

用 Go 语言编写一门工具的终极指南

原创 2017-10-31 OSC- 协作翻译 开源中国


协作翻译

原文：The ultimate guide to writing a Go tool

链接：<https://arslan.io/2017/09/14/the-ultimate-guide-to-writing-a-go-tool/>

译者：Tocy, 亚林瓜子, soaring

我以前构建过一个工具，以让生活更轻松。这个工具被称为：**gomodifytags**，它会根据字段名称自动填充结构体的标签字段。示例如下：



```
1 package main
2
3 type Example struct {
4     StatusID int64
5     Foo      string
6     Bar      bool
7
8     Server struct {
9         Address string
10        TLS      bool
11    }
12
13    DiskSize int64
14    Volumes  []string
15 }
NORMAL demo.go go 46% 7:1
:GoA_
```

（在 vim-go 中使用 gomodifytags 的一个用法示例）

使用这样的工具可以**轻松管理**结构体的多个字段。该工具还可以添加和删除标签，管理标签选项（如 omitempty），定义转换规则（snake_case、camelCase 等）等等。但是这个工具是如何工作的？在后台它究竟使用了哪些 Go 包？有很多这样的问题需要回答。

这是一篇非常长的博客文章，解释了如何编写类似这样的工具以及如何构建它的每一个细节。它包含许多特有的细节、提示和技巧和某些未知的 Go 位。

拿一杯咖啡，开始深入探究吧！

首先，列出这个工具需要完成的功能：


1. 它需要读取源文件，理解并能够解析 Go 文件
2. 它需要找到相关的结构体
3. 找到结构体后，需要获取其字段名称
4. 它需要根据字段名更新结构标签（根据转换规则，即：snake_case）

5. 它需要能够使用这些改动来更新文件，或者能够以可接受的方式输出改动

我们首先来看看**结构体标签的定义**是什么，之后我们会学习所有的部分，以及它们如何组合在一起，从而构建这个工具。

So, what's a **struct tag**?

```
type Example struct {  
    Foo string `json:"foo"`  
}
```



结构体的**标签值** (其内容，比如`json:"foo"`) **并不是官方标准的一部分**，不过，存在一个非官方的规范，使用 `reflect` 包定义了其格式，这种方法也被 `stdlib` (例如 `encoding/json`) 包所采用。它是通过 `reflect.StructTag` 类型定义的：

reflect.StructTag

type StructTag

A StructTag is the tag string in a struct field.

By convention, tag strings are a concatenation of optionally space-separated key:"value" pairs. Each key is a non-empty string consisting of non-control characters other than space (U+0020 ' '), quote (U+0022 '"'), and colon (U+003A ':'). Each value is quoted using U+0022 '"' characters and Go string literal syntax.

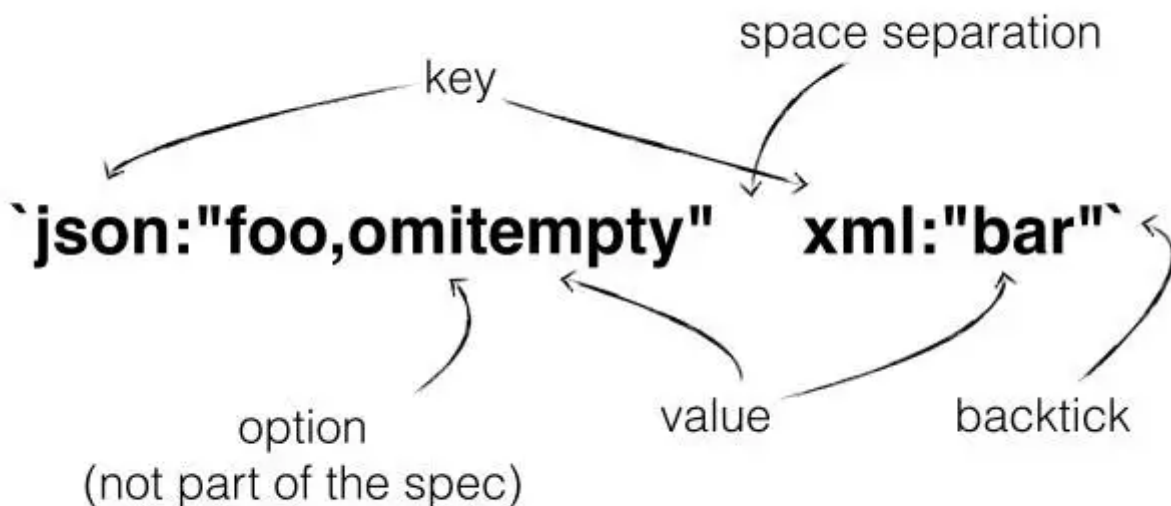
```
type StructTag string
```

结构标签的定义比较简洁所以不容易理解。该定义可以分解如下：

- 结构标签是一个字符串(字符串类型)
- 结构标签的 Key 是非引号字符串

- 结构标签的 value 是一个带引号的字符串
- 结构标签的 key 和 value 用冒号(:)分隔。冒号隔开的一个 key 和对应的 value 称为 “key value 对” 。
- 一个结构标签可以包含多个 key value 对(可选)。key-value 对之间用空格隔开。
- 可选设置不属于定义的一部分。类似 encoding/json 包将 value 解析为逗号分开的列表。value 的第一个逗号后面的任何部分都是可选设置的一部分，例如： **“foo,omitempty,string”** 。其中 value 拥有一个叫 “foo” 的名字和可选设置 [“omitempty” , "string"]
- 由于结构标签是一个字符串，需要双引号或者反引号包含。又因为 value 也需要引号包含，经常用反引号包含结构标签。

以上规则概况如下：



(结构标签的定义有许多隐含细节)

已经了解什么是结构标签，接下来可以根据需要修改结构标签。问题来了，如何才能很容易的对所做的修改进行解析？很幸运，`reflect.StructTag` 包含一个可以解析结构标签并返回特定 key 的 value 的方法。示例如下：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    tag := reflect.StructTag(`species:"gopher" color:"blue"`)
    fmt.Println(tag.Get("color"), tag.Get("species"))
}
```

输出：

```
blue gopher
```

如果 key 不存在则返回空串。

这是非常有帮助的，**但是**，它有一些附加说明，使其不适合我们，因为我们需要更多的灵活性。这些是：

- 它无法检测到标签是否存在**格式错误**（即：键被引用了，值是未引用等）
- 它不知道选项的**语义**
- 它没有办法**迭代现有的标签**或返回它们。我们必须知道我们要修改哪些标签。如果不知道其名字怎么办？
- 修改现有标签是不可能的。
- 我们不能重新**构建新的struct标签**。

为了改进这一点，我编写了一个自定义的Go包，它修复了上面的所有问题，并提供了一个可以轻松修改struct标签的每个方面的API。

structtag

Index

type Tag

- func (t *Tag) GoString() string
- func (t *Tag) HasOption(opt string) bool
- func (t *Tag) String() string

type Tags

- func Parse(tag string) (*Tags, error)
- func (t *Tags) AddOptions(key string, options ...string)
- func (t *Tags) Delete(keys ...string)
- func (t *Tags) DeleteOptions(key string, options ...string)
- func (t *Tags) Get(key string) (*Tag, error)
- func (t *Tags) Keys() []string
- func (t *Tags) Len() int
- func (t *Tags) Less(i int, j int) bool
- func (t *Tags) Set(tag *Tag) error
- func (t *Tags) String() string
- func (t *Tags) Swap(i int, j int)
- func (t *Tags) Tags() []*Tag

这个包被称为**structtag**，并且可以从github.com/fatih/structtag获取到。这个包允许我们以一种整洁的方式**解析和修改标签**。以下是一个完整的可工作的示例，复制/粘贴并自行尝试下：

```

package main

import (
    "fmt"

    "github.com/fatih/structtag"
)

func main() {
    tag := `json:"foo,omitempty,string" xml:"foo"`

    // parse the tag
    tags, err := structtag.Parse(string(tag))
    if err != nil {
        panic(err)
    }

    // iterate over all tags
    for _, t := range tags.Tags() {
        fmt.Printf("tag: %+v\n", t)
    }

    // get a single tag
    jsonTag, err := tags.Get("json")
    if err != nil {
        panic(err)
    }

    // change existing tag
    jsonTag.Name = "foo_bar"
    jsonTag.Options = nil
    tags.Set(jsonTag)

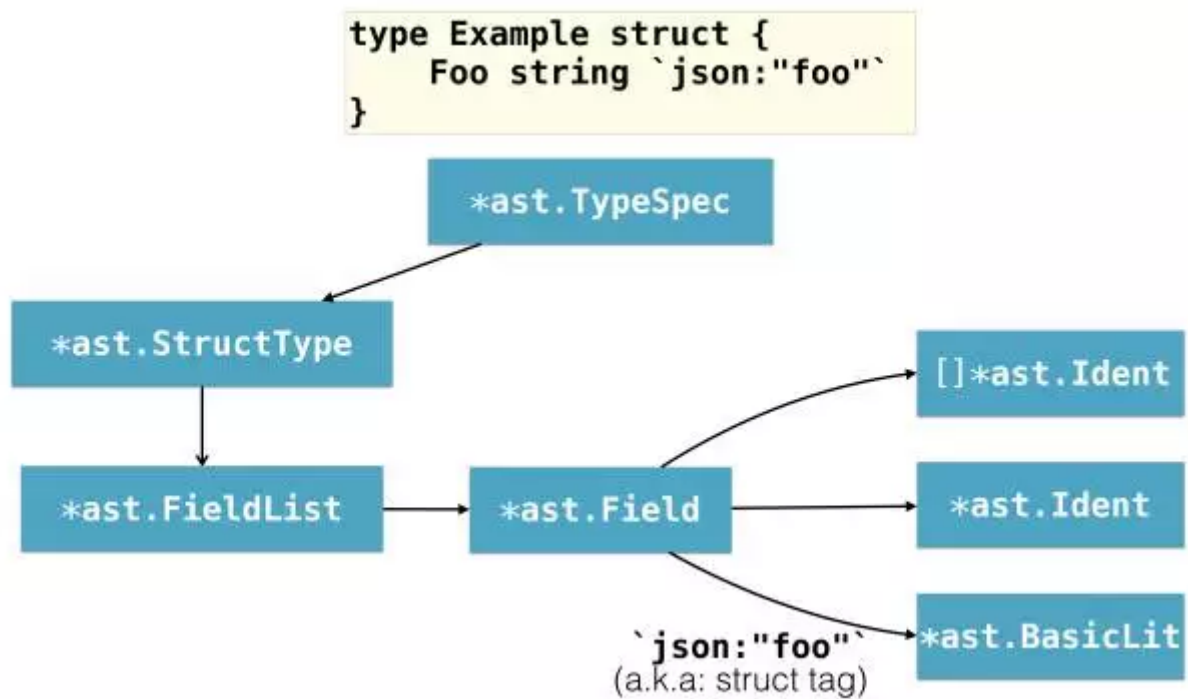
    // add new tag
    tags.Set(&structtag.Tag{
        Key:      "hcl",
        Name:     "foo",
        Options: []string{"squash"},
    })

    // print the tags
    fmt.Println(tags) // Output: json:"foo_bar" xml:"foo" hcl:"foo,squash"
}

```

既然我们已经知道如何解析一个struct标签了，以及修改它或创建一个新的，现在是时候来修改一个有效的Go源文件了。在上面的示例中，标签已经存在了，但是如何从现有的Go结构中获取标签呢？

简要回答：通过**AST**。AST（Abstract Syntax Tree，抽象语法树）允许我们从源代码中检索每个单独的标识符（node）。下图中你可以看到一个结构类型的AST（简化版）：



（结构体的基本的Go **ast.Node**表示）

在这棵树中，我们可以检索和操纵每个标识符，每个字符串和每个括号等。这些都由AST节点表示。例如，我们可以通过替换表示它的节点中的名字将字段名称从“Foo”更改为“Bar”。相同的逻辑也适用于struct标签。

要**得到Go AST**，我们需要解析源文件并将其转换为AST。实际上，这两者都是通过一个步骤处理的。

要做到这一点，我们将使用go/parser包来**解析**文件以获取（整个文件的）AST，然后使用go/ast包来遍历整棵树（我们也可以手动执行，但这是另一篇博文的主题）。下面代码你可以看到一个完整的例子：

```

package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
)

func main() {
    src := `package main
    type Example struct {
        Foo string ` + " `json:\"foo\"`" `

    fset := token.NewFileSet()
    file, err := parser.ParseFile(fset, "demo", src, parser.ParseComments)
    if err != nil {
        panic(err)
    }

    ast.Inspect(file, func(x ast.Node) bool {
        s, ok := x.(*ast.StructType)
        if !ok {
            return true
        }

        for _, field := range s.Fields.List {
            fmt.Printf("Field: %s\n", field.Names[0].Name)
            fmt.Printf("Tag:   %s\n", field.Tag.Value)
        }
        return false
    })
}

```

上面代码输出如下:

```

Field: Foo
Tag:   `json:"foo"`

```

上面代码执行以下操作：

- 我们定义了仅包含一个结构体的有效Go包的实例。
- 我们使用`go/parser`包来解析这个字符串。解析器包也可以从磁盘读取文件（或整个包）。
- 在我们解析之后，我们保存我们的节点（分配给变量文件）并查找由`*ast.StructType`定义的AST节点（参见AST映像作为参考）。遍历树是通过`ast.Inspect()`函数完成的。它会遍历所有节点，直到它收到`false`值。这是非常方便的，因为它不需要知道每个节点。
- 我们打印结构体的字段名称和结构标签。

我们现在可以完成两件重要的事情了，首先，我们知道如何解析一个 Go 源文件并检索其中结构体的标签（通过`go/parser`）。其次，我们知道如何解析 Go 结构体标签，并根据需要进行修改（通过 github.com/fatih/structtag）。

既然我们有了这些，我们可以通过使用这两个重要的代码片段开始构建我们的工具（名为 **gomodifytags**）。该工具应顺序执行以下操作：

1. 获取配置，以识别我们要修改哪个结构体
2. 根据配置查找和修改结构体
3. 输出结果

由于 **gomodifytags** 将主要由编辑器来执行，我们打算通过 CLI 标志传递配置信息。第二步包含多个步骤，如解析文件、找到正确的结构体，然后修改结构（通过修改 AST 完成）。最后，我们将输出结果，或是按照原始的 Go 源文件或是某种自定义协议（如 JSON，稍后再说）。

以下是 gomodifytags 简化之后的主要功能：

1. **Fetch** configuration settings
2. **Parse** content
3. **Find** selection
4. **Modify** the struct tag
5. **Output** the result

```
func main() {  
    var cfg config  
  
    node = cfg.parse()  
  
    start, end = cfg.findSelection(node)  
  
    rewritten = cfg.rewrite(node, start, end)  
  
    out = cfg.format(rewritten)  
  
    fmt.Println(out)  
}
```

让我们开始详细解释每个步骤。为了保持简单，我将尝试以萃取形式解释重要的部分。尽管一切都是一样的，一旦你读完了这篇博文，你将能够在无需任何指导的情况下通读整个源代码（你将会在本指南的最后找到所有资源）

让我们从第一步开始，了解如何**获取配置**。以下是我们的配置文件，其中包含所有的必要信息。


```

type config struct {
    // first section - input & output
    file      string
    modified io.Reader
    output    string
    write     bool

    // second section - struct selection
    offset    int
    structName string
    line      string
    start, end int

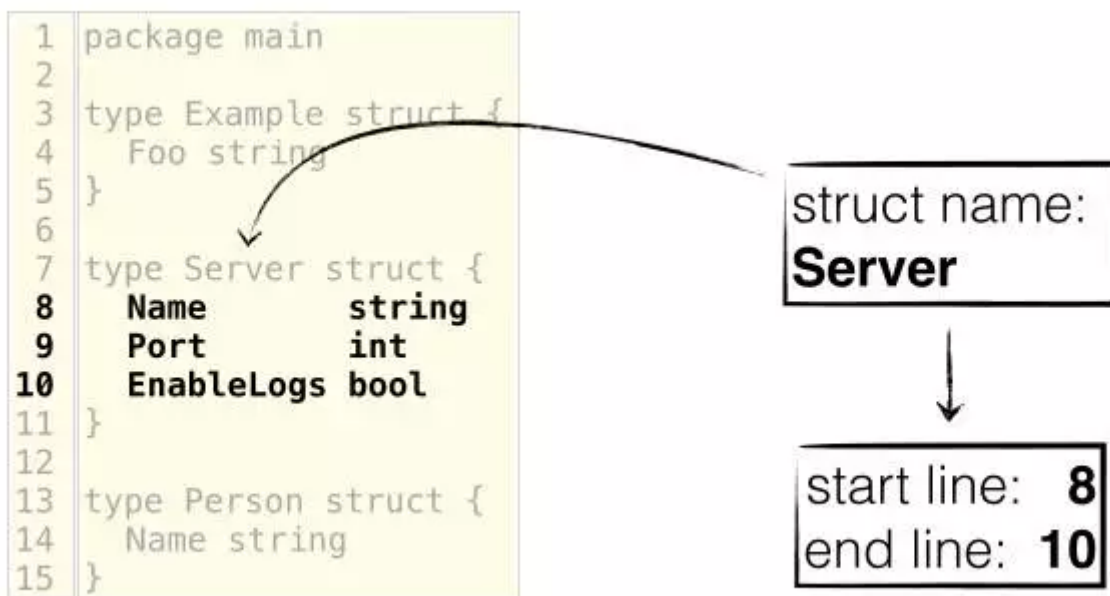
    // third section - struct modification
    remove []string
    add     []string
    override bool
    transform string
    sort    bool
    clear   bool
    addOpts []string
    removeOpts []string
    clearOpt bool
}

```

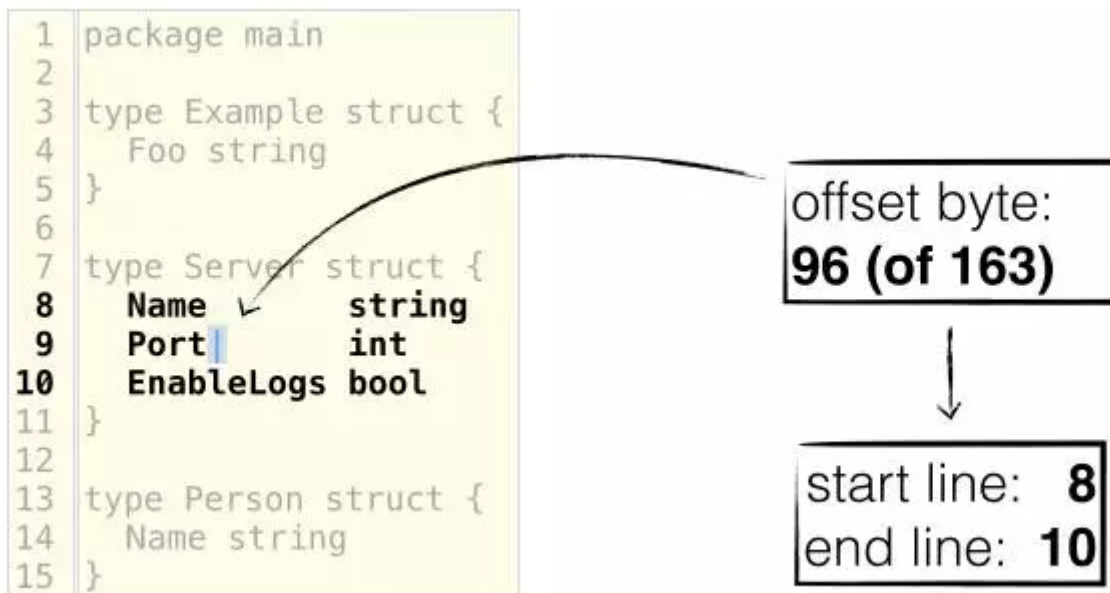
它分为三个主要部分：

第一部分包含有关如何和哪个文件要读入的配置。这可以是本地文件系统的文件名，也可以是直接来自stdin的数据（主要用在编辑器中）。它还设置了如何输出结果（Go源文件或JSON形式），以及我们是否应该覆写文件，而不是输出到stdout中。

第二部分定义了如何选择一个结构体及其字段。有多种方法可以做到这一点。我们可以通过它的偏移（光标位置）、结构名称，单行（仅指定字段）或一系列行来定义它。最后，我们总是需要得到起始行号。例如在下面的例子中，你可以看到一个例子，我们用它的名字来选择结构体，然后提取起始行号，以便我们可以选择正确的字段：



而编辑器最好使用**字节偏移量**。例如下面你可以看到我们的光标刚好在“Port”字段名称之后，从那里我们可以很容易地得到起始行号：



config配置中的**第三**部分实际上是一个到我们的structtagpackage的一对一的映射。它基本上允许我们在读取字段后将配置传递给structtag包。如你所知，structtag包允许我们解析一个struct标签并在各个部分进行修改。但是，它不会覆写或更新结构体的域值。

我们该如何获得配置呢？我们只需使用flag包，然后为配置中的每个字段创建一个标志，然后给他们赋值。举个例子：

```
flagFile := flag.String("file", "", "Filename to be parsed")
cfg := &config{
    file: *flagFile,
}
```

我们对配置中的每个字段执行相同操作。相关完整的列表请查看gomodifytag的当前master分支上的flag定义。

一旦我们有了配置，我们就可以做一些基本的验证了：

```

func main() {
    cfg := config{ ... }

    err := cfg.validate()
    if err != nil {
        log.Fatalln(err)
    }

    // continue parsing
}

// validate validates whether the config is valid or not
func (c *config) validate() error {
    if c.file == "" {
        return errors.New("no file is passed")
    }

    if c.line == "" && c.offset == 0 && c.structName == "" {
        return errors.New("-line, -offset or -struct is not passed")
    }

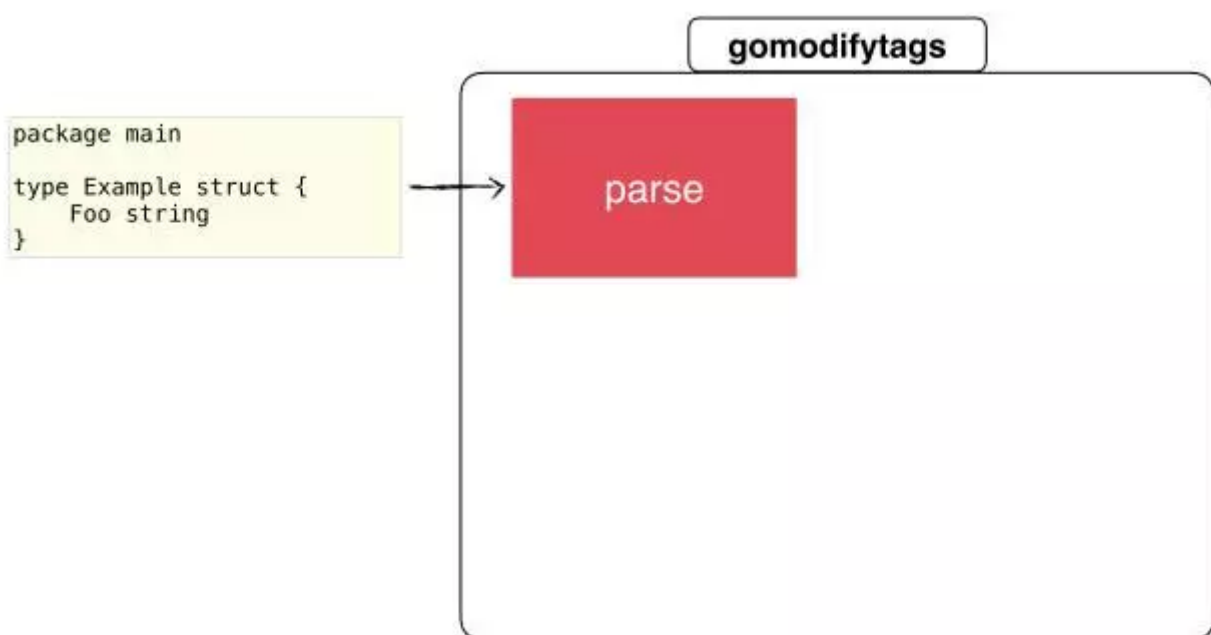
    if c.line != "" && c.offset != 0 ||
        c.line != "" && c.structName != "" ||
        c.offset != 0 && c.structName != "" {
        return errors.New("-line, -offset or -struct cannot be used together. pick one")
    }

    if (c.add == nil || len(c.add) == 0) &&
        (c.addOptions == nil || len(c.addOptions) == 0) &&
        !c.clear &&
        !c.clearOption &&
        (c.removeOptions == nil || len(c.removeOptions) == 0) &&
        (c.remove == nil || len(c.remove) == 0) {
        return errors.New("one of " +
            "[-add-tags, -add-options, -remove-tags, -remove-options, -clear-tags, -clear-options]" +
            " should be defined")
    }

    return nil
}

```

将验证部分代码放到一个单一的函数中，使得测试测试更简单。既然我们已经知道如何获取配置并进行验证，我们继续去解析文件：



我们在一开始就讨论了如何解析一个文件。这里解析的是config结构体中的方法。实际上，所有的方法都是config结构体的一部分：

```
func main() {
    cfg := config{}

    node, err := cfg.parse()
    if err != nil {
        return err
    }

    // continue find struct selection ...
}

func (c *config) parse() (ast.Node, error) {
    c.fset = token.NewFileSet()
    var contents interface{}
    if c.modified != nil {
        archive, err := buildutil.ParseOverlayArchive(c.modified)
        if err != nil {
            return nil, fmt.Errorf("failed to parse -modified archive: %v", err)
        }
        fc, ok := archive[c.file]
        if !ok {
            return nil, fmt.Errorf("couldn't find %s in archive", c.file)
        }
        contents = fc
    }

    return parser.ParseFile(c.fset, c.file, contents, parser.ParseComments)
}
```

解析函数只完成了一件事。解析源码并返回一个ast.Node。如果我们仅传递文件，这是非常简单的，在这种情况下，我们使用parser.ParseFile()函数。需要注意的是token.NewFileSet()，它创建一个类型为*token.FileSet。我们将它存储在c.fset中，但也传递给parser.ParseFile()函数。为什么呢？

因为fileset用于独立地为每个文件存储每个节点的位置信息。这将在以后对于获得ast.Node的确切信息非常有帮助（请注意，ast.Node使用一个紧凑的位置信息，称为token.Pos。要获取更多的信息，它需要通过token.FileSet.Position()函数来获取一个token.Position，其中包含更多的信息）

让我们继续。如果通过 stdin 传递源文件，它会变得更加有趣。config.modified 字段是易于测试的io.Reader，但实际上我们通过 stdin 传递它。我们如何检测是否需要从 stdin 读取呢？

我们询问用户是否想通过 stdin 传递内容。在这种情况下，本工具的用户需要传递--modified 标志（这是一个布尔标志）。如果用户传递了该标志，我们只需将 stdin 分配给 c.modified 即可：

```
flagModified = flag.Bool("modified", false,
    "read an archive of modified files from standard input")

if *flagModified {
    cfg.modified = os.Stdin
}
```

如果你再次检查上面的 config.parse() 函数，你将看到我们检查 .modified 字段是否已分配，因为 stdin 是一个任意数据的流，我们需要能够根据给定的协议对其进行解析。在这种情况下，我们假定其中包含以下内容：

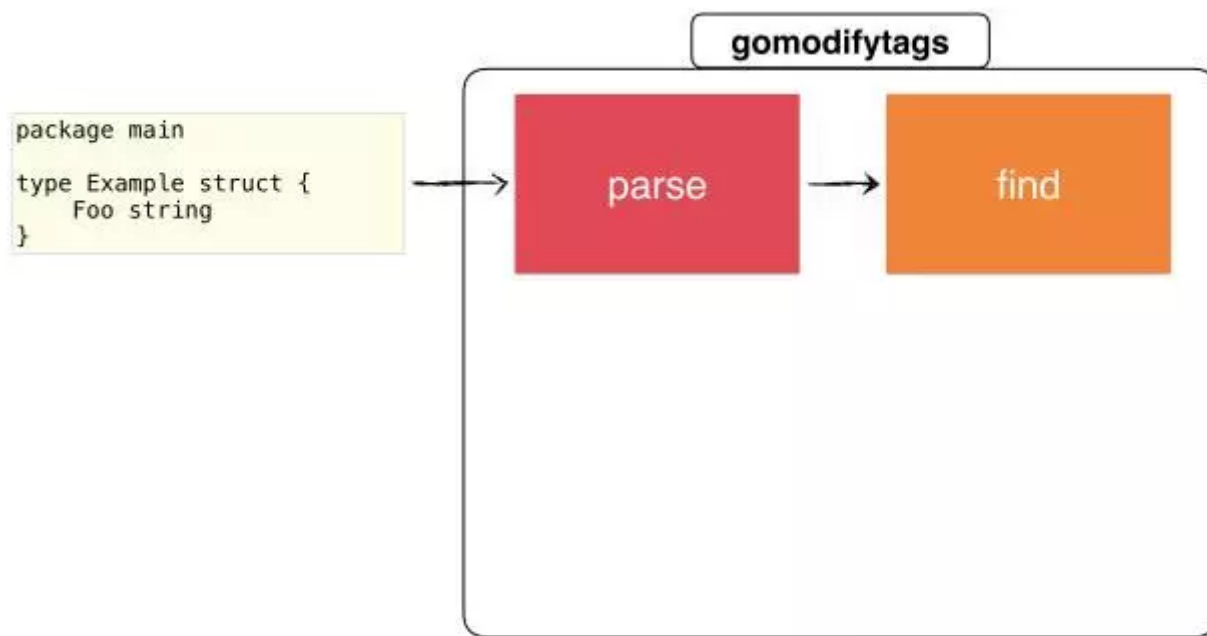
- 文件名，后跟换行符
- （十进制）文件大小，后跟换行符

- 文件的内容

因为我们知道文件大小，我们可以毫无问题地解析此文件的内容。任何大于给定文件大小的部分，我们仅需停止解析。

这种方法也被其他几种工具所使用（如 guru、gogetdoc 等），并且它对编辑器来说是非常有用的。因为这样可以让编辑器传递修改后的文件内容，并且无需保存到文件系统中。因此它被命名为“modified”。

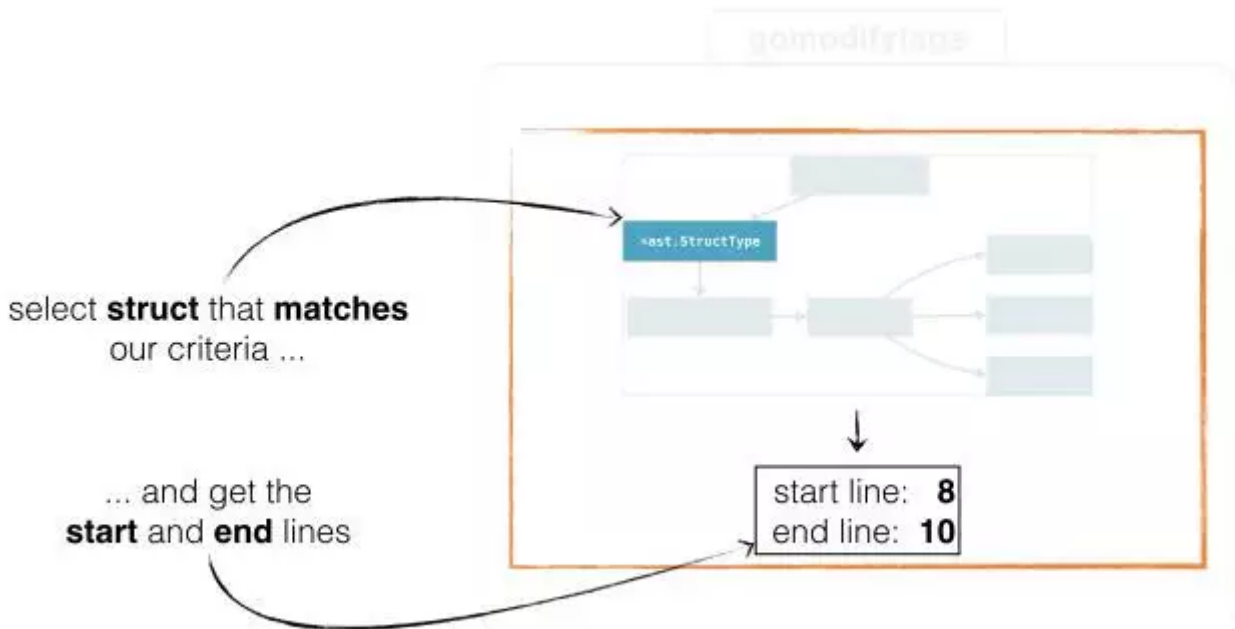
既然我们已经拥有了 Node，让我们继续下一步的“查找结构体”：



我们的主函数中，我们将使用在上一步中解析的 `ast.Node` 中调用 `findSelection()` 函数：

```
func main() {  
    // ... parse file and get ast.Node  
  
    start, end, err := cfg.findSelection(node)  
    if err != nil {  
        return err  
    }  
  
    // continue rewriting the node with the start&end position  
}
```

`cfg.findSelection()` 函数会根据配置文件和我们选定结构体的方式来返回指定结构体的开始和结束位置。它在给定 Node 上进行迭代，然后返回其起始位置（和以上的配置一节中的解释类似）：



(检索步骤会迭代所有 node ，直到其找到一个 `*ast.StructType` ，然后返回它在文件中的起始位置。)

但是怎么做呢？记住有三种模式。分别是line选择，offset和struct name：

```
// findSelection返回可被更改的字段开始和结束位置。
// 它取决于line, struct或offset选择。
func (c *config) findSelection(node ast.Node) (int, int, error) {
    if c.line != "" {
        return c.lineSelection(node)
    } else if c.offset != 0 {
        return c.offsetSelection(node)
    } else if c.structName != "" {
        return c.structSelection(node)
    } else {
        return 0, 0, errors.New("-line, -offset or -struct is not passed")
    }
}
```

line选择部分是最简单的部分。这里我们只返回标志值本身。所以如果用户通过标志"--line 3,50"，函数返回(3, 50, nil)。它所做的就是拆分标志值并将其转换为整数（并通过执行验证）：


```

func (c *config) lineSelection(file ast.Node) (int, int, error) {
    var err error
    splitted := strings.Split(c.line, ",")

    start, err := strconv.Atoi(splitted[0])
    if err != nil {
        return 0, 0, err
    }

    end := start
    if len(splitted) == 2 {
        end, err = strconv.Atoi(splitted[1])
        if err != nil {
            return 0, 0, err
        }
    }

    if start > end {
        return 0, 0, errors.New("wrong range. start line cannot be larger than end line")
    }

    return start, end, nil
}

```

当您选择一组行并高亮显示时，编辑器将使用此模式。

offset和struct name选择需要更多的工作。对于这些，我们需要首先收集所有给定的结构，以便我们可以计算偏移位置或搜索结构名称。为此，我们有一个首先收集所有结构体的函数：

```

// collectStructs将structType节点收集并映射到其位置
func collectStructs(node ast.Node) map[token.Pos]*structType {
    structs := make(map[token.Pos]*structType, 0)
    collectStructs := func(n ast.Node) bool {
        t, ok := n.(*ast.TypeSpec)
        if !ok {
            return true
        }

        if t.Type == nil {
            return true
        }

        structName := t.Name.Name

        x, ok := t.Type.(*ast.StructType)
        if !ok {
            return true
        }

        structs[x.Pos()] = &structType{
            name: structName,
            node: x,
        }
        return true
    }
    ast.Inspect(node, collectStructs)
    return structs
}

```

我们使用ast.Inspect()函数通过AST并向下搜索结构。

我们首先搜索*ast.TypeSpec，以便我们可以获得结构名称。搜索*ast.StructType会给我们结构本身，而不是它的名称。这就是为什么我们有一个自定义的structType类型，它保存了名称和结构节点本身。这在

各个地方都很方便。因为每个结构的位置是唯一的，并且在同一位置上不能有不同的两个结构，所以我们使用位置作为地图的键。

所以现在我们有了所有的结构体，我们最终可以返回一个结构体的起始位置和结束位置的偏移量和结构体名称模式。对于偏移位置，我们检查偏移是否在给定的结构之间：

```
func (c *config) offsetSelection(file ast.Node) (int, int, error) {
    structs := collectStructs(file)

    var encStruct *ast.StructType
    for _, st := range structs {
        structBegin := c.fset.Position(st.node.Pos()).Offset
        structEnd := c.fset.Position(st.node.End()).Offset

        if structBegin <= c.offset && c.offset <= structEnd {
            encStruct = st.node
            break
        }
    }

    if encStruct == nil {
        return 0, 0, errors.New("offset is not inside a struct")
    }

    // 偏移模式选择所有字段
    start := c.fset.Position(encStruct.Pos()).Line
    end := c.fset.Position(encStruct.End()).Line

    return start, end, nil
}
```

我们使用collectStructs()来收集所有的结构体，然后在这里迭代。存储了我们用来解析初始文件token.FileSet？

这是现在用于获取每个结构节点的 offset 信息（我们将其解码为一个token.Position，它为我们提供了.Offset字段）。我们所做的只是一个简单的检查和迭代，直到我们找到我们需要的struct（在这里命名为encStruct）：

```
for _, st := range structs {
    structBegin := c.fset.Position(st.node.Pos()).Offset
    structEnd := c.fset.Position(st.node.End()).Offset

    if structBegin <= c.offset && c.offset <= structEnd {
        encStruct = st.node
        break
    }
}
```

有了这些信息，我们可以提取我们发现的结构体的开始和结束位置：

```
start := c.fset.Position(encStruct.Pos()).Line
end := c.fset.Position(encStruct.End()).Line
```

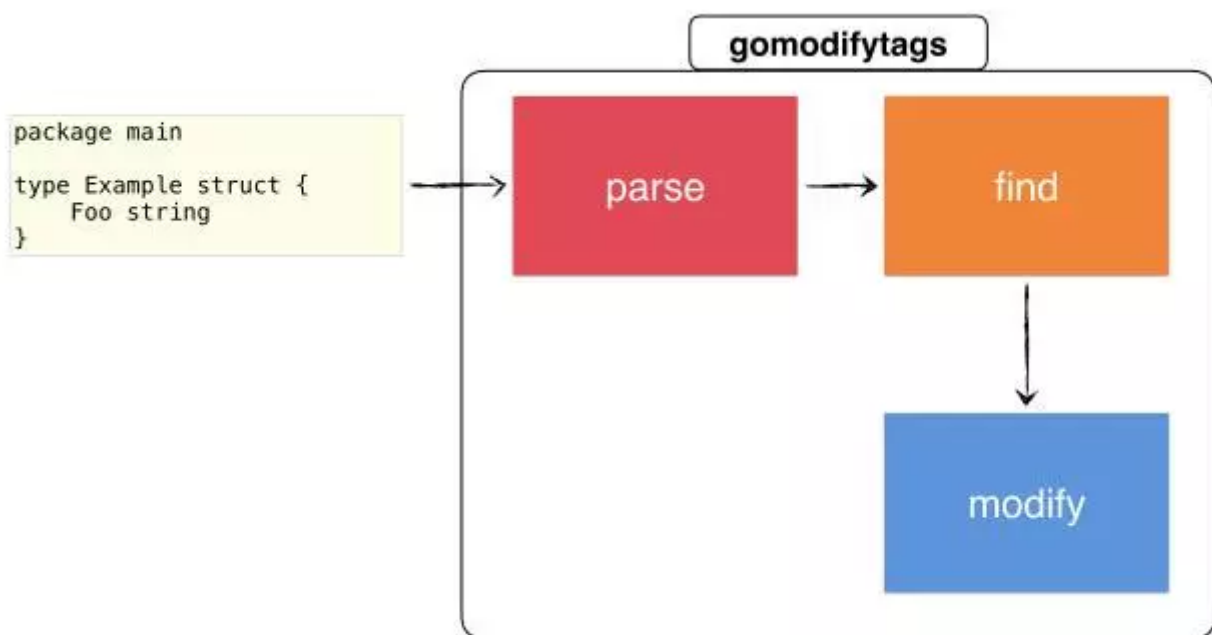
同样的逻辑也适用于结构体名称选择。我们所做的不只是检查偏移是否在给定的结构内，而是尝试检查结构名称，直到找到具有给定名称的结构体：

```
func (c *config) structSelection(file ast.Node) (int, int, error) {
    // ...

    for _, st := range structs {
        if st.name == c.structName {
            encStruct = st.node
        }
    }

    // ...
}
```

既然我们已经有了开始和结束位置，我能终于可以继续第三步了：修改结构体的字段：



在我们的主函数中，我们将会使用我们之前步骤解析的node来调用cfg.rewrite()函数：

```
func main() {
    // ... find start and end position of the struct to be modified

    rewrittenNode, errs := cfg.rewrite(node, start, end)
    if errs != nil {
        if _, ok := errs.(*rewriteErrors); !ok {
            return errs
        }
    }

    // continue outputting the rewritten node
}
```

这是这个工具的核心。在rewrite函数中，我们将会重写结构体中位于开始位置和结束位置之间的所有字段。在我们深入研究之前，下面是这个函数的概述：

```

// rewrite rewrites the node for structs between the start and end
// positions and returns the rewritten node
func (c *config) rewrite(node ast.Node, start, end int) (ast.Node, error) {
    errs := &rewriteErrors{errs: make([]error, 0)}

    rewriteFunc := func(n ast.Node) bool {
        // rewrite the node ...
    }

    if len(errs.errs) == 0 {
        return node, nil
    }

    ast.Inspect(node, rewriteFunc)
    return node, errs
}

```

正如你所看到的，我们再次在给定的node上使用 `ast.Inspect()` 函数来遍历该树。我们在`rewriteFunc` 函数中重写每个字段的标签（下面会有更多的内容）。

因为传递给 `ast.Inspect()` 的函数不会返回错误，所以我们将创建一个错误映射（使用 `errs` 变量来定义），然后在我们遍历树和处理每个单独的字段时收集错误的时候。我们来谈谈`rewriteFunc`的内部实现：

```

rewriteFunc := func(n ast.Node) bool {
    x, ok := n.(*ast.StructType)
    if !ok {
        return true
    }

    for _, f := range x.Fields.List {
        line := c.fset.Position(f.Pos()).Line

        if !(start <= line && line <= end) {
            continue
        }

        if f.Tag == nil {
            f.Tag = &ast.BasicLit{}
        }

        fieldName := ""
        if len(f.Names) != 0 {
            fieldName = f.Names[0].Name
        }

        // anonymous field
        if f.Names == nil {
            ident, ok := f.Type.(*ast.Ident)
            if !ok {
                continue
            }

            fieldName = ident.Name
        }

        res, err := c.process(fieldName, f.Tag.Value)
        if err != nil {
            errs.Append(fmt.Errorf("%s:%d:%d:%s",
                c.fset.Position(f.Pos()).Filename,
                c.fset.Position(f.Pos()).Line,
                c.fset.Position(f.Pos()).Column,
                err))
            continue
        }

        f.Tag.Value = res
    }

    return true
}

```

记得这个函数是在AST树种的每个节点上被调用的。因此，我们只查找类型为* ast.StructType的节点。一旦我们有了它，我们就开始迭代结构体的字段域了。

在这里我们使用开始和结束变量，决定了我们是否要修改该字段。如果该字段的位置在开始和结束之间，我们要继续修改，否则我们直接跳过：

```

if !(start <= line && line <= end) {
    continue // skip processing the field
}

```

接下来，我们检查是否存在标签。如果标签字段为空（a.k.a nil），那么我们将使用空标签来初始化该标签字段。这在后面的cfg.process()函数是用以避免程序挂起：

```
if f.Tag == nil {
    f.Tag = &ast.BasicLit{}
}
```

在我们继续之前，现在让我先解释一下有趣的一点知识。gomodifytags尝试获取字段的字段名称并处理它。然而，如果它只是一个匿名的字段呢？：

```
type Bar string

type Foo struct {
    Bar //this is an anonymous field
}
```

在这种情况下，因为没有字段名，我们会尝试从类型名中推断出字段名：

```
// if there is a field name use it
fieldName := ""
if len(f.Names) != 0 {
    fieldName = f.Names[0].Name
}

// if there is no field name, get it from type's name
if f.Names == nil {
    ident, ok := f.Type.(*ast.Ident)
    if !ok {
        continue
    }

    fieldName = ident.Name
}
```

一旦我们获得了字段名和标签值，我们就可以开始处理该字段了。cfg.process()函数负责处理具有给定字段名和标签值（如果有的话）的字段。它返回处理后的结果（在我们的例子中是格式化之后的struct tag）后，然后我们使用它来重写现有的标签值：

```
res, err := c.process(fieldName, f.Tag.Value)
if err != nil {
    errs.Append(fmt.Errorf("%s:%d:%d:%s",
        c.fset.Position(f.Pos()).Filename,
        c.fset.Position(f.Pos()).Line,
        c.fset.Position(f.Pos()).Column,
        err))
    continue
}

// rewrite the field with the new result,i.e: json:"foo"
f.Tag.Value = res
```

实际上，如果你记得structtag，它返回表示该标签实例的String()。在我们返回标签的最终表示之前，我们正在根据我们的需求使用structtag包的各种方法修改结构体。下面是一个简单的概述：

```

src := `package main
type Example struct {
    Foo string ` + " `json:\`foo\`" `}

fset := token.NewFileSet()
file, err := parser.ParseFile(fset, "demo",
if err != nil {
    panic(err)
}

ast.Inspect(file, func(x ast.Node) bool {
    s, ok := x.(*ast.StructType)
    if !ok {
        return true
    }

    for _, field := range s.Fields.List {
        // found field!
        field.Tag.Value = process(...)
    }

    return false
})

```

```

tags, err := structtag.Parse(tag)
if err != nil {
    return "", err
}

tags = c.removeTags(tags)
tags, err = c.removeTagOptions(tags)
if err != nil {
    return "", err
}

tags = c.clearTags(tags)
tags = c.clearOptions(tags)

tags, err = c.addTags(fieldName, tags)
if err != nil {
    return "", err
}

tags, err = c.addTagOptions(tags)
if err != nil {
    return "", err
}

return tags.String(), nil

```

(structtag包用于修改每个单独的字段)

例如，我们来扩展下process()中的removeTags()函数。此函数使用以下配置来创建待删除的标签数组（关键字）：

```

flagRemoveTags = flag.String("remove-tags", "", "Remove tags for the comma separated list of keys")

if *flagRemoveTags != "" {
    cfg.remove = strings.Split(*flagRemoveTags, ",")
}

```

在removeTags()中我们会检查是否有人使用了 remove-tags 选项。在这种情况下，我们将使用 structtag的tags.Delete()方法来移除tag：

```

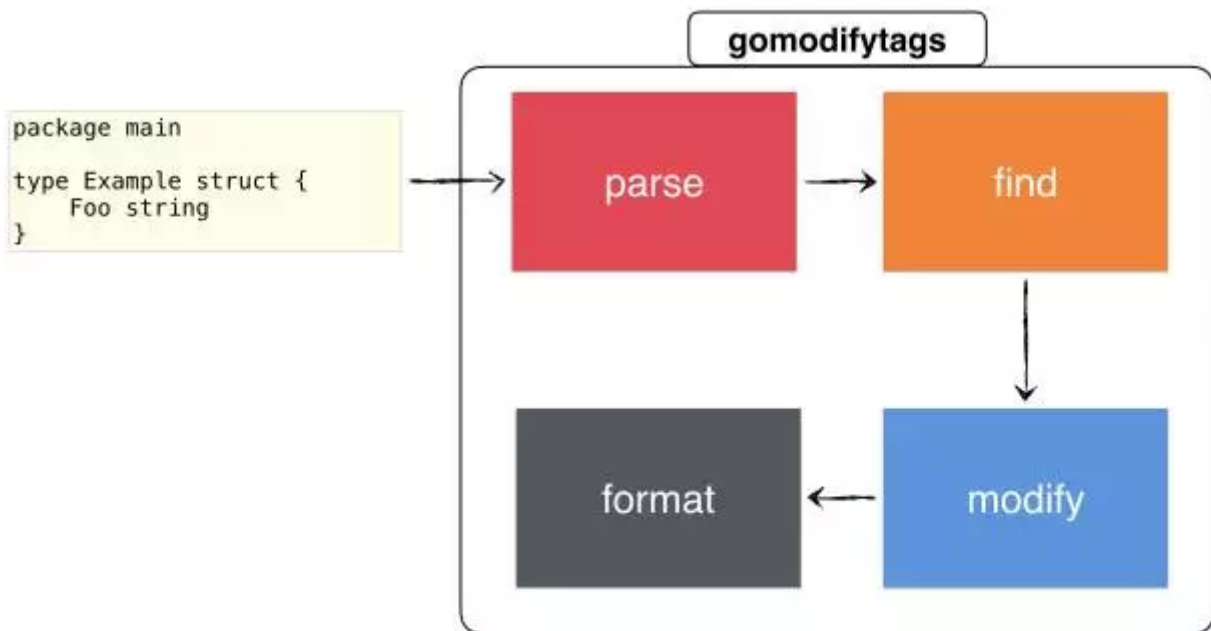
func (c *config) removeTags(tags *structtag.Tags) *structtag.Tags {
    if c.remove == nil || len(c.remove) == 0 {
        return tags
    }

    tags.Delete(c.remove...)
    return tags
}

```

同样的逻辑也适用于cfg.Process()中的所有独立函数。

既然我们已经完成了重写 node ，让我们探讨下最后一部分。输出和格式化结果：



在我们的主函数中，我们准备使用之前步骤中重写的 node 调用 `Icfg.format()` 函数：

```
func main() {
    // ... rewrite the node

    out, err := cfg.format(rewrittenNode, errs)
    if err != nil {
        return err
    }

    fmt.Println(out)
}
```

你需要注意的一件事是我们将会输出到 `stdout`。这会有诸多优势。首先，这允许任何人直接执行本工具，并看到输出结果。这并不会改变什么，但是允许使用该工具的用户能看到实时结果。第二，`stdout` 是可组合的，你可以将其重定向到任意位置，这样你甚至可以用于覆写原始工具。

然后我们深入看看 `format()` 函数：

```
func (c *config) format(file ast.Node, rwErrs error) (string, error) {
    switch c.output {
    case "source":
        // return Go source code
    case "json":
        // return a custom JSON output
    default:
        return "", fmt.Errorf("unknown output mode: %s", c.output)
    }
}
```

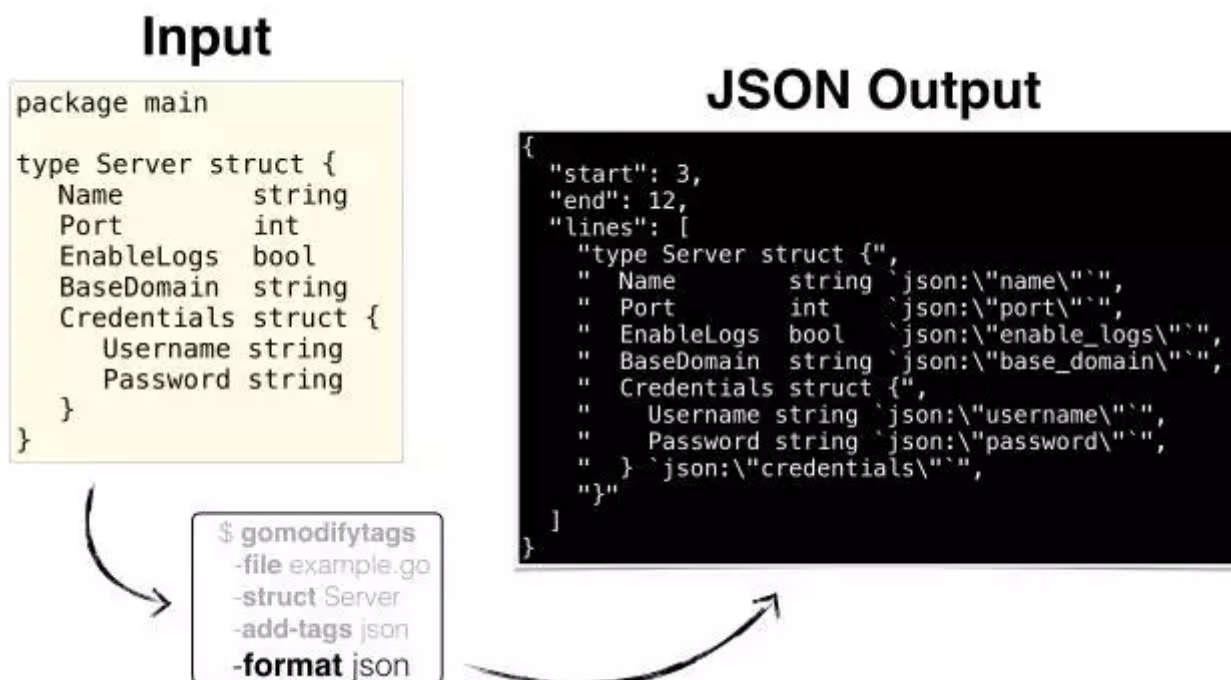
我们有2种输出模式。

第一种（“source”，源码模式）会使用 Go 语言格式来输出 `ast.Node`。这是默认选项，如果你在命令行或只想看看你文件中的改动，这个模式是非常适合的。

第二种选项（“json” 模式）更加高级，并且专门为其他环境（特别是编辑器）下设计的。它会将输出基于以下结构体进行编码：

```
type output struct {
    Start int    `json:"start"`
    End   int    `json:"end"`
    Lines []string `json:"lines"`
    Errors []string `json:"errors,omitempty"`
}
```

给该工具的一个输入及其对应输出（无任何错误发生的前提下）的一个概括如下：



回到 `format()` 函数上。正如前文所述，这里有两个模式。源代码模式使用 `go/format` 包来格式化一个 AST 为有效的 Go 源代码。这个包也被诸多其他官方工具使用，比如 `gofmt`。下面是“source”模式是如何实现的：

```
var buf bytes.Buffer
err := format.Node(&buf, c.fset, file)
if err != nil {
    return "", err
}

if c.write {
    err = ioutil.WriteFile(c.file, buf.Bytes(), 0)
    if err != nil {
        return "", err
    }
}

return buf.String(), nil
```

格式化包接收一个 `io.Writer`，然后开始格式化。这就是为什么我们创建一个中间缓冲（通过 `buf bytes.Buffer`）的原因，这样我们就可以在用户传递过来 `-write` 标志之后直接重写文件了。格式化之后，我们会返回表示该 buffer 的字符串，其中包含了格式化后的 Go 源代码。

json模式更有趣。因为我们返回了源码的一部分，所以我们需要准确地返回它的描述信息，这也意味着也包含了评论。问题在于，如果使用format.Node()输出单个结构体当注释是有损失的，也无法输出Go注释。

什么是有损注释呢？看看这个例子：

```
type example struct {  
    foo int  
  
    // this is a lossy comment  
  
    bar int  
}
```

每个字段的类型都为*ast.Field。此结构体具有* ast.Field.Comment字段，其中包含该特定字段的注释。

但是，在上面的例子中，它属于谁？是foo还是bar的一部分？

因为不可能区分这种情况，这些注释被称为松散的注释（译者注：此处前后不一致，这里是loosely，而前面是lossy，这里保持原意）。现在如果使用format.Node()函数输出上面的结构体，就会出现这个问题。当你输出它时，这是你得到的(<https://play.golang.org/p/peHsswF4JQ>):

```
type example struct {  
    foo int  
  
    bar int  
}
```

问题是有损注释是*ast.File的一部分，并与树分开。只有在输出整个文件时才会被输出的。所以此问题的解决方法是输出整个文件，然后剪切出我们要在JSON输出中返回的特定行：

```

var buf bytes.Buffer
err := format.Node(&buf, c.fset, file)
if err != nil {
    return "", err
}

var lines []string
scanner := bufio.NewScanner(bytes.NewBufferString(buf.String()))
for scanner.Scan() {
    lines = append(lines, scanner.Text())
}

if c.start > len(lines) {
    return "", errors.New("line selection is invalid")
}

out := &output{
    Start: c.start,
    End:   c.end,
    Lines: lines[c.start-1 : c.end], // cut out lines
}

o, err := json.MarshalIndent(out, "", " ")
if err != nil {
    return "", err
}

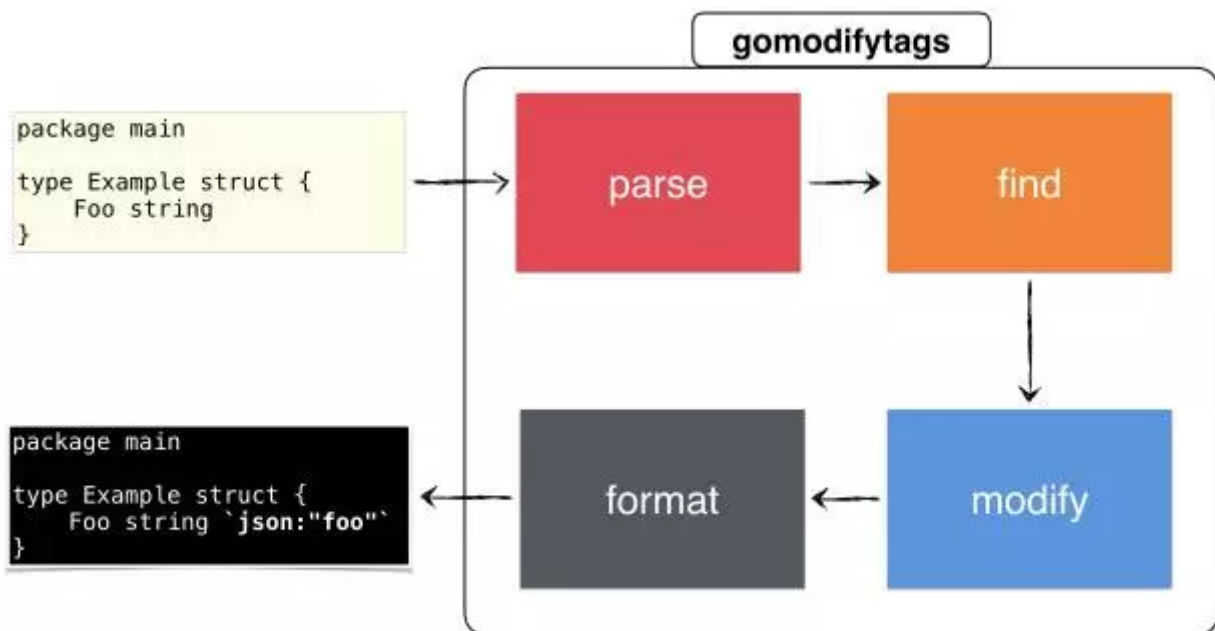
return string(o), nil

```

这就可以保证我们输出了所有的注释。

到此为止！

我们成功地完成了我们的工具，以下是我们在整个指南中所使用的完整的步骤图：



(`gomodifytags` 的概况)

回顾一下我们做了什么：

- 我们通过 CLI 标志检索配置文件

- 我们通过 go/parser 包**解析**文件来获得 ast.Node 。
- 在解析文件之后，我们**搜索**（沿树向下）以获取相应的结构，并获取其起位置，以便我们知道需要修改哪些字段
- 一旦我们有开始和结束位置，我们再次进入 ast.Node，**重写**位于开始和结束位置之间的每个字段（通过使用 structtag 包）
- 然后，我们将重写的节点**格式化**为有效的 Go 源代码或用于编辑器的自定义 JSON 输出

创建此工具后，我收到了很多很好的评论，提及了这个工具如何简化了他们的日常工作。尽管看起来它很容易实现，正如你所看到的，在整个指南中，我们已经看到有许多特殊情况需要处理。

gomodifytags 现在被下列编辑器和插件成功使用了数月，使数以千计的开发人员的生活更加高效：

- vim-go
- atom
- vscode
- acme

如果你对最初的源代码感兴趣，它可以在这里找到：

- <https://github.com/fatih/gomodifytags>

感谢阅读。如果这个指南启发你从头创建一个新的 Go 工具，请告知。