Список использованных источников:
1. Криптографические основы безопасности. [Электронный ресурс]. – Режим доступа: http://www.intuit.ru/department/security/networksec/ - Дата доступа: 27.12.2017

ОСОБЕННОСТИ КЭШИРОВАНИЯ КОМПОНЕНТОВ В UNITY

Институт информационных технологий БГУИР,
г. Минск, Республика Беларусь

Мультан Р.И. Мазур А.Д.

Бакунова О.М., ст. преподаватель каф ИСиТ, м.т.н.
Образцова О.Н., доцент каф. ИСиТ, к.т.н,, доцент

Огромная часть Unity-разработчиков понимают, что не стоит злоупотреблять дорогими для производительности вычислениями, к таким вычислениям относится и получение компонентов. Чтобы избежать дорогих вычислений необходимо использовать кэширование. В докладе рассмотрены различные способы кэширования, их реализация, особенности и производительность.

Данная статья будет рассматривать в основном внутреннее кэширование — это процесс получения различных компонентов на объекте. Объект в Unity, находящийся на сцене, представляет собой GameObject - т.е. вместилище для разнообразных компонентов. Компоненты бывают стандартными (Rigidbody, RectTransform, Animator), так бывают и написанными разработчиками(класс, который наследуется от MonoBehaviour). Для того, чтобы получить компонент на объекте можно использовать метод GetComponent(), но не стоит им сильно злоупотреблять. Если компонент используется в скрипте не одиножды, существует подход при котором возможно объявить его переменную и получить его компонент с объекта всего лишь один раз с помощью метода GetComponent() (рисунок 1)

.

```
public class CachingExample : MonoBehaviour
{
    private Rigidbody rigidbody;

    void Start()
    {
        rigidbody = GetComponent<Rigidbody>();
    }

    void Update()
    {
        rigidbody.AddForce(Vector3.up * Time.deltaTime);
    }
}
```

Рисунок 1 – Объявление компонента, как переменную.

Заметим, что прямое получение компонентов на объекте более производительно, чем остальные способы кэширования, в большинстве случаев. Другие вариации кэширования опираются на базовое и упрощают нам доступ к компонентам, лишая нас необходимости писать один и тот же код, когда нам необходим какой-либо компонент.

Для следующего метода кэширования используются свойства. Свойство - это разновидность члена класса, предоставляющий гибкий механизм для чтения, записи и вычисления значения частного поля. Явным преимуществом этого метода является то, что кэширования произойдет только тогда, когда мы первый раз обратимся к свойству. Явным недостатком является то, что нам приходится писать больше однообразного кода (рисунок 2).

```
public class PropertyCachingExample : MonoBehaviour
{
    private Rigidbody rigidbody;
    private bool isRigidbodyCached = false;

    public Rigidbody CachedRigidbody
    {
        get
        {
            if (!rigidbody)
            {
                isRigidbodyCached = true;
                rigidbody = GetComponent<Rigidbody>();
            }
            return rigidbody;
        }
    }
}
```

Рисунок 2 – Использование для хеширования свойств

Облегчить себе работу можно унаследовав классы от компонента, который кэширует употребляемые свойства. Данный метод не универсален. Для оптимального разрешения проблемы универсальности необходимо использовать шаблоны (рисунок 3).

```
public class InnerCache : MonoBehaviour
{
    Dictionary<Type, Component> cache = new Dictionary<Type, Component>();

    public T Get<T>() where T : Component
    {
        var type = typeof(T);
        Component item;
        if (!cache.TryGetValue(type, out item))
        {
            item = GetComponent<T>();
            cache.Add(type, item);
        }
        return item as T;
    }
}
```

Рисунок 3 – Пример шаблона

В языке программирования C# есть особенность под названием “методы расширения”. Данные методы позволяют нам добавлять свои методы в уже существующие типы без создания нового типа. Данный метод кэширования реализуется следующим способом:

```
public static class ExternalCache
{
    static Dictionary<GameObject, TestComponent> test = new Dictionary<GameObject, TestComponent>();

    public static TestComponent GetCachedTestComponent(this GameObject owner)
    {
        TestComponent item = null;
        if (!test.TryGetValue(owner, out item))
        {
            item = owner.GetComponent<TestComponent>();
            test.Add(owner, item);
        }
        return item;
    }
}
```

Рисунок 4 – Использование особенностей «методы расширения» .языка программирования C# .

После реализации метода расширения компонент можно будет получить из любого скрипта таким образом — gameObject.GetCachedTestComponent(). Таким образом, написав один раз метод расширения, мы избавляем себя от написания единообразного кода и инкапсулируем внутреннюю реализацию получения компонента у gameObject.

Недостатком этого варианта является постоянное слежение за мертвыми ссылками. Если за ними не следить (т.е.не чистить кэш), объем кэша будет расти и загружать память ссылками на уничтоженные объекты. Пока что самым производительным вариантом кэширования остается базовое кэширование — получение компоненты объекта с помощью GetComponent() при инициализации объекта. Зачастую окольные пути не совсем оптимальны и требуют от разработчика написания лишних строчек кода.

Теперь рассмотрим другой способ, в котором используются атрибуты. Атрибут представляет собой специальный инструмент, который позволяет вставить в сборку дополнительные метаданные. Метаданные - это данные в двоичном формате с описанием программы, хранящиеся либо в памяти, либо в переносимом хранящимся файле. Атрибуты сами по себе не выполняются и их необходимо использовать с помощью рефлексии, хоть рефлексия и достаточно дорогая операция. Рефлексия представляет собой процесс вскрытия типов во время выполнения приложения. Также можно использовать самописный атрибут для кэширования, а именно [AttributeUsage(AttributeTargets.Field)]private class CacheAttribute:Attribute (){}. Дальше пользоваться им для всех переменных класса[Cached] private TestedComponents Test. После этого необходимо реализовать класс, который способен принимать переменные классов с их атрибутами, а также подхватывать их при инициализации (рисунок 5).

```
public class AttributeCacheInherit : MonoBehaviour
{

    protected virtual void Awake()
    {
        CacheAll();
    }

    void CacheAll()
    {
        var type = GetType();
        CacheFields(GetFieldsToCache(type));
    }

    List<FieldInfo> GetFieldsToCache(Type type)
    {
        var fields = new List<FieldInfo>();
        foreach (var field in type.GetFields())
        {
            foreach (var a in field.GetCustomAttributes(false))
            {
                if (a is CachedAttribute)
                {
                    fields.Add(field);
                }
            }
        }
        return fields;
    }

    void CacheFields(List<FieldInfo> fields)
    {
        var iter = fields.GetEnumerator();
        while (iter.MoveNext())
        {
            var type = iter.Current.FieldType;
            iter.Current.SetValue(this, GetComponent(type));
        }
    }
```

Рисунок 5 – Пример вставки в сборку дополнительных метаданных

При создании наследуемого класса можно получать его переменные помощью атрибута[Cache], это позволит не думать о кэшировании. При этом можно использовать статический класс, с помощью которого при выполнении программы мы можем достать компоненту только единожды, и в дальнейшем ее использовать. Тогда и использование довольно сильно влияющей на производительность рефлексии нам не понадобится. Чтобы закэшировать типы нам нужно добавить строчку CacheHelper.CacheAll(this) в один из методов инициализации (методы Awake или Start). Теперь все, что было помечено в этом классе атрибутом [Cache], будет получено при вызове метода GetComponent().

Компоненту можно и не кэшировать в уже запущенном приложении, для этого можно воспользоваться редактором, перед запуском проекта. По функциональности такой метод кардинально не выделяется среди остальных, за исключением того, что все это происходит непосредственно перед запуском редактора. Эта функция кэширует компоненты для их дальнейшего использования. Хотя в этом варианте есть свои особенности. В конце статьи хотелось бы пройтись по тому, что было затронуто. Были рассмотрены разнообразные методы кэширования компонент, применения атрибутов. Функции, в которых основой является рефлексия. Такие методы могут быть использованы при создании Unity-приложений, если знать и учитывать его особенности. Один из методов позволяет писать меньше единообразного кода, но его производительность немного проигрывает “прямолинейному” решению. Другой на сегодняшний день требует немного больше внимания, но при этом не затрагивает конечную производительность.

Например, он не нуждается в ресурсах на явную манипуляцию, объекты нужно подготавливать явно, при приготовлении объекты обязаны находится на сцене, в шаблоны и объекты не должны вносится никаких изменений на других сценах.