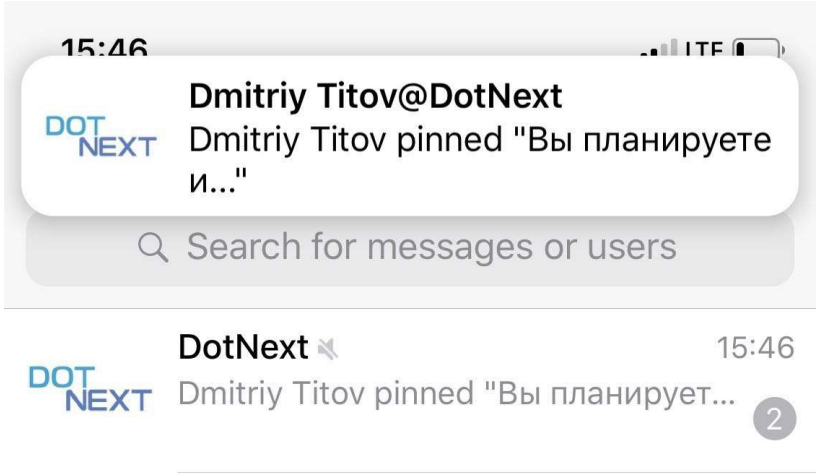
Yield at me 'cause I'm awaiting: асинхронные итераторы в C# 8

Андрей Карпов, ReSharper Team
andrew.karpov@jetbrains.com
Twitter: akarpov89

Часть 1.
Асинхронные последовательности и где они обитают

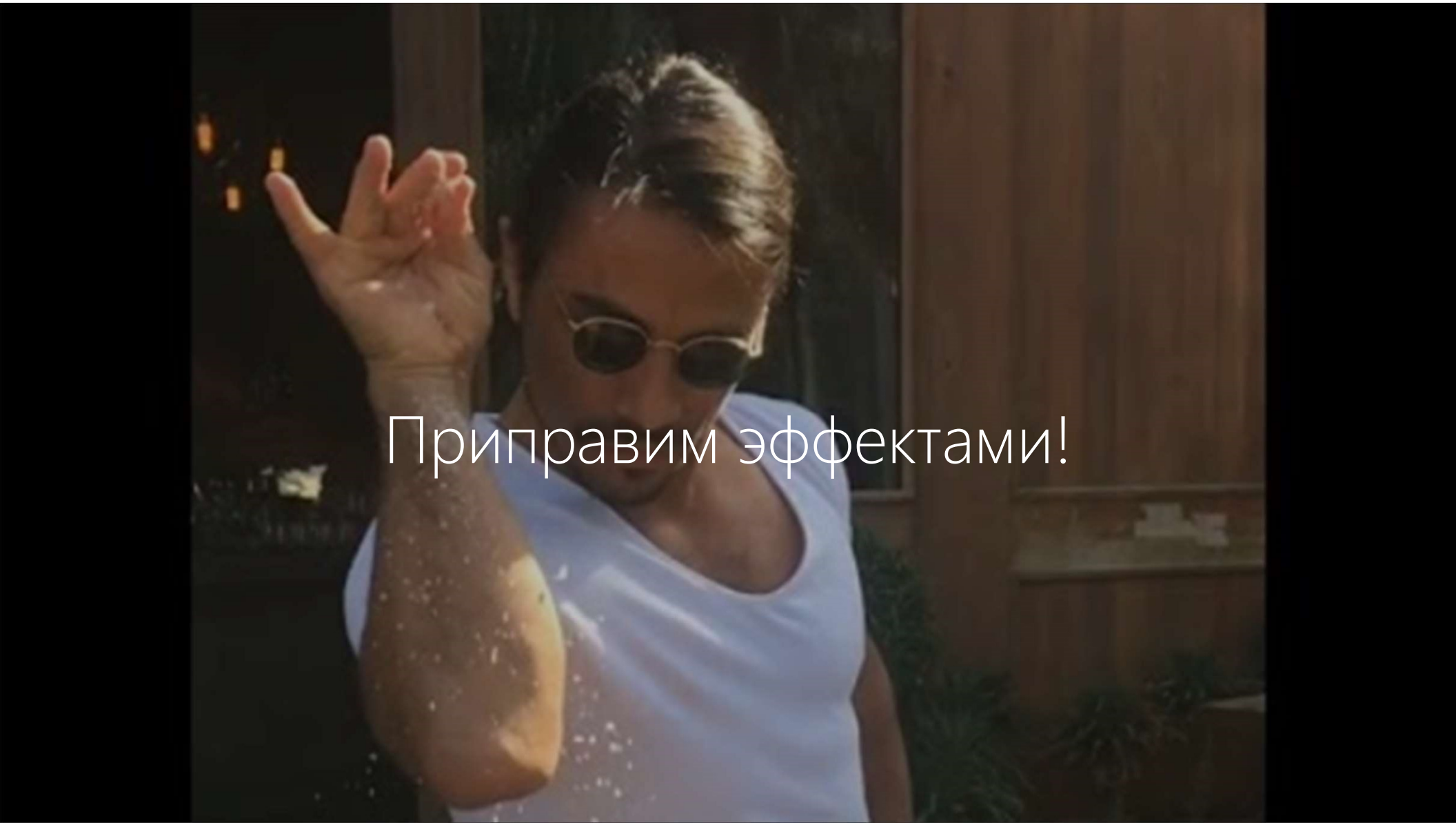
Pull и Push


13:54
#dotnext
CLR via C#


Dmitriy Titov@DotNext
Dmitriy Titov pinned "Вы планируете и..."
Search for messages or users
DotNext
15:46
Dmitriy Titov pinned "Вы планирует...
2

Pull как он есть

```
T Pull()
```


Приправим эффектами!

Отсутствие значения

```
Option<T> Pull()
```

Отсутствие значения

```
public readonly struct Option<T>
{
  public T Value { get; }
  public bool HasValue { get; }

  public Option(T value) => (Value, HasValue) = (value, true);
  public static readonly Option<T> None = default;
}
```

Исключения

```
Option<T> Pull() // throws
```

Исключения: из control-flow в data-flow

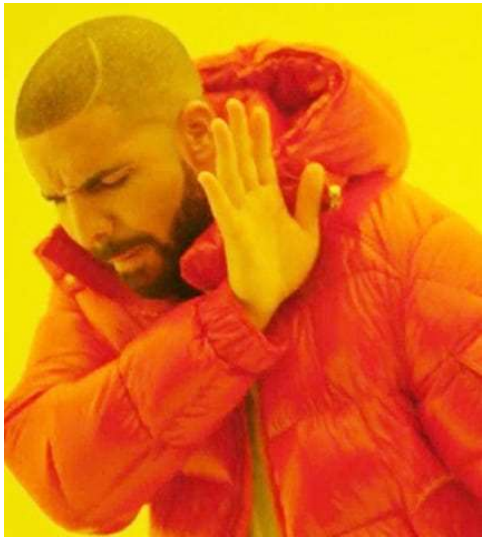
```
Try<Option<T>> Pull()
```

Исключения: из control-flow в data-flow

```
public readonly struct Try<T>
{
  public T Value { get; }
  public Exception Error { get; }

  public Try(T value) => (Value, Error) = (value, null);
  public Try(Exception error) => (Value, Error) = (default, error);
}
```

Откуда взять? Вернуть из другой функции!

```
Func<Try<Option<T>>> GetPuller()
```



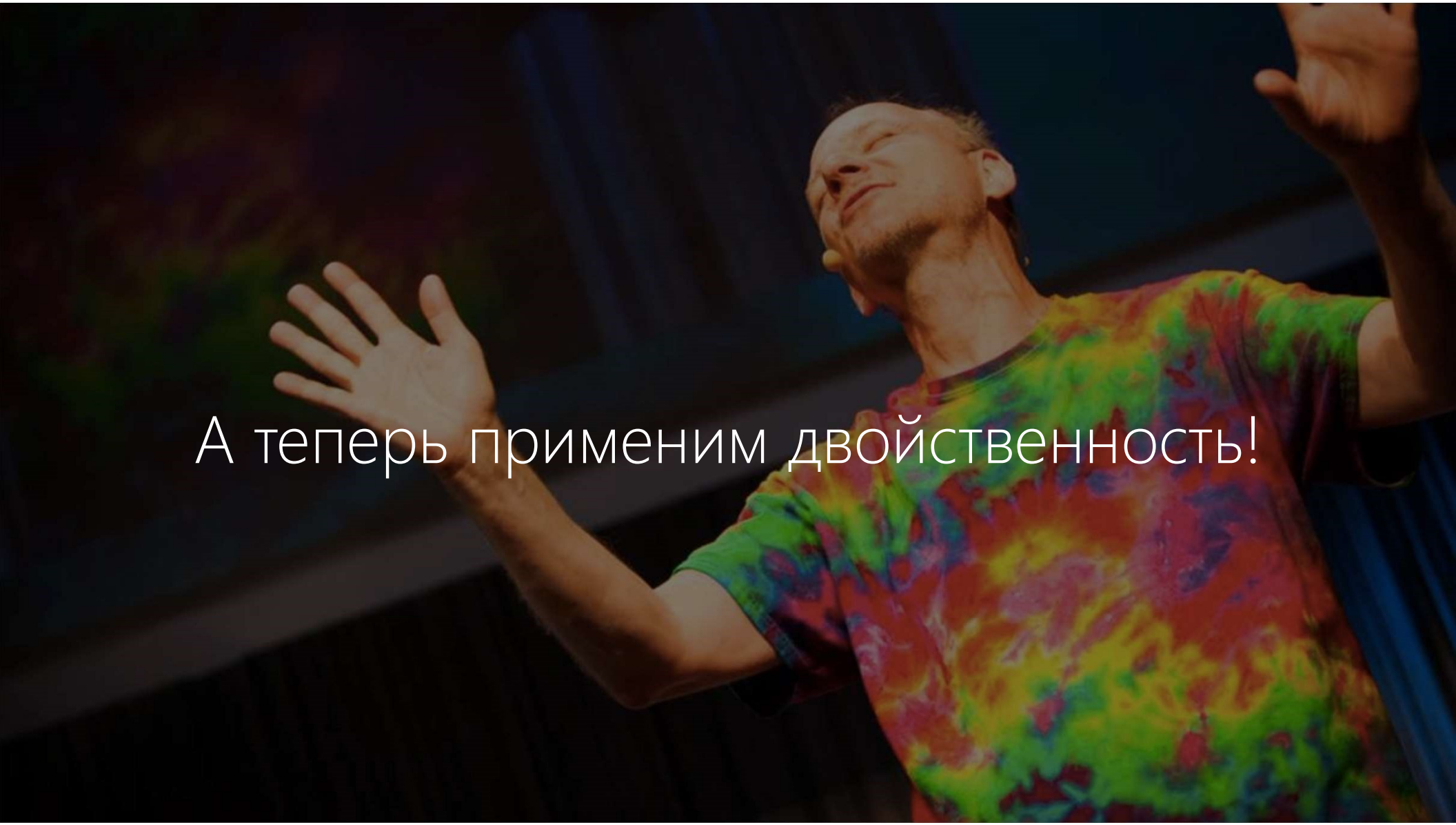
ООП!

```
interface IPullable<T>
{
  IPuller<T> GetPuller();
}

interface IPuller<T>
{
  Try<Option<T>> Pull();
}
```

Что-то хорошо знакомое!

```
interface IEnumerable<out T>
{
  IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T> : IDisposable
{
  bool MoveNext(); // throws
  T Current { get; }
}
```

расщепление
Try<Option<T>>


А теперь применим двойственность!

Меняем местами входы с выходами

```
Try<Option<T>> Pull()

void Push(Try<Option<T>>)
```

ООП!

```
interface IPushable<T>
{
  void SetPusher(IPusher<T> pusher);
}

interface IPusher<T>
{
  void Push(Try<Option<T>> value);
}
```

Что-то хорошо знакомое!

```
interface IObservable<out T>
{
  IDisposable Subscribe(IObserver<T> observer);
}

interface IObserver<in T>
{
  void OnNext(T value);
  void OnCompleted();
  void OnError(Exception error);
}
```

расщепление
Try<Option<T>>

Таааак, подожди...
Задержка (Latency)

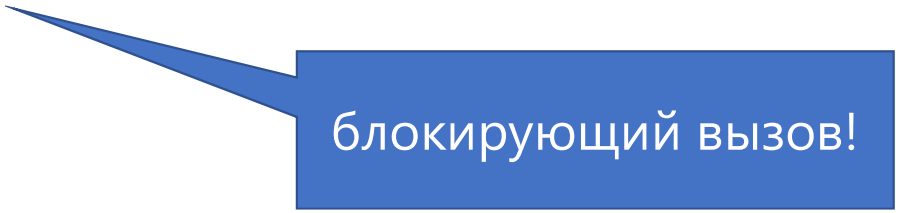
Медленный производитель

```
public void Process<T>(IEnumerator<T> slowProducer)
{
  while (slowProducer.MoveNext())
  {
    Consume(slowProducer.Current);
  }
}
```

блокирующий вызов!

Медленный потребитель

```
public void NotifyNext<T>(
  IObserver<T> slowConsumer, T item)
{
  slowConsumer.OnNext(item);
}
```



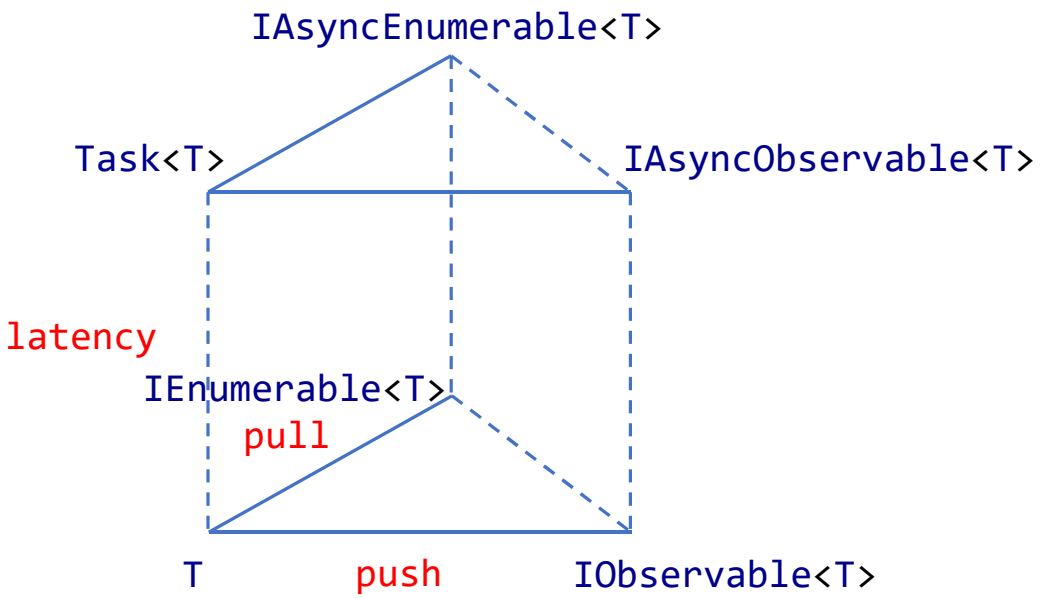
Асинхронный Pull

```
interface IAsyncEnumerable<out T>
{
  IAsyncEnumerator<T> GetAsyncEnumerator();
}

interface IAsyncEnumerator<out T> : IAsyncDisposable
{
  Task<bool> MoveNextAsync();
  T Current { get; }
}
```

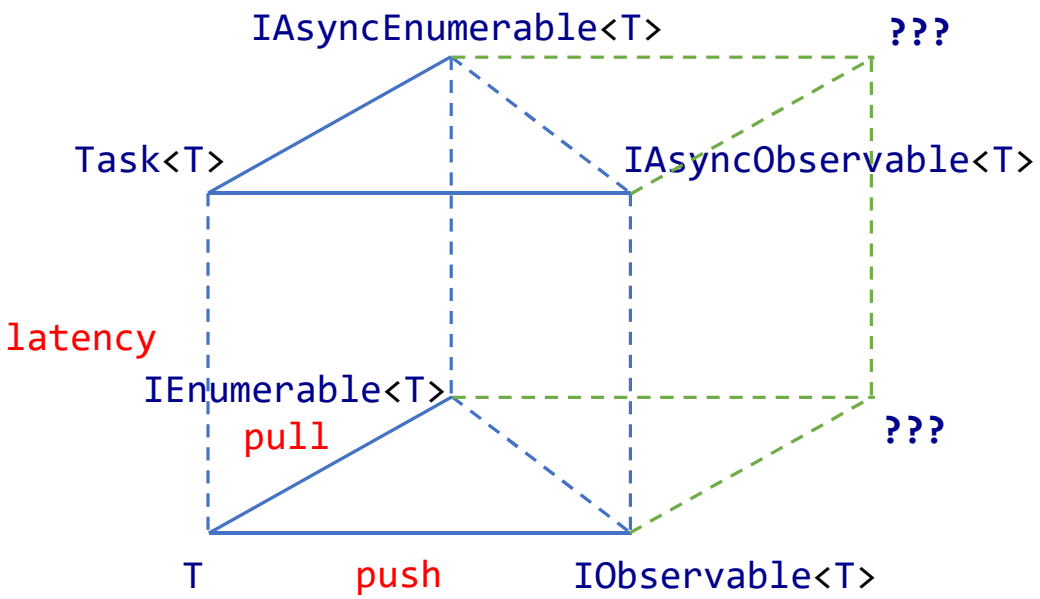
Асинхронный Push

```
interface IAsyncObservable<out T>
{
  Task<IAsyncDisposable> SubscribeAsync(IAsyncObserver<T> observer);
}

interface IAsyncObserver<in T>
{
  Task OnNextAsync(T value);
  Task OnErrorAsync(Exception error);
  Task OnCompletedAsync();
}
```

Собери их всех!



Собери их всех!



Где они обитают?

- Ix.NET (dotnet/reactive) `IAsyncEnumerable<T>`

Где они обитают?

- Ix.NET (dotnet/reactive) `IAsyncEnumerable<T>`
- F# AsyncSeq (fsprojects/FSharp.Control.AsyncSeq) `AsyncSeq<'T>`

```
type IAsyncEnumerator<'T> =
    abstract MoveNext : unit -> Async<'T option>
    inherit IDisposable
type IAsyncEnumerable<'T> =
    abstract GetEnumerator : unit -> IAsyncEnumerator<'T>
type AsyncSeq<'T> = IAsyncEnumerable<'T>
```

Где они обитают?

- Ix.NET (dotnet/reactive) `IAsyncEnumerable<T>`
- F# AsyncSeq (fsprojects/FSharp.Control.AsyncSeq) `AsyncSeq<'T>`
- AsyncRx.NET (dotnet/reactive) `IAsyncObservable<T>`

Где они обитают?

- Ix.NET (dotnet/reactive) `IAsyncEnumerable<T>`
- F# AsyncSeq (fsprojects/FSharp.Control.AsyncSeq) `AsyncSeq<'T>`
- AsyncRx.NET (dotnet/reactive) `IAsyncObservable<T>`
- Orleans (dotnet/orleans) `IAsyncObservable<T>`

Теперь и в BCL!

```
interface IAsyncEnumerable<out T>
{
  IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken token = default);
}
```

```
interface IAsyncEnumerator<out T> : IAsyncDisposable
{
  ValueTask<bool> MoveNextAsync();
  T Current { get; }
}
```

```
interface IAsyncDisposable
{
  ValueTask DisposeAsync();
}
```

*.NET Standard 2.1, .NET Core 3.0

Интересная альтернатива

```
interface IAsyncEnumerable<out T>
{
  IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken token = default);
}
```

```
interface IAsyncEnumerator<out T> : IAsyncDisposable
{
  ValueTask<bool> MoveNextAsync();
  bool TryGetNext(out current);
}
```

```
interface IAsyncDisposable
{
  ValueTask DisposeAsync();
}
```

Уже используется в Channels!

```
public abstract class Channel<TWrite, TRead>
{
  public ChannelReader<TRead> Reader { get; protected set; }
  public ChannelWriter<TWrite> Writer { get; protected set; }
}
public abstract class ChannelReader<T>
{
  public virtual IAsyncEnumerable<T> ReadAllAsync(
    CancellationToken cancellationToken = default);
}
```

Часть 2.
C# 8 спешит на помощь

Это баг!

```
class Issue
{
  public string Title { get; }
  public string Project { get; }
  public string Status { get; }
  …
}
```

Порционная выдача

```
class IssuesProvider
{
  Task<IssuesPage> GetAsync(string assignee, int skip);
}

class IssuesPage
{
  public List<Issue> Issues { get; }
  public int PageSize { get; }
}
```

Попытка №1: Task<IEnumerable<T>>

```
async Task<IEnumerable<Issue>> GetIssues()
{
  var allResults = new List<Issue>();
  var skip = 0;

  return allResults;
}
```

Попытка №1: Task<IEnumerable<T>>

```
async Task<IEnumerable<Issue>> GetIssues()
{
  var allResults = new List<Issue>();
  var skip = 0;
  while (true)
  {
    var issuesPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
  }
  return allResults;
}
```

Попытка №1: Task<IEnumerable<T>>

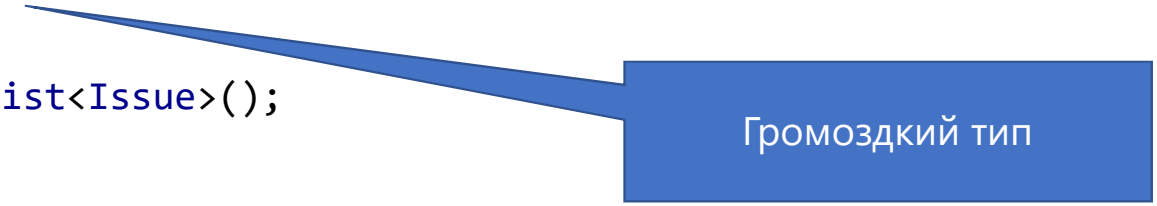
```
async Task<IEnumerable<Issue>> GetIssues()
{
  var allResults = new List<Issue>();
  var skip = 0;
  while (true)
  {
    var issuesPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (issuesPage.PageSize == 0) break;
    allResults.AddRange(issuesPage.Issues);
    skip += issuesPage.PageSize;
  }
  return allResults;
}
```

Попытка №1: Task<IEnumerable<T>>

```
async Task<IEnumerable<Issue>> GetIssues(IProgress<int> progress)
{
  var allResults = new List<Issue>();
  var skip = 0;
  while (true)
  {
    var issuesPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (issuesPage.PageSize == 0) break;
    allResults.AddRange(issuesPage.Issues);
    skip += issuesPage.PageSize;
    progress.Report(skip);
  }
  return allResults;
}
```

Попытка №1: Task<IEnumerable<T>>

```
async Task<IEnumerable<Issue>> GetIssues(IProgress<int> progress)
{
  var allResults = new List<Issue>();
  var skip = 0;
  while (true)
  {
    var issuesPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (issuesPage.PageSize == 0) break;
    allResults.AddRange(issuesPage.Issues);
    skip += issuesPage.PageSize;
    progress.Report(skip);
  }
  return allResults;
}
```



Попытка №1: Task<IEnumerable<T>>

```
async Task<IEnumerable<Issue>> GetIssues(IProgress<int> progress)
{
  var allResults = new List<Issue>();
  var skip = 0;
  while (true)
  {
    var issuesPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (issuesPage.PageSize == 0) break;
    allResults.AddRange(issuesPage.Issues);
    skip += issuesPage.PageSize;
    progress.Report(skip);
  }
  return allResults;
}
```

Накапливаем потенциально большой список

Попытка №1: Task<IEnumerable<T>>

```
async Task<IEnumerable<Issue>> GetIssues(IProgress<int> progress)
{
  var allResults = new List<Issue>();
  var skip = 0;
  while (true)
  {
    var issuesPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (issuesPage.PageSize == 0) break;
    allResults.AddRange(issuesPage.Issues);
    skip += issuesPage.PageSize;
    progress.Report(skip);
  }
  return allResults;
}
```

Попытка №2: IEnumerable<Task<T>>

```
IEnumerable<Task<IssuesPage>> GetIssues()
{
  var skip = 0;
  Task<IssuesPage> currentPage = null;
}
```

Попытка №2: IEnumerable<Task<T>>

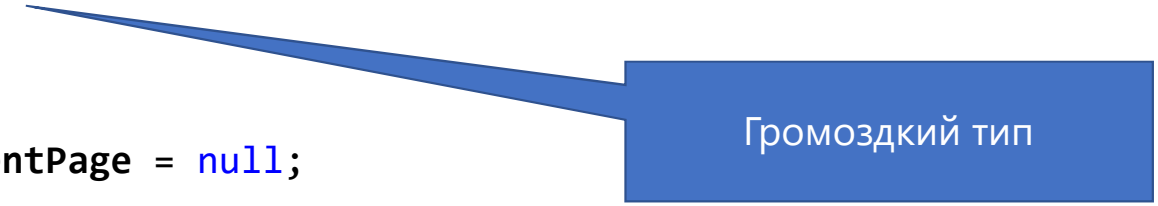
```
IEnumerable<Task<IssuesPage>> GetIssues()
{
  var skip = 0;
  Task<IssuesPage> currentPage = null;
  while (true) {
    currentPage = _client.GetAsync(assignee: "andrew.karpov", skip);
    yield return currentPage;
  }
}
```

Попытка №2: IEnumerable<Task<T>>

```
IEnumerable<Task<IssuesPage>> GetIssues()
{
  var skip = 0;
  Task<IssuesPage> currentPage = null;
  while (true) {
    if (currentPage != null)
    {
      if (currentPage.Result.PageSize == 0) yield break;
      skip += currentPage.Result.PageSize;
    }
    currentPage = _client.GetAsync(assignee: "andrew.karpov", skip);
    yield return currentPage;
  }
}
```

Попытка №2: IEnumerable<Task<T>>

```
IEnumerable<Task<IssuesPage>> GetIssues()
{
  var skip = 0;
  Task<IssuesPage> currentPage = null;
  while (true) {
    if (currentPage != null)
    {
      if (currentPage.Result.PageSize == 0) yield break;
      skip += currentPage.Result.PageSize;
    }
    currentPage = _client.GetAsync(assignee: "andrew.karpov", skip);
    yield return currentPage;
  }
}
```



Попытка №2: IEnumerable<Task<T>>

```
IEnumerable<Task<IssuesPage>> GetIssues()
{
  var skip = 0;
  Task<IssuesPage> currentPage = null;
  while (true) {
    if (currentPage != null)
    {
      if (currentPage.Result.PageSize == 0) yield break;
      skip += currentPage.Result.PageSize;
    }
    currentPage = _client.GetAsync(assignee: "andrew.karpov", skip);
    yield return currentPage;
  }
}
```

Неявно требуем await

Попытка №2: IEnumerable<Task<T>>

```
IEnumerable<Task<IssuesPage>> GetIssues()
{
  var skip = 0;
  Task<IssuesPage> currentPage = null;
  while (true) {
    if (currentPage != null)
    {
      if (currentPage.Result.PageSize == 0) yield break;
      skip += currentPage.Result.PageSize;
    }
    currentPage = _client.GetAsync(assignee: "andrew.karpov", skip);
    yield return currentPage;
  }
}
```

В конце всегда возвращаем пустую страницу


МОНАДЫ НЕ
КОМПОЗИРУЮТСЯ!

Попытка №3: IAsyncEnumerable<T> из Ix.NET

```
IAsyncEnumerable<IssuesPage> GetIssues() => AsyncEnumerable.CreateEnumerable(() =>
{
  return AsyncEnumerable.CreateEnumerator(
    moveNext: null, current: null, dispose: () => { });
});
```

Попытка №3: IAsyncEnumerable<T> из Ix.NET

```
IAsyncEnumerable<IssuesPage> GetIssues() => AsyncEnumerable.CreateEnumerable(() =>
{
  IssuesPage currentPage = null;
  async Task<bool> GetPage(CancellationToken token) {

  }
  return AsyncEnumerable.CreateEnumerator(
    moveNext: GetPage, current: () => currentPage, dispose: () => { });
});
```

Попытка №3: IAsyncEnumerable<T> из Ix.NET

```
IAsyncEnumerable<IssuesPage> GetIssues() => AsyncEnumerable.CreateEnumerable(() =>
{
  var skip = 0;
  IssuesPage currentPage = null;
  async Task<bool> GetPage(CancellationToken token) {
    currentPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (currentPage.PageSize == 0) return false;
    skip += currentPage.PageSize;
    return true;
  }
  return AsyncEnumerable.CreateEnumerator(
    moveNext: GetPage, current: () => currentPage, dispose: () => { });
});
```

C# 8: yield + await = асинхронный итератор

```
async IAsyncEnumerable<IssuesPage> GetIssuesAsync()
{
  var skip = 0;
  while (true)
  {
    var currentPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (currentPage.PageSize == 0) yield break;
    skip += currentPage.PageSize;
    yield return currentPage;
  }
}
```

C# 8: yield + await = асинхронный итератор

```
async IAsyncEnumerable<Issue> GetIssuesAsync()
{
  var skip = 0;
  while (true)
  {
    var currentPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (currentPage.PageSize == 0) yield break;
    skip += currentPage.PageSize;
    foreach (var issue in currentPage.Issues)
      yield return issue;
  }
}
```

C# 8: yield + await = асинхронный итератор

```
async IAsyncEnumerable<Issue> GetIssuesAsync()
{
  var skip = 0;
  while (true)
  {
    var currentPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (currentPage.PageSize == 0) yield break;
    skip += currentPage.PageSize;
    foreach (var issue in currentPage.Issues)
      yield return issue;
  }
}
```

Модификатор async

C# 8: yield + await = асинхронный итератор

```
async IAsyncEnumerable<Issue> GetIssuesAsync()
{
  var skip = 0;
  while (true)
  {
    var currentPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (currentPage.PageSize == 0) yield break;
    skip += currentPage.PageSize;
    foreach (var issue in currentPage.Issues)
      yield return issue;
  }
}
```

Возвращаемый тип
IAsyncEnumerable<T> или
IAsyncEnumerator<T>

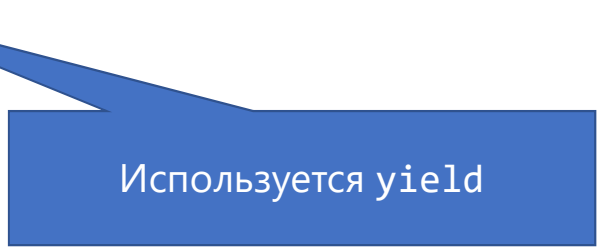
C# 8: yield + await = асинхронный итератор

```
async IAsyncEnumerable<Issue> GetIssuesAsync()
{
  var skip = 0;
  while (true)
  {
    var currentPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (currentPage.PageSize == 0) yield break;
    skip += currentPage.PageSize;
    foreach (var issue in currentPage.Issues)
      yield return issue;
  }
}
```

Используется await, await foreach или await using

C# 8: yield + await = асинхронный итератор

```
async IAsyncEnumerable<Issue> GetIssuesAsync()
{
  var skip = 0;
  while (true)
  {
    var currentPage = await _client.GetAsync(assignee: "andrew.karpov", skip);
    if (currentPage.PageSize == 0) yield break;
    skip += currentPage.PageSize;
    foreach (var issue in currentPage.Issues)
      yield return issue;
  }
}
```



C# 8: асинхронный foreach

```
async Task PrintIssuesAsync()
{
  await foreach (var issue in GetIssuesAsync())
  {
    PrintIssue(issue);
  }
}
```

Часть 3.
Lowering: искусство обмана

Асинхронный foreach

```
await foreach (var issuesPage in GetIssuesAsync())
{
  PrintPage(issuesPage);
}
```

Асинхронный foreach: lowering

```
var enumerator = GetIssuesAsync().GetAsyncEnumerator(/* параметр не передаётся */);
try
{
  while (await enumerator.MoveNextAsync())
  {
    var issuesPage = enumerator.Current;
    PrintPage(issuesPage);
  }
}
finally
{
  await enumerator.DisposeAsync();
}
```

Паттерн типа: уважаем форму

```
var enumerator = GetIssuesAsync().GetAsyncEnumerator();
try
{
  while (await enumerator.MoveNextAsync())
  {
    var issuesPage = enumerator.Current;
    PrintPage(issuesPage);
  }
}
finally
{
  await enumerator.DisposeAsync();
}
```

Может иметь один опциональный или params параметр любого типа

Паттерн типа: уважаем форму

```
var enumerator = GetIssuesAsync().GetAsyncEnumerator();
try
{
  while (await enumerator.MoveNextAsync())
  {
    var issuesPage = enumerator.Current;
    PrintPage(issuesPage);
  }
}
finally
{
  await enumerator.DisposeAsync();
}
```

Может иметь один опциональный или params параметр любого типа. Возвращаемый тип должен быть awaitable.

Паттерн типа: уважаем форму

```
var enumerator = GetIssuesAsync().GetAsyncEnumerator();
try
{
  while (await enumerator.MoveNextAsync())
  {
    var issuesPage = enumerator.Current;
    PrintPage(issuesPage);
  }
}
finally
{
  await enumerator.DisposeAsync();
}
```

Необязателен,
нет метода – нет вызова

Паттерн типа vs. интерфейс

```
struct MyAsyncEnumerable : IAsyncEnumerable<int>
{
  public async IAsyncEnumerator<int> GetAsyncEnumerator(
    CancellationToken cancellationToken /*= default*/)
  {
    yield return await Task.FromResult(0);
  }
}
```

Паттерн типа vs. интерфейс

```
struct MyAsyncEnumerable : IAsyncEnumerable<int>
{
  public async IAsyncEnumerator<int> GetAsyncEnumerator(
    CancellationToken cancellationToken /*= default*/)
  {
    yield return await Task.FromResult(0);
  }

  public async Task Print()
  {
    await foreach (var x in this)
      Console.WriteLine(x);
  }
}
```

Паттерн типа vs. интерфейс

```
struct MyAsyncEnumerable : IAsyncEnumerable<int>
{
  public async IAsyncEnumerator<int> GetAsyncEnumerator(
    CancellationToken cancellationToken /*= default*/)
  {
    yield return await Task.FromResult(0);
  }

  public async Task Print()
  {
    await foreach (var x in (IAsyncEnumerable<int>) this)
      Console.WriteLine(x);
  }
}
```

Паттерн типа vs. интерфейс

```
struct MyAsyncEnumerable : IAsyncEnumerable<int>
{
  public async IAsyncEnumerator<int> GetAsyncEnumerator(
    CancellationToken cancellationToken = default)
  {
    yield return await Task.FromResult(0);
  }

  public async Task Print()
  {
    await foreach (var x in this)
      Console.WriteLine(x);
  }
}
```

Отмена (Cancellation)

```
async Task PrintIssuesAsync()
{
  await foreach (var issue in GetIssuesAsync())
  {
    PrintIssue(issue);
  }
}
```

Отмена (Cancellation)

```
async Task PrintIssuesAsync(CancellationToken token)
{
  await foreach (var issue in GetIssuesAsync(token))
  {
    PrintIssue(issue);
  }
}
```

Отмена (Cancellation)

```
async Task PrintIssuesAsync(IAsyncEnumerable<Issue> issues)
{
  await foreach (var issue in issues)
  {
    PrintIssue(issue);
  }
}
```

Отмена (Cancellation)

```
async Task PrintIssuesAsync(IAsyncEnumerable<Issue> issues, CancellationToken token)
{
  await foreach (var issue in issues.WithCancellation(token))
  {
    PrintIssue(issue);
  }
}
```

Настройка контекста синхронизации

```
async Task PrintIssuesAsync(IAsyncEnumerable<Issue> issues, CancellationToken token)
{
  await foreach (var issue in issues.WithCancellation(token)
                                    .ConfigureAwait(continueOnCapturedContext: false))
  {
    PrintIssue(issue);
  }
}
```

Полезные расширения

```
public static class TaskExtensions
{
  public static ConfiguredCancelableAsyncEnumerable<T> ConfigureAwait<T>(
    this IAsyncEnumerable<T> source,
    bool continueOnCapturedContext);

  public static ConfiguredCancelableAsyncEnumerable<T> WithCancellation<T>(
    this IAsyncEnumerable<T> source,
    CancellationToken cancellationToken);
}
```

Так вот зачем тот параметр

```
public readonly struct ConfiguredCancelableAsyncEnumerable<T>
{
  public Enumerator GetAsyncEnumerator() =>
    new Enumerator(_enumerable.GetAsyncEnumerator(_cancellationToken),
                   _continueOnCapturedContext);
}
```

Делегируй и управляй

```
public readonly struct Enumerator
{
  public T Current => _enumerator.Current;

  public ConfiguredValueTaskAwaitable<bool> MoveNextAsync() =>
    _enumerator.MoveNextAsync().ConfigureAwait(_continueOnCapturedContext);

  public ConfiguredValueTaskAwaitable DisposeAsync() =>
    _enumerator.DisposeAsync().ConfigureAwait(_continueOnCapturedContext);
}
```

Есть асинхронный итератор

```
static async IAsyncEnumerable<int> ProduceAsync() {
  await Task.Delay(100); yield return 1;
  await Task.Delay(100); yield return 2;
  await Task.Delay(100); yield return 3;
}
```

И цикл

```
static async Task Main() {
  await foreach (var x in ProduceAsync())
  {
    Console.Write("{0} ", x);
  }
}
static async IAsyncEnumerable<int> ProduceAsync() {
  await Task.Delay(100); yield return 1;
  await Task.Delay(100); yield return 2;
  await Task.Delay(100); yield return 3;
}
```

Что выведет следующий код?

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync().WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
}
static async IAsyncEnumerable<int> ProduceAsync() {
  await Task.Delay(100); yield return 1;
  await Task.Delay(100); yield return 2;
  await Task.Delay(100); yield return 3;
}
```

A) 1 2 3
B) 1 (нормальное завершение)
C) 1 (исключение)

Что выведет следующий код?

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync().WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
}
static async IAsyncEnumerable<int> ProduceAsync() {
  await Task.Delay(100); yield return 1;
  await Task.Delay(100); yield return 2;
  await Task.Delay(100); yield return 3;
}
```

A) 1 2 3
B) 1 (нормальное завершение)
C) 1 (исключение)

Добавим параметр

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync(default).WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
}
static async IAsyncEnumerable<int> ProduceAsync(CancellationToken token = default) {
  await Task.Delay(100, token); yield return 1;
  await Task.Delay(100, token); yield return 2;
  await Task.Delay(100, token); yield return 3;
}
```

Добавим параметр

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync(default).WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
}
static async IAsyncEnumerable<int> ProduceAsync(CancellationToken token = default) {
  await Task.Delay(100, token); yield return 1;
  await Task.Delay(100, token); yield return 2;
  await Task.Delay(100, token); yield return 3;
}
```

По-прежнему выводит “1 2 3”!!!

Помечаем параметр атрибутом

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync(default).WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
}
static async IAsyncEnumerable<int> ProduceAsync([EnumeratorCancellation]
                                                CancellationToken token = default) {
  await Task.Delay(100, token); yield return 1;
  await Task.Delay(100, token); yield return 2;
  await Task.Delay(100, token); yield return 3;
}
```

Помечаем параметр атрибутом

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync(default).WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
}
static async IAsyncEnumerable<int> ProduceAsync([EnumeratorCancellation]
                                                CancellationToken token = default) {
  await Task.Delay(100, token); yield return 1;
  await Task.Delay(100, token); yield return 2;
  await Task.Delay(100, token); yield return 3;
}
```

Выводит “1” и завершается по исключению

А как насчёт локальных функций?

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync(default).WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
  static async IAsyncEnumerable<int> ProduceAsync([EnumeratorCancellation]
                                                  CancellationToken token = default) {
    await Task.Delay(100, token); yield return 1;
    await Task.Delay(100, token); yield return 2;
    await Task.Delay(100, token); yield return 3;
  }
}
```

А как насчёт локальных функций?

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync(default).WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
  static async IAsyncEnumerable<int> ProduceAsync([EnumeratorCancellation]
                                                  CancellationToken token = default) {
    await Task.Delay(100, token); yield return 1;
    await Task.Delay(100, token); yield return 2;
    await Task.Delay(100, token); yield return 3;
  }
}
```

Локальные функции не могут иметь атрибутов

А как насчёт локальных функций?

```
static async Task Main() {
  var cts = new CancellationTokenSource();
  await foreach (var x in ProduceAsync(default).WithCancellation(cts.Token))
  {
    Console.Write("{0} ", x);
    cts.Cancel();
  }
  static async IAsyncEnumerable<int> ProduceAsync([EnumeratorCancellation]
                                                  CancellationToken token = default) {
    await Task.Delay(100, token); yield return 1;
    await Task.Delay(100, token); yield return 2;
    await Task.Delay(100, token); yield return 3;
  }
}
```

В C# 8 будет можно!

Lowering асинхронного итератора

```
async IAsyncEnumerable<Issue> GetIssuesAsync([EnumeratorCancellation]
                                             CancellationToken token = default)
{
  var skip = 0;
  while (true)
  {
    var currentPage = await _client.GetAsync(assignee: "andrew.karpov", skip, token);
    if (currentPage.PageSize == 0) yield break;
    skip += currentPage.PageSize;
    foreach (var issue in currentPage.Issues)
      yield return issue;
  }
}
```

Тот-Чьё-Имя-Нельзя-Называть

```
[CompilerGenerated]
private sealed class VoldemortStateMachine
  : IAsyncEnumerable<Issue>, IAsyncEnumerator<Issue>,
    IValueTaskSource<bool>, IValueTaskSource,
    IAsyncStateMachine
{

}
```

Оптимизируй это

```
[CompilerGenerated]
private sealed class VoldemortStateMachine
  : IAsyncEnumerable<Issue>, IAsyncEnumerator<Issue>,
    IValueTaskSource<bool>, IValueTaskSource,
    IAsyncStateMachine
{

}
```

GetAsyncEnumerator может возвращать this

Оптимизируй то

```
[CompilerGenerated]
private sealed class VoldemortStateMachine
  : IAsyncEnumerable<Issue>, IAsyncEnumerator<Issue>,
    IValueTaskSource<bool>, IValueTaskSource,
    IAsyncStateMachine
{

}
```

Для избегания аллокаций Task в MoveNextAsync и DisposeAsync

Оптимизируй то

```
[CompilerGenerated]
private sealed class VoldemortStateMachine
  : IAsyncEnumerable<Issue>, IAsyncEnumerator<Issue>,
    IValueTaskSource<bool>, IValueTaskSource,
    IAsyncStateMachine
{

}
```

```
interface IValueTaskSource<out TResult>
{
  ValueTaskSourceStatus GetStatus(short token);
  void OnCompleted(Action<object> continuation, short token, …);
  TResult GetResult(short token);
}
```

Оптимизируй то

```
[CompilerGenerated]
private sealed class VoldemortStateMachine
  : IAsyncEnumerable<Issue>, IAsyncEnumerator<Issue>,
    IValueTaskSource<bool>, IValueTaskSource,
    IAsyncStateMachine
{
  ManualResetValueTaskSourceCore<bool> PromiseOfValueOrEnd;
}
```

Структура, которой делегируется реализация интерфейсов (.NET Standard 2.1, .NET Core 3.0)

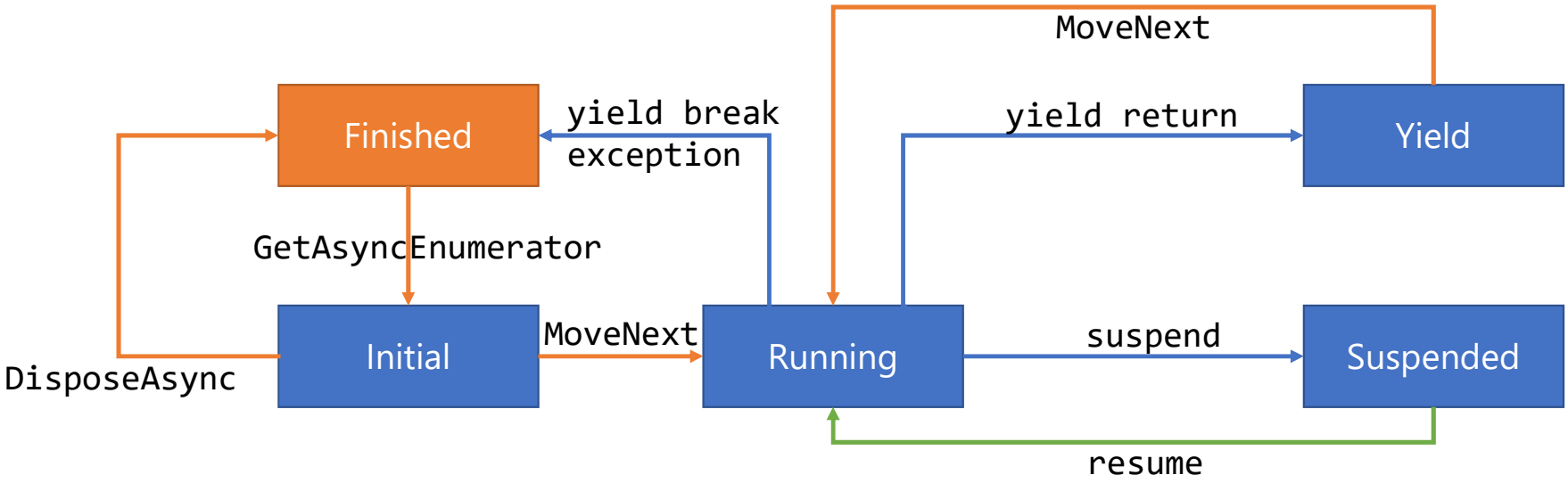
Оптимизируй то

```
[CompilerGenerated]
private sealed class VoldemortStateMachine
  : IAsyncEnumerable<Issue>, IAsyncEnumerator<Issue>,
    IValueTaskSource<bool>, IValueTaskSource,
    IAsyncStateMachine
{
  ManualResetValueTaskSourceCore<bool> PromiseOfValueOrEnd;
}

    // в MoveNextAsync
    new ValueTask<bool>((IValueTaskSource<bool>) this, PromiseOfValueOrEnd.Version)
```

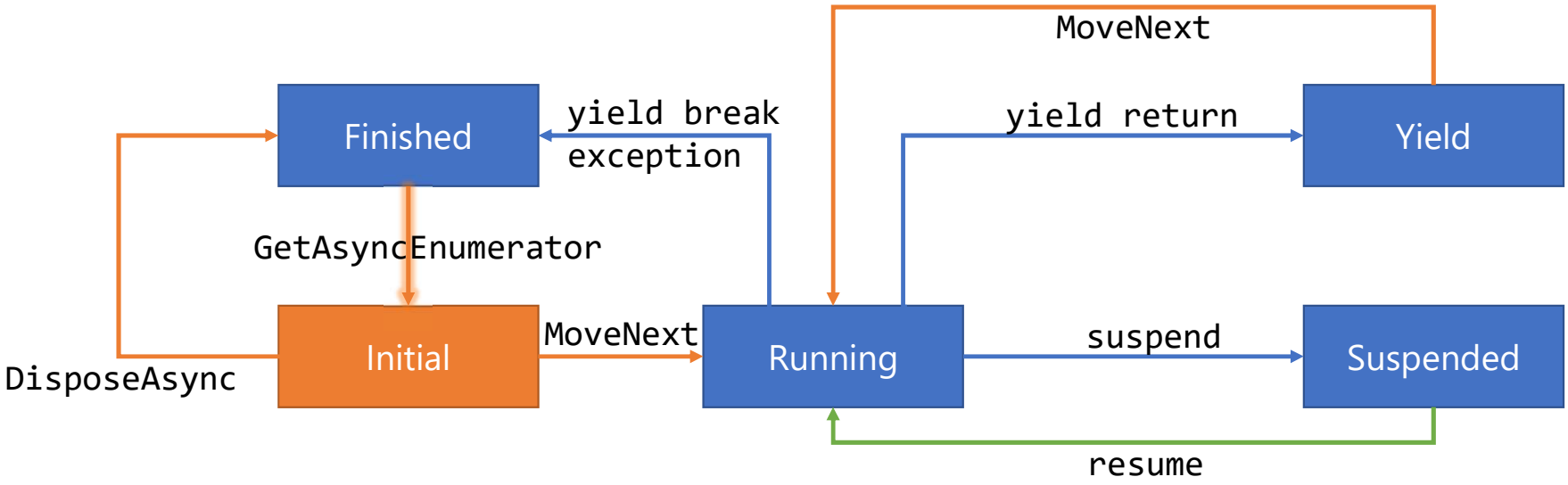
Движение – это жизнь!

```
[CompilerGenerated]
private sealed class VoldemortStateMachine
  : IAsyncEnumerable<Issue>, IAsyncEnumerator<Issue>,
    IValueTaskSource<bool>, IValueTaskSource,
    IAsyncStateMachine
{

}
```

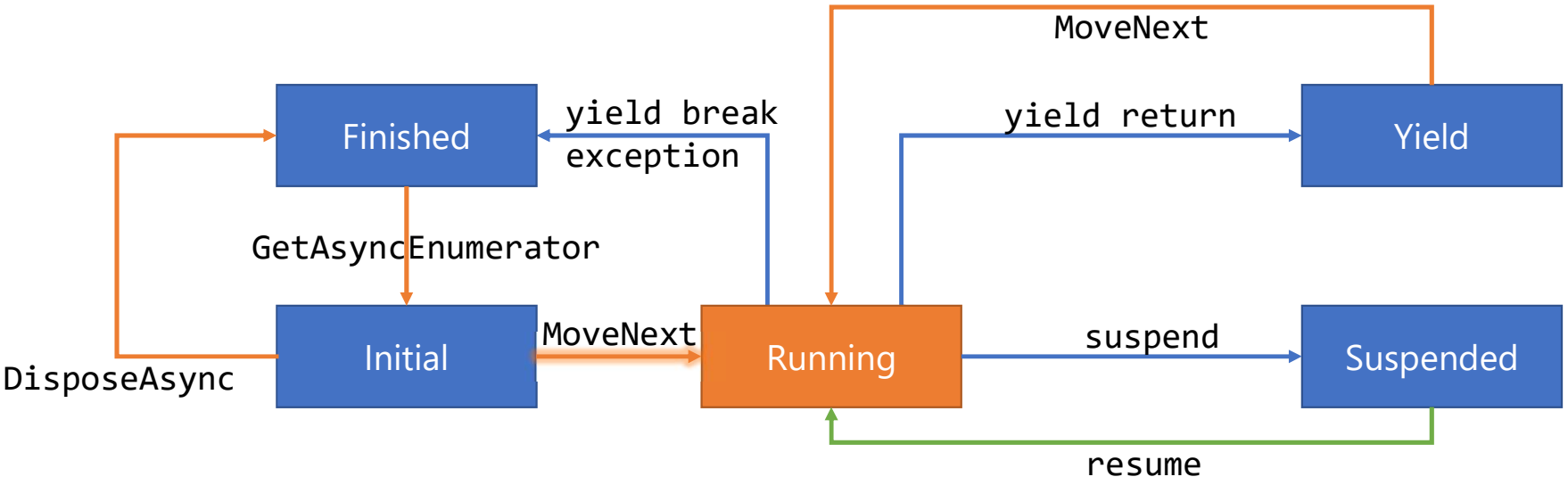
Переходы, переходики...



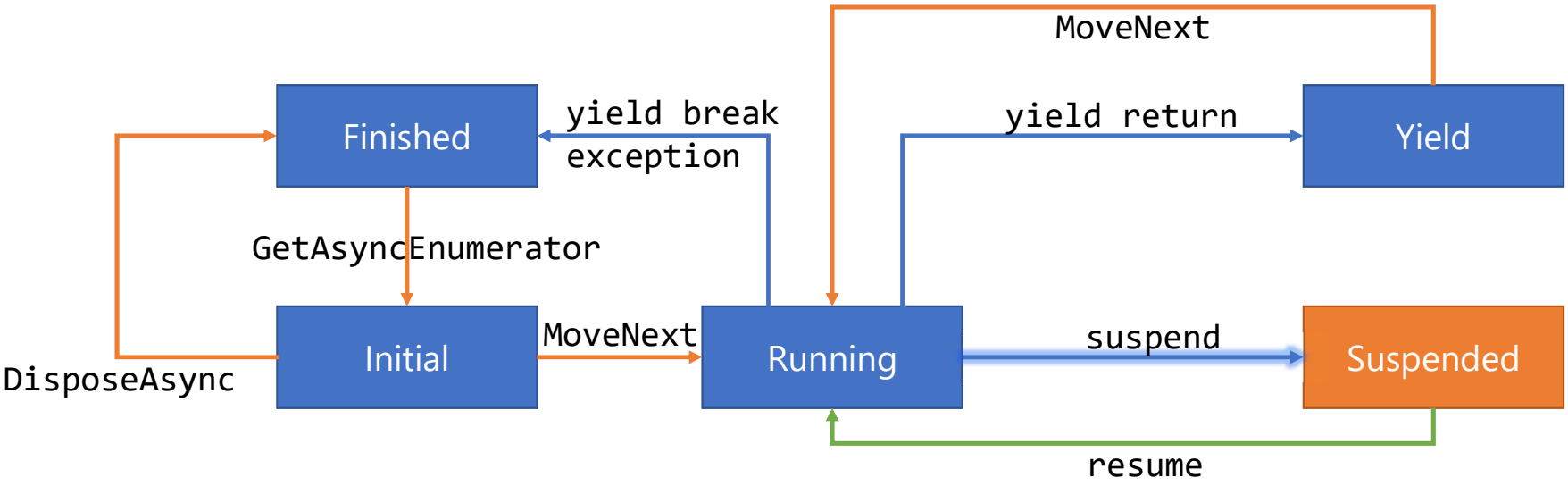
Переходы, переходики...



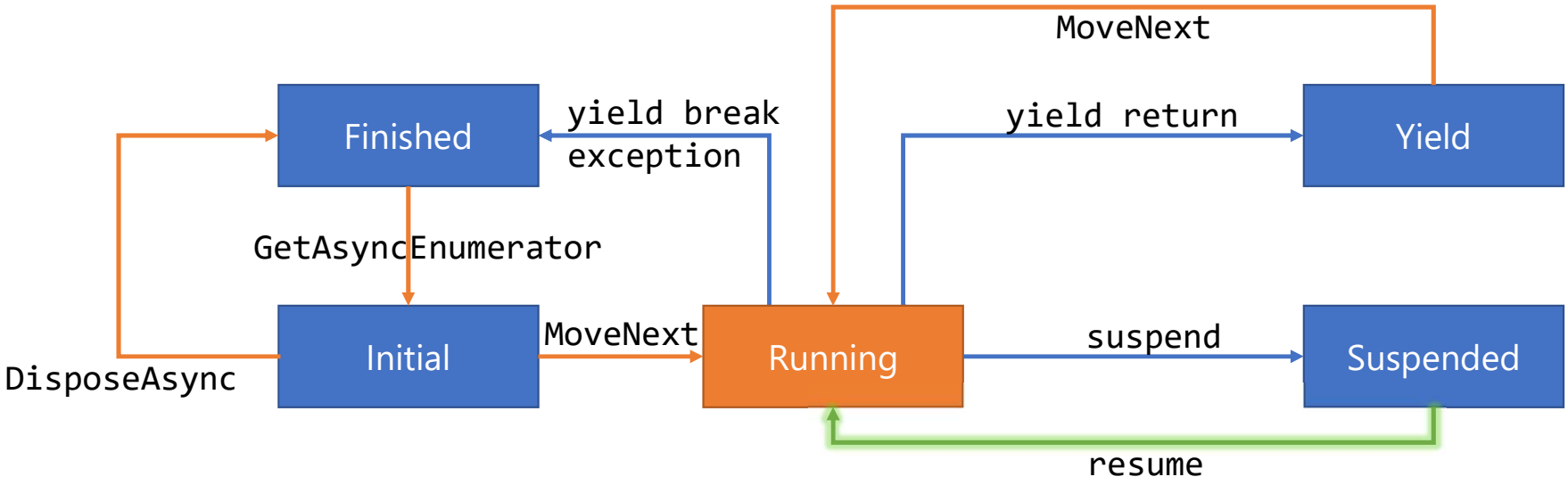
Переходы, переходики...



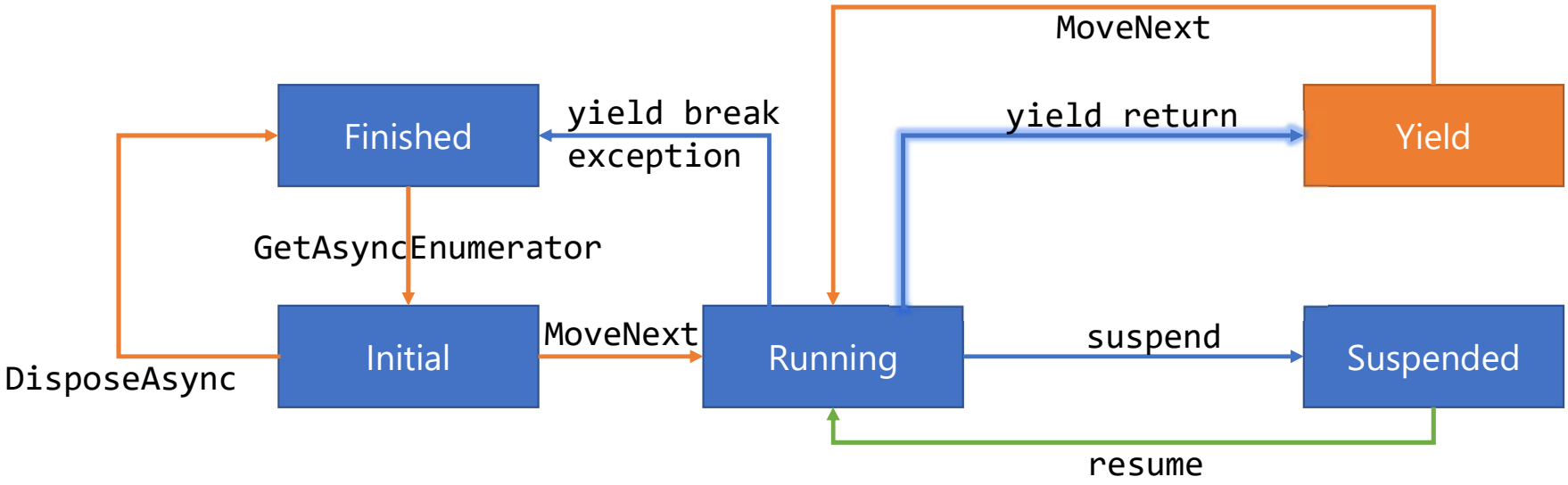
Переходы, переходики...



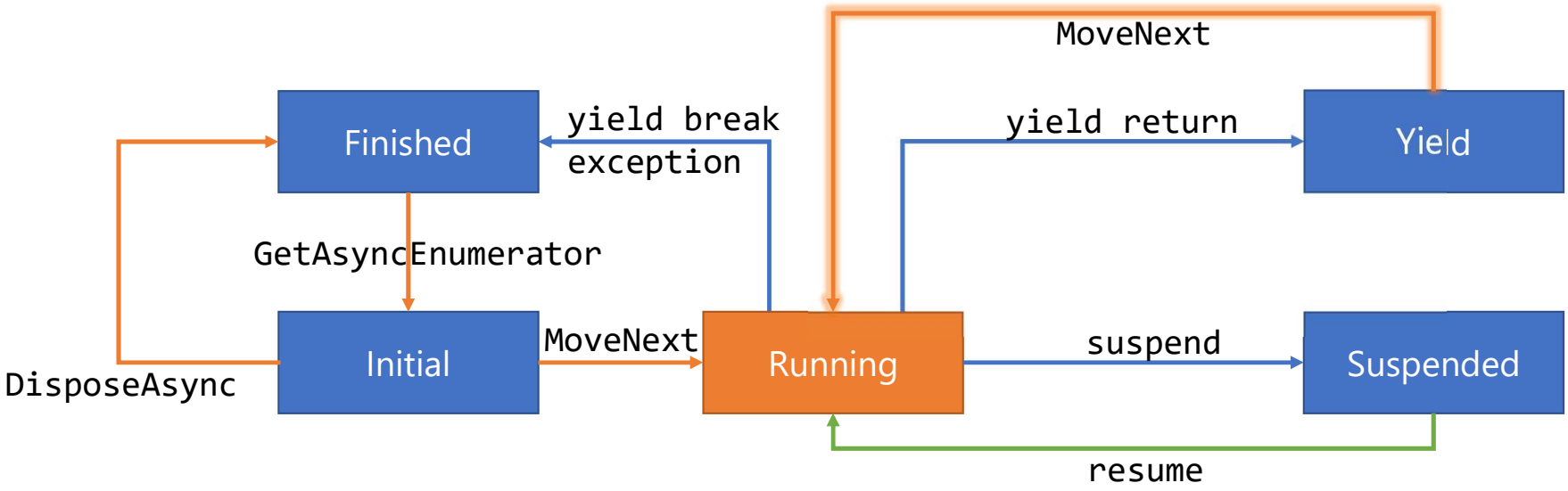
Переходы, переходики...



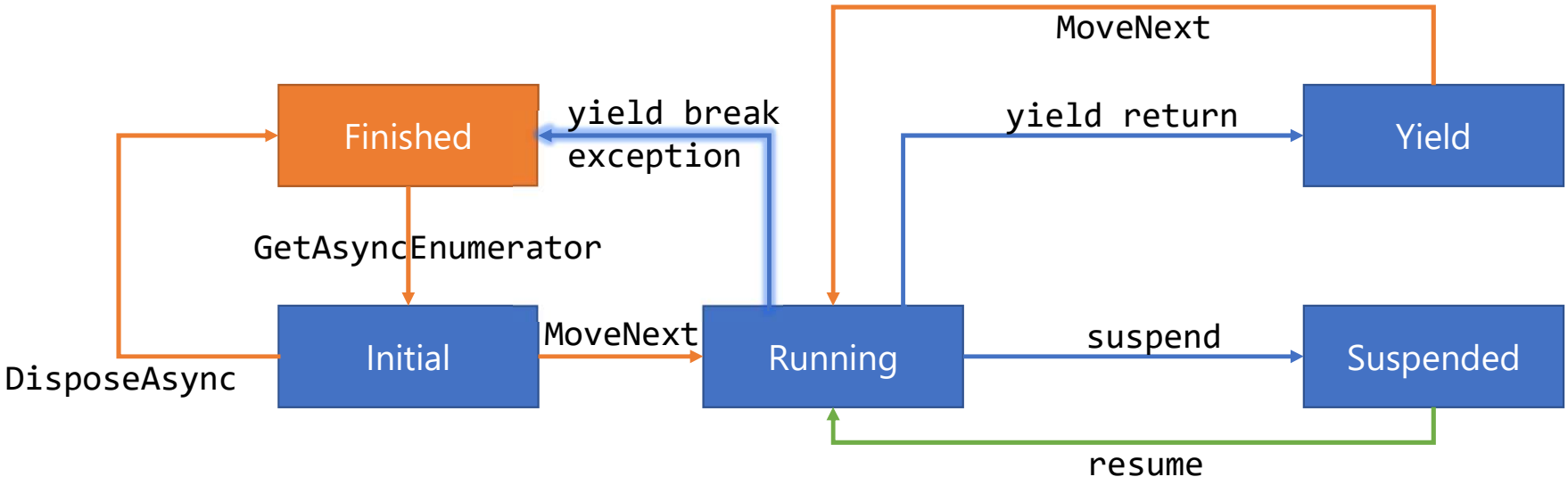
Переходы, переходики...



Переходы, переходики...



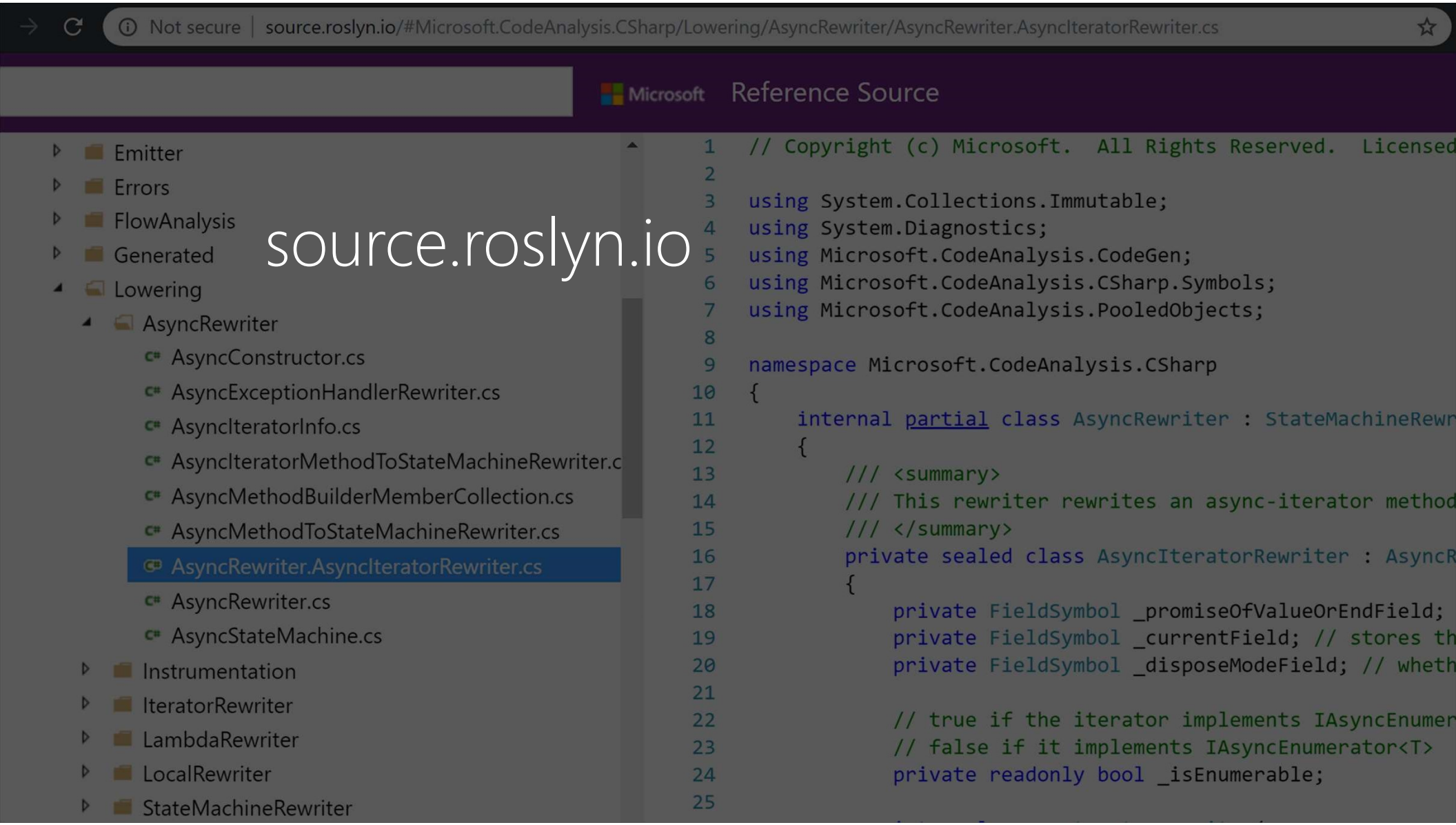
Переходы, переходики...



Время dotPeek

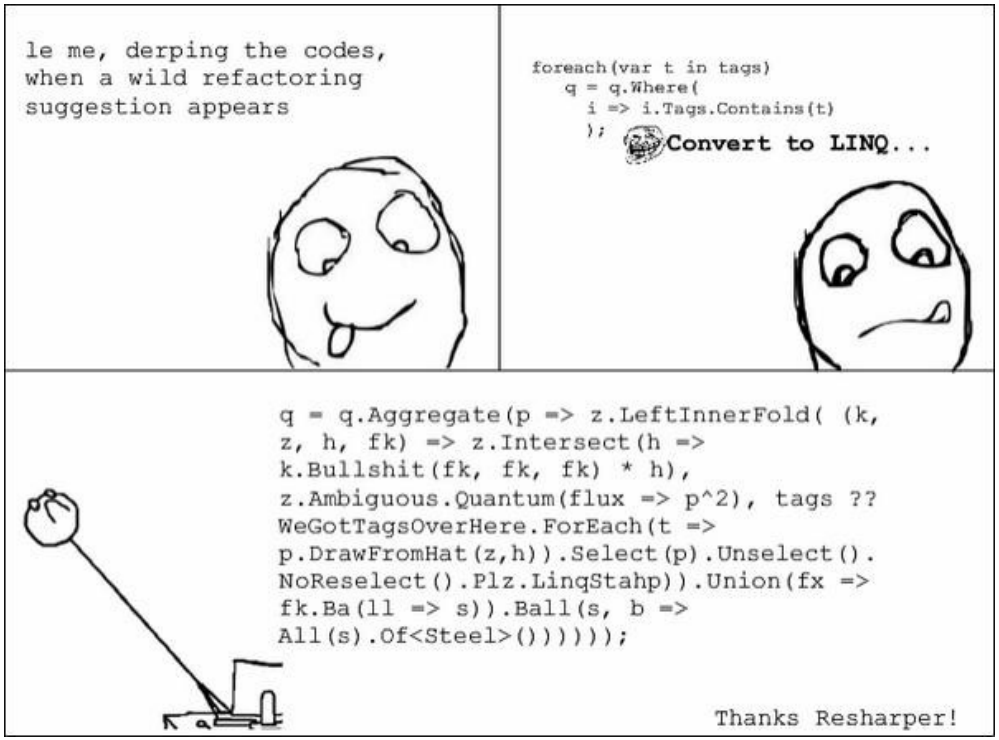
Go To Statement Considered Useful!

```
void IAsyncStateMachine.MoveNext() {
  try { switch (State) { /* goto, goto, goto... */  } }
  catch (Exception ex) {
    State = StateMachineStates.Finished;
    CombinedTokens?.Dispose(); CombinedTokens = null;
    PromiseOfValueOrEnd.SetException(ex);
    return;
  }
returnFalseLabel:
  State = StateMachineStates.Finished;
  CombinedTokens?.Dispose(); CombinedTokens = null;
  PromiseOfValueOrEnd.SetResult(false);
  return;
returnTrueLabel: PromiseOfValueOrEnd.SetResult(true);
}
```


Not secure | source.roslyn.io/#Microsoft.CodeAnalysis.CSharp/Lowering/AsyncRewriter/AsyncRewriter.AsyncIteratorRewriter.cs
Microsoft Reference Source
source.roslyn.io
Emitter
Errors
FlowAnalysis
Generated
Lowering
AsyncRewriter
AsyncConstructor.cs
AsyncExceptionHandlerRewriter.cs
AsyncIteratorInfo.cs
AsyncIteratorMethodToStateMachineRewriter.c
AsyncMethodBuilderMemberCollection.cs
AsyncMethodToStateMachineRewriter.cs
AsyncRewriter.AsyncIteratorRewriter.cs
AsyncRewriter.cs
AsyncStateMachine.cs
Instrumentation
IteratorRewriter
LambdaRewriter
LocalRewriter
StateMachineRewriter
1 // Copyright (c) Microsoft. All Rights Reserved. Licensed
2
3 using System.Collections.Immutable;
4 using System.Diagnostics;
5 using Microsoft.CodeAnalysis.CodeGen;
6 using Microsoft.CodeAnalysis.CSharp.Symbols;
7 using Microsoft.CodeAnalysis.PooledObjects;
8
9 namespace Microsoft.CodeAnalysis.CSharp
10 {
11 internal partial class AsyncRewriter : StateMachineRewr
12 {
13 /// <summary>
14 /// This rewriter rewrites an async-iterator method
15 /// </summary>
16 private sealed class AsyncIteratorRewriter : AsyncR
17 {
18 private FieldSymbol _promiseOfValueOrEndField;
19 private FieldSymbol _currentField; // stores th
20 private FieldSymbol _disposeModeField; // wheth
21
22 // true if the iterator implements IAsyncEnumer
23 // false if it implements IAsyncEnumerator<T>
24 private readonly bool _isEnumerable;
25

Часть 4.
Искусство асинхронных запросов

LINQ


le me, derping the codes,
when a wild refactoring
suggestion appears
foreach(var t in tags)
q = q.Where(
i => i.Tags.Contains(t)
);
Convert to LINQ...
q = q.Aggregate(p => z.LeftInnerFold((k,
z, h, fk) => z.Intersect(h =>
k.Bullshit(fk, fk, fk) * h),
z.Ambiguous.Quantum(flux => p^2), tags ??
WeGotTagsOverHere.ForEach(t =>
p.DrawFromHat(z,h)).Select(p).Unselect().
NoReselect().Plz.LinqStahp)).Union(fx =>
fk.Ba(ll => s)).Ball(s, b =>
All(s).Of<Steel>()))))));
Thanks Resharper!

NuGet

System.Interactive.Async 4.0.0-preview.1.build.745

Interactive Extensions Async Library used to express queries over asynchronous enumerable sequences.

ⓘ This is a prerelease version of System.Interactive.Async.

Операторы

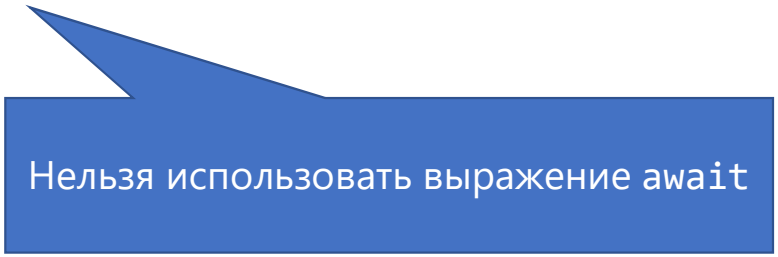
```
IAsyncEnumerable<TResult> Select<TSource, TResult>(
  this IAsyncEnumerable<TSource> source,
  Func<TSource, TResult> selector);
```

Синтаксис выражений запросов

```
var issuesInProgress =
    from issue in GetIssuesAsync()
    where issue.Status == "In progress"
    select issue.Title;
```

Синтаксис выражений запросов

```
var issues =
    from issue in GetIssuesAsync()
    where await predicate(issue)
    select issue.Title;
```



Агрегируем асинхронно

```
ValueTask<bool> AnyAsync<TSource>(
  this IAsyncEnumerable<TSource> source,
  CancellationToken cancellationToken = default);
```

Когда делегат асинхронный

```
ValueTask<long> SumAwaitAsync<TSource>(
  this IAsyncEnumerable<TSource> source,
  Func<TSource, ValueTask<long>> selector,
  CancellationToken cancellationToken = default);
```

Гранулярная отмена

```
IAsyncEnumerable<TResult>
SelectAwaitWithCancellation<TSource, TResult>(
  this IAsyncEnumerable<TSource> source,
  Func<TSource, CancellationToken, ValueTask<TResult>> selector);
```

Demo

Подведём итоги

- .NET Core 3.0 вводит стандартный API асинхронных pull последовательней

Подведём итоги

- .NET Core 3.0 вводит стандартный API асинхронных pull последовательней
- `IAsyncEnumerable<T>` имеет свои применения, это не замена `IEnumerable<T>`/`IObservable<T>`

Подведём итоги

- .NET Core 3.0 вводит стандартный API асинхронных pull последовательней
- `IAsyncEnumerable<T>` имеет свои применения, это не замена `IEnumerable<T>`/`IObservable<T>`
- C# 8 обеспечивает удобную языковую поддержку для генерирования и потребления асинхронных последовательностей

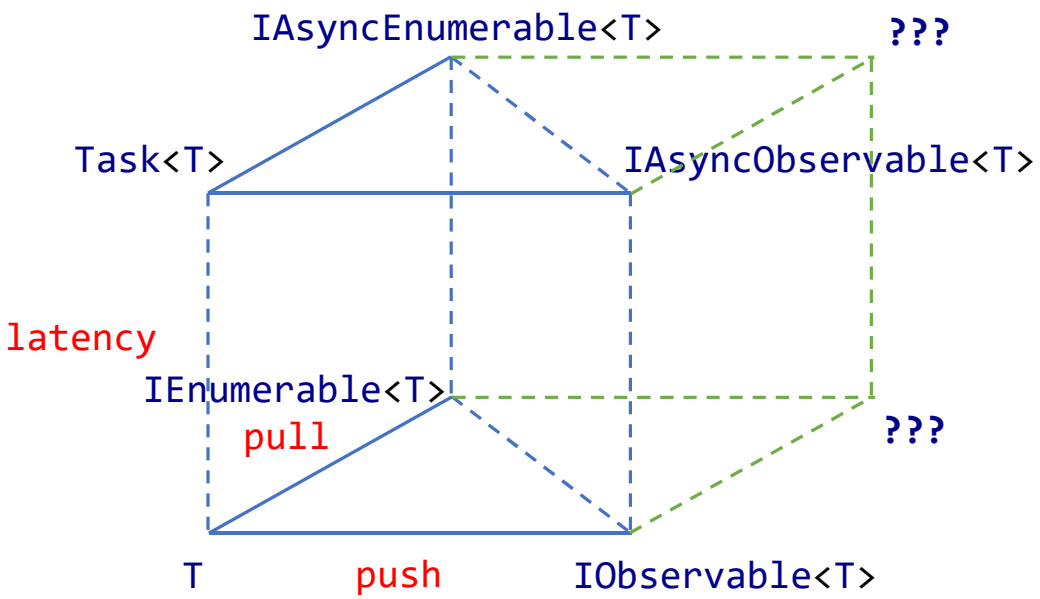
Подведём итоги

- .NET Core 3.0 вводит стандартный API асинхронных pull последовательней
- `IAsyncEnumerable<T>` имеет свои применения, это не замена `IEnumerable<T>`/`IObservable<T>`
- C# 8 обеспечивает удобную языковую поддержку для генерирования и потребления асинхронных последовательностей
- При работе с async streams следует учитывать отмену и контекст синхронизации

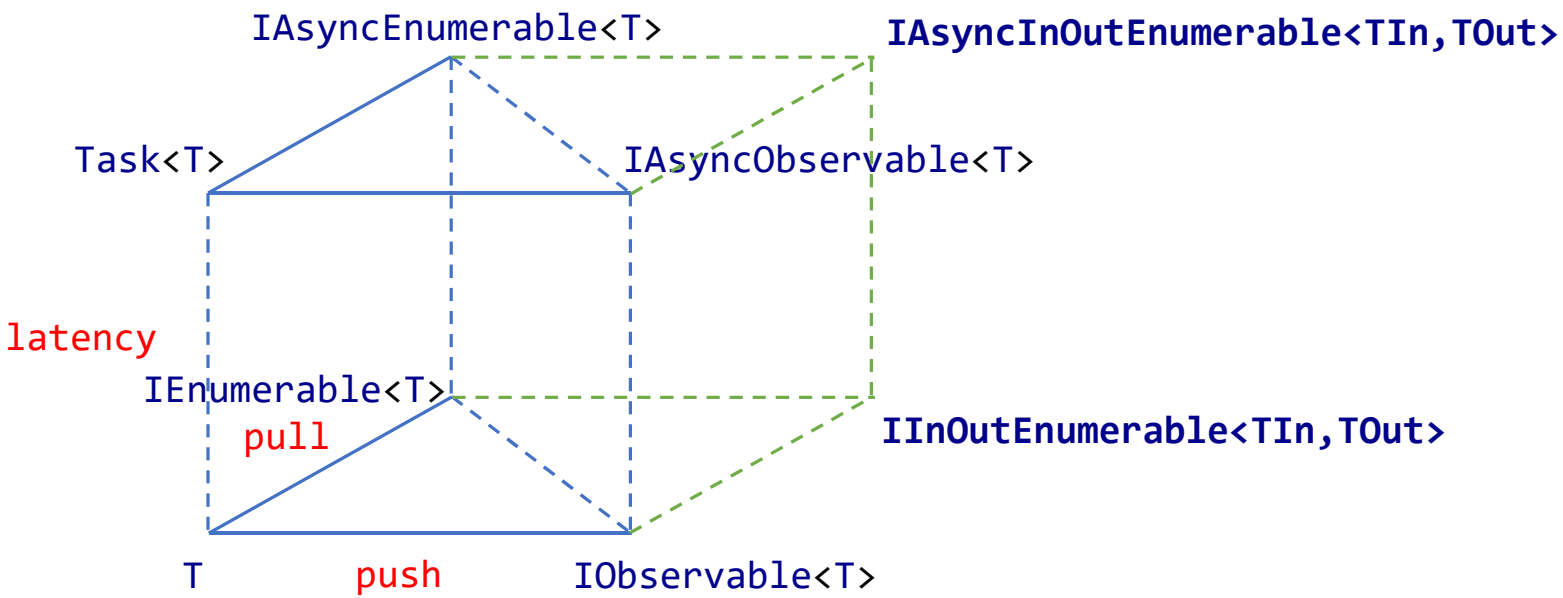
Подведём итоги

- .NET Core 3.0 вводит стандартный API асинхронных pull последовательней
- `IAsyncEnumerable<T>` имеет свои применения, это не замена `IEnumerable<T>`/`IObservable<T>`
- C# 8 обеспечивает удобную языковую поддержку для генерирования и потребления асинхронных последовательностей
- При работе с async streams следует учитывать отмену и контекст синхронизации
- LINQ доступен с помощью nuget-пакета `System.Interactive.Async`

Собери их всех!



Пофантазируем...



Push & Pull последовательности

```
interface IInOutEnumerable<in TIn, out TOut>
{
  IInOutEnumerator<TIn, TOut> GetEnumerator();
}

interface IInOutEnumerator<in TIn, out TOut>
{
  bool MoveNext(TIn input);
  TOut Current { get; }
}
```

Push & Pull последовательности

```
interface IAsyncInOutEnumerable<in TIn, out TOut>
{
  IAsyncInOutEnumerator<TIn, TOut> GetAsyncEnumerator();
}

interface IAsyncInOutEnumerator<in TIn, out TOut>
{
  ValueTask<bool> MoveNextAsync(TIn input);
  TOut Current { get; }
}
```

Полезные ресурсы

- https://github.com/dotnet/csharplang/blob/master/proposals/csharp-8.0/async-streams.md
- https://github.com/dotnet/roslyn/blob/master/docs/features/async-streams.md
- http://source.roslyn.io/#Microsoft.CodeAnalysis.CSharp/Lowering/AsyncRewriter/AsyncRewriter.AsyncIteratorRewriter.cs
- https://devblogs.microsoft.com/dotnet/understanding-the-whys-whats-and-whens-of-valuetask/
- https://github.com/dotnet/reactive
- https://github.com/akarnokd/async-enumerable-dotnet/wiki/Writing-operators
- http://fsprojects.github.io/FSharp.Control.AsyncSeq/library/AsyncSeq.html

Q & A