Сериализация

**Определение**

Сериализация — процесс перевода какой-либо структуры данных в последовательность битов.

Обратной к операции сериализации является операция десериализации (структуризации) — восстановление начального состояния структуры данных из битовой последовательности.

**Применение**

Сериализация используется для передачи объектов по сети и для сохранения их в файлы.

*Например, нужно создать распределенное приложение, разные части которого должны обмениваться данными со сложной структурой. В таком случае для типов данных, которые предполагается передавать, пишется код, который осуществляет сериализацию и десериализацию. Объект заполняется нужными данными, затем вызывается код сериализации. Результат сериализации передаётся каким-либо образом принимающей стороне. Приложение-получатель создает объект того же типа и вызывает код десериализации, в результате получая объект с теми же данными, что были в объекте, который мы отправили.*

Также сериализация может использоваться при осуществлении удаленных вызовов процедур или для обнаружения изменений в данных, которые изменяются со временем.

Стоит отметить, что для наиболее эффективного использования этих возможностей необходимо поддерживать независимость от архитектуры. *Например, необходимо иметь возможность надёжно воссоздавать сериализованный поток данных независимо от порядка байтов, использующегося в данной архитектуре. Это значит, что наиболее простая и быстрая процедура прямого копирования участка памяти, в котором размещается структура данных, не может работать надежно для всех архитектур.*

**Недостатки**

* Сериализация нарушает непрозрачность абстрактного типа данных, потенциально раскрывая частные детали реализации.
* В частности, тривиальные реализации, которые сериализуют все элементы данных, могут нарушать инкапсуляцию.
* В идеале сериализацию не стоит применять для больших объектов, так как при сериализации происходит полное считывание объекта от начала до конца.

*Так, например, если в объекте содержатся ссылки на другие объекты, то они также будут полностью считаны и сериализованы, если не указан запрет на сериализацию.*

**Форматы**

Существует множество форматов (как текстовых, так и бинарных), в которых можно хранить сериализованные данные. Например,

* Текстовые:
  + JSON
  + XML
  + YAML
* Бинарные:
  + BSON
  + Protobuf
  + CBOR
  + MessagePack

**Примеры**

JSON

Следующий пример показывает JSON-представление данных об объекте, описывающем человека. В данных присутствуют строковые поля имени и фамилии, информация об адресе и массив, содержащий список телефонов. Как видно из примера, значение может представлять собой вложенную структуру.

|  |
| --- |
| {  **"firstName"**: **"Иван"**,  **" lastName"**: **"Иванов"**,  **"address"**: {  **"streetAddress"**: **"Московское ш., 101, кв.101"**,  **"city"**: **"Ленинград"**,  **"postalCode"**: 101101  },  **"phoneNumbers"**: [  **"812 123-1234"**,  **"916 123-4567"**  ]  } |

XML

На языке XML подобная структура выглядела бы примерно так:

|  |
| --- |
| <**person**>  <**firstName**>Иван</**firstName**>  <**lastName**>Иванов</**lastName**>  <**address**>  <**streetAddress**>Московское ш., 101, кв.101</**streetAddress**>  <**city**>Ленинград</**city**>  <**postalCode**>101101</**postalCode**>  </**address**>  <**phoneNumbers**>  <**phoneNumber**>812 123-1234</**phoneNumber**>  <**phoneNumber**>916 123-4567</**phoneNumber**>  </**phoneNumbers**>  </**person**> |

Сериализация в Kotlin

Сериализация в Kotlin осуществляется с помощью плагина kotlinx.serialization

**Возможности**

* Поддержка классов и коллекций Kotlin
* Поддерживаемые форматы: JSON, CBOR, Protobuf
* Полная поддержка мультиплатформенности: JVM, JS, Native
* Можно создавать собственные сериализаторы

**Инициализация**

Для того чтобы класс можно было сериализировать достаточно указать перед ним аннотацию @Serializable. Для данного класса будет создан объект-компаньон со специальной функцией serializer(), которая возвращает объект типа KSerializer<T>. KSerializer<T> реализует интерфейсы SerializationStrategy<T> и DeserializationStrategy<T>. Таким образом, данный объект будет одновременно выполнять функцию сериализатора и десериализатор.

|  |
| --- |
| @Serializable  data class Data(val a: Int, val b: String = "42") {  val list = *listOf*(1, 2, 3, 4, 5)  }  val *dataSerializer*: KSerializer<Data> = Data.serializer() |

**Взаимодействие с текстовыми форматам**

Далее разберемся как представить объект в текстовом формате. Подробно разберём формат JSON. Взаимодействие с бинарными форматами Protobuf и CBOR происходит аналогично.

Реализация

Формат JSON представлен классом Json, который настраивается с помощью вспомогательного класса JsonConfiguration. Чтобы не настраивать каждый раз JsonConfiguration, существует два основных пресета: Default и Stable. Default рекомендуется к использованию, однако из-за эволюции библиотеки его конфигурация может быть изменена, если для вашего проекта это критично, то следует использовать пресет Stable, который не будет изменяться с выходом новых библиотек.

API

|  |
| --- |
| fun <T> stringify(serializer: SerializationStrategy<T>, obj: T): String  fun <T> parse(loader: DeserializationStrategy<T>, str: String): T |

Пример

|  |
| --- |
| @Serializable  data class Data(val a: Int, val b: String = "42") {  val list = *listOf*(1, 2, 3, 4, 5)  }  fun main() {  val json = Json(JsonConfiguration.Default)  val jsonData = json.stringify(Data.serializer(), Data(42))  *println*(jsonData) *// {"a":42,"b":"42","list":[1,2,3,4,5]}*  var obj = json.parse(Data.serializer(), """{"a":42,"b":"75","list":[1,2,3,4,100]}""")  *println*(obj) *// Data(a=42, b="75")*  *println*(obj.list) *// [1, 2, 3, 4, 100]*  obj = json.parse(Data.serializer(), """{"a":42}""")  *println*(obj) *// Data(a=42, b="42")*  *println*(obj.list) *// [1, 2, 3, 4, 5]*  } |

**Полезные аннотации**

@SerialName

Используется для переопределения имени переменной

|  |
| --- |
| @Serializable  data class Names(  @SerialName("value1")  val custom1: String,  @SerialName("value2")  val custom2: Int  )  *println*(json.stringify(Names.serializer(), Names("a", 42)))  *// {"value1":"a","value2":42}* |

@Required

Свойство со значением по умолчанию обязано присутствовать в сериализованной форме.

|  |
| --- |
| @Serializable  data class Data(val a: Int, val b: String = "42") {  @Required  val list = *listOf*(1, 2, 3, 4, 5)  }  fun main() {  val json = Json(JsonConfiguration.Default)  var obj = json.parse(Data.serializer(), """{"a":42,"list":[1,2,3,4,100]}""")  *println*(obj) *// Data(a=42, b="42")*  *println*(obj.list) *// [1, 2, 3, 4, 100]*  *// MissingFieldException: Field 'list' is required, but it was missing*  obj = json.parse(Data.serializer(), """{"a":42}""")  } |

@Transient

Исключает переменную из процесса сериализации/десериализации

|  |
| --- |
| @Serializable  **data class** Data(**val a**: Int, **val b**: String = **"42"**) {  @Transient  **val list** = *listOf*(1, 2, 3, 4, 5)  }  **fun** main() {  **val** json = Json(JsonConfiguration.**Default**)  **val** jsonData = json.stringify(Data.serializer(), Data(42))  *println*(jsonData) *// {"a":42,"b":"42"}*  **var** obj = json.parse(Data.serializer(), **"""{"a":42}"""**)  *println*(obj) *// Data(a=42, b="42")*  *println*(obj.**list**) *// [1, 2, 3, 4, 5]*  *// JsonDecodingException: Unexpected JSON token at offset 16: Encountered an unknown key 'list'*  obj = json.parse(Data.serializer(), **"""{"a":42,"list":[1,2,3,4,100]}"""**)  } |