会创建一个新的字符串对象并返回，多次使用会产生大量垃圾，不推荐使用。对于简单的字符串操作，推荐使用string.Format。对于复杂的字符串操作，推荐使用StringBuilder类。

PS: 此原则对应于《EffectiveC# Second Edition》中原则16。

原则12

区分值类型和引用类型

● C#中，class对应引用类型，struct对应值类型。

● C#不是C++，不能将所有类型定义成值类型并在需要时对其创建引用。C#也不是[**Java**](http://lib.csdn.net/base/javase)，不像Java中那样所有的东西都是引用类型。你必须在创建时就决定类型的表现行为，这相当重要，因为稍后的更改可能带来很多灾难性的问题。

● 值类型无法实现多态，因此其最佳用途就是存放数据。引用类型支持多态，因此用来定义应用程序的行为。

●  一般情况下，我们习惯用class，随意创建的大都是引用类型，若下面几点都肯定，那么应该创建struct值类型：

1）该类型主要职责在于数据存储吗？

2）该类型的公有接口都是由访问其数据成员的属性定义的吗？

3）你确定该类型绝不会有派生类型吗？

4）你确定该类型永远都不需要多态支持吗？

● 用值类型表示底层存储数据的类型，用引用类型来封装程序的行为。这样，你可以保证类暴露出的数据能以复制的形式安全提供，也能得到基于栈存储和使用内联方式存储带来的内存性能提升，更可以使用标准的面向对象技术来表达应用程序的逻辑。而倘若你对类型未来的用图不确定，那么应该选择引用类型。

PS: 此原则对应于《Effective C# Second Edition》中原则18。

原则13

保证0为值类型的有效状态

在创建自定义枚举值时，请确保0是一个有效的选项。若你定义的是标志(flag)，那么可以将0定义为没有选中任何状态的标志（比如None）。即作为标记使用的枚举值（即添加了Flags特性）应该总是将None设置为0。

PS: 此原则对应于《Effective C# Second Edition》中原则19。

原则14

保证值类型的常量性和原子性

常量性的类型使得我们的代码更加易于维护。不要盲目地为类型中的每一个属性都创建get和set访问器。对于那些目的是存储数据的类型，应该尽可能地保证其常量性和原子性。

PS: 此原则对应于《Effective C# Second Edition》中原则20。

原则15

限制类型的可见性

在保证类型可以完成其工作的前提下。你应该尽可能地给类型分配最小的可见性。也就是，仅仅暴露那些需要暴露的。尽量使用较低可见性的类来实现公有接口。可见性越低，能访问你功能的代码越少，以后可能出现的修改也就越少。

PS: 此原则对应于《Effective C# Second Edition》中原则21。

原则16

通过定义并实现接口替代继承

● 理解抽象基类（abstract class）和接口（interface）的区别：

1）接口是一种契约式的设计方式，一个实现某个接口的类型，必须实现接口中约定的方法。抽象基类则为一组相关的类型提供了一个共同的抽象。也就是说抽象基类描述了对象是什么，而接口描述了对象将如何表现其行为。

2）接口不能包含实现，也不能包含任何具体的数据成员。而抽象基类可以为派生类提供一些具体的实现。

3）基类描述并实现了一组相关类型间共用的行为。接口则定义了一组具有原子性的功能，供其他不相关的具体类型来实现。

● 理解好两者之间的差别，我们便可以创造更富表现力、更能应对变化的设计。使用类层次来定义相关的类型。用接口暴露功能，并让不同的类型实现这些接口。

PS: 此原则对应于《EffectiveC# Second Edition》中原则22。

原则17

理解接口方法和虚方法的区别

第一眼看来，实现接口和覆写虚方法似乎没有什么区别，实际上，实现接口和覆写虚方法之间的差别很大。

1）接口中声明的成员方法默认情况下并非虚方法，所以，派生类不能覆写基类中实现的非虚接口成员。若要覆写的话，将接口方法声明为virtual即可。

2）基类可以为接口中的方法提供默认的实现，随后，派生类也可以声明其实现了该接口，并从基类中继承该实现。

3）实现接口拥有的选择要比创建和覆写虚方法多。我们可以为类层次创建密封（sealed）的实现，虚实现或者抽象的契约。还可以创建密封的实现，并在实现接口的方法中提供虚方法进行调用。

PS: 此原则对应于《EffectiveC# Second Edition》中原则23。

原则18

用委托实现回调

在C#中，回调是用委托来实现的，主要要点如下：

1）委托为我们提供了类型安全的回调定义。虽然大多数常见的委托应用都和事件有关，但这并不是C#委托应用的全部场合。当类之间有通信的需要，并且我们期望一种比接口所提供的更为松散的耦合机制时，委托便是最佳的选择。

2）委托允许我们在运行时配置目标并通知多个客户对象。委托对象中包含一个方法的应用，该方法可以是静态方法，也可以是实例方法。也就是说，使用委托，我们可以和一个或多个在运行时联系起来的客户对象进行通信。

3）由于回调和委托在C#中非常常用，以至于C#特地以lambda表达式的形式为其提供了精简语法。

4）由于一些历史原因，.NET中的委托都是多播委托（multicast delegate）。多播委托调用过程中，每个目标会被依次调用。委托对象本身不会捕捉任何异常。因此，任何目标抛出的异常都会结束委托链的调用。

PS: 此原则对应于《EffectiveC# Second Edition》中原则24。

原则19

用事件模式实现通知

● 事件提供了一种标准的机制来通知监听者，而C#中的事件其实就是观察者模式的一个语法上的快捷实现。

● 事件是一种内建的委托，用来为事件处理函数提供类型安全的方法签名。任意数量的客户对象都可以将自己的处理函数注册到事件上，然后处理这些事件，这些客户对象无需在编译器就给出，事件也不必非要有订阅者才能正常工作。

● 在C#中使用事件可以降低发送者和可能的通知接受者之间的耦合，发送者可以完全独立于接受者进行开发。

PS: 此原则对应于《EffectiveC# Second Edition》中原则25。

原则20

避免返回对内部类对象的引用

● 若将引用类型通过公有接口暴露给外界，那么对象的使用者即可绕过我们定义的方法和属性来更改对象的内部结构，这会导致常见的错误。

● 共有四种不同的策略可以防止类型内部的[**数据结构**](http://lib.csdn.net/base/datastructure)遭到有意或无意的修改：

1）值类型。当客户代码通过属性来访问值类型成员时，实际返回的是值类型的对象副本。

2）常量类型。如System.String。

3）定义接口。将客户对内部数据成员的访问限制在一部分功能中。

4）包装器（wrapper）。提供一个包装器，仅暴露该包装器，从而限制对其中对象的访问。

PS: 此原则对应于《Effective C# Second Edition》中原则26。

原则21

仅用new修饰符处理基类更新

● 使用new操作符修饰类成员可以重新定义继承自基类的非虚成员。

● new修饰符只是用来解决升级基类所造成的基类方法和派生类方法冲突的问题。

● new操作符必须小心使用。若随心所欲的滥用，会造成对象调用方法的二义性。

PS: 此原则对应于《Effective C# Second Edition》中原则33