Mark Summerfield

Programming in Go

Creating Applications for the 21st Century

Developer's Library

Programming in Go

program—for example, stopping all goroutines and closing any open files—and returns its argument to the operating system.

If an `.m3u` file has been specified we attempt to read the entire file using the `ioutil.ReadFile()` function. This function returns all the file's bytes (as a `[]byte`) and an `error` which will be `nil` if the file was read without incident. If a problem occurred (e.g., the file doesn't exist or is unreadable), we use the `log.Fatal()` function to output the error to the console (actually to `os.Stderr`), and to terminate the program with an exit code of 1.

If the file is successfully read we convert its raw bytes to a `string`—this assumes that the bytes represent 7-bit ASCII or UTF-8 Unicode—and immediately pass the string to a custom `readM3uPlaylist()` function for parsing. The function returns a slice of `Songs` (i.e., a `[]Song`). We then write the song data using a custom `writePlsPlaylist()` function.

```
type Song struct {
    Title    string
    Filename string
    Seconds  int
}
```

Here we have defined a custom `Song` type using a `struct` (§6.4, ➤ 275) to provide convenient file-format-independent storage for the information about each song.

```
func readM3uPlaylist(data string) (songs []Song) {
    var song Song
    for _, line := range strings.Split(data, "\n") {
        line = strings.TrimSpace(line)
        if line == "" || strings.HasPrefix(line, "#EXTM3U") {
            continue
        }
        if strings.HasPrefix(line, "#EXTINF:") {
            song.Title, song.Seconds = parseExtinfLine(line)
        } else {
            song.Filename = strings.Map(mapPlatformDirSeparator, line)
        }
        if song.Filename != "" && song.Title != "" && song.Seconds != 0 {
            songs = append(songs, song)
            song = Song{}
        }
    }
    return songs
}
```

This function accepts the entire contents of an `.m3u` file as a single `string` and returns a slice of all the songs it is able to parse from the string. It begins by declaring an empty `Song` variable called `song`. Thanks to Go's practice of always initializing things to their zero value, `song`'s initial contents are two empty strings and a `Song.Seconds` value of 0.

At the heart of the function is a `for ... range` loop (§5.3, ➤ 203). The `strings.Split()` function is used to split the single string that holds the entire `.m3u` file's data into separate lines, and the `for` loop iterates over each of these lines. If a line is empty or is the first line (i.e., starts with the string literal `"#EXTM3U"`), the `continue` statement is reached; this simply passes control back to the `for` loop to force the next iteration—or the end of the loop if there are no more lines.

If the line begins with the `"#EXTINF:"` string literal, the line is passed to a custom `parseExtinfLine()` function for parsing: This function returns a `string` and an `int` which are immediately assigned to the current `song`'s `Song.Title` and `Song.Seconds` fields. Otherwise, it is assumed that the line holds the filename (including the path) of the current `song`.

Rather than storing the filename as is, the `strings.Map()` function is called with a custom `mapPlatformDirSeparator()` function to convert directory separators into those native for the platform the program is running on, and the resultant string is stored as the current `song`'s `Song.Filename`. The `strings.Map()` function is passed a mapping function with signature `func(rune) rune` and a `string`. For every character in the string the mapping function is called with the character replaced by the character returned by the passed-in function—which may be the same as the original one, of course. As usual with Go, a character is a `rune` whose value is the character's Unicode code point.

If the current song's filename and title are both nonempty, and if the song's duration isn't zero, the current song is appended to the `songs` return value (of type `[]Song`) and the current `song` is set to its zero value (two empty strings and `0`) by assigning an empty `Song` to it.

```
func parseExtinfLine(line string) (title string, seconds int) {
    if i := strings.IndexAny(line, "-0123456789"); i > -1 {
        const separator = ","
        line = line[i:]
        if j := strings.Index(line, separator); j > -1 {
            title = line[j+len(separator):]
            var err error
            if seconds, err = strconv.Atoi(line[:j]); err != nil {
                log.Printf("failed to read the duration for '%s': %v\n",
                    title, err)
                seconds = -1
            }
```

```
        }
    }
    return title, seconds
}
```

This function is used to parse lines of the form: `#EXTINF:`*`duration`*`,`*`title`* and where the *`duration`* is expected to be an integer, either `-1` or greater than zero.

The `strings.IndexAny()` function is used to find the position of the first digit or the minus sign. An index position of `-1` means not found; any other value is the index position of the first occurrence of any of the characters in the string given as the `strings.IndexAny()` function's second argument, in which case variable `i` holds the position of the first digit of the duration (or of `-`).

Once we know where the digits begin we slice the line to start at the digits. This effectively discards the `"#EXTINF:"` that was at the start of the string, so now the line has the form: *`duration`*`,`*`title`*.

The second `if` statement uses the `strings.Index()` function to get the index position of the first occurrence of the `","` string in the line—or `-1` if there is no such occurrence.

The title is the text from after the comma to the end of the line. To slice from after the comma we need the comma's starting position (`j`) and must add to this the number of bytes the comma occupies (`len(separator)`). Of course, we know that a comma is a 7-bit ASCII character and so has a length of one, but the approach shown here will work with any Unicode character, no matter how many bytes are used to represent it.

The duration is the number whose digits go from the start of the line up to but excluding the `j`-th byte (where the comma is). We convert the number into an `int` using the `strconv.Atoi()` function—and if the conversion fails we simply set the duration to `-1` which is an acceptable "unknown duration" value, and log the problem so that the user is aware of it.

```
func mapPlatformDirSeparator(char rune) rune {
    if char == '/' || char == '\\' {
        return filepath.Separator
    }
    return char
}
```

This function is called by the `strings.Map()` function (inside the `readM3uPlaylist()` function) for every character in a filename. It replaces any directory separator with the platform-specific directory separator. And any other character is returned unchanged.

Like most cross-platform programming languages and libraries, Go uses Unix-style directory separators internally on all platforms, even on Windows. However, for user-visible output and for human-readable data files, we prefer to use the platform-specific directory separator. To achieve this we can use the `filepath.Separator` constant which holds the `/` character on Unix-like systems and the `\` character on Windows.

In this example we don't know whether the paths we are reading use forward slashes or backslashes, so we have had to cater for both. However, if we know for sure that a path uses forward slashes we can use the `filepath.FromSlash()` function on it: This will return the path unchanged on Unix-like systems, but will replace forward slashes with backslashes on Windows.

```
func writePlsPlaylist(songs []Song) {
    fmt.Println("[playlist]")
    for i, song := range songs {
        i++
        fmt.Printf("File%d=%s\n", i, song.Filename)
        fmt.Printf("Title%d=%s\n", i, song.Title)
        fmt.Printf("Length%d=%d\n", i, song.Seconds)
    }
    fmt.Printf("NumberOfEntries=%d\nVersion=2\n", len(songs))
}
```

This function writes out the `songs` data in `.pls` format. It writes the data to `os.Stdout` (i.e., to the console), so file redirection must be used to get the output into a file.

The function begins by writing the section header (`"[playlist]"`), and then for every song it writes the song's filename, title, and duration in seconds, each on their own lines. Since each key must be unique a number is appended to each one, starting from 1. And at the end the two items of metadata are written.

3.8. Exercises

There are two exercises for this chapter, the first involving the modification of an existing command-line program, and the second requiring the creation of a web application (optionally) from scratch.

1. The previous section's `m3u2pls` program does a decent job of converting `.m3u` playlist files into `.pls` format. But what would make the program much more useful is if it could also perform the reverse conversion, from `.pls` format to `.m3u` format. For this exercise copy the `m3u2pls` directory to, say, `my_playlist` and create a new program called `playlist` that has the required functionality. Its usage message should be `usage: playlist <file.[pls|m3u]>`.

If the program is called with an `.m3u` file it should do exactly what the `m3u2pls` program does: Write the file's data in `.pls` format to the console. But if the program is called with a `.pls` file it should write the file's data in `.m3u` format, again to the console. The new functionality will require about 50 new lines of code. A straightforward solution is provided in the file `playlist/playlist.go`.

2. Data cleaning, matching, and mining applications that involve people's names can often produce better results by matching names by the way they sound rather than by how they are spelled. Many algorithms for name matching English language names are available, but the oldest and simplest is the Soundex algorithm.

   The classic Soundex algorithm produces a soundex value of a capital letter followed by three digits. For example, the names "Robert" and "Rupert" both have the same soundex value of "R163" according to most Soundex algorithms. However, the names "Ashcroft" and "Ashcraft" have a soundex value of "A226" according to some Soundex algorithms (including the one in the exercise solution), but "A261" according to others.

   The exercise is to write a web application that supports two web pages. The first page (with path `/`) should present a simple form through which the user can enter one or more names to see their soundex values—this is illustrated in Figure 3.3's left-hand screenshot. The second page (with path `/test`) should execute the application's `soundex()` function on a list of strings and compare each result to what we would expect—this is illustrated in Figure 3.3's right-hand screenshot.

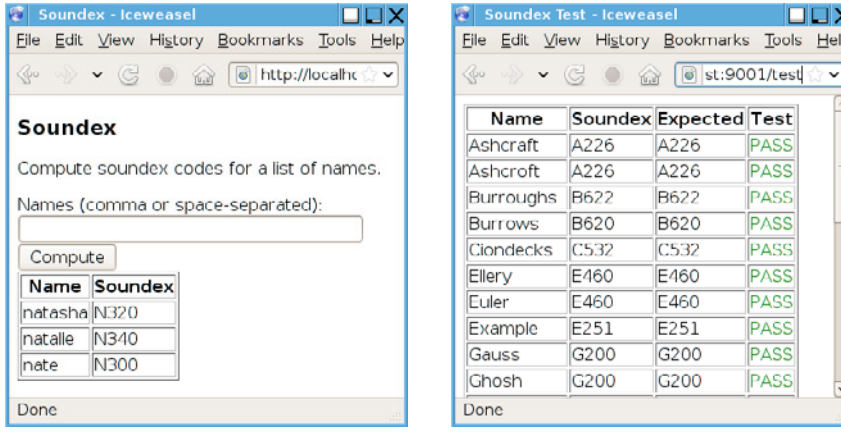

Figure 3.3 *The Soundex application on Linux*

Readers who would like a jump-start could copy one of the other web applications (`statistics`, `statistics_ans`, `quadratic_ans1`, `quadratic_ans2`) to

get the skeleton of the application up and running, and then just focus on the soundex and test page functionality.

A solution is in the file `soundex/soundex.go` and is about 150 lines; the `soundex()` function itself is 20 lines although it does rely on an `[]int` that maps capital letters to digits in a slightly subtle way. The solution's algorithm is based on the Python implementation shown on the Rosetta Code web site (`rosettacode.org/wiki/Soundex`) which produces slightly different results to the Go implementation shown on that site and from the one shown on Wikipedia (`en.wikipedia.org/wiki/Soundex`). The test data is in the file `soundex/soundex-test-data.txt`.

Naturally, readers are free to implement whichever version of the algorithm they prefer—or even implement a more advanced algorithm such as one of the Metaphone algorithms—and simply adjust the tests to match.