Networking and Streams

Julia provides a rich interface to deal with streaming I/O objects such as terminals, pipes and TCP sockets. This interface, though asynchronous at the system level, is presented in a synchronous manner to the programmer and it is usually unnecessary to think about the underlying asynchronous operation. This is achieved by making heavy use of Julia cooperative threading (coroutine) functionality.

Basic Stream I/O

All Julia streams expose at least a `read` and a `write` method, taking the stream as their first argument, e.g.:

```
julia> write(stdout, "Hello World");  # suppress return value 11 with ;
Hello World
julia> read(stdin, Char)

'\n': ASCII/Unicode U+000a (category Cc: Other, control)
```

Note that `write` returns 11, the number of bytes (in `"Hello World"`) written to `stdout`, but this return value is suppressed with the `;`.

Here Enter was pressed again so that Julia would read the newline. Now, as you can see from this example, `write` takes the data to write as its second argument, while `read` takes the type of the data to be read as the second argument.

For example, to read a simple byte array, we could do:

```
julia> x = zeros(UInt8, 4)
4-element Array{UInt8,1}:
 0x00
 0x00
 0x00
 0x00

julia> read!(stdin, x)
abcd
4-element Array{UInt8,1}:
 0x61
```

```
 0x62
 0x63
 0x64
```

However, since this is slightly cumbersome, there are several convenience methods provided. For example, we could have written the above as:

```
julia> read(stdin, 4)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

or if we had wanted to read the entire line instead:

```
julia> readline(stdin)
abcd
"abcd"
```

Note that depending on your terminal settings, your TTY may be line buffered and might thus require an additional enter before the data is sent to Julia.

To read every line from `stdin` you can use `eachline`:

```
for line in eachline(stdin)
    print("Found $line")
end
```

or `read` if you wanted to read by character instead:

```
while !eof(stdin)
    x = read(stdin, Char)
    println("Found: $x")
end
```

Text I/O

Note that the `write` method mentioned above operates on binary streams. In particular, values do not get converted to any canonical text representation but are written out as is:

```
julia> write(stdout, 0x61);  # suppress return value 1 with ;
a
```

Note that `a` is written to `stdout` by the `write` function and that the returned value is `1` (since `0x61` is one byte).

For text I/O, use the `print` or `show` methods, depending on your needs (see the documentation for these two methods for a detailed discussion of the difference between them):

```
julia> print(stdout, 0x61)
97
```

See Custom pretty-printing for more information on how to implement display methods for custom types.

IO Output Contextual Properties

Sometimes IO output can benefit from the ability to pass contextual information into show methods. The `IOContext` object provides this framework for associating arbitrary metadata with an IO object. For example, `:compact => true` adds a hinting parameter to the IO object that the invoked show method should print a shorter output (if applicable). See the `IOContext` documentation for a list of common properties.

Working with Files

Like many other environments, Julia has an `open` function, which takes a filename and returns an `IOStream` object that you can use to read and write things from the file. For example, if we have a file, `hello.txt`, whose contents are `Hello, World!`:

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)

julia> readlines(f)
1-element Array{String,1}:
 "Hello, World!"
```

If you want to write to a file, you can open it with the write (`"w"`) flag:

```
julia> f = open("hello.txt","w")
```

```
IOStream(<file hello.txt>)

julia> write(f,"Hello again.")
12
```

If you examine the contents of `hello.txt` at this point, you will notice that it is empty; nothing has actually been written to disk yet. This is because the `IOStream` must be closed before the write is actually flushed to disk:

```
julia> close(f)
```

Examining `hello.txt` again will show its contents have been changed.

Opening a file, doing something to its contents, and closing it again is a very common pattern. To make this easier, there exists another invocation of `open` which takes a function as its first argument and filename as its second, opens the file, calls the function with the file as an argument, and then closes it again. For example, given a function:

```
function read_and_capitalize(f::IOStream)
    return uppercase(read(f, String))
end
```

You can call:

```
julia> open(read_and_capitalize, "hello.txt")
"HELLO AGAIN."
```

to open `hello.txt`, call `read_and_capitalize` on it, close `hello.txt` and return the capitalized contents.

To avoid even having to define a named function, you can use the `do` syntax, which creates an anonymous function on the fly:

```
julia> open("hello.txt") do f
           uppercase(read(f, String))
       end
"HELLO AGAIN."
```

A simple TCP example

Let's jump right in with a simple example involving TCP sockets. This functionality is in a standard library package called `Sockets`. Let's first create a simple server:

```
julia> using Sockets

julia> @async begin
           server = listen(2000)
           while true
               sock = accept(server)
               println("Hello World\n")
           end
       end
Task (runnable) @0x00007fd31dc11ae0
```

To those familiar with the Unix socket API, the method names will feel familiar, though their usage is somewhat simpler than the raw Unix socket API. The first call to `listen` will create a server waiting for incoming connections on the specified port (2000) in this case. The same function may also be used to create various other kinds of servers:

```
julia> listen(2000) # Listens on localhost:2000 (IPv4)
Sockets.TCPServer(active)

julia> listen(ip"127.0.0.1",2000) # Equivalent to the first
Sockets.TCPServer(active)

julia> listen(ip"::1",2000) # Listens on localhost:2000 (IPv6)
Sockets.TCPServer(active)

julia> listen(IPv4(0),2001) # Listens on port 2001 on all IPv4 interfaces
Sockets.TCPServer(active)

julia> listen(IPv6(0),2001) # Listens on port 2001 on all IPv6 interfaces
Sockets.TCPServer(active)

julia> listen("testsocket") # Listens on a UNIX domain socket
Sockets.PipeServer(active)

julia> listen("\\\\.\\pipe\\testsocket") # Listens on a Windows named pipe
Sockets.PipeServer(active)
```

Note that the return type of the last invocation is different. This is because this server does not listen on TCP, but rather on a named pipe (Windows) or UNIX domain socket. Also note that Windows named pipe format has to be a specific pattern such that the name prefix (`\\.\pipe\`) uniquely identifies the

file type. The difference between TCP and named pipes or UNIX domain sockets is subtle and has to do with the `accept` and `connect` methods. The `accept` method retrieves a connection to the client that is connecting on the server we just created, while the `connect` function connects to a server using the specified method. The `connect` function takes the same arguments as `listen`, so, assuming the environment (i.e. host, cwd, etc.) is the same you should be able to pass the same arguments to `connect` as you did to listen to establish the connection. So let's try that out (after having created the server above):

```
julia> connect(2000)
TCPSocket(open, 0 bytes waiting)

julia> Hello World
```

As expected we saw "Hello World" printed. So, let's actually analyze what happened behind the scenes. When we called `connect`, we connect to the server we had just created. Meanwhile, the accept function returns a server-side connection to the newly created socket and prints "Hello World" to indicate that the connection was successful.

A great strength of Julia is that since the API is exposed synchronously even though the I/O is actually happening asynchronously, we didn't have to worry about callbacks or even making sure that the server gets to run. When we called `connect` the current task waited for the connection to be established and only continued executing after that was done. In this pause, the server task resumed execution (because a connection request was now available), accepted the connection, printed the message and waited for the next client. Reading and writing works in the same way. To see this, consider the following simple echo server:

```
julia> @async begin
           server = listen(2001)
           while true
               sock = accept(server)
               @async while isopen(sock)
                   write(sock, readline(sock, keep=true))
               end
           end
       end
Task (runnable) @0x00007fd31dc12e60

julia> clientside = connect(2001)
TCPSocket(RawFD(28) open, 0 bytes waiting)

julia> @async while isopen(clientside)
           write(stdout, readline(clientside, keep=true))
```

```
      end
Task (runnable) @0x00007fd31dc11870

julia> println(clientside,"Hello World from the Echo Server")
Hello World from the Echo Server
```

As with other streams, use `close` to disconnect the socket:

```
julia> close(clientside)
```

Resolving IP Addresses

One of the `connect` methods that does not follow the `listen` methods is `connect(host::String,port)`, which will attempt to connect to the host given by the `host` parameter on the port given by the `port` parameter. It allows you to do things like:

```
julia> connect("google.com", 80)
TCPSocket(RawFD(30) open, 0 bytes waiting)
```

At the base of this functionality is `getaddrinfo`, which will do the appropriate address resolution:

```
julia> getaddrinfo("google.com")
ip"74.125.226.225"
```

« Missing Values Parallel Computing »

Powered by Documenter.jl and the Julia Programming Language.