

Extracted from:

Effective Go Recipes

Fast Solutions to Common Tasks

This PDF file contains pages extracted from *Effective Go Recipes*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The
Pragmatic
Programmers

Effective Go Recipes

Fast Solutions to Common Tasks



Miki Tebeka
edited by Margaret Eldridge

Effective Go Recipes

Fast Solutions to Common Tasks

Miki Tebeka

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit <https://pragprog.com>.

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-846-8

Encoded using the finest acid-free high-entropy binary digits.

Book version: B1.0—August 16, 2023

Parsing Struct Tags

Task

You're building an ORM (object-relational mapper) for your company's proprietary database. An ORM simplifies working with the database; you store and retrieve Go structs directly instead of using []any.

Your first task is to parse a user struct and return the mapping between the field names and the database columns. For example for the following struct:

```
serialize/orm/orm.go
// Log is a log structure
type Log struct {
    Time time.Time `db:"ts"`
    Level int `db:"level"`
    Text string `db:"message"`
}
```

You should return the mapping of:

- Time → ts
- Level → level
- Text → message

Solution

You start by defining the parsing function signature:

```
serialize/orm/orm.go
func parseStructTags(s any) (map[string]string, error) {
```

First you use the reflect package to get the type of the parameter:

```
serialize/orm/orm.go
typ := reflect.TypeOf(s)
if typ.Kind() != reflect.Struct {
    return nil, fmt.Errorf("%s is not a struct", typ)
}
```

Then you iterate of the struct fields and extract the db key from the field tag:

```
serialize/orm/orm.go
m := make(map[string]string)
for i := 0; i < typ.NumField(); i++ {
```

```

    fld := typ.Field(i)
    if dbName := fld.Tag.Get("db"); dbName != "" {
        m[fld.Name] = dbName
    }
}
return m, nil

```

Discussion

Since `parseStructTags` should accept any value, the type of its parameter is `any`. Most of the time, using `any` is a “code smell” and you should use concrete types or an interface.

However, in this case, you can’t use a concrete type or an interface, so you’re left with using `any`.

Note: The term “code smell” was coined by Kent Beck and means a surface indication that usually corresponds to a deeper problem in the system.

Struct tags are used by many serialization packages; the built-in `encoding/json`, `encoding/xml`, and many other external packages such as `yaml` use them.

Struct tags allow you to add extra information about a field. They have a known format: `key:"value" key:"value" ...`. The `Field` type in the `reflect` package has methods to extract a specific key.

In some cases, the value can be more than just a name. Commonly it’s either a list of values separated by a `,` or a `name=value`. Here’s an example from a struct generated by Google’s `protoc` tool:

```
Value float64 `protobuf:"fixed64,1,opt,name=value,proto3" json:"value,omitempty"``
```

The struct tag has two keys: `protobuf` and `json` and both have complex values.

If you decide you need complex values (like `protobuf` in our example), you need design the format used in these tags since you’re basically implementing your own serialization format.

It’s best to copy what `encoding/json` or `protobuf` are doing instead of inventing your own.