链滴

Rust 中的序列化和反序列化

作者：lingyundu

原文链接：https://ld246.com/article/1610956031716

来源网站：链滴

序列化： 将数据结构或对象转换成二进制序列的过程

反序列化：将在序列化过程中所生成的二进制序列转换成数据结构或者对象的过程

Serde 是对 Rust 数据结构进行序列化和反序列化一种常用框架。

Serde 中的 trait

Serde 定义了 4 种 trait：

- **Deserialize** A data structure that can be deserialized from any data format supported by Serde.
- **Deserializer** A data format that can deserialize any data structure supported by Serde.
- **Serialize** A data structure that can be serialized into any data format supported by Serde.
- **Serializer** A data format that can serialize any data structure supported by Serde.

Serialize 和 Deserialize

Serialize 和 Deserialize 的定义：

```
pub trait Serialize {
    pub fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer;
}
pub trait Deserialize<'de>: Sized {
    pub fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>;
}
```

若要数据类型支持序列化和反序列化，则该类型需要实现 Serialize 和 Deserialize trait。Serde 提供 Rust 基础类型和标准库类型的 Serialize 和 Deserialize 实现。

对于自定义类型，我们可以自行实现 Serialize 和 Deserialize trait。另外，Serde 提供一个宏 serde_derive 来自动为结构体类型和枚举类型生成 Serialize 和 Deserialize 实现。该特性需要 Rust 编译器本在 1.31及以上，并且在 Cargo.toml 文件中配置 Serde 依赖时，需要使用 features 指定该特性。如：

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
```

然后就可以在代码中引入并使用了，例如：

```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}
```

Serializer 和 Deserializer

Serializer 和 Deserializer 的实现由第三方 crate 提供，例如：serde_json, serde_yaml 和 bincode。

社区实现的部分数据格式：

- JSON, the ubiquitous JavaScript Object Notation used - by many HTTP APIs.
- Bincode, a compact binary format used for IPC within - the Servo rendering engine.
- CBOR, a Concise Binary Object Representation designed - for small message size without th need for version - negotiation.
- YAML, a self-proclaimed human-friendly configuration - language that ain't markup langua e.
- MessagePack, an efficient binary format that resembles - a compact JSON.
- TOML, a minimal configuration format used by Cargo.
- Pickle, a format common in the Python world.
- RON, a Rusty Object Notation.
- BSON, the data storage and network transfer format - used by MongoDB.
- Avro, a binary format used within Apache Hadoop, with - support for schema definition.
- JSON5, A superset of JSON including some productions - from ES5.
- Postcard, a no_std and embedded-systems friendly - compact binary format.
- URL query strings, in the x-www-form-urlencoded format.
- Envy, a way to deserialize environment variables into - Rust structs. (deserialization only)
- Envy Store, a way to deserialize AWS Parameter Store - parameters into Rust structs. (deseri lization only)
- S-expressions, the textual representation of code and - data used by the Lisp language fami y.
- D-Bus's binary wire format.
- FlexBuffers, the schemaless cousin of Google's - FlatBuffers zero-copy serialization format.
- DynamoDB Items, the format used by rusoto_dynamodb to - transfer data to and from Dyn moDB.

Serde JSON

在 Cargo.toml 文件中添加依赖：

```
[dependencies]
serde_json = "1.0"
```

序列化

serde_json Crate 提供了 serde_json::to_string、serde_json::to_vec 和 serde_json::to_writer 三函数将 Rust 数据类型转为 JSON String、Vec<u8> 和 io::Write（比如文件或者 TCP 流）。

示例：

```
use serde::{Deserialize, Serialize};
use serde_json::Result;

#[derive(Serialize, Deserialize)]
struct Address {
    street: String,
    city: String,
}

fn print_an_address() -> Result<()> {
    // Some data structure.
    let address = Address {
        street: "10 Downing Street".to_owned(),
        city: "London".to_owned(),
    };

    // Serialize it to a JSON string.
    let j = serde_json::to_string(&address)?;

    // Print, write to a file, or send to an HTTP server.
    println!("{}", j);

    Ok(())
}
```

反序列化

serde_json Crate 提供了 serde_json::from_str、serde_json::from_slice 和 serde_json::from_reader 三个函数将字符串、字节切片（&[u8]）和 IO 输入（文件或者 TCP 流）解析为对应的 Rust 数据类。

示例：

```
use serde::{Deserialize, Serialize};
use serde_json::Result;

#[derive(Serialize, Deserialize)]
struct Person {
    name: String,
    age: u8,
    phones: Vec<String>,
}

fn typed_example() -> Result<()> {
    // Some JSON input data as a &str. Maybe this comes from the user.
    let data = r#"
        {
            "name": "John Doe",
            "age": 43,
            "phones": [
                "+44 1234567",
                "+44 2345678"
            ]
        }"#;
```

```
    // Parse the string of data into a Person object. This is exactly the
    // same function as the one that produced serde_json::Value above, but
    // now we are asking it for a Person as output.
    let p: Person = serde_json::from_str(data)?;

    // Do things just like with any other Rust data structure.
    println!("Please call {} at the number {}", p.name, p.phones[0]);

    Ok(())
}
```

如果一个 JSON 格式数据没有对应的 Rust 数据类型，那么可以将其解析为 serde_json::Value 类型这是 serde_json Crate 提供的一个递归的枚举类型，它可以表示任何有效的 JSON 数据，其定义如：

```
enum Value {
    Null,
    Bool(bool),
    Number(Number),
    String(String),
    Array(Vec<Value>),
    Object(Map<String, Value>),
}
```

将 JSON 格式数据解析为 serde_json::Value 类型，同样使用的是：serde_json::from_str、serde_json::from_slice 和 serde_json::from_reader 三个函数。这三个函数返回值是泛型的，其返回值类型由定给变量的类型决定。

示例：

```
use serde_json::{Result, Value};

fn untyped_example() -> Result<()> {
    // Some JSON input data as a &str. Maybe this comes from the user.
    let data = r#"
        {
            "name": "John Doe",
            "age": 43,
            "phones": [
                "+44 1234567",
                "+44 2345678"
            ]
        }"#;

    // Parse the string of data into serde_json::Value.
    let v: Value = serde_json::from_str(data)?;
    // let v = serde_json::from_str::<Value>(data)?; // 或者可以这样写

    // Access parts of the data by indexing with square brackets.
    println!("Please call {} at the number {}", v["name"], v["phones"][0]);

    Ok(())
}
```

RON

在 Cargo.toml 文件中添加依赖：

```
[dependencies]
ron = "0.6.4"
```

RON(Rusty Object Notation) 是一种与 Rust 语法类似的一种序列化格式，它支持所有的 Serde 数模型。

RON 示例：

```
Scene( // class name is optional
    materials: { // this is a map
        "metal": (
            reflectivity: 1.0,
        ),
        "plastic": (
            reflectivity: 0.5,
        ),
    },
    entities: [ // this is an array
        (
            name: "hero",
            material: "metal",
        ),
        (
            name: "monster",
            material: "plastic",
        ),
    ],
)
```

RON 格式特点：

- 使用 (..) 表示结构体，使用 {..} 表示 Map，使用 [..] 表示数组。在使用 JSON 时是不能区分结构和 Map 的。
- 可以添加注释，在 JSON 中不能添加注释。
- 与 Rust 语法一样，最后一个分项也加逗号。

ron::value::Value

与 serde_json Crate 类似，ron Crate 也使用递归枚举类型来表示 RON 格式数据。这个枚举类型是：on::value::Value，它的定义如下：

```
pub enum Value {
    Bool(bool),
    Char(char),
    Map(Map),
    Number(Number),
    Option(Option<Box<Value>>),
    String(String),
```

```
    Seq(Vec<Value>),
    Unit,
}
```

序列化和反序列化

ron Crate 提供的序列化函数：

- ron::ser::to_string: Serializes value and returns it as string.
- ron::ser::to_string_pretty: Serializes value in the recommended RON layout in a pretty way.
- ron::ser::to_writer: Serializes value into writer.
- ron::ser::to_writer_pretty: Serializes value into writer in a pretty way.

ron Crate 提供的反序列化函数：

- ron::de::from_bytes: Building a deserializer and deserializing a value of type T from bytes.
- ron::de::from_str: Reading data from a reader and feeding into a deserializer.
- ron::de::from_reader: Building a deserializer and deserializing a value of type T from a string.

示例：

```
use serde::{Deserialize, Serialize};
use std::fs::File;

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point1 = Point { x: 1, y: 2 };
    // Serializes point1 into a buffer
    let mut buf = Vec::<u8>::new();
    ron::ser::to_writer(&mut buf, &point1).unwrap();
    println!("buf = {:?}", buf);

    let point2: Point = ron::de::from_bytes(&buf).unwrap();
    println!("point2: {:?}", point2);

    let point3 = Point { x: 3, y: 4 };
    // Serializes point3 and returns it as string
    let point_str = ron::to_string(&point3).unwrap();
    println!("point_str: {:?}", point_str);

    let point4: Point = ron::from_str(&point_str).unwrap();
    println!("point4: {:?}", point4);

    let point5 = Point { x: 5, y: 6 };
    let file = File::create("foo.txt").unwrap();
    // Serializes point5 into foo.txt
```

```
    ron::ser::to_writer(file, &point5).unwrap();

    let file = File::open("foo.txt").unwrap();
    let point6: Point = ron::de::from_reader(file).unwrap();
    println!("point6: {:?}", point6);
}
```

BSON

在 Cargo.toml 文件中添加依赖：

```
[dependencies]
bson = "1.1.0"
```

BSON(Binary JSON) 是一种二进制形式的类似 JSON 的格式（文档）。

BSON 示例：

```
// JSON equivalent
{"hello": "world"}

// BSON encoding
\x16\x00\x00\x00                  // total document size
\x02                              // 0x02 = type String
hello\x00                         // field name
\x06\x00\x00\x00world\x00         // field value
\x00                              // 0x00 = type EOO ('end of object')
```

BSON 值

很多不同的数据类型可以表示为 BSON 值，Bson 枚举类型定义了可表示为 BSON 值的所有类型。

示例：创建 Bson 实例

```
// 直接创建 Bson 实例
let string = Bson::String("hello world".to_string());
let int = Bson::Int32(5);
let array = Bson::Array(vec![Bson::Int32(5), Bson::Boolean(false)]);

// 使用 into() 创建 Bson 实例
let string: Bson = "hello world".into();
let int: Bson = 5i32.into();

// 使用 `bson!` 宏创建 Bson 实例
let string = bson!("hello world");
let int = bson!(5);
let array = bson!([5, false]);
```

BSON documents

Bson 中的文档由一组有序的键值对组成，类似于 JSON 中的 Object。另外，还包含 Bson 值所占的间大小。

示例一：创建文档实例

```
let mut bytes = hex::decode("0C000000106900010000000").unwrap();
// 直接创建
let doc = Document::from_reader(&mut bytes.as_slice()).unwrap(); // { "i": 1 }

// 通过 doc! 创建
let doc = doc! {
   "hello": "world",
   "int": 5,
   "subdoc": { "cat": true },
};
```

示例二：访问文档成员

```
let doc = doc! {
   "string": "string",
   "bool": true,
   "i32": 5,
   "doc": { "x": true },
};

// attempt get values as untyped Bson
let none = doc.get("asdfadsf"); // None
let value = doc.get("string"); // Some(&Bson::String("string"))

// attempt to get values with explicit typing
let string = doc.get_str("string"); // Ok("string")
let subdoc = doc.get_document("doc"); // Some(Document({ "x": true }))
let error = doc.get_i64("i32"); // Err(...)
```

序列化和反序列化

bson Crate 提供的序列化函数：

- bson::to_bson: Encode a T Serializable into a BSON Value.
- bson::to_document: Encode a T Serializable into a BSON Document.

bson Crate 提供的反序列化函数：

- bson::from_bson: Decode a BSON Value into a T Deserializable.
- bson::from_document: Decode a BSON Document into a T Deserializable.

示例：

```
use bson::*;
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize)]
struct Person {
    name: String,
    age: i32,
    phones: Vec<String>,
```

```
}

fn main() {
    // Some BSON input data as a `Bson`.
    let bson_data: Bson = bson!({
        "name": "John Doe",
        "age": 43,
        "phones": [
            "+44 1234567",
            "+44 2345678"
        ]
    });

    // Deserialize the Person struct from the BSON data.
    let person: Person = bson::from_bson(bson_data).unwrap();

    // Do things just like with any other Rust data structure.
    println!("Redacting {}'s record.", person.name);

    // Get a serialized version of the input data as a `Bson`.
    let redacted_bson = bson::to_bson(&person).unwrap();
    println!("{}", redacted_bson);
}
```

相关资料

Serde

JSON

RON

BSON