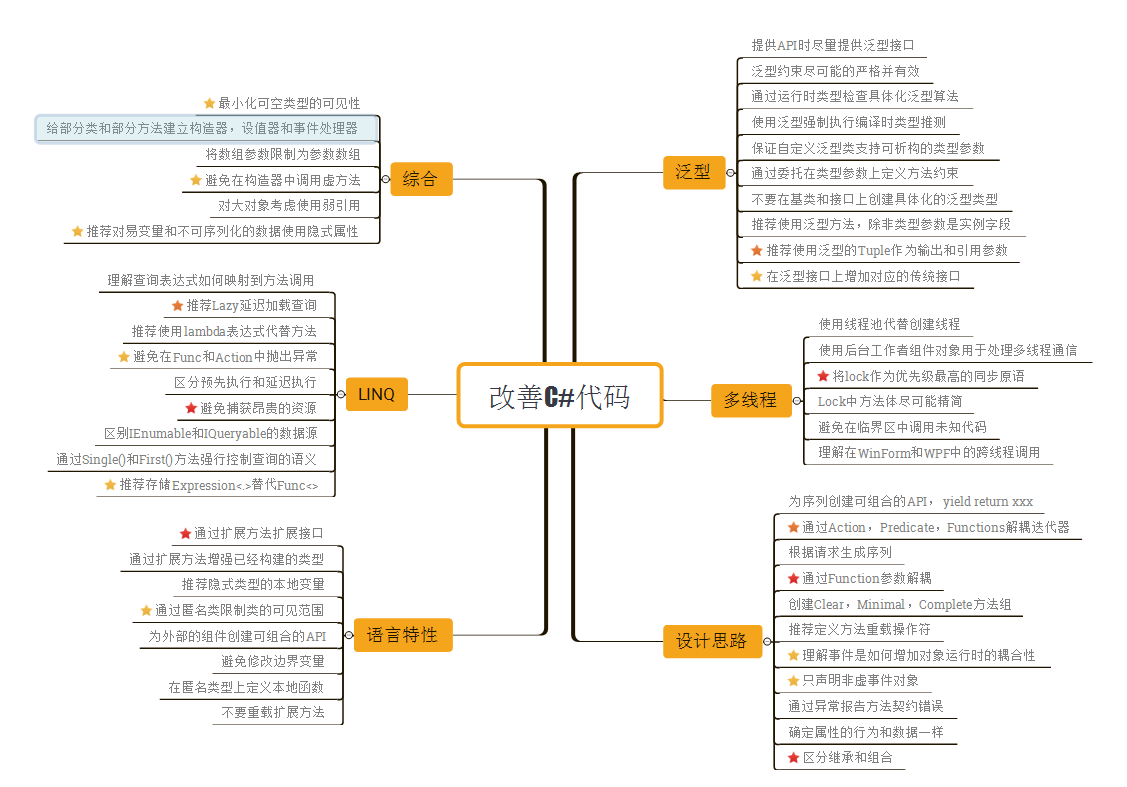
周末空闲，选读了一下一本很不错的C#语言使用的书，特此记载下便于对项目代码进行重构和优化时查看。

Standing On Shoulders of Giants，附上思维导图。



|  |  |
| --- | --- |
| 名称 | 内容和示例 |
| 提供API时尽量提供泛型接口 | Public interface IComparable<T>{ int CompareTo(T other) } |
| 泛型约束尽可能的严格并有效 | Public delegate T FactoryFunc<T>();  Public static T Factory<T>( FactoryFunc<T> newT) where T:new()  { T t = newt();} |
| 通过运行时类型检查具体化泛型算法 | 比如根据不同的集合类型优化相应算法 |
| 使用泛型强制执行编译时类型推测 | Public static T ReadFromStream(XmlReader inputStream)  { return (T)factory.Deserialize(inputStream) } |
| 保证自定义泛型类支持可析构的类型参数 | Public sealed class EngineDriver<T>:IDisposable where T:Engine, new()  {  public void Dispose(){ var resource = driver as IDisposable; if(resource != null) resource.Dispose(); }  } |
| 通过委托在类型参数上定义方法约束 | Public static T Add<T>(T left, T right, Func<T,T,T> addFunc){ return addFunc(right, left); } |
| 不要在基类和接口上创建具体化的泛型类型 | 尽可能使的基类和接口的适用范围更加的广阔 |
| 推荐使用泛型方法，除非类型参数是实例字段 | Public static T Max<T>(T left, T right)  { return Comparer<T>.Default.Compare(left, right) < 0 ? right : left } |
| 推荐使用泛型的Tuple作为输出和引用参数 | 当设置方法的返回值，或者在需要使用ref参数的情形时，使用Tuple<>元组可以使代码更清晰，当然如果参数比较复杂，还是选择建立对应的DTO类型为宜 |
| 在泛型接口上增加对应的传统接口 | 这个在大家基础架构时非常重要，每个方法均提供泛型版本和object版本，使得代码有很强的兼容性。  Public static bool CheckEquality(object left, object right)  { return left.Equals(right); }  Public static bool CheckEquality<T>(T left, T right) where T:IEquatable<T>  { return left.Equals(right); } |

|  |  |
| --- | --- |
| 名称 | 内容和示例 |
| 使用线程池代替创建线程 | 经过微软的官方测试，由自己调度线程和使用线程池，在每10万个计算消耗的平均时长比较中，前者所消耗时长为后者三倍，因而选用线程池作为默认多线程处理机制是合理的选择  Private static double ThreadPoolThreads(int numThreads)  {  var start = new Stopwatch();  Using(var e = new AutoResetEvent(false)){  int workerThreads = numThreads;  start.Start();//**watch.ElapsedMilliseconds, watch.Restart(), watch.Stop();**  for(var I = 0; I < numThreads; thread++)  ThreadPool.QueueUserWorkItem( (x)=>{  // to do  If(Interlocked.Decrement(ref workThreads) == 0)  { e.Set(); }  });  }} |
| 使用后台工作者组件对象用于处理多线程通信 | 现在已经不再使用后台Worker，而推荐使用Task任务模型替代它，其逻辑为 |
| 将lock作为优先级最高的同步原语 | 使用lock相当于使用了Monitor.Enter和Exit，不过要方便很多，使用的是临界区的概念。Public int TotalNum  {  get{ lock(syncObj) return total; }  set{ lock(syncObj) total++;}  } |
| Lock中方法体尽可能精简 | 在使用lock时，一定不要使用lock(this)和lock(typeof(MyType))的形式，这会造成很多的问题，必须保证锁的对象不是公开无法被外部使用的，常见的对方法加锁的形式有：  1．使用特性，[MethodImpl(MethodImplOptions.Synchronized)]  2.使用私有变量作为锁变量 private object syncHandler = new object(); 此外还有一种复杂点的形式如下。  Private object syncHandle;  Private object GetSyncHandle(){  InterLocked.CompareExchange(ref syncHandle, new object(), null);  } |
| 避免在临界区中调用未知代码 | 比如不要在临界区中使用事件，因为事件的处理方法由调用方注册，是未知的，会造成相关的问题，一定要保证临界区中方法的确定性 |
| 理解在WinForm和WPF中的跨线程调用 | 做过WinForm编程的亲，一定遇到过一个InvalidOperationException，内容为跨线程操作非法，访问Control的线程不是创建线程，这其实是Winform、WPF等框架对UI的保护，避免多个不同线程修改UI值的情况。这种情况主要有一下三种方式来处理，最推荐的解决方案为第二种。   1. 在Form的构造方法中加入 Control.CheckForIllegalCrossThreadCalls =false; (不推荐) 2. Private void UpdateControl(string msg)   {  Action<string> act = (x)=>{this.controlA.Text = x; }  if(controlA.InvokeRequired)  { this.controlA.Invoke(act, msg); }  else{ act(); }  }   1. 使用BackgroundWorker组件(elide) |

|  |  |
| --- | --- |
| 名称 | 内容和示例 |
| 为序列创建可组合的API， yield return xxx | Public static IEnumerable<int> Square(IEnumerable<int> nums)  { foreach(var num in nums) yield return num \* num; } |
| 通过Action，Predicate，Functions解耦迭代器 | Public static IEnumerable<T> Filter<T>(IEnumerable<T> sequence, Predicate<T> filterFunc)  { if(filterFunc(int)) yield return item; } |
| 根据请求生成序列 | [IEnumerable<int>].TakeWhile(num => num < 5); |
| 通过Function参数解耦 | Public static T Sum<T>(IEnumerable<T> sequence, T total, Func<T,T,T> accumulator)  { foreach(T item in sequence){ total = accumulator(total, item); return total; } } |
| 创建清晰，最小化，完整的方法组 | 即在提供方法时，尽可能的保证完备性（支持主要的类型） |
| 推荐定义方法重载操作符 | 还记得在学习C++时，很推荐重载操作符，不过在面向对象语言的今天，使用可读性更强的方法更合理 |
| 理解事件是如何增加对象运行时的耦合性 | public event EventHandler<WorkerEventArgs> OnProgress;  public void DoLotsOfStuff()  {  for (var i = 0; i < 100; i++)  {  SomeWork();  var args = new WorkerEventArgs();  args.Percent = i;  //关于这个=，我总是不算特别明白，不过记得是线程安全的代码  //可以理解为，使用这个，其他调用这个事件的对象就不会被锁定  var progHandler = OnProgress;  if (progHandler != null)  {  //注意这里的this  progHandler(this, args);  }  if (args.Cancel)  return;  }  }  这里想补充的是，event属于编译时解耦，你可以看到，该事件的订阅者都没有入侵事件所属的发布者（发布者-订阅者默认），但实际上，在运行时，所有的订阅者其实是和事件紧密关联在一起的，订阅者们修改共享数据的操作存在很大的不确定性。简而言之，事件是编译时解耦，运行时耦合的。 |
| 只声明非虚事件对象 | 在.NET中，事件提供了类似属性的简易语法，通过add,remove方法添加相关事件处理程序，其实event就是delegate的包装器，这个特殊的委托便于**应用事件处理模型**，同时提供线程安全性。由于事件的运行时耦合性，如果使用虚事件容易造成未知的错误，  private EventHandler<WorkerEventArgs> progressEvent;  public event EventHandler<WorkerEventArgs> OnProgress  {  [MethodImpl(MethodImplOptions.Synchronized)]  add { progressEvent += value; }  [MethodImpl(MethodImplOptions.Synchronized)]  remove { progressEvent -= value; }  } |
| 通过异常报告方法契约错误 | 当出现业务异常流程时，推荐抛出异常而不是使用TryXXX组合的方式，因为这样代码更加简单易懂。当然在与业务无关的，如简单数据转换的场景下，使用TryXXX是很好的选择 |
| 确定属性的行为和数据一样 | 让属性尽可能的简单，不要将复杂逻辑放在属性，如果需要可以通过提供相应方法的方式，使得代码更加通俗易懂，且使得调用人坚信属性的调用不会造成任何的性能影响 |
| 区分继承和组合 | 在适当的场景下，用组合代替继承是常见的代码设计模式，这样可以减少类的污染，在选用策略模式的场景下，组合使用的非常的多，常见的形式如下：  public interface IContract{ void SampleImplMethod(); }  public class MyInnerClass:IContract{ public void SampleImplMethod (){ //elided }}  public class MyOuterClass:IContract{  private IContract impl = new MyInnerClass();  public void SampleImplMethod (){  impl.SampleImplMethod();  }} |

|  |  |
| --- | --- |
| 名称 | 内容和示例 |
| 通过扩展方法扩展接口 | Public static bool LessThan<T>(this T left, T right) where T : IComparable<T>  { return left.CompareTo(right) < 0; } |
| 通过扩展方法增强已经构建的类型 | 这部分很容易理解，比如你使用系统提供的相关类，无法修改源码（虽然已开源），这时为了代码的便捷性和可读性，使用扩展方法增强该类变得非常有效 |
| 推荐隐式类型的本地变量 | 简单方便 |
| 通过匿名类限制类的可见范围 | 使得代码的封装性更好，更加健壮 |
| 为外部的组件创建可组合的API | 要求提供的API具有更好的健壮性，功能相对完整并独立，复用性更强，例如尽量不要使用可空类型作为接口参数等 |
| 避免修改绑定的变量 | 这部分内容涉及闭包，通过以下的例子可以很容易的理解  public void Test()  {  int index = 0;  Func<IEnumerable<int>> sequence = () => Generate(30, () => index ++);  index = 20;  foreach (var item in sequence())  {  Console.WriteLine(item);  }  }  private IEnumerable<int> Generate(int num, Func<int> act)  {  for (; num > 0; num--)  {  yield return act();  }  } |
| 在匿名类型上定义本地函数 | public void Test01()  {  var randomNumbers = new Random();  var sequence = (from x in Generate(100, () => randomNumbers.NextDouble() \* 100)  let y = randomNumbers.NextDouble() \* 100  select new { x, y }).TakeWhile(point => point.x < 75);  foreach (var item in sequence)  {  Console.WriteLine(item);  }  } |
| 不要重载扩展方法 | 由于个人创建扩展方法的普遍性和完备性不强，重载此类方法容易降低程序的健壮性 |

|  |  |
| --- | --- |
| 名称 | 内容和示例 |
| 理解查询表达式如何映射到方法调用 |  |
| 推荐Lazy延迟加载查询 |  |
| 推荐使用lambda表达式代替方法 |  |
| 避免在Func和Action中抛出异常 |  |
| 区分预先执行和延迟执行 |  |
| 避免捕获昂贵的资源 |  |
| 区别IEnumable和IQueryable的数据源 |  |
| 通过Single()和First()方法强行控制查询的语义 |  |
| 推荐存储Expression<.>替代Func<> |  |

|  |  |
| --- | --- |
| 名称 | 内容和示例 |
| 最小化可空类型的可见性 |  |
| 给部分类和部分方法建立构造器，设值器和事件处理器 |  |
| 将数组参数限制为参数数组 |  |
| 避免在构造器中调用虚方法 |  |
| 对大对象考虑使用弱引用 |  |
| 推荐对易变量和不可序列化的数据使用隐式属性 |  |

参考文献：

1. [美]Bill, Wagner. More Effective C#[M]. 北京:人民邮电出版社, 2009.