Чем значимые типы отличаются от ссылочных?

Вы уверены, что ЗНАЕТЕ ответ?

Кирилл Маурин

Задача для интервью

- Есть несколько структур, реализующих один интерфейс
- Есть алгоритм, работающий с этим интерфейсом
- Нужна реализация алгоритма, работающая со всеми структурами
- Без боксинга, копипасты и SMS

Подсказка

- А еще без приведений типов и виртуальных вызовов

Еще подсказка

- А чем тут могут помочь обобщения (генерики)?

Вопрос на засыпку

- Сколько кандидатов дали мне правильный ответ на интервью?

Ответ

- На данный момент ровно 0

Задача попроще

- В чем проблема кода ниже?

```
readonly TEnumerator _enumerator;
```

Подсказка

- А что, если итератор — структура?

Подсказка

- А что, если итератор — структура?

Еще подсказка

- В чем разница между readonly полем ссылочного и значимого типов?

Первое отличие

- Семантика копирования

Второе отличие

- Отсутствие информации о типе периода выполнения

Третье отличие

- Встраивание

Ответ на вторую загадку

- Итератор может быть структурой
- Поле только для чтения ссылочного типа означает неизменность ссылки
- Значимого типа — неизменность содержания
- Итератор по построению изменяемый
- Вызов изменяющего метода для структуры в поле только для чтения будет произведен после копирования

Четвертое отличие

- Размещение на стеке

Пятое отличие

- Нет возможности принудительно инициализировать в конструкторе
- Созданная в обход явного конструктора структура будет заполнена нулями

Шестое отличие

- Возможность превращения в ссылочный тип как вручную, так и автоматически

Главное отличие

- Для типов-параметров код генериков создается на лету отдельно для каждой комбинации конкретных типов
- Даже при наличии ограничений на реализацию интерфейса все равно генерируются прямые невиртуальные вызовы без боксинга и приведения типов

Ответ на вторую загадку

- Алгоритм реализуется в виде генерика с ограничением параметра-типа на интерфейс

Интересные факты

- Можно писать высокоуровневый хорошо читаемый высокопроизводительный код
- Можно писать без аллокаций. Совсем.
- Можно писать без классов. Совсем.
- .NET имеет очень слабый оптимизатор в сравнении с JVM
- .NET имеет примерно равную производительность с JVM

Ключ для объекта

```
public readonly struct Key<T, TKey> :
    IEquatable<Key<T, TKey>>
{
    internal Key(TKey id) => Unwrap = id;

    public TKey Unwrap { get; }

    public override string ToString()
        => $"Id<{nameof(TEntity)}>({Unwrap})";

    . . .
}
```

Идентификатор для сущности

```
public struct Id<TEntity, T> :
    IEquatable<Id<TEntity, T>>
    where TEntity : IEntity<T>
{
    internal Id(T id) => Unwrap = id;

    public T Unwrap { get; }

    public override string ToString()
        => $"Id<{nameof(TEntity)}>({Unwrap})";

    . . .
}
```

Идентификатор для сущности

```
public struct Id<TEntity, T> :
    IEquatable<Id<TEntity, T>>
    where TEntity : IEntity<T>
{
    internal Id(T id) => Unwrap = id;

    public T Unwrap { get; }

    public override string ToString()
        => $"Id<{nameof(TEntity)}>({Unwrap})";

    . . .
}
```

Фабричные методы расширения

```
public static Id<T, TKey> AsId<T, TKey>(this TKey id, T _)
    where T : IEntity<TKey>
    => new Id<T, Tkey>(id);

public static Id<T, TKey> GetId<T, TKey>(this T entity)
    where T : IEntity<TKey>
    => new Id<T, TKey>(entity.Id);
```

Плюсы структур-оберток

- Нулевая добавочная стоимость по памяти
- Говорящее имя типа
- Удобные сигнатуры методов
- Несовместимость с обычными элементарными типами
- Удобное отображение в отладчике
- Удобное создания без явных параметров-типов

Всегда действительная ссылка

```
public readonly struct NotNull<T> :
    IEquatable<NotNull<T>>, IOption<T>
{
    internal NotNull([NotNull]T unwrap)
        => Unwrap = unwrap;

    [NotNull]
    public T Unwrap { get; };

    . . .
}
```

Фабричные методы расширения

```
public static NotNull<T> EnsureNotNull<T>
    ([NotNull]this T value, string name)
    => TryCreate(value, out var result)
    ? result
    : throw new ArgumentNullException(name);

public static NotNull<T> CannotBeNull<T>
    ([NotNull]this T reference)
    where T: class
    => new NotNull<T>(reference);
```

Особенности структуры NotNull

- Это обертка со всеми вытекающими
- Честный тип вместо атрибутной магии
- Достаточно проверить один раз и дальше передавать безбоязненно
- Проверять можно с помощью R# или во время исполнения
- Без инициализации внутри будет null

То, чего может и не быть

- Нельзя получить значение без проверки
- После проверки возвращается гарантированно непустое
- Исключения не бросаются

```
public interface IOption<T>
{
    bool TryGetValue(out NotNull<T> value);
    bool HasValue { get; }
}
```

Наивная реализация

```
public sealed class OptionClass<T> : IOption<T>, IEquatable<OptionClass<T>>
{
    internal OptionClass(in T value) => Value = value;
    OptionClass() { }

    public bool HasValue { get; } = true;
    public bool TryGetValue(out NotNull<T> value)
    {
        value = Value;
        return HasValue;
    }
    NotNull<T> Value { get; }
    . . .
}
```

Особенности OptionClass

- Решает проблему null почти полностью
- Аллокации в куче на каждый экземпляр
- Хранение флага на каждый экземпляр
- Ссылка на OptionClass сама может быть null

Структура-обертка

```
public readonly struct OptionStruct<T> : IOption<T>, IEquatable<OptionStruct<T>>
{
    internal OptionStruct(in T value) => (HasValue, Value) = (true, value);

    public bool HasValue { get; }
    public bool TryGetValue(out NotNull<T> value)
    {
        value = Value;
        return HasValue;
    }
    NotNull<T> Value { get; }
    . . .
}
```

Особенности OptionStruct

- Нет аллокаций в куче
- Неинициализированная структура — корректное пустое значение
- Интерфейс качественно лучше , чем у Nullable
- Хранение флага на каждый экземпляр

Структура-обертка для ссылок

```
public readonly struct Option<T> : IOption<T>, IEquatable<Option<T>>
    where T : class
{
    internal Option([CanBeNull]T value) => Value = value;

    public bool TryGetValue(out NotNull<T> value)
    {
        value = new NotNull<T>(Value);
        return HasValue;
    }
    public bool HasValue => Value != null;
    T Value { get; }
    . . .
}
```

Особенности Option

- Нулевая дополнительная стоимость по памяти
- Неинициализированная структура — корректное пустое значение
- Интерфейс качественно лучше , чем у Nullable
- Не может работать с типами-значениями

Проблемы обобщений в C#

- Ограничения не учитываются при выборе перегруженного метода
- Ограничения не учитываются при выводе типов
- Надо выводить все типы-параметры или указывать все их явно

Перегрузка методов и ограничения

- Код ниже выдаст ошибку компиляции
- У обоих методов с точки зрения компилятора идентичные сигнатуры
- Разные ограничения компилятор учитывать не будет

```
void Method<T>(T argument) where T : Interface1;

void Method<T>(T argument) where T : Interface2;
```

Вывод типов и ограничения

- Метод ниже всегда будет требовать указания всех параметров-типов вручную
- Тип T можно вывести из TOption через ограничение, но компилятор этого делать не будет

```
void Method<T, TOption>(TOption argument)
    where TOption : IOption<T>;
```

Все или ничего

- Метод ниже всегда будет требовать указания всех параметров-типов вручную
- Тип TKey выводится из параметра, хотелось бы тип T указать явно
- Компилятор частичный вывод типов-параметров делать не будет

```
public static Id<T, TKey> AsId<T, TKey>
    (this TKey id)
```

Главный вопрос

- Неужели даже работу с Option нельзя написать обобщенно и эффективно?

Главный ответ

```
public readonly struct Option<T, TO>
    : IOption<T>, IEquatable<Option<T, TO>>
    where TO : IOption<T>
{
    public Option(in TO option) => Unwrap = option;

    public bool TryGetValue(out NotNull<T> value)
        => Unwrap.TryGetValue(out value);
    public bool HasValue => Unwrap.HasValue;
    public TO Unwrap { get; }
    . . .
}
```

Особенности Option с двумя параметрами

- Хранение флага только если это необходимо
- Обобщенный исходный код
- Генерация двоичного кода для каждого значимого типа-параметра, включая NotNull<T> и Null<T>
- Можно добавлять свои реализации Ioption<T>
- Потенциально обертки могут быть полностью выпилены при оптимизации JIT

Ложка дегтя

- Сигнатуры обобщенных методов страшноватые
- Сигнатуры методов расширения жуткие
- Компилятор пока оптимизирует далеко не все, что хотелось бы

```
void Method<T, Toption>
    (Option<T, TOption> argument)
    where TOption : IOption<T>;
```

Так в чем же отличие значимых типов?

- При подготовке доклада не использовались специальные знания о типах
- Для ответов на обе задачи достаточно информации из Рихтера
- Приемы из доклада актуальны для .NET версии 2.0 и выше

Вопросы?

leo.bonart@gmail.com

https://github.com/Kirill-Maurin/Sample.Struct