.NET中的泛型集合

在这里主要介绍常见的泛型集合，很多时候其并发时的线程安全性常常令我们担忧。因而简述下.NET并发时线程安全特性，其详情请见MSDN。

* 普通集合都不支持多重并发写操作
* 部分支持单线程写和并发读操作
* 同时.NET4添加了大量并发集合

首先介绍常见的泛型集合接口，其大部分都位于System.Collection.Generic命名空间。

* IEnumerable<T>，其可以获取一个IEnumerator<T>迭代器，如果从数据库的角度来看，前者是表，后者是游标，同时这两个接口是唯一具有可变性的集合接口。
* ICollection<T>，它扩展了IEnumerable<T>，添加了Count和IsReadOnly属性，Add和Remove等操作方法，Contains等判定函数，所有的标准泛型集合都实现了该接口。
* IList<T>，提供定位功能，包括一个索引器、Insert和RemoveAt，我们通常认为可以通过索引对该泛型集合进行随机访问。、
* IDictionary<TKey, TValue>，表示键值对集合，扩展了ICollection<KeyValuePair<TKey, TValue>>，取值可以用TryXXX方式。
* ISet<T>表示唯一值集，包含大量集合操作：交、并、补。

接下来介绍具体的集合泛型集合类型，在实际中需要根据具体场景选择最适合的集合类型。

* List<T>，其是列表的默认选择，内含一个数组，并且提供列表的逻辑大小Count和后台数组的大小Capacity，当数组满了时，会进行扩容。由于是连续型的数据结构，其添加删除操作的成本较高，提供二分查找，查找效率高。同时，其Sort操作会修改原始列表的内容，与OrderBy不同，并且Sort是不稳定的，会出现相等元素顺序不同的情况。
* 数组，最基础的集合，均派生自System.Array，包括一维数组T[10]，二维数组T[10, 20]等，通过Array类的静态方法进行ConvertAll、FindAll和BinarySearch等操作。
* Colletion<T>，位于System.Colletion.ObjectModel命名空间，为BindingList<T>和ObservableCollection<T>等扩展类型提供基类。与双向绑定相关的集合类型，注意它们只会在包装器发生变化发出通知，而基础列表改变时不会引发任何事件。
* ReadOnlyCollection<T>和ReadOnlyObservableCollection<T>，其也类似于包装器，后者实现了INotifyCollectionChanged, INotifyPropertyChanged两个接口。
* Dictionary<TKey, TValue>，使用散列表，查找性能的优劣取决于散列函数的优劣，默认使用Equals和GetHashCode，可以通过制定IEqualityComparer<TKey>作为参数。
* SortList<TKey, TValue>和SortedDictionary<TKey, TValue>，两者都是字典类，前者内部维护一个排序的数组，添加删除操作的事件复杂度为O(n)，后者内部维护一个红黑树，添加删除操作事件复杂度为O(log n)，但会消耗更多的堆内存，使用IComparer<TKey>作比较。
* HashSet<T>，是不含值的Dictionary<,>，具有相同性能特性，并且所维护顺序一般与添加顺序无关。
* SortedSet<T>，是没有值得SortedDictionary<,>，维护一个红黑树，添加删除和检查操作的事件复杂度为O(log n)。提供GetViewBetween方法返回介于原始集上下限之间的另一个SortedSet<T>，注意这是一个动态的视图，会随着原始集的改变而改变。尽管看起来很方便，但需要注意的是“天下没有免费的午餐”，为保持内部一致性，操作的代价更大。
* Queue<T>，构建一个环形缓冲区，实际维护一个基础数组，包含两个索引，分别记住入队和出队的位置(Slot)，如果入队指针追上出队指针，则进行扩容。提供Enqueue、Dequeue、Peek等方法进行入队、出队、查看操作。
* Stack<T>，其实现更简单，可以看做是一个提供Push、Pop、Peek操作的List<T>。

最后介绍并行集合，也就是线程安全的集合。（注意所有的并发类型都未实现IList<T>接口）

* IProducerConsumerCollection<T>和BlockingCollection<T>，前者是生产者/消费者模型中数据存储的抽象，后者是其包装类，使用ConcurrentQueue<T>作为后台存储，提供ToArray方法获得集合当前状态快照，TryXXX方法允许有效的失败模式减少对锁的需求。（例如，当队列中只有一个项时，两个线程同时判断它是否有项，并且都返回true，这是一个线程执行了出队操作，而另外一个线程在执行出队操作时，将抛出异常，因而需要对验证队列是否有项操作和有项就出队操作作为一个整体，需要添加锁）
* ConcurrentBag<T>，ConcurrentQueue<T>，ConcurrentStack<T>，它们是对IProducerConsumerCollection<T>的实现，其GetEnumerator()方法返回集合快照，迭代时可以改变集合，但该改变不会反应到迭代器中。
* ConcurrentDictionary<TKey, TValue>， 实现了IDictionary<TKey, TValue>接口。支持并发的读写和线程安全的迭代，但不同是，其在迭代过程中对字典的改变不能确定是否反应到迭代器上。

小节：在日常工作中，当遇到需要并发操作非集合类型的全局变量时，需要使用锁来处理；而当是集合类型时，就需要使用对应的并行集合类来处理，其能很好的TPL协作在一起。尤其在使用非线程安全的字典类进行并发操作时，有时会出现死循环等情形，尤其需要注意。

参考文献

1. Jon, Skeet. 深入理解C#(第3版)[M]. 北京:人民邮电出版社, 2014. 469-483