**Асинхронный в DI**

При внедрении зависимостей инжектор разрешает зависимости целевого класса только один раз, часто после первого создания класса. Другими словами, внедрение — это одноразовый процесс, который не отслеживает внедренные зависимости, чтобы обновить их позже. Если зависимость не готова в момент внедрения, привязка либо не будет разрешена в случае необязательных привязок, либо завершится полным сбоем, выдав ошибку.

Это создает дилемму при реализации зависимостей, которые разрешаются асинхронно. Вы можете обойти ограничения DI, тщательно разработав код таким образом, чтобы внедрение происходило после завершения процесса. Это требует тщательного планирования, что приводит к повышенной сложности в настройке, а также подвержено ошибкам.async

Кроме того, можно внедрить промежуточный объект, который отслеживает результат операции. Когда вам нужно получить доступ к зависимости, вы можете использовать этот промежуточный объект, чтобы проверить, выполнена ли задача, и получить результирующий объект. С помощью экспериментальной поддержки мы хотели бы предоставить способы решения этой проблемы в Extenject. Вы можете найти расширения в папке **Plugins/Zenject/OptionalExtras/Async**.asyncasyncasyncasync

**Пример**

Давайте посмотрим, как мы можем внедрить асинхронные зависимости через промежуточный объект. В качестве этого посредника реализуются асинхронные расширения. Вы можете использовать его следующим образом.AsyncInject<T>

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.BindAsync<IFoo>().FromMethod(async () =>

{

await Task.Delay(100);

return (IFoo) new Foo();

}).AsCached();

}

}

public class Bar : IInitializable, IDisposable

{

readonly AsyncInject<IFoo> \_asyncFoo;

IFoo \_foo;

public Bar(AsyncInject<IFoo> asyncFoo)

{

\_asyncFoo = asyncFoo;

}

public void Initialize()

{

if (!\_asyncFoo.TryGetResult(out \_foo))

{

\_asyncFoo.Completed += OnFooReady;

}

}

private void OnFooReady(IFoo foo)

{

\_foo = foo;

}

public void Dispose()

{

\_asyncFoo.Completed -= OnFooReady;

}

}

Здесь мы используем для передачи лямбда, которая ожидает 100 мс, а затем возвращает только что созданный экземпляр. Этот метод может быть любым другим методом с сигнатурой . \*Примечание: ключевое слово является деталью реализации и, следовательно, не является частью подписи. Метод расширения предоставляет отдельное связующее для операций. Эта связующая в настоящее время ограничена несколькими поставщиками. Такие функции, как Pooling и Factory, в настоящее время не поддерживаются.BindAsync<IFoo>().FromMethod()asyncFooTask<T> Method()asyncBindAsync<T>asyncFromX()

При приведенной выше привязке экземпляр добавляется в контейнер. Поскольку область действия задана на операцию, она будет выполняться только один раз и сохранит результат. Важно отметить, что операции не начнутся до тех пор, пока эта привязка не будет разрешена. Если вы хотите, чтобы операция начиналась сразу после установки, используйте опцию.AsyncInject<IFoo>AsCached()AsyncInject<IFoo>asyncasyncNonLazy()

После внедрения в , мы можем проверить, доступно ли возвращаемое значение операции, вызвав . метод. Этот метод вернется, если нет результата для возврата. Если результат еще не готов, мы можем прослушать событие, чтобы получить возвращаемое значение, когда операция завершится.BarasyncTryGetResultfalseCompletedasync

В качестве альтернативы мы можем использовать следующие методы для проверки доступности результатов.

// Use HasResult to check if result exists

if (\_asyncFoo.HasResult)

{

// Result will throw error if prematurely used.

var foo = \_asyncFoo.Result;

}

// AsyncInject<T> provides custom awaiter

IFoo foo = await \_asyncFoo;

# Автоматическое Mocking

Mocking

Тест двойной, макетный, заглушка, фальшивка или шпион? Для простоты в этой документации используется слово «макет» для всех.

Одной из действительно крутых особенностей DI является тот факт, что он делает тестирование кода намного, намного проще. Это связано с тем, что вы можете легко заменить одну зависимость на другую, используя другой корень композиции. Например, если вы хотите протестировать только определенный класс (назовем его Foo) и не заботитесь о тестировании его зависимостей, вы можете написать для них макеты, чтобы вы могли специально изолировать Foo.

public class Foo

{

IWebServer \_webServer;

public Foo(IWebServer webServer)

{

\_webServer = webServer;

}

public void Initialize()

{

//...

var x = \_webServer.GetSomething();

//...

}

}

В этом примере у нас есть класс Foo, который взаимодействует с веб-сервером для получения контента. Обычно это очень сложно проверить по следующим причинам:

* Вам нужно будет настроить среду, в которой он может правильно подключаться к веб-серверу (настройка портов, URL-адресов и т. Д.).
* Выполнение теста может быть медленнее и ограничивать объем тестирования, которое вы можете выполнить
* Сам веб-сервер может содержать ошибки, поэтому вы не можете с уверенностью изолировать Foo как проблемную часть теста
* Вы не можете легко настроить значения, возвращаемые веб-сервером, для тестирования отправки различных входных данных в класс Foo

Однако, если мы создадим фиктивный класс для IWebServer, мы сможем решить все эти проблемы:

public class MockWebServer : IWebServer

{

//...

}

Тогда подключите его в нашем инсталляторе:

Container.Bind<IWebServer>().To<MockWebServer>().AsSingle();

Затем вы можете реализовать поля интерфейса IWebServer и настроить их в зависимости от того, что вы хотите протестировать на Foo. Надеюсь, вы видите, как это может значительно облегчить жизнь при написании тестов.

Чтобы избежать написания всех макетов классов, как в приведенном выше примере класса MockWebServer. Zenject позволяет автоматизировать этот процесс с помощью библиотеки имитации, которая сделает всю работу за вас. Zenject поддерживает Moq и NSubstitute. Оба чаще всего используются в полевых условиях, но имеют разные подходы к издевательствам. И в этом документе они будут обрабатываться по-другому.

Обратите внимание, что по умолчанию в Zenject не включено автоматическое издевательство. Если вы хотите использовать функцию автоматического макетирования.zip вам необходимо извлечь AutoMoq или AutoSubstitute.zip как описано ниже.

## Использование MOQ

Если вы хотите использовать Moq, вам нужно перейти в каталог установки Zenject и извлечь содержимое в тот же каталог. Затем извлеченную папку следует переместить в .Zenject\OptionalExtras\AutoMoq.zipZenject\OptionalExtras\TestFrameWork\Editor

Обратите внимание, что существует несколько версий Moq.dll включенных в zip-архив, и что вы должны использовать ту, которая предназначена для версии среды выполнения сценариев, которую вы настроили в настройках проигрывателя. Также обратите внимание, что если вы используете среду выполнения сценариев версии 3.5, вам также может потребоваться изменить «Уровень совместимости API» с «Подмножество .NET 2.0» на «.NET 2.0»

После извлечения пакета автоматического макетирования достаточно использовать следующий синтаксис для имитации различных частей вашего проекта:

Container.Bind<IFoo>().FromMock();

Или, в качестве альтернативы, если мы хотим настроить значения для нашего фиктивного класса (а не просто создавать значения по умолчанию):

var foo = new Mock<IFoo>();

foo.Setup(x => x.Bar).Returns("a");

Container.BindInstance(foo.Object);

Для получения более подробной информации см. документацию для [Moq](https://github.com/moq/moq4)

## Использование NSubstitute

Если вы хотите использовать NSubstitute, вам нужно перейти в каталог установки Zenject и извлечь содержимое . Затем извлеченную папку следует переместить в .Zenject\OptionalExtras\AutoSubstitute.zipZenject\OptionalExtras\TestFrameWork\Editor

Издевательство, огрызок, фальшивка, шпион, испытание двойника? Строгий или свободный? Нет, просто замените нужный вам тип!

После извлечения пакета автоматической замены вы готовы создать замену с помощью одной строки кода:

Container.Bind<ICalculator>().FromSubstitute();

Вместо того, чтобы писать:

var calculator = Substitute.For<ICalculator>();

Container.BindInstance(calculator);

### Значения авто

После создания подстановки некоторые свойства и методы будут автоматически возвращать значения, отличные от NULL. Например, любые свойства или методы, возвращающие интерфейс, делегат или чисто виртуальный класс\*, автоматически возвращают сами замены. Это обычно называется рекурсивным макетированием и может быть полезно, потому что вы можете избежать явного создания каждой замены, что означает меньше кода. Другие типы, такие как String и Array, по умолчанию будут возвращать пустые значения, а не нули.

### Установка возвращаемых значений

После создания замены это легко, как 1, 2, 3; Чтобы задать возвращаемые значения методов и свойств, выполните следующие действия.

calculator.Add(1, 2).Returns(3);

## Шпаргалка по инсталляторам

Ниже приведено множество случайно подобранных примеров привязок, которые вы можете включить в один из своих установщиков.

Для получения дополнительных примеров вам также может быть интересно прочитать некоторые из модульных тестов (см. и каталоги)Zenject/OptionalExtras/UnitTestsZenject/OptionalExtras/IntegrationTests

public override void InstallBindings()

{

// Create a new instance of Foo for every class that asks for it

Container.Bind<Foo>().AsTransient();

// Create a new instance of Foo for every class that asks for an IFoo

Container.Bind<IFoo>().To<Foo>().AsTransient();

// Non generic version of the above

Container.Bind(typeof(IFoo)).To(typeof(Foo)).AsTransient();

///////////// AsSingle

// Create one definitive instance of Foo and re-use that for every class that asks for it

Container.Bind<Foo>().AsSingle();

// Create one definitive instance of Foo and re-use that for every class that asks for IFoo

Container.Bind<IFoo>().To<Foo>().AsSingle();

// Bind the same instance to multiple types

// In this example, the same instance of Foo will be used for all three types

// (we have to use the non-generic version of Bind when mapping to multiple types)

Container.Bind(typeof(Foo), typeof(IFoo), typeof(IFoo2)).To<Foo>().AsSingle();

///////////// BindInterfaces

// This will have the exact same effect as the above line

// Bind all interfaces that Foo implements and Foo itself to a new singleton of type Foo

Container.BindInterfacesAndSelfTo<Foo>().AsSingle();

// Bind only the interfaces that Foo implements to an instance of Foo

// This can be useful if you don't want any classes to directly reference the concrete

// derived type

Container.BindInterfacesTo<Foo>().AsSingle();

///////////// FromInstance

// Use the given instance everywhere that Foo is used

// Note that in this case there's no good reason to use FromInstance

Container.Bind<Foo>().FromInstance(new Foo());

// This is simply a shortcut for the above binding

// This can be a bit nicer since the type argument can be deduced from the parameter

Container.BindInstance(new Foo());

// Bind multiple instances at once

Container.BindInstances(new Foo(), new Bar());

///////////// Binding primitive types

// BindInstance is more commonly used with primitive types

// Use the number 10 every time an int is requested

Container.Bind<int>().FromInstance(10);

Container.Bind<bool>().FromInstance(false);

// Or equivalently:

Container.BindInstance(10);

Container.BindInstance(false);

// You'd never really want to do the above though - you should almost always use a When condition for primitive values

Container.BindInstance(10).WhenInjectedInto<Foo>();

///////////// FromMethod

// Create instance of Foo when requested, using the given method

// Note that for more complex construction scenarios, you might consider using a factory

// instead with FromFactory

Container.Bind<Foo>().FromMethod(GetFoo);

// Randomly return one of several different implementations of IFoo

// We use Instantiate here instead of just new so that Foo1 gets its members injected

Container.Bind<IFoo>().FromMethod(GetRandomFoo);

// You an also use an anonymouse delegate directly

Container.Bind<Foo>().FromMethod(ctx => new Foo());

// This is equivalent to AsTransient

Container.Bind<Foo>().FromMethod(ctx => ctx.Container.Instantiate<Foo>());

InstallMore();

}

Foo GetFoo(InjectContext ctx)

{

return new Foo();

}

IFoo GetRandomFoo(InjectContext ctx)

{

switch (UnityEngine.Random.Range(0, 3))

{

case 0:

{

return ctx.Container.Instantiate<Foo1>();

}

case 1:

{

return ctx.Container.Instantiate<Foo2>();

}

}

return ctx.Container.Instantiate<Foo3>();

}

void InstallMore()

{

///////////// FromResolveGetter

// Bind to a property on another dependency

// This can be helpful to reduce coupling between classes

Container.Bind<Foo>().AsSingle();

Container.Bind<Bar>().FromResolveGetter<Foo>(foo => foo.GetBar());

// Another example using values

Container.Bind<string>().FromResolveGetter<Foo>(foo => foo.GetTitle());

///////////// FromNewComponentOnNewGameObject

// Create a new game object at the root of the scene and add the Foo MonoBehaviour to it

Container.Bind<Foo>().FromNewComponentOnNewGameObject().AsSingle();

// You can also specify the game object name to use using WithGameObjectName

Container.Bind<Foo>().FromNewComponentOnNewGameObject().WithGameObjectName("Foo1").AsSingle();

// Bind to an interface instead

Container.Bind<IFoo>().To<Foo>().FromNewComponentOnNewGameObject().AsSingle();

///////////// FromComponentInNewPrefab (singleton)

// Create a new game object at the root of the scene using the given prefab

// After zenject creates a new GameObject from the given prefab, it will

// search the prefab for a component of type 'Foo' and return that

GameObject prefab = null;

Container.Bind<Foo>().FromComponentInNewPrefab(prefab).AsSingle();

// Bind to interface instead

Container.Bind<IFoo>().To<Foo>().FromComponentInNewPrefab(prefab).AsSingle();

// You can also add multiple components

// Note here that only one instance of the given prefab will be

// created

// For this to work, there must be both a Foo MonoBehaviour and

// a Bar MonoBehaviour somewhere on the prefab

Container.Bind(typeof(Foo), typeof(Bar)).FromComponentInNewPrefab(prefab).AsSingle();

///////////// FromComponentInNewPrefab (Transient)

// Instantiate a new copy of 'prefab' every time an instance of Foo is

// requested by a constructor parameter, injected field, etc.

Container.Bind<Foo>().FromComponentInNewPrefab(prefab).AsTransient();

// Bind to interface instead

Container.Bind<IFoo>().To<Foo>().FromComponentInNewPrefab(prefab);

///////////// Identifiers

// Bind a globally accessible string with the name 'PlayerName'

// Note however that a better option might be to create a Settings object and bind

// that instead

Container.Bind<string>().WithId("PlayerName").FromInstance("name of the player");

// This is the equivalent of the line above, and is a bit more readable

Container.BindInstance("name of the player").WithId("PlayerName");

// We can also use IDs to bind multiple instances of the same type:

Container.BindInstance("foo").WithId("FooA");

Container.BindInstance("asdf").WithId("FooB");

InstallMore2();

}

// Then when we inject these dependencies we have to use the same ID:

public class Norf

{

[Inject(Id = "FooA")]

string \_foo;

}

public class Qux

{

[Inject(Id = "FooB")]

string \_foo;

}

public void InstallMore2()

{

///////////// AsCached

// In this example, we bind three instances of Foo, including one without an ID

// We have to use AsCached here because Foo is not a singleton, but we also

// do not want a new Foo created every time like AsTransient

// This will result in a maximum of 3 instances of Foo

Container.Bind<Foo>().AsCached();

Container.Bind<Foo>().WithId("FooA").AsCached();

Container.Bind<Foo>().WithId("FooA").AsCached();

InstallMore3();

}

// When an ID is unspecified in an [Inject] field, it will use the first

// instance

// Bindings without IDs can therefore be used as a default and we can

// specify IDs for specific versions of the same type

public class Norf2

{

[Inject]

Foo \_foo;

}

// Qux2.\_foo will be the same instance as Norf2.\_foo

// This is because we are using AsCached rather than AsTransient

public class Qux2

{

[Inject]

Foo \_foo;

[Inject(Id = "FooA")]

Foo \_foo2;

}

public void InstallMore3()

{

///////////// Conditions

// This will make Foo only visible to Bar

// If we add Foo to the constructor of any other class it won't find it

Container.Bind<Foo>().AsSingle().WhenInjectedInto<Bar>();

// Use different implementations of IFoo dependending on which

// class is being injected

Container.Bind<IFoo>().To<Foo1>().AsSingle().WhenInjectedInto<Bar>();

Container.Bind<IFoo>().To<Foo2>().AsSingle().WhenInjectedInto<Qux>();

// Use "Foo1" as the default implementation except when injecting into

// class Qux, in which case use Foo2

// This works because if there is a condition match, that takes precedence

Container.Bind<IFoo>().To<Foo1>().AsSingle();

Container.Bind<IFoo>().To<Foo2>().AsSingle().WhenInjectedInto<Qux>();

// Allow depending on Foo in only a few select classes

Container.Bind<Foo>().AsSingle().WhenInjectedInto(typeof(Bar), typeof(Qux), typeof(Baz));

// Supply "my game" for any strings that are injected into the Gui class with the identifier "Title"

Container.BindInstance("my game").WithId("Title").WhenInjectedInto<Gui>();

// Supply 5 for all ints that are injected into the Gui class

Container.BindInstance(5).WhenInjectedInto<Gui>();

// Supply 5 for all ints that are injected into a parameter or field

// inside type Gui that is named 'width'

// Note that this is usually not a good idea since the name of a field can change

// easily and break the binding but shown here as an example of a more complex

// condition

Container.BindInstance(5.0f).When(ctx =>

ctx.ObjectType == typeof(Gui) && ctx.MemberName == "width");

// Create a new 'Foo' for every class that is created as part of the

// construction of the 'Bar' class

// So if Bar has a constructor parameter of type Qux, and Qux has

// a constructor parameter of type IFoo, a new Foo will be created

// for that case

Container.Bind<IFoo>().To<Foo>().AsTransient().When(

ctx => ctx.AllObjectTypes.Contains(typeof(Bar)));

///////////// Complex conditions example

var foo1 = new Foo();

var foo2 = new Foo();

Container.Bind<Bar>().WithId("Bar1").AsCached();

Container.Bind<Bar>().WithId("Bar2").AsCached();

// Here we use the 'ParentContexts' property of inject context to sync multiple corresponding identifiers

Container.BindInstance(foo1).When(c => c.ParentContexts.Where(x => x.MemberType == typeof(Bar) && x.Identifier == "Bar1").Any());

Container.BindInstance(foo2).When(c => c.ParentContexts.Where(x => x.MemberType == typeof(Bar) && x.Identifier == "Bar2").Any());

// This results in:

Assert.That(Container.ResolveId<Bar>("Bar1").Foo == foo1);

Assert.That(Container.ResolveId<Bar>("Bar2").Foo == foo2);

///////////// FromResolve

// FromResolve does another lookup on the container

// This will result in IBar, IFoo, and Foo, all being bound to the same instance of

// Foo which is assume to exist somewhere on the given prefab

GameObject fooPrefab = null;

Container.Bind<Foo>().FromComponentInNewPrefab(fooPrefab).AsSingle();

Container.Bind<IBar>().To<Foo>().FromResolve();

Container.Bind<IFoo>().To<IBar>().FromResolve();

// This will result in the same behaviour as the above

Container.Bind(typeof(Foo), typeof(IBar), typeof(IFoo)).To<Foo>().FromComponentInNewPrefab(fooPrefab).AsSingle();

InstallMore4();

}

public class FooInstaller : Installer<FooInstaller>

{

public FooInstaller(string foo)

{

}

public override void InstallBindings()

{

}

}

public class FooInstallerWithArgs : Installer<string, FooInstallerWithArgs>

{

public FooInstallerWithArgs(string foo)

{

}

public override void InstallBindings()

{

}

}

void InstallMore4()

{

///////////// Installing Other Installers

// Immediately call InstallBindings() on FooInstaller

FooInstaller.Install(Container);

// Before calling FooInstaller, configure a property of it

Container.BindInstance("foo").WhenInjectedInto<FooInstaller>();

FooInstaller.Install(Container);

// The arguments can also be added to the Installer<> generic arguments to make them

// strongly typed

FooInstallerWithArgs.Install(Container, "foo");

///////////// Manual Use of Container

// This will fill in any parameters marked as [Inject] and also call any [Inject] methods

var foo = new Foo();

Container.Inject(foo);

// Return an instance for IFoo, using the bindings that have been added previously

// Internally it is what is triggered when you fill in a constructor parameter of type IFoo

// Note: It will throw an exception if it cannot find a match

Container.Resolve<IFoo>();

// Same as the above except returns null when it can't find the given type

Container.TryResolve<IFoo>();

// Return a list of 2 instances of type Foo

// Note that in this case simply calling Resolve<IFoo> will trigger an exception

Container.BindInstance(new Foo());

Container.BindInstance(new Foo());

var foos = Container.ResolveAll<IFoo>();

// Create a new instance of Foo and inject on any of its members

// And fill in any constructor parameters Foo might have

Container.Instantiate<Foo>();

GameObject prefab1 = null;

GameObject prefab2 = null;

// Instantiate a new prefab and have any injectables filled in on the prefab

GameObject go = Container.InstantiatePrefab(prefab1);

// Instantiate a new prefab and return a specific monobehaviour

Foo foo2 = Container.InstantiatePrefabForComponent<Foo>(prefab2);

// Add a new component to an existing game object

Foo foo3 = Container.InstantiateComponent<Foo>(go);

}

# CompositeInstaller

## Содержание

Подробности

## Знакомство

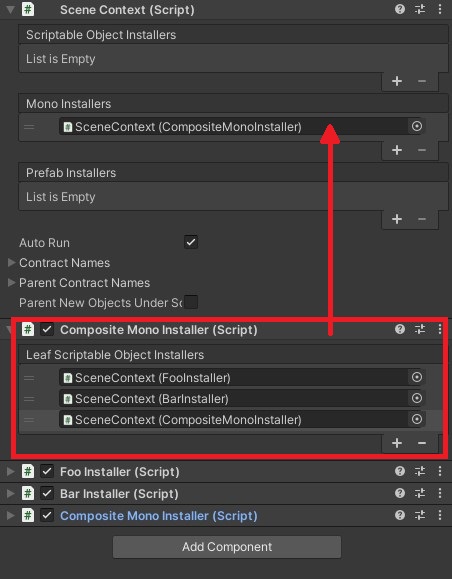
CompositeInstaller — это составной шаблон установщиков Zenject.  
CompositeInstaller может создавать другие установщики и может использоваться следующим образом:

* Многоразовый инсталлятор группы других инсталляторов
* Слабо связанный установщик других установщиков
* Установщик прокси-сервера для установщиков для некоторых функций

Например, предположим, что вы используете ресурс CompositeScriptableObjectInstaller в пакете, предоставленном другим разработчиком.  
Если разработчик добавит несколько установщиков в ресурс CompositeScriptableObjectInstaller в пакете, все, что вам нужно сделать, это обновить пакет, и вы сможете получить функции новых установщиков.

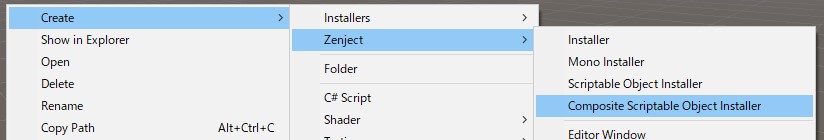
### CompositeMonoInstaller

* Добавьте компонент "CompositeMonoInstaller" в любой GameObject
* Установите для "MonoInstaller" (включая "CompositeMonoInstaller") значение "CompositeMonoInstaller"
* Установите для параметра "CompositeMonoInstaller" значение "Context"

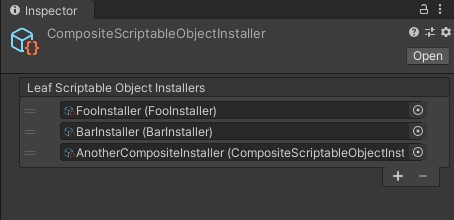
[](https://github.com/modesttree/Zenject/blob/master/Documentation/Images/CompositeInstaller/CompositeMonoInstallerInspector.jpg)

### CompositeScriptableObjectInstaller

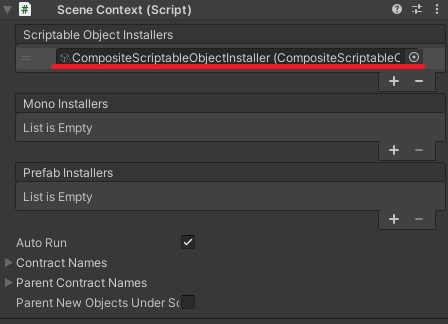
* Выберите, чтобы создать ресурсCreate -> Zenject -> Composite Scriptable Object Installer

[](https://github.com/modesttree/Zenject/blob/master/Documentation/Images/CompositeInstaller/CompositeScriptableObjectInstallerCreateAsset.jpg)

* Присвойте параметру "ScriptableObjectInstaller" (включая "CompositeScriptableObjectInstaller") значение "CompositeScriptableObjectInstaller"

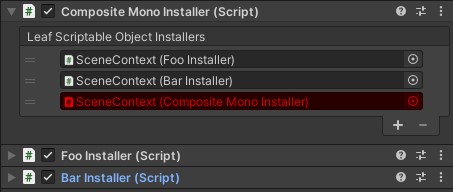
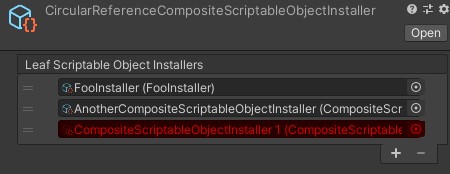
[](https://github.com/modesttree/Zenject/blob/master/Documentation/Images/CompositeInstaller/CompositeScriptableObjectInstallerInspector.jpg)

* Присвойте параметру "CompositeScriptableObjectInstaller" значение любого контекста

[](https://github.com/modesttree/Zenject/blob/master/Documentation/Images/CompositeInstaller/CompositeScriptableObjectInstallerInstall.jpg)

### К вашему сведению:

* Если в составном установщике обнаружены какие-либо циклические ссылки, свойство инспектора будет выделено красным цветом в качестве предупреждения

[](https://github.com/modesttree/Zenject/blob/master/Documentation/Images/CompositeInstaller/CompositeMonoInstallerCircularReference.jpg) [](https://github.com/modesttree/Zenject/blob/master/Documentation/Images/CompositeInstaller/CompositeScriptableObjectInstallerCircularReference.jpg)

# екораторские переплеты

## Содержание

Подробности

## Знакомство

Еще одна особенность Zenject, которая может открыть интересные дизайнерские возможности, — это декораторские крепления. Это позволяет легко реализовать [узор декоратора](https://en.wikipedia.org/wiki/Decorator_pattern).

## Пример

В качестве примера предположим, что у нас есть класс, который содержит характеристики для данного врага в нашей игре, и что у нас есть два врага (орк и демон):

public interface IEnemyStats

{

float Damage

{

get;

}

float Health

{

get;

}

}

public class OrcStats : IEnemyStats

{

public float Damage

{

get { return 1; }

}

public float Health

{

get { return 50; }

}

}

public class DemonStats : IEnemyStats

{

public float Damage

{

get { return 7; }

}

public float Health

{

get { return 20; }

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.Bind<IEnemyStats>().To<OrcStats>().AsSingle();

}

}

Скажем также, что иногда мы хотим улучшать случайных юнитов, чтобы добавить в игру некоторые вариации. Для этого мы можем использовать декораторы:

public class WeaponUpgradeEnemyDecorator : IEnemyStats

{

readonly IEnemyStats \_stats;

public WeaponUpgradeEnemyDecorator(IEnemyStats stats)

{

\_stats = stats;

}

public float Damage

{

get { return \_stats.Damage + 2; }

}

public float Health

{

get { return \_stats.Health; }

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.Bind<IEnemyStats>().To<OrcStats>().AsSingle();

Container.Decorate<IEnemyStats>().With<WeaponUpgradeEnemyDecorator>();

}

}

Используя так, как мы делали здесь, любой использующий код может продолжать использовать то же самое, что и раньше, но с обновлением, чтобы нанести больший ущерб. Это обновление также может быть применено к любому производному классу IEnemyStats, то есть к орку или демону.Container.DecorateIEnemyStats

Мы также можем объединить декораторов вместе, поэтому, если мы хотим применить другое обновление для здоровья, мы можем сделать это одновременно:

public class ShieldUpgradeEnemyDecorator : IEnemyStats

{

readonly IEnemyStats \_stats;

public ShieldUpgradeEnemyDecorator(IEnemyStats stats)

{

\_stats = stats;

}

public float Damage

{

get { return \_stats.Damage; }

}

public float Health

{

get { return \_stats.Health + 20; }

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.Bind<IEnemyStats>().To<OrcStats>().AsSingle();

Container.Decorate<IEnemyStats>().With<WeaponUpgradeEnemyDecorator>();

Container.Decorate<IEnemyStats>().With<ShieldUpgradeEnemyDecorator>();

}

}

Так что теперь характеристики наших орков будут улучшаться как по урону, так и по здоровью.

Обратите внимание, что порядок, в котором мы применяем наших декораторов, не имеет значения в этом случае, но может иметь значение в других случаях. Декораторы будут применяться в том порядке, в котором они добавлены, поэтому в данном случае это будет выглядеть следующим образом ShieldUpgradeEnemyDecorator(WeaponUpgradeEnemyDecorator(OrcStats()))

Еще один простой способ использования декораторов — добавить дополнительное ведение журнала, проверку выходных значений или профилирование существующих интерфейсов. Например:

public interface ISaveGameHandler

{

void SaveGame();

}

public class SaveGameHandler : ISaveGameHandler

{

public void SaveGame()

{

// Some long running operation

}

}

public class SaveGameProfilerDecorator : ISaveGameHandler

{

readonly ISaveGameHandler \_handler;

public SaveGameProfilerDecorator(ISaveGameHandler handler)

{

\_handler = handler;

}

public void SaveGame()

{

var stopwatch = new Stopwatch();

stopwatch.Start();

\_handler.SaveGame();

stopwatch.Stop();

Debug.Log(string.format("Took {0:0.00} seconds to save the game!", stopwatch.Elapsed.TotalSeconds));

}

}

public void InstallBindings()

{

Container.Bind<ISaveGameHandler>().To<SaveGameHandler>().AsSingle();

Container.Decorate<ISaveGameHandler>().With<SaveGameProfilerDecorator>();

}

## Синтаксис привязки

Container.Decorate<**ContractType**>()

.With<**DecoratorType**>()

.From**ConstructionMethod**()

.WithArguments(**Arguments**)

.(**Copy**|**Move**)Into(**All**|**Direct**)SubContainers();

Где:

* **ContractType** = тип, который декорируется. Объект этого типа будет внедрен в класс DecoratorType.
* **DecoratorType** = класс декоратора. Это должен быть конкретный тип, а также должен принимать в качестве впрыскиваемого параметра/поля объект типа ContractType.

Другие значения имеют тот же эффект, что и [описанный здесь](https://github.com/modesttree/Zenject/blob/master/README.md#binding).

Обратите внимание, что мы можем определить любой метод построения From, который мы хотим здесь - нам не нужно по умолчанию использовать FromNew, как в приведенных выше примерах.

# Заводов

## Содержание

Подробности

## Динамическое создание объектов с фабриками

Одна из вещей, которая часто сбивает с толку людей, плохо знакомых с внедрением зависимостей, — это вопрос о том, как динамически создавать новые объекты после полного запуска приложения/игры. Например, если вы пишете игру, в которой вы порождаете новых врагов на протяжении всей игры, то вам нужно будет создать новые экземпляры класса «Враг», и вы захотите убедиться, что этот объект будет внедрен с зависимостями, как и все объекты, которые являются частью исходного графа объектов. Рекомендуемый способ добиться этого — использовать фабрики.

Как и в случае с основной документацией, я рекомендую хотя бы прочитать раздел «Введение», а затем, при необходимости, пропустить раздел «Дополнительно»

## Знакомство:

### Теория

Помните, что важной частью внедрения зависимостей является резервирование использования контейнера строго для «корневого слоя композиции». Класс контейнера включается в качестве зависимости сам по себе автоматически, поэтому ничто не мешает вам игнорировать это правило и вставлять контейнер в любые классы, которые вы хотите. Например, будет работать следующий код:(DiContainer)

public class Enemy

{

DiContainer Container;

public Enemy(DiContainer container)

{

Container = container;

}

public void Update()

{

...

var player = Container.Resolve<Player>();

WalkTowards(player.Position);

...

etc.

}

}

Однако приведенный выше код является примером антишаблона. Это будет работать, и вы можете использовать контейнер для получения доступа ко всем другим классам в вашем приложении, однако, если вы сделаете это, вы не сможете воспользоваться преимуществами внедрения зависимостей. Это, кстати, известно как [шаблон Service Locator](https://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/).

Обратите внимание, что единственным исключением из этого правила являются заводы и установщики. Опять же, заводы и установщики составляют то, что мы называем «корневым слоем композиции».

Конечно, способ внедрения зависимостей будет следующим:

public class Enemy

{

Player \_player;

public Enemy(Player player)

{

\_player = player;

}

public void Update()

{

...

WalkTowards(\_player.Position);

...

}

}

Но теперь каждое место, где нужно создать новый экземпляр, также должно предоставить экземпляр , и мы вернулись к проблеме, упомянутой [в основном разделе теории](https://github.com/modesttree/Zenject/blob/master/README.md#theory). Поэтому, чтобы решить эту проблему, фабрики должны использоваться для создания каждого динамического экземпляра, чтобы гарантировать, что эти дополнительные зависимости будут заполнены zenject.EnemyPlayer

### Пример

Рекомендуемый способ сделать это в Zenject следующий:

public class Player

{

}

public class Enemy

{

readonly Player \_player;

public Enemy(Player player)

{

\_player = player;

}

public class Factory : PlaceholderFactory<Enemy>

{

}

}

public class EnemySpawner : ITickable

{

readonly Enemy.Factory \_enemyFactory;

public EnemySpawner(Enemy.Factory enemyFactory)

{

\_enemyFactory = enemyFactory;

}

public void Tick()

{

if (ShouldSpawnNewEnemy())

{

var enemy = \_enemyFactory.Create();

// ...

}

}

}

public class TestInstaller : MonoInstaller

{

public override void InstallBindings()

{

Container.BindInterfacesTo<EnemySpawner>().AsSingle();

Container.Bind<Player>().AsSingle();

Container.BindFactory<Enemy, Enemy.Factory>();

}

}

При использовании вышеуказанного вместо , все зависимости для класса Enemy (например, Player) будут автоматически заполнены.Enemy.Factorynew Enemy

Мы также можем добавить параметры времени выполнения на нашу фабрику. Например, предположим, что мы хотим рандомизировать скорость каждого врага, чтобы добавить в нашу игру интересную вариацию. Нашим классом врагов становится:

public class Enemy

{

readonly Player \_player;

readonly float \_speed;

public Enemy(float speed, Player player)

{

\_player = player;

\_speed = speed;

}

public class Factory : PlaceholderFactory<float, Enemy>

{

}

}

public class EnemySpawner : ITickable

{

readonly Enemy.Factory \_enemyFactory;

public EnemySpawner(Enemy.Factory enemyFactory)

{

\_enemyFactory = enemyFactory;

}

public void Tick()

{

if (ShouldSpawnNewEnemy())

{

var newSpeed = Random.Range(MIN\_ENEMY\_SPEED, MAX\_ENEMY\_SPEED);

var enemy = \_enemyFactory.Create(newSpeed);

// ...

}

}

}

public class TestInstaller : MonoInstaller

{

public override void InstallBindings()

{

Container.BindInterfacesTo<EnemySpawner>().AsSingle();

Container.Bind<Player>().AsSingle();

Container.BindFactory<float, Enemy, Enemy.Factory>();

}

}

Динамические параметры, предоставляемые конструктору, объявляются путем предоставления дополнительных универсальных аргументов базовому классу . содержит метод с заданными типами параметров, который затем может быть вызван другими классами, такими как .EnemyPlaceholderFactory<>Enemy.FactoryPlaceholderFactory<>CreateEnemySpawner

Enemy.Factory всегда намеренно остается пустым и просто является производным от встроенного класса Zenject, который обрабатывает работу по использованию DiContainer для создания нового экземпляра . Он вызывается потому, что на самом деле не контролирует, как создается объект напрямую. Способ создания объекта объявляется в установщике так же, как и для зависимостей, отличных от заводских. Например, если бы наш класс был MonoBehaviour на префабе, мы могли бы установить его следующим образом:PlaceholderFactory<>EnemyPlaceholderFactoryEnemy

public class Enemy : MonoBehaviour

{

Player \_player;

// Note that we can't use a constructor anymore since we are a MonoBehaviour now

[Inject]

public void Construct(Player player)

{

\_player = player;

}

public class Factory : PlaceholderFactory<Enemy>

{

}

}

public class TestInstaller : MonoInstaller

{

public GameObject EnemyPrefab;

public override void InstallBindings()

{

Container.BindInterfacesTo<EnemySpawner>().AsSingle();

Container.Bind<Player>().AsSingle();

Container.BindFactory<Enemy, Enemy.Factory>().FromComponentInNewPrefab(EnemyPrefab);

}

}

И точно так же, если вы хотите создать экземпляр своего динамического объекта с помощью , , , , и т. Д. (см. [раздел «Переплет](https://github.com/modesttree/Zenject/blob/master/README.md#binding)» для получения полной информации)FromMethodFromNewComponentOnNewGameObjectFromInstanceFromSubContainerResolve

Использование может быть особенно полезным, если динамически создаваемый объект имеет множество собственных зависимостей. Вы можете заставить его вести себя как «Фасад» (см. [раздел «Подконтейнеры](https://github.com/modesttree/Zenject/blob/master/Documentation/SubContainers.md)» для получения подробной информации о вложенных контейнерах / фасадах)FromSubContainerResolve

Нет требования, чтобы класс был вложенным классом, однако мы обнаружили, что это очень полезное соглашение. В обоих приведенных выше примерах мы могли бы установить его таким образом и обойти необходимость во вложенном фабричном классе:Enemy.FactoryEnemy,

Container.BindFactory<Enemy, PlaceholderFactory<Enemy>>()

Однако у этого есть несколько недостатков:

1. Изменение списка параметров не обнаруживается во время компиляции. Если он внедряется непосредственно по всей кодовой базе, когда мы добавляем параметр скорости и, следовательно, меняем его на , то мы получим ошибки времени выполнения, когда zenject не сможет найти зависимость (или ошибки проверки, если вы используете проверку). Однако, если мы используем производный класс, то, если мы позже решим получить от него, мы получим ошибки компилятора (в каждом месте, которое вызывает метод), которые легче пойматьPlaceholderFactory<Enemy>PlaceholderFactory<float, Enemy>PlaceholderFactory<Enemy>Enemy.FactoryPlaceholderFactory<float, Enemy>Create
2. Это менее многословно. Внедрение везде гораздо более читабельно, чем , особенно по мере роста списка параметров.Enemy.FactoryPlaceholderFactory<float, Enemy>

Вот почему мы рекомендуем вместо этого использовать вложенный фабричный класс.

Другие вещи, о которых следует знать:

* Проверка может быть особенно полезна для динамически создаваемых объектов, так как в противном случае вы не сможете обнаружить ошибку до тех пор, пока фабрика не будет вызвана в какой-то момент во время выполнения (см. [раздел проверки](https://github.com/modesttree/Zenject/blob/master/README.md#object-graph-validation) для получения более подробной информации о проверке)
* Обратите внимание, что для динамически создаваемых экземпляров MonoBehaviours (например, при использовании с инъекцией всегда должно происходить до и , поэтому мы рекомендуем использовать / для логики инициализации и использовать метод inject строго для сохранения зависимостей (т.е. аналогично конструкторам для немоноповеденческих моделей)FromComponentInNewPrefabBindFactory)AwakeStartAwakeStart
* В отличие от незаводского внедрения, у вас может быть несколько параметров среды выполнения, объявленных с одним и тем же типом. В этом случае порядок, в котором значения передаются на фабрику, будет соответствовать порядку параметров - при условии, что вы используете конструктор или внедрение метода. Однако обратите внимание, что это не относится к полевым или имущественным инъекциям. В этих случаях порядок введения значений не гарантирует, что он будет соответствовать порядку объявления, так как эти поля извлекаются с помощью, что не гарантирует порядок, как описано [здесь](https://docs.microsoft.com/en-us/dotnet/api/system.type.getfields?redirectedfrom=MSDN&view=netcore-3.1#System_Type_GetFields)Type.GetFields

### Синтаксис привязки

Container.BindFactory<**ContractType**, **PlaceholderFactoryType**>()

.WithId(**Identifier**)

.WithFactoryArguments(**Factory Arguments**)

.To<**ResultType**>()

.From**ConstructionMethod**()

.As**Scope**()

.WithArguments(**Arguments**)

.OnInstantiated(**InstantiatedCallback**)

.When(**Condition**)

.NonLazy()

.(**Copy**|**Move**)Into(**All**|**Direct**)SubContainers();

Где:

* **ContractType** = тип контракта, возвращенный из заводского методаCreate
* **PlaceholderFactoryType** = класс, производный от PlaceholderFactory<>
* **WithFactoryArguments** = Если вы хотите вставить дополнительные аргументы в производный класс фабрики-заполнителя, вы можете включить их сюда. Обратите внимание, что это относится к фактическому созданному типу, а не к фабрике.WithArguments
* **Scope** = Обратите внимание, что в отличие от незаводских привязок, по умолчанию используется AsCached, а не AsTransient, что почти всегда требуется для фабрик, поэтому в большинстве случаев вы можете оставить это неуказанным.

Другие методы привязки имеют ту же функциональность, что и [незаводские привязки](https://github.com/modesttree/Zenject/blob/master/README.md#binding).

### Абстрактные фабрики

Приведенное выше описание фабрик отлично подходит для большинства случаев, однако бывают случаи, когда вы не хотите напрямую зависеть от конкретного класса, а вместо этого хотите, чтобы ваша фабрика возвращала интерфейс. Такая фабрика называется абстрактной фабрикой.

Давайте создадим пример сценария, в котором у нас есть несколько различных реализаций данного интерфейса:

public interface IPathFindingStrategy

{

...

}

public class AStarPathFindingStrategy : IPathFindingStrategy

{

...

}

public class RandomPathFindingStrategy : IPathFindingStrategy

{

...

}

Для этого примера давайте также предположим, что мы должны создать экземпляр IPathFindingStrategy во время выполнения. В противном случае это было бы так же просто, как выполнить в одном из наших установщиков.Container.Bind<IPathFindingStrategy>().To<TheImplementationWeWant>().AsSingle();

Это делается очень похоже на то, как работают неабстрактные фабрики. Одно из отличий состоит в том, что мы не можем включить фабрику в качестве вложенного класса в интерфейс (это запрещено в C#), но в остальном это ничем не отличается:

public class PathFindingStrategyFactory : PlaceholderFactory<IPathFindingStrategy>

{

}

public class GameController : IInitializable

{

PathFindingStrategyFactory \_strategyFactory;

IPathFindingStrategy \_strategy;

public GameController(PathFindingStrategyFactory strategyFactory)

{

\_strategyFactory = strategyFactory;

}

public void Initialize()

{

\_strategy = \_strategyFactory.Create();

// ...

}

}

public class GameInstaller : MonoInstaller

{

public bool UseAStar;

public override void InstallBindings()

{

Container.BindInterfacesTo<GameController>().AsSingle();

if (UseAStar)

{

Container.BindFactory<IPathFindingStrategy, PathFindingStrategyFactory>().To<AStarPathFindingStrategy>();

}

else

{

Container.BindFactory<IPathFindingStrategy, PathFindingStrategyFactory>().To<RandomPathFindingStrategy>();

}

}

}

## Продвинутый:

### Пользовательские фабрики

Хорошо, но что, если я не знаю, какой тип я хочу создать, пока приложение не будет запущено? Или что, если у меня есть особые требования к созданию экземпляров класса «Враг», на которые не распространяется ни один из методов строительства?

В этих случаях вы можете создать то, что мы называем «пользовательской фабрикой», а затем напрямую вызывать или использовать [методы в DiContainer](https://github.com/modesttree/Zenject/blob/master/README.md#dicontainer-methods) или использовать любой метод, необходимый для создания объекта. Например, продолжая предыдущий пример с фабрикой, предположим, что вы хотите иметь возможность изменять значение времени выполнения (сложность), которое определяет, какие типы врагов будут созданы.new Enemy

public enum Difficulties

{

Easy,

Hard,

}

public interface IEnemy

{

}

public class EnemyFactory : PlaceholderFactory<IEnemy>

{

}

public class Demon : IEnemy

{

}

public class Dog : IEnemy

{

}

public class DifficultyManager

{

public Difficulties Difficulty

{

get;

set;

}

}

public class CustomEnemyFactory : IFactory<IEnemy>

{

DiContainer \_container;

DifficultyManager \_difficultyManager;

public CustomEnemyFactory(DiContainer container, DifficultyManager difficultyManager)

{

\_container = container;

\_difficultyManager = difficultyManager;

}

public IEnemy Create()

{

if (\_difficultyManager.Difficulty == Difficulties.Hard)

{

return \_container.Instantiate<Demon>();

}

return \_container.Instantiate<Dog>();

}

}

public class GameController : IInitializable

{

readonly EnemyFactory \_enemyFactory;

public GameController(EnemyFactory enemyFactory)

{

\_enemyFactory = enemyFactory;

}

public void Initialize()

{

var enemy = \_enemyFactory.Create();

// ...

}

}

public class TestInstaller : MonoInstaller

{

public override void InstallBindings()

{

Container.BindInterfacesTo<GameController>().AsSingle();

Container.Bind<DifficultyManager>().AsSingle();

Container.BindFactory<IEnemy, EnemyFactory>().FromFactory<CustomEnemyFactory>();

}

}

Другими словами, создайте новый класс, производный от привязки, а затем используйте метод в привязке для его подключения.IFactory<Enemy>FromFactory

Вы также можете напрямую позвонить и здесь вместо использования DiContainer (хотя в этом случае и их члены не будут вводиться).new Dog()new Demon()DogDemon

Обратите внимание, что на самом деле это сокращение, как описано в [разделе привязки](https://github.com/modesttree/Zenject/blob/master/README.md#binding). Использование вместо — это более эффективный способ указания пользовательских фабрик, так как пользовательская фабрика может быть создана с использованием любого метода построения, включая , , , и т. д.FromFactory<CustomEnemyFactory>()FromIFactory(b => b.To<CustomEnemyFactory>().AsCached());FromIFactoryFromFactoryFromSubContainerResolveFromInstanceFromComponentInNewPrefab

Одна из проблем с нашим выше заключается в том, что он не проверяется правильно. Если мы добавим зависимости к классам or, и эти зависимости не будут привязаны ни к каким установщикам, то мы не узнаем об этом до времени выполнения. Поэтому, если мы не проверим каждый уровень сложности, может пройти некоторое время, прежде чем мы осознаем эту проблему.CustomEnemyFactoryDemonDog

Таким образом, лучший способ сделать это будет следующим:

public class CustomEnemyFactory : IFactory<IEnemy>

{

Dog.Factory \_dogFactory;

Demon.Factory \_demonFactory;

DifficultyManager \_difficultyManager;

public CustomEnemyFactory(

DifficultyManager difficultyManager, Dog.Factory dogFactory, Demon.Factory demonFactory)

{

\_dogFactory = dogFactory;

\_demonFactory = demonFactory;

\_difficultyManager = difficultyManager;

}

public IEnemy Create()

{

if (\_difficultyManager.Difficulty == Difficulties.Hard)

{

return \_demonFactory.Create();

}

return \_dogFactory.Create();

}

}

С приведенным выше изменением все зависимости, отсутствующие в списке параметров конструктора demon или dog, будут перехвачены во время проверки, а не во время выполнения.

Обратите внимание, что если вы настаиваете на использовании методов DiContainer напрямую, вы все равно можете проверить необходимые зависимости, выполнив заводскую реализацию, как описано [здесь](https://github.com/modesttree/Zenject/blob/master/Documentation/Factories.md#implementing-validatable).IValidatable

### Прямое использование IFactory

Если вы вообще не хотите определять какие-либо дополнительные фабричные классы, вы можете внедрить их непосредственно в любые используемые классы, а затем использовать метод, чтобы подключить их к методу построения. Чтобы повторно использовать приведенный выше пример, это будет выглядеть следующим образом:IFactory<>BindIFactory

public class GameController : IInitializable

{

IFactory<IPathFindingStrategy> \_strategyFactory;

IPathFindingStrategy \_strategy;

public GameController(IFactory<IPathFindingStrategy> strategyFactory)

{

\_strategyFactory = strategyFactory;

}

public void Initialize()

{

\_strategy = \_strategyFactory.Create();

// ...

}

}

public class GameInstaller : MonoInstaller

{

public bool UseAStar;

public override void InstallBindings()

{

Container.BindInterfacesTo<GameController>().AsSingle();

if (UseAStar)

{

Container.BindIFactory<IPathFindingStrategy>().To<AStarPathFindingStrategy>();

}

else

{

Container.BindIFactory<IPathFindingStrategy>().To<RandomPathFindingStrategy>();

}

}

}

В некоторых случаях это может быть проще, чем производное, однако при непосредственном использовании возникают [те же проблемы](https://github.com/modesttree/Zenject/blob/master/Documentation/Factories.md#using-factories-directly), что и упомянутые выше (то есть он более подвержен ошибкам при изменении списка параметров, а в некоторых случаях может быть более подробным)PlaceholderFactoryPlaceholderFactory

### Пользовательский заводской интерфейс

В некоторых случаях может потребоваться избежать прямой связи с фабричным классом и вместо этого предпочесть использовать базовый класс или пользовательский интерфейс. Вы можете сделать это, используя метод, а не так:BindFactoryCustomInterfaceBindFactory

public interface IMyFooFactory : IFactory<Foo>

{

}

public class Foo

{

public class Factory : PlaceholderFactory<Foo>, IMyFooFactory

{

}

}

public class Runner : IInitializable

{

readonly IMyFooFactory \_fooFactory;

public Runner(IMyFooFactory fooFactory)

{

\_fooFactory = fooFactory;

}

public void Initialize()

{

var foo = \_fooFactory.Create();

// ...

}

}

public class FooInstaller : MonoInstaller<FooInstaller>

{

public override void InstallBindings()

{

Container.BindFactoryCustomInterface<Foo, Foo.Factory, IMyFooFactory>();

}

}

Обратите внимание, что существует эквивалентный метод для пулов памяти, вызываемых такжеBindMemoryPoolCustomInterface

### Сборная фабрика

В некоторых случаях может потребоваться, чтобы код, вызывающий метод Create, также предоставил префаб для использования для нового объекта. Вы можете вызвать напрямую, но это нарушило бы наше правило о внедрении DiContainer только в «корневой слой композиции» (т.е. фабрики и установщики), поэтому было бы лучше вместо этого написать пользовательскую фабрику, подобную этой:DiContainer.InstantiatePrefabForComponent

public class Foo

{

public class Factory : PlaceholderFactory<UnityEngine.Object, Foo>

{

}

}

public class FooFactory : IFactory<UnityEngine.Object, Foo>

{

readonly DiContainer \_container;

public FooFactory(DiContainer container)

{

\_container = container;

}

public Foo Create(UnityEngine.Object prefab)

{

return \_container.InstantiatePrefabForComponent<Foo>(prefab);

}

}

public override void InstallBindings()

{

Container.BindFactory<UnityEngine.Object, Foo, Foo.Factory>().FromFactory<FooFactory>();

}

However, this kind of custom factory is common enough that there is a helper class included for this purpose called PrefabFactory. So you could just do this instead:

public class Foo

{

public class Factory : PlaceholderFactory<UnityEngine.Object, Foo>

{

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public GameObject Prefab;

public override void InstallBindings()

{

Container.BindFactory<UnityEngine.Object, Foo, Foo.Factory>().FromFactory<PrefabFactory<Foo>>();

}

}

A similar helper class is provided when instantiating a prefab from a resource path. For example:

public class Foo

{

public class Factory : PlaceholderFactory<string, Foo>

{

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public GameObject Prefab;

public override void InstallBindings()

{

Container.BindFactory<string, Foo, Foo.Factory>().FromFactory<PrefabResourceFactory<Foo>>();

}

}

Одна вещь, о которой следует помнить при использовании PrefabResource или PrefabResourceFactory, заключается в том, что проверка в этих случаях не выполняется. Таким образом, если бы в нашем классе Foo выше отсутствовала зависимость, мы бы не обнаружили это до времени выполнения. Это невозможно, так как для проверки требуется сборная конструкция.

### Внедрение IValidatable

Если вам нужно использовать методы создания экземпляров DiContainer напрямую, но вы все равно хотите проверить динамически создаваемые графы объектов, вы все равно можете сделать это, реализовав интерфейс. Чтобы повторно использовать тот же пример из приведенного выше, это будет выглядеть следующим образом:IValidatable

public class CustomEnemyFactory : IFactory<IEnemy>, IValidatable

{

DiContainer \_container;

DifficultyManager \_difficultyManager;

public CustomEnemyFactory(DiContainer container, DifficultyManager difficultyManager)

{

\_container = container;

\_difficultyManager = difficultyManager;

}

public IEnemy Create()

{

if (\_difficultyManager.Difficulty == Difficulties.Hard)

{

return \_container.Instantiate<Demon>();

}

return \_container.Instantiate<Dog>();

}

public void Validate()

{

\_container.Instantiate<Dog>();

\_container.Instantiate<Demon>();

}

}

public class TestInstaller : MonoInstaller

{

public override void InstallBindings()

{

Container.BindFactory<IEnemy, EnemyFactory>().FromFactory<CustomEnemyFactory>();

}

}

Обратите внимание, что нет необходимости привязывать интерфейс к нашему заводу. Для вызова метода достаточно просто реализовать интерфейс, а также сделать нашу фабрику частью графа объектов.IValidatableIValidatableValidate

В этом методе, чтобы вручную проверить графы динамических объектов, вы просто создаете их экземпляры. Обратите внимание, что на самом деле это не приведет к созданию экземпляров этих объектов (эти вызовы фактически возвращают null здесь). Суть в том, чтобы сделать «пробный прогон», фактически ничего не создавая экземпляров, чтобы доказать полный граф объектов. Дополнительные сведения о проверке см. в [разделе проверки](https://github.com/modesttree/Zenject/blob/master/README.md#object-graph-validation).Validate

# Пулы памяти

## Содержание

Подробности

## Знакомство

### Пример

Прежде чем разбираться в пулах памяти, было бы полезно понять фабрики, поэтому, пожалуйста, сначала прочитайте [введение в фабрики](https://github.com/modesttree/Zenject/blob/master/Documentation/Factories.md).

Разработка игр в Unity не займет много времени, прежде чем вы поймете, что правильное управление памятью очень важно, если вы хотите, чтобы ваша игра работала без сбоев (особенно на мобильных устройствах). В зависимости от ограничений платформы и типа игры, над которой вы работаете, может быть очень важно максимально избегать ненужного распределения кучи. Одним из очень эффективных способов сделать это является использование пулов памяти.

В качестве примера рассмотрим случай, когда мы динамически создаем класс:

public class Foo

{

public class Factory : PlaceholderFactory<Foo>

{

}

}

public class Bar

{

readonly Foo.Factory \_fooFactory;

readonly List<Foo> \_foos = new List<Foo>();

public Bar(Foo.Factory fooFactory)

{

\_fooFactory = fooFactory;

}

public void AddFoo()

{

\_foos.Add(\_fooFactory.Create());

}

public void RemoveFoo()

{

\_foos.RemoveAt(0);

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.Bind<Bar>().AsSingle();

Container.BindFactory<Foo, Foo.Factory>();

}

}

Здесь каждый раз, когда мы его вызываем, всегда будет выделяться новая куча памяти. И каждый раз, когда мы вызываем , класс перестает ссылаться на один из экземпляров, и поэтому память для этого экземпляра будет помечена для сборки мусора. Если это произойдет достаточное количество раз, то в конце концов сработает сборщик мусора, и вы получите (иногда очень заметный) всплеск в своей игре.Bar.AddFooBar.RemoveFooBarFoo,

Мы можем исправить этот всплеск, используя вместо этого пулы памяти:

public class Foo

{

public class Pool : MemoryPool<Foo>

{

}

}

public class Bar

{

readonly Foo.Pool \_fooPool;

readonly List<Foo> \_foos = new List<Foo>();

public Bar(Foo.Pool fooPool)

{

\_fooPool = fooPool;

}

public void AddFoo()

{

\_foos.Add(\_fooPool.Spawn());

}

public void RemoveFoo()

{

var foo = \_foos[0];

\_fooPool.Despawn(foo);

\_foos.Remove(foo);

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.Bind<Bar>().AsSingle();

Container.BindMemoryPool<Foo, Foo.Pool>();

}

}

Как вы можете видеть, это работает очень похоже на фабрики, за исключением того, что терминология немного отличается ( вместо , а не ) и, в отличие от фабрик, вы должны вернуть экземпляр в пул, а не просто перестать ссылаться на него.PoolFactorySpawnCreate

В этой новой реализации, описанной выше, при каждом вызове к , будет происходить некоторое начальное выделение кучи, но если вы вызовете, то последовательно это приведет к повторному использованию предыдущего экземпляра и, следовательно, сэкономит вам выделение кучи.AddFoo()RemoveFoo()AddFoo()

Это лучше, но мы, возможно, все равно захотим избежать пиков из-за первоначального распределения кучи. Один из способов сделать это — сделать все распределения кучи сразу при запуске игры:

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.Bind<Bar>().AsSingle();

Container.BindMemoryPool<Foo, Foo.Pool>().WithInitialSize(10);

}

}

Когда мы используем это в операторе Bind для нашего пула, сразу после запуска будет создано 10 экземпляров для заполнения пула.WithInitialSizeFoo

### Синтаксис привязки

Синтаксис пулов памяти практически идентичен синтаксису фабрик, с несколькими новыми методами привязки, такими как и . Кроме того, в отличие от , нет необходимости указывать параметры фабрики в качестве общих аргументов для WithInitialSizeExpandByBindFactoryBindMemoryPool

Как и в случае с фабриками, рекомендуемым соглашением является использование открытого вложенного класса с именем Pool (хотя это всего лишь соглашение)

public class Foo

{

public class Pool : MemoryPool<Foo>

{

}

}

Параметры добавляются путем добавления общих аргументов:

public class Foo

{

public class Pool : MemoryPool<string, int, Foo>

{

}

}

Полный формат привязки следующий:

Container.BindMemoryPool<**ObjectType, MemoryPoolType**>()

.With**(InitialSize|FixedSize)**

.WithMaxSize(**MaxSize**)

.ExpandBy**(OneAtATime|Doubling)**()

.WithFactoryArguments(**Factory Arguments**)

.To<**ResultType**>()

.WithId(**Identifier**)

.From**ConstructionMethod**()

.As**Scope**()

.WithArguments(**Arguments**)

.OnInstantiated(**InstantiatedCallback**)

.When(**Condition**)

.CopyIntoAllSubContainers()

.NonLazy();

Где:

* **InitialSize** — определяет, сколько элементов будет заполнено в пул при запуске. Может быть полезно установить здесь значение, чтобы избежать скачков, вызванных распределением во время игры.
* **FixedSize** - Если этот параметр установлен, пул изначально будет заполнен суммой, указанной здесь, и если эта сумма будет превышена, будут выброшены исключения.
* **MaxSize** — если в пул возвращается достаточное количество элементов, размер которого превышает это значение, то оставшиеся элементы будут уничтожены. Это может быть полезно для обеспечения того, чтобы пул занимал максимальный объем памяти.
* **ObjectType** = тип класса, экземпляр которого создается пулом памяти
* MemoryPoolType = тип производного класса **MemoryPool**, который часто является вложенным классом .ObjectType
* **ExpandBy** = определяет поведение, вызываемое при достижении максимального размера пула. Обратите внимание, что при указании эта опция недоступна. Возможны следующие варианты:WithFixedSize
  + **ExpandByOneAtATime** — выделяйте новые экземпляры только по одному по мере необходимости.
  + **ExpandByDoublebling** — когда пул заполнен и запрашивается новый экземпляр, пул удваивается в размерах, прежде чем вернуть запрошенный экземпляр. Этот подход может быть полезен, если вы предпочитаете иметь большие нечастые выделения множеству небольших частых выделений
* **WithFactoryArguments** = Если вы хотите вставить дополнительные аргументы в производный класс MemoryPool, вы можете включить их сюда. Обратите внимание, что это относится к фактическому созданному типу, а не к пулу памяти.WithArguments
* **Scope** = Обратите внимание, что в отличие от обычных привязок, по умолчанию используется AsCached, а не AsTransient, что почти всегда является тем, что вам нужно, поэтому в большинстве случаев вы можете оставить это неуказанным. Это только в некоторых редких случаях, когда вам нужно иметь уникальные пулы для каждого класса, который использует пул.

Остальные методы привязки ведут себя так же, как и обычные методы привязки, описанные [здесь](https://github.com/modesttree/Zenject/blob/master/README.md#binding)

### Сброс элементов в пуле

Одна очень важная вещь, о которой следует помнить при использовании пулов памяти вместо фабрик, заключается в том, что вы должны убедиться, что полностью «сбросили» данный экземпляр. Это необходимо, иначе у вас может быть состояние от предыдущей «жизни» экземпляра к поведению нового экземпляра.

Объект можно сбросить, реализовав любой из следующих методов в производном классе пула памяти:

public class Foo

{

public class Pool : MemoryPool<Foo>

{

protected override void OnCreated(Foo item)

{

// Called immediately after the item is first added to the pool

}

protected override void OnDestroyed(Foo item)

{

// Called immediately after the item is removed from the pool without also being spawned

// This occurs when the pool is shrunk either by using WithMaxSize or by explicitly shrinking the pool by calling the `ShrinkBy` / `Resize methods

}

protected override void OnSpawned(Foo item)

{

// Called immediately after the item is removed from the pool

}

protected override void OnDespawned(Foo item)

{

// Called immediately after the item is returned to the pool

}

protected override void Reinitialize(Foo foo)

{

// Similar to OnSpawned

// Called immediately after the item is removed from the pool

// This method will also contain any parameters that are passed along

// to the memory pool from the spawning code

}

}

}

В большинстве случаев вам, вероятно, придется реализовать только метод Reinitialize. Например, давайте введем некоторое состояние в наш первый пример, добавив значение Position в :Foo

public class Foo

{

Vector3 \_position = Vector3.zero;

public void Move(Vector3 delta)

{

\_position += delta;

}

public class Pool : MemoryPool<Foo>

{

protected override void Reinitialize(Foo foo)

{

foo.\_position = Vector3.zero;

}

}

}

Обратите внимание, что наш класс pool свободен для доступа к закрытым переменным внутри из-за того, что это вложенный класс.Foo

Или, если бы мы хотели избежать дублирования в и , мы могли бы сделать это следующим образом:FooFoo.Pool

public class Foo

{

Vector3 \_position;

public Foo()

{

Reset();

}

public void Move(Vector3 delta)

{

\_position += delta;

}

void Reset()

{

\_position = Vector3.zero;

}

public class Pool : MemoryPool<Foo>

{

protected override void Reinitialize(Foo foo)

{

foo.Reset();

}

}

}

### Параметры среды выполнения

Как и в случае с фабриками, вы также можете передавать параметры среды выполнения при создании новых экземпляров классов в пуле. Разница в том, что вместо параметров, вводимых в класс, они передаются методу:Reinitialize

public class Foo

{

Vector3 \_position;

Vector3 \_velocity;

public Foo()

{

Reset(Vector3.zero);

}

public void Tick()

{

\_position += \_velocity \* Time.deltaTime;

}

void Reset(Vector3 velocity)

{

\_position = Vector3.zero;

\_velocity = Vector3.zero;

}

public class Pool : MemoryPool<Vector3, Foo>

{

protected override void Reinitialize(Vector3 velocity, Foo foo)

{

foo.Reset(velocity);

}

}

}

public class Bar

{

readonly Foo.Pool \_fooPool;

readonly List<Foo> \_foos = new List<Foo>();

public Bar(Foo.Pool fooPool)

{

\_fooPool = fooPool;

}

public void AddFoo()

{

float maxSpeed = 10.0f;

float minSpeed = 1.0f;

\_foos.Add(\_fooPool.Spawn(

Random.onUnitSphere \* Random.Range(minSpeed, maxSpeed)));

}

public void RemoveFoo()

{

var foo = \_foos[0];

\_fooPool.Despawn(foo);

\_foos.Remove(foo);

}

}

### Фабрики, пулы и схема утилизации

Подход, описанный выше, работает довольно хорошо, но имеет следующие недостатки:

* Каждый раз, когда мы делаем класс пулируемым, нам всегда нужно добавить шаблонный код, в котором мы должны сделать подкласс, а затем вызвать метод экземпляра 'Reset' на нашем объекте, передавая ему любые параметры. Было бы проще, если бы это было как-то автоматизировано, а не дублировалось для каждого объединенного объекта.MemoryPool
* Любой код, порождающий объекты пула, должен поддерживать ссылку на класс пула, чтобы он мог вызывать метод Despawn. Этот код на самом деле не заботится о том, объединен ли объект в пул или нет. Тот факт, что объект объединен в пул, является скорее деталью реализации, и поэтому было бы лучше, если бы он был абстрагирован от кода, который его использует.
* Каждый раз, когда мы хотим преобразовать некоторые объекты, не входящие в пул, для использования пула, нам приходится изменять много кода. Мы должны удалить производный класс, а затем изменить весь код using для вызова вместо , а затем также не забыть вызвать PlaceholderFactorySpawnCreateDespawn

Мы можем решить эти проблемы, используя и шаблон Dispose. Любой код может вызвать метод factory Create так же, как и для объектов, не входящих в пул, а затем вызвать Dispose, чтобы автоматически вернуть объект в пул.PlaceholderFactory

Например:

public class Foo : IPoolable<IMemoryPool>, IDisposable

{

IMemoryPool \_pool;

public void Dispose()

{

\_pool.Despawn(this);

}

public void OnDespawned()

{

\_pool = null;

}

public void OnSpawned(IMemoryPool pool)

{

\_pool = pool;

}

public class Factory : PlaceholderFactory<Foo>

{

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.BindFactory<Foo, Foo.Factory>().FromPoolableMemoryPool(x => x.WithInitialSize(2));

}

}

Для этого мы используем вложенный производный класс PlaceholderFactory, как и для объектов, не входящих в пул, а затем реализуем интерфейс. Для этого потребуется определить и методы, которые будут обрабатывать логику, которую мы использовали в предыдущих примерах.IPoolable<IMemoryPool>OnSpawnedOnDespawnedReset

Затем мы также можем реализовать, а затем вернуться в данный пул всякий раз, когда это вызывается.IDisposableDispose

Затем при привязке фабрики мы можем использовать метод для настройки пула с начальным начальным значением, максимальным размером и методом расширения, а также методом, который используется для создания объекта.FromPoolableMemoryPool

Примечание: Если вы столкнулись с проблемами IL2CPP AOT при описанном выше подходе, вам, возможно, придется сделать класс пула памяти более явным, выполнив вместо этого следующее:

public class TestInstaller : MonoInstaller<TestInstaller>

{

public override void InstallBindings()

{

Container.BindFactory<Foo, Foo.Factory>().FromPoolableMemoryPool<Foo, FooPool>(x => x.WithInitialSize(2));

}

public class FooPool : PoolableMemoryPool<IMemoryPool, Foo>

{

}

}

Это происходит из-за того, что иногда IL2CPP не включает автоматическиPoolableMemoryPool

### Пулы памяти для GameObjects

Пулы памяти для GameObjects работают аналогичным образом. Например:

public class Foo : MonoBehaviour

{

Vector3 \_velocity;

[Inject]

public void Construct()

{

Reset(Vector3.zero);

}

public void Update()

{

transform.position += \_velocity \* Time.deltaTime;

}

void Reset(Vector3 velocity)

{

transform.position = Vector3.zero;

\_velocity = velocity;

}

public class Pool : MonoMemoryPool<Vector3, Foo>

{

protected override void Reinitialize(Vector3 velocity, Foo foo)

{

foo.Reset(velocity);

}

}

}

public class Bar

{

readonly Foo.Pool \_fooPool;

readonly List<Foo> \_foos = new List<Foo>();

public Bar(Foo.Pool fooPool)

{

\_fooPool = fooPool;

}

public void AddFoo()

{

float maxSpeed = 10.0f;

float minSpeed = 1.0f;

\_foos.Add(\_fooPool.Spawn(

Random.onUnitSphere \* Random.Range(minSpeed, maxSpeed)));

}

public void RemoveFoo()

{

var foo = \_foos[0];

\_fooPool.Despawn(foo);

\_foos.Remove(foo);

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public GameObject FooPrefab;

public override void InstallBindings()

{

Container.Bind<Bar>().AsSingle();

Container.BindMemoryPool<Foo, Foo.Pool>()

.WithInitialSize(2)

.FromComponentInNewPrefab(FooPrefab)

.UnderTransformGroup("Foos");

}

}

Основное отличие здесь заключается в том, что теперь происходит от вместо . — вспомогательный класс, который будет автоматически включать и отключать игровой объект для нас при его добавлении/удалении из пула. Реализация для этого проста:Foo.PoolMonoMemoryPoolMemoryPoolMonoMemoryPoolMonoMemoryPool

public abstract class MonoMemoryPool<TParam1, TValue> : MemoryPool<TParam1, TValue>

where TValue : Component

{

Transform \_originalParent;

protected override void OnCreated(TValue item)

{

item.gameObject.SetActive(false);

// Record the original parent which will be set to whatever is used in the UnderTransform method

\_originalParent = item.transform.parent;

}

protected override void OnDestroyed(TValue item)

{

GameObject.Destroy(item.gameObject);

}

protected override void OnSpawned(TValue item)

{

item.gameObject.SetActive(true);

}

protected override void OnDespawned(TValue item)

{

item.gameObject.SetActive(false);

if (item.transform.parent != \_originalParent)

{

item.transform.SetParent(\_originalParent, false);

}

}

}

Поэтому, если вы переопределите один из этих методов, вам придется обязательно вызвать базовую версию.

Также стоит отметить тот факт, что для того, чтобы эта логика работала, наше MonoBehaviour должно лежать в основе префаба, поскольку в противном случае будет отключено только преобразование, связанное с любыми дочерними элементами.Foo

Вы также можете использовать шаблон здесь, используя аналогичный подход, описанный выше для не-MonoBehaviours, который будет выглядеть следующим образом:Dispose

public class Foo : MonoBehaviour, IPoolable<Vector3, IMemoryPool>, IDisposable

{

Vector3 \_velocity;

IMemoryPool \_pool;

public void Dispose()

{

\_pool.Despawn(this);

}

public void Update()

{

transform.position += \_velocity \* Time.deltaTime;

}

public void OnDespawned()

{

\_pool = null;

\_velocity = Vector3.zero;

}

public void OnSpawned(Vector3 velocity, IMemoryPool pool)

{

transform.position = Vector3.zero;

\_pool = pool;

\_velocity = velocity;

}

public class Factory : PlaceholderFactory<Vector3, Foo>

{

}

}

public class Bar

{

readonly Foo.Factory \_fooFactory;

readonly List<Foo> \_foos = new List<Foo>();

public Bar(Foo.Factory fooFactory)

{

\_fooFactory = fooFactory;

}

public void AddFoo()

{

float maxSpeed = 10.0f;

float minSpeed = 1.0f;

var foo = \_fooFactory.Create(

Random.onUnitSphere \* Random.Range(minSpeed, maxSpeed));

foo.transform.SetParent(null);

\_foos.Add(foo);

}

public void RemoveFoo()

{

if (\_foos.Any())

{

var foo = \_foos[0];

foo.Dispose();

\_foos.Remove(foo);

}

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public GameObject FooPrefab;

public override void InstallBindings()

{

Container.Bind<Bar>().AsSingle();

Container.BindFactory<Vector3, Foo, Foo.Factory>().FromMonoPoolableMemoryPool(

x => x.WithInitialSize(2).FromComponentInNewPrefab(FooPrefab).UnderTransformGroup("FooPool"));

}

}

Обратите внимание, что, в отличие от других примеров, мы получаем от , реализуем и используем вместо . Также будьте осторожны с потенциальными [ошибками АОТ](https://github.com/modesttree/Zenject/blob/master/Documentation/MemoryPools.md#aot-error) при таком подходе.PlaceholderFactoryIDisposableFromMonoPoolableMemoryPoolFromPoolableMemoryPool

## Продвинутый

### Пулы статической памяти

Другой подход к пулам памяти заключается в том, чтобы вообще не утруждать себя установкой пула памяти, а вместо этого хранить пул статически с помощью класса. Например:StaticMemoryPool

public class Foo

{

public static readonly StaticMemoryPool<Foo> Pool =

new StaticMemoryPool<Foo>(OnSpawned, OnDespawned);

static void OnSpawned(Foo that)

{

// Initialize

}

static void OnDespawned(Foo that)

{

// Reset

}

}

public class PoolExample : MonoBehaviour

{

public void Update()

{

var foo = Foo.Pool.Spawn();

// Use foo

Foo.Pool.Despawn(foo);

}

}

В этом случае доступ к пулу осуществляется непосредственно как к статическому члену класса. Этот подход может быть полезен для объектов, которые не имеют зависимостей, или для случаев, когда вы не хотите беспокоиться о том, чтобы всегда устанавливать его везде. Однако следует помнить, что, в отличие от обычных пулов памяти, объекты в пуле останутся в памяти даже после смены сцен, если только пул не будет очищен вручную путем вызова .FooFoo.Pool.Clear

Опять же, вы также можете использовать шаблон для этого подхода. Например:Dispose

public class Foo : IDisposable

{

public static readonly StaticMemoryPool<Foo> Pool =

new StaticMemoryPool<Foo>(OnSpawned, OnDespawned);

public void Dispose()

{

Pool.Despawn(this);

}

static void OnSpawned(Foo that)

{

// Initialize

}

static void OnDespawned(Foo that)

{

// Reset

}

}

public class PoolExample : MonoBehaviour

{

public void Update()

{

var foo = Foo.Pool.Spawn();

// Use foo

foo.Dispose();

}

}

Вы также можете включить параметры времени выполнения в объект в пуле, используя универсальные аргументы, аналогичные обычным пулам памяти.StaticMemoryPool

public class Foo : IDisposable

{

public static readonly StaticMemoryPool<string, Foo> Pool =

new StaticMemoryPool<string, Foo>(OnSpawned, OnDespawned);

public string Value

{

get; private set;

}

public void Dispose()

{

Pool.Despawn(this);

}

static void OnSpawned(string value, Foo that)

{

that.Value = value;

}

static void OnDespawned(Foo that)

{

that.Value = null;

}

}

Кроме того, как и в случае с обычными пулами памяти, вы можете использовать интерфейс в сочетании с тем, чтобы избежать некоторого шаблонного кода и использовать методы экземпляра вместо статических методов:IPoolablePoolableStaticMemoryPool

public class Foo : IPoolable<string>, IDisposable

{

public static readonly PoolableStaticMemoryPool<string, Foo> Pool =

new PoolableStaticMemoryPool<string, Foo>();

public string Data

{

get; private set;

}

public void Dispose()

{

Pool.Despawn(this);

}

public void OnSpawned(string data)

{

Data = data;

}

public void OnDespawned()

{

Data = null;

}

}

### Использование инструкций и шаблона «Распоряжение»

У следующего подхода есть несколько недостатков:

public class PoolExample : MonoBehaviour

{

public void Update()

{

var foo = Foo.Pool.Spawn();

// Use foo

foo.Dispose();

}

}

1. Мы всегда должны помнить о вызове в конце метода. Мы могли бы легко забыть об этом и привести к тому, что на кадр будет происходить много выделений.Dispose()
2. Если мы хотим иметь несколько операторов возврата в функции, мы должны дублировать код очистки для каждого случая, что может быть еще более подвержено ошибкам
3. Если между и , возникает исключение, объект не будет возвращен в пул.SpawnDispose

Простой способ решить эти проблемы — добавить блок try-final:

public class PoolExample : MonoBehaviour

{

public void Update()

{

var foo = Foo.Pool.Spawn();

try

{

// Use foo

}

finally

{

foo.Dispose();

}

}

}

Или, что аналогично, мы могли бы добавить оператор using:

public class PoolExample : MonoBehaviour

{

public void Update()

{

using (var foo = Foo.Pool.Spawn())

{

// Use foo

}

}

}

Эти подходы гарантируют, что объект Foo будет возвращен в пул независимо от того, возникнет ли исключение или метод завершится досрочно. Это еще одна причина, по которой полезно использовать шаблон для объектов в пуле памяти.Dispose

### Пул списков

Статические пулы памяти особенно полезны для распространенных структур данных, таких как списки или словари. Zenject включает в себя несколько стандартных пулов памяти именно для этой цели, которые вы можете использовать. Например, предположим, что вы пишете объект, который должен перебирать каждый компонент игрового объекта в каждом кадре. Вы можете реализовать это следующим образом:MonoBehaviour

public class PoolExample : MonoBehaviour

{

public void Update()

{

var components = this.GetComponents(typeof(Component));

foreach (var component in components)

{

// Some logic

}

}

}

Однако, если вы запустите сцену с добавленным к ней MonoBehaviour и откроете профилировщик единства, вы увидите, что на кадр выделено около 48 байт. Мы можем избавиться от этого, используя класс и сделав вместо этого следующее:Zenject.ListPool

public class PoolExample : MonoBehaviour

{

public void Update()

{

var components = ListPool<Component>.Instance.Spawn();

this.GetComponents(typeof(Component), components);

foreach (var component in components)

{

// Some logic

}

ListPool<Component>.Instance.Despawn(components);

}

}

Zenject также включает в себя , и классы, которые можно использовать аналогичным образом.DictionaryPoolHashSetPoolArrayPool

### Блок утилизации

Zenject также предоставляет класс, который представляет собой просто набор объектов, которые удаляются сразу при вызове. Он также может быть полезен в сочетании с инструкцией using для случаев, когда вы выделяете несколько временных экземпляров из одного пула или нескольких пулов. Например, предположим, что нам нужно создать несколько временных объектов в нашем классе. Мы могли бы сделать это так:DisposeBlockIDisposableDisposeBlock.DisposePoolExample

public class PoolExample : MonoBehaviour

{

public void Update()

{

using (var foo = Foo.Pool.Spawn())

using (var bar = Bar.Pool.Spawn())

{

// Some logic

}

}

}

Это будет работать, но не масштабируется, если мы создаем более нескольких объектов. Поэтому лучшей альтернативой может быть использование класса DisposeBlock следующим образом:

public class PoolExample : MonoBehaviour

{

public void Update()

{

using (var block = DisposeBlock.Spawn())

{

var foo = Foo.Pool.Spawn();

var bar = Bar.Pool.Spawn();

block.Add(foo);

block.Add(bar);

// Some logic

}

}

}

Мы можем еще больше упростить это, используя метод:DisposeBlock.Spawn

public class PoolExample : MonoBehaviour

{

public void Update()

{

using (var block = DisposeBlock.Spawn())

{

var foo = block.Spawn(Foo.Pool);

var bar = block.Spawn(Bar.Pool);

// Some logic

}

}

}

Мы также можем использовать DisposeBlock, чтобы улучшить наш пример выше и избежать необходимости явного вызова Despawn:ListPool

public class PoolExample : MonoBehaviour

{

public void Update()

{

using (var block = DisposeBlock.Spawn())

{

var components = block.Spawn(ListPool<Component>.Instance);

this.GetComponents(typeof(Component), components);

foreach (var component in components)

{

// Some logic

}

}

}

}

Поскольку списки появления являются такой распространенной операцией, DisposeBlock включает для нее вспомогательный метод, поэтому вместо этого его можно упростить до следующего:

public class PoolExample : MonoBehaviour

{

public void Update()

{

using (var block = DisposeBlock.Spawn())

{

var components = block.SpawnList<Component>();

this.GetComponents(typeof(Component), components);

foreach (var component in components)

{

// Some logic

}

}

}

}

### PoolableMemoryPool

Если вы предпочитаете не следовать схеме удаления, описанной выше, но также хотите избежать шаблонного кода из исходного подхода с использованием метода, вы также можете сделать это с помощью или .ResetPoolableMemoryPoolMonoPoolableMemoryPool

Например:

public class Foo : IPoolable<string>

{

public string Data

{

get; private set;

}

public void OnDespawned()

{

Data = null;

}

public void OnSpawned(string data)

{

Data = data;

}

public class Pool : PoolableMemoryPool<string, Foo>

{

}

}

Реализация очень проста и просто вызывает методы экземпляра в классе:PoolableMemoryPoolIPoolable

public class PoolableMemoryPool<TParam1, TValue> : MemoryPool<TParam1, TValue>

where TValue : IPoolable<TParam1>

{

protected override void OnDespawned(TValue item)

{

item.OnDespawned();

}

protected override void Reinitialize(TParam1 p1, TValue item)

{

item.OnSpawned(p1);

}

}

Если вы предпочитаете, вы также можете сделать методы и закрытыми, используя функцию c# «явная реализация интерфейса», которая позволит вызывать методы и только через интерфейс:OnSpawnedOnDespawnedOnSpawnedOnDespawnedIPoolable

public class Foo : IPoolable<string>

{

public string Data

{

get; private set;

}

void IPoolable<string>.OnDespawned()

{

Data = null;

}

void IPoolable<string>.OnSpawned(string data)

{

Data = data;

}

public class Pool : PoolableMemoryPool<string, Foo>

{

}

}

### Абстрактные пулы памяти

Как и в [случае с абстрактными фабриками](https://github.com/modesttree/Zenject/blob/master/Documentation/Factories.md#abstract-factories), иногда может потребоваться создать пул памяти, возвращающий интерфейс, с конкретным типом, определяемым в установщике. Это очень похоже на абстрактные фабрики. Например:

public interface IFoo

{

}

public class Foo1 : IFoo

{

}

public class Foo2 : IFoo

{

}

public class FooPool : MemoryPool<IFoo>

{

}

public class Bar

{

readonly FooPool \_fooPool;

readonly List<IFoo> \_foos = new List<IFoo>();

public Bar(FooPool fooPool)

{

\_fooPool = fooPool;

}

public void AddFoo()

{

\_foos.Add(\_fooPool.Spawn());

}

public void RemoveFoo()

{

var foo = \_foos[0];

\_fooPool.Despawn(foo);

\_foos.Remove(foo);

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public bool Use1;

public override void InstallBindings()

{

Container.Bind<Bar>().AsSingle();

if (Use1)

{

Container.BindMemoryPool<IFoo, FooPool>().WithInitialSize(10).To<Foo1>();

}

else

{

Container.BindMemoryPool<IFoo, FooPool>().WithInitialSize(10).To<Foo2>();

}

}

}

Возможно, мы также захотим добавить метод в интерфейс и вызвать его Reset()IFooReinitialize()

### Подконтейнеры/фасады и пулы памяти

Вы можете задаться вопросом, если вы используете динамические подконтейнеры и фасады, как лучше всего поместить весь подконтейнер в пул? Рассмотрим следующий пример:

public class EnemyFacade : MonoBehaviour, IPoolable<IMemoryPool>, IDisposable

{

IMemoryPool \_pool;

public void OnSpawned(IMemoryPool pool)

{

\_pool = pool;

}

public void OnDespawned()

{

\_pool = null;

}

public void Dispose()

{

\_pool.Despawn(this);

}

public class Factory : PlaceholderFactory<EnemyFacade>

{

}

}

public class EnemyMoveHandler : MonoBehaviour

{

Settings \_settings;

float \_velocity;

[Inject]

public void Construct(Settings settings)

{

\_settings = settings;

}

public void Update()

{

if (Input.GetKey(KeyCode.UpArrow))

{

\_velocity += Time.deltaTime \* \_settings.Acceleration;

}

transform.position += Vector3.forward \* \_velocity \* Time.deltaTime;

}

[Serializable]

public class Settings

{

public float Acceleration;

}

}

public class Runner : ITickable

{

readonly EnemyFacade.Factory \_enemyFactory;

EnemyFacade \_enemy;

public Runner(EnemyFacade.Factory enemyFactory)

{

\_enemyFactory = enemyFactory;

}

public void Tick()

{

if (Input.GetKeyDown(KeyCode.Space) && \_enemy == null)

{

\_enemy = \_enemyFactory.Create();

}

if (Input.GetKeyDown(KeyCode.Escape) && \_enemy != null)

{

\_enemy.Dispose();

\_enemy = null;

}

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public EnemyMoveHandler.Settings EnemyMoveHandlerSettings;

public GameObject EnemyPrefab;

public override void InstallBindings()

{

Container.BindInstance(EnemyMoveHandlerSettings);

Container.BindInterfacesTo<Runner>().AsSingle();

Container.BindFactory<EnemyFacade, EnemyFacade.Factory>()

.FromMonoPoolableMemoryPool<EnemyFacade>(b => b

.WithInitialSize(1)

.FromSubContainerResolve()

.ByNewPrefabMethod(EnemyPrefab, InstallEnemy)

.UnderTransformGroup("Enemies"));

}

void InstallEnemy(DiContainer subContainer)

{

subContainer.Bind<EnemyFacade>().FromNewComponentOnRoot().AsSingle();

subContainer.Bind<EnemyMoveHandler>().FromNewComponentOnRoot().AsSingle().NonLazy();

}

}

Если мы добавим этот код в проект, то у нас должна быть сцена, в которой пользователь может нажать пробел, чтобы создать экземпляр Enemy, и нажать escape, чтобы удалить его. Затем, после появления, пользователь может нажать стрелку вверх, чтобы применить некоторую скорость к вражескому объекту.

Если вы воссоздаете этот пример, обратите внимание, что не имеет значения, что вы используете для настройки для представления нашего врага. Я просто добавил куб, выбрав его в меню, и создал префаб прямо из него.EnemyPrefabTestInstallerGameObject -> 3D Object -> Cube

Это работает, но, как вы увидите, если вы запустите его, с этим есть проблема. После появления нового врага, добавления к нему скорости, деспауна, а затем появления снова, скорость остается неизменной. Когда мы создаем что-то из пула, мы всегда хотим, чтобы состояние было установлено на значения по умолчанию. В этом примере это довольно просто, и в нашем подконтейнере всего два класса, но в реальном приложении может быть много классов, каждый со своим собственным состоянием, которое должно быть сброшено событием despawn.

Мы могли бы исправить это, заставив методы reset явно вызывать другие методы сброса на других объектах, но было бы неплохо, если бы существовал более общий способ сделать это. Таким образом, вместо этого вы можете добавить встроенный класс в подконтейнер, а затем реализовать интерфейс для любых классов, которые необходимо сбросить:EnemyFacadePoolableManagerIPoolable

public class EnemyMoveHandler : MonoBehaviour, IPoolable

{

Settings \_settings;

float \_velocity;

[Inject]

public void Construct(Settings settings)

{

\_settings = settings;

}

public void OnDespawned()

{

}

public void OnSpawned()

{

\_velocity = 0;

transform.position = Vector3.zero;

}

public void Update()

{

if (Input.GetKey(KeyCode.UpArrow))

{

\_velocity += Time.deltaTime \* \_settings.Acceleration;

}

if (Input.GetKey(KeyCode.DownArrow))

{

\_velocity -= Time.deltaTime \* \_settings.Acceleration;

}

transform.position += Vector3.forward \* \_velocity \* Time.deltaTime;

}

[Serializable]

public class Settings

{

public float Acceleration;

}

}

public class EnemyFacade : MonoBehaviour, IPoolable<IMemoryPool>, IDisposable

{

[Inject]

PoolableManager \_poolableManager;

IMemoryPool \_pool;

public void OnSpawned(IMemoryPool pool)

{

\_pool = pool;

\_poolableManager.TriggerOnSpawned();

}

public void OnDespawned()

{

\_pool = null;

\_poolableManager.TriggerOnDespawned();

}

public void Dispose()

{

\_pool.Despawn(this);

}

public class Factory : PlaceholderFactory<EnemyFacade>

{

}

}

public class TestInstaller : MonoInstaller<TestInstaller>

{

public EnemyMoveHandler.Settings EnemyMoveHandlerSettings;

public GameObject EnemyPrefab;

public override void InstallBindings()

{

Container.BindInstance(EnemyMoveHandlerSettings);

Container.BindInterfacesTo<Runner>().AsSingle();

Container.BindFactory<EnemyFacade, EnemyFacade.Factory>()

.FromMonoPoolableMemoryPool<EnemyFacade>(b => b

.WithInitialSize(2)

.FromSubContainerResolve()

.ByNewPrefabMethod(EnemyPrefab, InstallEnemy)

.UnderTransformGroup("Enemies"));

}

static void InstallEnemy(DiContainer subContainer)

{

subContainer.Bind<EnemyFacade>().FromNewComponentOnRoot().AsSingle();

subContainer.BindInterfacesAndSelfTo<EnemyMoveHandler>().FromNewComponentOnRoot().AsSingle().NonLazy();

subContainer.Bind<PoolableManager>().AsSingle();

}

}

Note the following:

* We've installed in our subcontainerPoolableManager
* In EnemyFacade, we have to call the and methods on PoolableManager to trigger the and methods for the rest of the subcontainer classesTriggerOnSpawnedTriggerOnDespawnedOnSpawnedOnDespawned
* You can also pass generic arguments to to allow passing runtime parameters to the so they are forwarded on to your derived classesPoolableManager<>TriggerOnSpawnedIPoolable<>

Note also that the order that the classes are triggered will use the same execution order that is set with the method, just like the other standard interfaces. Also note that the methods are called in the reverse order compared to .IPoolableBindExecutionOrderOnDespawnedOnSpawned

### Непосредственное создание экземпляров пулов памяти

Для сложных сценариев, связанных с пользовательскими фабриками, может быть желательно напрямую создавать экземпляры пулов памяти. В этом случае вам просто нужно убедиться, что вы предоставили производный класс, который будет использоваться для создания новых экземпляров, и всю информацию о настройках, которая обычно предоставляется через операторы bind. Например:IFactory<>

public class BarFactory : IFactory<Bar>

{

public Bar Create()

{

...

[Custom creation logic]

...

}

}

var settings = new MemoryPoolSettings(

// Initial size

1,

// Max size

int.MaxValue,

PoolExpandMethods.Double);

var pool = \_container.Instantiate<MemoryPool<Bar>>(

new object[] { settings, new MyBarFactory<Bar>() });

### Проверка очистки бассейна

Одна из ошибок, которая иногда может возникнуть при использовании пулов памяти, заключается в том, что созданные объекты не возвращаются в пул должным образом. Это может произойти, если метод вызывается, но тогда программист забывает добавить совпадающий /.SpawnDespawnDispose

Это не всегда проблема, а иногда даже может быть преднамеренным. Предполагая, что созданный объект не является MonoBehaviour, как только объект больше не используется, он все равно будет автоматически уничтожен сборщиком мусора C#, поэтому забывание в этом случае не приведет к утечке памяти. Единственным недостатком является то, что когда объект не возвращается в пул, это означает, что пул должен создать больше объектов в следующий раз, что может быть немного менее эффективным.Despawn

Кроме того, конечно, если у вашего объекта есть пользовательская логика удаления, которую важно запустить, то она также будет пропущена, что может быть проблемой.

Чтобы обнаружить эти ошибки, Zenject включает дополнительный класс, который вы можете установить, который будет создавать исключения при наличии активно порожденных объектов при закрытии сцены. Просто добавьте следующее в один из установщиков сцены:

Container.BindInterfacesTo<PoolCleanupChecker>().AsSingle()

И тогда каждый раз, когда вы забудете позвонить, вы будете получать сообщения об ошибках в консоли после завершения режима игры в Unity. Все, что делает этот класс, - это проверяет, что в каждом пуле нет активных элементов во время события.DespawnLateDispose

Кроме того, поскольку в некоторых случаях это делается намеренно, можно подавить отчеты об ошибках для определенных пулов, передав список типов в качестве аргумента в PoolCleanupChecker:

Container.BindInterfacesTo<PoolCleanupChecker>().AsSingle()

Container.BindInstance(

new List<Type>()

{

typeof(Foo.Pool),

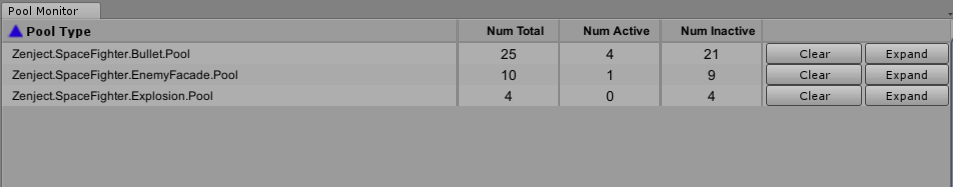
typeof(Bar.Pool),

})

.WhenInjectedInto<PoolCleanupChecker>();

### Монитор пула памяти

Zenject также включает в себя экспериментальное окно редактора, которое можно использовать для мониторинга размеров всех пулов памяти в сцене. Вы можете открыть его, щелкнув внутри Unity, и он должен выглядеть следующим образом:Window -> Zenject Pool Monitor

[](https://github.com/modesttree/Zenject/blob/master/Documentation/Images/PoolMonitor.png?raw=true)

Это может быть полезно, если вы хотите убедиться, что все бассейны, которые вы используете, имеют разумные размеры. Это также может быть полезно как способ выбрать хороший размер для использования с методом, чтобы гарантировать, что все распределения происходят сразу во время экрана загрузки, а не постепенно во время игры.WithInitializeSize

Кроме того, вы можете использовать кнопки «Развернуть» и «Очистить», а также окно профилировщика Unity, чтобы увидеть, насколько пул влияет на общее использование памяти.

# Сигналы

## Содержание

Подробности

* Знакомство
  + [Теория](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#theory)
  + [Быстрый старт сигналов](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#quick-start)
  + [Декларация сигналов](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#declaration)
  + [Синтаксис привязки деклараций](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#declaration-syntax)
  + [Сигнальная стрельба](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#firing)
  + [Связывание сигналов с помощью BindSignal](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#bindsignal)
  + [SignalBusInstaller](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#signalbusinstaller)
  + [Когда использовать сигналы](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#when-to-use-signals)
* Продвинутый
  + [Абстрактные сигналы](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#abstract-signals)
  + [Сигналы с подконтейнерами](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#use-with-subcontainers)
  + [Асинхронные сигналы](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#async-signals)
  + [Настройки сигнала](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#settings)
  + [Идентификаторы](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#identifiers)

## Знакомство

### Мотивация / Теория

Учитывая два класса A и B, которые должны общаться, ваши варианты обычно следующие:

1. Прямой вызов метода на B из A. В этом случае А тесно связан с В.
2. Обратим зависимость, заставив B наблюдать событие на A. В этом случае B тесно связан с A

В качестве третьего варианта, в некоторых случаях может быть лучше, чтобы ни один из них не знал о другом. Таким образом, ваш код будет максимально слабо связан. Этого можно достичь, заставив A и B взаимодействовать с промежуточным объектом (в данном случае с сигналами zenject), а не напрямую друг с другом.

Также обратите внимание, что, хотя результат будет более слабо связан, это не всегда будет лучше. Сигналы могут быть использованы не по назначению, как и любой шаблон программирования, поэтому вы должны рассматривать каждый случай на предмет того, является ли он хорошим кандидатом для них или нет.

### Быстрый старт сигналов

Если вы просто хотите немедленно приступить к работе, см. следующий пример, в котором показано базовое использование:

public class UserJoinedSignal

{

public string Username;

}

public class GameInitializer : IInitializable

{

readonly SignalBus \_signalBus;

public GameInitializer(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void Initialize()

{

\_signalBus.Fire(new UserJoinedSignal() { Username = "Bob" });

}

}

public class Greeter

{

public void SayHello(UserJoinedSignal userJoinedInfo)

{

Debug.Log("Hello " + userJoinedInfo.Username + "!");

}

}

public class GameInstaller : MonoInstaller<GameInstaller>

{

public override void InstallBindings()

{

SignalBusInstaller.Install(Container);

Container.DeclareSignal<UserJoinedSignal>();

Container.Bind<Greeter>().AsSingle();

Container.BindSignal<UserJoinedSignal>()

.ToMethod<Greeter>(x => x.SayHello).FromResolve();

Container.BindInterfacesTo<GameInitializer>().AsSingle();

}

}

Чтобы запустить, просто скопируйте и вставьте приведенный выше код в новый файл с именем, затем создайте пустую сцену с новым контекстом сцены и прикрепите новый установщик.GameInstaller

Существует несколько способов создания обработчиков сигналов. Другой подход может заключаться в следующем

public class Greeter : IInitializable, IDisposable

{

readonly SignalBus \_signalBus;

public Greeter(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void Initialize()

{

\_signalBus.Subscribe<UserJoinedSignal>(OnUserJoined);

}

public void Dispose()

{

\_signalBus.Unsubscribe<UserJoinedSignal>(OnUserJoined);

}

void OnUserJoined(UserJoinedSignal args)

{

SayHello(args.Username);

}

public void SayHello(string userName)

{

Debug.Log("Hello " + userName + "!");

}

}

public class GameInstaller : MonoInstaller<GameInstaller>

{

public override void InstallBindings()

{

SignalBusInstaller.Install(Container);

Container.DeclareSignal<UserJoinedSignal>();

// Here, we can get away with just binding the interfaces since they don't refer

// to each other

Container.BindInterfacesTo<Greeter>().AsSingle();

Container.BindInterfacesTo<GameInitializer>().AsSingle();

}

}

В качестве последнего альтернативного подхода вы также можете объединить сигналы zenject с библиотекой UniRx и сделать это следующим образом:

public class Greeter : IInitializable, IDisposable

{

readonly SignalBus \_signalBus;

readonly CompositeDisposable \_disposables = new CompositeDisposable();

public Greeter(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void Initialize()

{

\_signalBus.GetStream<UserJoinedSignal>()

.Subscribe(x => SayHello(x.Username)).AddTo(\_disposables);

}

public void Dispose()

{

\_disposables.Dispose();

}

public void SayHello(string userName)

{

Debug.Log("Hello " + userName + "!");

}

}

Обратите внимание, что если вы пойдете по этому пути, вам необходимо включить интеграцию UniRx, как описано [здесь](https://github.com/modesttree/Zenject/blob/master/README.md#unirx-integration).

Как видно из приведенных выше примеров, вы можете либо напрямую привязать метод обработчика к сигналу в установщике с помощью (первый пример), либо подключить обработчик сигнала и отсоединиться от него (второй и третий примеры)BindSignal

Подробные сведения о том, как это работает, объясняются в следующих разделах.

### Декларация сигналов

Перед объявлением сигнала необходимо создать класс, который будет его представлять. Например:

public class PlayerDiedSignal

{

}

Любые параметры, передаваемые вместе с сигналом, должны быть добавлены в качестве общедоступных членов или свойств. Например:

public class WeaponEquippedSignal

{

public Player Player;

public IWeapon Weapon;

}

Вы также можете подумать о том, чтобы сделать классы сигналов неизменяемыми, поэтому наш WeaponEquippedSignal может быть лучше написан следующим образом:

public class WeaponEquippedSignal

{

public WeaponEquippedSignal(Player player, IWeapon weapon)

{

Player = player;

Weapon = weapon;

}

public IWeapon Weapon

{

get; private set;

}

public Player Player

{

get; private set;

}

}

В этом нет необходимости, но вы можете сделать это, чтобы убедиться, что какие-либо обработчики сигналов не пытаются изменить значения параметров сигнала, что может негативно повлиять на поведение других обработчиков сигналов.

После того, как мы создали наш класс сигнала, нам просто нужно объявить его где-нибудь в установщике:

public override void InstallBindings()

{

Container.DeclareSignal<PlayerDiedSignal>();

}

Любые объекты, которые находятся в контейнере, где он объявлен, или в любом подконтейнере, теперь могут прослушивать сигнал, а также запускать его.

### Синтаксис привязки деклараций

Оператор DeclareSignal имеет следующий формат:

Container.DeclareSignal<**SignalType**>()

.WithId(**Identifier**)

.**(RequireSubscriber|OptionalSubscriber|OptionalSubscriberWithWarning)**()

.**(RunAsync|RunSync)**()

.WithTickPriority(**TickPriority**)

.(**Copy**|**Move**)Into(**All**|**Direct**)SubContainers();

Где:

* **SignalType** — пользовательский класс, представляющий сигнал
* **Идентификатор** = значение, используемое для уникальной идентификации привязки. В большинстве случаев это можно игнорировать, но может быть полезно в тех случаях, когда вы хотите определить несколько различных сигналов, используя один и тот же тип сигнала.
* RequireSubscriber/**OptionalSubscriber**/**OptionalSubscriberWithWarning** — эти значения определяют, как должен вести себя сигнал, когда он срабатывает, но с ним не связаны подписчики. Если он не переопределен в [ZenjectSettings](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#settings), по умолчанию используется OptionalSubscriber, который в этом случае ничего не сделает. Если задано значение RequireSubscriber, исключения будут создаваться в случае нулевых подписчиков. OptionalSubscriberWithWarning находится на полпути между ними, где он будет выдавать предупреждение журнала консоли вместо исключения. Какой из них вы выберете, зависит от того, насколько строгим вы предпочитаете свое приложение, и имеет ли значение, действительно ли обрабатывается данный сигнал или нет.
* **RunAsync**/**RunSync** — эти значения определяют, будет ли сигнал запускаться синхронно или асинхронно:

**RunSync** — это означает, что при вызове сигнала немедленно вызываются все подписанные методы обработчика.SignalBus.Fire

**RunAsync** — это означает, что при подаче сигнала подписанные методы не будут вызываться до тех пор, пока не будут вызваны позже (как указано в параметре TickPriority).

Обратите внимание, что если он не переопределен в [ZenjectSettings](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#settings), значение по умолчанию выполняется синхронно. [Здесь](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#async-signals) вы найдете обсуждение асинхронных сигналов и то, почему иногда вы можете захотеть использовать их вместо них.

* **TickPriority** = тиковый приоритет для выполнения методов обработчика сигналов. Обратите внимание, что это применимо только при использовании **RunAsync**.
* (**Копировать**|**Переехать**) В(**Все**|**Direct**)SubContainers = Такое же поведение, как описано в [основном разделе, посвященном привязке](https://github.com/modesttree/Zenject/blob/master/README.md#binding).

Обратите внимание, что значение по умолчанию для **RunSync**/**RunAsync** и **RequireSubscriber/OptionalSubscriber** можно переопределить, изменив [ZenjectSettings](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#settings)

### Сигнальная стрельба

Чтобы вызвать сигнал, добавьте ссылку на класс, а затем вызовите метод следующим образом:SignalBusFire

public class UserJoinedSignal

{

}

public class UserManager

{

readonly SignalBus \_signalBus;

public UserManager(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void DoSomething()

{

\_signalBus.Fire<UserJoinedSignal>();

}

}

Или, если у сигнала есть параметры, вы захотите создать его новый экземпляр, например:

public class UserJoinedSignal

{

public string Username;

}

public class UserManager

{

readonly SignalBus \_signalBus;

public UserManager(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void DoSomething()

{

\_signalBus.Fire(new UserJoinedSignal() { Username = "Bob" });

}

}

При вызове SignalBus ожидает, что сигнал будет объявлен, и если сигнал не объявлен, будет выдано исключение. Если вы хотите вызывать независимо от объявления сигнала, используйте метод, который игнорирует необъявленные сигналы. Вы можете использовать внешний вид следующим образом:Fire()Fire()TryFire()TryFire()

public class UserJoinedSignal

{

}

public class UserManager

{

readonly SignalBus \_signalBus;

public UserManager(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void DoSomething()

{

// Generic version

\_signalBus.TryFire<UserJoinedSignal>(); // Nothing happens if UserJoinedSignal is NOT declared

// Non-Generic version

\_signalBus.TryFire(new UserJoinedSignal()); // Nothing happens if UserJoinedSignal is NOT declared

}

}

### Связывание сигналов с помощью BindSignal

Как упоминалось выше, в дополнение к возможности прямого подключения к сигналам на сигнальной шине (через или ) вы также можете напрямую привязать сигнал к классу обработки внутри установщика. Этот подход имеет свои преимущества и недостатки по сравнению с прямой подпиской в классе обработки, поэтому опять же сводится к личным предпочтениям.SignalBus.SubscribeSignalBus.GetStream

Команда BindSignal имеет следующий формат:

Container.BindSignal<**SignalType**>()

.WithId(**Identifier**)

.ToMethod(**Handler**)

.From(**ConstructionMethod**)

.(**Copy**|**Move**)Into(**All**|**Direct**)SubContainers();

Где:

* **SignalType** — пользовательский класс, представляющий сигнал
* **Идентификатор** = значение, используемое для уникальной идентификации привязки. В большинстве случаев это можно игнорировать. Обратите внимание, что при использовании идентификаторов сигналов вы должны использовать тот же идентификатор для DeclareSignal (а также Fire, Subscribe и т. Д.).
* **ConstructionMethod** — при привязке к описанному выше методу экземпляра также необходимо определить, откуда берется этот экземпляр. Более подробную информацию см. в разделе «Обработчик» ниже
* (**Копировать**|**Переехать**) В(**Все**|**Direct**)SubContainers = Такое же поведение, как описано в [основном разделе, посвященном привязке](https://github.com/modesttree/Zenject/blob/master/README.md#binding).
* **Обработчик** — метод, который должен срабатывать при срабатывании сигнала. Это имеет несколько вариаций:

**1. Статический метод**

Container.BindSignal<UserJoinedSignal>().ToMethod(s => Debug.Log("Hello user " + s.Username));

Обратите внимание, что метод также может быть без параметров:

Container.BindSignal<UserJoinedSignal>().ToMethod(() => Debug.Log("Received UserJoinedSignal signal"))

Также обратите внимание, что в этом случае нет возможности указать значение для, так как экземпляр не требуетсяFrom

**2. Метод экземпляра напрямую**

Например:

public class Greeter

{

public void SayHello(UserJoinedSignal signal)

{

Debug.Log("Hello " + signal.Username + "!");

}

}

Container.Bind<Greeter>().AsSingle();

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).FromResolve();

В этом случае мы хотим, чтобы сигнал запускал метод. Обратите внимание, что в этом случае нам нужно указать значение для, потому что для вызова данного метода необходим экземпляр.Greeter.SayHelloFrom

Как и в случае со статическими методами, вы также можете выполнить привязку к методу без параметров:

public class Greeter

{

public void SayHello()

{

Debug.Log("Hello there!");

}

}

Container.Bind<Greeter>().AsSingle();

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).FromResolve();

Однако мы можем использовать любой метод строительства, который захотим. Под капотом фактически расширяется до следующего:FromResolveFromResolve

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).From(x => x.FromResolve().AsCached());

Существует также еще один ярлык для случаев, когда классы обработчика больше нигде в контейнере не доступныFromNew

// These are both equivalent

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).FromNew();

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).From(x => x.AsCached());

Итак, если бы нам не нужно было внедрять класс Greeter где-либо еще, мы могли бы также реализовать его следующим образом:

public class Greeter

{

public void SayHello(UserJoinedSignal signal)

{

Debug.Log("Hello " + signal.Username + "!");

}

}

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).FromNew();

Таким образом, нам вообще не нужна отдельная привязка для Greeter. Вы также можете предоставить множество других видов аргументов, включая привязку к лениво созданному экземпляру, фабричному методу, пользовательской фабрике, фасаду в подконтейнере и т. д.FromMonoBehaviour

**3. Метод экземпляра с отображением**

Также могут быть случаи, когда аргументы метода обработки непосредственно содержат аргументы сигнала. Например:

public class Greeter

{

public void SayHello(string username)

{

Debug.Log("Hello " + username + "!");

}

}

В этом случае вы можете привязать сигнал к методу, который выполняет отображение параметров для нас:

Container.Bind<Greeter>().AsSingle();

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>((x, s) => x.SayHello(s.Username)).FromResolve()

### SignalBusInstaller

Сигналы являются дополнительной функцией Zenject. При импорте Zenject, если вы не хотите включать сигналы, вы можете просто снять галочку с папки. В результате этого сигналы не включаются автоматически, поэтому вам приходится явно устанавливать их самостоятельно, вызвав одного из ваших установщиков.OptionalExtras/SignalsSignalBusInstaller.Install(Container)

Вы можете сделать это только один раз в установщике, или вы можете сделать это в каждой сцене в установщике. Обратите внимание, что вам нужно сделать это только один раз, а затем вы можете использовать сигналы в контейнере, в который вы передаете, а также любые подконтейнеры, поэтому, если вы устанавливаете в , вам не нужно устанавливать в ProjectContextSceneContextSignalBusInstaller,ProjectContextSceneContext.

### Когда использовать сигналы

Сигналы наиболее подходят в качестве механизма связи, когда:

1. Сигнал может прослушивать несколько заинтересованных приемников
2. Отправителю не нужно получать результат от получателя
3. Отправителю даже все равно, будет ли он получен. Другими словами, отправитель не должен полагаться на изменение какого-либо состояния при вызове сигнала для правильной работы последующей логики отправителя. В идеале сигналы можно рассматривать как события «выстрелил и забыл»
4. Отправитель запускает сигнал нечасто или в непредсказуемое время

Это всего лишь эмпирические правила, но их полезно иметь в виду при использовании сигналов. Чем менее логически связан отправитель с ответным поведением получателей, тем более подходящим он является по сравнению с другими формами связи, такими как прямые вызовы методов, интерфейсы, члены классов событий C# и т. д. Это также одна из причин, по которой вы можете рассмотреть возможность использования [асинхронных сигналов](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#async-signals)

Когда программой, управляемой событиями, злоупотребляют, можно оказаться в «аду обратного вызова», где события вызывают другие события и т. Д. и которые делают всю систему невозможной для понимания. Так что сигналы вообще следует использовать с осторожностью. Лично мне нравится использовать сигналы для высокоуровневых игровых событий, а затем использовать другие формы связи (потоки unirx, события c#, прямые вызовы методов, интерфейсы) для большинства других вещей.

## Продвинутый

### Абстрактные сигналы

Одна из проблем сигналов заключается в том, что, когда вы подписываетесь на их типы, вы связываете свои конкретные типы сигналов с подписчиками

Например, допустим, у меня есть игрок, и я хочу сохранить игру, когда закончу уровень. Хорошо, легко, я создаю, а затем подписываю его на свой, затем я также хочу сохранить, когда достигну контрольной точки, снова я создаю, а затем подписываюсь на него, вы начинаете получать что-то вроде этого...SignalLevelCompletedSaveGameSystemSignalCheckpointReachedSaveGameSystem

public class Example

{

SignalBus signalBus;

public Example(Signalbus signalBus) => this.signalBus = signalBus;

public void CheckpointReached() => signalBus.Fire<SignalCheckpointReached>();

public void CompleteLevel() => signalBus.Fire<SignalLevelCompleted>();

}

public class SaveGameSystem

{

public SaveGameSystem(SignalBus signalBus)

{

signalBus.Subscribe<SignalCheckpointReached>(x => SaveGame());

signalBus.Subscribe<SignalLevelCompleted>(x => SaveGame());

}

void SaveGame() { /\*Saves the game\*/ }

}

//in your installer

Container.DeclareSignal<SignalLevelCompleted>();

Container.DeclareSignal<SignalCheckpointReached>();

//your signal types

public struct SignalCheckpointReached{}

public struct SignalLevelCompleted{}

And then you realize you are coupling the types and to . shouldn't know about those "non related with saving" events...signalLevelCompletedSignalCheckpointReachedSaveGameSystemSaveGameSystem

So let's give the power of interfaces to signals! So i have the and both implementing **ISignalGameSaver** and my just Subscribes to **ISignalGameSaver** for saving the game So when i fire any of those signals the saves the game. Then you have something like this...SignalCheckpointReachedSignalLevelCompletedSaveGameSystemSaveGameSystem

public class Example

{

SignalBus signalBus;

public Example(Signalbus signalBus) => this.signalBus = signalBus;

public void CheckpointReached() => signalBus.AbstractFire<SignalCheckpointReached>();

public void CompleteLevel() => signalBus.AbstractFire<SignalLevelCompleted>();

}

public class SaveGameSystem

{

public SaveGameSystem(SignalBus signalBus)

{

signalBus.Subscribe<ISignalGameSaver>(x => SaveGame());

}

void SaveGame() { /\*Saves the game\*/ }

}

//in your installer

Container.DeclareSignalWithInterfaces<SignalLevelCompleted>();

Container.DeclareSignalWithInterfaces<SignalCheckpointReached>();

//your signal types

public struct SignalCheckpointReached : ISignalGameSaver{}

public struct SignalLevelCompleted : ISignalGameSaver{}

public interface ISignalGameSaver{}

Now your doesnt knows about CheckPoints nor Level events, and just reacts to signals that save the game. The main difference is in the Signal declaration and FiringSaveGameSystem

* DeclareSignalWithInterfaces works like but it declares the interfaces too.DeclareSignal
* AbstractFire is the same that but it fires the interfacesjust if you have Declared the signal with interfaces otherwise it will throw an exception.Fire

Ok, let's show even more power. Now i create another signal for the WorldDestroyed Achievement "SignalWorldDestroyed" But i also want my SoundSystem to play sounds when i reach a checkpoint and/or unlock an Achievement So the code could look like this.

public class Example

{

SignalBus signalBus;

public Example(Signalbus signalBus) => this.signalBus = signalBus;

public void CheckpointReached() => signalBus.AbstractFire<SignalCheckpointReached>();

public void DestroyWorld() => signalBus.AbstractFire<SignalWorldDestroyed>();

}

public class SoundSystem

{

public SoundSystem(SignalBus signalBus)

{

signalBus.Subscribe<ISignalSoundPlayer>(x => PlaySound(x.soundId));

}

void PlaySound(int soundId) { /\*Plays the sound with the given id\*/ }

}

public class AchievementSystem

{

public AchievementSystem(SignalBus signalBus)

{

signalBus.Subscribe<ISignalAchievementUnlocker>(x => UnlockAchievement(x.achievementKey));

}

void UnlockAchievement(string key) { /\*Unlocks the achievement with the given key\*/ }

}

//in your installer

Container.DeclareSignalWithInterfaces<SignalCheckpointReached>();

Container.DeclareSignalWithInterfaces<SignalWorldDestroyed>();

//your signal types

public struct SignalCheckpointReached : ISignalGameSaver, ISignalSoundPlayer

{

public int SoundId { get => 2} //or configured in a scriptable with constants instead of hardcoded

}

public struct SignalWorldDestroyed : ISignalAchievementUnlocker, ISignalSoundPlayer

{

public int SoundId { get => 4}

public string AchievementKey { get => "WORLD\_DESTROYED"}

}

//Your signal interfaces

public interface ISignalGameSaver{}

public interface ISignalSoundPlayer{ int SoundId {get;}}

public interface ISignalAchievementUnlocker{ string AchievementKey {get;}}

Он предлагает большую модульность и абстракцию для сигналов, Вы посылаете конкретный сигнал, рассказывающий о том, что вы сделали, и даете им функциональность через реализацию интерфейса

### Сигналы с подконтейнерами

Сигналы видны только на уровне контейнера, где они объявлены, и ниже. Например, вы можете использовать поддержку нескольких сцен в Unity и разделить игру на сцену с графическим интерфейсом и сцену окружения. В сцене с графическим интерфейсом вы можете подать сигнал, указывающий на то, что наложение всплывающего окна графического интерфейса было открыто/закрыто, чтобы сцена окружения могла приостановить/возобновить активность. Одним из способов достижения этой цели может быть объявление сигнала в установщике ProjectContext (или [родительском элементе](https://github.com/modesttree/Zenject/blob/master/README.md#scene-parenting) общей сцены), затем подписка на него в сцене среды, а затем запуск из сцены графического интерфейса.

### Асинхронные сигналы

В некоторых случаях может быть желательно запускать данный сигнал асинхронно. Асинхронные сигналы имеют следующие преимущества:

1. Порядок обновления, в котором запускаются обработчики сигналов, может быть более предсказуемым. При использовании синхронных сигналов методы обработчика сигналов выполняются одновременно с подачей сигнала, который может быть запущен в любое время в течение кадра или, в некоторых случаях, в нескольких местах, если сигнал срабатывает несколько раз. Это может привести к некоторым проблемам с порядком обновления. При использовании асинхронных сигналов обработчики сигналов всегда выполняются одновременно в кадре, настроенном TickPriority.
2. Асинхронные сигналы могут способствовать меньшему взаимодействию между отправителем и получателем, что часто является тем, что вам нужно. Как объяснялось [выше](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#when-to-use-signals), сигналы работают лучше всего, когда они используются для событий «выстрелил и забыл», когда отправитель не заботится о поведении каких-либо слушателей. Делая сигнал асинхронным, он может обеспечить это разделение, потому что методы обработчика сигналов будут выполняться позже, и поэтому отправитель фактически не может напрямую использовать результат поведения обработчиков.
3. Неожиданные изменения состояния могут произойти при подаче только одного сигнала. Например, объект А может вызвать сигнал, который вызовет некоторую логику, которая в конечном итоге приведет к удалению А. Если сигнал был выполнен синхронно, то стек вызовов может в конечном итоге вернуться к объекту А, где был выпущен сигнал, и объект А может попытаться выполнить команды впоследствии, что вызовет проблемы (поскольку объект А уже будет удален)

Это не означает, что асинхронные сигналы превосходят синхронные сигналы. Асинхронные сигналы также имеют свои риски.

1. Отладка может быть более сложной, так как из трассировки стека неясно, где был запущен сигнал.
2. Некоторые части штата могут быть не синхронизированы друг с другом. Если класс A инициирует асинхронный сигнал, требующий ответа от класса B, то между моментом срабатывания сигнала и вызовом метода обработчика в классе B будет некоторый период, когда B не синхронизирован с A, что может привести к некоторым ошибкам.
3. В целом система может быть более сложной, чем при использовании синхронных сигналов, и, следовательно, труднее для понимания.

### Настройки сигнала

Большинство настроек по умолчанию для сигналов можно переопределить с помощью свойства settings, находящегося в . Его также можно настроить на уровне каждого контейнера, задав свойство. Для сигналов это включает в себя следующее:ProjectContextDiContainer.Settings

**Default Sync Mode (Режим синхронизации по** умолчанию) — это значение управляет значением по умолчанию для свойства /, если оно не указано. По умолчанию он установлен на синхронный, поэтому будет предполагать, что если он не указан вызовом . Поэтому, если вы являетесь поклонником асинхронных сигналов, вы можете установить для этого параметра асинхронный, чтобы вместо этого предполагать асинхронность.DeclareSignalRunSyncRunAsyncRunSyncDeclareSignal

**Missing Handler Default Response (Отсутствующий ответ обработчика** по умолчанию) — это значение управляет значением по умолчанию, если **для** вызова . По умолчанию для него задано значение **OptionalSubscriber**.DeclareSignal

**Требовать строгую отмену подписки** - Если значение true, это приведет к возникновению исключений, если сцена заканчивается, а обработчики сигналов все еще существуют, которые еще не отписались. По умолчанию это false.

**Асинхронный приоритет** тиков по умолчанию — это значение управляет приоритетом тиков по умолчанию, если он используется, но не установлен. По умолчанию для него установлено значение 1, что приведет к тому, что обработчики сигналов будут вызываться сразу после того, как будут вызваны все обычные тикируемые объекты. Это значение по умолчанию выбрано потому, что оно гарантирует, что сигнал обрабатывается в том же кадре, в котором он запускается, что может быть важно, если сигнал влияет на то, как кадр визуализируется.RunAsyncDeclareSignalWithTickPriority

# Сигналы

## Содержание

Подробности

* Знакомство
  + [Теория](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#theory)
  + [Быстрый старт сигналов](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#quick-start)
  + [Декларация сигналов](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#declaration)
  + [Синтаксис привязки деклараций](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#declaration-syntax)
  + [Сигнальная стрельба](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#firing)
  + [Связывание сигналов с помощью BindSignal](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#bindsignal)
  + [SignalBusInstaller](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#signalbusinstaller)
  + [Когда использовать сигналы](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#when-to-use-signals)
* Продвинутый
  + [Абстрактные сигналы](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#abstract-signals)
  + [Сигналы с подконтейнерами](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#use-with-subcontainers)
  + [Асинхронные сигналы](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#async-signals)
  + [Настройки сигнала](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#settings)
  + [Идентификаторы](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#identifiers)

## Знакомство

### Мотивация / Теория

Учитывая два класса A и B, которые должны общаться, ваши варианты обычно следующие:

1. Прямой вызов метода на B из A. В этом случае А тесно связан с В.
2. Обратим зависимость, заставив B наблюдать событие на A. В этом случае B тесно связан с A

В качестве третьего варианта, в некоторых случаях может быть лучше, чтобы ни один из них не знал о другом. Таким образом, ваш код будет максимально слабо связан. Этого можно достичь, заставив A и B взаимодействовать с промежуточным объектом (в данном случае с сигналами zenject), а не напрямую друг с другом.

Также обратите внимание, что, хотя результат будет более слабо связан, это не всегда будет лучше. Сигналы могут быть использованы не по назначению, как и любой шаблон программирования, поэтому вы должны рассматривать каждый случай на предмет того, является ли он хорошим кандидатом для них или нет.

### Быстрый старт сигналов

Если вы просто хотите немедленно приступить к работе, см. следующий пример, в котором показано базовое использование:

public class UserJoinedSignal

{

public string Username;

}

public class GameInitializer : IInitializable

{

readonly SignalBus \_signalBus;

public GameInitializer(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void Initialize()

{

\_signalBus.Fire(new UserJoinedSignal() { Username = "Bob" });

}

}

public class Greeter

{

public void SayHello(UserJoinedSignal userJoinedInfo)

{

Debug.Log("Hello " + userJoinedInfo.Username + "!");

}

}

public class GameInstaller : MonoInstaller<GameInstaller>

{

public override void InstallBindings()

{

SignalBusInstaller.Install(Container);

Container.DeclareSignal<UserJoinedSignal>();

Container.Bind<Greeter>().AsSingle();

Container.BindSignal<UserJoinedSignal>()

.ToMethod<Greeter>(x => x.SayHello).FromResolve();

Container.BindInterfacesTo<GameInitializer>().AsSingle();

}

}

Чтобы запустить, просто скопируйте и вставьте приведенный выше код в новый файл с именем, затем создайте пустую сцену с новым контекстом сцены и прикрепите новый установщик.GameInstaller

Существует несколько способов создания обработчиков сигналов. Другой подход может заключаться в следующем

public class Greeter : IInitializable, IDisposable

{

readonly SignalBus \_signalBus;

public Greeter(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void Initialize()

{

\_signalBus.Subscribe<UserJoinedSignal>(OnUserJoined);

}

public void Dispose()

{

\_signalBus.Unsubscribe<UserJoinedSignal>(OnUserJoined);

}

void OnUserJoined(UserJoinedSignal args)

{

SayHello(args.Username);

}

public void SayHello(string userName)

{

Debug.Log("Hello " + userName + "!");

}

}

public class GameInstaller : MonoInstaller<GameInstaller>

{

public override void InstallBindings()

{

SignalBusInstaller.Install(Container);

Container.DeclareSignal<UserJoinedSignal>();

// Here, we can get away with just binding the interfaces since they don't refer

// to each other

Container.BindInterfacesTo<Greeter>().AsSingle();

Container.BindInterfacesTo<GameInitializer>().AsSingle();

}

}

В качестве последнего альтернативного подхода вы также можете объединить сигналы zenject с библиотекой UniRx и сделать это следующим образом:

public class Greeter : IInitializable, IDisposable

{

readonly SignalBus \_signalBus;

readonly CompositeDisposable \_disposables = new CompositeDisposable();

public Greeter(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void Initialize()

{

\_signalBus.GetStream<UserJoinedSignal>()

.Subscribe(x => SayHello(x.Username)).AddTo(\_disposables);

}

public void Dispose()

{

\_disposables.Dispose();

}

public void SayHello(string userName)

{

Debug.Log("Hello " + userName + "!");

}

}

Обратите внимание, что если вы пойдете по этому пути, вам необходимо включить интеграцию UniRx, как описано [здесь](https://github.com/modesttree/Zenject/blob/master/README.md#unirx-integration).

Как видно из приведенных выше примеров, вы можете либо напрямую привязать метод обработчика к сигналу в установщике с помощью (первый пример), либо подключить обработчик сигнала и отсоединиться от него (второй и третий примеры)BindSignal

Подробные сведения о том, как это работает, объясняются в следующих разделах.

### Декларация сигналов

Перед объявлением сигнала необходимо создать класс, который будет его представлять. Например:

public class PlayerDiedSignal

{

}

Любые параметры, передаваемые вместе с сигналом, должны быть добавлены в качестве общедоступных членов или свойств. Например:

public class WeaponEquippedSignal

{

public Player Player;

public IWeapon Weapon;

}

Вы также можете подумать о том, чтобы сделать классы сигналов неизменяемыми, поэтому наш WeaponEquippedSignal может быть лучше написан следующим образом:

public class WeaponEquippedSignal

{

public WeaponEquippedSignal(Player player, IWeapon weapon)

{

Player = player;

Weapon = weapon;

}

public IWeapon Weapon

{

get; private set;

}

public Player Player

{

get; private set;

}

}

В этом нет необходимости, но вы можете сделать это, чтобы убедиться, что какие-либо обработчики сигналов не пытаются изменить значения параметров сигнала, что может негативно повлиять на поведение других обработчиков сигналов.

После того, как мы создали наш класс сигнала, нам просто нужно объявить его где-нибудь в установщике:

public override void InstallBindings()

{

Container.DeclareSignal<PlayerDiedSignal>();

}

Любые объекты, которые находятся в контейнере, где он объявлен, или в любом подконтейнере, теперь могут прослушивать сигнал, а также запускать его.

### Синтаксис привязки деклараций

Оператор DeclareSignal имеет следующий формат:

Container.DeclareSignal<**SignalType**>()

.WithId(**Identifier**)

.**(RequireSubscriber|OptionalSubscriber|OptionalSubscriberWithWarning)**()

.**(RunAsync|RunSync)**()

.WithTickPriority(**TickPriority**)

.(**Copy**|**Move**)Into(**All**|**Direct**)SubContainers();

Где:

* **SignalType** — пользовательский класс, представляющий сигнал
* **Идентификатор** = значение, используемое для уникальной идентификации привязки. В большинстве случаев это можно игнорировать, но может быть полезно в тех случаях, когда вы хотите определить несколько различных сигналов, используя один и тот же тип сигнала.
* RequireSubscriber/**OptionalSubscriber**/**OptionalSubscriberWithWarning** — эти значения определяют, как должен вести себя сигнал, когда он срабатывает, но с ним не связаны подписчики. Если он не переопределен в [ZenjectSettings](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#settings), по умолчанию используется OptionalSubscriber, который в этом случае ничего не сделает. Если задано значение RequireSubscriber, исключения будут создаваться в случае нулевых подписчиков. OptionalSubscriberWithWarning находится на полпути между ними, где он будет выдавать предупреждение журнала консоли вместо исключения. Какой из них вы выберете, зависит от того, насколько строгим вы предпочитаете свое приложение, и имеет ли значение, действительно ли обрабатывается данный сигнал или нет.
* **RunAsync**/**RunSync** — эти значения определяют, будет ли сигнал запускаться синхронно или асинхронно:

**RunSync** — это означает, что при вызове сигнала немедленно вызываются все подписанные методы обработчика.SignalBus.Fire

**RunAsync** — это означает, что при подаче сигнала подписанные методы не будут вызываться до тех пор, пока не будут вызваны позже (как указано в параметре TickPriority).

Обратите внимание, что если он не переопределен в [ZenjectSettings](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#settings), значение по умолчанию выполняется синхронно. [Здесь](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#async-signals) вы найдете обсуждение асинхронных сигналов и то, почему иногда вы можете захотеть использовать их вместо них.

* **TickPriority** = тиковый приоритет для выполнения методов обработчика сигналов. Обратите внимание, что это применимо только при использовании **RunAsync**.
* (**Копировать**|**Переехать**) В(**Все**|**Direct**)SubContainers = Такое же поведение, как описано в [основном разделе, посвященном привязке](https://github.com/modesttree/Zenject/blob/master/README.md#binding).

Обратите внимание, что значение по умолчанию для **RunSync**/**RunAsync** и **RequireSubscriber/OptionalSubscriber** можно переопределить, изменив [ZenjectSettings](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#settings)

### Сигнальная стрельба

Чтобы вызвать сигнал, добавьте ссылку на класс, а затем вызовите метод следующим образом:SignalBusFire

public class UserJoinedSignal

{

}

public class UserManager

{

readonly SignalBus \_signalBus;

public UserManager(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void DoSomething()

{

\_signalBus.Fire<UserJoinedSignal>();

}

}

Или, если у сигнала есть параметры, вы захотите создать его новый экземпляр, например:

public class UserJoinedSignal

{

public string Username;

}

public class UserManager

{

readonly SignalBus \_signalBus;

public UserManager(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void DoSomething()

{

\_signalBus.Fire(new UserJoinedSignal() { Username = "Bob" });

}

}

При вызове SignalBus ожидает, что сигнал будет объявлен, и если сигнал не объявлен, будет выдано исключение. Если вы хотите вызывать независимо от объявления сигнала, используйте метод, который игнорирует необъявленные сигналы. Вы можете использовать внешний вид следующим образом:Fire()Fire()TryFire()TryFire()

public class UserJoinedSignal

{

}

public class UserManager

{

readonly SignalBus \_signalBus;

public UserManager(SignalBus signalBus)

{

\_signalBus = signalBus;

}

public void DoSomething()

{

// Generic version

\_signalBus.TryFire<UserJoinedSignal>(); // Nothing happens if UserJoinedSignal is NOT declared

// Non-Generic version

\_signalBus.TryFire(new UserJoinedSignal()); // Nothing happens if UserJoinedSignal is NOT declared

}

}

### Связывание сигналов с помощью BindSignal

Как упоминалось выше, в дополнение к возможности прямого подключения к сигналам на сигнальной шине (через или ) вы также можете напрямую привязать сигнал к классу обработки внутри установщика. Этот подход имеет свои преимущества и недостатки по сравнению с прямой подпиской в классе обработки, поэтому опять же сводится к личным предпочтениям.SignalBus.SubscribeSignalBus.GetStream

Команда BindSignal имеет следующий формат:

Container.BindSignal<**SignalType**>()

.WithId(**Identifier**)

.ToMethod(**Handler**)

.From(**ConstructionMethod**)

.(**Copy**|**Move**)Into(**All**|**Direct**)SubContainers();

Где:

* **SignalType** — пользовательский класс, представляющий сигнал
* **Идентификатор** = значение, используемое для уникальной идентификации привязки. В большинстве случаев это можно игнорировать. Обратите внимание, что при использовании идентификаторов сигналов вы должны использовать тот же идентификатор для DeclareSignal (а также Fire, Subscribe и т. Д.).
* **ConstructionMethod** — при привязке к описанному выше методу экземпляра также необходимо определить, откуда берется этот экземпляр. Более подробную информацию см. в разделе «Обработчик» ниже
* (**Копировать**|**Переехать**) В(**Все**|**Direct**)SubContainers = Такое же поведение, как описано в [основном разделе, посвященном привязке](https://github.com/modesttree/Zenject/blob/master/README.md#binding).
* **Обработчик** — метод, который должен срабатывать при срабатывании сигнала. Это имеет несколько вариаций:

**1. Статический метод**

Container.BindSignal<UserJoinedSignal>().ToMethod(s => Debug.Log("Hello user " + s.Username));

Обратите внимание, что метод также может быть без параметров:

Container.BindSignal<UserJoinedSignal>().ToMethod(() => Debug.Log("Received UserJoinedSignal signal"))

Также обратите внимание, что в этом случае нет возможности указать значение для, так как экземпляр не требуетсяFrom

**2. Метод экземпляра напрямую**

Например:

public class Greeter

{

public void SayHello(UserJoinedSignal signal)

{

Debug.Log("Hello " + signal.Username + "!");

}

}

Container.Bind<Greeter>().AsSingle();

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).FromResolve();

В этом случае мы хотим, чтобы сигнал запускал метод. Обратите внимание, что в этом случае нам нужно указать значение для, потому что для вызова данного метода необходим экземпляр.Greeter.SayHelloFrom

Как и в случае со статическими методами, вы также можете выполнить привязку к методу без параметров:

public class Greeter

{

public void SayHello()

{

Debug.Log("Hello there!");

}

}

Container.Bind<Greeter>().AsSingle();

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).FromResolve();

Однако мы можем использовать любой метод строительства, который захотим. Под капотом фактически расширяется до следующего:FromResolveFromResolve

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).From(x => x.FromResolve().AsCached());

Существует также еще один ярлык для случаев, когда классы обработчика больше нигде в контейнере не доступныFromNew

// These are both equivalent

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).FromNew();

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).From(x => x.AsCached());

Итак, если бы нам не нужно было внедрять класс Greeter где-либо еще, мы могли бы также реализовать его следующим образом:

public class Greeter

{

public void SayHello(UserJoinedSignal signal)

{

Debug.Log("Hello " + signal.Username + "!");

}

}

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>(x => x.SayHello).FromNew();

Таким образом, нам вообще не нужна отдельная привязка для Greeter. Вы также можете предоставить множество других видов аргументов, включая привязку к лениво созданному экземпляру, фабричному методу, пользовательской фабрике, фасаду в подконтейнере и т. д.FromMonoBehaviour

**3. Метод экземпляра с отображением**

Также могут быть случаи, когда аргументы метода обработки непосредственно содержат аргументы сигнала. Например:

public class Greeter

{

public void SayHello(string username)

{

Debug.Log("Hello " + username + "!");

}

}

В этом случае вы можете привязать сигнал к методу, который выполняет отображение параметров для нас:

Container.Bind<Greeter>().AsSingle();

Container.BindSignal<UserJoinedSignal>().ToMethod<Greeter>((x, s) => x.SayHello(s.Username)).FromResolve()

### SignalBusInstaller

Сигналы являются дополнительной функцией Zenject. При импорте Zenject, если вы не хотите включать сигналы, вы можете просто снять галочку с папки. В результате этого сигналы не включаются автоматически, поэтому вам приходится явно устанавливать их самостоятельно, вызвав одного из ваших установщиков.OptionalExtras/SignalsSignalBusInstaller.Install(Container)

Вы можете сделать это только один раз в установщике, или вы можете сделать это в каждой сцене в установщике. Обратите внимание, что вам нужно сделать это только один раз, а затем вы можете использовать сигналы в контейнере, в который вы передаете, а также любые подконтейнеры, поэтому, если вы устанавливаете в , вам не нужно устанавливать в ProjectContextSceneContextSignalBusInstaller,ProjectContextSceneContext.

### Когда использовать сигналы

Сигналы наиболее подходят в качестве механизма связи, когда:

1. Сигнал может прослушивать несколько заинтересованных приемников
2. Отправителю не нужно получать результат от получателя
3. Отправителю даже все равно, будет ли он получен. Другими словами, отправитель не должен полагаться на изменение какого-либо состояния при вызове сигнала для правильной работы последующей логики отправителя. В идеале сигналы можно рассматривать как события «выстрелил и забыл»
4. Отправитель запускает сигнал нечасто или в непредсказуемое время

Это всего лишь эмпирические правила, но их полезно иметь в виду при использовании сигналов. Чем менее логически связан отправитель с ответным поведением получателей, тем более подходящим он является по сравнению с другими формами связи, такими как прямые вызовы методов, интерфейсы, члены классов событий C# и т. д. Это также одна из причин, по которой вы можете рассмотреть возможность использования [асинхронных сигналов](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#async-signals)

Когда программой, управляемой событиями, злоупотребляют, можно оказаться в «аду обратного вызова», где события вызывают другие события и т. Д. и которые делают всю систему невозможной для понимания. Так что сигналы вообще следует использовать с осторожностью. Лично мне нравится использовать сигналы для высокоуровневых игровых событий, а затем использовать другие формы связи (потоки unirx, события c#, прямые вызовы методов, интерфейсы) для большинства других вещей.

## Продвинутый

### Абстрактные сигналы

Одна из проблем сигналов заключается в том, что, когда вы подписываетесь на их типы, вы связываете свои конкретные типы сигналов с подписчиками

Например, допустим, у меня есть игрок, и я хочу сохранить игру, когда закончу уровень. Хорошо, легко, я создаю, а затем подписываю его на свой, затем я также хочу сохранить, когда достигну контрольной точки, снова я создаю, а затем подписываюсь на него, вы начинаете получать что-то вроде этого...SignalLevelCompletedSaveGameSystemSignalCheckpointReachedSaveGameSystem

public class Example

{

SignalBus signalBus;

public Example(Signalbus signalBus) => this.signalBus = signalBus;

public void CheckpointReached() => signalBus.Fire<SignalCheckpointReached>();

public void CompleteLevel() => signalBus.Fire<SignalLevelCompleted>();

}

public class SaveGameSystem

{

public SaveGameSystem(SignalBus signalBus)

{

signalBus.Subscribe<SignalCheckpointReached>(x => SaveGame());

signalBus.Subscribe<SignalLevelCompleted>(x => SaveGame());

}

void SaveGame() { /\*Saves the game\*/ }

}

//in your installer

Container.DeclareSignal<SignalLevelCompleted>();

Container.DeclareSignal<SignalCheckpointReached>();

//your signal types

public struct SignalCheckpointReached{}

public struct SignalLevelCompleted{}

And then you realize you are coupling the types and to . shouldn't know about those "non related with saving" events...signalLevelCompletedSignalCheckpointReachedSaveGameSystemSaveGameSystem

So let's give the power of interfaces to signals! So i have the and both implementing **ISignalGameSaver** and my just Subscribes to **ISignalGameSaver** for saving the game So when i fire any of those signals the saves the game. Then you have something like this...SignalCheckpointReachedSignalLevelCompletedSaveGameSystemSaveGameSystem

public class Example

{

SignalBus signalBus;

public Example(Signalbus signalBus) => this.signalBus = signalBus;

public void CheckpointReached() => signalBus.AbstractFire<SignalCheckpointReached>();

public void CompleteLevel() => signalBus.AbstractFire<SignalLevelCompleted>();

}

public class SaveGameSystem

{

public SaveGameSystem(SignalBus signalBus)

{

signalBus.Subscribe<ISignalGameSaver>(x => SaveGame());

}

void SaveGame() { /\*Saves the game\*/ }

}

//in your installer

Container.DeclareSignalWithInterfaces<SignalLevelCompleted>();

Container.DeclareSignalWithInterfaces<SignalCheckpointReached>();

//your signal types

public struct SignalCheckpointReached : ISignalGameSaver{}

public struct SignalLevelCompleted : ISignalGameSaver{}

public interface ISignalGameSaver{}

Now your doesnt knows about CheckPoints nor Level events, and just reacts to signals that save the game. The main difference is in the Signal declaration and FiringSaveGameSystem

* DeclareSignalWithInterfaces works like but it declares the interfaces too.DeclareSignal
* AbstractFire is the same that but it fires the interfacesjust if you have Declared the signal with interfaces otherwise it will throw an exception.Fire

Ok, let's show even more power. Now i create another signal for the WorldDestroyed Achievement "SignalWorldDestroyed" But i also want my SoundSystem to play sounds when i reach a checkpoint and/or unlock an Achievement So the code could look like this.

public class Example

{

SignalBus signalBus;

public Example(Signalbus signalBus) => this.signalBus = signalBus;

public void CheckpointReached() => signalBus.AbstractFire<SignalCheckpointReached>();

public void DestroyWorld() => signalBus.AbstractFire<SignalWorldDestroyed>();

}

public class SoundSystem

{

public SoundSystem(SignalBus signalBus)

{

signalBus.Subscribe<ISignalSoundPlayer>(x => PlaySound(x.soundId));

}

void PlaySound(int soundId) { /\*Plays the sound with the given id\*/ }

}

public class AchievementSystem

{

public AchievementSystem(SignalBus signalBus)

{

signalBus.Subscribe<ISignalAchievementUnlocker>(x => UnlockAchievement(x.achievementKey));

}

void UnlockAchievement(string key) { /\*Unlocks the achievement with the given key\*/ }

}

//in your installer

Container.DeclareSignalWithInterfaces<SignalCheckpointReached>();

Container.DeclareSignalWithInterfaces<SignalWorldDestroyed>();

//your signal types

public struct SignalCheckpointReached : ISignalGameSaver, ISignalSoundPlayer

{

public int SoundId { get => 2} //or configured in a scriptable with constants instead of hardcoded

}

public struct SignalWorldDestroyed : ISignalAchievementUnlocker, ISignalSoundPlayer

{

public int SoundId { get => 4}

public string AchievementKey { get => "WORLD\_DESTROYED"}

}

//Your signal interfaces

public interface ISignalGameSaver{}

public interface ISignalSoundPlayer{ int SoundId {get;}}

public interface ISignalAchievementUnlocker{ string AchievementKey {get;}}

Он предлагает большую модульность и абстракцию для сигналов, Вы посылаете конкретный сигнал, рассказывающий о том, что вы сделали, и даете им функциональность через реализацию интерфейса

### Сигналы с подконтейнерами

Сигналы видны только на уровне контейнера, где они объявлены, и ниже. Например, вы можете использовать поддержку нескольких сцен в Unity и разделить игру на сцену с графическим интерфейсом и сцену окружения. В сцене с графическим интерфейсом вы можете подать сигнал, указывающий на то, что наложение всплывающего окна графического интерфейса было открыто/закрыто, чтобы сцена окружения могла приостановить/возобновить активность. Одним из способов достижения этой цели может быть объявление сигнала в установщике ProjectContext (или [родительском элементе](https://github.com/modesttree/Zenject/blob/master/README.md#scene-parenting) общей сцены), затем подписка на него в сцене среды, а затем запуск из сцены графического интерфейса.

### Асинхронные сигналы

В некоторых случаях может быть желательно запускать данный сигнал асинхронно. Асинхронные сигналы имеют следующие преимущества:

1. Порядок обновления, в котором запускаются обработчики сигналов, может быть более предсказуемым. При использовании синхронных сигналов методы обработчика сигналов выполняются одновременно с подачей сигнала, который может быть запущен в любое время в течение кадра или, в некоторых случаях, в нескольких местах, если сигнал срабатывает несколько раз. Это может привести к некоторым проблемам с порядком обновления. При использовании асинхронных сигналов обработчики сигналов всегда выполняются одновременно в кадре, настроенном TickPriority.
2. Асинхронные сигналы могут способствовать меньшему взаимодействию между отправителем и получателем, что часто является тем, что вам нужно. Как объяснялось [выше](https://github.com/modesttree/Zenject/blob/master/Documentation/Signals.md#when-to-use-signals), сигналы работают лучше всего, когда они используются для событий «выстрелил и забыл», когда отправитель не заботится о поведении каких-либо слушателей. Делая сигнал асинхронным, он может обеспечить это разделение, потому что методы обработчика сигналов будут выполняться позже, и поэтому отправитель фактически не может напрямую использовать результат поведения обработчиков.
3. Неожиданные изменения состояния могут произойти при подаче только одного сигнала. Например, объект А может вызвать сигнал, который вызовет некоторую логику, которая в конечном итоге приведет к удалению А. Если сигнал был выполнен синхронно, то стек вызовов может в конечном итоге вернуться к объекту А, где был выпущен сигнал, и объект А может попытаться выполнить команды впоследствии, что вызовет проблемы (поскольку объект А уже будет удален)

Это не означает, что асинхронные сигналы превосходят синхронные сигналы. Асинхронные сигналы также имеют свои риски.

1. Отладка может быть более сложной, так как из трассировки стека неясно, где был запущен сигнал.
2. Некоторые части штата могут быть не синхронизированы друг с другом. Если класс A инициирует асинхронный сигнал, требующий ответа от класса B, то между моментом срабатывания сигнала и вызовом метода обработчика в классе B будет некоторый период, когда B не синхронизирован с A, что может привести к некоторым ошибкам.
3. В целом система может быть более сложной, чем при использовании синхронных сигналов, и, следовательно, труднее для понимания.

### Настройки сигнала

Большинство настроек по умолчанию для сигналов можно переопределить с помощью свойства settings, находящегося в . Его также можно настроить на уровне каждого контейнера, задав свойство. Для сигналов это включает в себя следующее:ProjectContextDiContainer.Settings

**Default Sync Mode (Режим синхронизации по** умолчанию) — это значение управляет значением по умолчанию для свойства /, если оно не указано. По умолчанию он установлен на синхронный, поэтому будет предполагать, что если он не указан вызовом . Поэтому, если вы являетесь поклонником асинхронных сигналов, вы можете установить для этого параметра асинхронный, чтобы вместо этого предполагать асинхронность.DeclareSignalRunSyncRunAsyncRunSyncDeclareSignal

**Missing Handler Default Response (Отсутствующий ответ обработчика** по умолчанию) — это значение управляет значением по умолчанию, если **для** вызова . По умолчанию для него задано значение **OptionalSubscriber**.DeclareSignal

**Требовать строгую отмену подписки** - Если значение true, это приведет к возникновению исключений, если сцена заканчивается, а обработчики сигналов все еще существуют, которые еще не отписались. По умолчанию это false.

**Асинхронный приоритет** тиков по умолчанию — это значение управляет приоритетом тиков по умолчанию, если он используется, но не установлен. По умолчанию для него установлено значение 1, что приведет к тому, что обработчики сигналов будут вызываться сразу после того, как будут вызваны все обычные тикируемые объекты. Это значение по умолчанию выбрано потому, что оно гарантирует, что сигнал обрабатывается в том же кадре, в котором он запускается, что может быть важно, если сигнал влияет на то, как кадр визуализируется.RunAsyncDeclareSignalWithTickPriority