Article

Compressing and saving a string to the file system

Compress the contents of a Unicode string and store the result on the file system.

Overview

In this article, you'll learn how to use AppleArchive to compress a `String` structure, and write the compressed data to a file in macOS.

The code below compresses a string using the `Algorithm.lzfse` algorithm, and stores the result as `lorem.txt` in an AppleArchive file named `lorem.aar`. The code writes `lorem.aar` to the user's `Downloads` directory.

Create the source string

Create a string that contains the data the code compresses.

In a real-world app, you'll most likely generate the string from a source such as user input. For this example, specify the string as a literal:

```
static var loremString = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam fermentum vesti
[...]
"""
```

Specify the compressed file path

Create a `FilePath` structure that specifies the file name and location of the AppleArchive file that stores the compressed data. You must add read and write file access to the Downloads folder in

the Signing and Capabilities pane. To learn more about configuring the App Sandbox, see App Sandbox.

The following code creates a file path to `lorem.aar`:

```
let archiveFileName = "lorem.aar"

let archiveFilePath: FilePath = {
    guard let downloadURL = FileManager.default.urls(for: .downloadsDirectory,
                                                     in: .userDomainMask).first,
          let archiveFilePath = FilePath(downloadURL.appendingPathComponent(archiveF
        fatalError("Unable to create archive file path.")
    }

    return archiveFilePath
}()
```

Create the file stream to write the compressed File

Use `fileStream(path:mode:options:permissions:)` to create the file stream that writes the compressed file to the file system. In this case, use the `writeOnly` mode. Set the options as:

`create`
: To specify that the byte stream creates the file if it doesn't already exist.

`truncate`
: To specify that if the file exists, the byte stream truncates it to zero bytes before it performs any operations.

```
guard let writeFileStream = ArchiveByteStream.fileStream(
        path: archiveFilePath,
        mode: .writeOnly,
        options: [ .create, .truncate ],
        permissions: FilePermissions(rawValue: 0o644)) else {
    return
}
defer {
    try? writeFileStream.close()
}
```

Create the compression stream

Create the compression stream, and specify the compression algorithm as `lzfse`. Specify the file-writing stream as the stream that receives the compressed data:

```
guard let compressStream = ArchiveByteStream.compressionStream(
        using: .lzfse,
        writingTo: writeFileStream) else {
    return
}
defer {
    try? compressStream.close()
}
```

Create the encoding stream

Create the encoding stream. The encoding stream encodes its data as a byte stream, and sends the encoded data to the compression stream:

```
guard let encodeStream = ArchiveStream.encodeStream(writingTo: compressStream) else
    return
}
defer {
    try? encodeStream.close()
}
```

Define the archive header

Define the header for the archive file. The header contains three fields:

The PAT field contains the file path. Specify the unarchived file name for the PAT field:

```
let header = ArchiveHeader()
header.append(.string(key: ArchiveHeader.FieldKey("PAT"),
                      value: "lorem.txt"))
```

The TYP field contains the compressed file type. Specify `regularFile` for the TYP field:

```
header.append(.uint(key: ArchiveHeader.FieldKey("TYP"),
                    value:  UInt64(ArchiveHeader.EntryType.regularFile.rawValue)))
```

The DAT field contains the compressed file payload. Specify the size of the uncompressed data, in bytes, for the DAT field:

```
header.append(.blob(key: ArchiveHeader.FieldKey("DAT"),
                    size: UInt64(loremString.utf8.count)))
```

For more information about three-letter keys, see `init(_:)`.

Finally, write the header to the encode stream:

```
do {
    try encodeStream.writeHeader(header)
} catch {
    print("Failed to write header.")
}
```

Write the string to the encode stream

Use doc://com.apple.documentation/documentation/applearchive/archivestream/3589317-writeblob to write the contents of the string as a blob to the encode stream. In turn, the encode stream writes to the compression stream and then, the compression stream writes to the file stream. Finally, the file stream writes the archive file to the file system:

```
do {
    try loremString.withUTF8 { textPtr in

        let rawBufferPointer = UnsafeRawBufferPointer(textPtr)

        try encodeStream.writeBlob(key: ArchiveHeader.FieldKey("DAT"),
                                   from: rawBufferPointer)
    }
} catch {
    print("Failed to write blob.")
}
```

On return, `lorem.aar` exists as an AppleArchive file in the user's `Downloads` directory and contains a single compressed text file, `lorem.txt`. The content of this text file is `loremString`.

See Also

Directories, Files, and Data Archives

- Compressing single files
  Compress a single file and store the result on the file system.
- Decompressing single files
  Recreate a single file from a compressed file.
- Compressing file system directories
  Compress the contents of an entire directory and store the result on the file system.
- Decompressing and extracting an archived directory
  Recreate an entire file system directory from an archive file.
- Decompressing and Parsing an Archived String
  Recreate a string from an archive file.