# Golang标准库——mime

## mime

mime实现了MIME的部分规定。

### func AddExtensionType

// AddExtensionType sets the MIME type associated with  
// the extension ext to typ. The extension should begin with  
// a leading dot, as in ".html".  
func AddExtensionType(ext, typ string) error {  
 if !strings.HasPrefix(ext, ".") {  
 return fmt.Errorf("mime: extension %q missing leading dot", ext)  
 }  
 once.Do(initMime)  
 return setExtensionType(ext, typ)  
}

函数将扩展名和mimetype建立偶联；扩展名应以点号开始，例如".html"。

### func FormatMediaType

func FormatMediaType(t string, param map[string]string) string

函数根据[RFC 2045](https://links.jianshu.com/go?to=http://tools.ietf.org/html/rfc2045" \t "_blank)和 [RFC 2616](https://links.jianshu.com/go?to=http://tools.ietf.org/html/rfc2616" \t "_blank)的规定将媒体类型t和参数param连接为一个mime媒体类型，类型和参数都采用小写字母。任一个参数不合法都会返回空字符串。

### func ParseMediaType

func ParseMediaType(v string) (mediatype string, params map[string]string, err error)

函数根据[RFC 1521](https://links.jianshu.com/go?to=http://tools.ietf.org/html/rfc1521" \t "_blank)解析一个媒体类型值以及可能的参数。媒体类型值一般应为Content-Type和Conten-Disposition头域的值（参见[RFC 2183](https://links.jianshu.com/go?to=http://tools.ietf.org/html/rfc2183" \t "_blank)）。成功的调用会返回小写字母、去空格的媒体类型和一个非空的map。返回的map映射小写字母的属性和对应的属性值。

### func TypeByExtension

func TypeByExtension(ext string) string

函数返回与扩展名偶联的MIME类型。扩展名应以点号开始，如".html"。如果扩展名未偶联类型，函数会返回""。

内建的偶联表很小，但在unix系统会从本地系统的一或多个mime.types文件（参加下表）进行增补。

/etc/mime.types/etc/apache2/mime.types/etc/apache/mime.types

Windows系统的mime类型从注册表获取。文本类型的字符集参数默认设置为"utf-8"。

* multipart
* quotedprintable

## multipart

multipart实现了MIME的multipart解析，参见[RFC 2046](https://links.jianshu.com/go?to=http://tools.ietf.org/html/rfc2046" \t "_blank)。该实现适用于HTTP（[RFC 2388](https://links.jianshu.com/go?to=http://tools.ietf.org/html/rfc2388" \t "_blank)）和常见浏览器生成的multipart主体。

### type File

// File is an interface to access the file part of a multipart message.  
// Its contents may be either stored in memory or on disk.  
// If stored on disk, the File's underlying concrete type will be an \*os.File.  
type File interface {  
 io.Reader  
 io.ReaderAt  
 io.Seeker  
 io.Closer  
}

File是一个接口，实现了对一个multipart信息中文件记录的访问。它的内容可以保持在内存或者硬盘中，如果保持在硬盘中，底层类型就会是\*os.File。

### type FileHeader

// A FileHeader describes a file part of a multipart request.  
type FileHeader struct {  
 Filename string  
 Header textproto.MIMEHeader  
 Size int64  
  
 content []byte  
 tmpfile string  
}

FileHeader描述一个multipart请求的（一个）文件记录的信息。

#### func (\*FileHeader) Open

func (fh \*FileHeader) Open() (File, error)

Open方法打开并返回其关联的文件。

### type Part

// A Part represents a single part in a multipart body.  
type Part struct {

// 主体的头域，如果存在，是按Go的http.Header风格标准化的，如"foo-bar"改变为"Foo-Bar"。

// 有一个特殊情况，如果"Content-Transfer-Encoding"头的值是"quoted-printable"。

// 该头将从本map中隐藏，而主体会在调用Read时透明的解码。  
 Header textproto.MIMEHeader  
  
 mr \*Reader  
  
 disposition string  
 dispositionParams map[string]string  
  
 // r is either a reader directly reading from mr, or it's a  
 // wrapper around such a reader, decoding the  
 // Content-Transfer-Encoding  
 r io.Reader  
  
 n int // known data bytes waiting in mr.bufReader  
 total int64 // total data bytes read already  
 err error // error to return when n == 0  
 readErr error // read error observed from mr.bufReader  
}

Part代表multipart主体的单独一个记录。

#### func (\*Part) FileName

func (p \*Part) FileName() string

返回Part 的Content-Disposition 头的文件名参数。

#### func (\*Part) FormName

func (p \*Part) FormName() string

如果p的Content-Disposition头值为"form-data"，则返回名字参数；否则返回空字符串。

#### func (\*Part) Read

func (p \*Part) Read(d []byte) (n int, err error)

Read方法读取一个记录的主体，也就是其头域之后到下一记录之前的部分。

#### func (\*Part) Close

func (p \*Part) Close() error

### type Form

// Form is a parsed multipart form.  
// Its File parts are stored either in memory or on disk,  
// and are accessible via the \*FileHeader's Open method.  
// Its Value parts are stored as strings.  
// Both are keyed by field name.  
type Form struct {  
 Value map[string][]string  
 File map[string][]\*FileHeader  
}

Form是一个解析过的multipart表格。它的File参数部分保存在内存或者硬盘上，可以使用\*FileHeader类型属性值的Open方法访问。它的Value 参数部分保存为字符串，两者都以属性名为键。

#### func (\*Form) RemoveAll

func (f \*Form) RemoveAll() error

删除Form关联的所有临时文件。

### type Reader

// Reader is an iterator over parts in a MIME multipart body.  
// Reader's underlying parser consumes its input as needed. Seeking  
// isn't supported.  
type Reader struct {  
 bufReader \*bufio.Reader  
  
 currentPart \*Part  
 partsRead int  
  
 nl []byte // "\r\n" or "\n" (set after seeing first boundary line)  
 nlDashBoundary []byte // nl + "--boundary"  
 dashBoundaryDash []byte // "--boundary--"  
 dashBoundary []byte // "--boundary"  
}

Reader是MIME的multipart主体所有记录的迭代器。Reader的底层会根据需要解析输入，不支持Seek。

#### func NewReader

func NewReader(r io.Reader, boundary string) \*Reader

函数使用给出的MIME边界和r创建一个multipart读取器。

边界一般从信息的"Content-Type" 头的"boundary"属性获取。可使用mime.ParseMediaType函数解析这种头域。

func main() {  
 msg := &mail.Message{  
 Header: map[string][]string{  
 "Content-Type": []string{"multipart/mixed; boundary=foo"},  
 },  
 Body: strings.NewReader(  
 "--foo\r\nFoo: one\r\n\r\nA section\r\n" +  
 "--foo\r\nFoo: two\r\n\r\nAnd another\r\n" +  
 "--foo--\r\n"),  
 }  
 mediaType, params, err := mime.ParseMediaType(msg.Header.Get("Content-Type"))  
 if err != nil {  
 log.Fatal(err)  
 }  
 if strings.HasPrefix(mediaType, "multipart/") {  
 mr := multipart.NewReader(msg.Body, params["boundary"])  
 for {  
 p, err := mr.NextPart()  
 if err == io.EOF {  
 return  
 }  
 if err != nil {  
 log.Fatal(err)  
 }  
 slurp, err := ioutil.ReadAll(p)  
 if err != nil {  
 log.Fatal(err)  
 }  
 fmt.Printf("Part %q: %q\n", p.Header.Get("Foo"), slurp)  
 }  
 }  
}

输出：

Part "one": "A section"  
Part "two": "And another"

#### func (\*Reader) ReadForm

func (r \*Reader) ReadForm(maxMemory int64) (f \*Form, err error)

ReadForm解析整个multipart信息中所有Content-Disposition头的值为"form-data"的记录。它会把最多maxMemory字节的文件记录保存在内存里，其余保存在硬盘的临时文件里。

#### func (\*Reader) NextPart

func (r \*Reader) NextPart() (\*Part, error)

NextPart返回multipart的下一个记录或者返回错误。如果没有更多记录会返回io.EOF。

### type Writer

// A Writer generates multipart messages.  
type Writer struct {  
 w io.Writer  
 boundary string  
 lastpart \*part  
}

Writer类型用于生成multipart信息。

#### func NewWriter

func NewWriter(w io.Writer) \*Writer

NewWriter函数返回一个设定了一个随机边界的Writer，数据写入w。

#### func (\*Writer) FormDataContentType

func (w \*Writer) FormDataContentType() string

方法返回w对应的HTTP multipart请求的Content-Type的值，多以multipart/form-data起始。

#### func (\*Writer) Boundary

func (w \*Writer) Boundary() string

方法返回该Writer的边界。

#### func (\*Writer) SetBoundary

func (w \*Writer) SetBoundary(boundary string) error

SetBoundary方法重写Writer默认的随机生成的边界为提供的boundary参数。方法必须在创建任何记录之前调用，boundary只能包含特定的ascii字符，并且长度应在1-69字节之间。

#### func (\*Writer) CreatePart

func (w \*Writer) CreatePart(header textproto.MIMEHeader) (io.Writer, error)

CreatePart方法使用提供的header创建一个新的multipart记录。该记录的主体应该写入返回的Writer接口。调用本方法后，任何之前的记录都不能再写入。

#### func (\*Writer) CreateFormField

func (w \*Writer) CreateFormField(fieldname string) (io.Writer, error)

CreateFormField方法使用给出的属性名调用CreatePart方法。

#### func (\*Writer) CreateFormFile

func (w \*Writer) CreateFormFile(fieldname, filename string) (io.Writer, error)

CreateFormFile是CreatePart方法的包装， 使用给出的属性名和文件名创建一个新的form-data头。

#### func (\*Writer) WriteField

func (w \*Writer) WriteField(fieldname, value string) error

WriteField方法调用CreateFormField并写入给出的value。

#### func (\*Writer) Close

func (w \*Writer) Close() error

Close方法结束multipart信息，并将结尾的边界写入底层io.Writer接口。

## quotedprintable

quotedprintable包实现 [RFC 2045](https://links.jianshu.com/go?to=http://tools.ietf.org/html/rfc2045" \t "_blank).指定的quoted-printable编码。

### type Reader

// Reader is an iterator over parts in a MIME multipart body.  
// Reader's underlying parser consumes its input as needed. Seeking  
// isn't supported.  
type Reader struct {  
 bufReader \*bufio.Reader  
  
 currentPart \*Part  
 partsRead int  
  
 nl []byte // "\r\n" or "\n" (set after seeing first boundary line)  
 nlDashBoundary []byte // nl + "--boundary"  
 dashBoundaryDash []byte // "--boundary--"  
 dashBoundary []byte // "--boundary"  
}

Reader是带引号的可打印解码器。

#### func NewReader

func NewReader(r io.Reader) \*Reader

NewReader返回带引号的可打印阅读器，从r解码。

func main() {  
 for \_, s := range []string{  
 `=48=65=6C=6C=6F=2C=20=47=6F=70=68=65=72=73=21`,  
 `invalid escape: <b style="font-size: 200%">hello</b>`,  
 "Hello, Gophers! This symbol will be unescaped: =3D and this will be written in =\r\none line.",  
 } {  
 b, err := ioutil.ReadAll(quotedprintable.NewReader(strings.NewReader(s)))  
 fmt.Printf("%s %v\n", b, err)  
 }  
}

输出：

Hello, Gophers! <nil>  
invalid escape: <b style="font-size: 200%">hello</b> <nil>   
Hello, Gophers! This symbol will be unescaped: = and this will be written in one  
 line. <nil>

#### func (\*Reader) Read

func (r \*Reader) Read(p []byte) (n int, err error)

读取从基础读取器读取和解码带引号的可打印数据。

### type Writer

// A Writer generates multipart messages.  
type Writer struct {  
 w io.Writer  
 boundary string  
 lastpart \*part  
}

Writer是实现io.WriteCloser的带引号的可打印书写器。

#### func NewWriter

func NewWriter(w io.Writer) \*Writer

NewWriter返回一个写入w的新Writer。

#### func (\*Writer) Close

func (w \*Writer) Close() error

Close关闭Writer，将所有未写入的数据刷新到基础io.Writer，但不关闭基础io.Writer。

#### func (\*Writer) Write

func (w \*Writer) Write(p []byte) (n int, err error)

Write使用带引号的可打印编码对p进行编码，并将其写入基础io.Writer。 它将行长度限制为76个字符。 在关闭Writer之前，不必刷新已编码的字节。