

управление данными в микросервисах на C#

устройство .NET (2)

асинхронное программирование в .NET

асинхронное программирование в .NET

compute bound vs i/o bound

compute bound

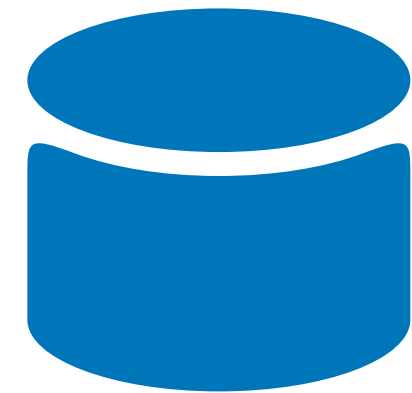
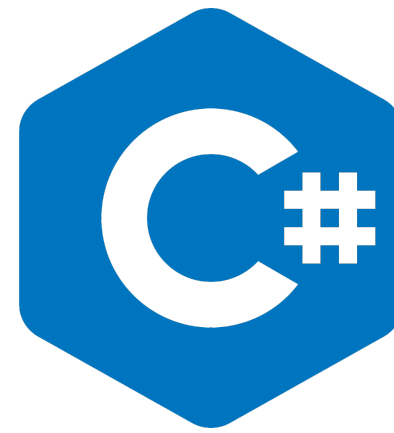
- основной задачей являются вычисления
- серьёзность характеризуется количеством или сложностью вычислений

i/o bound

- основной задачей является работа с устройствами ввода/вывода
- серьёзность определяется длительностью ожидания ответа от устройств ввода/вывода

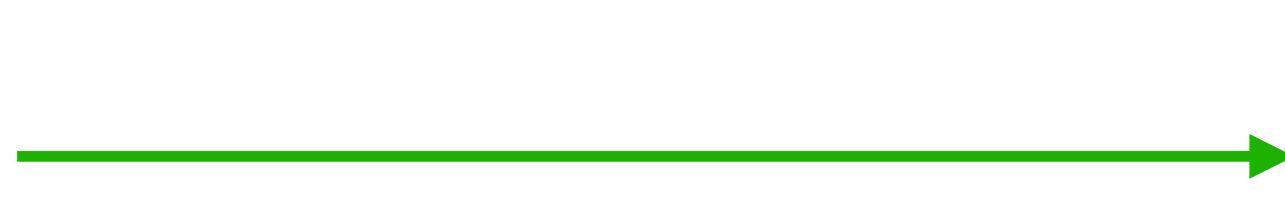
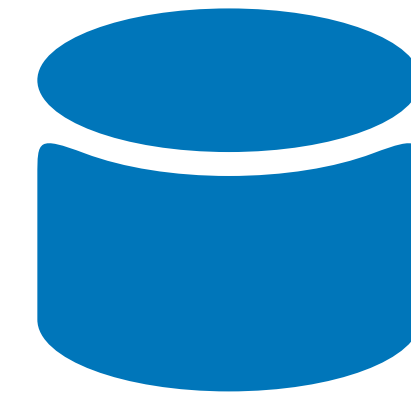
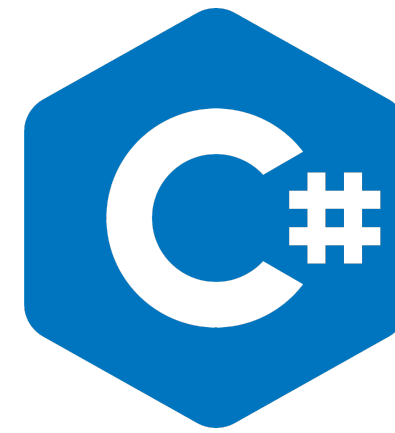
асинхронное программирование в .NET

блокирующие i/o операции



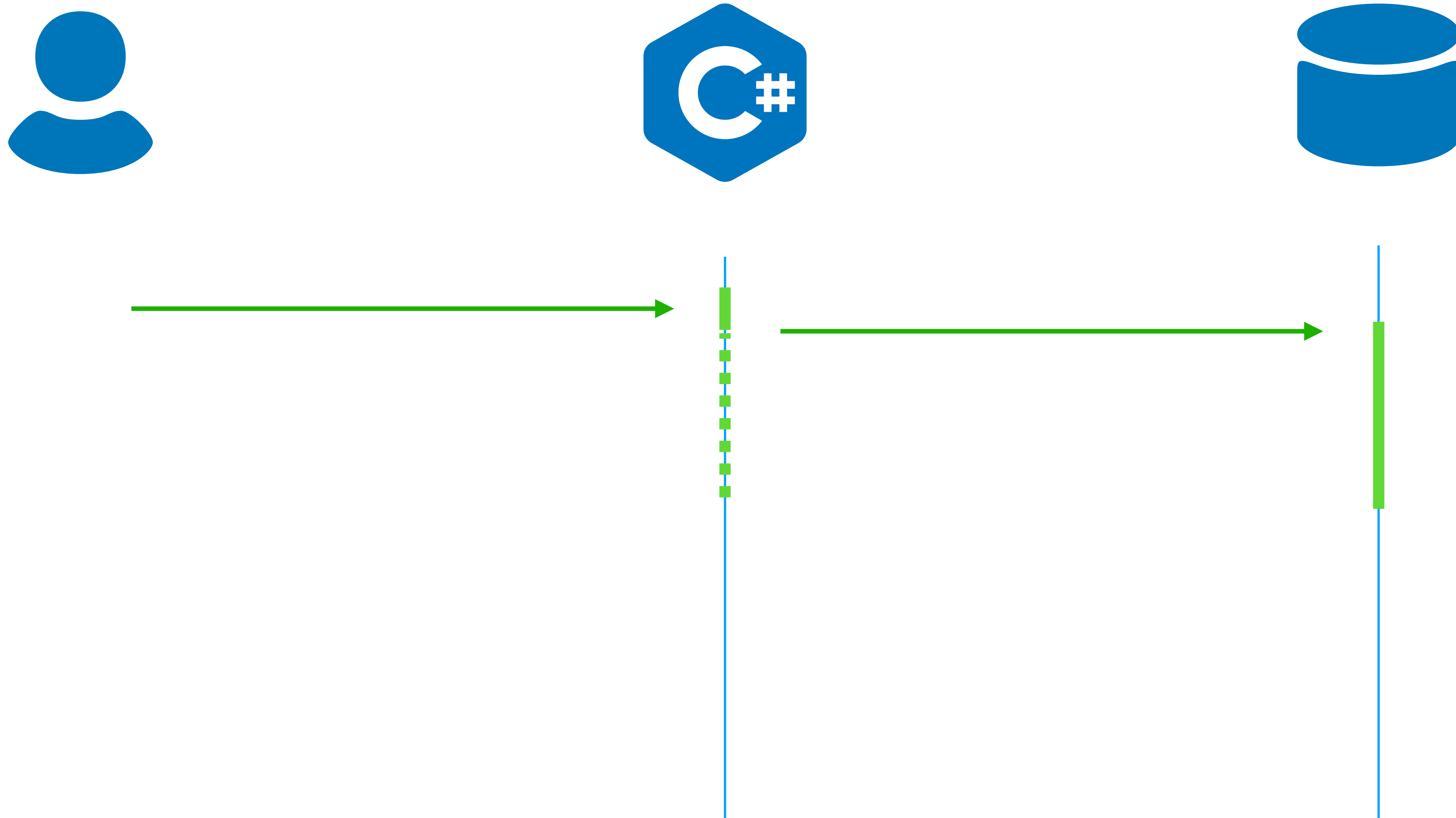
асинхронное программирование в .NET

блокирующие i/o операции



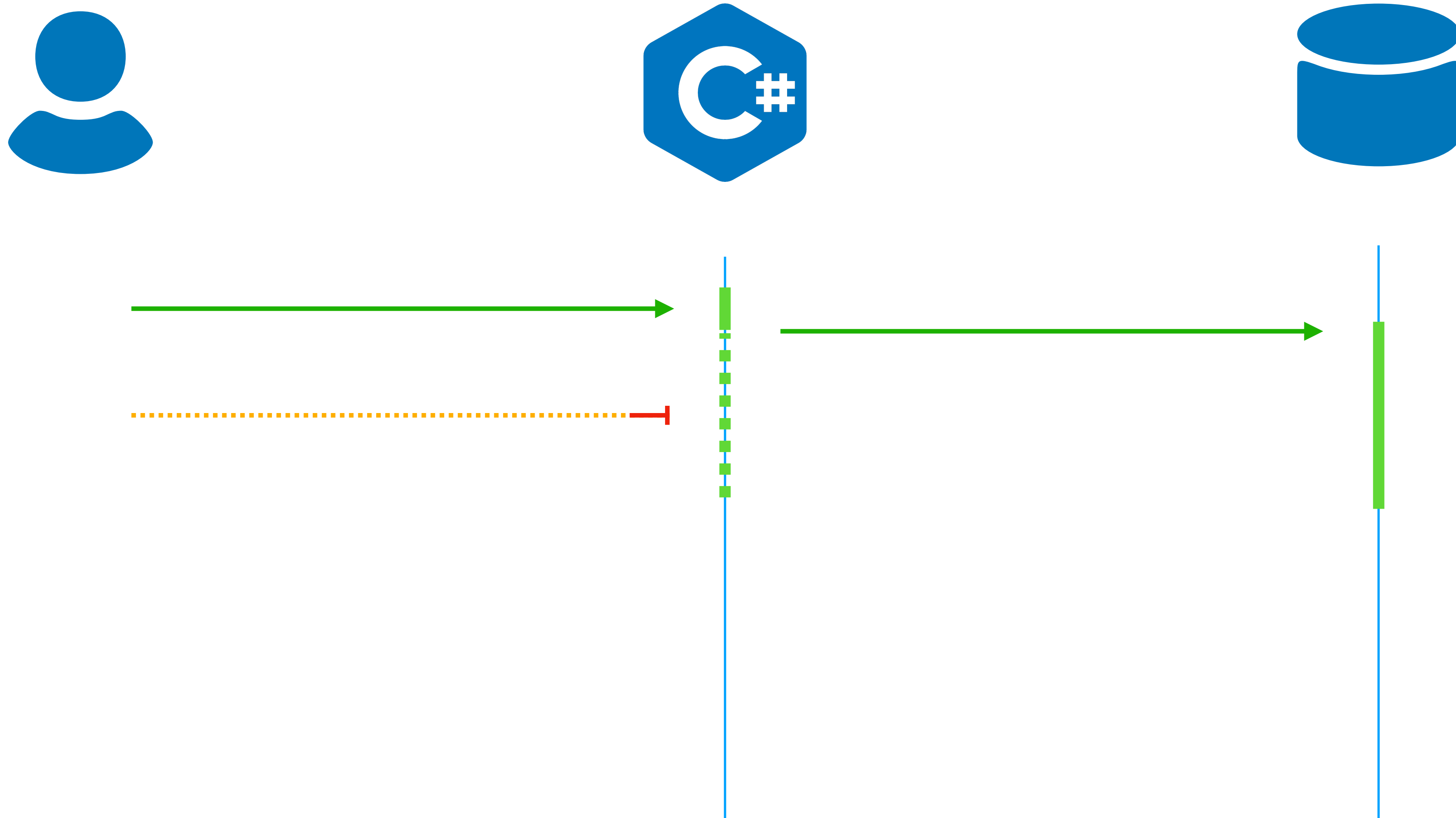
асинхронное программирование в .NET

блокирующие i/o операции



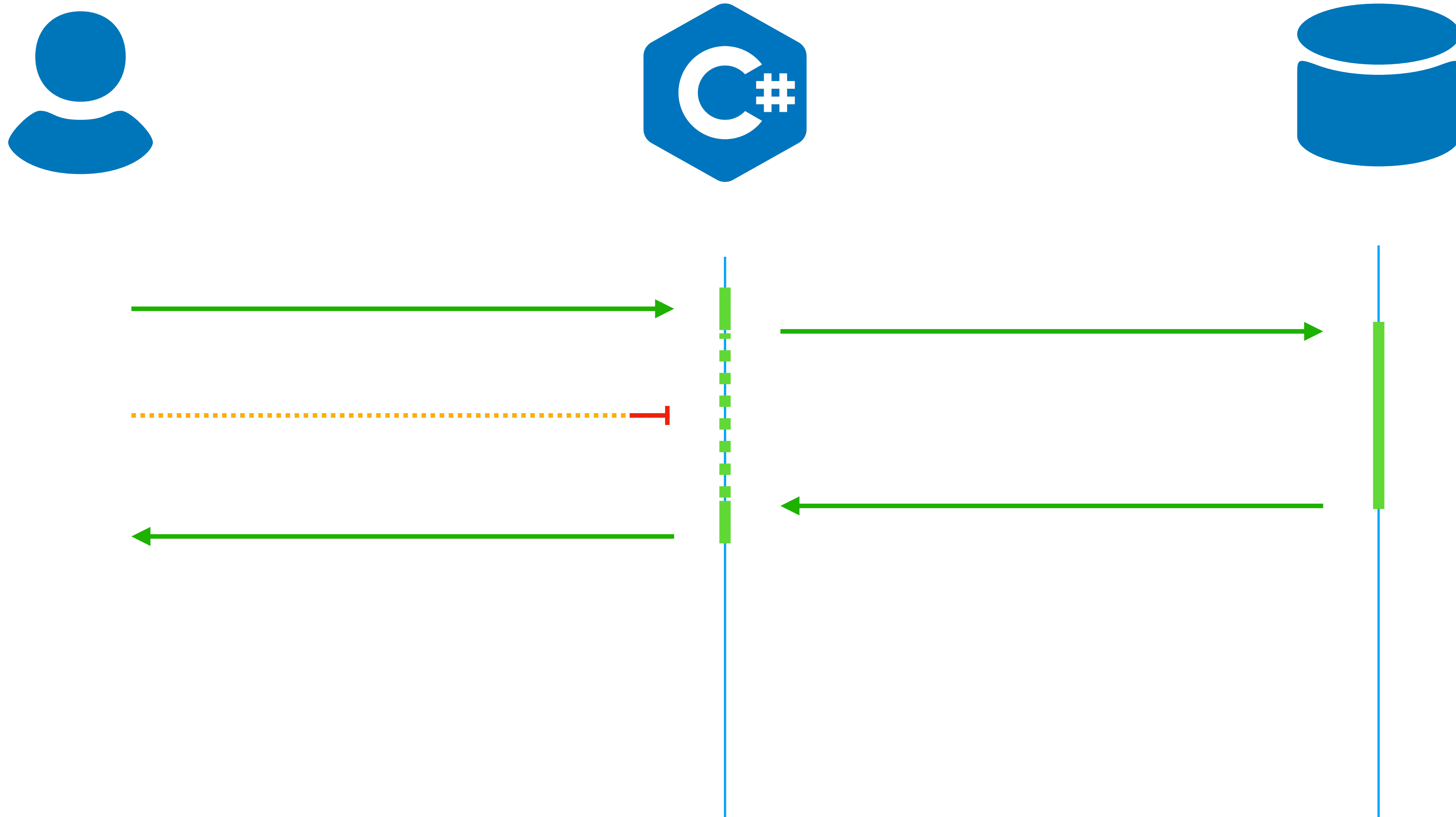
асинхронное программирование в .NET

блокирующие i/o операции



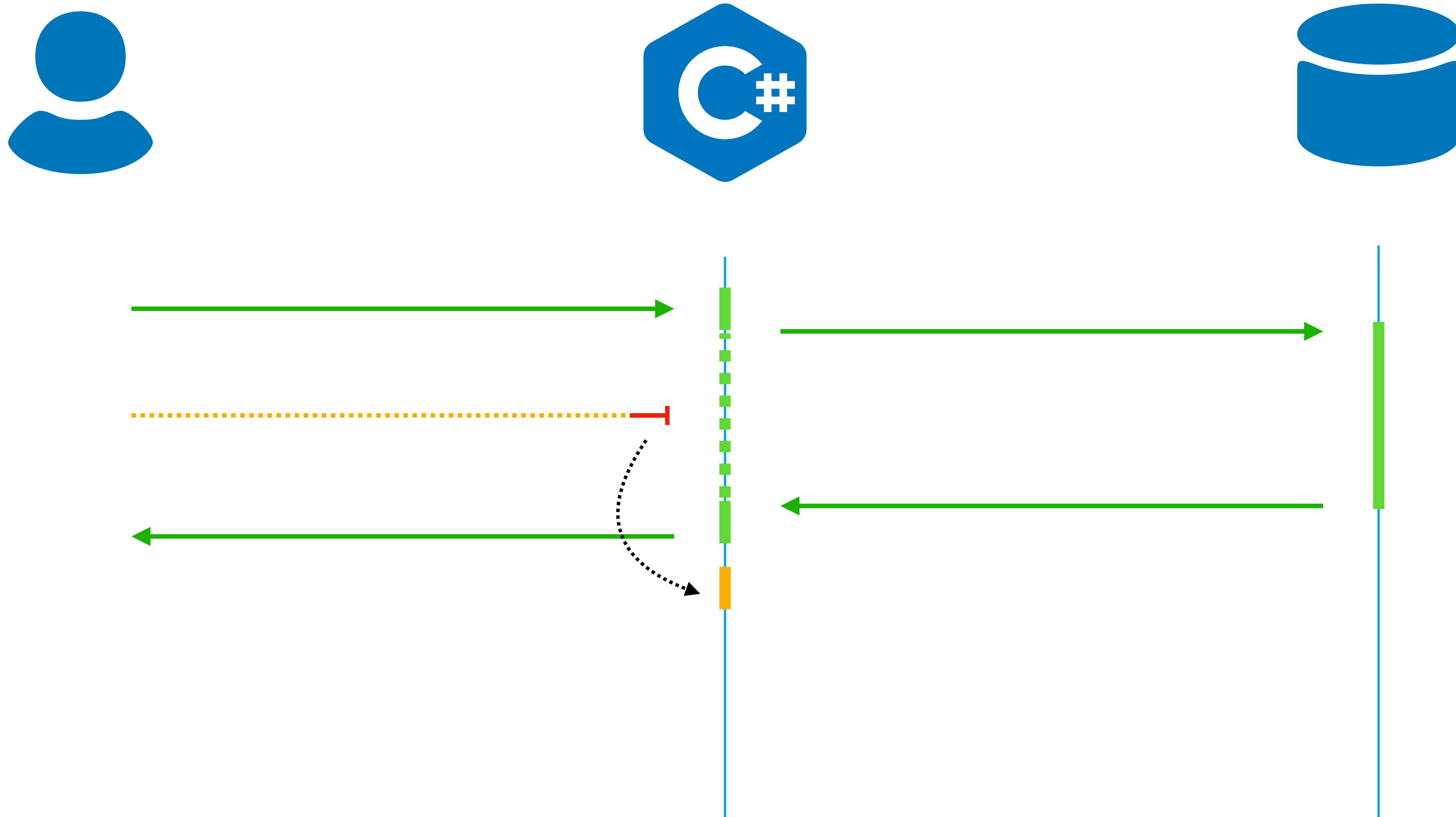
асинхронное программирование в .NET

блокирующие i/o операции



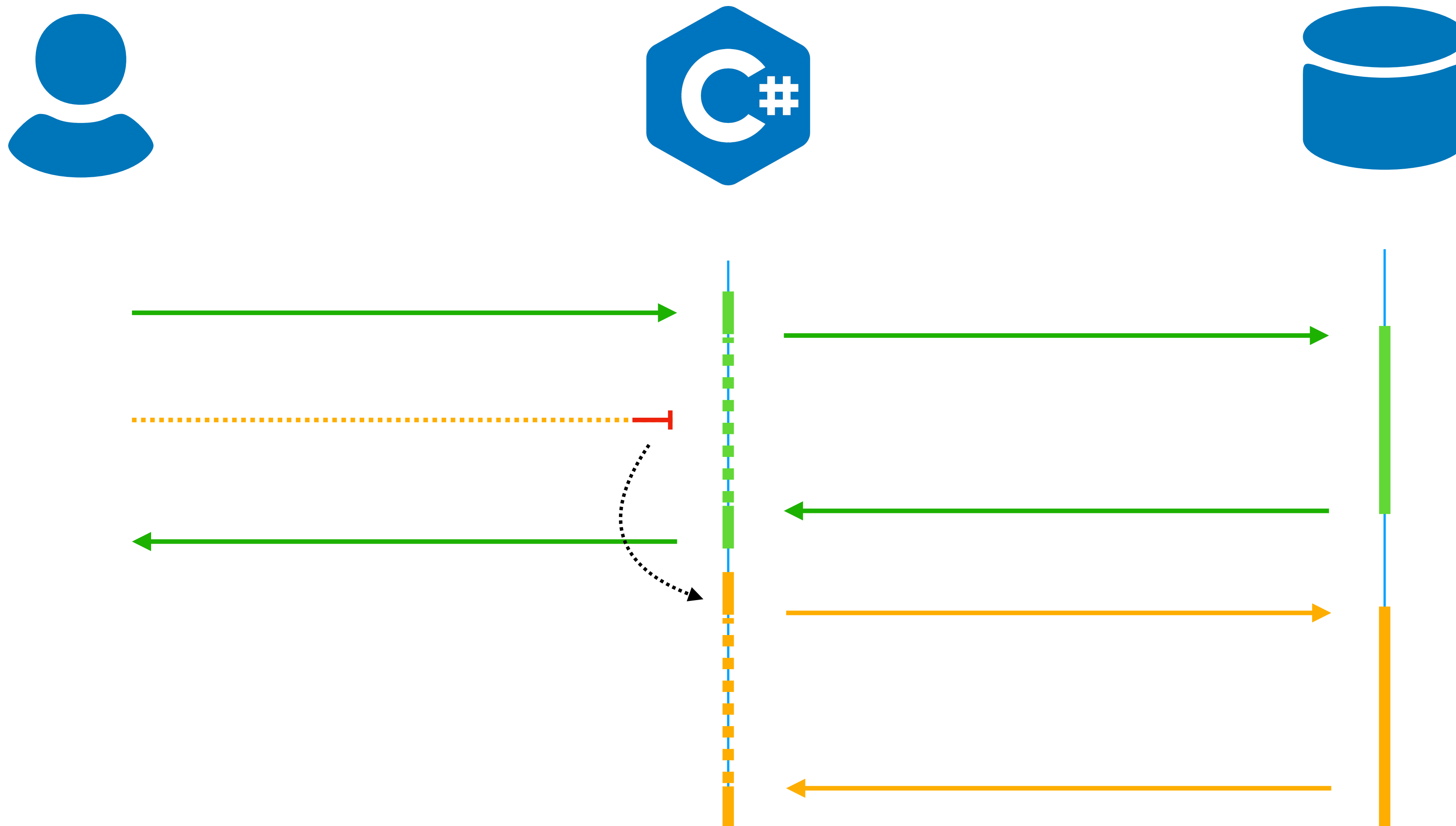
асинхронное программирование в .NET

блокирующие i/o операции



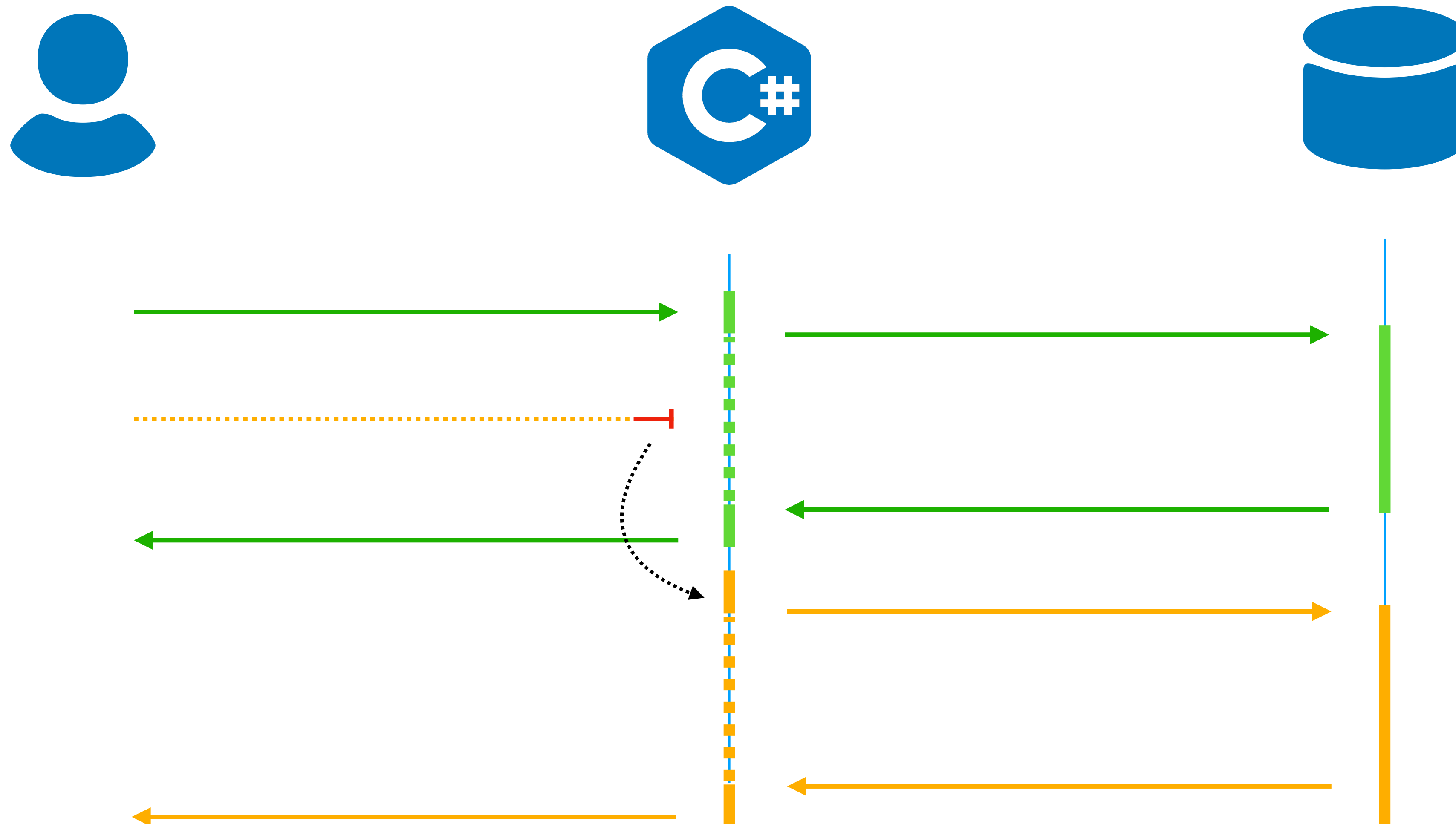
асинхронное программирование в .NET

блокирующие i/o операции



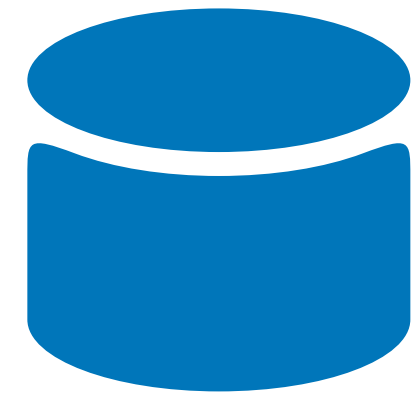
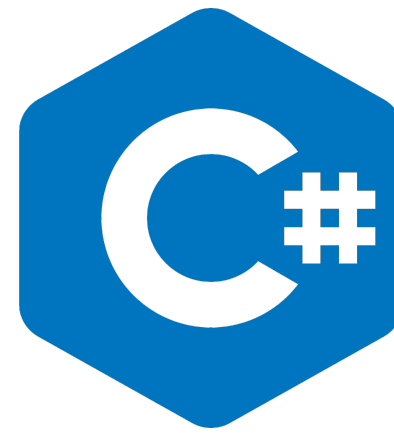
асинхронное программирование в .NET

блокирующие i/o операции



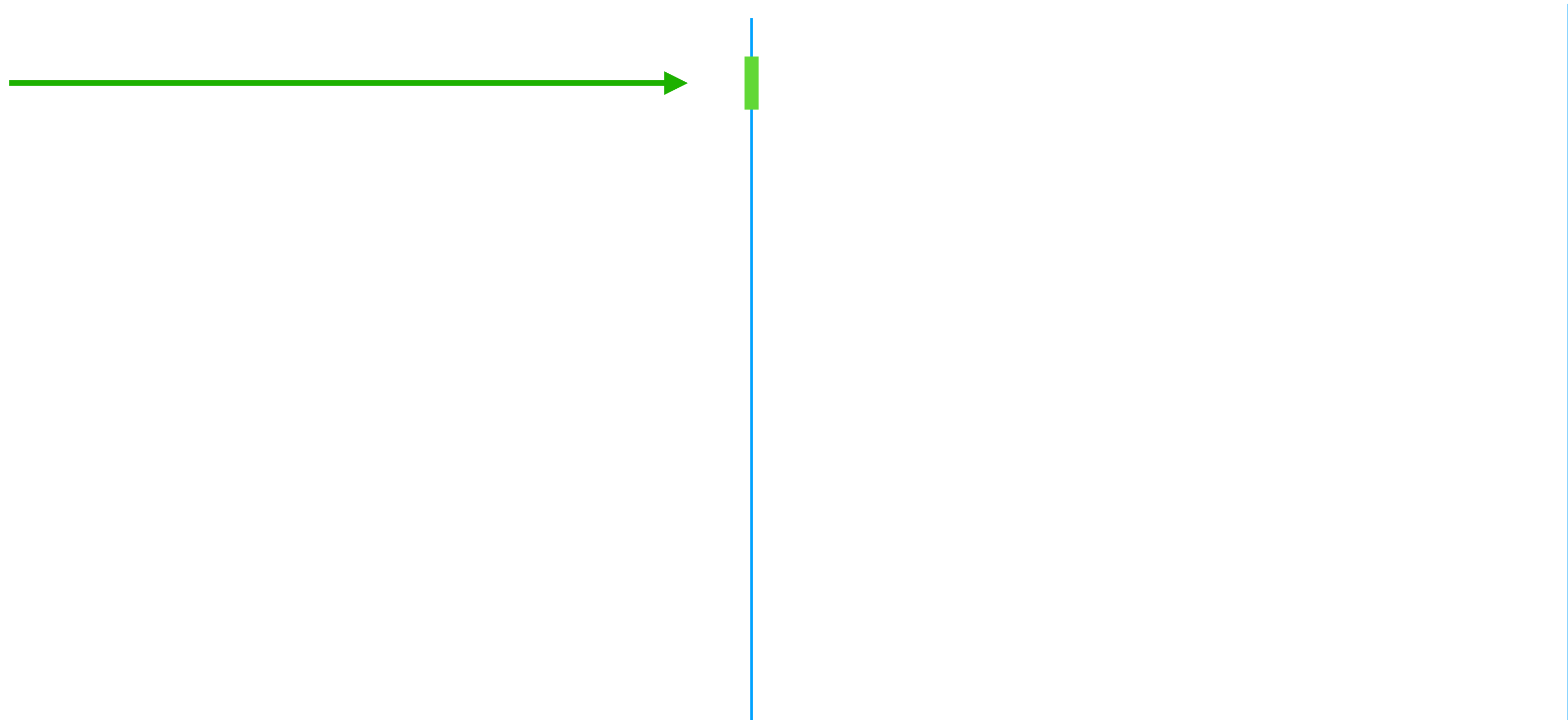
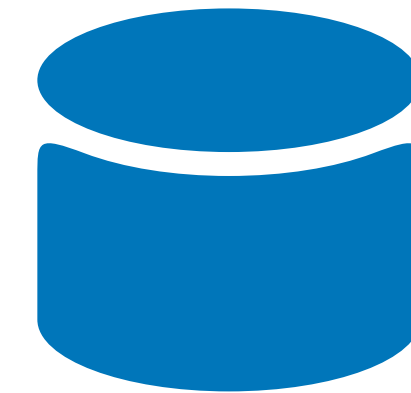
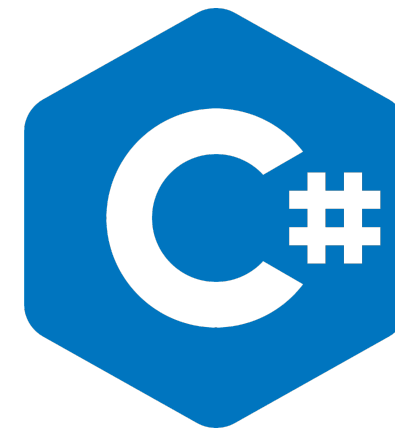
асинхронное программирование в .NET

асинхронные i/o операции



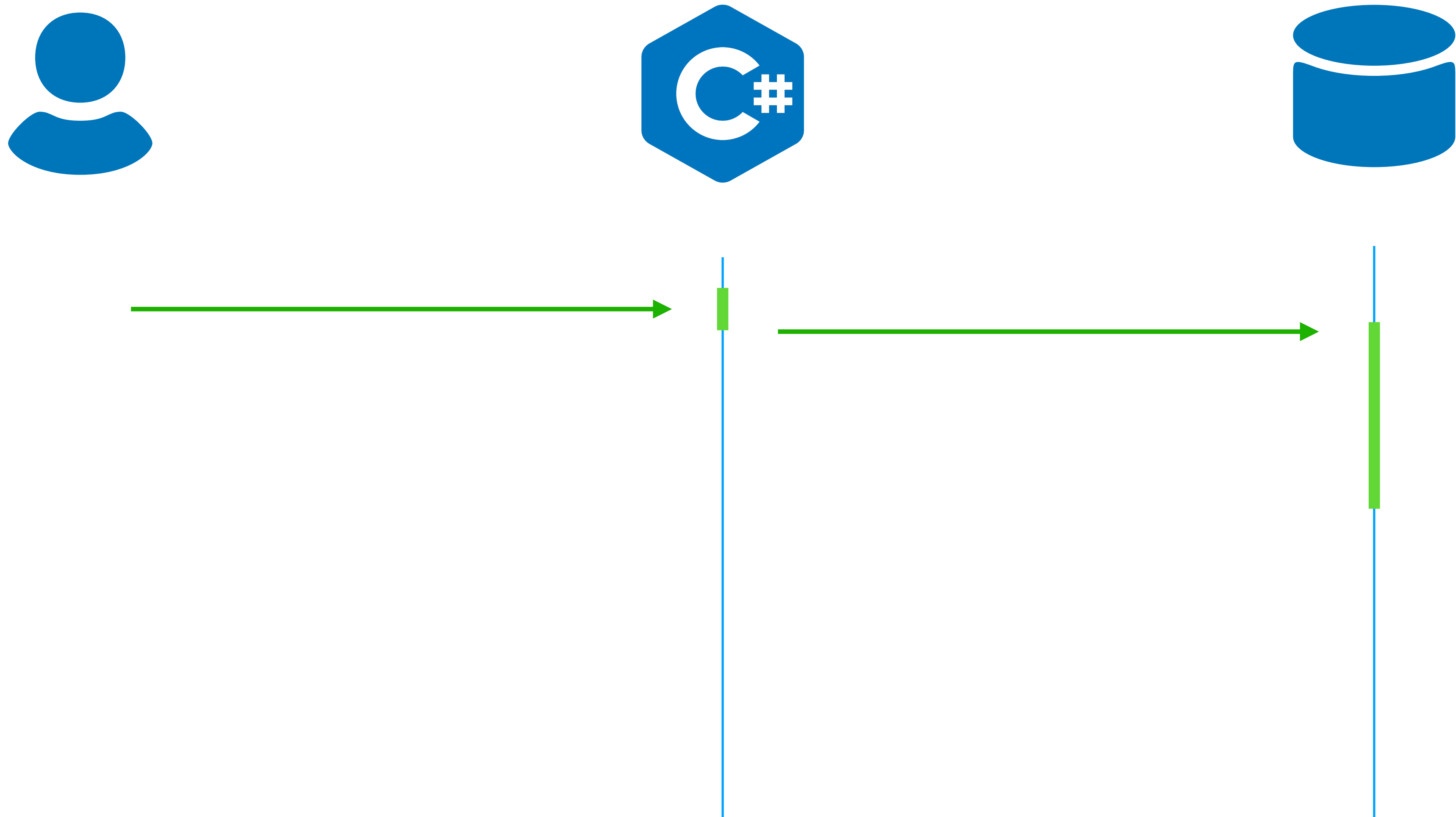
асинхронное программирование в .NET

асинхронные i/o операции



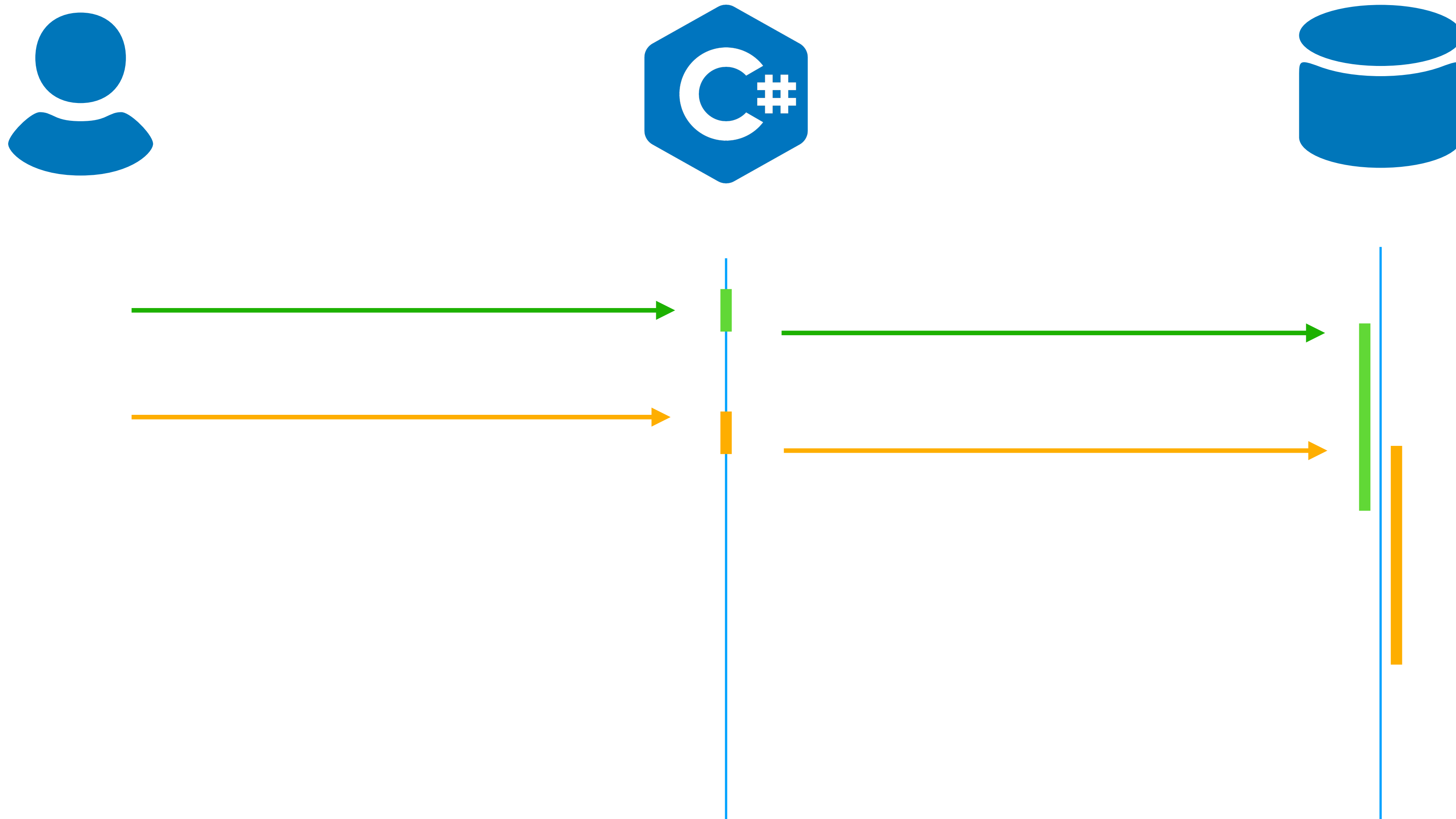
асинхронное программирование в .NET

асинхронные i/o операции



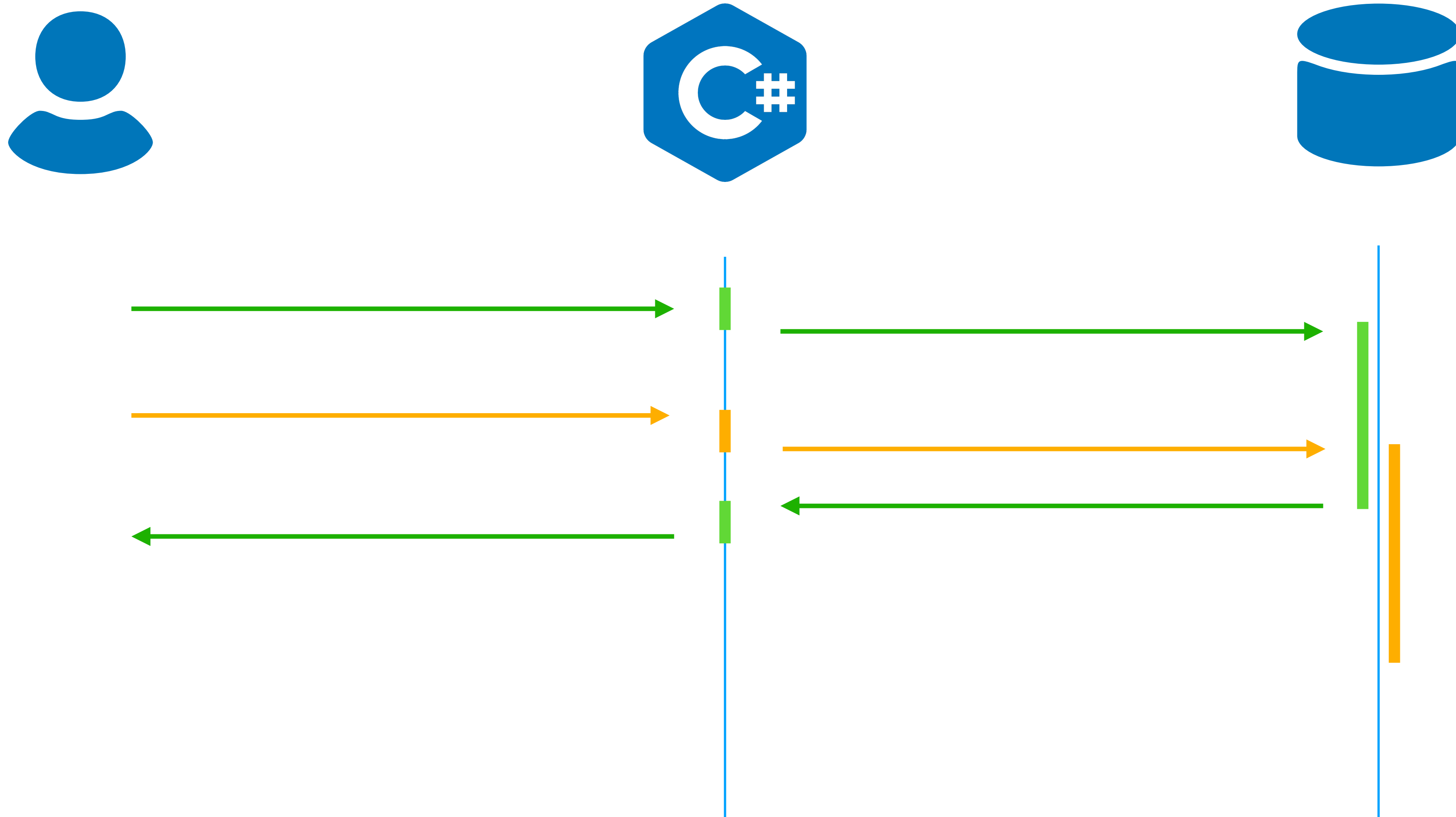
асинхронное программирование в .NET

асинхронные i/o операции



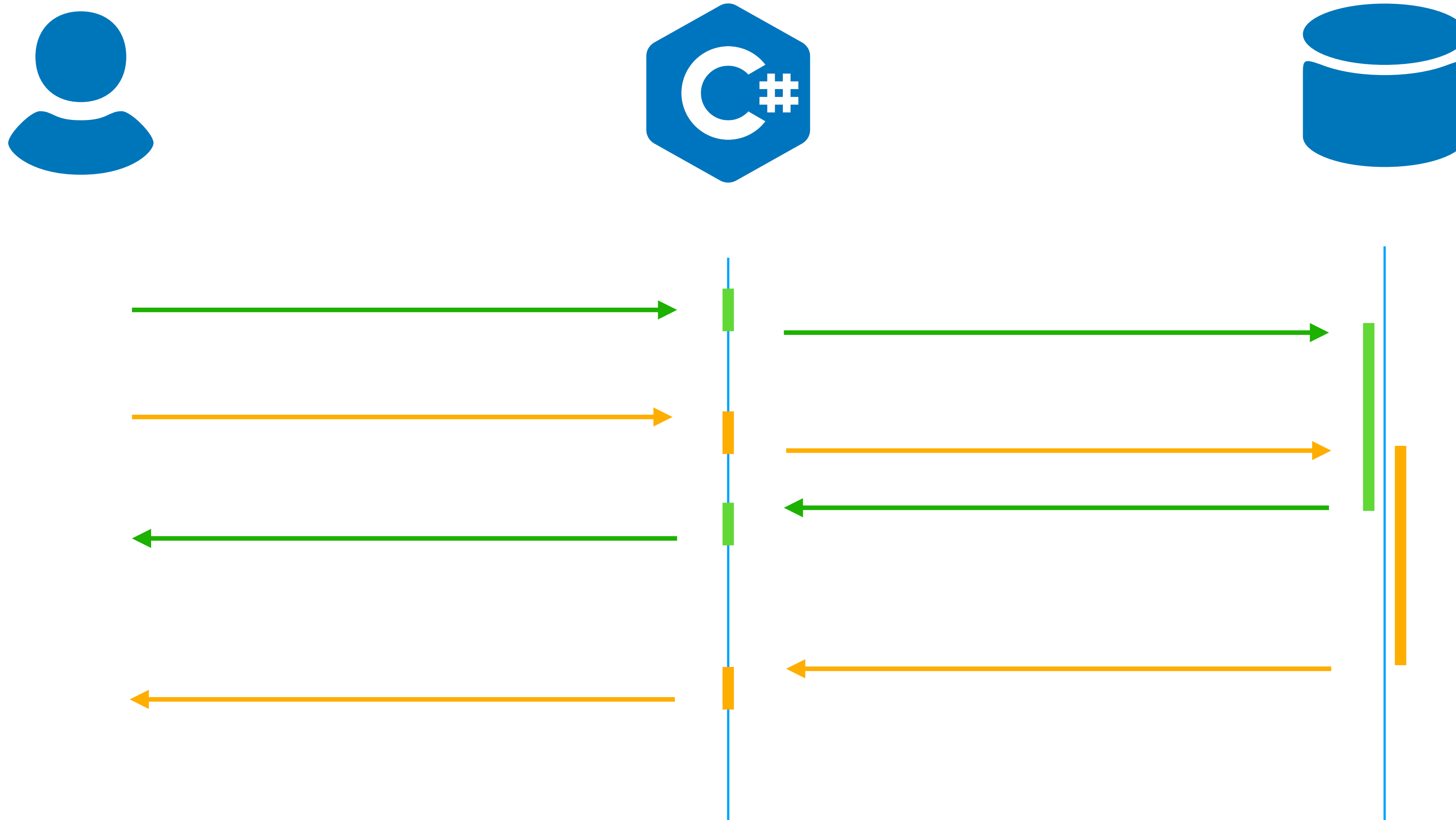
асинхронное программирование в .NET

асинхронные i/o операции



асинхронное программирование в .NET

асинхронные i/o операции



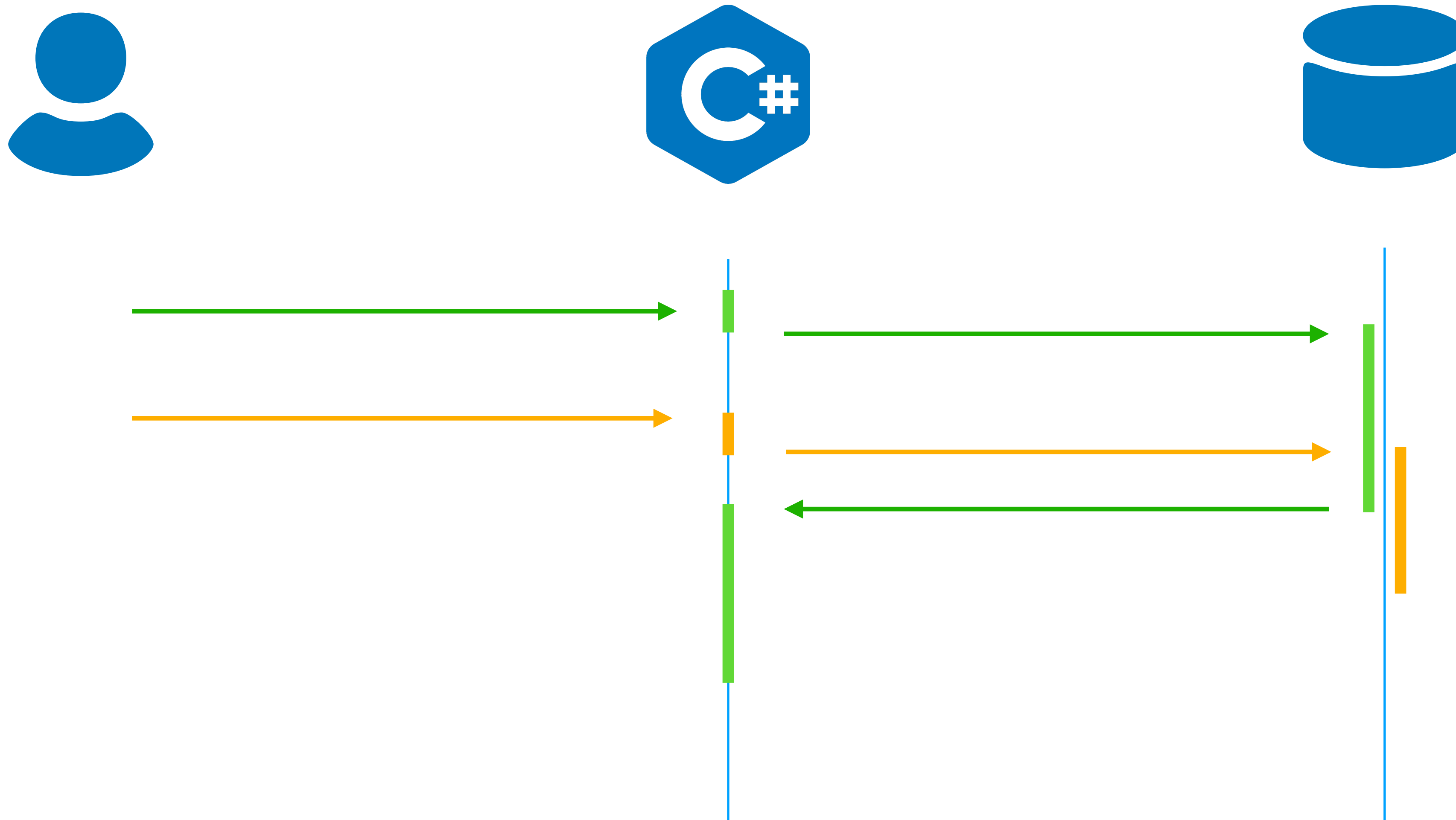
асинхронное программирование в .NET

асинхронные i/o операции

- позволяет потоку выполнять какую-либо работу во время ожидания устройств ввода/вывода
- позволяет сократить общее время выполнения операций, за счёт отсутствия блокировок для ожидания
- не ускоряют выполнение кода, но оптимизируют использование ресурсов

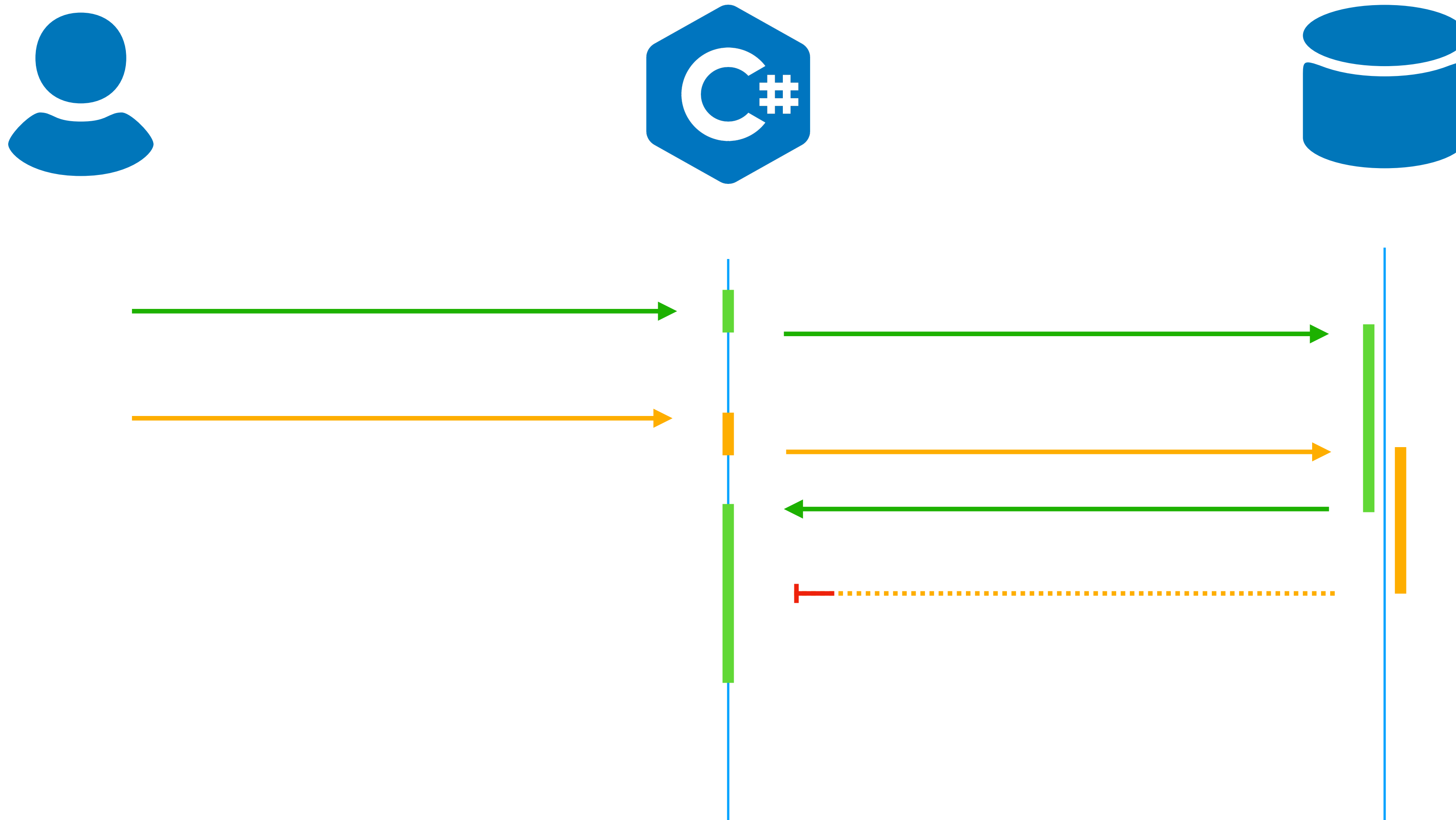
асинхронное программирование в .NET

асинхронные i/o операции



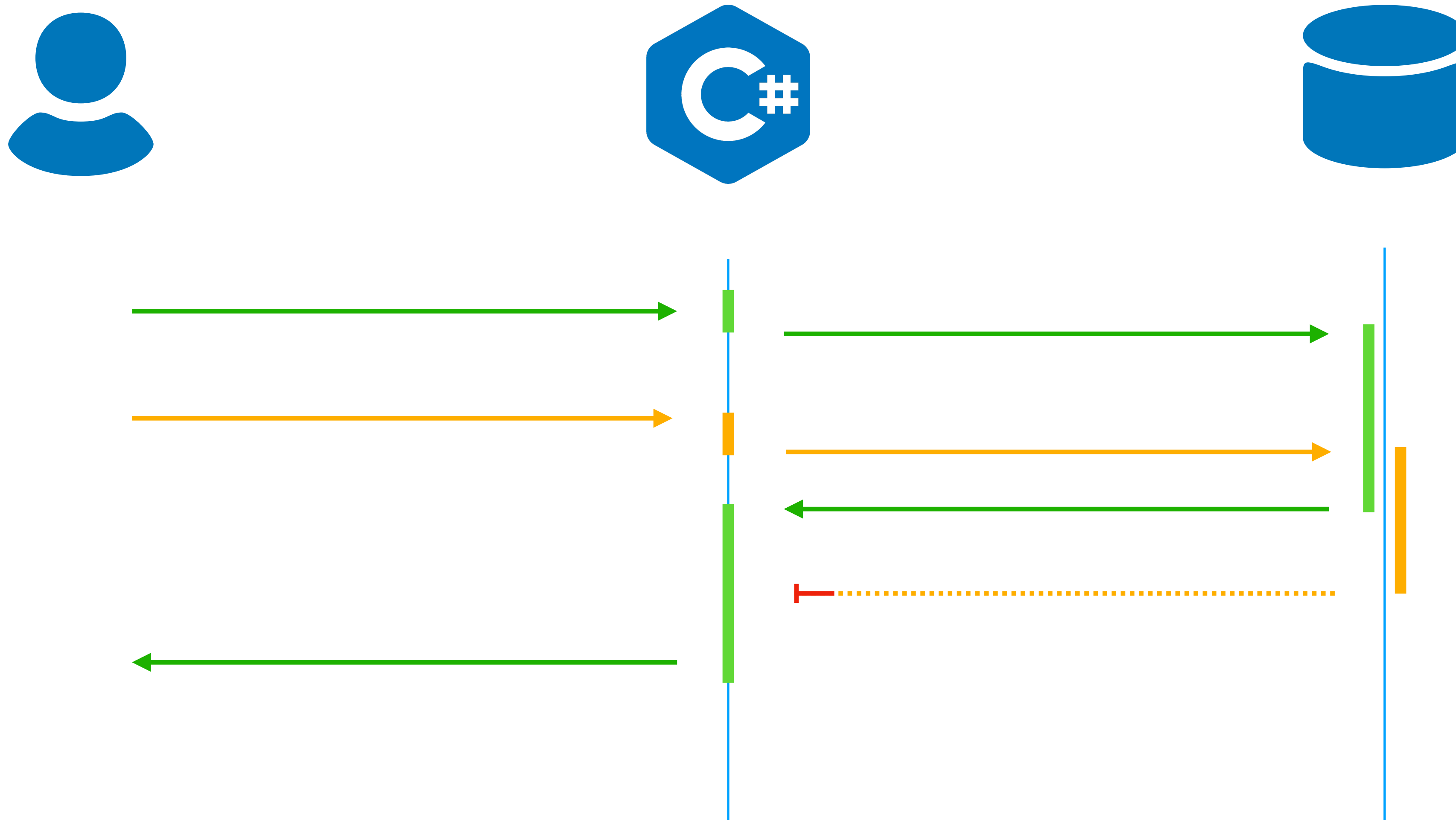
асинхронное программирование в .NET

асинхронные i/o операции



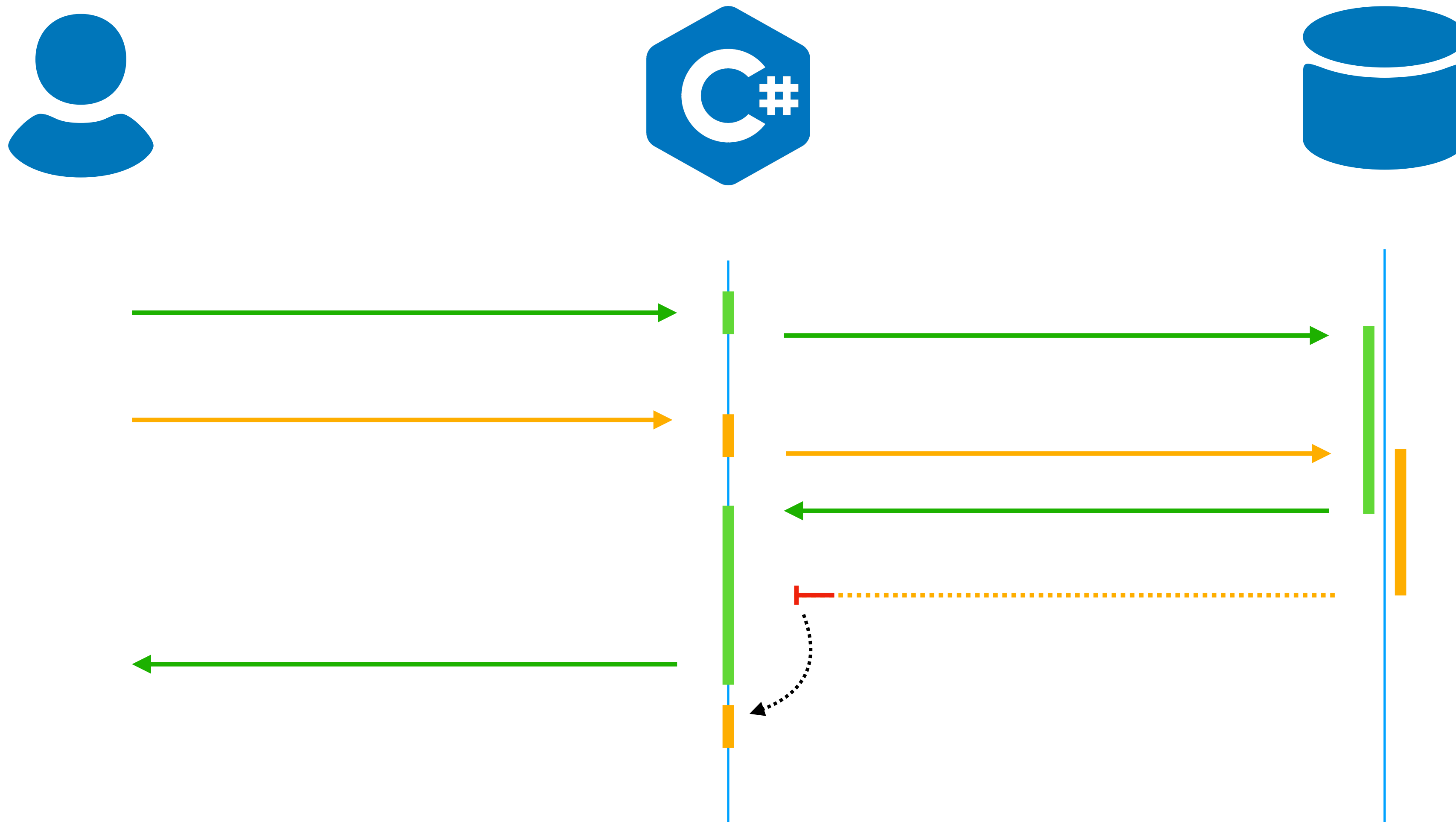
асинхронное программирование в .NET

асинхронные i/o операции



асинхронное программирование в .NET

асинхронные i/o операции



асинхронное программирование в .NET

ThreadPool

- какой-то заранее выделенный набор потоков
- обрабатывают асинхронные операции по мере их готовности

```
ThreadPool.QueueUserWorkItem(_ => Console.WriteLine("Hello from thread pool!"));
```

асинхронное программирование в .NET Task

- представляют асинхронную операцию
- позволяют получить TaskAwaiter – представляющий результат операции

асинхронное программирование в .NET

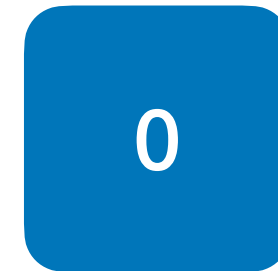
async/await

```
public async Task DoSomethingAsync()
{
    var data = await GetDataAsync();
    var modifiedData = data.ToUpper();

    await SendDataAsync(modifiedData);
}
```

асинхронное программирование в .NET

async/await



```
public async Task DoSomethingAsync()
{
    var data = await GetDataAsync();
    var modifiedData = data.ToUpper();

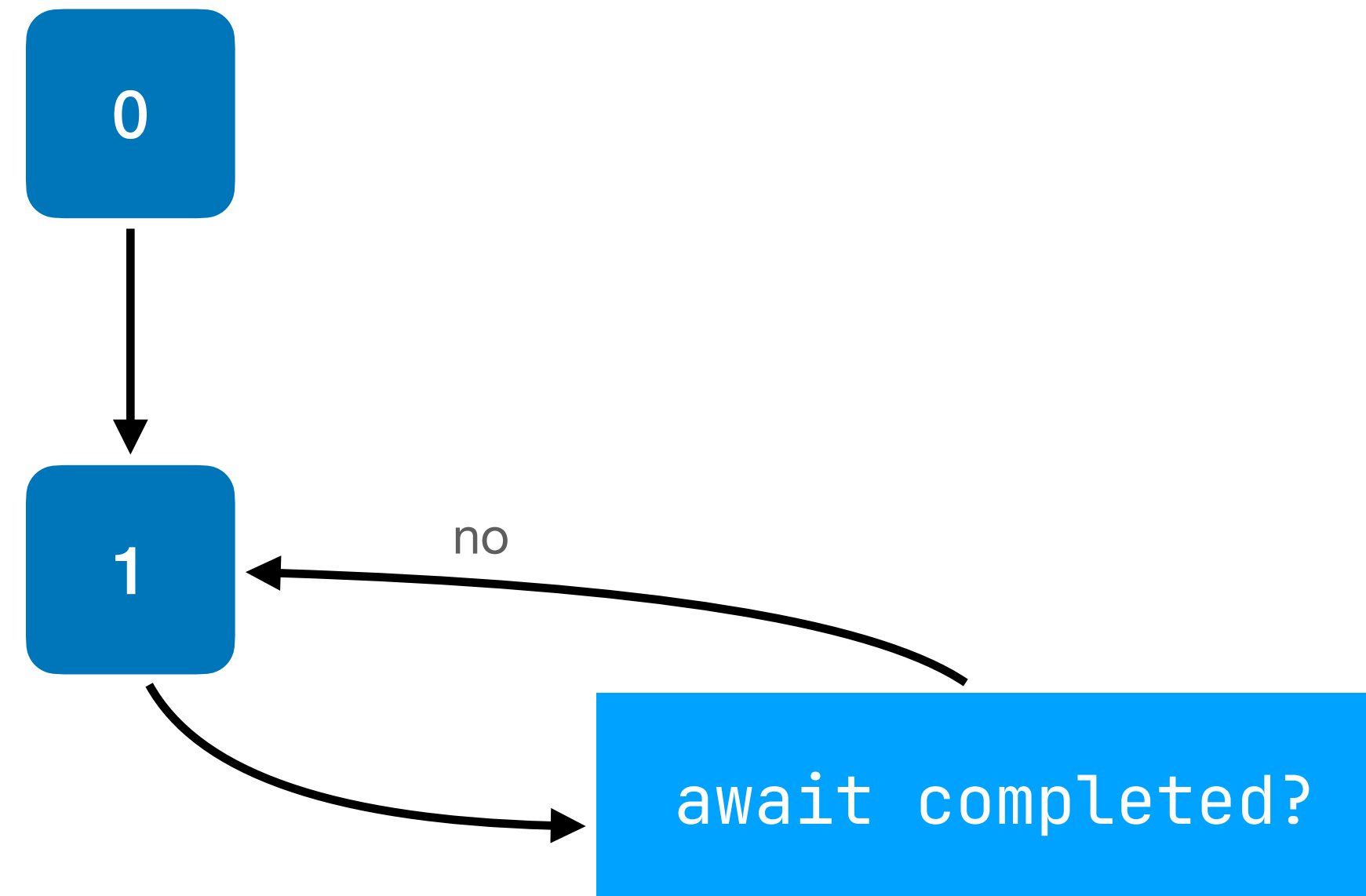
    await SendDataAsync(modifiedData);
}
```

асинхронное программирование в .NET

async/await

```
public async Task DoSomethingAsync()
{
    var data = await GetDataAsync();
    var modifiedData = data.ToUpper();

    await SendDataAsync(modifiedData);
}
```

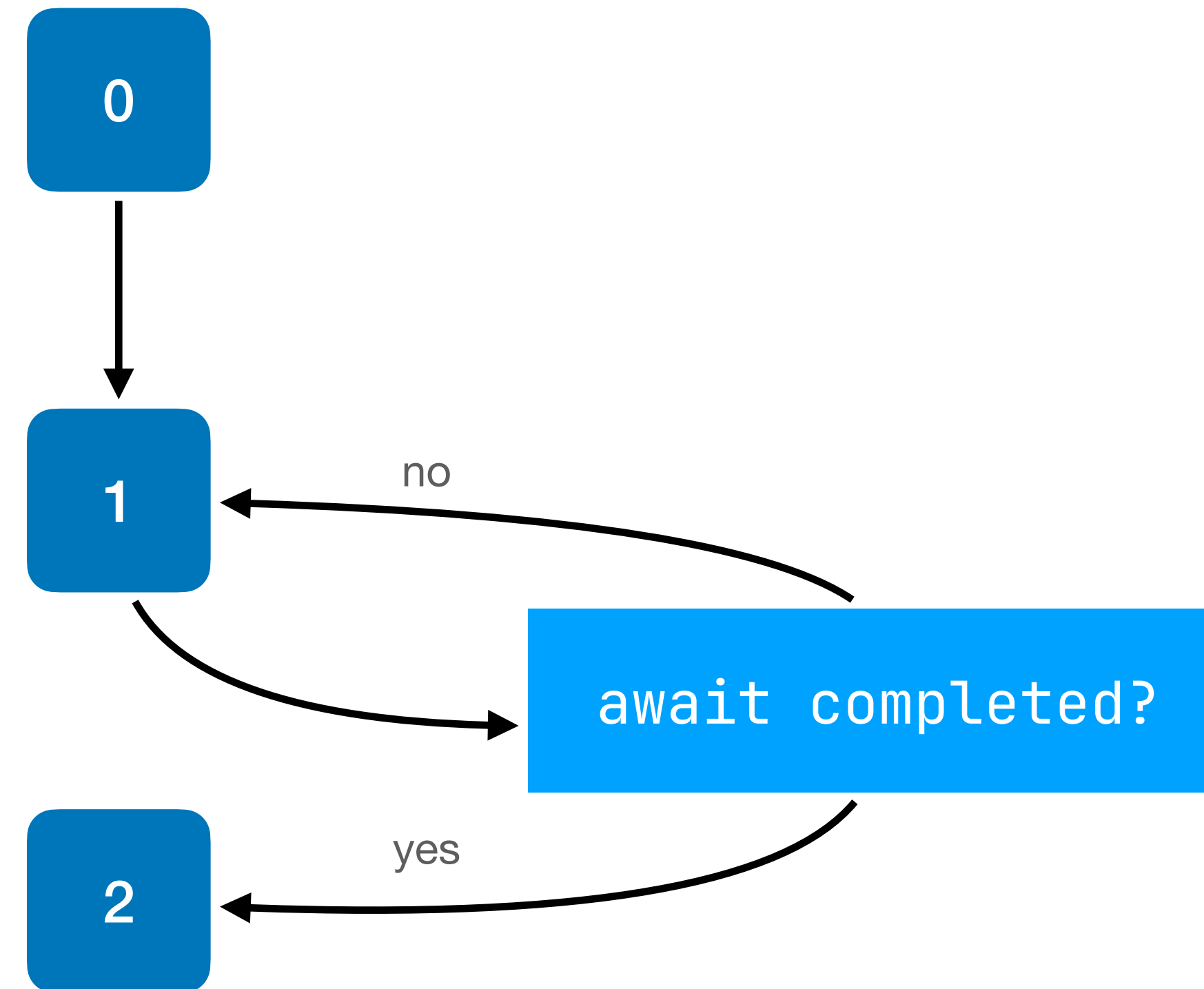


асинхронное программирование в .NET

async/await

```
public async Task DoSomethingAsync()
{
    var data = await GetDataAsync();
    var modifiedData = data.ToUpper();

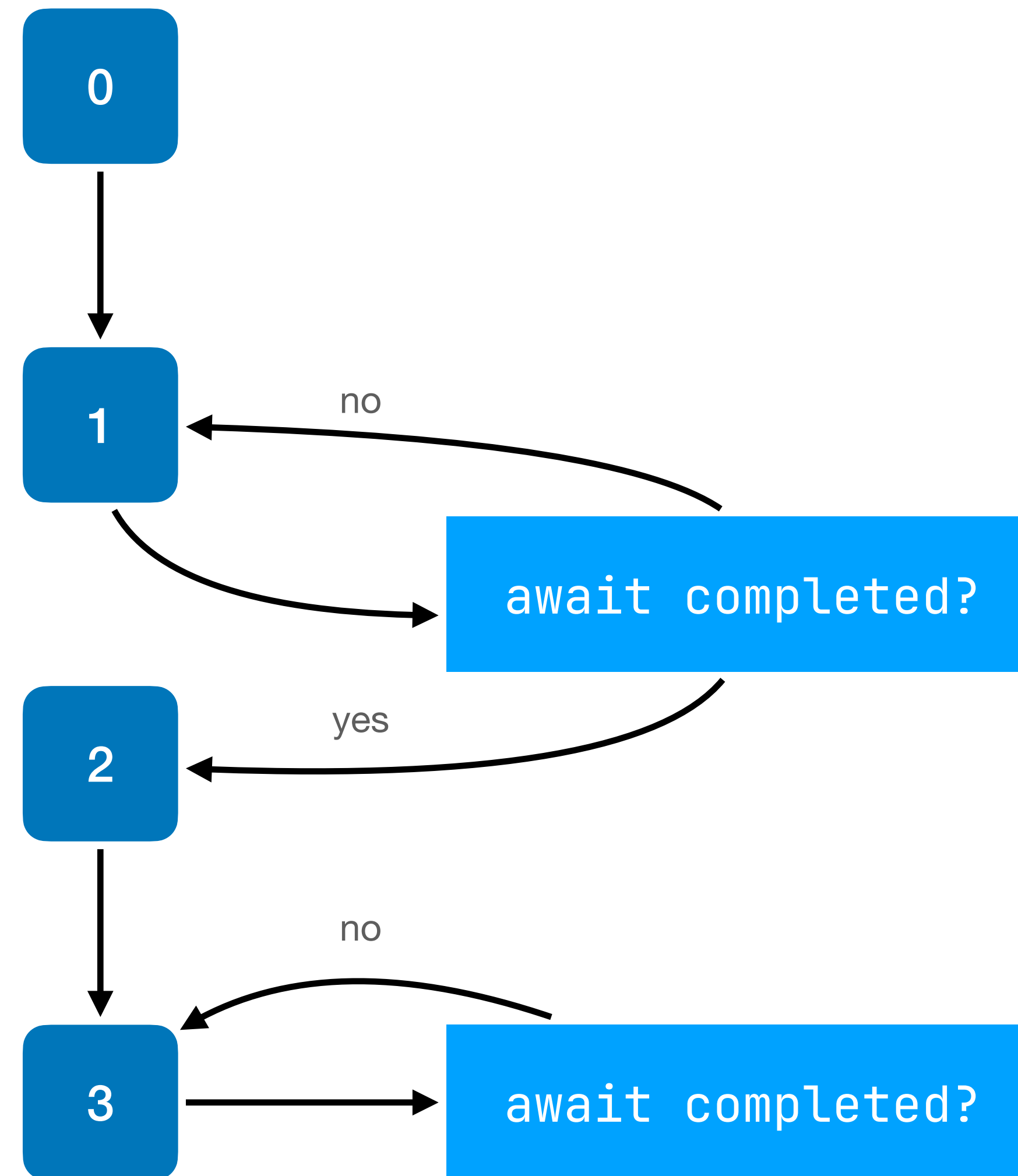
    await SendDataAsync(modifiedData);
}
```



асинхронное программирование в .NET

async/await

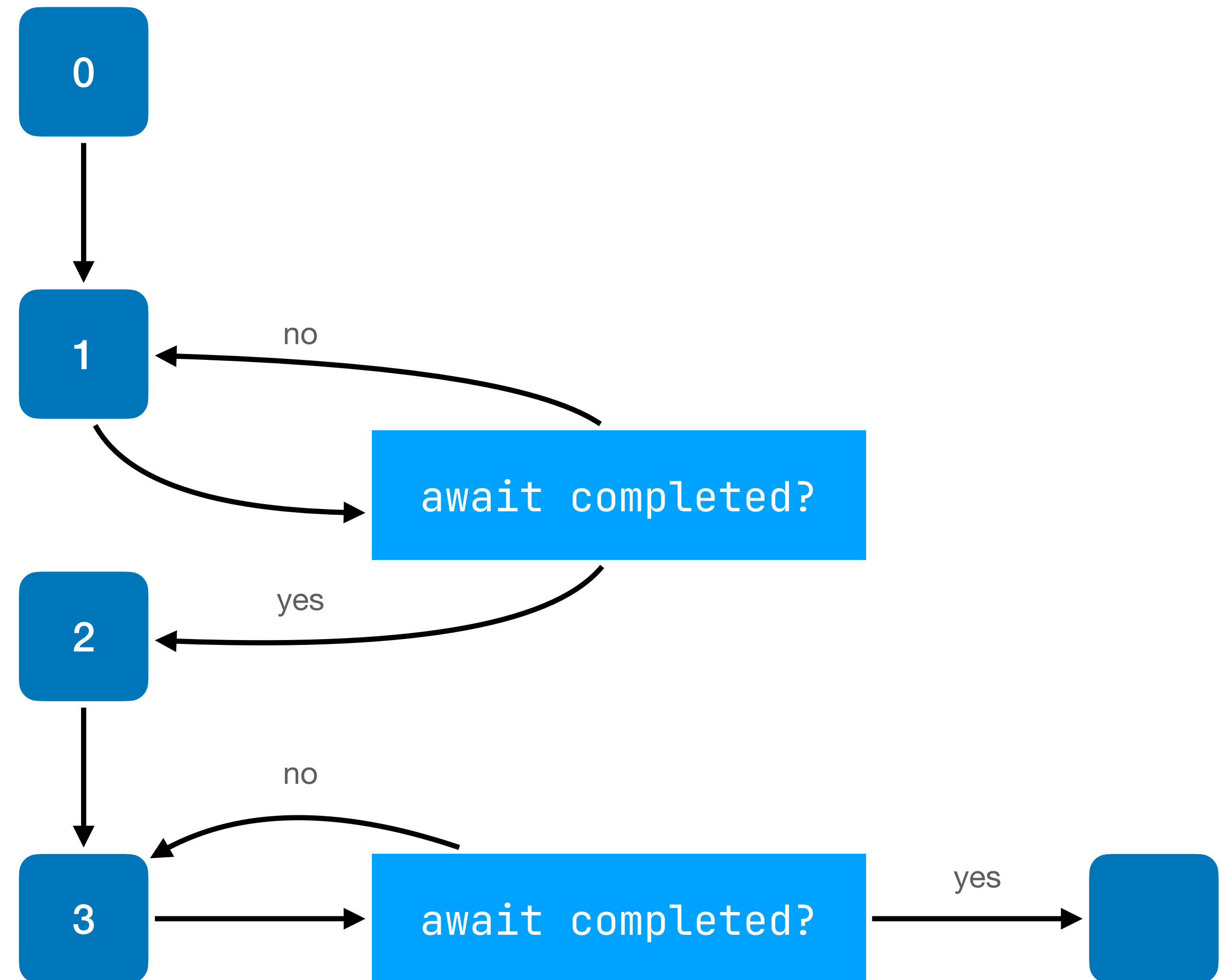
```
public async Task DoSomethingAsync()  
{  
    var data = await GetDataAsync();  
    var modifiedData = data.ToUpper();  
  
    await SendDataAsync(modifiedData);  
}
```



асинхронное программирование в .NET

async/await

```
public async Task DoSomethingAsync()  
{  
    var data = await GetDataAsync();  
    var modifiedData = data.ToUpper();  
  
    await SendDataAsync(modifiedData);  
}
```



асинхронное программирование в .NET

TaskCompletionSource

- позволяет вручную управлять циклом жизни Task, получаемой через него
- позволяет задать Task результат, ошибку, отмену

асинхронное программирование в .NET

TaskCompletionSource

```
public Task<string> GetFileContentAsync(string fileName)
{
    var tcs = new TaskCompletionSource<string>();
    CallOsFileLoad(fileName: fileName, onCompleted: value => tcs.SetResult(value));

    return tcs.Task;
}
```


асинхронное программирование в .NET

CancellationToken

```
public Task<string> GetFileContentAsync(string fileName, CancellationToken cancellationToken)
{
    var tcs = new TaskCompletionSource<string>();

    CancellationTokenRegistration cancellationRegistration = cancellationToken.Register(
        () => tcs.SetCanceled(cancellationToken));

    CallOsFileLoad(
        fileName: fileName,
        onCompleted: value =>
        {
            tcs.SetResult(value);
            cancellationRegistration.Dispose();
        },
        cancellationToken: cancellationToken);

    return tcs.Task;
}
```

асинхронное программирование в .NET

CancellationToken

```
public Task<string> GetFileContentWithTimeoutAsync(
    string fileName,
    TimeSpan timeout,
    CancellationToken cancellationToken)
{
    var timeoutTokenSource = new CancellationTokenSource();
    timeoutTokenSource.CancelAfter(timeout);

    var linkedTokenSource = CancellationTokenSource.CreateLinkedTokenSource(
        cancellationToken,
        timeoutTokenSource.Token);

    return GetFileContentAsync(fileName, linkedTokenSource.Token);
}
```

параллельный
асинхронный код

параллельный асинхронный код

Task.When...

Возвращает Task, завершающийся по завершению всех переданных Task

```
Task Task.WhenAll(IEnumerable<Task> tasks)
```

```
Task<TResult[]> Task.WhenAll<TResult>(IEnumerable<Task<TResult>> tasks)
```

Возвращает Task, завершающийся когда завершится одна из переданных Task

```
Task<TTask> Task.WhenAny<TTask>(IEnumerable<TTask> where TTask : Task)
```

параллельный асинхронный код

класс Parallel

```
await Parallel.ForAsync(  
    0, 10,  
    async (i, ct) => await SendNumberAsync(i, ct));
```

```
await Parallel.ForEachAsync(  
    Enumerable.Range(0, 10),  
    async (i, ct) => await SendNumberAsync(i, ct));
```

параллельный асинхронный код

асинхронные примитивы синхронизации

```
class AsyncExecutor
{
    private readonly SemaphoreSlim _semaphore = new(initialCount: 1, maxCount: 1);

    public async Task ExecuteExclusiveAsync()
    {
        await _semaphore.WaitAsync();

        try
        {
            // some critical section
        }
        finally
        {
            _semaphore.Release();
        }
    }
}
```

**IDisposable,
IAsyncDisposable, ValueTask**

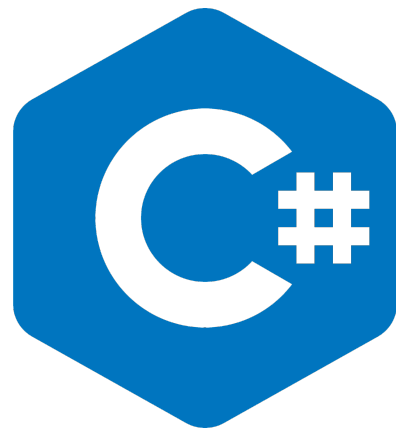
dispose pattern

пример блокирования файла

```
WriteToFileNoDispose("file.txt", "content1");  
WriteToFileNoDispose("file.txt", "content2");  
  
void WriteToFileNoDispose(string fileName, string content)  
{  
    FileStream file = File.OpenWrite(fileName);  
    file.Write(Encoding.Default.GetBytes(content));  
}
```


dispose pattern

пример блокирования файла

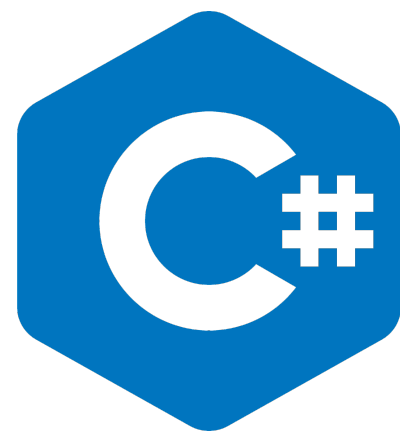


GC

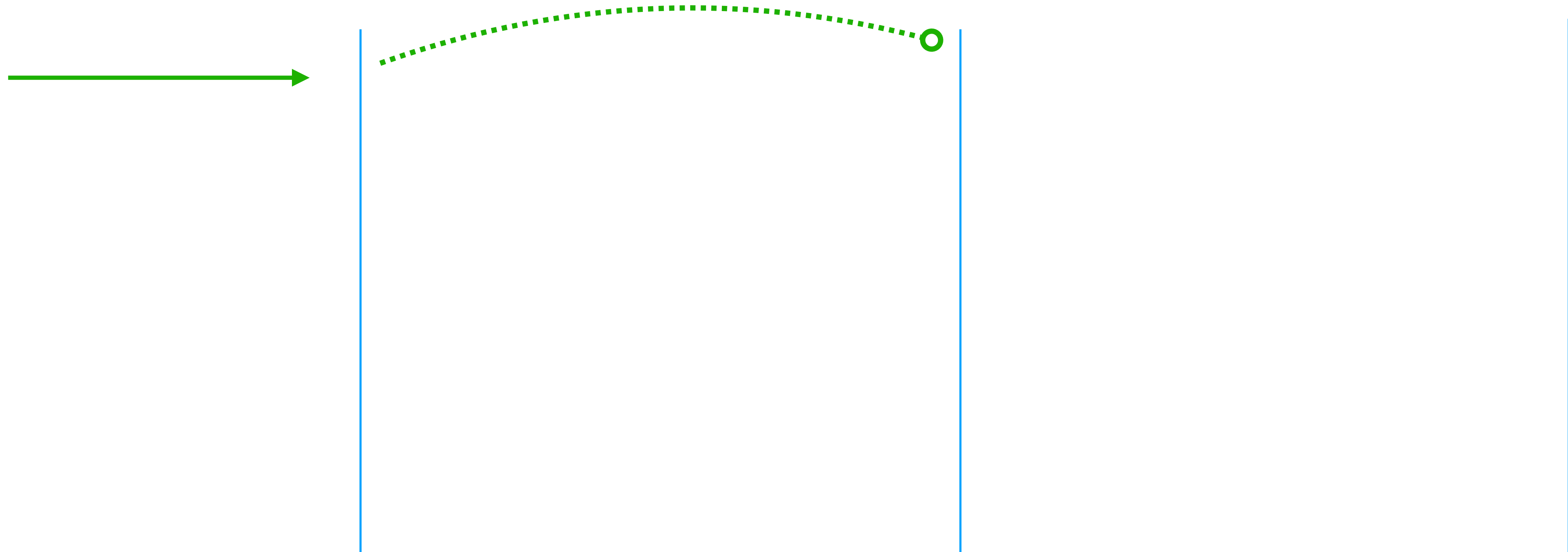


dispose pattern

пример блокирования файла

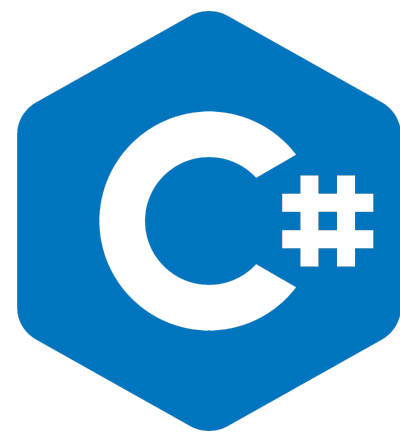


GC

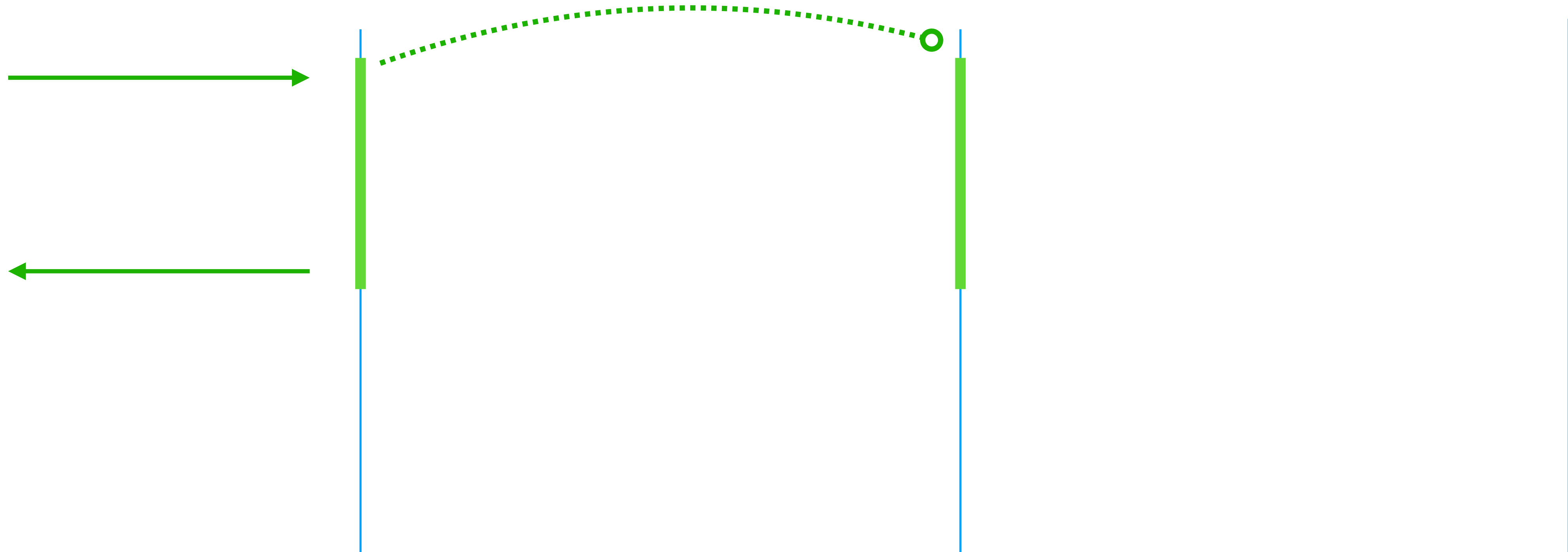


dispose pattern

пример блокирования файла

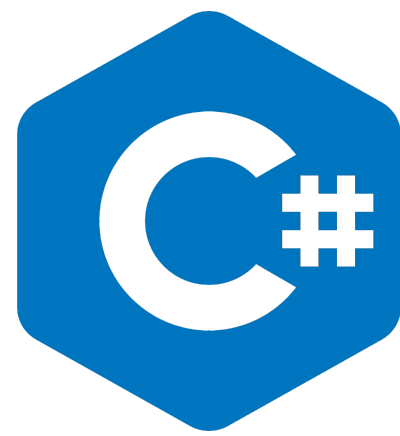


GC

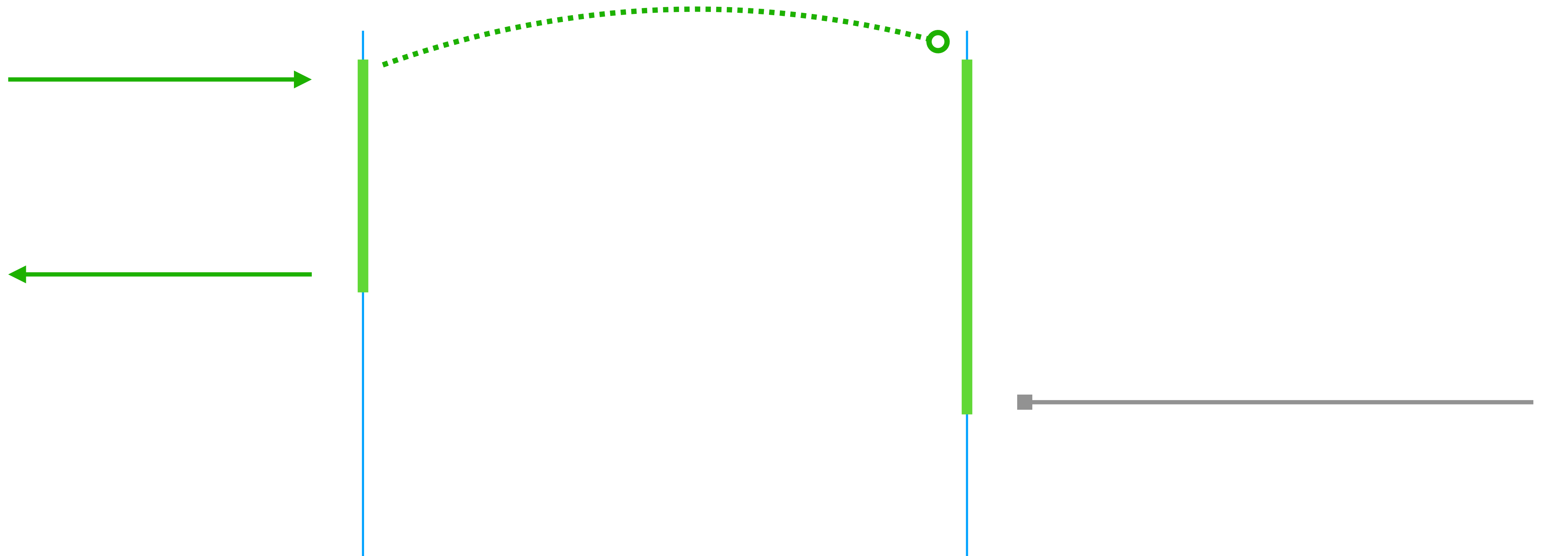


dispose pattern

пример блокирования файла

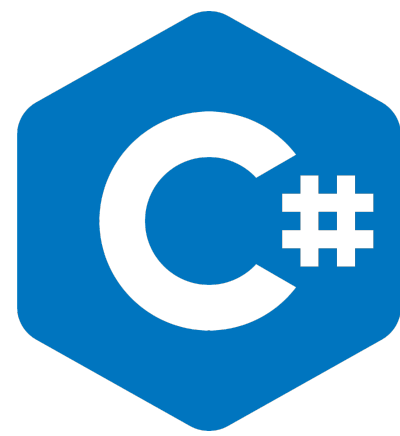


GC

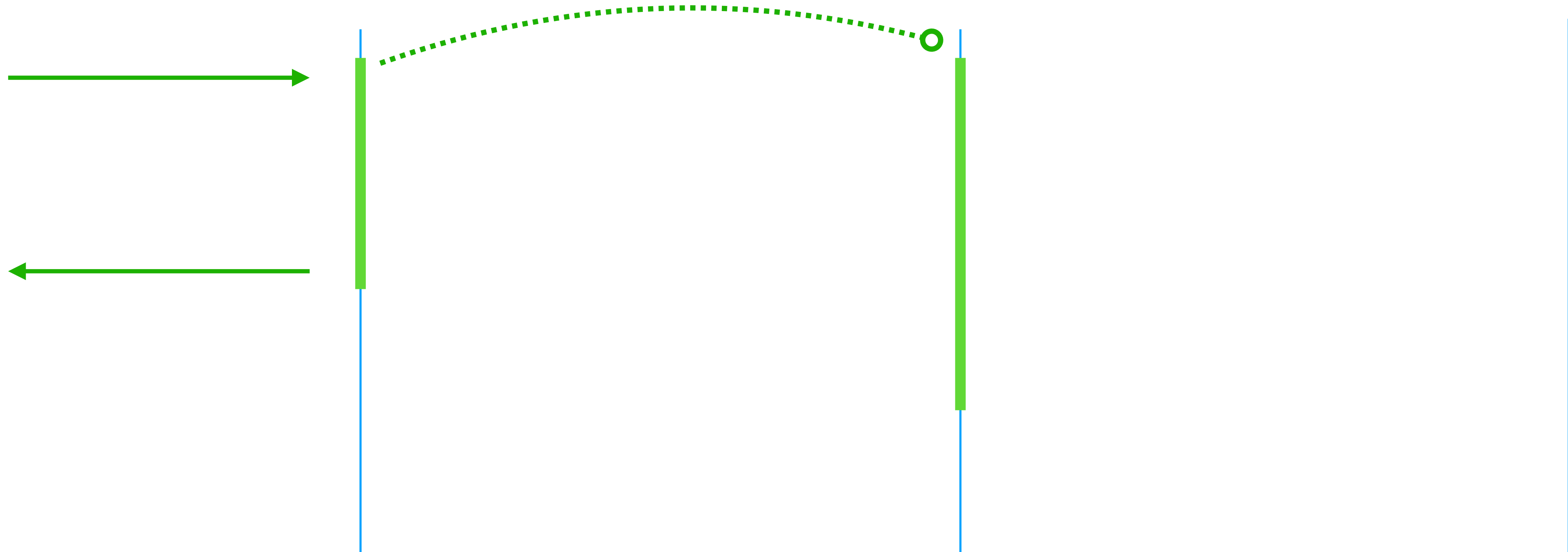


dispose pattern

пример блокирования файла

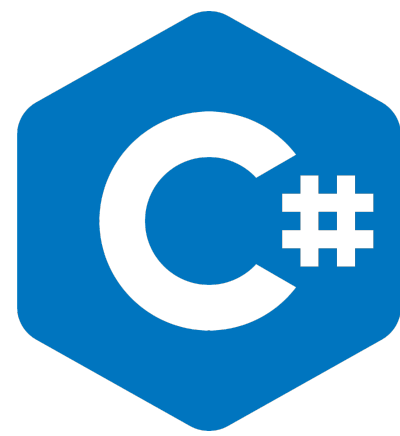


GC

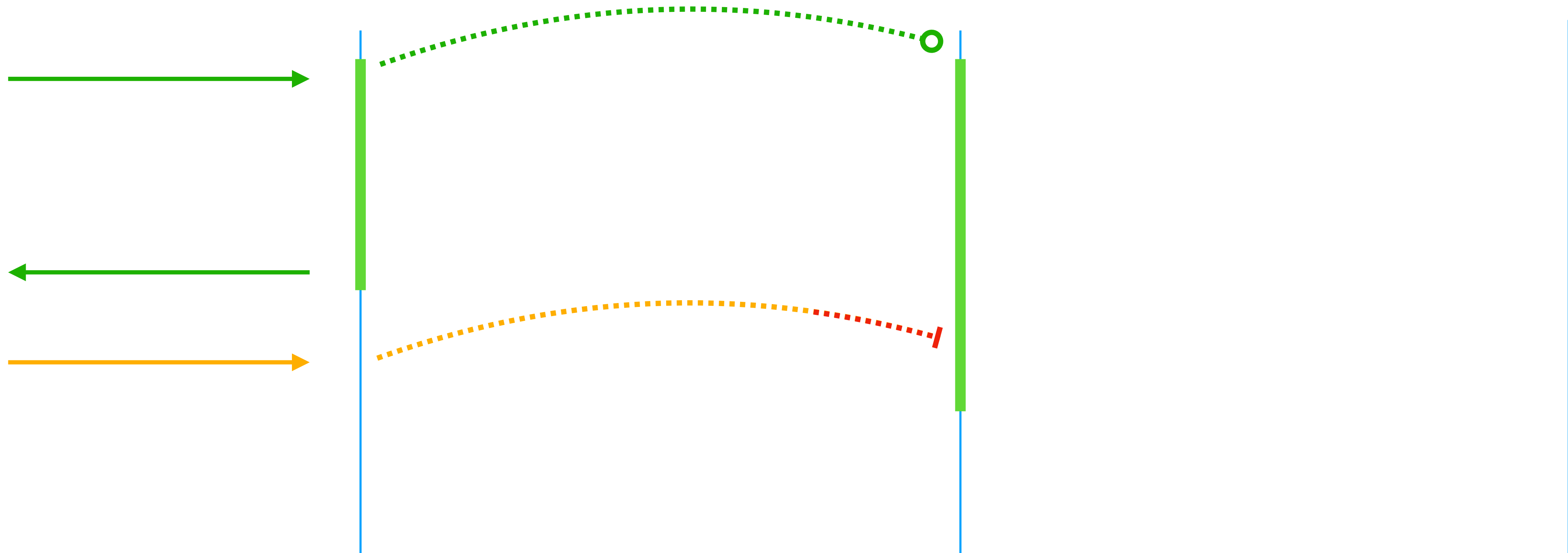


dispose pattern

пример блокирования файла



GC



dispose pattern

пример блокирования файла

```
WriteToFileNoDispose("file.txt", content: "content1");  
WriteToFileNoDispose("file.txt", content: "content2");  
  
void WriteToFileNoDispose(string fileName, string content)  
{  
    FileStream file = File.OpenWrite(fileName);  
    file.Write(buffer: Encoding.Default.GetBytes(content));  
}
```

Unhandled exception

☐ Stop ☒ Resume

IOException Stack Trace Explorer

System.IO.IOException Create breakpoint : The process cannot access the file '/Users/george/Documents/dotnet/playground/Playground/bin/Debug/net8.0/file.txt' because it is being used by another process.

dispose pattern

IDisposable/IAsyncDisposable

```
public interface IDisposable
{
    void Dispose();
}
```

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


dispose pattern

IDisposable/IAsyncDisposable

```
public abstract class Stream : IDisposable, IAsyncDisposable
```

```
    public class FileStream : Stream
```

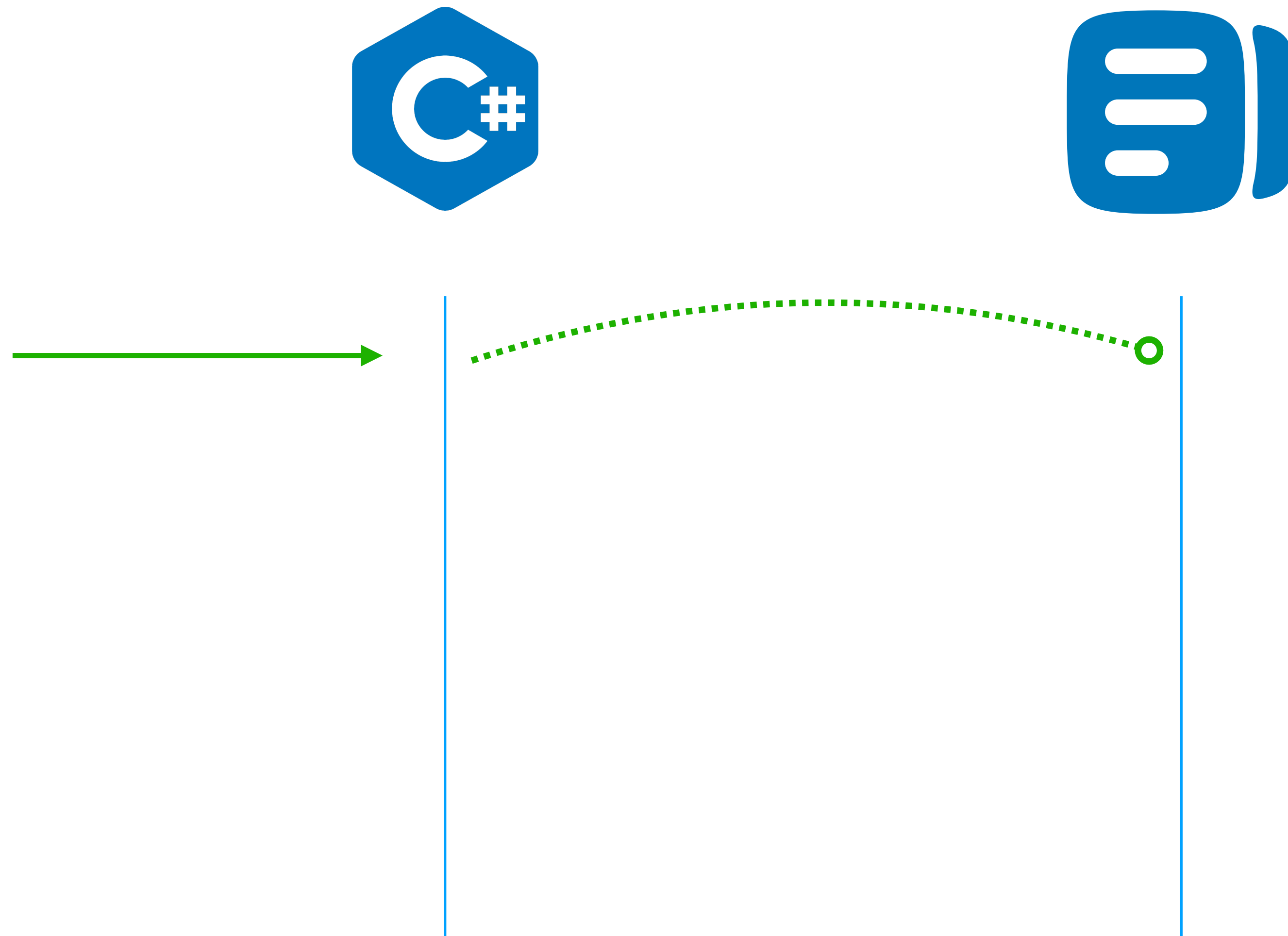
dispose pattern

using

```
WriteToFileNoDispose("file.txt", "content1");  
WriteToFileNoDispose("file.txt", "content");  
  
void WriteToFileNoDispose(string fileName, string content)  
{  
    using FileStream file = File.OpenWrite(fileName);  
    file.Write(Encoding.Default.GetBytes(content));  
}
```

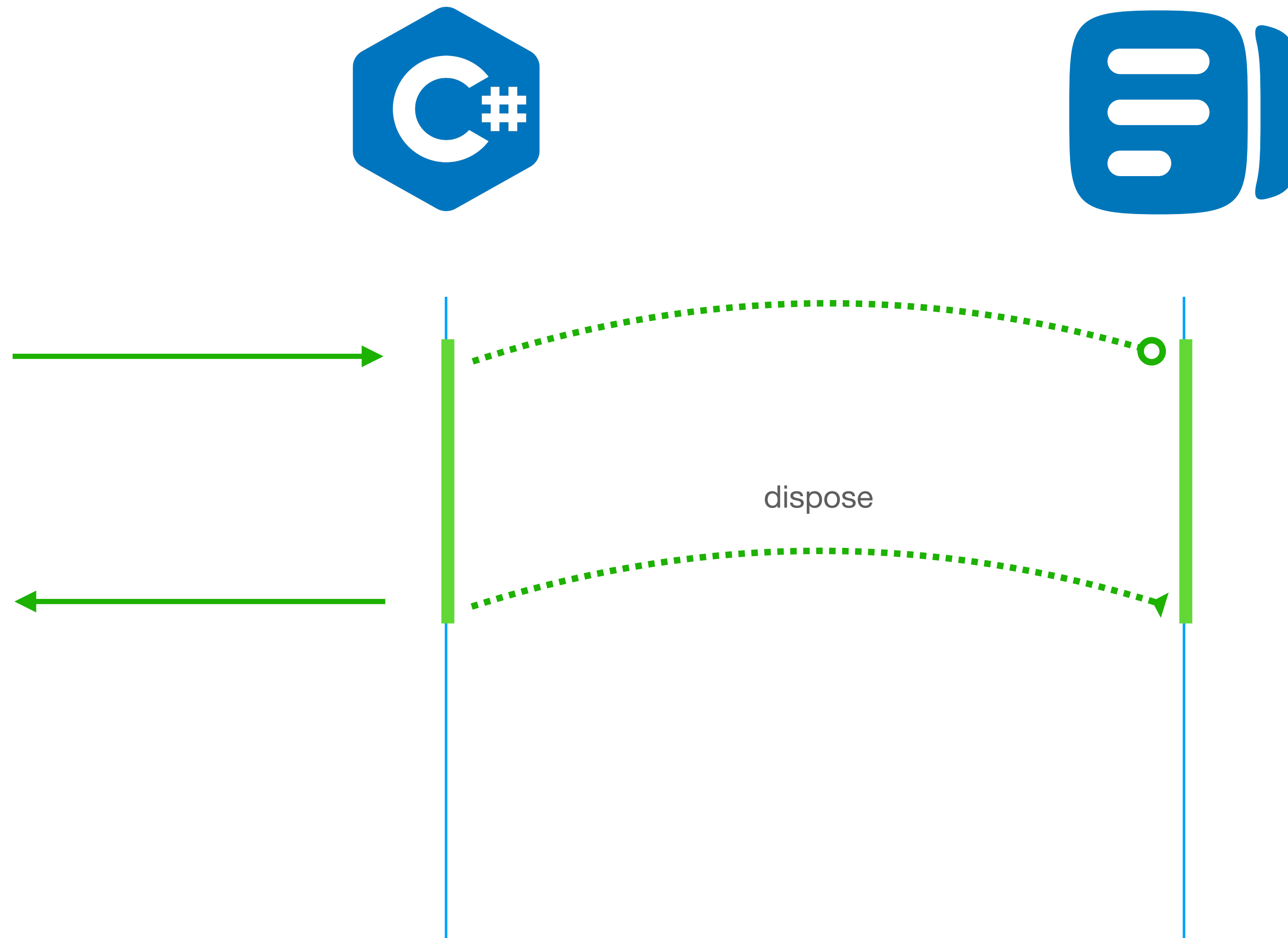
dispose pattern

пример блокирования файла



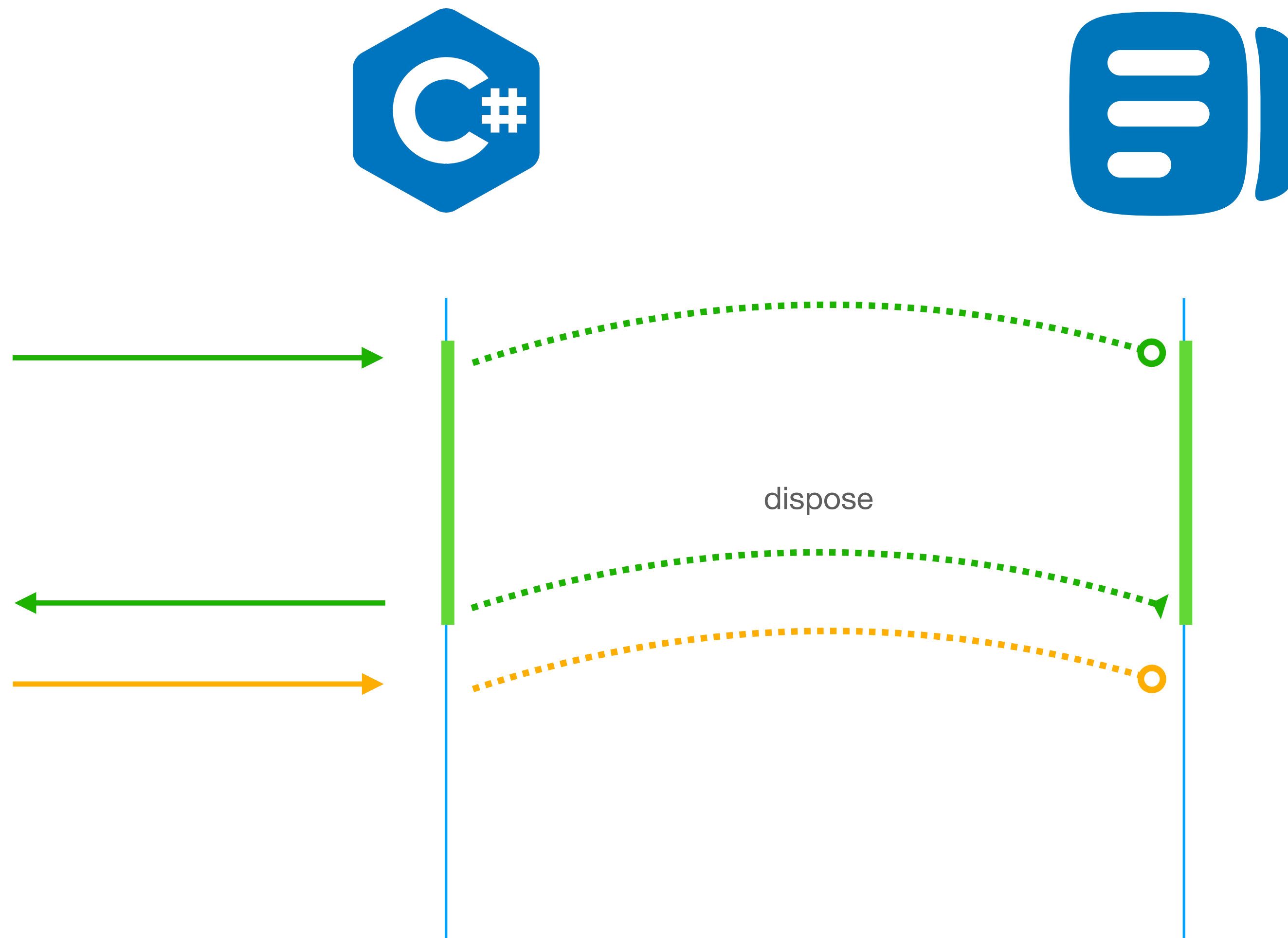
dispose pattern

пример блокирования файла



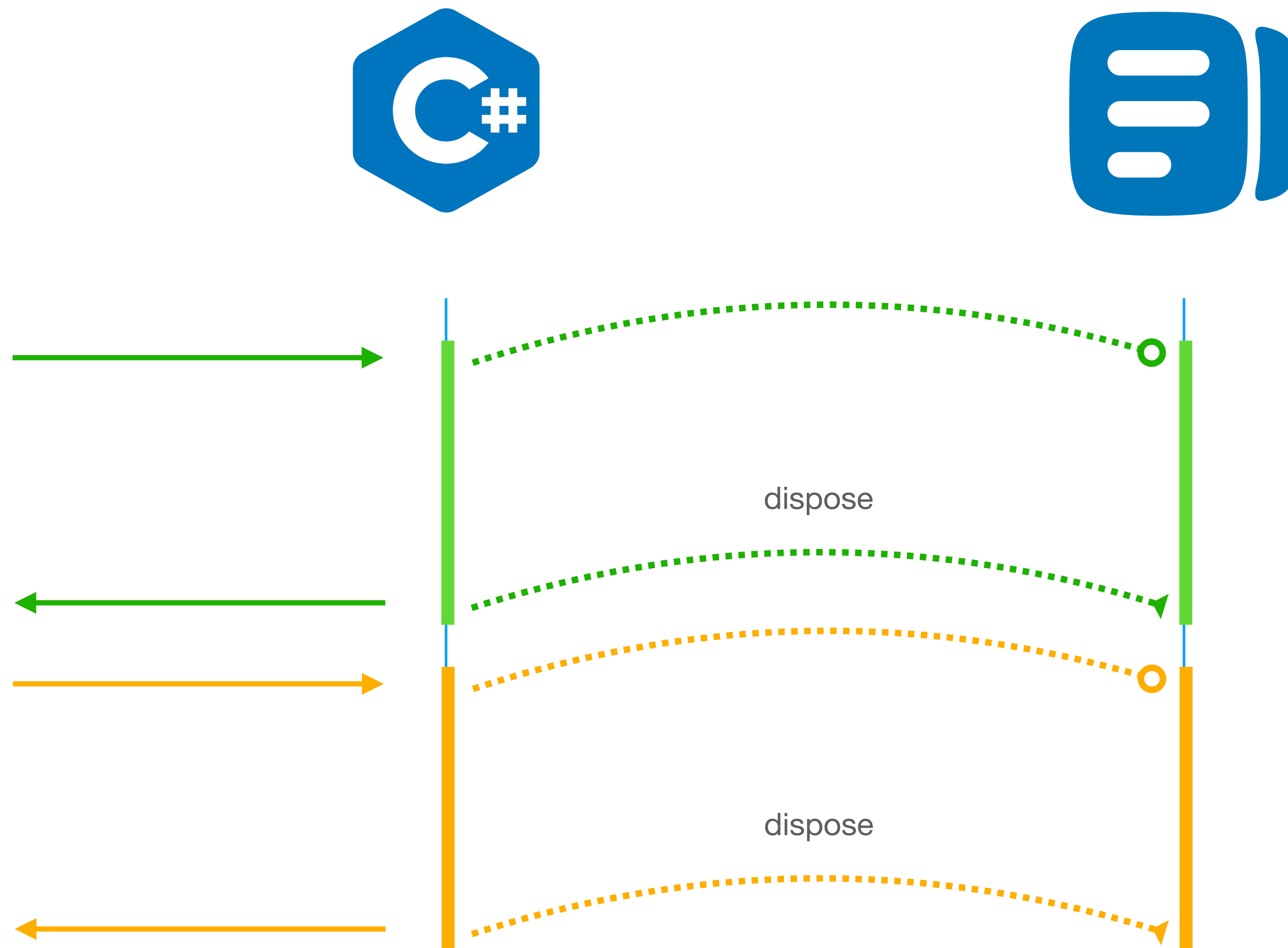
dispose pattern

пример блокирования файла



dispose pattern

пример блокирования файла



ValueTask

- позволяет обойтись без аллокаций на куче в случае отсутствия асинхронных операций
- поддерживает `await`
- приводит к аллокации `Task`, если в теле метода есть `await`

ValueTask

```
public async ValueTask<string> GetValueAsync(int key)
{
    if (_cache.TryGetValue(key, out string? value))
        return value;

    value = await GetValueUncachedAsync(key);
    _cache[key] = value;

    return value;
}
```

если значение есть в кеше,
асинхронной операции не будет,
следовательно не будет аллокаций

если есть асинхронная операция,
то внутри ValueTask будет лежать Task,
следовательно аллокации будут

IAsyncEnumerable

async с коллекциями

проблема

```
async Task<IEnumerable<int>> GetDataAsync() { }
```

```
IEnumerable<int> data = await GetDataAsync();  
IEnumerable<int> modifiedData = data.Select(x ⇒ x * 2);
```

```
IEnumerable<int> modifiedData = (await GetDataAsync()).Select(x ⇒ x * 2);
```

async с коллекциями

проблема

```
async Task<IEnumerable<int>> GetAllDataAsync(CancellationToken cancellationToken)
{
    var data = new List<int>();

    for (int i = 0; i < 100; i++)
    {
        IEnumerable<int> page = await GetPageAsync(i, cancellationToken);
        data.AddRange(page);
    }

    return data;
}

async Task<IEnumerable<int>> GetPageAsync(int page, CancellationToken cancellationToken) { ... }
```

async с коллекциями

проблема

- более сложный state-management методов
- лишние аллокации для промежуточных коллекций
- отсутствие потоковой обработки данных (обрабатываем только когда все данные получены)

async с коллекциями

асинхронные генераторы

```
async IEnumerable<int> GetAllDataAsync(
    [EnumeratorCancellation] CancellationToken cancellationToken)
{
    for (int i = 0; i < 100; i++)
    {
        IEnumerable<int> page = await GetPageAsync(i, cancellationToken);

        foreach (int value in page)
        {
            yield return value;
        }
    }
}
```

async с коллекциями

асинхронные генераторы

- async методы, возвращающие IEnumerable
- используют yield синтаксис как обычные генераторы
- позволяют потребителям обрабатывать данные по мере их получения
- CancellationToken в асинхронных генераторах должен помечаться атрибутом [EnumeratorCancellation] (это необходимо для корректной работы метода .WithCancellation у IEnumerable)

async с коллекциями

EnumeratorCancellation и .WithCancellation

- метод .WithCancellation используется для определения дополнительного токена отмены
- если вы вызываете метод .WithCancellation, то будет создан linked CancellationTokenSource на токен, что вы передали изначально в асинхронный генератор и тот что передали в .WithCancellation
- атрибут [EnumeratorCancellation] необходим как раз для того, чтобы компилятор знал, какой параметр будет использоваться в качестве токена отмены этого генератора

async с коллекциями

EnumeratorCancellation и .WithCancellation

Вывод в консоль не остановится, так как компилятор не переопределит параметр на linked token отмены

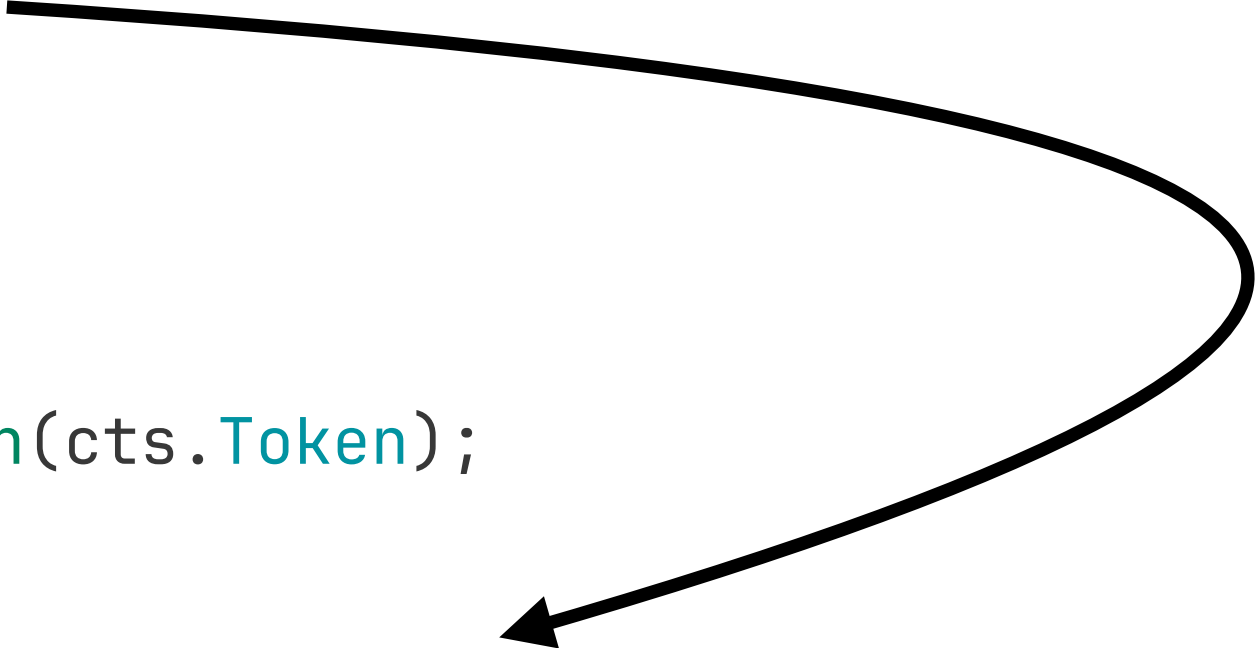
```
var cts = new CancellationTokenSource();
cts.CancelAfter(TimeSpan.FromSeconds(5));

var enumerable = GenerateInfiniteAsync(default).WithCancellation(cts.Token);

await foreach (var value in enumerable)
{
    Console.WriteLine(value);
}

async IAsyncEnumerable<int> GenerateInfiniteAsync(
    CancellationToken cancellationToken)
{
    var i = 0;

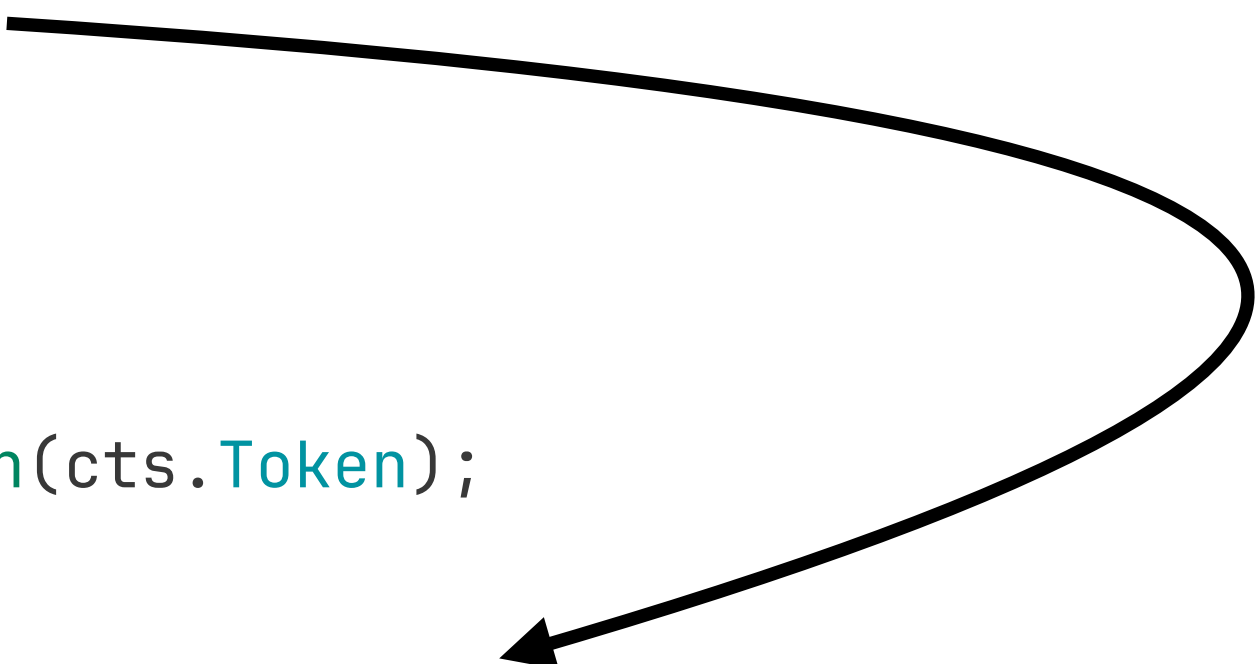
    while (cancellationToken.IsCancellationRequested is false)
    {
        yield return i++;
    }
}
```



async с коллекциями

EnumeratorCancellation и .WithCancellation

Вывод в консоль остановится через 5 секунды, так как компилятор знает какой параметр переопределить



```
var cts = new CancellationTokenSource();
cts.CancelAfter(TimeSpan.FromSeconds(5));

var enumerable = GenerateInfiniteAsync(default).WithCancellation(cts.Token);

await foreach (var value in enumerable)
{
    Console.WriteLine(value);
}

async IAsyncEnumerable<int> GenerateInfiniteAsync(
    [EnumeratorCancellation] CancellationToken cancellationToken)
{
    var i = 0;

    while (cancellationToken.IsCancellationRequested is false)
    {
        yield return i++;
    }
}
```

async с коллекциями

обработка IEnumerable

- await foreach
- System.Linq.Async

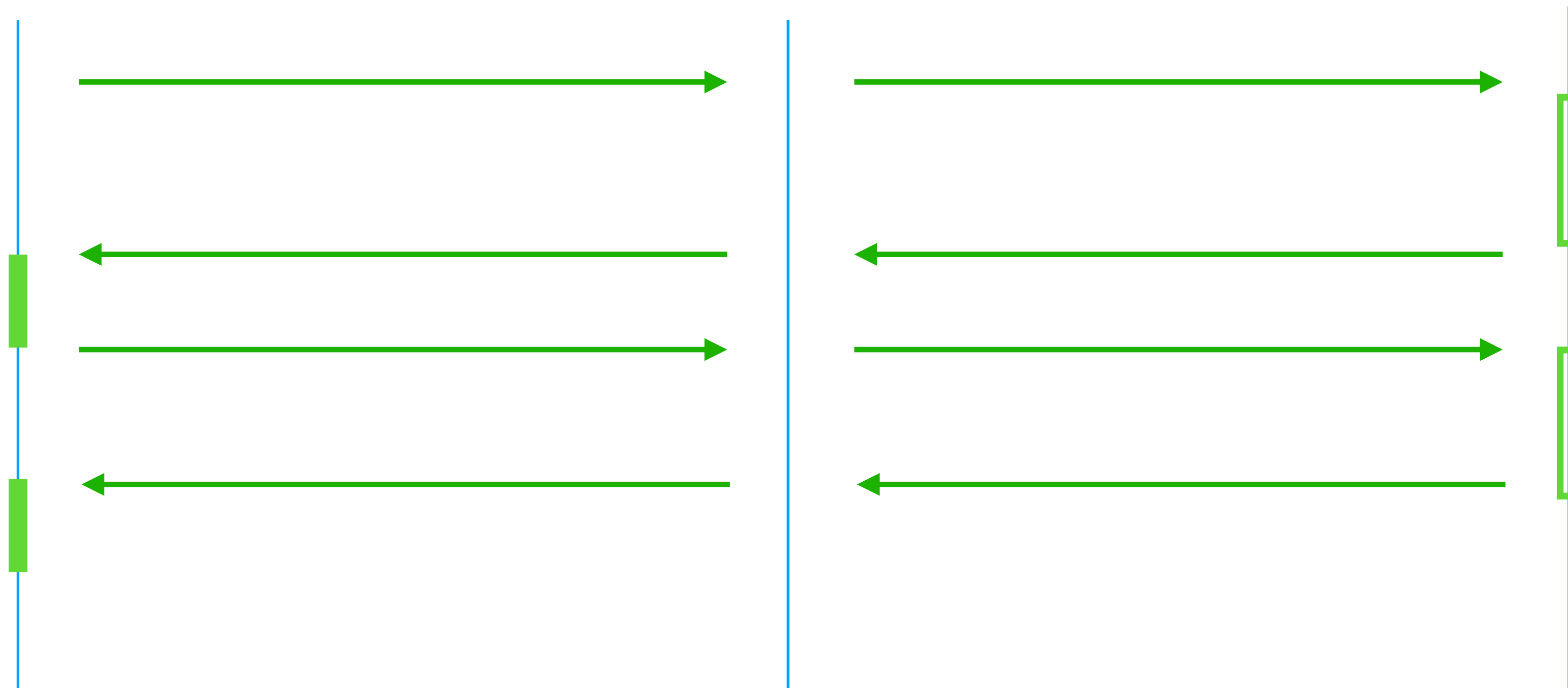
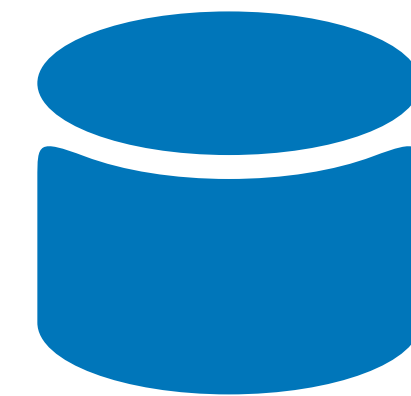
channels

асинхронная потоковая обработка данных

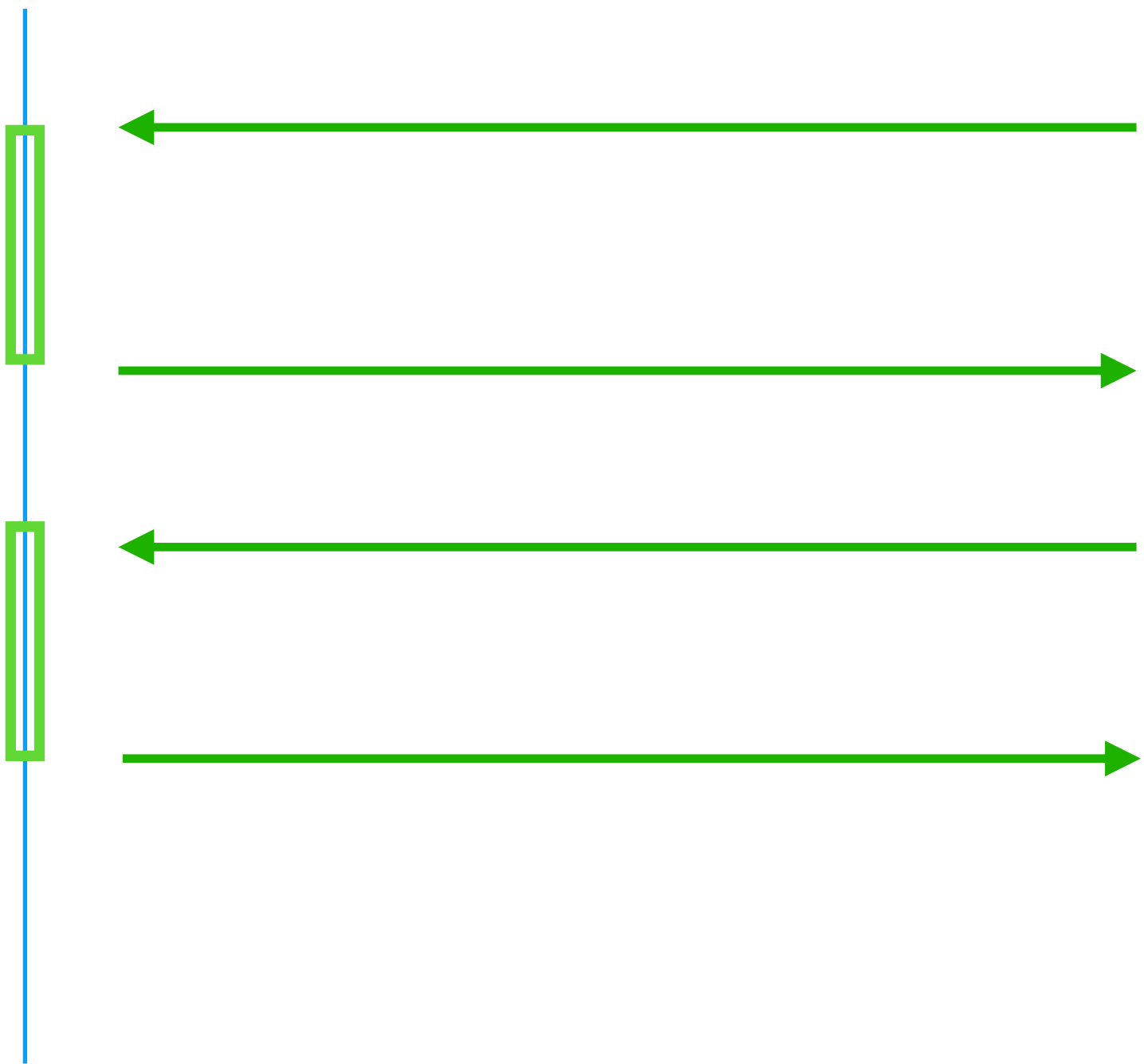
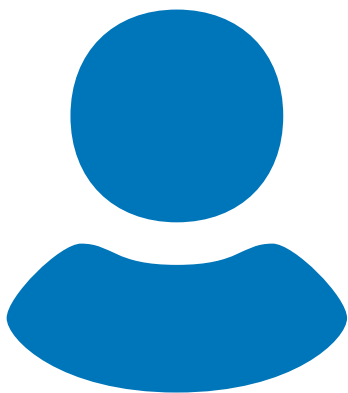
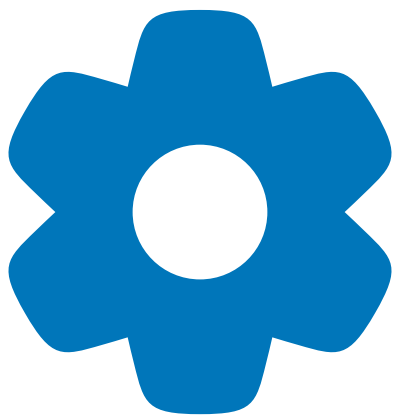
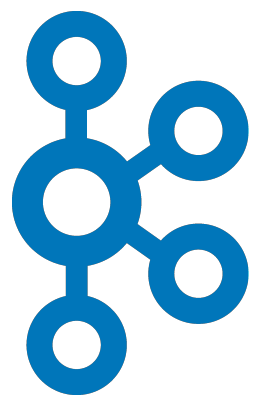
IAsyncEnumerable



IAsyncEnumerable



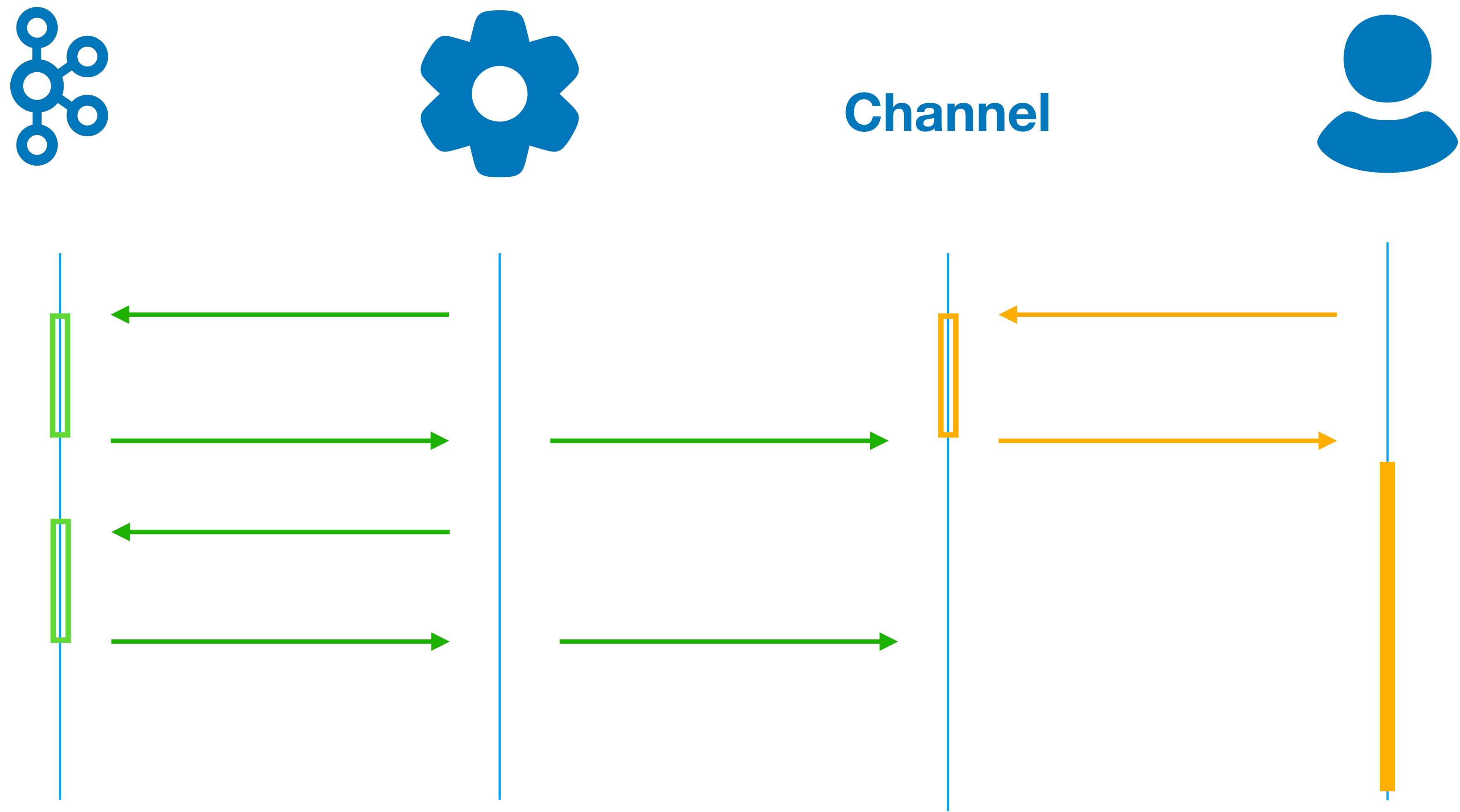
асинхронная потоковая обработка данных



?

асинхронная потоковая обработка данных

channels



класс Channel

создание каналов

- CreateUnbounded
 - не ограничены по размеру буферизированных значений
- CreateBounded
 - ограничены по размеру буферизированных значений

класс Channel

ChannelWriter

- объект реализующий запись в канал
- `ValueTask WriteAsync(T, CancellationToken)`
- при записи в Bounded канал, если он заполнен, будет выполнено ожидание, пока в нём освободится место
- при записи в Unbounded канал, если он заполнен, он расширит свой буффер, значение будет записано сразу

класс Channel

ChannelReader

- объект реализующий чтение из канала
- `ValueTask<T> ReadAsync(CancellationToken)`
- если в канале не будет непрочтенных сообщений, будет выполнено ожидание
- `IAsyncEnumerable<T> ReadAllAsync(CancellationToken)`

класс Channel

заккрытие канала

- Complete
- TryComplete
 - может вернуть false, если канал уже был закрыт

класс Channel

UnboundedChannelOptions

- bool SingleReader
- bool SingleWriter

класс Channel

BoundedChannelOptions

- bool SingleReader
- bool SingleWriter
- int Capacity
- BoundedChannelFullMode FullMode
 - Wait
 - DropNewest
 - DropOldest
 - DropWrite