Прямые конвертации (DirectConvertor)

Прямые конвертации являются альтернативой сериализации. Используются два формата:

- текстовый, как наиболее репрезентативный, конвертирующий поля классов в текст и разделяющий поля отступами (\t) и переносом строки (\n)
- бинарный, как наиболее экономный, конвертирующий поля классов в последовательность байтов.

При этом может использоваться запись сразу в файл или получение дампа для последующей записи или передачи по сети.

Многие подходы для управления сериализацией используют атрибуты классов и рефлексию (reflection), что создает многие неудобства и замедляет процесс. Прямые конвертации вместо этого используют ряд разработанных классов и интерфейсов, которые позволяют гибко настраивать какие поля будут конвертироваться в единый файл/дамп данных.

В данной версии поддерживаются конвертация, часто используемых полей:

- Примитивных типов: string, int, float, bool
- Специфичных типов: Guid, Vector3, Transform
- Иерархическое включение (агрегация) объектов, в том числе компонент юнити (префабы)
- Обобщенные списки List<> как с примитивным типом, так и объектом (в том числе вложенным списком или словарем)
- Словари Dictionary<,> с ключом string или int, и значением примитивного типа, объекта (в том числе вложенные списки или словари)

При необходимости новые типы добавляются сравнительно просто.

Класс DirectConvert

В классе, который управляет сохранениями и знает какие данные нужно сохранять, можно или создать класс DirectConvert (агрегировать) или от него наследоваться. После чего нужно указать какие данные будут конвертироваться:

- для записи надо вызвать метод `void Set(object argObject)`, указывая любой объект для записи.
- для восстановления объекта вызвать метод `T Get<T>(T argObject)`, выбирая в той же последовательности объекты, которые были переданы для записи

Пример:

```
private void SetProtocol()
{
```

```
        dConvert = new DirectConvert();
        dConvert.Set(Tag);
        dConvert.Set(AllowSave);
        dConvert.Set(Number);
        dConvert.Set(MainPosition);
    }

    private void GetProtocol()
    {
        Tag = dConvert.Get(Tag);
        AllowSave = dConvert.Get(AllowSave);
        Number = dConvert.Get(Number);
        MainPosition = dConvert.Get(MainPosition);
    }
```

После этого можно вызвать один из методов сохранения или загрузки:

- Сохранение в файл (в бинарном или текстовом формате)
  `void Save(string argFileName, ConvertorType argConvertorType)`
- Преобразование в последовательность байтов
  `byte[] BinSave()`
- Восстановление данных из строки байтов в формате 00-00-00-
- `byte[] DampLoad(string argDampTxt)`
- Восстановление данных из последовательности данных
- `byte[] DampLoad(byte[] argBuffer)`
- Восстановление данных из файла (в бинарном или текстовом формате)
- `void Load(string argFileName, ConvertorType argConvertorType)`

Интерфейс IDirectConvertData

Если объект является вложенным, в объекты, которые были указаны для конвертирования, то он должен реализовать интерфейс IDirectConvertData. Например, если для:

```
    public List<AngleInfo> Angles = new List<AngleInfo>();
```

указано

```
    dConvert.Set(Angles);
```

то класс AngleInfo должен реализовать интерфейс IDirectConvertData.

Для удобства этот интерфейс уже реализован в классах DirectConvert и MainBehaviour и поэтому можно просто от них наследовать класс и только переопределить методы GetData()/SetData(), если такое наследование нежелательно, то дополнительно надо реализовать список Data, и пользовательский тэг режима обработки. И для помещения в очередь дополнительно реализованы методы ClearQ/GetQ/SetQ, аналогичные уже рассмотренным Get/Set.

```
    public interface IDirectConvertData
    {
        List<object> Data { get; }
        string DataTag { get; set; }
        void GetData();
        void SetData();
    }
```

Например, так:

```
public class AngleInfo : DirectConvert
{
    public int AddInfo = 1;
    public Vector3 Value;

    public override void SetData()
    {
        Value = GetQ(Value);
        AddInfo = GetQ(AddInfo);
    }

    public override void GetData()
    {
        ClearQ();
        SetQ(Value);
        SetQ(AddInfo);
    }
}
```

Класс ObjectTag

В ряде случаев, вы захотите по-разному сохранять тот же объект, в зависимости от того, где он находится (в каком контексте используется). Для этого вы можете при передаче объекта наделить его собственным тэгом. Например,

```
public class Entity : MainBehaviour
{
	public Guid ObjectId;
	public float Count;
	public List<Entity> Contain = new List<Entity>();

	public override void SetData()
	{
		ObjectId = GetQ(ObjectId);
		if (dataTag != "OnlyId")
		{
			Count = GetQ(Count);
			Contain = GetQ(Contain);
		}
	}
	public override void GetData()
	{
		ClearQ();
		SetQ(ObjectId);
		if (dataTag != "OnlyId")
		{
			SetQ(Count);
			SetQ(new ObjectTag(Contain, "OnlyId"));
		}
	}
}
```

В обычном режиме тэг будет пустым, но т.к. для списка List<Entity> Contain, указан тэг = OnlyId, то при записи этого списка тэг будет равен этому значению, и вы сможете управлять логикой того, какие из полей класса подлежат записи, т.е. в данном случае только ObjectId.

Интерфейс IPrefabCreate

В Юнити игровые объекты (GameObject) создаются специфичным образом через фабричный метод Instantiate, а через вызов конструктора их нельзя создать. Поэтому многие подходы к сериализации, использующие рефлекшен, не могут работать с такими классами. Кроме того, это вынуждает разработчиков отделять данные от поведения (например, используется в подходе ECS (EntityComponentSystem), что на самом деле является нарушением принципов ООП). Используя прямые конвертации, нужно просто реализовать интерфейс:

```
public interface IPrefabCreate
{
    EntityManager EManager { get; }
    string PrefabName { get; set; }
}
```

Пример класса EntityManager включен в поставку. Он представляет собой оболочку над созданием префабов в Юнити. Для этого он читает текстовый файл ModelList.txt из директории Resources, в котором указан список путей к префабам. Эти префабы должны содержать некий простой компонент (`Entity`), в котором есть идентификация. При инициализации объект EntityManager загружает образцы всех префабов под определенным именем префаба. Кроме того, он предоставляет два метода для создания (`CreateObject`) и удаления (`DestroyObject`) префаба. Этот класс расширяемый и вы его можете расширять в соответствии с вашими потребностями, важно лишь оставить реализацию метода

```
GameObject CreateObject(string argModelName)
```

При реализации интерфейса `IPrefabCreate`, кроме предоставления доступа к `EntityManager`, еще нужно предоставить свойство с именем префаба. Причем важно, при реализации метода GetData имя префаба записать в поток первым. Это связанно с тем, что для оптимизации данные читаются только в одном направлении, и прежде, чем создать свойства зависимые от юнити, нужно создать префаб, а для этого знать его имя (идентификацию).

Пример:

```
public string ModelName = "";
public string PrefabName
{
	get { return ModelName; }
	set { ModelName = value; }
}

public override void GetData()
{
	ClearQ();
	SetQ(ModelName);
}
```

Интерфейс IVersion

Для поддержки разных версий наборов данных, используется версия класса. Для этого надо реализовать интерфейс

```
public interface IVersion
{
    float LoadVersion { get; set; }
    float SaveVersion { get; }
}
```

LoadVersion позволяет понять какая версия была загружена из файла/дампа, а SaveVersion позволяет управлять тем в какой версии будет записан файл/дамп. Для удобства LoadVersion уже реализован в классах DirectConvert и MainBehaviour. Но чтобы указать, что для класса используется версионность надо реализацию приписать конкретному классу и указать SaveVersion, например:

```
public class Entity : MainBehaviour, IPrefabCreate, IVersion
{
   public float SaveVersion
   {
          get { return 1.02f; }
   }
}
```

И использовать в условиях формируя список полей, учитывая их версию:

```
   public override void GetData()
   {
          ClearQ();
          SetQ(ModelName);
          SetQ(Id);
          SetQ(SpeciesId);
          SetQ(ObjectId);
          if (dataTag != "OnlyId")
          {
                 SetQ(Count);
                 if (LoadVersion == 1.02f)
                 {
                        SetQ((int)LocatedIn);
                        SetQ(Transform);
                 }
                 SetQ(new ObjectTag(Contain, "OnlyId"));
          }
   }
```

Рекурсия вложенных объектов

Следить за рекурсией вложенных объектов не является задачей прямых конвертаций. Но её можно легко избежать, например, для графа храня отдельно уникальные узлы и отдельно переходы между ними. Или же должно быть известно в каком списке хранятся уникальные объекты, а в каких ссылки на эти объекты.

В этом случае, во включенном примере показано, что для таких списков можно использовать тэг сохраняя лишь идентификаторы объектов (пример см. в разделе ObjectTag), а уже после загрузки восстановить ссылки на сами объекты по идентификации:

```
private Dictionary<Guid, Entity> UniqueEntity = new Dictionary<Guid, Entity>();
public void RecoverEntity()
{
	for (int i = 0; i < entities.Count; i++)
	{
		if (UniqueEntity.Keys.Contains(entities[i].ObjectId) == false)
		{
			UniqueEntity.Add(entities[i].ObjectId, entities[i]);
		}
	}
	for (int i = 0; i < entities.Count; i++)
	{
		for (int j = 0; j < entities[i].Contain.Count; j++)
		{
			entities[i].Contain[j] =
			UniqueEntity[entities[i].Contain[j].ObjectId];
		}
	}
}
```