Article

Compressing and decompressing data with input and output filters

Compress and decompress streamed or from-memory data, using input and output filters.

Overview

The code in this article uses the Compression framework's `InputFilter` and `OutputFilter` classes to encode (compress) and decode (decompress) a string. The code writes the encoded result to a `Data` structure.

The code in this sample is useful in applications that store or transmit files, such as PDF or text, where saving or sending smaller files can improve performance and reduce storage overhead. This sample app implements *stream compression*, where it reads chunks of data from a source buffer repeatedly to compress or decompress data, and appends each chunk to a destination buffer.

Use the input and output filters API when working with data that's streamed to or from memory — for example, when reading from or writing to a file. If you're compressing and decompressing data that's held entirely in memory, consider using `compressed(using:)` and `decompressed(using:)`. These functions provide a simple API to compress and decompress data in a single step.

Create the source data

Typically, your app dynamically generates the source data that it compresses, but for this example, the source data is a hard-coded string.

```
let sourceData = """
    Lorem ipsum dolor sit amet consectetur adipiscing elit mi
    nibh ornare proin blandit diam ridiculus, faucibus mus
```

```
    dui eu vehicula nam donec dictumst sed vivamus bibendum
    aliquet efficitur. Felis imperdiet sodales dictum morbi
    vivamus augue dis duis aliquet velit ullamcorper porttitor,
    lobortis dapibus hac purus aliquam natoque iaculis blandit
    montes nunc pretium.
    """.data(using: .utf8)!
```

On return, `sourceData` contains the UTF-8 representation of the source string.

Specify the page size

`InputFilter` and `OutputFilter` instances compress and decompress pages of data. Specify the number of bytes in each page to read from or write to a stream. Smaller values allow your app to report progress or perform other tasks at higher frequencies than larger values. However, larger values allow your app to compress or decompress using fewer steps, possibly in less time.

For this example, use a page size of 128 bytes:

```
let pageSize = 128
```

Create the compression destination buffer

Create an empty mutable `Data` structure to receive the compressed data.

```
var compressedData = Data()
```

Create the output filter

Create an `OutputFilter` instance, and specify the operation as `FilterOperation.compress` and the compression algorithm as `Algorithm.lzfse`. For more information on other compression algorithms, see `compression_algorithm`.

The final initializer parameter is a closure that the instance calls as it writes each compressed block of data to `compressedData`.

```
let outputFilter: OutputFilter
do {
    outputFilter = try OutputFilter(.compress,
                                    using: .lzfse) {
        (data: Data?) -> Void in
```

```
        if let data = data {
            compressedData.append(data)
        }
    }
} catch {
    fatalError("Error occurred creating output filter: \(error.localizedDescription)
}
```

Compress the data

Iterate over the source data and call the `subdata(in:)` method to copy `pageSize` chunks to `subdata`. The `write(_:)` method compresses each chunk and uses the closure specified in the `OutputFilter` initializer to write the result to `compressedData`.

```
do {
    var index = 0
    let bufferSize = sourceData.count

    while true {
        let rangeLength = min(pageSize, bufferSize - index)

        let subdata = sourceData.subdata(in: index ..< index + rangeLength)
        index += rangeLength

        try outputFilter.write(subdata)

        if (rangeLength == 0) {
            break
        }
    }
} catch {
    fatalError("Error occurred during encoding: \(error.localizedDescription).")
}
```

On return, `compressedData` contains a compressed version of the original source data.

Create the decompression destination buffer

Create a mutable, empty `Data` structure to receive the decompressed data.

```
var decompressedData = Data()
```

Create the input filter

Create an `InputFilter` instance, and specify the operation as `FilterOperation.decompress` and the compression algorithm as `Algorithm.lzfse`.

The final initializer parameter is a closure the instance calls as it reads each compressed block of data.

```
let inputFilter: InputFilter<Data>
do {
    var index = 0
    let bufferSize = compressedData.count

    inputFilter = try InputFilter(.decompress,
                                  using: .lzfse) { (length: Int) -> Data? in
        let rangeLength = min(length, bufferSize - index)
        let subdata = compressedData.subdata(in: index ..< index + rangeLength)
        index += rangeLength

        return subdata
    }
} catch {
    fatalError("Error occurred creating input filter: \(error.localizedDescription).
}
```

Decompress the data

You iterate over the compressed data by repeatedly calling `readData(ofLength:)`, until the function returns nil. With each iteration, append the data returned by the input filter to `decompressedData`.

```
do {
    while let page = try inputFilter.readData(ofLength: pageSize) {
        decompressedData.append(page)
    }
} catch {
    fatalError("Error occurred during decoding: \(error.localizedDescription).")
}
```

Create a string from the decompressed data

Use `init(data:encoding:)` to recreate a string from the decompressed data.

```
let decompressedString = String(data: decompressedData,
                                encoding: .utf8)
```

On return, `decompressedString` contains the original text shown in Compressing and decompressing data with input and output filters.

Select input and output filters based on requirements

You're not tied to using output filters for compression and input filters for decompression. You can select the appropriate compressor-decompressor based on your app's requirements.

For example, the code below shows an `InputFilter` instance as the compressor:

```
var compressedData = Data()

do {
    var index = 0
    let bufferSize = sourceData.count

    let inputFilter = try InputFilter(.compress,
                                      using: .lzfse) { (length: Int) -> Data? in
        let rangeLength = min(length, bufferSize - index)
        let subdata = sourceData.subdata(in: index ..< index + rangeLength)
        index += rangeLength

        return subdata
    }

    while let page = try inputFilter.readData(ofLength: pageSize) {
        compressedData.append(page)
    }
} catch {
    fatalError("Error occurred during encoding: \(error.localizedDescription).")
}
```

The code below shows an `OutputFilter` instance as the decompressor:

```
var decompressedData = Data()

do {
    let outputFilter = try OutputFilter(.decompress,
                                        using: .lzfse) {(data: Data?) -> Void in
                                            if let data = data {
                                                decompressedData.append(data)
                                            }
    }

    var index = 0
    let bufferSize = compressedData.count

    while true {
        let rangeLength = min(pageSize, bufferSize - index)

        let subdata = compressedData.subdata(in: index ..< index + rangeLength)
        index += rangeLength
        try outputFilter.write(subdata)
        if (rangeLength == 0) {
            break
        }
    }
} catch {
    fatalError("Error occurred during decoding: \(error.localizedDescription).")
}
```

See Also

Compression

{} Compressing and decompressing files with stream compression

Perform compression for all files and decompression for files with supported extension types.

Compressing and decompressing data with buffer compression

Compress a string, write it to the file system, and decompress the same file using buffer compression.