Параллельная работа с динамическими структурами

- Динамические коллекции List<T>, Stack<T>, Queue<T>, Dictionary<TKey, TValue> не являются потокобезопасными
- Изменения, вносимые в нескольких потоках одновременно, могут привести к проблеме «data race»:
  - потеря элементов;
  - выход за границы и обращение к несуществующим элементам;
- Для обеспечения взаимно-согласованного доступа можно использовать:
  - обычные коллекции + средства синхронизации (lock, Monitor, ..)
  - конкурентные коллекции

System.Collections.Concurrent;

ConcurrentQueue<T>	FIFO-очередь
ConcurrentStack<T>	LIFO-стэк
ConcurrentBag<T>	Неупорядоченная коллекция
ConcurrentDictionary<T>	Словарь
BlockingCollection<T>	Ограниченная коллекция

Конкурентные коллекции vs. List<T>

- Не требуется применения дополнительных средств синхронизации
- Используют *неблокирующие алгоритмы* для синхронизации, поэтому являются более эффективными, чем применение средств синхронизации и «обычных» коллекций
- Конкурентные коллекции могут быть созданы на базе существующих коллекций, реализующих перечислимый интерфейс

```
var seqQ = new Queue<int>();
seqQ.Enqueue(..);
..
var concQ = new ConcurrentQueue(seqQ);
```

- «Обычные» коллекции не позволяют вносить изменения в коллекцию при осуществлении перечисления

```
var bag = new ConcurrentBag<int>();
for(int i=0; i<10; i++)
        bag.Add(i);
foreach(int k in bag)
{
        bag.Add(-1);
        Console.Write(k);
}
```

Конкурентные коллекции

- Все конкурентные коллекции реализуют интерфейс **IProducerConsumerCollection<T>** с основными методами:

  bool TryAdd(T item);

  bool TryTake(out T item);

Объект	Добавить	Извлечь
ConcurrentStack	Push	TryPop
ConcurrentQueue	Enqueue	TryDequeue, TryPeek
ConcurrentBag	Add	TryTake
BlockingCollection	Add, TryAdd	Take, TryTake
Dictionary	TryAdd, AddOrUpdate, GetOrAdd	TryRemove

ConcurrentBag

- Для каждого потока, использующего объект ConcurrentBag, создается локальный буфер
- При добавлении элементы помещаются в локальный буфер текущего потока; нет необходимости в синхронизации
- При извлечении элементов в первую очередь обрабатывается локальная очередь текущего потока, а затем локальные очереди других потоков

Очередь с блокировкой

- Объект **BlockingCollection<T>** реализует модель «производитель-потребитель» со встроенными механизмами ожидания и сигнализации
- Все операции осуществляются *потокобезопасно*
- Метод **Take** вызывается потребителем и приводит к блокировке в случае отсутствия элементов
- Метод **Add** вызывается производителем и приводит к блокировке в случае наполненности коллекции
- Поддерживается возможность «завершения» добавления с помощью метода **CompleteAdding**

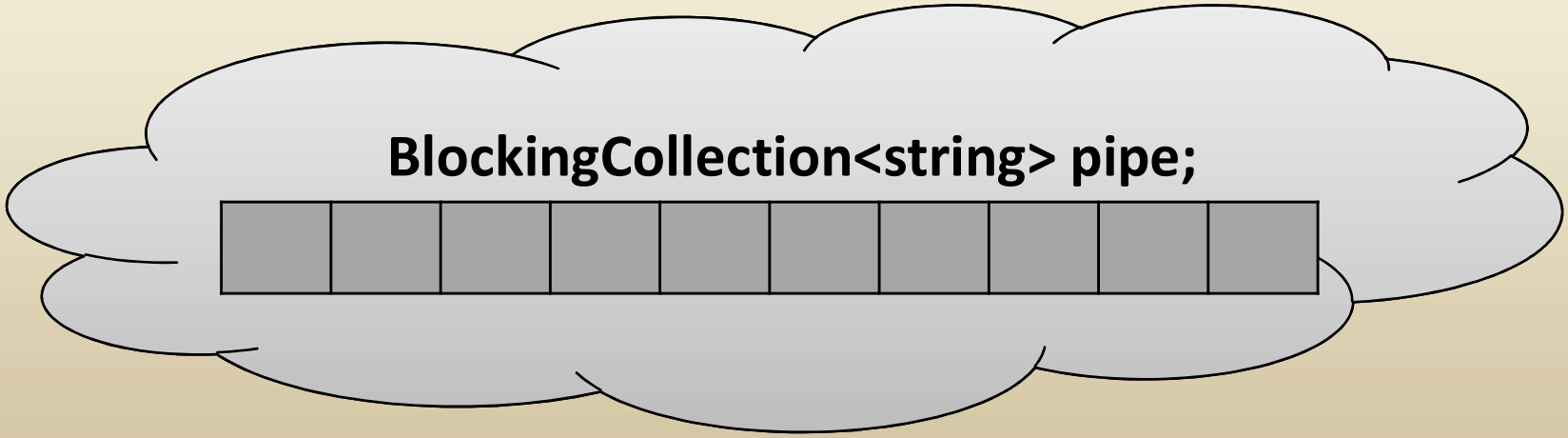
Сценарий «producer-consumer»

```
// Producer #1
w
```

```
// Producer #2
while(true)
    ..
    pipe.Add(msg);
    ..
```

```
// Consumer #1
```

```
// Consumer #2
while(true)
    ..
    msg = pipe.Take();
    ..
```



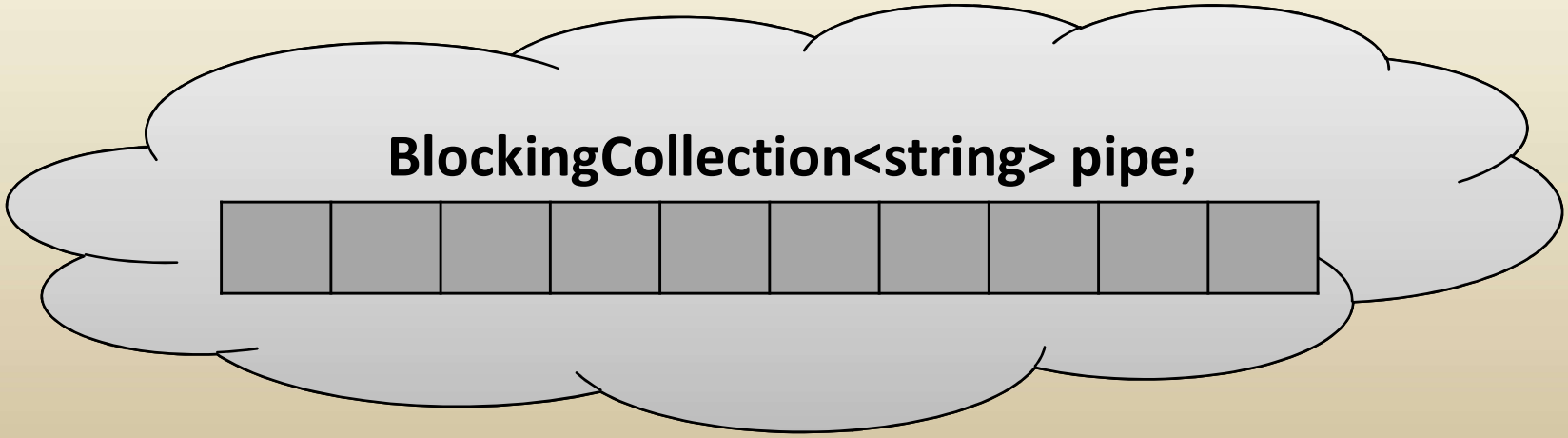
Сценарий «producer-consumer»

```
// Producer #1
w
```

```
// Producer #2
while(true)
    ..
    b = pipe.TryAdd(msg);
    ..
```

```
// Consumer #1
```

```
// Consumer #2
while(true)
    ..
    b = pipe.TryTake(out msg);
    ..
```



TryTake/TryAdd

- Методы TryAdd, TryTake возвращают true или false в зависимости от успешности операции
- **TryTake** возвращает **false**, если не удалось извлечь элемент:
  - очередь пустая;
  - другой поток работает с очередью;
- **TryAdd** возвращает **false**, если
  - очередь переполнена;
  - другой поток работает с очередью;
  - добавление запрещено;

Размер коллекции BlockingCollection

- При создании объекта можем указать максимальный размер очереди:

```
var pipe = new BlockingCollection<int>(100);
```

- Если очередь полностью заполнена, то добавление не выполняется; метод Add приводит к блокировке потока.

Сигнализация в BlockingCollection

- Производитель выполняет вызов **pipe.CompleAdding()** при завершении записи в коллекцию
- Потребители проверяют статус коллекции с помощью свойств:
  - **pipe.IsAddingCompleted:** true – коллекция завершенная
  - **pipe.IsCompleted:** true – коллекция пустая и завершенная
- Добавление элементов с помощью **Add** в завершенную коллекцию приводит к исключению
- Извлечение элементов из завершенной и пустой коллекции с помощью **Take** приводит к исключению
- Метод **TryAdd** для завершенной коллекции возвращает **false**
- Метод **TryTake** для завершенной и пустой коллекции возвращает **false**

Конкурентные словарь

```
bool b; string key; int value;
var dic = new ConcurrentDictionary<string, int>();
..
// Попытка добавления пары «ключ-значение»
b = dic.TryAdd(key, value);
// Попытка извлечения элемента
b = dic.TryGetValue(key, out value);
// Попытка удаления элемента
b = dic.TryRemove(key, out value);
// Попытка обновления элемента
b = dic.TryUpdate(key,  newValue, compareValue);
```

Сценарий «добавить-или-изменить»

```
// Сценарий с обычным словарем
    if(dic.ContainsKey(sKey))
            dic[sKey] = NewValue;
    else
            dic.Add(sKey, InitValue);
```

```
ConcDic.AddOrUpdate(
                    // ключ
                    sKey,
                    // значение для нового элемента
                     InitValue,
                    // обновление
                    (sKey, oldValue) => NewValue);
```

Сценарий «добавить-или-прочитать»

Сценарий получения/добавления элемента

```
if(dic.ContainsKey(sKey))
        // читаем текущее значение
        sCurrValue = dic[sKey];
else
        // добавляем новый элемент
        dic.Add(sKey, sValue);
```

```
sCurrValue = ConcDic.GetOrAdd(sKey, sValue);
```