QSocket: Addresses

This thread has been locked by a moderator.

36

IMPORTANT If you haven't yet read Calling BSD Sockets from Swift, do that first.

The biggest problem with calling BSD Sockets from Swift is how to represent an address. The C API works in terms of `struct sockaddr *`, which is a 'base class' with various 'subclasses', like `struct sockaddr_in *`. C lets you [1] freely cast between these, but that's harder in Swift. My solution for this — and remember that one of my key goals is to create a wrapper that's good for test projects — is to represent addresses as strings.

I start by defining a namespace for this stuff:

```
/// A namespace for helpers that work with BSD Sockets addresses.
///
/// These convert between IP address strings and the `sockaddr` pointers used by
/// the BSD Sockets API. For example, here's how you can call `connect` with an
/// IP address and string.
///
/// ```swift
/// let success = try QSockAddr.withSockAddr(address: "1.2.3.4", port: 12345) { sa, saLen in
///     connect(fd, sa, saLen) >= 0
/// }
/// ```
///
/// This example calls ``withSockAddr(address:port:_:)``, which is what you use
/// when passing an address into BSD Sockets. There's also
/// ``fromSockAddr(sa:saLen:)``, to use when getting an address back from BSD
/// Sockets.
///
/// > important: Representing addresses as strings is potentially very
/// inefficient.  For example, if you were to wrap the BSD Sockets `sendto` call
/// in this way, you would end up doing a string-to-address conversion every
/// time you sent a datagram!  However, it _is_ very convenient, making it perfect
/// for small test projects, wrapping weird low-level APIs, and so on.
///
/// Keep in mind that I rarely use BSD Sockets for _networking_ these days.
/// Apple platforms have better networking APIs; see TN3151 [Choosing the right
/// networking API][tn3151] for the details.
///
/// [tn3151]: <[TN3151: Choosing the right networking API | Apple Developer Documentation]
(https://developer.apple.com/documentation/technotes/tn3151-choosing-the-right-networking-api)>

public enum QSockAddr {
}
```

I then extend it with various helpers. The first is for calling a routine that takes a `sockaddr` input:

```
extension QSockAddr {

    /// Calls a closure with a socket address and length.
    ///
    /// Use this to pass an address in to a BSD Sockets call. For example:
    ///
    /// ```swift
    /// let success = try QSockAddr.withSockAddr(address: "1.2.3.4", port: 12345) { sa, saLen in
    ///     connect(fd, sa, saLen) >= 0
    /// }
    /// ```
    ///
    /// - Parameters:
    ///   - address: The address as a string. This can be either an IPv4 or IPv6
    ///   address, in any format accepted by `getaddrinfo` when the
    ///   `AI_NUMERICHOST` flag is set.
    ///   - port: The port number.
    ///   - body: A closure to call with the corresponding `sockaddr` pointer
    ///   and length.
    /// - Returns: The value returned by that closure.

    public static func withSockAddr<ReturnType>(
        address: String,
        port: UInt16,
        _ body: (_ sa: UnsafePointer<sockaddr>, _ saLen: socklen_t) throws -> ReturnType
    ) throws -> ReturnType {
        var addrList: UnsafeMutablePointer<addrinfo>? = nil
        var hints = addrinfo()
        hints.ai_flags = AI_NUMERICHOST | AI_NUMERICSERV
        let err = getaddrinfo(address, "\(port)", &hints, &addrList)
        guard err == 0 else { throw QSockAddr.NetDBError(code: err) }
        guard let addr = addrList else { throw QSockAddr.NetDBError(code: EAI_NODATA) }
        defer { freeaddrinfo(addrList) }
        return try body(addr.pointee.ai_addr, addr.pointee.ai_addrlen)
    }
}
```

The second is for calling a routine that returns a `sockaddr`:

```
extension QSockAddr {

    /// Creates an address by calling a closure to fill in a socket address and
    /// length.
    ///
    /// Use this to get an address back from a BSD Sockets call. For example:
    ///
    /// ```swift
    /// let peer = try QSockAddr.fromSockAddr() { sa, saLen in
    ///     getpeername(fd, sa, &saLen) >= 0
    /// }
    /// guard peer.result else { … something went wrong … }
    /// … use peer.address and peer.port …
    /// ```
    ///
    /// - Parameter body: The closure to call. It passes this a mutable pointer
    /// to a `sockaddr` and an `inout` length. The closure is expected to
    /// populate that memory with an IPv4 or IPv6 address, or throw an error.
    /// - Returns: A tuple containing the closure result, the address string,
    /// and the port.

    public static func fromSockAddr<ReturnType>(_ body: (_ sa: UnsafeMutablePointer<sockaddr>, _ saLen: inout socklen_t) throws ->
ReturnType) throws -> (result: ReturnType, address: String, port: UInt16) {
        var ss = sockaddr_storage()
        var saLen = socklen_t(MemoryLayout<sockaddr_storage>.size)
        return try withUnsafeMutablePointer(to: &ss) {
            try $0.withMemoryRebound(to: sockaddr.self, capacity: 1) { sa in
                let result = try body(sa, &saLen)
                let (address, port) = try fromSockAddr(sa: sa, saLen: saLen)
                return (result, address, port)
            }
        }
    }
}
```

And the third is for when you start with a `sockaddr` pointer:

```
extension QSockAddr {

    /// Creates an address from an address pointer and length.
    ///
    /// Use this when you have an existing `sockaddr` pointer, for example, when
    /// working with `getifaddrs`.
    ///
    /// - Parameters:
    ///   - sa: The address pointer
    ///   - saLen: The address length.
    /// - Returns: A tuple containing the address string, and the port.

    public static func fromSockAddr(sa: UnsafeMutablePointer<sockaddr>, saLen: socklen_t) throws -> (address: String, port: UInt16)
{
        var host = [CChar](repeating: 0, count: Int(NI_MAXHOST))
        var serv = [CChar](repeating: 0, count: Int(NI_MAXSERV))
        let err = getnameinfo(sa, saLen, &host, socklen_t(host.count), &serv, socklen_t(serv.count), NI_NUMERICHOST |
NI_NUMERICSERV)
        guard err == 0 else { throw QSockAddr.NetDBError(code: err) }
        guard let port = UInt16(String(cString: serv)) else { throw QSockAddr.NetDBError(code: EAI_SERVICE) }
        return (String(cString: host), port)
    }
}
```

Finally, these routines rely on this error type:

```
extension QSockAddr {

    /// Wraps an error coming from the DNS subsytem.
    ///
    /// The code values correspond to `EAI_***` in `<netdb.h>`.

    struct NetDBError: Error {
        var code: CInt
    }
}
```

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

[1] Or does it? There's some debate as to whether the BSD Sockets API is legal C (-:

Network

Reply

Posted 5 days ago by eskimo

Add a Comment