ASCII ART WEB DOCUMENTATION.

main.go

This Go file defines a simple web server that handles different routes and serves static files. Here's a breakdown of each part:

Package Declaration

```
package main
```

This declares that this file belongs to the `main` package, which is the starting point for a Go program.

Imports

```
import (
        "fmt"
        "net/http"
        handler "web/handlers"
)
```

This imports three packages:

- **`fmt`**: Provides functions for formatting text, such as printing to the console.
- **`net/http`**: Provides HTTP client and server implementations.
- **`handler "web/handlers"`**: Imports the `handlers` package from the `web` directory and aliases it as `handler`. This package presumably contains custom HTTP handlers.

Main Function

```
func main() {
        http.HandleFunc("/", handler.FormHandler)
        http.HandleFunc("/ascii-art", handler.AsciiArtHandler)
        http.HandleFunc("/favicon.ico", http.NotFound) // Handle favicon.ico requests
        http.Handle("/static/", http.StripPrefix("/static/", http.FileServer(http.Dir("./static"))))
        fmt.Println("Server started at http://localhost:8080")
        err := http.ListenAndServe(":8080", nil)
        if err != nil {
                fmt.Println("Error starting server:", err)
        }
}
```

Route Handlers

1. **Root Handler**

```
http.HandleFunc("/", handler.FormHandler)
```

This maps the root URL (`/`) to the `FormHandler` function in the `handler` package.

2. **ASCII Art Handler**

```
http.HandleFunc("/ascii-art", handler.AsciiArtHandler)
```

This maps the `/ascii-art` URL to the `AsciiArtHandler` function in the `handler` package.

3. **Favicon Handler**

```
http.HandleFunc("/favicon.ico", http.NotFound)
```

This maps requests for the favicon (`/favicon.ico`) to the `http.NotFound` handler, which returns a 404 not found response.

4. **Static File Server**

```
http.Handle("/static/", http.StripPrefix("/static/",
http.FileServer(http.Dir("./static"))))
```

This serves static files from the ./static directory. Requests to URLs starting with /static/ will have the /static/ prefix stripped before looking for the corresponding file in the ./static directory.

Starting the Server

```
fmt.Println("Server started at http://localhost:8080")
err := http.ListenAndServe(":8080", nil)
if err != nil {
        fmt.Println("Error starting server:", err)
}
```

1. **Print Start Message**

```
fmt.Println("Server started at http://localhost:8080")
```

This prints a message to the console indicating that the server has started.

2. **Start Server**

```
err := http.ListenAndServe(":8080", nil)
if err != nil {
    fmt.Println("Error starting server:", err)
}
```

This starts the HTTP server on port 8080. If there is an error starting the server, it will print an error message.

Summary

- The file sets up a web server with three routes (`/`, `/ascii-art`, and `/favicon.ico`).
- It serves static files from the `./static` directory.
- It starts the server on port 8080 and handles any startup errors.

handlers.go

This file contains the implementation of the handlers referenced in the first file, which are part of the `handler` package. The handlers manage specific routes and perform various tasks based on incoming HTTP requests. Here's a detailed breakdown of each component in this file:

Package Declaration

package handler
This declares that the file is part of the `handler` package, which is referenced in the `main` package.

Imports

```
import (
	"html/template"
	"net/http"
	asciiArt "web/ascii-funcs"
	utils "web/utilities"
)
```

This imports several packages:
- `**html/template**`: Provides support for HTML templates.
- `**net/http**`: Provides HTTP client and server implementations.
- `**asciiArt "web/ascii-funcs"**`: Imports the `ascii-funcs` package from the `web` directory and aliases it as `asciiArt`. This package is presumably responsible for loading ASCII characters.
- `**utils "web/utilities"**`: Imports the `utilities` package from the `web` directory and aliases it as `utils`. This package likely contains utility functions.

Handlers

FormHandler

```
func FormHandler(w http.ResponseWriter, r *http.Request) {
	if r.URL.Path != "/" {
		http.Error(w, "HTTP status 404 - page not found", http.StatusNotFound)
		return
	}

	tmpl := template.Must(template.ParseFiles("templates/form.html"))
	err := tmpl.Execute(w, nil)
	if err != nil {
		http.Error(w, "HTTP status 500 - Internal Server Errors",
http.StatusInternalServerError)
	}
}
```

1. **URL Path Check**

```
  if r.URL.Path != "/" {
    http.Error(w, "HTTP status 404 - page not found", http.StatusNotFound)
    return
  }
```

Checks if the request URL is exactly `/`. If not, it returns a 404 error.

2. **Template Parsing and Execution**

```
tmpl := template.Must(template.ParseFiles("templates/form.html"))
err := tmpl.Execute(w, nil)
if err != nil {
    http.Error(w, "HTTP status 500 - Internal Server Errors", http.StatusInternalServerError)
}
```

- Parses the `form.html` template.
- Executes the template, sending its output to the HTTP response.
- If there's an error during execution, it returns a 500 internal server error.

AsciiArtHandler

```
func AsciiArtHandler(w http.ResponseWriter, r *http.Request) {
	if r.Method != http.MethodPost {
		http.Error(w, "HTTP status 405 - method not allowed", http.StatusMethodNotAllowed)
		return
	}

	text := r.FormValue("text")
	banner := r.FormValue("banner")

	if text == "" {
		http.Error(w, "HTTP status 400 - Bad Request: 'text' parameter is required", http.StatusBadRequest)
		return
	}

	if banner == "" {
		banner = "standard"
	}

	asciiChars, err := asciiArt.LoadAsciiChars("banners/" + banner + ".txt")
	if err != nil {
		http.Error(w, "500 internal server error: could not load banner", http.StatusInternalServerError)
		return
	}

	art := utils.GenerateAsciiArt(text, asciiChars)

	tmpl := template.Must(template.ParseFiles("templates/result.html"))
	err = tmpl.Execute(w, art)
	if err != nil {
		http.Error(w, "500 internal server error", http.StatusInternalServerError)
	}
}
```

1. **Method Check**

```
if r.Method != http.MethodPost {
    http.Error(w, "HTTP status 405 - method not allowed", http.StatusMethodNotAllowed)
    return
}
```

Checks if the request method is `POST`. If not, it returns a 405 error.

2. **Form Values Extraction**

```
text := r.FormValue("text")
banner := r.FormValue("banner")
```

Extracts the `text` and `banner` parameters from the form data.

3. **Parameter Validation**

```
if text == "" {
    http.Error(w, "HTTP status 400 - Bad Request: 'text' parameter is required", http.StatusBadRequest)
    return
}

if banner == "" {
    banner = "standard"
}
```

- Checks if the `text` parameter is provided. If not, it returns a 400 error.
- Sets a default value of `"standard"` for the `banner` if it is not provided.

4. **ASCII Characters Loading**

```
asciiChars, err := asciiArt.LoadAsciiChars("banners/" + banner + ".txt")
if err != nil {
    http.Error(w, "500 internal server error: could not load banner", http.StatusInternalServerError)
    return
}
```

Loads the ASCII characters from the specified banner file. If there is an error, it returns a 500 internal server error.

5. **ASCII Art Generation**

```
art := utils.GenerateAsciiArt(text, asciiChars)
```

Uses the `GenerateAsciiArt` function from the `utilities` package to generate ASCII art from the text and loaded characters.

6. ***Template Parsing and Execution***

```
tmpl := template.Must(template.ParseFiles("templates/result.html"))
err = tmpl.Execute(w, art)
if err != nil {
    http.Error(w, "500 internal server error", http.StatusInternalServerError)
}
```

- Parses the `result.html` template.
- Executes the template with the generated ASCII art.

- If there's an error during execution, it returns a 500 internal server error.

Summary

- **FormHandler** serves the form page for ASCII art submission.
- **AsciiArtHandler** processes form submissions, generates ASCII art, and displays the result.
- Both handlers manage various HTTP errors and respond accordingly.
- The handlers utilize templates to render HTML pages and utility functions for generating ASCII art.

Form.html

This HTML file defines a simple web page for an ASCII Art Generator application. Here's a detailed breakdown of its structure and components:

Document Type and Language

```
<!DOCTYPE html>
<html lang="en">
```

- Declares the document as an HTML5 document.
- Specifies the language of the document as English.

Head Section

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ASCII Art Generator</title>
    <link rel="stylesheet" href="/static/css/styles.css">
</head>
```

1. **Meta Tags**
   - **`<meta charset="UTF-8">`**: Sets the character encoding to UTF-8.
   - **`<meta name="viewport" content="width=device-width, initial-scale=1.0">`**: Ensures proper scaling on different devices by setting the viewport width to the device width.

2. **Title**
   - **`<title>ASCII Art Generator</title>`**: Sets the title of the webpage, displayed on the browser tab.

3. **Stylesheet Link**
   - **`<link rel="stylesheet" href="/static/css/styles.css">`**: Links to an external CSS file for styling the webpage. The file is located in the `/static/css/` directory.

Body Section

```
<body>
    <h1>ASCII Art Generator</h1>
    <form id="asciiForm" action="/ascii-art" method="POST">
        <label for="text">Text:</label><br>
        <textarea id="text" name="text" rows="10" cols="30" placeholder="Enter text here..."></textarea><br><br>
```

```
    <label for="banner">Banner Style:</label><br>
    <select id="banner" name="banner">
      <option value="standard">Standard</option>
      <option value="shadow">Shadow</option>
      <option value="thinkertoy">Thinkertoy</option>
    </select><br><br>
    <input type="submit" value="Generate">
  </form>
</body>
```

1. **Heading**
   - `**<h1>ASCII Art Generator</h1>**`: Displays a level 1 heading with the text "ASCII Art Generator".

2. **Form**
   `<form id="asciiForm" action="/ascii-art" method="POST">`

   - `**<form>**`: Defines a form element for user input.
   - `**id="asciiForm"**`: Assigns an ID to the form for reference in JavaScript or CSS.
   - `**action="/ascii-art"**`: Specifies that the form data should be submitted to the `/ascii-art` endpoint.
   - `**method="POST"**`: Uses the POST method to submit the form data.

3. **Text Input**

```
  <label for="text">Text:</label><br>
  <textarea id="text" name="text" rows="10" cols="30" placeholder="Enter text
here..."></textarea><br><br>
```

   - `**<label for="text">Text:</label>**`: Label for the text input area.
   - `**<textarea id="text" name="text" rows="10" cols="30" placeholder="Enter text here..."></textarea>**`: A textarea element for users to input the text to be converted into ASCII art.

4. **Banner Style Selection**

```
  <label for="banner">Banner Style:</label><br>
  <select id="banner" name="banner">
    <option value="standard">Standard</option>
    <option value="shadow">Shadow</option>
    <option value="thinkertoy">Thinkertoy</option>
  </select><br><br>
```

   - `**<label for="banner">Banner Style:</label>**`: Label for the banner style selection.
   - `**<select id="banner" name="banner">**`: A dropdown menu for selecting the banner style.
- `**<option>**` elements: Provide three options (`Standard`, `Shadow`, and `Thinkertoy`) for the banner style.

5. **Submit Button**

```
  <input type="submit" value="Generate">
```

   - `**<input type="submit" value="Generate">**`: A submit button that sends the form data when clicked.

Script Section

```
<script src="/static/js/scripts.js" defer></script>
</html>
```

1. **Script Link**
   - `**<script src="/static/js/scripts.js" defer></script>**`: Links to an external JavaScript file located in the `/static/js/` directory.
   - `**defer**`: Ensures the script is executed after the HTML document has been completely parsed.

Summary

- This HTML file defines the user interface for an ASCII Art Generator.
- It includes a form where users can input text and select a banner style.
- The form submits the data to the `/ascii-art` endpoint using the POST method.
- The page links to external CSS and JavaScript files for additional styling and functionality.

result.html

This HTML file defines the web page for displaying the result of the ASCII Art Generator. Here is a detailed explanation of its structure and components:

Document Type and Language

```
<!DOCTYPE html>
<html lang="en">
```

- Declares the document as an HTML5 document.
- Specifies the language of the document as English.

Head Section

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ASCII Art Result</title>
    <link rel="stylesheet" href="/static/css/styles.css">
</head>
```

1. **Meta Tags**
   - `**<meta charset="UTF-8">**`: Sets the character encoding to UTF-8.
- `**<meta name="viewport" content="width=device-width, initial-scale=1.0">**`: Ensures proper scaling on different devices by setting the viewport width to the device width.
2. **Title**
   - `**<title>ASCII Art Result</title>**`: Sets the title of the webpage, displayed on the browser tab.

3. **Stylesheet Link**
   - `**<link rel="stylesheet" href="/static/css/styles.css">**`: Links to an external CSS file for styling the webpage. The file is located in the `/static/css/` directory.

Body Section

```
<body>
   <div class="result">
   <h1>ASCII Art Result</h1>
   <pre>{{.}}</pre>
   <a href="/">Go Back</a>
</div>
</body>
```

1. **Result Container**

```
   <div class="result">
```

   - `**<div class="result">**`: A `div` element with the class `result`, used to contain and style the result section.

2. **Heading**

```
   <h1>ASCII Art Result</h1>
```

   - `**<h1>ASCII Art Result</h1>**`: Displays a level 1 heading with the text "ASCII Art Result".

3. **Preformatted Text**

```
   <pre>{{.}}</pre>
```

   - `**<pre>{{.}}</pre>**`: A `pre` element used to display preformatted text. The `{{.}}` is a placeholder for the ASCII art generated by the server. In Go templates, `{{.}}` represents the data passed to the template.

4. **Go Back Link**

```
   <a href="/">Go Back</a>
```

   - `**<a href="/">Go Back</a>**`: An anchor element that provides a link to go back to the home page (`/`).

Script Section

```
<script src="/static/js/scripts.js" defer></script>
</html>
```

1. **Script Link**
   - `**<script src="/static/js/scripts.js" defer></script>**`: Links to an external JavaScript file located in the `/static/js/` directory.
   - `**defer**`: Ensures the script is executed after the HTML document has been completely parsed.

Summary

- This HTML file defines the layout for displaying the result of the ASCII Art Generator.
- It includes a container (`div`) for the result with a heading, the ASCII art output, and a link to go back to the home page.
- The `pre` element with `{{.}}` is a placeholder for the ASCII art content generated by the server.
- The page links to external CSS and JavaScript files for additional styling and functionality.

Scripts.js

This JavaScript file handles the form submission for the ASCII Art Generator application, ensuring that the form data can be submitted asynchronously (without reloading the page) and that multiline text input is supported. Here is a detailed breakdown of its components and functionality:

Form Submission Handling

```
document.getElementById('asciiForm').onsubmit = function(e) {
    e.preventDefault();
```

- **`document.getElementById('asciiForm').onsubmit`**: Sets an event handler for the form's `onsubmit` event. This means that when the form is submitted, the specified function will be executed.
- **`function(e) { e.preventDefault(); }`**: The event handler function takes an event object `e` as an argument. `e.preventDefault()` prevents the default form submission behavior, which would normally reload the page.

Extracting Form Data

```
const text = document.getElementById('text').value;
const banner = document.getElementById('banner').value;
```

- **`const text = document.getElementById('text').value;`**: Retrieves the value from the textarea with the ID `text`, which contains the user's input text.
- **`const banner = document.getElementById('banner').value;`**: Retrieves the selected value from the dropdown menu with the ID `banner`, which contains the selected banner style.

Sending Form Data with Fetch API

```
fetch('/ascii-art', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
    },
    body: new URLSearchParams({
        text: text,
        banner: banner,
    }),
})
```

- **`fetch('/ascii-art', { ... })`**: Sends an HTTP request to the `/ascii-art` endpoint using the Fetch API.
- **`method: 'POST'`**: Specifies that the request method is POST.
- **`headers: { 'Content-Type': 'application/x-www-form-urlencoded', }`**: Sets the `Content-Type` header to `application/x-www-form-urlencoded`, indicating that the form data is URL-encoded.
- **`body: new URLSearchParams({ text: text, banner: banner, })`**: Encodes the form data (text and banner) as URL parameters and sets it as the request body.

Handling the Response

```
.then(response => response.text())
.then(data => {
    window.history.pushState({}, '', '/ascii-art');
    document.open();
    document.write(data);
    document.close();
});
```

- `**.then(response => response.text())**`: When the fetch request completes, this chain of `.then` methods handles the response. `response.text()` extracts the response body as a plain text string.
- `**.then(data => { ... })**`: Processes the extracted text (`data`) from the response.
  - `**window.history.pushState({}, '', '/ascii-art');**`: Updates the browser's URL to `/ascii-art` without reloading the page or navigating to a new page. This helps maintain the correct URL in the address bar.
  - `**document.open(); document.write(data); document.close();**`: Replaces the current document's contents with the response data. `document.open()` opens the document for writing, `document.write(data)` writes the new content, and `document.close()` closes the document, rendering the new content.

Summary

- This JavaScript code captures the form submission event for the ASCII Art Generator.
- It prevents the default form submission, extracts the form data, and sends it to the server using the Fetch API.
- The server's response is then used to update the page content dynamically without a full page reload, ensuring a smooth user experience.

styles.css

This CSS file defines the styling for the ASCII Art Generator application. It includes styles for various HTML elements, ensuring a consistent and visually appealing layout. Here's a detailed breakdown of the styles defined in this file:

General Styles

```
body {
    font-family: Arial, sans-serif;
    background-color: #f8f9fa;
}
```

- Sets the font family for the entire document to Arial, with sans-serif as a fallback.
- Sets the background color of the entire document to a light gray (`#f8f9fa`).

```
a {
    text-decoration: none;
}
```

- Removes the underline from all anchor (`<a>`) elements.

Heading Styles

```
h1 {
    color: #007BFF;
    text-align: center;
    font-size: 40px;
}
```

- Sets the text color of all level 1 headings (`<h1>`) to a shade of blue (`#007BFF`).
- Centers the text horizontally.
- Sets the font size to 40 pixels.

Result Container Styles

```
.result {
    width: 800px;
    margin: auto;
}

.result h1 {
    text-align: center;
}
```

- **`.result`**: Defines a container with a fixed width of 800 pixels, centered horizontally using `margin: auto;`.
- **`.result h1`**: Ensures that any `<h1>` within the `.result` container is centered.

Form Styles

```
#asciiForm {
    background-color: #fff;
    width: 500px;
    height: 300px;
    border-radius: 4px;
    box-shadow: 0px 4px 8px rgba(0, 0, 0, 0.2);
}
```

- **`#asciiForm`**: Styles the form container with a white background, a fixed width of 500 pixels, a fixed height of 300 pixels, rounded corners (4px radius), and a subtle shadow.

```
form {
    margin: 20px auto;
    padding: 40px 20px;
    display: flex;
    flex-direction: column;
}
```

- Centers the form with `margin: 20px auto`.
- Adds padding around the form contents.
- Uses flexbox to arrange form elements in a vertical column.

Textarea Styles

```
textarea {
    width: 99%;
    height: 50px;
    border-radius: 4px;
}
```

- Sets the width of all `<textarea>` elements to 99% of the parent container's width.
- Sets the height to 50 pixels.
- Adds rounded corners with a 4-pixel radius.

Submit Button Styles

```
input[type="submit"] {
    padding: 15px 20px;
    background-color: #007BFF;
    color: white;
    border: none;
    cursor: pointer;
    font-size: 18px;
    border-radius: 4px;
}

input[type="submit"]:hover {
    background-color: #0056b3;
}
```

- **`input[type="submit"]`**: Styles the submit button with padding, a blue background, white text, no border, a pointer cursor, an 18-pixel font size, and rounded corners.
- **`input[type="submit"]:hover`**: Changes the background color to a darker blue (`#0056b3`) when the button is hovered over.

Result Link Styles

```
.result a {
    display: block; /* Ensure the link takes up the full width */
    margin: 20px auto;
    width: 50%; /* Span 50% of the .result width */
    text-align: center;
    padding: 15px 20px;
    background-color: #007BFF;
    color: white;
    border: none;
    cursor: pointer;
    font-size: 18px;
    border-radius: 4px;
}

.result a:hover {
    background-color: #0056b3;
}
```

- `**.result a**`: Styles the anchor element within the `.result` container to behave like a button with similar styles to the submit button. It spans 50% of the container's width, is centered, and has padding, background color, text color, and other styles similar to the submit button.
- `**.result a:hover**`: Changes the background color to a darker blue when the link is hovered over.

Dropdown (Select) Styles

```
#banner {
    background-color: #f8f9fa;
    width: 100%;
    height: 40px;
    border-radius: 4px;
}
```

- `**#banner**`: Styles the dropdown menu with a light gray background, full width, a fixed height of 40 pixels, and rounded corners.

Preformatted Text Styles

```
pre {
    width: 100%; /* or specify a fixed width like 800px */
    max-width: 100%;
    background-color: #fff;
    padding: 20px;
    border-radius: 4px;
    margin: 20px auto;
    box-shadow: 0px 4px 8px rgba(0, 0, 0, 0.2);
    overflow-x: auto;
}
```

- `**pre**`: Styles the `<pre>` element, used for displaying preformatted text (ASCII art). It takes the full width of the parent container, has a white background, padding, rounded corners, a shadow, and horizontal overflow handling.

Summary

- The CSS file styles the overall layout, headings, form, textarea, submit button, links, dropdown, and preformatted text to create a consistent and visually appealing design for the ASCII Art Generator application.
- It ensures elements are centered, have appropriate padding and margins, and uses colors and shadows to enhance the visual hierarchy and user experience.

Loadascii.go

This Go file is part of the `utils` package and contains a function to load ASCII characters from a file into a map. The map is used to generate ASCII art. Here's a detailed breakdown of its components and functionality:

Package and Imports

```
package utils

import (
	"bufio"
	"fmt"
	"os"
)
```

- `**package utils**`: Declares that this file belongs to the `utils` package.
- `**import ( "bufio" "fmt" "os" )**`: Imports the necessary packages:
  - `**"bufio"**`: Provides buffered I/O operations.
  - `**"fmt"**`: Implements formatted I/O.
  - `**"os"**`: Provides a platform-independent interface to operating system functionality.

Constants

```
const (
	expectedAsciiChars = 95
	expectedlines = 8
)
```

- `**expectedAsciiChars = 95**`: Defines the expected number of ASCII characters (95 printable characters from space to tilde).
- `**expectedlines = 8**`: Defines the expected number of lines per ASCII character.

LoadAsciiChars Function

```
func LoadAsciiChars(filename string) (map[byte][]string, error) {
```

- `**func LoadAsciiChars(filename string) (map[byte][]string, error)**`: Declares a function named `LoadAsciiChars` which takes a `filename` as a parameter and returns a map of ASCII characters and an error.

Opening the File

```
file, err := os.Open(filename)
if err != nil {
	if os.IsNotExist(err) {
		return nil, fmt.Errorf("file '%s' not found", filename)
	} else {
		return nil, fmt.Errorf("opening file: %w", err)
	}
}
defer file.Close()
```

- `**file, err := os.Open(filename)**`: Tries to open the specified file.
- `**if err != nil { ... }**`: Checks if there was an error opening the file.
  - `**if os.IsNotExist(err) { ... }**`: If the file does not exist, it returns an appropriate error message.
  - `**else { return nil, fmt.Errorf("opening file: %w", err) }**`: If there's another error, it returns a formatted error message.

- `**defer file.Close()**`: Ensures that the file is closed when the function returns.

Initializing Variables

```
asciiChars := make(map[byte][]string)
scanner := bufio.NewScanner(file)
currentChar := byte(' ') // or 32
count := 0
scanner.Scan()
```

- `**asciiChars := make(map[byte][]string)**`: Initializes a map to hold the ASCII characters and their corresponding string slices.
- `**scanner := bufio.NewScanner(file)**`: Creates a new scanner to read the file line by line.
- `**currentChar := byte(' ')**`: Sets the initial character to the space character (ASCII 32).
- `**count := 0**`: Initializes a counter to keep track of the number of lines read for the current character.
- `**scanner.Scan()**`: Skips the first line, assuming it's not part of the ASCII art data.

Reading and Storing ASCII Characters

```
for scanner.Scan() {
	line := scanner.Text()
	if count != expectedlines {
		asciiChars[currentChar] = append(asciiChars[currentChar], line)
		count++
	} else {
		currentChar++
		count = 0
	}
}
```

- `**for scanner.Scan() { line := scanner.Text() ... }**`: Reads each line of the file.
- `**if count != expectedlines { ... } else { ... }**`: Checks if the current character's line count is less than `expectedlines`.
  - `**asciiChars[currentChar] = append(asciiChars[currentChar], line)**`: Adds the current line to the current character's slice.
  - `**count++**`: Increments the line count.
  - `**else { currentChar++ count = 0 }**`: If `expectedlines` lines have been read for the current character, move to the next character and reset the line count.

Error Handling and Validation

```
if len(asciiChars) == 0 {
	return nil, fmt.Errorf("file '%s' is empty", filename)
}
```

```
if err := scanner.Err(); err != nil {
        return nil, fmt.Errorf("error reading file: %w", err)
}
if len(asciiChars) != expectedAsciiChars {
        return nil, fmt.Errorf("expected %d but got %d ASCII CHARACTERS. Ensure you use the
correct number of ASCII CHARACTERS", expectedAsciiChars, len(asciiChars))
}
if count != expectedlines {
        return nil, fmt.Errorf("expected %d lines but got %d lines. Ensure the file has the correct
number of lines per character", expectedlines, count)
}
```

- `**if len(asciiChars) == 0 { ... }**`: Checks if the map is empty and returns an error if so.
- `**if err := scanner.Err(); err != nil { ... }**`: Checks for any scanning errors and returns an appropriate error message.
- `**if len(asciiChars) != expectedAsciiChars { ... }**`: Validates the number of ASCII characters read. If it doesn't match `expectedAsciiChars`, it returns an error.
- `**if count != expectedlines { ... }**`: Validates the number of lines per character. If the last character doesn't have `expectedlines` lines, it returns an error.

Returning the Result

```
return asciiChars, nil
```

- Returns the populated map of ASCII characters and a nil error.

Summary

- The `LoadAsciiChars` function reads an ASCII art file and maps each character to its corresponding lines.
- It handles file opening, reading, and error checking.
- The function ensures the file contains the expected number of ASCII characters and lines per character, returning an error if any validation fails.
- If successful, it returns a map of ASCII characters, each represented by a slice of strings.

loadascii_test.go

This Go file is a test suite for the `LoadAsciiChars` function in the `utils` package. It uses the `testing` package to define and run unit tests. Here's a detailed breakdown of its components and functionality:

Package and Imports

```
package utils

import (
        "io/ioutil"
        "os"
        "reflect"
        "testing"
)
```

- `**package utils**`: Declares that this file belongs to the `utils` package.
- `**import ( "io/ioutil" "os" "reflect" "testing" )**`: Imports the necessary packages:
  - `**"io/ioutil"**`: Provides functions for I/O operations, including file read/write.
  - `**"os"**`: Provides functions for interacting with the operating system.
  - `**"reflect"**`: Provides functions for runtime reflection, used here to compare data structures.
  - `**"testing"**`: Provides support for automated testing of Go packages.

Test Function: TestLoadAsciiChars

```
func TestLoadAsciiChars(t *testing.T) {
```

- `**func TestLoadAsciiChars(t *testing.T)**`: Declares a test function for `LoadAsciiChars`. The `t *testing.T` parameter provides methods for reporting and logging errors during test execution.

Setup: Creating an Empty Test File

```
// Create an empty standard.txt file for testing
if err := ioutil.WriteFile("standard.txt", nil, 0o644); err != nil {
        t.Fatalf("Error creating test file: %v", err)
}
defer os.Remove("standard.txt")
```

- Creates an empty file named `standard.txt` with appropriate permissions.
- `**if err := ioutil.WriteFile("standard.txt", nil, 0o644); err != nil { ... }**`: Checks for errors during file creation and logs a fatal error if any.
- `**defer os.Remove("standard.txt")**`: Ensures that the test file is removed after the test function completes, regardless of whether it passes or fails.

Test Cases Definition

```
tests := []struct {
        name        string
        filename    string
        expectedMap  map[byte][]string
        expectedError string
}{
        {
                name:         "Non-existent file",
                filename:     "nonexistent.txt",
                expectedMap:  nil,
                expectedError: "file 'nonexistent.txt' not found",
        },
        {
                name:         "Empty file",
                filename:     "standard.txt",
                expectedMap:  nil,
                expectedError: "file 'standard.txt' is empty",
        },
}
```

- Defines a slice of test cases. Each test case is a struct with the following fields:
  - `**name**`: A string describing the test case.
  - `**filename**`: The name of the file to be tested.
  - `**expectedMap**`: The expected output map from `LoadAsciiChars`.
- `**expectedError**`: The expected error message from `LoadAsciiChars`.

Running the Test Cases

```
for _, tt := range tests {
	t.Run(tt.name, func(t *testing.T) {
		asciiChars, err := LoadAsciiChars(tt.filename)

		if err != nil && err.Error() != tt.expectedError {
			t.Errorf("Got error '%v', expected '%v'", err.Error(), tt.expectedError)
		}

		if !reflect.DeepEqual(asciiChars, tt.expectedMap) {
			t.Errorf("Got map '%v', expected '%v'", asciiChars, tt.expectedMap)
		}
	})
}
```

- Iterates over each test case.
- **`t.Run(tt.name, func(t *testing.T) { ... })`**: Runs each test case in a separate subtest identified by `tt.name`.
- **`asciiChars, err := LoadAsciiChars(tt.filename)`**: Calls the `LoadAsciiChars` function with the test case's filename.
- **`if err != nil && err.Error() != tt.expectedError { ... }`**: Checks if the error returned matches the expected error. If not, it logs an error.
- **`if !reflect.DeepEqual(asciiChars, tt.expectedMap) { ... }`**: Compares the returned map with the expected map using `reflect.DeepEqual`. If they are not equal, it logs an error.

Summary

- The `TestLoadAsciiChars` function tests the `LoadAsciiChars` function for different scenarios.
- It sets up an empty file for testing and defines two test cases:
  - One for a non-existent file, expecting a "file not found" error.
  - One for an empty file, expecting a "file is empty" error.
- Each test case is run using `t.Run` to isolate subtests and provide clear output.
- The function uses `reflect.DeepEqual` to compare the actual and expected maps, ensuring accurate and deep comparison of the data structures.

printascii.go

This Go file is part of the `utils` package and contains functions for handling ASCII art generation and processing. Let's break down each component of the file:

Package and Imports

```
package utils

import (
	"fmt"
	"strings"
)
```

- **`package utils`**: Declares that this file belongs to the `utils` package.

- `**import ( "fmt" "strings" )**`: Imports the necessary packages:
  - `**"fmt"**`: Provides formatted I/O functions.
  - `**"strings"**`: Provides functions for manipulating strings.

ReplaceSpecChars Function

```
// ReplaceSpecChars replaces escape sequences with their corresponding special characters.
func ReplaceSpecChars(s string) string {
	replace := strings.NewReplacer(
		"\\r", "\r", // Replace '\r' with carriage return
		"\\b", "\b", // Replace '\b' with backspace
		"\\t", "    ", // Replace '\t' with four spaces
		"\\f", "\f", // Replace '\f' with form feed
		"\\a", "\a", // Replace '\a' with alert (bell)
		"\\v", "\v", // Replace '\v' with vertical tab
	)
	return replace.Replace(s)
}
```

- `**ReplaceSpecChars(s string) string**`: Replaces escape sequences in a string with their corresponding special characters.
- Uses `strings.NewReplacer` to define the replacements and `replace.Replace(s)` to apply them.

PrintAsciiArt Function

```
// PrintAsciiArt prints the given text as ASCII art using the provided map of characters.
func PrintAsciiArt(text string, asciiChars map[byte][]string) {
	text = ReplaceSpecChars(text)
	// Check if any character is outside the ASCII range (32-127)
	for _, char := range text {
		if char > 127 || char < 32 {
			fmt.Printf("Error: Character %q is not accepted\n", char)
			return
		}
	}

	// Print each line of the ASCII art
	for i := 0; i < 8; i++ {
		PrintLine(text, asciiChars, i)
		fmt.Println()
	}
}
```

- `**PrintAsciiArt(text string, asciiChars map[byte][]string)**`: Prints the given text as ASCII art using the provided map of characters.
- Calls `ReplaceSpecChars` to handle escape sequences.
- Checks if any character in the text is outside the ASCII range (32-127) and prints an error if found.
- Prints each line of the ASCII art using the `PrintLine` function.

PrintLine Function

```
// PrintLine prints a single line of the ASCII art for the given text.
```

```
func PrintLine(text string, asciiChars map[byte][]string, line int) {
	for _, char := range text {
		if char == '\n' {
			fmt.Println()
		} else {
			fmt.Print(asciiChars[byte(char)][line]) // Print the ASCII representation of
the character
		}
	}
}
```

- `**PrintLine(text string, asciiChars map[byte][]string, line int)**`: Prints a single line of the ASCII art for the given text.
- Iterates through each character in the text and prints the corresponding line from the ASCII art map.

ProcessArguments Function

```
// ProcessArguments processes the input arguments and prints ASCII art for each argument.
func ProcessArguments(args string, asciiChars map[byte][]string) {
	arguments := strings.Split(args, "\\n") // Split arguments by '\n'
	// Handle empty input or newlines
	countSpaces := 0
	for _, arg := range arguments {
		if arg == "" {
			countSpaces++
			if countSpaces < len(arguments) {
				fmt.Println() // Print a newline for each empty argument except the
last one
			}
		} else {
			PrintAsciiArt(arg, asciiChars) // Print ASCII art for the non-empty argument
		}
	}
}
```

- `**ProcessArguments(args string, asciiChars map[byte][]string)**`: Processes the input arguments and prints ASCII art for each argument.
- Splits the input arguments by `\n` and handles each argument individually.
- If an argument is empty, it prints a newline. Otherwise, it calls `PrintAsciiArt` to print the ASCII art for the argument.

GenerateAsciiArt Function

```
func GenerateAsciiArt(text string, asciiChars map[byte][]string) string {
	var result strings.Builder
	text = ReplaceSpecChars(text)
```

```
        for _, line := range strings.Split(text, "\n") {
                for i := 0; i < 8; i++ {
                        for _, char := range line {
                                result.WriteString(asciiChars[byte(char)][i])
                        }
                        result.WriteString("\n")
                }
                result.WriteString("\n")
        }
        return result.String()
}
```

- `**GenerateAsciiArt(text string, asciiChars map[byte][]string) string**`: Generates ASCII art for the given text and returns it as a string.
- Uses `strings.Builder` for efficient string concatenation.
- Calls `ReplaceSpecChars` to handle escape sequences.
- Splits the text by `\n` and processes each line to generate the ASCII art.
- Appends each line of ASCII art to the `strings.Builder` and returns the final string.

Summary

- `**ReplaceSpecChars**`: Handles escape sequences in a string.
- `**PrintAsciiArt**`: Prints the given text as ASCII art, ensuring characters are within the ASCII range.
- `**PrintLine**`: Prints a specific line of the ASCII art for the given text.
- `**ProcessArguments**`: Processes and prints ASCII art for multiple arguments, handling newlines and empty inputs.
- `**GenerateAsciiArt**`: Generates and returns ASCII art as a string for the given text.

These functions together provide a comprehensive set of utilities for handling ASCII art generation and printing.

printascii_test.go

This Go file contains a unit test for the `PrintAsciiArt` function from the `utils` package, along with a helper function `CaptureOutput` to capture stdout during testing. Let's break down each part of the file:

Package and Imports

package utils

```
import (
        "bytes"
        "io"
        "os"
        "path/filepath"
        "testing"
)
```

- `**package utils**`: Indicates that this file belongs to the `utils` package.
- Imports necessary packages:
  - `**"bytes"**`: Provides functions for manipulating byte slices.
  - `**"io"**`: Provides basic interfaces for I/O operations.
  - `**"os"**`: Provides functions for operating system functionalities.
  - `**"path/filepath"**`: Provides functions to manipulate file paths.
  - `**"testing"**`: Provides support for automated testing in Go.

Test Function: Test_PrintAsciiArt

```
func Test_PrintAsciiArt(t *testing.T) {
	// Compute the correct path to the standard.txt file
	basePath, err := os.Getwd()
	if err != nil {
		t.Fatalf("Failed to get working directory: %v", err)
	}
	asciiFilePath := filepath.Join(basePath, "..", "standard.txt")

	// Load the ASCII characters from the sample banner file for testing
	asciiChars, err := LoadAsciiChars(asciiFilePath)
	if err != nil {
		t.Fatalf("Failed to load ASCII characters: %v", err)
	}

	tests := []struct {
		name   string
		input  string
		output string
	}{
		{
			name:  "Simple Text",
			input: "Hello",
			output: `_    _          _   _
| |  | |        | | | |
| |__| |  ___   | | | |  ___
|  __  | / _ \  | | | | / _ \
| |  | ||  __/  | | | || (_) |
|_|  |_| \___|  |_| |_| \___/


`,
		},
	}
	for _, tt := range tests {
		t.Run(tt.name, func(t *testing.T) {
			output := CaptureOutput(func() {
				PrintAsciiArt(tt.input, asciiChars)
			})

			if output != tt.output {
				t.Errorf("Expected output:\n%s\nGot:\n%s", tt.output, output)
			}
```

```
            })
        }
}
```

Explanation:

1. **Setup**:
   - **Compute the path to `standard.txt`**: Uses `os.Getwd()` to get the current working directory and constructs the path to `standard.txt` relative to it (`asciiFilePath`).
   - **Load ASCII characters**: Calls `LoadAsciiChars` to load ASCII characters from `standard.txt` into `asciiChars` for testing.

2. **Tests**:
   - Defines a slice of test cases (`tests`) where each test case contains:
     - `**name**`: Descriptive name for the test case.
     - `**input**`: Input string to be passed to `PrintAsciiArt`.
     - `**output**`: Expected output when `PrintAsciiArt` is called with `input`.

3. **Iteration and Execution**:
   - Iterates over each test case (`tt`) in `tests`.
   - Uses `t.Run(tt.name, func(t *testing.T) { ... })` to run each test case as a subtest with a descriptive name (`tt.name`).
   - **Captures Output**: Calls `CaptureOutput` to capture the output of `PrintAsciiArt(tt.input, asciiChars)`.
   - **Assertion**: Compares the captured output (`output`) with the expected output (`tt.output`). If they don't match, logs an error (`t.Errorf`).

Helper Function: CaptureOutput

```
// Helper function to capture stdout
func CaptureOutput(f func()) string {
        old := os.Stdout
        r, w, _ := os.Pipe()
        os.Stdout = w

        f()

        w.Close()
        os.Stdout = old
        var buf bytes.Buffer
        io.Copy(&buf, r)
        return buf.String()
}
```

- **CaptureOutput(f func()) string**: Helper function to capture stdout from a function `f`.
- **Steps**:
  - **Redirect stdout**: Saves the current `os.Stdout` to `old` and creates a new pipe (`r`, `w`) for capturing stdout.
  - **Set `os.Stdout` to `w`**: Redirects stdout to the write end of the pipe.
  - **Execute `f()`**: Executes the provided function `f`, which in this case, is `PrintAsciiArt`.
  - **Close `w`**: Closes the write end of the pipe to flush the output.

  - **Restore `os.Stdout`**: Resets `os.Stdout` to its original value (`old`).
  - **Read from pipe**: Copies the contents of the pipe (`r`) to a `bytes.Buffer` (`buf`).
  - **Return captured output**: Converts the `bytes.Buffer` to a string and returns it (`buf.String()`).

Summary

- **Test_PrintAsciiArt**: Tests the `PrintAsciiArt` function using predefined test cases.
  - Loads ASCII characters from a file (`standard.txt`).
  - Executes `PrintAsciiArt` with various inputs (`tt.input`) and compares the output to expected results (`tt.output`).
- **CaptureOutput**: Helper function to capture stdout during testing, allowing comparison against expected output.

This setup ensures that `PrintAsciiArt` functions correctly for different inputs as defined in the test cases, and `CaptureOutput` facilitates checking the output without printing to the console during testing.