PackageCompiler.jl

By Matthew Roughan, September, 2021, `Julia v1.6.0, PackageCompiler.jl v1.4.1`

Julia is fast out of the box. But it does have one headache, which is that when you start it up and load packages, that can be quite slow. That is annoying if you signed up to Julia for the speed.

`PackageCompile.jl` addresses this problem.

It also provides a means to create stand-alone Julia programs that some could run on their computer without installing Julia themselves.

It's simple to use, and has big benefits. So let's have a look at how and why it works.

Links

Main documents

- https://github.com/JuliaLang/PackageCompiler.jl
- https://julialang.github.io/PackageCompiler.jl/dev/

Tutorials

- https://medium.com/coffee-in-a-klein-bottle/speeding-up-julia-precompilation-97f39d151a9f
- https://julialang.github.io/PackageCompiler.jl/dev/examples/plots.html

Introduction

Julia uses Just-In-Time (JIT) compilation. When you call a function for the first time [1], Julia compiles it. Subsequent calls to the function don't need to repeat this process, but it means the first call will be slow. That is problematic if:

1. You only call the function once, for instance, if you are using a Julia script from another source, e.g., using it in batch mode from a shell script.
2. You want to compile a lot of functions, e.g., when you load a series of big packages at the start of your session.
3. You need to restart Julia often, such as when you are creating new data structures.

The package compiler fixes this issue.

A second problem is that not everyone wants to install Julia in order to use your code. Stand-alone apps are a standard way to distribution software functionality and the package compiler let's you create such apps.

Finally, the package compiler can create a C-library that you could call from another language (if it can't call Julia directly).

I won't talk about the last two tricks here (they are harder) but its well worth knowing they are possible.

What to do

A custom `sysimage` is how we get pre-compilation going to make things fast. A `sysimage` is like a Julia session with preloaded packages, global variables and your functions already compiled. When you run `julia` you are running their default `sysimage` which has the Julia compiler, the standard libraries and the REPL built in, but all those tasty packages you like to use are not.

So how do we do it: there are 4 easy steps.

1. Install `PackageCompiler` as you would any other package but you will need to have a C-compiler installed on your computer. That will be system dependent, but its a standard thing to do (on Windows the package install will do it for you. On Mac's `homebrew` might be a good choice. On linux, `gcc` or `clang`.)
2. Create a pre-compilation file to help the compiler work out what it should pre-compile.
3. Run the function `create_sysimage`, which will create a `.so` shared library file.
4. Start Julia using the new `.so` file using the `--sysimage` flag. If you are calling Julia using the command line this is easy. If calling from Jupyter you need to install a new kernel with the `--sysimage` command.

Example

Making fast plots: this example comes largely from https://julialang.github.io/PackageCompiler.jl/dev/examples/plots.html, but I like `PlotlyJS` so I have added that.

One of the most common problems in Julia is the "time to first plot." The plot libraries are big and take a while to load. For example take the script (I am putting this in `precompile_plots.jl`) .

```
using Plots
using PlotlyJS
plotlyjs()
p = Plots.plot(rand(5), rand(5))
```

Just running this (the first time) took nearly 20 seconds on my fairly beefy computer, e.g.,

```
julia> @time include("precompile_plots.jl")
 17.081378 seconds (43.79 M allocations: 2.617 GiB, ...
```

The second time was much, much faster:

```
julia> @time include("precompile_plots.jl")
  0.002361 seconds (2.56 k allocations: 218.078 KiB)
```

You can see the massive difference.

I will use my `precompile_plots.jl` file above for Step 2. We perfom Step 3 in an interactive Julia REPL by calling two commands:

```
julia> using PackageCompiler
julia> create_sysimage([:Plots, :PlotlyJS],
                       sysimage_path="sys_plots.so",
                       precompile_execution_file="precompile_plots.jl")
```

That process won't be fast – on my machine it took 196 seconds (more than 3 minutes).

Now quit out of Julia, you should see a file called `sys_plots.so`. This is your `sysimage` or shared library file.

For Step 4 just restart Julia with the command (on the command line)

```
julia --sysimage sys_plots.so
```

When I do this, and then rerun my script, I get the following:

```
julia> @time include("precompile_plots.jl")
 0.922924 seconds (2.67 M allocations: 158.302 MiB, ...
```

Note that this is still compiling something (it has more memory allocations and takes longer). So we could perhaps improve our pre-compile script.

However that's all you fundamentally need to get a massive speed up in time-to-first-plot.

If you want it to use it from within Jupyter or VSCode you're going to have to make sure they use the `sysimage`.

How it works

The docs for `PackageCompiler` describe Julia's compilation as Just-Ahead-Of-Time (JAOT) rather tha JIT. Julia doesn't recompile code based on run-time data, which is perhaps a property of JIT compilation. That means we can pre-compile code before running it.

The shared library `sysimage` provides Julia with a set of functionality that has be pre-compiled.

The term "pre-compiled" is apparently misleading. Julia's dynamic typing may still need to do some work, so everything isn't completely sorted. Julia needs to work out what types the functions will be compiled for. `PackageCompiler` does that by tracing an exemplar session and recording which methods were used. The pre-compile file provides the exemplar session, so include commands in it that you might want to be fast. But you don't have to do everything. It seems OK at inferring what it needs, if not perfect.

The `create_sysimage` function has lots of options, e.g., see https://julialang.github.io/PackageCompiler.jl/dev/refs.html. For instance, you can input a vector of pre-compilation files.

Useful bits and pieces:

- There is a dictionary that keeps track of loaded modules called `Base.loaded_modules`
- You can do all this from the command line using the compilers yourself – that seems hard to me. But it shows you what is happening.

No free lunch

There is no free lunch, only free snacks. The costs of pre-compiling are

1. You are locked into the pre-compiled code, *i.e.,* the packages etc. that were installed in the `sysimage` are the ones that will be used.
2. Running Julia with custom `sysimages` will take up extra memory because they include more code. My `sysimage` was a file 214 MB in size. When I start Julia without it, the session uses 97.4 MB. When I run it with the `sysimage` it takes 172.9 MB. The difference is large, but probably not important for most applications I have in mind, particular as Julia uses 246 MB if I load in `Plots` and `PlotlyJS` manually.
3. Starting Julia with a custom `sysimage` will be slightly slower than starting a vanilla Julia session, though much, much faster than starting Julia and loading the packages. This is because there is a small extra time involved in loading the extra pre-compiled code but we are talking about 10s of milliseconds.
4. There are some tricks and traps. The first time I created the example above, I decide to use `PyPlot` and when I tried to load the `sysimage` Julia crashed with a segmentation fault. I think it was because it didn't want to create all that ugly Python stuff. So there are some limits on what packages you can pre-compile.

Notes

[1] Julia pre-compiles many standard functions and operators, such as `+` so these are OK.

[2] A `.so` file is a shared library file. It is a pretty standard thing these days. It provides compiled resources (called objects, but actually a bit different, *e.g.*, methods) that can be loaded into another program so that the calling application doesn't have to provide them. The "shared" part of the name comes from the fact that it is dynamically linked at **run time** [3], and that means that these can be easily updated and shared between multiple applications at once. The alternative – static linking – requires that the methods become part of the calling application when it is compiled, which makes them less flexible and blows up the size of the application particularly if the application only needs one small part of a larger library.

Note that a `.so` file will be system specific, so if you create one on your Mac, it won't work on your Linux box.

https://stackoverflow.com/questions/9809213/what-are-a-and-so-files

https://www.baeldung.com/linux/a-so-extension-files

[3] The `run time` being referred to here is the time at which you run `julia`, not the time at which you run the functions themselves (I think).