АНАЛИЗ ПРОБЛЕМЫ ПРЕОБРАЗОВАНИЯ ДАННЫХ ФОРМАТА JSON В СТРОГО ТИПИЗИРОВАННЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ GOLANG

Коптева А.В.[1], Князев И.В.[2]

[1]Коптева Анна Витальевна – старший разработчик программного обеспечения, Яндекс, г. Москва;
[2]Князев Илья Вадимович – старший разработчик программного обеспечения, June Homes, г. Белгород

Аннотация: *в статье анализируется парсинг данных в формате JSON на статически типизированном языке программирования, структура кода на Golang, использование и практические примеры.*
Ключевые слова: *json, parser, golang, javascript.*

Вступление.

JSON - это текстовое представление структурированных данных, основанное на парах ключ-значение и упорядоченных списках.

JSON формат широко применяется для передачи информации между веб-клиентом и веб-сервером, в том числе в мобильных приложениях и веб-сайтах. Хотя JSON является производным от JavaScript, он поддерживается либо изначально, либо через библиотеки на большинстве основных языков программирования. В этой статье рассматривается поддержка формата JSON в языке программирования Golang.

JSON и статическая типизация

Парсинг данных в формате JSON на строго типизированном языке таком как Golang имеет особенности. Самой большой проблемой является то, что компилятор должен заранее знать, какие типы содержит JSON, чтобы выделить память для объектов.

Существует 2 способа решения данной проблемы. Самый простой - это проанализировать JSON в определенную структуру, если типы данных заранее известны. Любое поле, не описанное в структуре, игнорируется. Начну с описания данного способа.

Парсинг данных в структуру (анг Struct)

Для работы с json golang имеет встроенную библиотеку “encoding/json”. Код парсинга выглядит следующим образом:

```
type Book struct {
   Id string     `json:"id"`
   Title string  `json:"title"`
}

data := []byte(`
   {
      "id": "12345",
      "title": "War and Peace"
   }
`)

var book Book
err := json.Unmarshal(data, &book)
```

Для синтаксического анализа json в Go используется метод Unmarshal(). После выполнения кода получаем заполненную пользовательскую структуру Book.

Преобразование структуры (анг Struct)

Преобразование структуры работает точно так же, как синтаксический анализ, но в обратном порядке:

```
data, err := json.Marshal(app)
```

Как и в случае со всеми структурами в Go, важно помнить, что только поля с заглавной первой буквой видны внешним программам, таким как JSON Marshaller.

Теги для структур (анг Struct)

Для того, чтобы указать парсеру, как анализировать определённое поле используются теги - это «помеченные» данные, включенные в структуру между обратными кавычками.

Название поля

Go требует, чтобы все экспортируемые поля начинались с заглавной буквы. Однако в JSON не принято использовать этот стиль. Для этого используется тег, чтобы синтаксический анализатор знал, где на самом деле искать значение.

В коде это выглядит следующим образом:

```
type Author struct {
   Name string `json: "name"`
}
```

Пустое поле

Парсер JSON также принимает флаг в теге, чтобы он знал, что делать, если поле пустое. Флаг omitempty указывает парсеру не включать значение JSON в вывод, если это является нулевым значением для этого типа.

Нулевым значением для чисел является 0, для строк - это пустая строка, для словарей (*анг* map), срезов (*анг* Slice) и указателей - nil. Пример, как включать флаг omitempty:

```
type Author struct {
   Name string `json: "name, omitempty"`
}
```

Стоит обратить внимание, что флаг находится внутри кавычек.

В случае если бы *Name* был пустой строкой, при преобразовании ее в JSON, *name* не было бы включено в вывод.

Другими словами, если *name* == "":

- С omitempty значение JSON будет {}
- Без opitempty значение JSON будет {"name": ""}

Флаг omitempty полезен, чтобы пометить поле как устаревшее и больше не включать его в вывод.

Пропуск поля

Чтобы синтаксический анализатор JSON пропустил поле, необходимо дать ему имя «-». Например:

```
type Book struct {
   Id string     `json:"id"`
   Title string  `json:"-"`
}
```

Это иметь смысл для полей, которые будут анализироваться, если они доступны, но никогда не будут выводиться.

Вложенные поля

Вложенные поля относятся к структурам, которые являются свойствами других структур. Парсер JSON будет рекурсивно анализировать вложенное поле любого типа, например словарь (*анг* map), срез (*анг* Slice) или другую структуру. Для поля, которое может быть любым типом, можно использовать тип интерфейса (interface{}).

Go также поддерживает вложение одной структуры в другую. Например:

```
type Book struct {
   Id string `json:"id"`
}

type Author struct {
   Name string `json:"name"`
}

type BookInfo struct {
   Book
   Author
}
```

Для примера выше можно проанализировать значение с помощью типа *BookInfo*, и оно будет иметь все свойства. Также можно получить из него вложенные структуры .*Book* или .*Author*. Например:

```
data := []byte(`
   {
      "id": "12345",
      "title": "War and Peace"
   }
`)

var bookInfo BookInfo
err := json.Unmarshal(data, &bookInfo)

book := bookInfo.Book
author := bookInfo.Author

// AND/OR

bookId := bookInfo.Id
authorName := bookInfo.Name
```

Объединение подобных структур очень ценно для случаев, когда есть API, включающий некоторые дополнительные данные с фактическим значением.

Обработка ошибок

Необходимо всегда проверять параметр err, возвращаемый Marshal и Unmarshal. С помощью этого параметра можно узнать, есть ли ошибка в синтаксисе анализируемого JSON. Если вы его не отметите, ваша программа продолжит выполнение с обнуленной структурой, которую вы уже создали, что может вызвать запутанное поведение ниже по течению. Если пропустить проверку, то программа продолжит выполнение с нулевой (*анг* nil) структурой, что приведёт к неконсистентности данных.

Ошибки во время преобразования структуры встречаются реже, но они могут возникнуть, если Go не может понять, как преобразовать один из типов в JSON. Например, если преобразовывать что-то, содержащее нулевой указатель.

Чтобы не обрабатывать ошибки при каждом преобразовании, можно преобразовать ошибки в панику (*анг* panic) с помощью функции MustMarshal:

```
func MustMarshal(data interface{}) []byte {
   out, err := json.Marshal(data)
   if err != nil {
      panic(err)
   }

   return out
}
```

Парсинг данных в интерфейс

Если тип данных в JSON заранее не известен, то можно получить общий интерфейс (*анг* interface{}) и проанализировать его. Общий интерфейс (interface{}) - это способ определения переменной в Go любых типов данных. Затем во время выполнения Golang выделит соответствующую память, для хранения всех данных.

Вот как это выглядит:

```
var parsed interface{}
err := json.Unmarshal(data, &parsed)
```

На самом деле использование синтаксического анализа трудоемко, так как Go не может использовать его, не зная конкретный тип. Код парсинга выглядит следующим образом:

```
switch parsed.(type) {
   case int:
      someGreatIntFunction(parsed.(int))
   case map:
      someMapThing(parsed.(map))
   default:
      panic("JSON type not understood")
```

```
}
```

Если, например, известно, что значение JSON является объектом, то можно распарсить его в map[string]interface{}. Это дает возможность ссылаться на определенные ключи. Пример:

```
var parsed map[string]interface{}

data := []byte(`
   {
      "id": "12345",
      "age": 29
   }
`)

err := json.Unmarshal(data, &parsed)
```

После такого преобразования можно обращаться к конкретным ключам:

```
parsed["id"]
```

Однако значения словаря по-прежнему содержат в себе интерфейс, поэтому необходимо выполнять преобразование типа, чтобы использовать их:

```
idString := parsed["id"].(string)
```

Go использует шесть типов для всех значений, анализируемых в интерфейсах:

- bool, для логических значений JSON
- float64, для чисел JSON
- string, для строк JSON
- []interface {}, для массивов JSON
- map[string]interface{}, для объектов JSON
- nil, для JSON null

Это означает, что числа всегда будут иметь тип float64, и, например, их нужно будет преобразовывать в int. Для того, чтобы напрямую получить целые числа, необходимо использовать метод UseNumber(). Этот метод возвращает объект, который можно преобразовать в float64 или int.

Точно так же все объекты, преобразованные в интерфейс, будут иметь тип map[string]interface{}, и их необходимо вручную сопоставить с целевой структурой.

Парсинг чисел

Выше говорилось о проблеме с преобразованием чисел javascript в значения Golang при использовании интерфейсов. Javascript имеет только один числовой тип, поэтому Golang всегда должен использовать float64, если явно не запрашивать целое число. Еще один важный момент, который следует понимать: поскольку Javascript имеет только 64-битные числа с плавающей запятой, невозможно точно представить целое число, превышающее 53 бит. Для того, чтобы передать клиенту 64-битное целое число, Go 1.3 добавил возможность передавать его как строку:

```
type MyStruct struct {
   Int64String int64 `json:",string"`
}
```

Создание кодируемых типов

Модуль JSON включает два интерфейса. Marshaler и Unmarshaler.

Для обоих интерфейсов требуется только один метод. При добавлении этих двух методов в свой тип, он будет кодироваться как JSON. Прекрасным примером является тип time.Time.

Другой пример:

```
type Month struct {
   MonthNumber int
   YearNumber int
}
```

```
func (m Month) MarshalJSON() ([]byte, error){
   return []byte(fmt.Sprintf("%d/%d", m.MonthNumber, m.YearNumber)), nil
}

func (m *Month) UnmarshalJSON(value []byte) error {
   parts := strings.Split(string(value), "/")
   m.MonthNumber = strconv.ParseInt(parts[0], 10, 32)
   m.YearNumber = strconv.ParseInt(parts[1], 10, 32)

   return nil
}
```

Go решает, какой Marshaler и Unmarshaler использовать в зависимости от типов, которые кодируются или декодируются. При кодировании или декодировании интерфейса, Go не может определить, какой маршаллер использовать, и по умолчанию он будет использовать шесть типов, перечисленных выше.

Заключение

Из-за строгой типизации в Golang работа с JSON форматом данных имеет ряд трудностей. В том числе преобразование сложных типов или чисел. Но встроенные инструменты Golang, которые были рассмотрены в данной статье, максимально упрощают преобразование различных типов данных. Стоит заметить, что Golang довольно молодой язык программирования и его создатели проделали огромную работу для улучшения инструментов парсинга данных.

Список литературы

1. Документация Golang / [Электронный ресурс], 2021. Режим доступа: https://pkg.go.dev/encoding/json/ (дата обращения: 04.09.2021).
2. Документация JSON. [Электронный ресурс], 2021. Режим доступа: https://www.json.org/json-en.html/ (дата обращения: 02.09.2021).
3. *Цукалос М.* Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go. М.: Прогресс книга, 2021. 720 с.
4. *Доновaн Алан А.А., Керниган Брайан У.* Язык программирования Go. М.: Вильямс, 2018. 432 с.
5. *Батчер М., Фарина М.* Go на практике. М.: ДМК Пресс, 2017. 376 с.