C# Developer. Professional

Serialization

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

Сериализация

Елена Сычева

Team Lead Full-Stack Developer

Об опыте:

Более 15 лет опыта работы разработчиком (C#, Angular, .Net, React, NodeJs)

Телефон / эл. почта / соц. сети:
https://t.me/lentsych

Правила вебинара

Активно
участвуем

Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения

Индивидуально

Время, необходимое
на активность

Пишем в чат

Говорим голосом

Документ

Ответьте себе или
задайте вопрос

Маршрут вебинара

- Что это? Зачем это?
- Форматы
- Интерфейсы
- Стандартные инструменты
- Практика. Сравнение скорости
- Proto gRPC

Цели вебинара

К концу занятия вы сможете

1. использовать механизмы сериализации и результирующие форматы сериализации;
2. применять стандартные способы сериализации.
3. Создавать кастомную сериализацию

Смысл

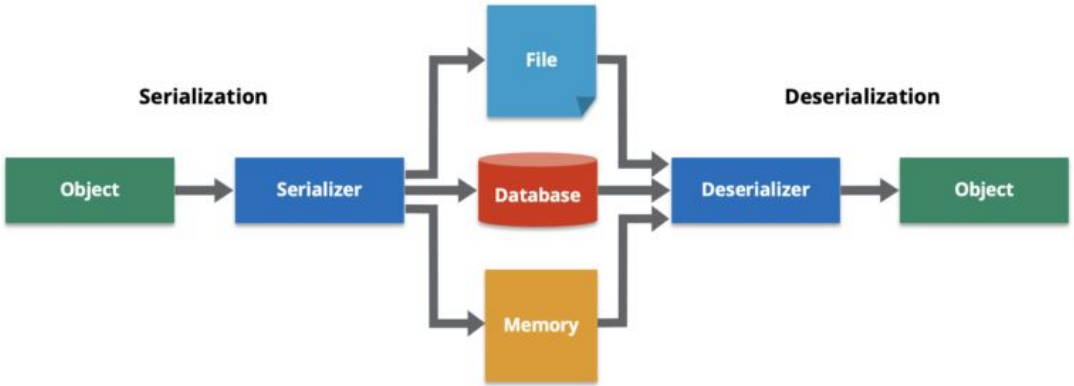
Зачем вам это уметь

1. Это фундаментальная концепция, лежащая в основе многих аспектов современной разработки программного обеспечения
2. Дает навыки более эффективной обработки данных, обеспечения бесперебойной связи между различными частями приложения и поддержания целостности данных на различных платформах и системах

Сериализация. Что это?

Сериализация

— это процесс преобразования объекта в формат, который можно легко сохранить или передать. Этот формат обычно является двоичным или текстовым представлением, которое сохраняет состояние и данные объекта. Преобразовав объект в serial формат, становится возможным сохранить его в файле, отправить по сети или сохранить в базе данных.



Назначение

Основная цель сериализации — сохранение состояния объекта на носителе или передача его по сети. Это важно для различных приложений, таких как:

1. **Сохранение данных**: сохранение состояния объекта в файле или базе данных для последующего извлечения.
2. **Связь**: отправка объектов по сети между различными системами или компонентами, включение удаленных вызовов процедур (RPC) и веб-служб.
3. **Резервное копирование и восстановление**: создание резервных копий объектов и их восстановление по мере необходимости для обеспечения целостности и непрерывности данных.

Десериализация

Десериализация — это процесс преобразования сериализованных данных обратно в объект. Этот обратный процесс берет сериализованный формат и реконструирует исходный объект с его состоянием и данными в целости и сохранности.

Десериализация имеет решающее значение для:

- Чтения сохраненных данных: загрузки ранее сохраненных объектов из файла или базы данных для восстановления их состояния.
- Приема данных: получения сериализованных объектов по сети и их реконструкции для использования в приложениях.
- Обмена данными: упрощения обмена данными между различными системами путем преобразования сериализованных данных в пригодные для использования объекты.

Почему актуально

1. Сохранение данных

Сохранение и загрузка состояния: сериализация позволяет сохранять состояние объекта в хранилище (например, файл или базу данных) и восстанавливать его позже, что важно для приложений, которым необходимо запоминать пользовательские настройки, игровые состояния или другие постоянные данные.

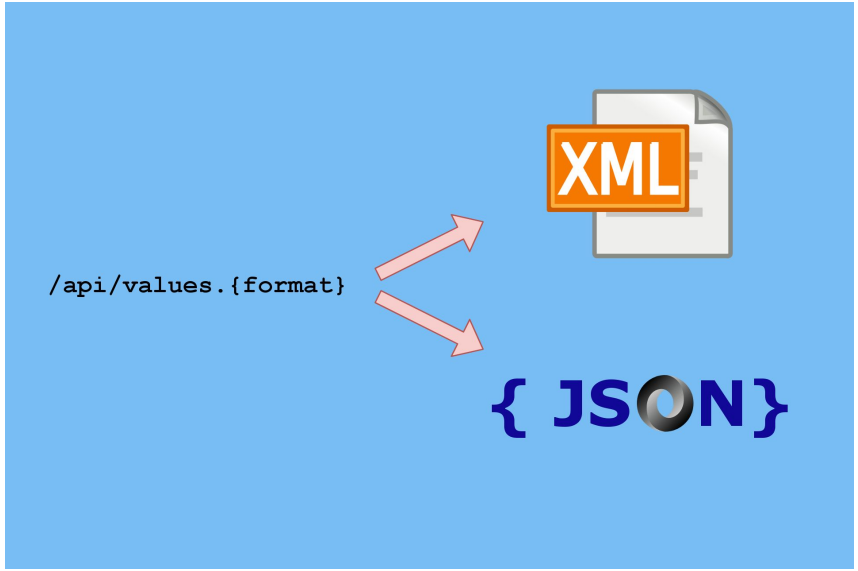
Резервное копирование и восстановление: помогает создавать резервные копии состояний объектов и восстанавливать их по мере необходимости, обеспечивая целостность и непрерывность данных.

Почему актуально

2. Передача данных

Сетевая связь: сериализация используется для отправки данных по сетям. Например, в веб-приложениях данные часто сериализуются в JSON или XML перед отправкой клиенту или серверу.

Веб-сервисы: многие веб-сервисы, такие как RESTful API, используют форматы сериализации, такие как JSON и XML, для связи между клиентами и серверами.

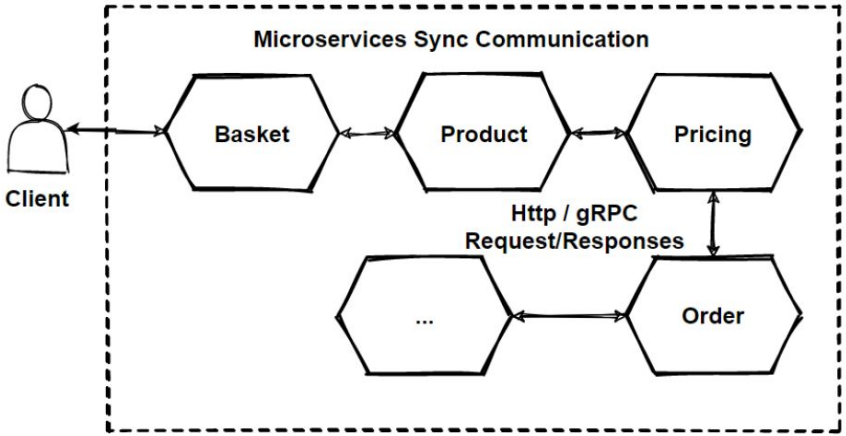


Почему актуально

3. Взаимодействие

Кроссплатформенная совместимость: сериализация позволяет различным системам (возможно, написанным на разных языках) обмениваться данными. Для этой цели широко используются такие форматы, как JSON, XML и Protocol Buffers.

Микросервисы: в архитектурах микросервисов сервисы часто взаимодействуют посредством сериализованных сообщений, что обеспечивает бесперебойное взаимодействие между сервисами.

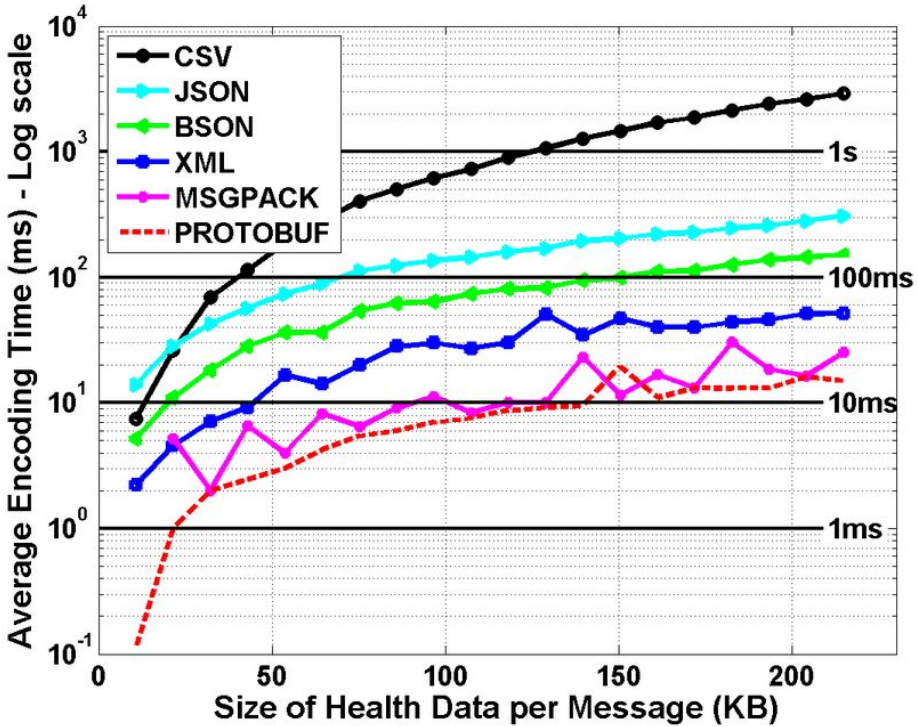


Почему актуально

4. Оптимизация производительности

Эффективность: понимание различных форматов сериализации помогает выбрать наиболее эффективный для вашего варианта использования. Например, Protocol Buffers намного быстрее и компактнее по сравнению с JSON и XML.

Управление ресурсами: правильная сериализация может сократить объем передаваемых или хранимых данных, оптимизируя использование ресурсов.

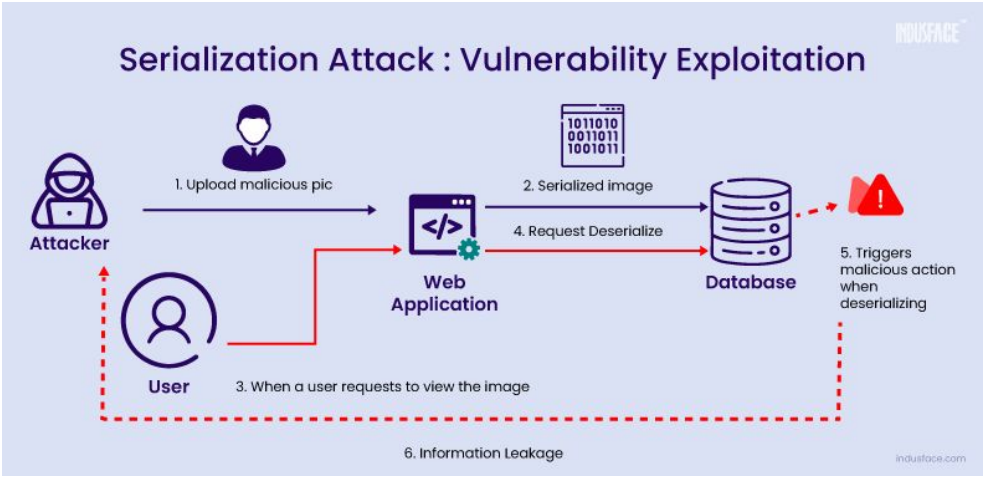


Почему актуально

5. Безопасность

Безопасная обработка данных: знание недостатков определенных форматов сериализации (например, уязвимостей в BinaryFormatter) помогает делать безопасный выбор и избегать потенциальных рисков безопасности.

Проверка данных: сериализация часто включает проверку форматов данных, что может быть дополнительным уровнем безопасности от некорректных или вредоносных данных.

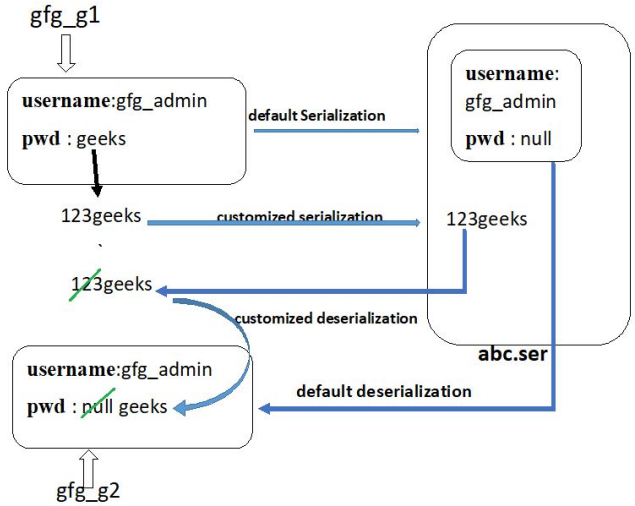


Почему актуально

6. Гибкость и контроль

Пользовательская сериализация: реализация пользовательской сериализации позволяет осуществлять детальный контроль над тем, как сериализуются и десериализуются объекты, что может быть критически важным для сложных или конфиденциальных структур данных.

Версионирование: Правильные методы сериализации могут обрабатывать версии, позволяя приложениям развиваться, не нарушая совместимости со старыми версиями сериализованных данных.

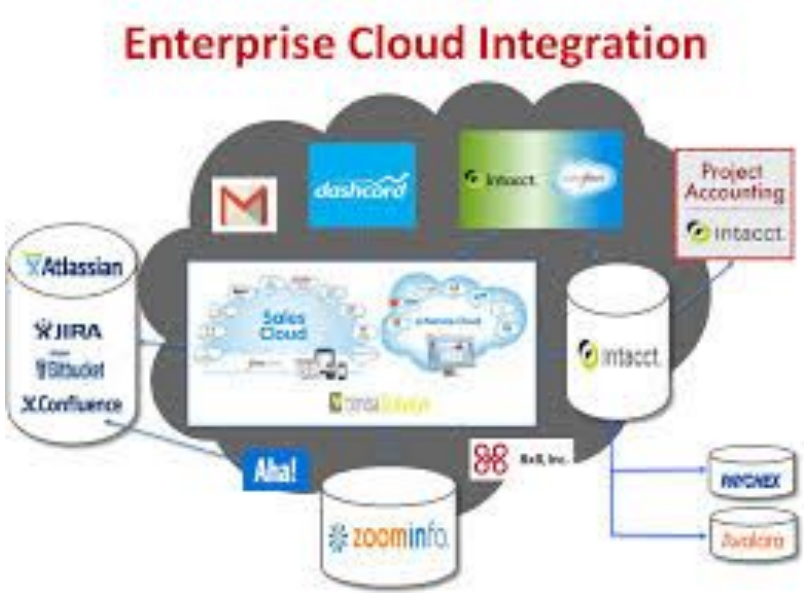


Почему актуально

7. Реальные приложения

Корпоративные системы: Многие корпоративные системы в значительной степени полагаются на сериализацию для обмена данными между различными службами и системами.

Облачные вычисления: Облачные приложения и службы часто используют сериализацию для эффективного хранения и извлечения данных.



Serialization Formats

Binary Serialization

Плюсы:

Быстрота: Двоичная сериализация, как правило, быстрее текстовых форматов, поскольку она напрямую преобразует объекты в двоичную форму.
Эффективное хранение: Двоичные данные занимают меньше места по сравнению с текстовыми форматами, что делает их более эффективными для хранения.

Минусы:

Зависимость от платформы: Двоичная сериализация часто зависит от платформы, то есть данные, сериализованные на одной платформе, могут не поддаваться десериализации на другой.
Неудобочитаемость для человека: Двоичные данные нелегко читаются человеком, что затрудняет отладку и ручное редактирование.
Риски безопасности: Двоичная сериализация может быть подвержена уязвимостям безопасности, особенно если данные поступают из ненадежных источников.

Binary Serialization

```
public class BinarySerializationExample
{
    public static void Main()
    {
        Person person = new Person { Name = "Alice", Age = 30 };
        BinaryFormatter formatter = new BinaryFormatter();

        using (FileStream stream = new FileStream("person.bin", FileMode.Create))
        {
            formatter.Serialize(stream, person);
        }

        using (FileStream stream = new FileStream("person.bin", FileMode.Open))
        {
            Person deserializedPerson = (Person)formatter.Deserialize(stream);
            Console.WriteLine($"{deserializedPerson.Name}, {deserializedPerson.Age}");
        }
    }
}
```

```
[Serializable]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Binary Serialization

Уязвимости:

Удаленное выполнение кода (RCE): BinaryFormatter может быть использован для удаленного выполнения кода, если ненадежные данные десериализуются. Злонамеренные субъекты могут создать сериализованную полезную нагрузку, которая при десериализации выполняет произвольный код в целевой системе. Это особенно опасно в сценариях, где сериализованные данные получены из ненадежных источников по сети.
Небезопасная десериализация: Десериализация данных без надлежащей проверки или безопасности может привести к различным проблемам безопасности, включая подделку данных, утечку данных и повышение привилегий. Злоумышленники могут манипулировать сериализованными данными, чтобы внедрить вредоносный контент или изменить состояние десериализованного объекта.

Binary Serialization

Рекомендации:

Избегайте использования BinaryFormatter: учитывая присущие ему риски безопасности, рекомендуется избегать использования BinaryFormatter в новых приложениях. Microsoft объявила BinaryFormatter устаревшим и исключила из .NET 8.0, рекомендует использовать более безопасные альтернативы.

Использование альтернатив: рассмотрите возможность использования современных и безопасных библиотек сериализации, таких как System.Text.Json, Newtonsoft.Json или Protocol Buffers (protobuf). Эти библиотеки обеспечивают лучшую безопасность и производительность.

Проверка данных: если необходима десериализация, убедитесь, что данные проверены и очищены перед десериализацией. Реализуйте проверки безопасности и используйте безопасные методы кодирования для снижения рисков.

XML Serialization

Плюсы:

Человекочитаемый: XML основан на тексте и понятен человеку, что упрощает отладку и редактирование вручную.
Независимость от платформы: XML — это стандартный формат, который можно использовать на разных платформах и языках программирования.
Широкое использование: XML широко применяется во многих отраслях для обмена данными.

Минусы:

Многословный: XML, как правило, более многословен, чем другие форматы, что приводит к увеличению размера файлов.
Медленнее: синтаксический анализ и сериализация XML обычно медленнее по сравнению с двоичными форматами.

XML Serialization

```
public class XMLSerializationExample
{
    public static void Main()
    {
        Person person = new Person { Name = "Alice", Age = 30 };
        XmlSerializer serializer = new XmlSerializer(typeof(Person));

        using (StreamWriter writer = new StreamWriter("person.xml"))
        {
            serializer.Serialize(writer, person);
        }

        using (StreamReader reader = new StreamReader("person.xml"))
        {
            Person deserializedPerson = (Person)serializer.Deserialize(reader);
            Console.WriteLine($"{deserializedPerson.Name}, {deserializedPerson.Age}");
        }
    }
}
```

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

JSON Serialization

Плюсы:

Человекочитаемый: JSON основан на тексте и удобен для чтения человеком, что упрощает понимание и отладку.
Независимость от платформы: JSON — широко распространенный стандарт, совместимый со многими платформами и языками программирования.
Широкое использование: JSON является фактическим стандартом обмена данными в веб-сервисах и API.

Минусы:

Менее эффективный: JSON менее эффективен с точки зрения размера и скорости по сравнению с двоичными форматами.

JSON Serialization

```
public class JSONSerializationExample
{
    public static void Main()
    {
        Person person = new Person { Name = "Alice", Age = 30 };
        string jsonString = JsonSerializer.Serialize(person);
        Console.WriteLine(jsonString);

        Person deserializedPerson = JsonSerializer.Deserialize<Person>(jsonString);
        Console.WriteLine($"{deserializedPerson.Name}, {deserializedPerson.Age}");
    }
}
```

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

SOAP Serialization

Плюсы:

Расширяемость: SOAP обладает высокой расширяемостью и поддерживает сложные типы данных и шаблоны связи.
Стандартизирован для веб-сервисов: SOAP — это стандартизированный протокол, широко используемый в веб-сервисах корпоративного уровня.

Минусы:

Многословность: сообщения SOAP очень многословны, что приводит к увеличению размера сообщений.
Медленнее: анализ и обработка сообщений SOAP могут быть медленнее по сравнению с другими форматами.
Менее популярен сегодня: с ростом популярности служб RESTful и JSON SOAP стал менее популярным.

SOAP Serialization

```
public class SOAPSerializationExample
{
    public static void Main()
    {
        Person person = new Person { Name = "Alice", Age = 30 };
        SoapFormatter formatter = new SoapFormatter();

        using (FileStream stream = new FileStream("person.soap", FileMode.Create))
        {
            formatter.Serialize(stream, person);
        }

        using (FileStream stream = new FileStream("person.soap", FileMode.Open))
        {
            Person deserializedPerson = (Person)formatter.Deserialize(stream);
            Console.WriteLine($"{deserializedPerson.Name}, {deserializedPerson.Age}");
        }
    }
}
```

```
[Serializable]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Custom Serialization

Плюсы:

Полный контроль: Пользовательская сериализация обеспечивает полный контроль над тем, как сериализуются и десериализуются объекты.
Гибкость: Вы можете оптимизировать сериализацию для конкретных случаев использования и требований.

Минусы:

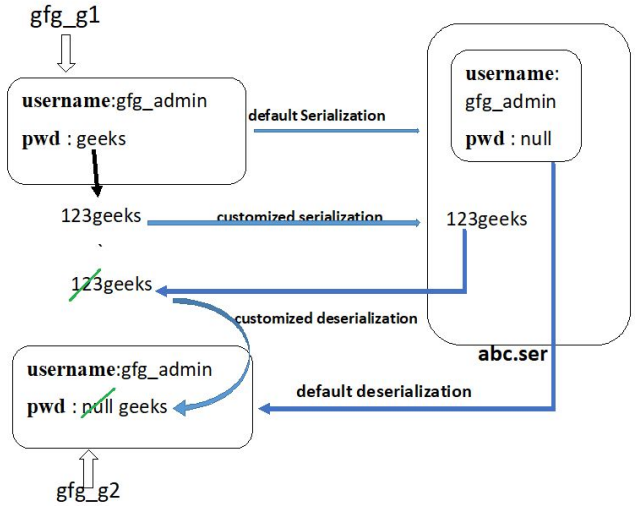
Требуемые усилия: Реализация пользовательской сериализации требует больше усилий и понимания процесса сериализации.

Почему актуально

Гибкость и контроль

Пользовательская сериализация: реализация пользовательской сериализации позволяет осуществлять детальный контроль над тем, как сериализуются и десериализуются объекты, что может быть критически важным для сложных или конфиденциальных структур данных.

Версионирование: Правильные методы сериализации могут обрабатывать версии, позволяя приложениям развиваться, не нарушая совместимости со старыми версиями сериализованных данных.



Custom Serialization

```
public class CustomSerializationExample
{
    public static void Main()
    {
        Person person = new Person { Name = "Alice", Age = 30 };

        IFormatter formatter = new BinaryFormatter();
        using (FileStream stream = new FileStream("person.custom", FileMode.Create))
        {
            formatter.Serialize(stream, person);
        }

        using (FileStream stream = new FileStream("person.custom", FileMode.Open))
        {
            Person deserializedPerson = (Person)formatter.Deserialize(stream);
            Console.WriteLine($"{deserializedPerson.Name}, {deserializedPerson.Age}");
        }
    }
}
```

```
[Serializable]
public class Person : ISerializable
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person() { }

    protected Person(SerializationInfo info, StreamingContext context)
    {
        Name = info.GetString("Name");
        Age = info.GetInt32("Age");
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("Name", Name);
        info.AddValue("Age", Age);
    }
}
```

Serializable атрибут

.NET Core

[Serializable]: помечает класс как сериализуемый, позволяя сериализовать и десериализовать экземпляры класса.

[DataContract] , [DataMember]: используются для определения контрактов данных в службах WCF (Windows Communication Foundation) для сериализации и десериализации данных

[JsonProperty] (Newtonsoft.Json or System.Text.Json.Serialization): позволяет указывать имена свойств JSON во время сериализации и десериализации объектов.

Почему некоторые форматы требуют атрибута Serializable

Двоичная сериализация (например, с использованием BinaryFormatter) требует атрибута Serializable. Это связано с тем, что среде выполнения .NET требуется явное разрешение на сериализацию внутреннего состояния объектов, включая закрытые поля.
Без атрибута Serializable среда выполнения не может гарантировать безопасность и целостность сериализованных данных, что может привести к непредсказуемому поведению или уязвимостям безопасности.

Пользовательская сериализация:
Пользовательская сериализация (например, реализация ISerializable) также требует атрибута Serializable. Это позволяет разработчикам контролировать процесс сериализации, настраивая способ записи и чтения данных объекта.

DataContractSerializer:
DataContractSerializer, который часто используется для сериализации XML и JSON более контролируемым и гибким образом, требует атрибутов Serializable или DataContract, чтобы пометить типы для сериализации. Это помогает указать, какие элементы сериализуются, и обеспечивает поддержку управления версиями.

Почему некоторые форматы НЕ требуют атрибута Serializable

XmlSerializer не требует атрибута Serializable, поскольку он использует общедоступные свойства и поля для сериализации. Ему не нужен доступ к закрытым членам, и он полагается на более простой подход на основе отражения.
Это обеспечивает большую гибкость, но означает, что закрытые данные не сериализуются по умолчанию, и разработчикам необходимо явно контролировать, что сериализуется через открытые члены.

JsonSerializer (System.Text.Json и Newtonsoft.Json):

Сериализаторам JSON обычно не требуется атрибут Serializable. Они полагаются на общедоступные свойства и поля для сериализации, подобно XmlSerializer.
Эти сериализаторы используют отражение для обнаружения структуры объектов и сериализации общедоступных членов, что упрощает работу с классами POCO (Plain Old CLR Objects) без дополнительных атрибутов.

Проверки во время выполнения и во время компиляции

Применение атрибута Serializable происходит во время выполнения, а не во время компиляции.

Почему?

- Гибкость
- Динамическое поведение
- Обратная совместимость
- Обработка ошибок

```
[Serializable]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    [NonSerialized]
    private int secretCode; // This field will not be serialized

    public Person(string name, int age, int secretCode)
    {
        Name = name;
        Age = age;
        this.secretCode = secretCode;
    }
}
```

Интерфейсы

ISerializable Interface

Назначение:

Интерфейс ISerializable позволяет объекту управлять собственными процессами сериализации и десериализации. Это особенно полезно, когда вам нужно настраиваемое поведение во время сериализации, например, управление закрытыми полями или управление версиями.

Реализация:

Чтобы реализовать интерфейс ISerializable, класс должен определить метод **GetObjectData**. Кроме того, класс должен предоставить специальный **конструктор, который принимает параметры SerializationInfo и StreamingContext для десериализации**.

```
// Special constructor for deserialization
protected Person(SerializationInfo info, StreamingContext context)
{
    Name = info.GetString("Name");
    Age = info.GetInt32("Age");
    SecretCode = info.GetInt32("SecretCode");
}

// Method to control serialization
public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue("Name", Name);
    info.AddValue("Age", Age);
    info.AddValue("SecretCode", SecretCode);
}
```

IFormatter Interface

Назначение:

Интерфейс IFormatter предоставляет функциональные возможности для форматирования сериализованных объектов. Он определяет методы сериализации и десериализации объектов, предоставляя разработчикам контроль над форматом сериализации.

Реализация:

- Serialize(Stream, Object): сериализует объект и записывает его в предоставленный поток.
- Deserialize(Stream): десериализует данные из предоставленного потока и восстанавливает исходный объект.

IFormatter Interface

```
public class SimpleFormatter : IFormatter
{
    public SerializationBinder Binder { get; set; }
    public StreamingContext Context { get; set; }
    public ISurrogateSelector SurrogateSelector { get; set; }

    public SimpleFormatter()
    {
        Context = new StreamingContext(StreamingContextStates.All);
    }
```

```
[Serializable]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public override string ToString()
    {
        return $"Name: {Name}, Age: {Age}";
    }
}
```

```
public void Serialize(Stream stream, object obj)
{
    using (StreamWriter writer = new StreamWriter(stream))
    {
        if (obj is Person person)
        {
            writer.WriteLine(person.Name);
            writer.WriteLine(person.Age);
        }
    }
}

public object Deserialize(Stream stream)
{
    using (StreamReader reader = new StreamReader(stream))
    {
        string name = reader.ReadLine();
        int age = int.Parse(reader.ReadLine());
        return new Person(name, age);
    }
}
```

IFormatter Interface

```
public void Serialize(Stream stream, object obj)
{
    using (StreamWriter writer = new StreamWriter(stream))
    {
        if (obj is Person person)
        {
            writer.WriteLine(person.Name);
            writer.WriteLine(person.Age);
        }
    }
}

public object Deserialize(Stream stream)
{
    using (StreamReader reader = new StreamReader(stream))
    {
        string name = reader.ReadLine();
        int age = int.Parse(reader.ReadLine());
        return new Person(name, age);
    }
}
```

IFormatter Interface

```
public class CustomFormatterExample
{
    public static void Main()
    {
        Person person = new Person("Alice", 30);

        // Serialize the object using custom formatter
        IFormatter formatter = new SimpleFormatter();
        using (FileStream stream = new FileStream("person.txt", FileMode.Create))
        {
            formatter.Serialize(stream, person);
        }

        // Deserialize the object using custom formatter
        using (FileStream stream = new FileStream("person.txt", FileMode.Open))
        {
            Person deserializedPerson = (Person)formatter.Deserialize(stream);
            Console.WriteLine(deserializedPerson);
        }
    }
}
```

Итоги

Интерфейс **ISerializable**

Цель: позволяет объекту управлять собственной сериализацией и десериализацией.
Реализация: требует метод GetObjectData и специальный конструктор для десериализации.
Вариант использования: когда вам нужно настроить способ сериализации объекта, например, включить закрытые поля или управлять сложной логикой сериализации.

Интерфейс **IFormatter**

Цель: предоставляет функциональность для форматирования сериализованных объектов.
Методы: Serialize(Stream, Object) и Deserialize(Stream).
Вариант использования: когда вам нужно определить пользовательский формат сериализации, например, фирменный двоичный формат или упрощенный текстовый формат.

Стандартные средства

BinaryFormatter (Deprecated)

BinaryFormatter используется для двоичной сериализации, преобразуя объекты в двоичный формат. Он сериализует все элементы, включая закрытые поля, обеспечивая полное и компактное представление объектов. Удален по причине уязвимостей

XmlSerializer

XmlSerializer используется для сериализации XML, преобразуя объекты в формат XML. Он сериализует общедоступные свойства и поля, что делает его пригодным для совместимого обмена данными.

DataContractSerializer

DataContractSerializer используется для сериализации XML и JSON. Он предоставляет возможность сериализации и десериализации объектов на основе контракта, что делает его пригодным для сервис-ориентированных приложений.

```
[DataContract]
public class Person
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public int Age { get; set; }
}
```

DataContractSerializer

```
public class DataContractSerializerExample
{
    public static void Main()
    {
        Person person = new Person { Name = "Alice", Age = 30 };
        DataContractSerializer serializer = new DataContractSerializer(typeof(Person));

        // Serialize the object to XML
        using (FileStream stream = new FileStream("person.xml", FileMode.Create))
        {
            serializer.WriteObject(stream, person);
        }

        // Deserialize the object from XML
        using (FileStream stream = new FileStream("person.xml", FileMode.Open))
        {
            Person deserializedPerson = (Person)serializer.ReadObject(stream);
            Console.WriteLine($"{deserializedPerson.Name}, {deserializedPerson.Age}");
        }
    }
}
```

JsonSerializer (System.Text.Json)

System.Text.Json.JsonSerializer — это современная и эффективная библиотека для сериализации JSON. Он включен в .NET Core 3.0 и более поздних версий и обеспечивает высокую производительность и безопасность.

Newtonsoft.Json

```
public class NewtonsoftJsonExample
{
    public static void Main()
    {
        Person person = new Person { Name = "Alice", Age = 30 };

        // Serialize the object to JSON
        string jsonString = JsonConvert.SerializeObject(person);
        Console.WriteLine(jsonString);

        // Deserialize the object from JSON
        Person deserializedPerson = JsonConvert.DeserializeObject<Person>(jsonString);
        Console.WriteLine($"{deserializedPerson.Name}, {deserializedPerson.Age}");
    }
}
```

System.Text.Json vs Newtonsoft.Json

System.Text.Json	Newtonsoft.Json
● Представлено в .NET Core 3.0. ● Встроенная библиотека в .NET Core и более поздних версиях. ● Разработан, чтобы быть легким и производительным.	● Сторонняя библиотека, также известная как Json.NET. ● Очень популярен и широко используется в сообществе .NET. ● Предоставляет богатый набор функций и возможностей настройки.

System.Text.Json vs Newtonsoft.Json

Производительность

System.Text.Json

- Как правило, быстрее с точки зрения сериализации и десериализации благодаря облегченной конструкции.
- Оптимизирован для высокой производительности, особенно в сценариях, где требуется минимальная настройка.
- Сравнительные тесты часто показывают, что System.Text.Json работает быстрее в простых случаях использования.

Newtonsoft.Json

- Немного медленнее, чем System.Text.Json, особенно в простых сценариях.
- Разница в производительности может быть незначительной для сложных задач сериализации и десериализации, в которых используются расширенные функции.

System.Text.Json vs Newtonsoft.Json

Возможности

System.Text.Json	Newtonsoft.Json
• Ограниченный набор функций по сравнению с Newtonsoft.Json. • Базовые возможности сериализации и десериализации. • Поддерживает пользовательские конвертеры, но с менее интуитивно понятным API. • Отсутствие поддержки некоторых сложных сценариев, таких как полиморфная десериализация, аннотации данных и более сложная настройка.	• Богатый набор функций, включая расширенные параметры сериализации и десериализации. • Поддерживает пользовательские преобразователи, преобразователи контрактов и расширенную настройку. • Предоставляет такие функции, как проверка схемы JSON, LINQ to JSON и надежная обработка ошибок. • Расширенная поддержка атрибутов для управления поведением сериализации (например, JsonProperty, JsonIgnore).

ProtoBuf

Что такое Proto (буферы протоколов)?

Protocol Buffers (Proto) — это не зависящий от языка и платформы, расширяемый механизм для сериализации структурированных данных.

Разработанный Google, Proto используется для определения структур данных и их эффективной сериализации.

Основные характеристики:

Компактность: буферы протоколов генерируют меньшие двоичные представления по сравнению с JSON или XML.
Скорость: сериализация и десериализация выполняются быстрее благодаря компактному двоичному формату.
Поддержка языков: поддерживает несколько языков программирования, включая C#, Java, Python и другие.

Что такое Proto (буферы протоколов)?

```
syntax = "proto3";

message Person {
  string name = 1;
  int32 age = 2;
  string email = 3;
}
```

```
syntax = "proto3";

service PersonService {
  rpc GetPerson(PersonRequest) returns (PersonResponse);
}

message PersonRequest {
  int32 id = 1;
}

message PersonResponse {
  string name = 1;
  int32 age = 2;
  string email = 3;
}
```

Что такое Proto (буферы протоколов)?

```
public class Person : IMessage<Person>
{
    // Generated code for serialization and deserialization
    // based on the .proto file definitions

    // Properties
    public string Name { get; set; }
    public int Age { get; set; }

    // Serialization and deserialization methods
    public void WriteTo(CodedOutputStream output)
    {
        output.WriteString(Name);
        output.WriteInt32(Age);
    }

    public int CalculateSize()
    {
        // Calculate size based on fields
        return CodedOutputStream.ComputeStringSize(Name) + CodedOutputStream.ComputeInt32S
    }

    public void MergeFrom(CodedInputStream input)
    {
        // Deserialize fields
        Name = input.ReadString();
        Age = input.ReadInt32();
    }

    // Additional required methods omitted for brevity
}
```

Что такое Proto (буферы протоколов)?

```
public class ProtobufExample
{
    public static void Main()
    {
        Person person = new Person { Name = "Alice", Age = 30 };

        // Serialize the object
        using (FileStream stream = new FileStream("person.bin", FileMode.Create))
        {
            person.WriteTo(CodedOutputStream.CreateInstance(stream));
        }

        // Deserialize the object
        Person deserializedPerson = new Person();
        using (FileStream stream = new FileStream("person.bin", FileMode.Open))
        {
            deserializedPerson.MergeFrom(CodedInputStream.CreateInstance(stream));
        }
        Console.WriteLine($"{deserializedPerson.Name}, {deserializedPerson.Age}");
    }
}
```

Почему безопасен?

У BinaryFormatter есть серьезные проблемы с безопасностью из-за включения информации о типе в сериализованные данные, способности реконструировать сложные графы объектов и возможности удаленного выполнения кода посредством обратных вызовов десериализации.

Protobuf снижает эти риски, используя предопределенную схему, избегая информации о типах во время выполнения, обеспечивая безопасность типов и ограничивая сложность десериализованных объектов. Такая конструкция делает Protobuf более безопасным выбором для двоичной сериализации, особенно в сценариях, включающих ненадежные данные.

Список материалов для изучения

1. https://learn.microsoft.com/ru-ru/dotnet/standard/serialization/
2. https://learn.microsoft.com/ru-ru/dotnet/standard/serialization/controlling-xml-serialization-using-attributes
3. https://www.newtonsoft.com/json/help/html/SerializingJSON.htm
4. https://protobuf.dev/getting-started/csharptutorial/

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Рефлексия

Рефлексия

С какими впечатлениями уходите с вебинара?

Как будете применять на практике то,
что узнали на вебинаре?