Deploying Julia

Strategies, Architectures and Pitfalls

Avik Sengupta
JuliaHub, Inc.

Who am I

- Julia developer since 2012
- Production Julia code since 2016
- Products and services in Julia

What I'll talk about today

- Packaging – Manifests, Docker, Precompilation
- Sysimages, PackageCompiler
- JuliaC
- Miscellenous Tips and tricks

What does deployment mean to you?

- Where do you deploy
  - Public or Private Cloud
  - HPC / Supercomputer
  - Desktop / Workstations
- What do you deploy
  - Batch processes (*Model training / Simulations*)
  - Long running service (*Web apps / API services*)
  - Desktop UI
  - OpenSource / Internal / Commercial

What does deployment mean to me?

- Run code on a machine that is not yours
- Code comes with implicit assumptions about the environment they will run in
  - Operating System
  - Instruction set
  - Dependent libraries and their versions
  - Configuration files

Send/Publish a file with Julia source code

- Simple 😀
- User needs Julia installed 🙁
- User need to manually install packages used 🙁
- User may get different versions of packages used 🙁
- User may use a different Julia version 🙁
- Source code might be proprietary 🙁
- Not really feasible except for the very simplest of codes

Packaging – Project + Manifest

- Project.toml + Manifest.toml
  - Always check in Manifest for Applications
  - User gets the same version of packages 😀
- User needs to install Julia & instantiate 🙁
  - The correct version of Julia
    - New Juliaup functionality to pick Julia versions from project
- Same package works across Operating Systems
- Multiple files to track if all you need is a script 🙁
  - In 1.13, you can store project+manifest in source files

Packaging – Registry

- Always use a private registry for private code
    - Unless you are single developer, deploying to single machine
- A registry is only a git repository, so no major infrastructure is required
- LocalRegistry.jl can help with managing a registry
- Makes dependency versioning and version tracking possible

Considerations for HPC Clusters

- Set your depot to a shared filesystem
  - Shared filesystems are slow
- Make sure to precompile before running a distributed job
  - Don’t want thousands of repeated precompiles
  - Consider running with `--compiled-modules=strict`
  - Precompile on login node if possible
- Set CPU target appropriately
  - Especially for heterogenous nodes
- Figure out correct MPI config

Packaging - Docker

- Public Julia docker images
- Instantiate a manifest during docker build
    - Internet access needed at build time
    - Precompile: `Pkg.precompile()` or use a precompile workload
    - Manage CPU targets unless building on the same machines as deploying
    - `JULIA_CPU_TARGET=generic;sandybridge,-xsaveopt,clone_all;haswell,-rdrnd,base(1)`
- Use memory hints at runtime
    - `julia --heap-size-hint=4G`
- Set thread limits appropriately

Aside on Precompile workloads

- Julia only compiles the code that runs at top level
- If you have functions that take a long time to compile, but haven't been run at top level (ie at compile time) then you will compile at runtime, and pay a latency cost
- Run those functions at top level or
- Use PrecompileTools.jl

System Images (or sysimage)

- “Serialized snapshot of a julia session”,
- Contains julia objects, compiled code, code yet to be compiled, etc.
- Relatively fast to load
- Julia comes with a sysimage containing the base and (most) standard libraries

Custom Sysimages solve the compile time latency

```
pkg> add ModellingToolkit
…
186 dependencies successfully precompiled in 611 seconds

julia> @time using ModelingToolkit
  3.443558 seconds (4.96 M allocations: 341.655 MiB, 7.86% gc time)


julia> @time using ModelingToolkit
 32.879128 seconds (6.31 M allocations: 414.853 MiB, 1.66% gc time, 
1.48% compilation time: 84% of which was recompilation)
```

Enter PackageCompiler.jl

Custom Sysimages solve the compile time latency

```
julia> using PackageCompiler

julia> PackageCompiler.create_sysimage(["ModelingToolkit", "GLMakie"];
sysimage_path="mtk_makie_sys.so")

C:\> julia +1.12 --sysimage=mtk_makie_sys.so --project=.

julia> @time using ModelingToolkit
 0.000991 seconds (505 allocations: 27.688 KiB)
```

Custom sysimages with packagecompiler

- Reduce compile time latency 😀
- Needs a C compiler on the build machine
- Need to use the same version of Julia to load it 🙁
- Artifacts need to be downloaded separately 🙁
- Need sources to resolve new packages in environment
- Build for each OS (and no ability to cross-compile)
- Needs a loooong time to create, and results are biiiiiig 🙁

```
.rwxr-xr-x  1.5G avik   29 Dec 19:07   my_custom_sys.so
```

Apps with PackageCompiler

- Package Julia+sysimage+artifacts+entrypoint

```
julia> PackageCompiler.create_app( "App", "AppCompiled")
```

```
$ tree AppCompiled/
AppCompiled/
├── bin
│   ├── HelloApp
│   └── julia
├── lib
│   ├── julia
│   │   ├── libLLVM.so.18.1jl
│   │   ├── libatomic.so.1.2.0
│   │   ├── libblastrampoline.so.5
│   │   ├── libdSFMT.so
│   │   ├── libgcc_s.so.1
```

Relocatability

- Application should not depend on particular filesystem path
- Paths are used to find different kinds of resources
  - Binaries / Libraries / Data artifacts
- Your package and ALL its dependencies must be relocatable
- Use jll packages and artifacts for native dependencies
- Don’t store file paths or pointers in module globals – use `__init__` functions instead.
  - Actually, use `OncePerProcess` instead (1.12+)

JuliaC

- Create small, standalone binaries (~1.5Mib)
- All code needs to be statically inferable
- 1.12+ only
- Key innovation is ***trim*** support in the compiler
- Needs infrastructure around the compiler, including PackageCompiler

JuliaC

```
pkg> app add JuliaC

$ juliac HelloApp --output-exe hello –trim

$ ls -al hello
-rwxr-xr-x 1 avik avik 1691304 Jan  5 10:54 hello

$ ./hello
Hello, world
```

JuliaC

- JuliaC binaries depend on `libjulia`

```
$ ldd ./hello
        libjulia.so.1.12 => /home/avik/.julia/juliaup/julia-1.12.3+0.x64.linux.gnu/lib/libjulia.so.1.12 (0x00007cac188a1000)
        libjulia-internal.so.1.12 => /home/avik/.julia/juliaup/julia-1.12.3+0.x64.linux.gnu/lib/julia/libjulia-internal.so.1.12 (0x00007cac18200000)
        libunwind.so.8 => /home/avik/.julia/juliaup/julia-1.12.3+0.x64.linux.gnu/lib/julia/libunwind.so.8 (0x00007cac17a00000)
```

Packaging with JuliaC

```
$ juliac HelloApp --output-exe hello --trim --bundle hellodir
```

```
$ tree hellodir
├── bin
│   └── hello
└── lib
    ├── julia
    │   ├── libatomic.so
………………
    │   └── libz.so.1.3.1
    └── libjulia.so.1.12.3
4 directories, 57 files
```

Considerations for long running services

- Latency of first request
  - Good precompile workload, or
  - Warmup requests
- Concurrency – threads or tasks or processes?
- Recommend not putting Julia web services directly on the public internet
  - Proxy via apache/nginx/caddy etc
  - Pay attention to package vulnerabilities

Considerations for cloud native usage

- Interfaces to cloud platforms - AWS.jl / Azure.jl
- Memory is the biggest cost on public cloud services
- Shared disks are slow, fast IO costs more
- Consider running Julia processes on Kubernetes
    - Kuber.jl provides complete set of entities and operations

Tell us your stories

References

- https://github.com/GunnarFarneback/LocalRegistry.jl
- https://juliahpc.github.io/
- Package compiler webinar: https://youtu.be/J6h6Tj8IluE
- https://github.com/JuliaLang/JuliaC.jl/
- Acknowlegements: Kristoffer Carlsson, Mosè Giordano, Chris Rackauckas, Jeff Bezanson