Signals

Оглавление

[Назначение системы 2](#_Toc20689607)

[Быстрый старт 2](#_Toc20689608)

[Типы событий 3](#_Toc20689609)

[Состояния 3](#_Toc20689610)

[Сетевые сообщения 3](#_Toc20689611)

[Использование нескольких шин 4](#_Toc20689612)

[Зацикленные события 4](#_Toc20689613)

[Отписка от событий 4](#_Toc20689614)

[Логирование событий 4](#_Toc20689615)

[Запрет сообщений 5](#_Toc20689616)

[Возникновение исключений в обработчиках 5](#_Toc20689617)

[Связывание данных (Data Binding) 5](#_Toc20689618)

[Методы Signal, Signal<T>, Signal<T1, T2> 6](#_Toc20689619)

[Методы и свойства State<T> 7](#_Toc20689620)

[Fluent синтаксис 7](#_Toc20689621)

[Упрощенный синтаксис публикации событий 8](#_Toc20689622)

[Scope Bus 8](#_Toc20689623)

[События и короутины 9](#_Toc20689624)

# Назначение системы

Signals представляет собой шину сообщений (Event Bus) и предназначен для связывания различных компонентов Unity между собой.

Помимо связывания компонентов, Signals предоставляет возможности передачи сетевых сообщений (Networked Events), хранения данных (State) и связывания визуальных компонентов с данными (Data Binding).

# Быстрый старт

Для начала работы с системой, создайте класс Bus следующего содержания:

class Bus

{

public readonly static Signal<string> ShowNotificationRequest;

static Bus() => BusHelper.InitFields<Bus>();

}

Этот демонстрационный класс содержит одно событие ShowNotificationRequest и статический конструктор.

Статический конструктор вызывает метод BusHelper.InitFields<T>(). Этот метод автоматически создает объекты событий и присваивает их полям в классе Bus. Это позволяет не создавать объект события вручную.

В данном случае поле ShowNotificationRequest будет проинициализировано автоматически и вам не нужно ничего присваивать этому полю.

Поле ShowNotificationRequest имеет тип Signal<string> что обозначает событие с одним аргументом типа string.

После того, как класс Bus создан, вы можете начинать использование шины в своих пользовательских классах.

Для подписки на событие, напишите в методе Awake:

Bus.ShowNotificationRequest.Subscribe(this, s => Debug.Log($"Note: {s}"));

В этой строке вы подписываетесь на событие ShowNotificationRequest. При срабатывании события в консоль будет выведен текст сообщения.

Первым аргументом в методе Subscribe передается ссылка на текущий компонент. Это необходимо для того, что бы работала автоматическая отписка от события, когда компонент будет уничтожен. Таким образом, отписываться от события нигде не нужно.

Вторым аргументом передается делегат, который сработает при наступлении события.

Для вызова события напишите следующее:

Bus.ShowNotificationRequest.Publish("Hello from Signal!");

Здесь происходит вызов события ShowNotificationRequest с параметром "Hello from Signal!".

При вызове этого метода, все компоненты подписавшиеся на событие получат сообщение.

Подписчики получают сообщение в порядке подписки. Первый подписавшийся всегда получит сообщение первым.

# Типы событий

Система поддерживает следующие типы для событий:

Signal – событие без аргументов.

Signal<T> - событие с одним аргументом.

Signal<T1, T2> - событие с двумя аргументами.

State<T> - состояние типа T.

# Состояния

Событие не имеет состояния. Его задача – вызов методов одних компонентов, по запросу других компонентов.

Помимо событий, Signals поддерживает также и состояния. Они описываются классом State<T>. Этот класс поддерживает те же методы, что и Signal<T>, но помимо этого, State<T> хранит последнее переданное в него значение. Это значение вы можете получить через свойство Value.

Таким образом, состояние State<T> можно рассматривать как поле типа T с нотификацией об изменениях его значения. Подписавшись на State<T> вы будете получать сообщения об изменении значения состояния.

# Сетевые сообщения

Signals поддерживает прозрачную передачу сообщений по сети. При этом событие, вызванное одним из игроков, сработает не только у него, но и у остальных игроков.

Для поддержки сети (на примере Photon) нужно сделать следующее:

1. Написать класс, реализующий интерфейс INetworkBusService. Пример такого класса для Photon – смотрите в файле PhotonNetworkBusService.cs. Класс PhotonNetworkBusService реализует интерфейс INetworkBusService и вызывает метод в Awake:

BusToNetworkBridge.Init<Bus>(this);

Этот вызов заставляет использовать PhotonNetworkBusService как сетевой сервис для шины Bus.

Для работы этого класса необходимо установить Photon.

2. Создать на сцене GameObject и разместить в нем компонент PhotonNetworkBusService.

3. Пометить сетевые события шины атрибутом[NetworkedSignal]. Эти события автоматически будут передаваться по сети, и срабатывать у всех игроков.

Например:

class Bus

{

[NetworkedSignal]

public readonly static Signal<string> ShowNotificationRequest;

...

}

Здесь вызов события ShowNotificationRequest любым игроком, будет срабатывать у всех игроков, а не только у того, кто его вызвал.

# Использование нескольких шин

Signals позволяет создавать любое количество классов, содержащих события. Вы также можете размещать события внутри любых ваших компонентов.

# Зацикленные события

Может возникнуть ситуация, когда подписчик на событие сам же вызывает это событие.

Например:

Bus.ShowNotificationRequest.Subscribe(this, s =>

Bus.ShowNotificationRequest.Publish("Hello!"));

Signals автоматически пресекает зацикленные события. Если в данный момент отрабатывает событие то все другие попытки вызова данного события будут игнорироваться.

Таким образом, в приведенном примере вызов Bus.ShowNotificationRequest.Publish("Hello!") будет проигнорирован.

# Отписка от событий

Signals поддерживает автоматическую отписку от событий для скриптов, унаследованных от MonoBehaviour. Если скрипт будет уничтожен (Dispose) то он автоматически будет отписан от событий.

Автоматическая отписка касается только подписки с вызовом методов Subscribe(…), либо с вызовом метода JoinTo(…).

Система также поддерживает и подписку без автоматической отписки. Для этого нужно подписываться на события методом SubscribeRaw(…).

Для принудительной отписки от события, вызывайте метод Unsubscribe().

# Логирование событий

Для включения/отключения логирования событий – используйте свойство SignalBase.LogEvents. По умолчанию – логирование включено.

# Запрет сообщений

Вы можете отключить отправку любых сообщений, присвоив True свойству SignalBase.SuppressSignals. Эта настройка отключает любые сообщения в любой шине.

# Возникновение исключений в обработчиках

EvenLayer игнорирует исключения, возникающие в обработчиках событий. Если один из подписчиков вызвал исключение, остальные подписчики все равно будут вызваны.

# Связывание данных (Data Binding)

Signals поддерживает автоматическую передачу данных в UI компоненты и получение данных от UI компонентов.

Для связывания события или состояния с UI компонентом, вызовите метод Bind(). Например:

Bus.ShowNotificationRequest.Bind(myInputField);

Здесь myInputField – поле ввода типа InputField. После вызова этого метода, вызов события ShowNotificationRequest будет заносить значение в myInputField. И наоборот, ввод текста в myInputField будет вызывать событие ShowNotificationRequest.

Не все типы событий и не все UI компоненты могут быть связаны. Это зависит от типа события, от типа аргумента и от типа UI компонента.

Ниже приводится таблица возможной связи событий с компонентами:

|  |  |  |  |
| --- | --- | --- | --- |
| Тип события | TwoWay Binding | OneWay Binding, Subscribe only | OneWay Binding, Publish only |
| Signal<string>, State<string> | InputField, TMP\_InputField\* | Text, TextMeshProUGUI\* |  |
| Signal<float>, State<float> | InputField, Slider, TMP\_InputField\* | Text, TextMeshProUGUI\* |  |
| Signal<int>,  State<int> | InputField, Slider, Dropdown, Toggle, TMP\_InputField\*, TMP\_Dropdown\* | Text, TextMeshProUGUI\* |  |
| Signal<bool>,  State<bool> | Toggle |  |  |
| Signal<Enum>,  State<Enum> | Dropdown, TMP\_Dropdown\* |  | Button |
| Signal |  |  | Button |
| Signal<Sprite>, State<Sprite> |  | Image |  |
| Signal<Texture>,  State<Texture> |  | RawImage |  |
| Signal<Color>,  State<Color> |  | RawImage, Image, Text, TextMeshProUGUI\* |  |

\* Компоненты TextMeshPro могут быть связаны только если они установлены. Для отключения связки с TMP – задайте директиву компилятора NO\_TEXTMESHPRO.

При вызове метода Bind для State<T>, компонент UI сразу получает текущее значение состояния. При связывании с Signal<T> компонент получит значение при первом возникновении события.

Метод Bind имеет несколько полезных перегрузок:

* Bind<T>(this Signal<T> state, Button button, T value)

Нажатие на кнопку публикует значение value.

* Bind<T>(this Signal<T> state, Button button, T value, Color selectedColor)

Нажатие на копку публикует значение value. Если значение value опубликовано, кнопка принимает цвет selectedColor.

* BindInteractable<T>(this Signal<T> state, Selectable selectable, Func<T, bool> converter)

Компонент становится интерактивным, если возникает событие, и выполняется условие.

* BindActive<T>(this Signal<T> state, GameObject gameObject, Func<T, bool> converter)
* BindActive<T>(this Signal<T> state, Component component, Func<T, bool> converter)

Объект становится активным, если выполняется условие.

# Методы Signal, Signal<T>, Signal<T1, T2>

Ниже описаны методы класса Signal<T>. Методы Signal и Signal<T1, T2> - аналогичны, разница только в количестве аргументов.

* public SubscriberInfo SubscribeRaw(Action<T> callback)

Подписка на событие. При срабатывании события будет вызван делегат callback.

* public SubscriberInfo Subscribe(Component component, Action<T> callback)

Подписка на событие, с автоматической отпиской. В метод передается связанный с отпиской компонент (обычно это this). При уничтожении компонента, он будет автоматически отписан. При срабатывании события будет вызван делегат callback.

* public SubscriberInfo Subscribe(Component component, Func<T, bool> condition, Action<T> callback)

Подписка на событие, с автоматической отпиской. В метод передается связанный с отпиской компонент (обычно это this). При уничтожении компонента, он будет автоматически отписан.

Вторым аргументом идет условие condition. При срабатывании события и выполнении условия condition, будет вызван делегат callback.

* public SubscriberInfo Subscribe(Component component, Func<T, bool> condition, Action callback)

Аналогично предыдущему, но в делегат callback не передается аргумент события. Эта перегрузка метода полезна, если вы уже проверили аргумент в condition, и вам не нужно значение аргумента в callback.

* public bool IsComponentSubscribed(Component component)

Возвращает true, если компонент подписан на данное событие.

* public void Publish(T data)

Публикация события, передача аргумента события.

* public bool Unsubscribe(Component component)

Принудительная отписка компонента от события.

* public void Unsubscribe(Action<T> callback)

Принудительная отписка делегата от события.

* public void UnsubscribeAll()

Отписка всех подписчиков.

# Методы и свойства State<T>

Для State<T> работают те же методы, что и для Signal<T>.

А также следующие:

* public State(T data)

Конструктор, принимающий стартовое значение состояния. При вызове конструктора, метод Publish не будет вызван.

* public T Value{get;set;}

Свойство, возвращающее значение состояния (типа T). При присвоении, если значение поменялось, будет вызвано событие.

* public void PublishForced(T value)

Будет присвоено новое значение состояния и вызван метод Publish, независимо от того, поменялось значение или нет.

* public void PublishSilent(T value)

Будет присвоено новое значение состояния, но метод Publish не будет вызван, независимо от того, поменялось значение или нет.

# Fluent синтаксис

Для методов подписки Subscribe, SubscribeRaw и Bind реализован Fluent синтаксис. Вы можете вызывать несколько дополнительных методов после метода Subscribe, SubscribeRaw или Bind:

* public SubscriberInfo Condition(Func<T, bool> condition)

Обработчик события будет вызван только при выполнении условия condition.

Вы можете назначить только одно условие для события. Если условие уже было указано в методе Subscribe, вызов метода Condition() вызовет исключение.

* public SubscriberInfo JoinWith(Component component, bool callOnlyWhenActive = false)

Связывает событие с компонентом component. При уничтожении компонента, он будет автоматически отписан.

Параметр callOnlyWhenActive запретит вызов события для компонента, если GameObject в котором находится компонент - не активен.

* public SubscriberInfo InvokeOnce()

Автоматически отписывает подписчика после первого срабатывания события. Если задано условие срабатывания, то подписчик все равно будет отписан, даже если условие не сработает для данного вызова события.

* public SubscriberInfo OnUnsubscribed(Action callback)

Задает делегат, который будет вызван после того, как подписчик будет отписан от события.

# Упрощенный синтаксис публикации событий

Вместо вызова метода Publish вы можете использовать оператор +=, указав справа от него передаваемый в событие параметр.

Например:

|  |  |
| --- | --- |
| Обычный синтаксис | Упрощенный синтаксис |
| Bus.mySignal.Publish(); | Bus.mySignal += true; |
| Bus.mySignal.Publish(“Hello”); | Bus.mySignal += “Hello”; |
| Bus.mySignal.Publish(12, “ABC”); | Bus.mySignal += new Tuple<int, string>(12, “ABC”); |

Для того, что бы упрощенный синтаксис работал, вы не должны указывать модификатор readonly в декларации события.

# Scope Bus

Signals поддерживает упрощенный механизм событий, который не требует предварительной декларации события. В Scope вы можете публиковать и подписываться на события, со строкой в качестве идентификатора события.

Например:

Scope.Signal<string>("MyNotification").Subscribe(...);

* подписка на событие “MyNotification”.

Scope.Signal<string>("MyNotification").Publish("Hello");

* публикация события “MyNotification” с передачей параметра.

Методы Scope.Signal()/Signal()/State() возвращают объекты типа SignalBase. Поэтому для событий из Scope работают все методы, которые работают для обычных событий, в том числе Data Binding.

Networked Signals в данный момент не поддерживаются в Scope.

# События и короутины

Класс-расширение CoroutineExtensions содержит методы, которые позволяют ожидать событий внутри короутин.

Для Signal и State<T> доступен метод Wait:

public static IEnumerator Wait(this Signal e, Func<bool> condition = null)

Метод возвращает IEnumerator и может использоваться для ожидания наступления события без аргументов или State.

Для всех событий также доступен метод GetWaiter:

public static Waiter<T> GetWaiter<T>(this Signal<T> e, Func<T, bool> condition = null)

Этот метод возвращает объект типа Waiter, который в свою очередь реализует интерфейс IEnumerator. Также, объект Waiter содержит данные, переданные в событие.

Пример использования Wait() и GetWaiter():

IEnumerator MyCoroutine()

{

yield return Bus.MyState.Wait();

Debug.Log("MyState published " + Bus.MyState.Value);

var waiter = Bus.MySignal.GetWaiter();

yield return waiter;

Debug.Log("MySignal published " + waiter.Data);

}

# Связанные события

Signals поддерживает связанные события – когда вызов одного события влечет за собой вызов другого события. Связанные события поддерживаются для Signal<T> и State<T>.

Связывание событий осуществляется методами JoinWith:

* public void JoinWith<TM>(Signal<TM> masterSignal, Func<TM, T> convert)
* public void JoinWith<TM>(Signal<TM> otherSignal, Func<TM, T> convert, Func<T, TM> backConvert)

Пример связывания:

class Bus

{

/// <summary>My Color</summary>

public static State<Color> Color;

/// <summary>Channel R of My Color</summary>

[NetworkedSignal]

public static State<float> ColorChannelR;

static Bus()

{

BusHelper.InitFields<Bus>();

//join ColorChannelR and Color (twoway)

Bus.ColorChannelR.JoinWith(Bus.Color, color => color.r, red => { var c = Bus.Color.Value; c.r = red; return c; });

}

}

В примере изменение состояния Color влечет изменение состояния ColorChannelR. И наоброт – изменение ColorChannelR приводит к изменению Color.