A Deep Dive into the Mill Scala Build Tool

Li Haoyi, Scaladays Madrid 13 Sep 2023

Hello everyone. My name is Haoyi, and this talk is a Deep Dive into the Mill Scala Build Tool

About Myself

Been using Scala since 2012

Work on the Databricks Developer Platform

Lots of Scala OSS work: Scala.js, Mill, Ammonite, uPickle, Fastparse, …

Author of Hands-on Scala Programming

About myself: I’ve been working with Scala for over 10 years now. I currently work on the Databricks Developer Platform. I’ve done a lot of Open Source Scala work. And I’m the author of the book Hands on Scala Programming

Why Care About Mill

To start off, why should anyone care about the Mill build tool? Aren’t there plenty of other build tools already on the market? SBT? Gradle? Maven? Bazel? And so on?

Why Care About Mill

Gradle is such garbage. I can't count the number of times our ex-Gradler has said "you're not going to like the answer..." - Hacker News

SBT's bizarre abstraction and unreadable syntax is a huge, frustrating obstacle to adopting Scala - Reddit

I only have good things to say about Mill, except maybe that I wish it had been released 10 years earlier or so :) - Reddit

The developer experience is so delightful compared to the other build system in the JVM land. And the task model seems just right - Hacker News

For someone who may know nothing about Mill, perhaps the best proof is social: people *like* Mill! Now people may grumble about Gradle or fuss over SBT, but the feedback from those using Mill is largely positive.

Perhaps the last line is worth highlighting: “the task model seems just right”. Although most build tools are not fun, Mill aims to be something different, and largely succeeds. This talk will cover what it does and why it works

A Deep Dive into the Mill Scala Build Tool

1. **Why Build Tools Are Hard**
2. Getting Started with Mill
3. Mill Fundamentals
4. Why Mill Works

We'll cover 4 sections: first, we'll discuss why build tools are hard. Next, we'll dip our toes into the getting started experience with Mill, using it to build a few small Scala projects. We'll then dive into Mill's fundamentals to get a feel for its core building blocks. Lastly, we'll use this experience as a base to discuss *why* Mill works as well as it does

So first, Why Build Tools Are Hard

1.1 Build Tools As Pure Functional Programs

```python
def make_tiramisu(eggs, sugar1, wine, cheese, cream, fingers, espresso, sugar2, cocoa):
    beat_eggs = beat(eggs)
    mixture = beat(beat_eggs, sugar1, wine)
    whisked = whisk(mixture)
    beat_cheese = beat(cheese)
    cheese_mixture = beat(whisked, beat_cheese)
    whipped_cream = whip(cream)
    folded_mixture = fold(cheese_mixture, whipped_cream)
    sweet_espresso = dissolve(sugar2, espresso)
    wet_fingers = soak2seconds(fingers, sweet_espresso)
    assembled = assemble(folded_mixture, wet_fingers)
    complete = sift(assembled, cocoa)
    ready_tiramisu = refrigerate(complete)
    return ready_tiramisu
```

Build pipelines and pure functional programming have a lot in common. Consider this “functional” pseudocode for making a cake, a Tiramisu. While it’s Python pseudocode, it’s pretty straightforward: a function that takes arguments, transforms the arguments or combines them via other functions, and returns the final result. No side effects, no mutability, every step just takes inputs and return outputs. Pure functional programming

1.2 Build Tools As Pure Functional Programs

```python
def make_tiramisu(eggs, sugar1, wine, cheese, cream, fingers, espresso, sugar2, cocoa):
    return refrigerate(
        sift(
            assemble(
                fold(
                    beat(
                        whisk(
                            beat(beat(eggs), sugar1, wine)
                        ),
                        beat(cheese)
                    ),
                    whip(cream)
                ),
                soak2seconds(fingers, dissolve(sugar2, espresso))
            ),
            cocoa
        )
    )
```

We can re-arrange this code a bit to eliminate the intermediate values,

1.3 Build Tools As Pure Functional Programs

```
def make_tiramisu(eggs, sugar1, wine, cheese, cream, fingers, espresso, sugar2, cocoa):

    return refrigerate(
        sift(
            assemble(
                fold(
                    beat(
                        whisk(
                            beat(beat(eggs), sugar1, wine)
                        ),
                        beat(cheese)
                    ),
                    whip(cream)
                ),
                soak2seconds(fingers, dissolve(sugar2, espresso))
            ),
            cocoa
        )
    )
```

Draw a graph to indicate the dataflow,

1.4 Build Tools As Pure Functional Programs

<table>
<tr><td>4 (70 g) large egg yolks</td><td>beat</td><td rowspan="3">beat</td><td rowspan="3">whisk over steam</td><td rowspan="4">beat</td><td rowspan="5">fold</td><td rowspan="8">assemble</td><td rowspan="9">sift</td><td rowspan="9">refrigerate 4 hours</td></tr>
<tr><td>1/2 cup (100 g) granulated sugar</td><td rowspan="2"></td></tr>
<tr><td>1/2 cup (120 mL) sweet Marsala wine</td></tr>
<tr><td>1 lb. (450 g) mascarpone cheese</td><td>beat</td><td colspan="2"></td></tr>
<tr><td>1 cup (240 mL) heavy cream</td><td>whip to soft peaks</td><td colspan="3"></td></tr>
<tr><td>about 40 ladyfinger cookies</td><td></td><td rowspan="3">soak 2 seconds</td><td colspan="3" rowspan="3"></td></tr>
<tr><td>12 oz. (355 mL) prepared espresso</td><td rowspan="2">dissolve</td></tr>
<tr><td>2 tsp. granulated sugar</td></tr>
<tr><td>2 Tbs. (11 g) cocoa powder</td><td colspan="6"></td></tr>
</table>

And find that the graph is identical to the graph represented by this so-called "pipeline" view of the same recipe.

1.4 Build Tools As Pure Functional Programs

Both the “pipeline diagram” and the pure functional program describe the same thing: a dataflow graph!

1.4 Build Tools As Pure Functional Programs

```
beat_eggs = beat(eggs)
mixture = beat(beat_eggs, sugar1, wine)
whisked = whisk(mixture)
beat_cheese = beat(cheese)
cheese_mixture = beat(whisked, beat_cheese)
whipped_cream = whip(cream)
folded_mixture = fold(cheese_mixture, whipped_cream)
sweet_espresso = dissolve(sugar2, espresso)
```

The structure of a build pipeline is, at its core, identical to the structure of a simple pure functional program!

1.4 Build Tools As Pure Functional Programs

```
beat_eggs = beat(eggs)
mixture = beat(beat_eggs, sugar1, wine)
whisked = whisk(mixture)
beat_cheese = beat(cheese)
cheese_mixture = beat(whisked, beat_cheese)
whipped_cream = whip(cream)
folded_mixture = fold(cheese_mixture, whipped_cream)
sweet_espresso = dissolve(sugar2, espresso)
```

Explicit dependencies between computations

No Side Effects, uncontrolled mutation

Build Graph = Dataflow Graph = Call Graph

Both involve explicit dependencies between computations. Both lack side effects, avoiding spooky action at a distance. The build graph matches the dataflow graph the same way a pure-functional call graph matches the dataflow graph. They are simply different ways of representing the exact same graph

So if you want a way for a user to specify some kind of graph data structure to configure their build, having them write some simple pure-functional code is a very reasonable starting point.

1.5 So What's Hard About Build Tools?

So why are build tools such a mess? Why aren't we all writing our build pipelines in simple pure-functional Haskell? Or in MIT-Scheme?

1.5 So What's Hard About Build Tools?

1. What tasks depends on what?
2. Where do input files come from?
3. What needs to run in what order?
4. What can be parallelized and what can't?
5. Where can tasks read/write to disk?
6. How are tasks cached?
7. How are tasks run from the CLI?
8. How are cross-builds (across different configurations) handled?
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build?
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things?
15. How is in-memory caching handled?

So it turns out that apart from managing a dataflow graph, build tools have a bunch of additional requirements that diverge from those of simple pure-functional programs.

Some questions the build graph helps answer: What tasks depend on what? What runs in what order?

But for other questions, the build graph helps not at all: How does caching work? CLI invocations? Cross-builds? Custom tasks?

This list is not exhaustive, but already we can see the problem: build tools need to answer a lot of questions that naive pure functional programs can ignore: question around caching, I/O, introspectability, naming, and so on. Naive pure functions are by default opaque and anonymous: the only thing you do is run them. Build tools need somewhat more than that

1.5 So What's Hard About Build Tools?

1. What tasks depends on what? - SBT Scope Delegation
2. Where do input files come from? - SBT How to Track File Inputs and Outputs
3. What needs to run in what order?
4. What can be parallelized and what can't?
5. Where can tasks read/write to disk? - Gradle Working With Files
6. How are tasks cached? - Gradle Incremental Build
7. How are tasks run from the CLI? - SBT Parsing and Tab Completion
8. How are cross-builds (across different configurations) handled?
9. How do I define my own custom tasks? - Gradle Authoring Tasks
10. How do tasks pass data to each other?
11. How to manage the repetition in a build? - SBT Plugins and Best Practices
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things? - SBT Paths and Globs, Gradle Working with Files
15. How is in-memory caching handled?

One thing to note is that each bullet here is easily a whole chapter worth of documentation - tens of pages to learn how a particular build tool does a particular thing. I've added references to relevant sections of the Gradle and SBT documentation to illustrate just how much detail you might need to know about a single bullet point. And as mentioned earlier, this list is not exhaustive!

Unless we find some shortcut, it seems inevitable that any build tool would need you to read 10, 15, 20 chapters worth of documentation in order to learn how it does… even the most foundational build-tool things!

And anyone who has tried to use SBT or Gradle or Bazel, in-depth, this may sound familiar. Any one of these tools has hundreds of pages of documentation, which is somehow both "too much" but also "not enough". These tools aren't hard due to lack of documentation, or due to too much documentation, they are hard because there simply is *so much stuff to learn* that it proves to be a challenge. E.g. I may be familiar with the filesystem API in half a dozen languages, but throw me into a Gradle plugin and I'll have to learn yet another way of doing things totally different from anything I have seen before.

A Deep Dive into the Mill Scala Build Tool

1. **Why Build Tools Are Hard**
2. Getting Started with Mill
3. Mill Fundamentals
4. Why Mill Works

So that's why build tools are hard. While they may naively look like simple, like pure functional programs, they have a lot more details that they need to manage, and if not done carefully it can result in an explosion of things that a user would need to learn to be productive.

A Deep Dive into the Mill Scala Build Tool

1. Why Build Tools Are Hard
2. **Getting Started with Mill**
3. Mill Fundamentals
4. Why Mill Works

Let’s table those thoughts for now, and we will re-visit them after we’ve seen Mill in action. For now, let’s touch on getting started with using Mill for Scala

2 Getting Started with Mill

1. **Single Scala Module**
2. Customizing Tasks
3. Multiple Scala Modules

We'll cover three examples: a single Scala module, customizing some tasks on that module, and a multi-module project

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

This is a basic Mill build for a single Scala module. The module uses Scala 2.13.11, has two third-party dependencies, and has a test suite with a testing framework configured. `with ScalaModule` specifies that this module is for building Scala code - while `RootModule` puts the module at the root of the repo rather than under a `foo/` subfolder.

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
// src/
```

Since the module lives at the root of the repo, the source code would live under `src/`.

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```scala
// src/Foo.scala
package foo
import scalatags.Text.all._
import mainargs.{main, ParserForMethods}

object Foo {
  def generateHtml(text: String) = h1(text).toString

  @main
  def main(text: String) =
    println(generateHtml(text))

  def main(args: Array[String]): Unit =
    ParserForMethods(this).runOrExit(args)
}
```

Here's the example `Foo.scala` file we'll be using. It's a trivial example: just does some argument parsing, renders some HTML, and prints it.

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
> ./mill resolve _
assembly
...
clean
...
compile
...
run
...
show
...
inspect
...
```

You can run `mill resolve _` with a wildcard to list out the available tasks to run,

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
> ./mill inspect compile
compile(ScalaModule.scala:212)
    Compiles the current module to
    generate compiled classfiles.
Inputs:
    scalaVersion
    upstreamCompileOutput
    allSourceFiles
    compileClasspath
```

You can run `mill inspect` to look up the metadata of a particular task: where it was defined, it's scaladoc, it's inputs, etc.

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
> ./mill compile
...
compiling 1 Scala source to...
```

`mill compile` to compile the module

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
> ./mill compile
...
compiling 1 Scala source to...

> ./mill run --text hello
<h1>hello</h1>
```

`mill run` to run it

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
> ./mill compile
...
compiling 1 Scala source to...

> ./mill run --text hello
<h1>hello</h1>

> ./mill test
...
+ foo.FooTests.simple ...  <h1>hello</h1>
+ foo.FooTests.escaping ...
<h1>&lt;hello&gt;</h1>
```

`mill test` to run its test suite,

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
> ./mill assembly
```

Next, we can use `mill assembly` to generate the assembly jar you can then run later outside of the build tool.

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
> ./mill assembly

> ./mill show assembly
".../out/assembly.dest/out.jar"
```

You can use `mill show` to show the metadata output of a task, in this case displaying the exact name of the generated assembly

2.1 Single Scala Module

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
  def ivyDeps = Agg(
    ivy"com.lihaoyi::scalatags:0.8.2",
    ivy"com.lihaoyi::mainargs:0.4.0"
  )

  object test extends ScalaTests {
    def ivyDeps = Agg(ivy"com.lihaoyi::utest:0.7.11")
    def testFramework = "utest.runner.Framework"
  }
}
```

```
> ./mill assembly

> ./mill show assembly
".../out/assembly.dest/out.jar"

> java -jar out/assembly.dest/out.jar --text hello
<h1>hello</h1>

> ./out/assembly.dest/out.jar --text hello
<h1>hello</h1>
```

And finally, we can run `java -jar`, or just run the assembly directly if you want to execute it outside of the build tool

And there you have it! That's all you need to get started using Mill: downloading third party dependencies, compiling, running tests, and packaging a module for deployment and distribution. You can even publish things to Maven Central right out of the box. While the `build.sc` on the left isn't *trivial*, hopefully it's simple enough you can skim over, guess what the various parts do, and guess correctly.

Note that even with such a trivial example, Mill is incremental by default: if you run `assembly` a second time, it will re-use the earlier assembly unless you make changes to files in the `src/` or `resources/` folder. Mill does not do unnecessary work, and will re-use whatever it can unless some file changes force something to be re-built

2 Getting Started with Mill

1. Single Scala Module

2. **Customizing Tasks**

3. Multiple Scala Modules

So that's building and packaging single Scala module, end-to-end.

Next, let's look at how customizing how a task or module works.

2.2 Customizing Tasks

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"
}
```

Let’s start with the example from before, but with the test suite and third-party dependencies stripped out for simplicity. This is now a *truly* minimal Mill config for a single Scala module!

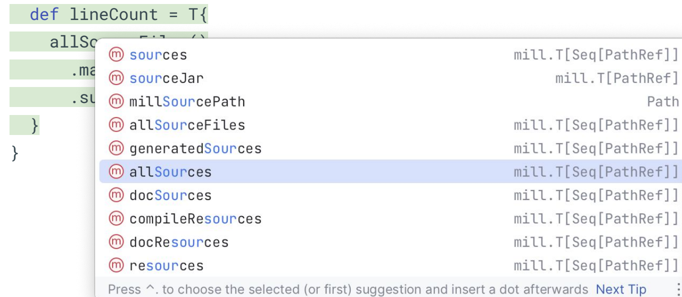
2.2 Customizing Tasks

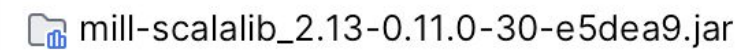
```
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }
}
```

Starting from that, we can define a new task `lineCount` that depends on the existing task `allSourceFiles`

2.2 Customizing Tasks

```
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
```





`allSourceFiles` is built in, defined by `ScalaModule`, and your IDE can help you list, search, and show you the return-type and scaladoc for these tasks if you're not familiar

2.2 Customizing Tasks

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }
}
```

`lineCount` takes the output of `allSourceFiles`, counts how many lines are in each file, and sums them up returning a single integer. That's all we need to create a custom task!

2.2 Customizing Tasks

```
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }
}
```

```
> ./mill show lineCount
19
```

Even though `lineCount` isn't used anywhere, we can still use `mill show` from the CLI to see its return value

2.2 Customizing Tasks

```
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }
}
```

```
> ./mill show lineCount
19

> ./mill inspect lineCount
lineCount(build.sc:7)
Inputs:
    allSourceFiles
```

or `mill inspect` to see the metadata about it, e.g. where it is defined and what it depends on

2.2 Customizing Tasks

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }

  override def resources = T{
    os.write(
      T.dest / "line-count.txt", "" + lineCount()
    )
    Seq(PathRef(T.dest))
  }
}
```

Next, if we want to actually *use* `lineCount`, we can do so for example by overriding `resources` to return a folder with a single generated file, `line-count.txt` containing the value of `lineCount`. Again, if you didn't know about the `resources` task, your IDE will gladly help you find what you want

2.2 Customizing Tasks

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }

  override def resources = T{
    os.write(
      T.dest / "line-count.txt", "" + lineCount()
    )
    Seq(PathRef(T.dest))
  }
}
```

```scala
// src/Foo.scala
package foo

object Foo{
  def main(args: Array[String]): Unit = {
    val lineCount = scala.io.Source
      .fromResource("line-count.txt")
      .mkString

    println(s"Line Count: $lineCount")
  }
}
```

Now with an appropriate `src/Foo.scala` file, in this example containing a main method that reads the resource and prints out its contents,

2.2 Customizing Tasks

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }

  override def resources = T{
    os.write(
      T.dest / "line-count.txt", "" + lineCount()
    )
    Seq(PathRef(T.dest))
  }
}
```

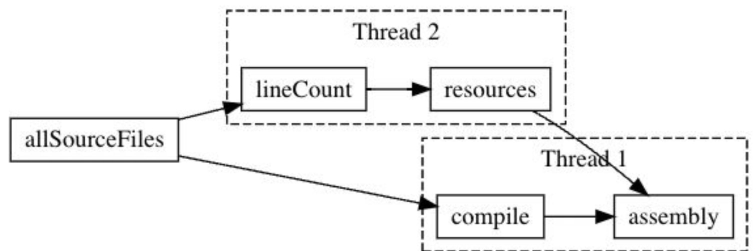
```scala
// src/Foo.scala
package foo

object Foo{
  def main(args: Array[String]): Unit = {
    val lineCount = scala.io.Source
      .fromResource("line-count.txt")
      .mkString

    println(s"Line Count: $lineCount")
  }
}
```

```
> ./mill run
...
Line Count: 19
```

While this is a toy example, it shows how easy it is to customize your Mill build to include the kinds of custom logic common in most real-world projects. Almost every project needs to do *something* custom, but that *something* differs from project to project. Mill can't hard-code support for everyone's requirements, but it can make it easy and safe for you to implement them yourselves and have them integrate well with the rest of the build system.

Like the builtin `compile` or `assembly` tasks, `lineCount` and `resources` are cached and incremental by default: `line-count.txt` will not be re-generated unless `allSourceFiles` changes due to a change in the `src/` folder.

2.2 Customizing Tasks



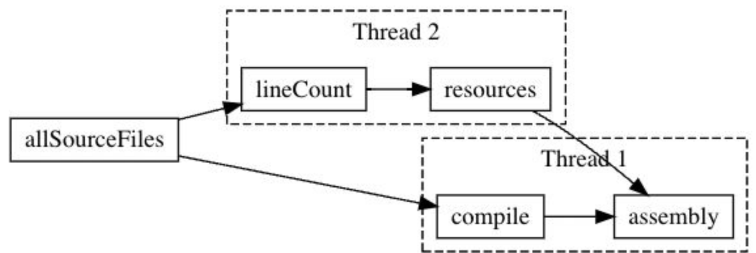

```scala
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }

  override def resources = T{
    os.write(
      T.dest / "line-count.txt", "" + lineCount()
    )
    Seq(PathRef(T.dest))
  }
}
```

```scala
object Foo{
  def main(args: Array[String]): Unit = {
    val lineCount = scala.io.Source
      .fromResource("line-count.txt")
      .mkString

    println(s"Line Count: $lineCount")
  }
}
```

```
> ./mill run
...
Line Count: 19
```

They are parallelizable: `lineCount` and `resources` can run in parallel with `compile` if given the threads.

2.2 Customizing Tasks




```
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }

  override def resources = T{
    os.write(
      T.dest / "line-count.txt", "" + lineCount()
    )
    Seq(PathRef(T.dest))
  }
}
```

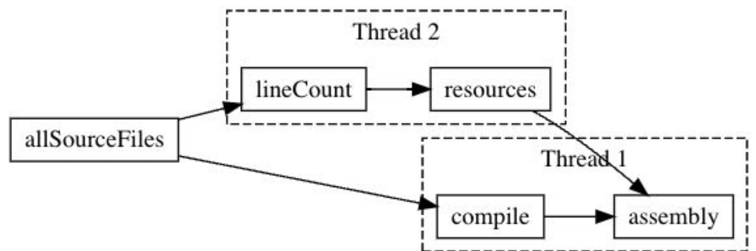
```
object Foo{
  def m
    val
      .
      .

    pri
  }
}
```

```
> ./mill show lineCount
19

> ./mill inspect lineCount
lineCount(build.sc:7)
Inputs:
    allSourceFiles
```

```
> ./mill run
...
Line Count: 19
```

They can be introspected with `show` and `inspect` as discussed earlier.

2.2 Customizing Tasks




```
// build.sc
import mill._, scalalib._

object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles()
      .map(f => os.read.lines(f.path).size)
      .sum
  }

  override def resources = T
    os.write(
      T.dest / "line-count.txt", "" + lineCount()
    )
    Seq(PathRef(T.dest))
  }
}
```

```
object Foo{
  def m
    val
      .
      .
```

```
> ./mill show lineCount
19

> ./mill inspect lineCount
```

Overrides member from JavaModule (mill.scalalib)

Press ⌘U to navigate

```
> ./mill run
...
Line Count: 19
```

And they are principled: unlike the ad-hoc monkey patching or mutation common in other build tools, Mill uses object-oriented overrides for these customizations, meaning your IDE understands and can help you navigate up and down the inheritance hierarchy and overrides.

While these things may not matter for such a trivial example that runs quickly, they become increasingly important as the size and complexity of your build logic grows.

2 Getting Started with Mill

1. Single Scala Module

2. Customizing Tasks

3. **Multiple Scala Modules**

Lastly, lets look at an example of multiple Scala modules in a single Mill build.

2.3 Multiple Scala Modules

```scala
// build.sc
import mill._, scalalib._

trait MyModule extends ScalaModule {
  def scalaVersion = "2.13.11"
}

object foo extends MyModule {
  def moduleDeps = Seq(bar)
  def ivyDeps = Agg(ivy"com.lihaoyi::mainargs:0.4.0")
}

object bar extends MyModule {
  def ivyDeps = Agg(ivy"com.lihaoyi::scalatags:0.8.2")
}
```

This example build contains two modules, foo and bar. You can define multiple modules the same way you define a single module, via `object`s, using `def moduleDeps` to specify the relationship between them.

Note that we split out the `scalaVersion` config common to both modules into a separate `trait MyModule extends ScalaModule`. This lets us avoid the need to copy-paste common configuration, while still letting us define any per-module config such as `ivyDeps` unique to each particular module.

2.3 Multiple Scala Modules

```scala
// build.sc
import mill._, scalalib._

trait MyModule extends ScalaModule {
  def scalaVersion = "2.13.11"
}

object foo extends MyModule {
  def moduleDeps = Seq(bar)
  def ivyDeps = Agg(ivy"com.lihaoyi::mainargs:0.4.0")
}

object bar extends MyModule {
  def ivyDeps = Agg(ivy"com.lihaoyi::scalatags:0.8.2")
}
```

```
build.sc
foo/
    src/
        Foo.scala
    resources/
        ...
bar/
    src/
        Bar.scala
    resources/
        ...
out/
    foo/
        compile.json
        compile.dest/
        ...
    bar/
        compile.json
        compile.dest/
        ...
```

This setup expects the following project layout. Both source code and output files in Mill follow the module hierarchy, so e.g. input to the `foo` module lives in `foo/src/` and output to `foo.compile` lives in `out/foo/compile.json` and the folder `out/foo/compile.dest/`

2.3 Multiple Scala Modules

```scala
// build.sc
import mill._, scalalib._

trait MyModule extends ScalaModule {
  def scalaVersion = "2.13.11"
}

object foo extends MyModule {
  def moduleDeps = Seq(bar)
  def ivyDeps = Agg(ivy"com.lihaoyi::mainargs:0.4.0")
}

object bar extends MyModule {
  def ivyDeps = Agg(ivy"com.lihaoyi::scalatags:0.8.2")
}
```

```
> ./mill resolve __.run
foo.run
bar.run
```

You can `resolve` to see what targets are available

2.3 Multiple Scala Modules

```scala
// build.sc
import mill._, scalalib._

trait MyModule extends ScalaModule {
  def scalaVersion = "2.13.11"
}

object foo extends MyModule {
  def moduleDeps = Seq(bar)
  def ivyDeps = Agg(ivy"com.lihaoyi::mainargs:0.4.0")
}

object bar extends MyModule {
  def ivyDeps = Agg(ivy"com.lihaoyi::scalatags:0.8.2")
}
```

```
> ./mill resolve __.run
foo.run
bar.run

> ./mill __.compile
```

Use the wildcards to run multiple tasks at once, e.g. compiling everything

Mill will ensure that the modules and tasks - such as compile - are evaluated in the right order, and re-evaluated as necessary when source code in each module changes.

2.3 Multiple Scala Modules

```scala
// build.sc
import mill._, scalalib._

trait MyModule extends ScalaModule {
  def scalaVersion = "2.13.11"
}

object foo extends MyModule {
  def moduleDeps = Seq(bar)
  def ivyDeps = Agg(ivy"com.lihaoyi::mainargs:0.4.0")
}

object bar extends MyModule {
  def ivyDeps = Agg(ivy"com.lihaoyi::scalatags:0.8.2")
}
```

```
> ./mill resolve __.run
foo.run
bar.run

> ./mill __.compile

> ./mill foo.run --foo-text hello --bar-text world
Foo.value: hello
Bar.value: <p>world</p>
```

You can invoke a specific module's `run` task with CLI parameters, running it's main method

2.3 Multiple Scala Modules

```scala
// build.sc
import mill._, scalalib._

trait MyModule extends ScalaModule {
  def scalaVersion = "2.13.11"
}

object foo extends MyModule {
  def moduleDeps = Seq(bar)
  def ivyDeps = Agg(ivy"com.lihaoyi::mainargs:0.4.0")
}

object bar extends MyModule {
  def ivyDeps = Agg(ivy"com.lihaoyi::scalatags:0.8.2")
}
```

```
> ./mill resolve __.run
foo.run
bar.run

> ./mill __.compile

> ./mill foo.run --foo-text hello --bar-text world
Foo.value: hello
Bar.value: <p>world</p>

> ./mill show {foo,bar}.assembly
{
  "foo.assembly": "out/foo/assembly.dest/out.jar",
  "bar.assembly": "out/foo/assembly.dest/out.jar"
}
```

Or generate multiple assemblies at once using brace expansion

2 Getting Started with Mill

1. Single Scala Module

2. Customizing Tasks

3. **Multiple Scala Modules**

So that's it for setting up multiple Scala modules

We've just gone through a quick tour of what it's like getting started using Mill to compile some Scala code: a single module, a module with customizations, and multiple related modules. As you've seen, Mill comes with most common workflows built in, including generating self-contained assemblies and publishing artifacts, meaning you can be productive right off the bat.

Because we rely so much on the defaults provided by the `ScalaModule` trait, the build files we've seen so far are really nothing more than glorified YAML key-value pair configs. But, Mill is much more than just a bunch of helpers you plug key-value pairs into

A Deep Dive into the Mill Scala Build Tool

1. Why Build Tools Are Hard

2. Getting Started with Mill

3. **Mill Fundamentals**

4. Why Mill Works

Let’s now dive into using Mill without such pre-built scaffolding, so we can truly understand what Mill is all about!

3.Mill Fundamentals

1. **Tasks**

2. Modules

3. DIY Java Modules

This section on Mill Fundamentals has 3 parts: we'll look at the core tasks that Mill uses. We'll then look into how Modules work. We'll combine this knowledge together to come up with our own DIY implementation Java Modules, that we can use to compile and package a multi-module Java codebase. These will be similar to the builtin ScalaModule traits we saw earlier, but instead built ourselves entirely from first principles.

The goal of this is to give you a good understanding of how Mill's building blocks work, what they're like to use, and how they can stack together to form something greater than the sum of its parts

The first core concept to understand in Mill are Tasks

3.1 Tasks

```
// build.sc
import mill._

def sources = T.source { millSourcePath / "src" }

def allSources = T {
  os.walk(sources().path)
    .filter(_.ext == "java")
    .map(PathRef(_))
}

def lineCount: T[Int] = T {
  println("Computing line count")
  allSources()
    .map(p => os.read.lines(p.path).size)
    .sum
}
```

The most important Tasks to know about are `T.source` task, which specify an input file or folder, and `T{}` targets, which specify some kind of computation dependent on one or more other upstream tasks.

Here, we have a `T.source` named `sources` that references a `src/` folder, an `allSources` target that walks the `sources` folder and finds all Java files within it, and a `lineCount` target that takes these `allSources`, counts their lines, and sums it up.

3.1 Tasks

```
// build.sc
import mill._

def sources = T.source { millSourcePath / "src" }

def allSources = T {
  os.walk(sources().path)
    .filter(_.ext == "java")
    .map(PathRef(_))
}

def lineCount: T[Int] = T {
  println("Computing line count")
  allSources()
    .map(p => os.read.lines(p.path).size)
    .sum
}
```

This simple build sets up the following build graph: `sources` is used by `allSources`, which is then used by `lineCount`

Here we have no `ScalaModule` or any other such helpers. So the only logic we have in this `build.sc` file is the logic you see right in front of you. `sources`, `allSources`, and `lineCount`

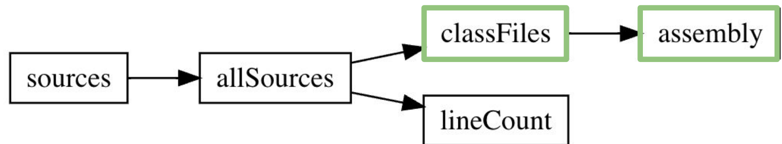
3.1 Tasks

```
// build.sc
import mill._

def sources = T.source { millSourcePath / "src" }

def allSources = T {
  os.walk(sources().path)
    .filter(_.ext == "java")
    .map(PathRef(_))
}

def lineCount: T[Int] = T {
  println("Computing line count")
  allSources()
    .map(p => os.read.lines(p.path).size)
    .sum
}
```

```
> ./mill show lineCount
Computing line count
16
```

Again, `lineCount` is not used anywhere, but we can run it and use `show` to print its value.

3.1 Tasks

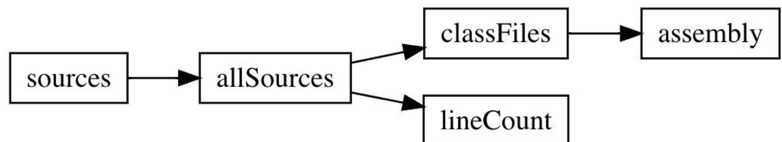



```
def classFiles = T {
  println("Generating classfiles")
  val javacArgs = allSources().map(_.path)
  os.proc("javac", javacArgs, "-d", T.dest)
    .call(cwd = T.dest)

  PathRef(T.dest)
}

def assembly = T {
  println("Generating assembly")
  val jarPath = T.dest / "out.jar"
  os.proc("jar", "-cfe", jarPath, "foo.Foo", ".")
    .call(cwd = classFiles().path)

  PathRef(T.dest / "out.jar")
}
```

Targets can run arbitrary code. Here, we add two targets: a `classFiles` target that loads `allSources` and passes them to a `javac` subprocess to compile, and a `assembly` target the loads `classFiles` and passes them to the a `jar` subprocess to bundle up into an `out.jar` file.

The resultant task graph is shown in the top right, now with additional `classFiles` and `assembly` tasks, highlighted in green, downstream of `allSources`

3.1 Tasks

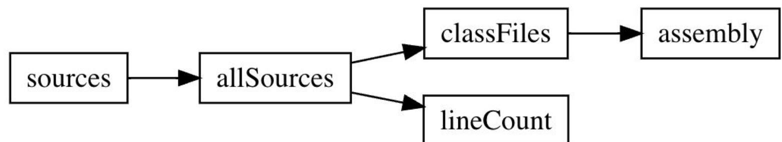



```
def classFiles = T {
  println("Generating classfiles")
  val javacArgs = allSources().map(_.path)
  os.proc("javac", javacArgs, "-d", T.dest)
    .call(cwd = T.dest)

  PathRef(T.dest)
}

def assembly = T {
  println("Generating assembly")
  val jarPath = T.dest / "out.jar"
  os.proc("jar", "-cfe", jarPath, "foo.Foo", ".")
    .call(cwd = classFiles().path)

  PathRef(T.dest / "out.jar")
}
```

```
> ./mill show assembly
Generating classfiles
Generating assembly
".../out/assembly.dest/out.jar"
```

Again, you can see that if we run `assembly`, both `classFiles` and `assembly` are evaluated the first time

3.1 Tasks

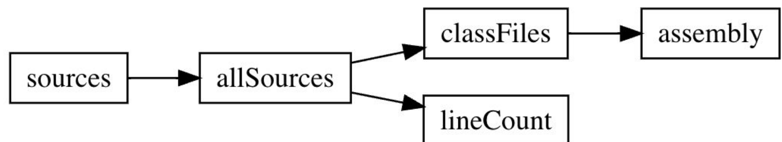



```
def classFiles = T {
  println("Generating classfiles")
  val javacArgs = allSources().map(_.path)
  os.proc("javac", javacArgs, "-d", T.dest)
    .call(cwd = T.dest)

  PathRef(T.dest)
}

def assembly = T {
  println("Generating assembly")
  val jarPath = T.dest / "out.jar"
  os.proc("jar", "-cfe", jarPath, "foo.Foo", ".")
    .call(cwd = classFiles().path)

  PathRef(T.dest / "out.jar")
}
```

```
> ./mill show assembly
Generating classfiles
Generating assembly
".../out/assembly.dest/out.jar"

> ./mill show assembly
".../out/assembly.dest/out.jar"
```

They are re-used if we run `assembly` again

3.1 Tasks

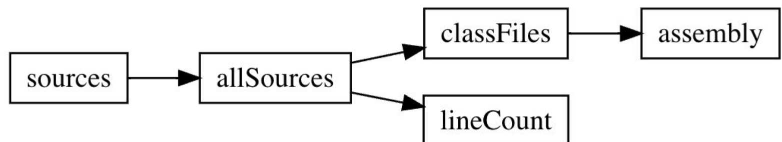



```
def classFiles = T {
  println("Generating classfiles")
  val javacArgs = allSources().map(_.path)
  os.proc("javac", javacArgs, "-d", T.dest)
    .call(cwd = T.dest)

  PathRef(T.dest)
}

def assembly = T {
  println("Generating assembly")
  val jarPath = T.dest / "out.jar"
  os.proc("jar", "-cfe", jarPath, "foo.Foo", ".")
    .call(cwd = classFiles().path)

  PathRef(T.dest / "out.jar")
}
```

```
> ./mill show assembly
Generating classfiles
Generating assembly
".../out/assembly.dest/out.jar"

> ./mill show assembly
".../out/assembly.dest/out.jar"

> echo "package hello2" > src/hello2.java
```

And when we make more changes to the `src/` folder

3.1 Tasks

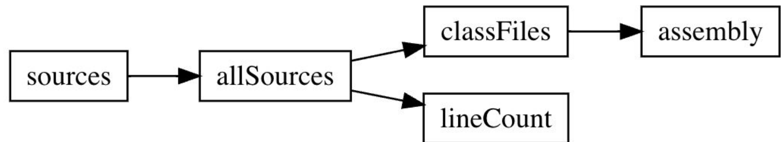



```
def classFiles = T {
  println("Generating classfiles")
  val javacArgs = allSources().map(_.path)
  os.proc("javac", javacArgs, "-d", T.dest)
    .call(cwd = T.dest)

  PathRef(T.dest)
}

def assembly = T {
  println("Generating assembly")
  val jarPath = T.dest / "out.jar"
  os.proc("jar", "-cfe", jarPath, "foo.Foo", ".")
    .call(cwd = classFiles().path)

  PathRef(T.dest / "out.jar")
}
```

```
> ./mill show assembly
Generating classfiles
Generating assembly
".../out/assembly.dest/out.jar"

> ./mill show assembly
".../out/assembly.dest/out.jar"

> echo "package hello2" > src/hello2.java

> ./mill show assembly
Generating classfiles
Generating assembly
".../out/assembly.dest/out.jar"
```

Both `classFiles` and `assembly` are re-evaluated

3.1 Tasks

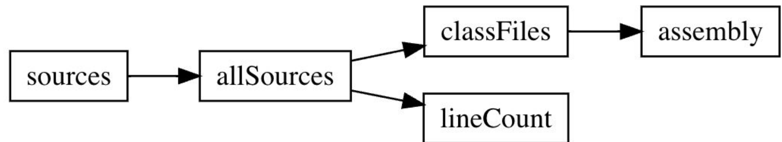



```
def classFiles = T {
  println("Generating classfiles")
  val javacArgs = allSources().map(_.path)
  os.proc("javac", javacArgs, "-d", T.dest)
    .call(cwd = T.dest)

  PathRef(T.dest)
}

def assembly = T {
  println("Generating assembly")
  val jarPath = T.dest / "out.jar"
  os.proc("jar", "-cfe", jarPath, "foo.Foo", ".")
    .call(cwd = classFiles().path)

  PathRef(T.dest / "out.jar")
}
```

```
> ./mill show assembly
Generating classfiles
Generating assembly
".../out/assembly.dest/out.jar"

> ./mill show assembly
".../out/assembly.dest/out.jar"

> echo "package hello2" > src/hello2.java

> ./mill show assembly
Generating classfiles
Generating assembly
".../out/assembly.dest/out.jar"

> java -jar out/assembly.dest/out.jar  i am cow
Foo.value: 31337
args: i am cow
```

And this jar is a normal Java Jar file, and can be run using `java -jar`

3.1 Tasks




```
// build.sc
import mill._

def sources = T.source { millSourcePath / "src" }

def allSources = T {
  os.walk(sources().path)
    .filter(_.ext == "java")
    .map(PathRef(_))
}

def lineCount: T[Int] = T {
  println("Computing line count")
  allSources()
    .map(p => os.read.lines(p.path).size)
    .sum
}
```

```
def classFiles = T {
  println("Generating classfiles")
  val javacArgs = allSources().map(_.path.toString
  os.proc("javac", javacArgs, "-d", T.dest)
    .call(cwd = T.dest)

  PathRef(T.dest)
}

def assembly = T {
  println("Generating jar")
  val jarPath = T.dest / "out.jar"
  os.proc("jar", "-cfe", jarPath, "foo.Foo", ".")
    .call(cwd = classFiles().path)

  PathRef(T.dest / "out.jar")
}
```

Here’s the full code for this worked example, with the task graph shown in the top right

Mill tasks can contain arbitrary code, and have logic either running in-process on the JVM or via sub-processes. The assumption that Mill makes is that all inputs to the task that may change are defined by upstream tasks (whether Targets or Sources), and all outputs for a task must be included in the return value. Tasks may read and write to disk, but only within their dedicated `T.dest` folder, which helps avoid filesystem conflicts or unexpected interactions.

In this way, Mill lets you implement a task however you want, but guides you to a pure-functional style, even for task that may include heavy use of the io, disk, and filesystem

3.Mill Fundamentals

1. Tasks

2. **Modules**

3. DIY Java Modules

Now that we've played around with some simple custom tasks, using them to compile and some package Java code, let's look at how Mill Modules work

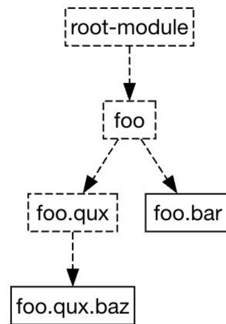
3.2 Modules

```
// build.sc
import mill._

object foo extends Module {
  def bar = T { "hello" }
  object qux extends Module {
    def baz = T { "world" }
  }
}
```

Modules, at their simplest, are just namespaces you can put tasks in. Here, we have `foo` module containing the `foo.bar` target, and the `foo.qux` module containing the `foo.qux.baz` target.

3.2 Modules

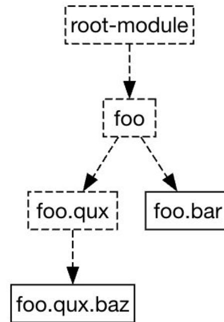
```
// build.sc
import mill._

object foo extends Module {
  def bar = T { "hello" }
  object qux extends Module {
    def baz = T { "world" }
  }
}
```



Modules form a tree. This simple build file forms the tree shown above, with dashed boxes indicating the modules and solid boxes indicating the targets at the leaves of the tree

3.2 Modules

```scala
// build.sc
import mill._

object foo extends Module {
  def bar = T { "hello" }
  object qux extends Module {
    def baz = T { "world" }
  }
}
```

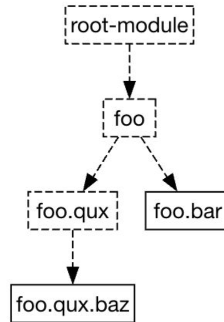



```
> ./mill show foo.bar
"hello"

> ./mill show foo.qux.baz
"world"
```

On the right, you can see how we can run these targets from the CLI: foo.bar and foo.qux.baz

3.2 Modules

```
// build.sc
import mill._

object foo extends Module {
  def bar = T { "hello" }
  object qux extends Module {
    def baz = T { "world" }
  }
}
```




```
> ./mill show foo.bar
"hello"

> ./mill show foo.qux.baz
"world"

> cat out/foo/bar.json
..."value": "hello"...

> cat out/foo/qux/baz.json
..."value": "world"...
```

We can also look at their metadata directly in the `out/` folder at their respective paths out/foo/bar.json and out/foo/qux/baz.json.

This example is trivial - the tasks just return constant strings - but they illustrate the core module tree structure and how it is reflected throughout Mill: in code, logically, in the CLI, and on disk

3.2 Modules

```
// build.sc
import mill._

trait FooModule extends Module {
  def bar: T[String] // required override
  def qux = T { bar() + " world" }
}
```

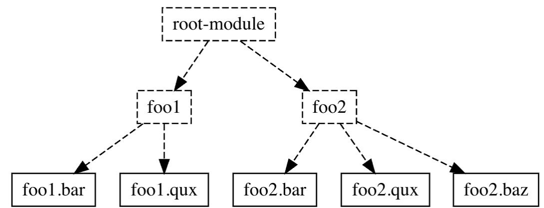
Modules can be turned into traits for re-use. We saw some of this in our earlier example with `trait MyModule extends ScalaModule`, but here’s a standalone example that does not make use of any builtin helpers. We have `trait FooModule` extending `mill.Module`, defining one abstract target `bar` and one concrete target `qux`

3.2 Modules

```scala
// build.sc
import mill._

trait FooModule extends Module {
  def bar: T[String] // required override
  def qux = T { bar() + " world" }
}

object foo1 extends FooModule{
  def bar = "hello"
  def qux = super.qux().toUpperCase
}
object foo2 extends FooModule {
  def bar = "hi"
  def baz = T { qux() + " I am Cow" }
}
```

We can extend it using two modules `foo1` and `foo2` that each extend `FooModule` with their own customizations: different implementations for `def bar`, `foo1` of them overrides `qux`, and `foo2` adds a new target `baz`.

3.2 Modules

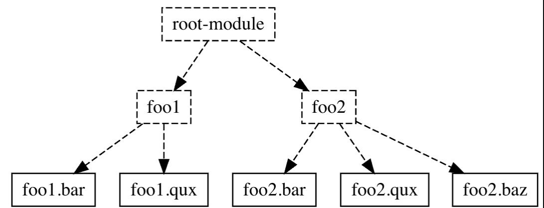



```scala
// build.sc
import mill._

trait FooModule extends Module {
  def bar: T[String] // required override
  def qux = T { bar() + " world" }
}

object foo1 extends FooModule{
  def bar = "hello"
  def qux = super.qux().toUpperCase
}
object foo2 extends FooModule {
  def bar = "hi"
  def baz = T { qux() + " I am Cow" }
}
```

This results in a module tree as shown above

3.2 Modules

```
// build.sc
import mill._

trait FooModule extends Module {
  def bar: T[String] // required override
  def qux = T { bar() + " world" }
}

object foo1 extends FooModule{
  def bar = "hello"
  def qux = super.qux().toUpperCase
}
object foo2 extends FooModule {
  def bar = "hi"
  def baz = T { qux() + " I am Cow" }
}
```

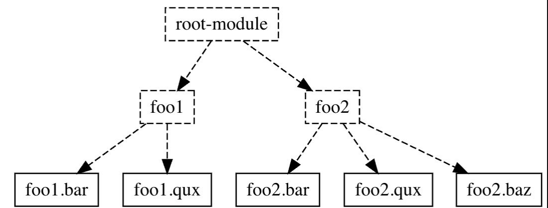



```
> ./mill show foo1.bar
"hello"
```

And you can see the values printed by `show` are as you would expect: `foo1.bar` is "hello"

3.2 Modules

```
// build.sc
import mill._

trait FooModule extends Module {
  def bar: T[String] // required override
  def qux = T { bar() + " world" }
}

object foo1 extends FooModule{
  def bar = "hello"
  def qux = super.qux().toUpperCase
}
object foo2 extends FooModule {
  def bar = "hi"
  def baz = T { qux() + " I am Cow" }
}
```

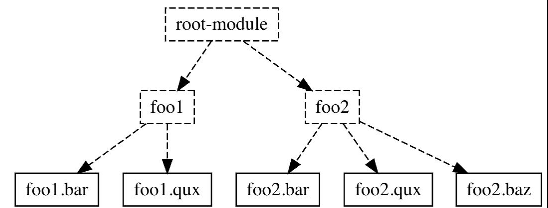



```
> ./mill show foo1.bar
"hello"

> ./mill show foo1.qux
"HELLO WORLD"
```

`foo1.qux` is “HELLO WORLD”, all uppercase because of the override that calls `toUpperCase`

3.2 Modules

```
// build.sc
import mill._

trait FooModule extends Module {
  def bar: T[String] // required override
  def qux = T { bar() + " world" }
}

object foo1 extends FooModule{
  def bar = "hello"
  def qux = super.qux().toUpperCase
}
object foo2 extends FooModule {
  def bar = "hi"
  def baz = T { qux() + " I am Cow" }
}
```

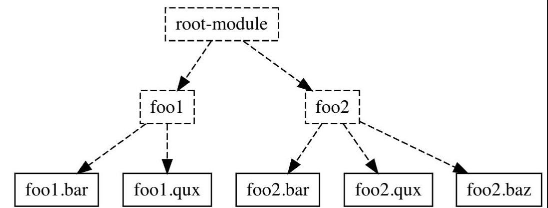



```
> ./mill show foo1.bar
"hello"

> ./mill show foo1.qux
"HELLO WORLD"

> ./mill show foo2.qux
"hi world"
```

`foo2.qux` is “hi world”

3.2 Modules

```
// build.sc
import mill._

trait FooModule extends Module {
  def bar: T[String] // required override
  def qux = T { bar() + " world" }
}

object foo1 extends FooModule{
  def bar = "hello"
  def qux = super.qux().toUpperCase
}
object foo2 extends FooModule {
  def bar = "hi"
  def baz = T { qux() + " I am Cow" }
}
```




```
> ./mill show foo1.bar
"hello"

> ./mill show foo1.qux
"HELLO WORLD"

> ./mill show foo2.qux
"hi world"

> ./mill show foo2.baz
"hi world I am Cow"
```

While `foo2.baz`, the new target we added, is “hi world I am Cow”

The built-in `mill.scalalib` package uses traits to define `ScalaModule`, `JavaModule`, `ScalaTestsModule`, etc. each of which contain some set of "standard" operations such as `compile`, `jar` or `assembly`.. But those traits are not privileged in any way, and you can easily extend and customize them to your specific own use case, or write your own `Module` traits from scratch if you need something different.

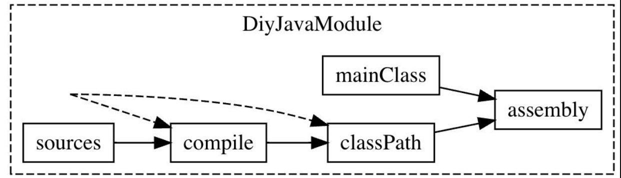
3.Mill Fundamentals

1. Tasks
2. Modules
3. **DIY Java Modules**

Now that we have learned about tasks, and we've learned about modules, let's dig into this example where we use both of them together in a more realistic setting. We'll write a re-usable Module trait that can compile a multi-module Java codebase into distributable assemblies, from first principles

```scala
import mill._
trait DiyJavaModule extends Module{
  def moduleDeps: Seq[DiyJavaModule] = Nil
  def mainClass: T[Option[String]] = None
  def upstream: T[Seq[PathRef]] = T{
    T.traverse(moduleDeps)(_.classPath)().flatten
  }
  def sources = T.source(millSourcePath / "src")
  def compile = T {
    val allSources = os.walk(sources().path)
    val cpFlag = upstream().map(_.path).mkString(":")
    os.proc("javac", "-cp", cpFlag, allSources, "-d", T.dest).call()
    PathRef(T.dest)
  }
  def classPath = T{ Seq(compile()) ++ upstream() }
  def assembly = T {
    for(cp <- classPath()) os.copy(cp.path, T.dest, mergeFolders = true)

    val mainFlags = mainClass().toSeq.flatMap(Seq("-e", _))
    os.proc("jar", "-c", mainFlags, "-f", T.dest / s"out.jar", ".")
      .call(cwd = T.dest)

    PathRef(T.dest / s"out.jar")
  }
}
```

3.3 DIY Java Modules



The code on the left implements our own minimal version of `mill.scalalib.JavaModule`, named `DiyJavaModule`, from first principles.

This is similar to the sets of targets we saw earlier: you have a `sources` T.source task, a `compile` task that shells out to `javac`, and an `assembly` task that shells out to the `jar` command. But we also have some notable changes to call out:

- All the tasks are bundled together inside a `trait DiyJavaModule extends Module`. This lets us instantiate the entire target graph multiple times
- `def moduleDeps` lets us define the module-level dependencies, which are then used by `def upstream`
- `def upstream` turns the module-level dependencies defined by `moduleDeps` into task-level dependencies, via the `T.traverse` helper that combines all their `classPath`s, each a sequence of paths, together into one big sequence. I won't go too deeply into this, but it works similar to `traverse` methods you may have seen elsewhere, such as `Future.traverse`
- We need to combine the `upstream` classpath and `compile` classpath into a single `classPath task, and now that we have multiple classpath folders, `assembly` needs to merge all of them before generating the jar

Overall, it's a lot of code, but relatively straightforward: `compile` takes the Java source code and passes them to a `javac` subprocess, `assembly` takes multiple folders full of compiled classfiles and passing them to a `jar` subprocesses

This results in the target graph on the right, with the dashed arrows coming into `compile` and `classPath` indicating the unknown inputs from `def upstream`/`def moduleDeps` that we might not know yet

3.3 DIY Java Modules

```scala
import mill._
trait DiyJavaModule extends Module{
  def moduleDeps: Seq[DiyJavaModule] = Nil
  def mainClass: T[Option[String]] = None
  def upstream: T[Seq[PathRef]] = T{
    T.traverse(moduleDeps)(_.classPath)().flatten
  }
  def sources = T.source(millSourcePath / "src")
  def compile = T {
    val allSources = os.walk(sources().path)
    val cpFlag = upstream().map(_.path).mkString(":")
    os.proc("javac", "-cp", cpFlag, allSources, "-d", T.dest).call()
    PathRef(T.dest)
  }
  def classPath = T{ Seq(compile()) ++ upstream() }
  def assembly = T {
    for(cp <- classPath()) os.copy(cp.path, T.dest, mergeFolders = true)

    val mainFlags = mainClass().toSeq.flatMap(Seq("-e", _))
    os.proc("jar", "-c", mainFlags, "-f", T.dest / s"out.jar", ".")
      .call(cwd = T.dest)

    PathRef(T.dest / s"out.jar")
  }
}
```

```scala
object foo extends DiyJavaModule {
  def moduleDeps = Seq(bar)
  def mainClass = Some("foo.Foo")

  object bar extends DiyJavaModule
}

object qux extends DiyJavaModule {
  def moduleDeps = Seq(foo)
  def mainClass = Some("qux.Qux")
}
```

`DiyJavaModule` can be used by instantiating it: just `object foo extends DiyJavaModule` while implementing or overriding anything we want. Same as we saw when using the built-in `ScalaModule` traits. Modules can be siblings like `foo` and `qux`, or nested like with `foo.bar` inside `foo`.

3.3 DIY Java Modules

```
import mill._
trait DiyJavaModule extends Module{
  def moduleDeps: Seq[DiyJavaModule] = Nil
  def mainClass: T[Option[String]] = None
  def upstream: T[Seq[PathRef]] = T{
    T.traverse(moduleDeps)(_.classPath)().flatten
  }
  def sources = T.source(millSourcePath / "src")
  def compile = T {
    val allSources = os.walk(sources().path)
    val cpFlag = upstream().map(_.path).mkString(":")
    os.proc("javac", "-cp", cpFlag, allSources, "-d", T.dest).call()
    PathRef(T.dest)
  }
  def classPath = T{ Seq(compile()) ++ upstream() }
  def assembly = T {
    for(cp <- classPath()) os.copy(cp.path, T.dest, mergeFolders = true)

    val mainFlags = mainClass().toSeq.flatMap(Seq("-e", _))
    os.proc("jar", "-c", mainFlags, "-f", T.dest / s"out.jar", ".")
      .call(cwd = T.dest)

    PathRef(T.dest / s"out.jar")
  }
}
```

```
object foo extends DiyJavaModule {
  def moduleDeps = Seq(bar)
  def mainClass = Some("foo.Foo")

  object bar extends DiyJavaModule
}

object qux extends DiyJavaModule {
  def moduleDeps = Seq(foo)
  def mainClass = Some("qux.Qux")
}

> ./mill show qux.assembly
".../out/qux/assembly.dest/qux.jar"
```

That’s all we need to start constructing Java assemblies using our `DiyJavaModule`s

3.3 DIY Java Modules

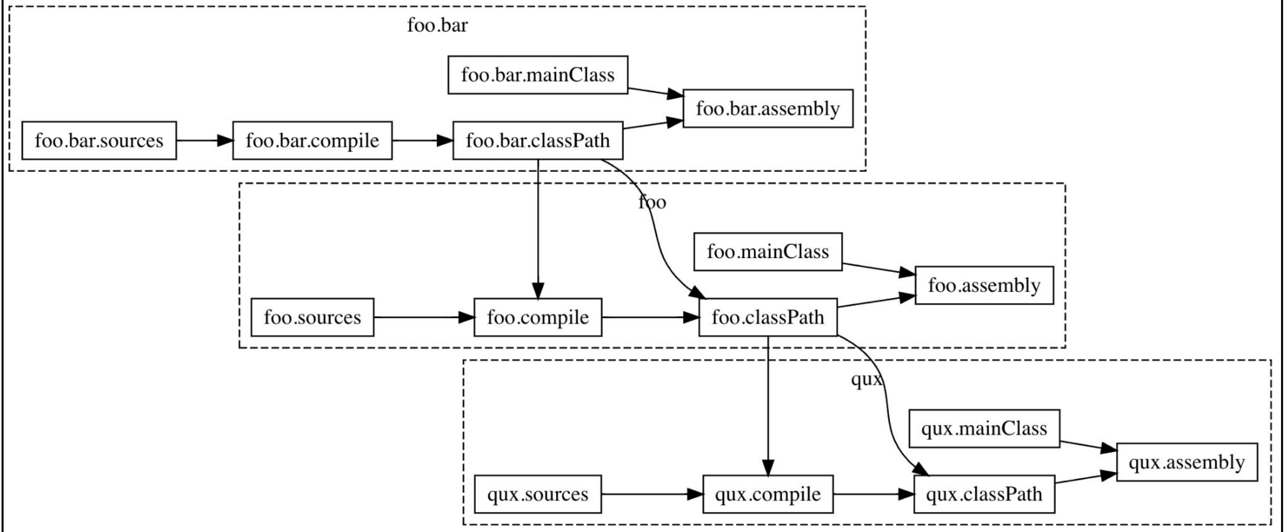
```scala
import mill._
trait DiyJavaModule extends Module{
  def moduleDeps: Seq[DiyJavaModule] = Nil
  def mainClass: T[Option[String]] = None
  def upstream: T[Seq[PathRef]] = T{
    T.traverse(moduleDeps)(_.classPath)().flatten
  }
  def sources = T.source(millSourcePath / "src")
  def compile = T {
    val allSources = os.walk(sources().path)
    val cpFlag = upstream().map(_.path).mkString(":")
    os.proc("javac", "-cp", cpFlag, allSources, "-d", T.dest).call()
    PathRef(T.dest)
  }
  def classPath = T{ Seq(compile()) ++ upstream() }
  def assembly = T {
    for(cp <- classPath()) os.copy(cp.path, T.dest, mergeFolders = true)

    val mainFlags = mainClass().toSeq.flatMap(Seq("-e", _))
    os.proc("jar", "-c", mainFlags, "-f", T.dest / s"out.jar", ".")
      .call(cwd = T.dest)

    PathRef(T.dest / s"out.jar")
  }
}
```

```scala
object foo extends DiyJavaModule {
  def moduleDeps = Seq(bar)
  def mainClass = Some("foo.Foo")

  object bar extends DiyJavaModule
}

object qux extends DiyJavaModule {
  def moduleDeps = Seq(foo)
  def mainClass = Some("qux.Qux")
}
```

```
> ./mill show qux.assembly
".../out/qux/assembly.dest/qux.jar"

> java -jar out/qux/assembly.dest/qux.jar
Foo.value: 31337
Bar.value: 271828
Qux.value: 9000
```

Which we can then run separately, outside of Mill. The source code for `foo/src/`, `foo/bar/src/` and `qux/src/` are not shown, for this example they are trivial. But the point of this example is not the Java source code itself, but the way we have written our own DiyJavaModule trait from first principles, instantiated it a few times, and can now use it to compile and package a multi-module Java codebase. Just like that!

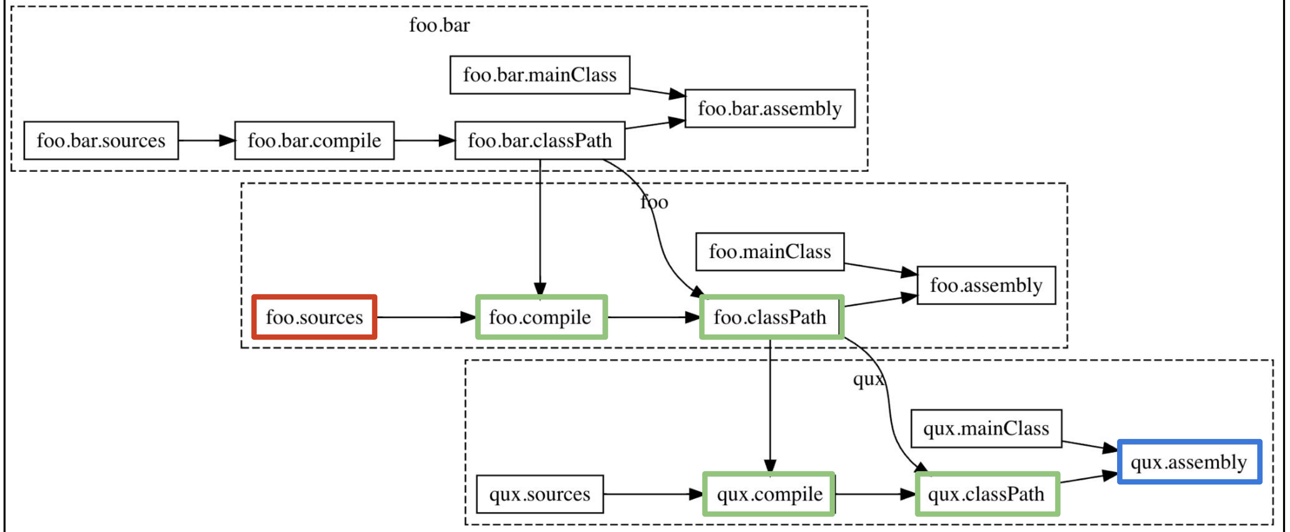
3.3 DIY Java Modules



These is the final build graphs, after we've instantiated and wired together our three `DiyJavaModule`s. The concrete modules `foo.bar`, `foo`, and `qux` all have their edges going into `compile` and classPath` wired up to the upstream module's `classPath` task.

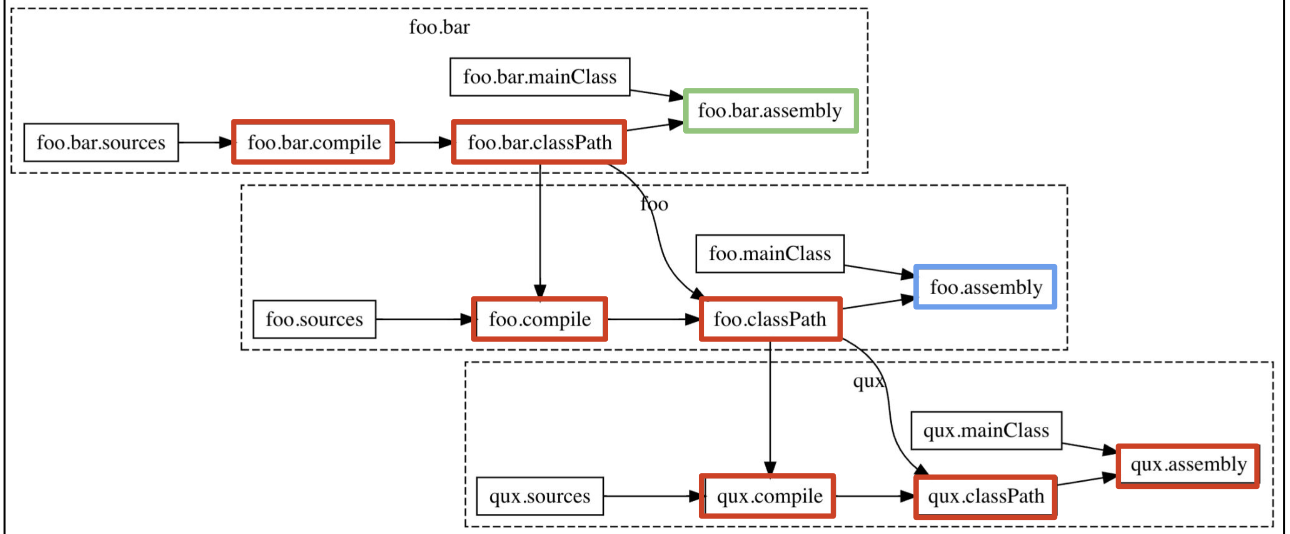
Like any other set of Targets and Modules, the compilation and packaging of the Java code via `DiyJavaModule` is:

3.3 DIY Java Modules

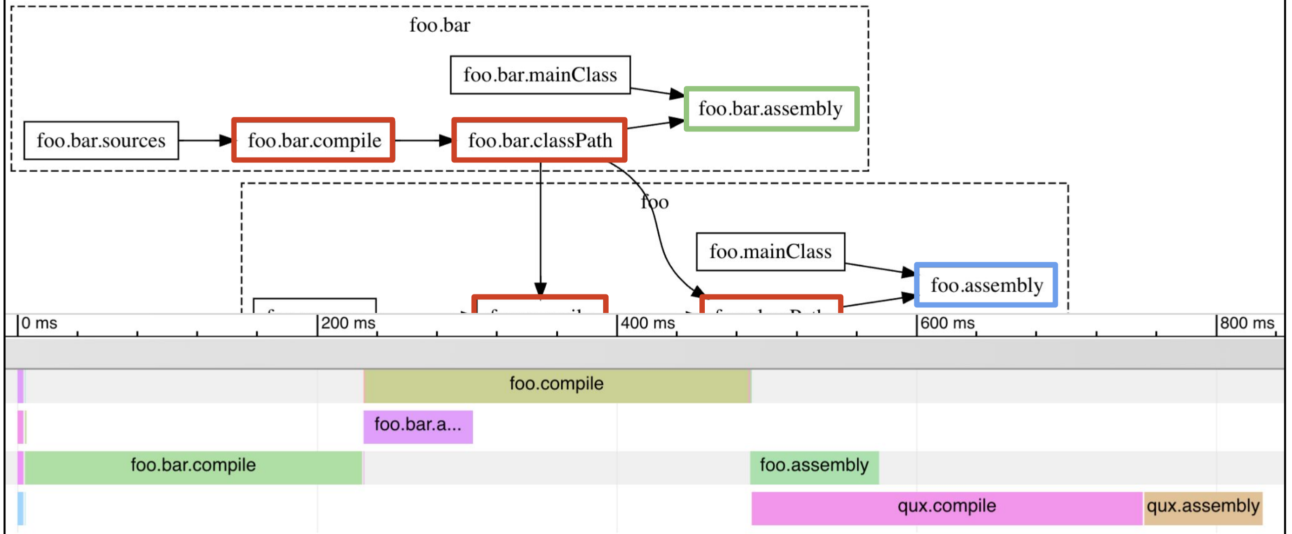


- Incremental: if you change a file in `foo/src/` and run `qux.assembly`, `foo.compile` and `qux.compile` will be re-evaluated, but `foo.bar.compile` will not as it does not transitively depend on `foo.sources`

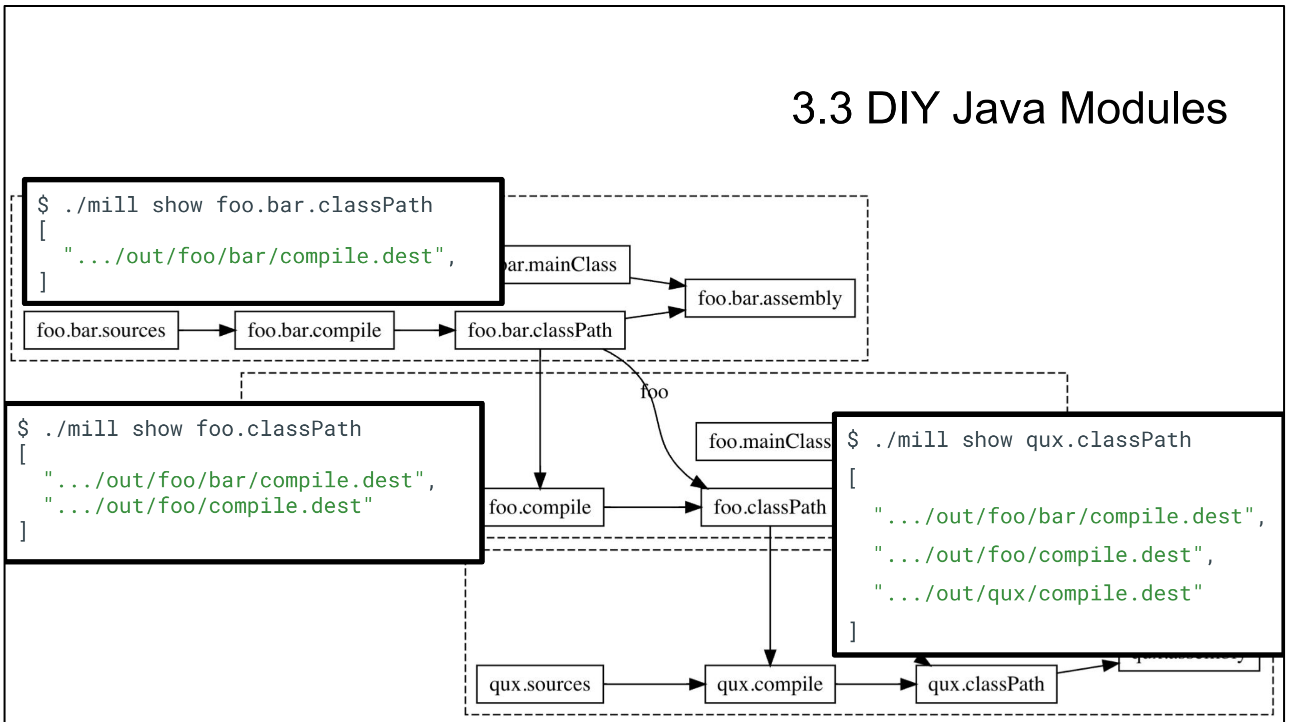
3.3 DIY Java Modules



- Parallelizable: if you give Mill 3 threads, \`foo.bar.compile\`/\`foo.compile\`/\`qux.compile\` will have to happen one after the other due to the dependencies between them, but \`foo.bar.assembly\`/\`foo.assembly\`/\`qux.assembly\` can happen in parallel because they have no dependencies

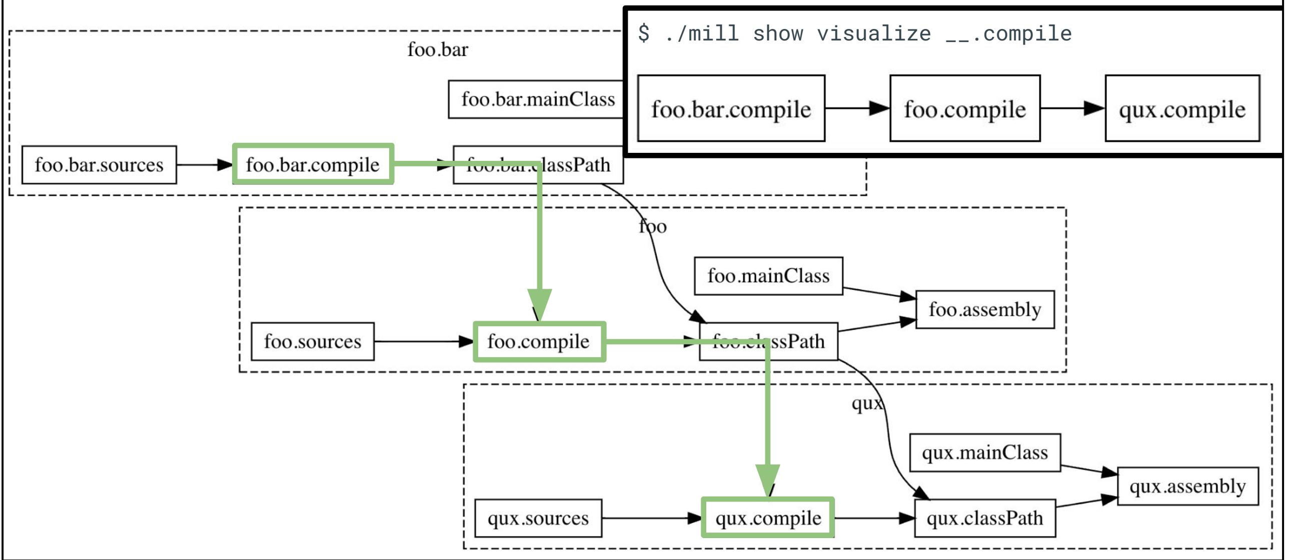


Mill even produces parallel profiles you can load into any chrome browser, so you can see exactly which tasks are running in serial or in parallel with one another, and how long they took



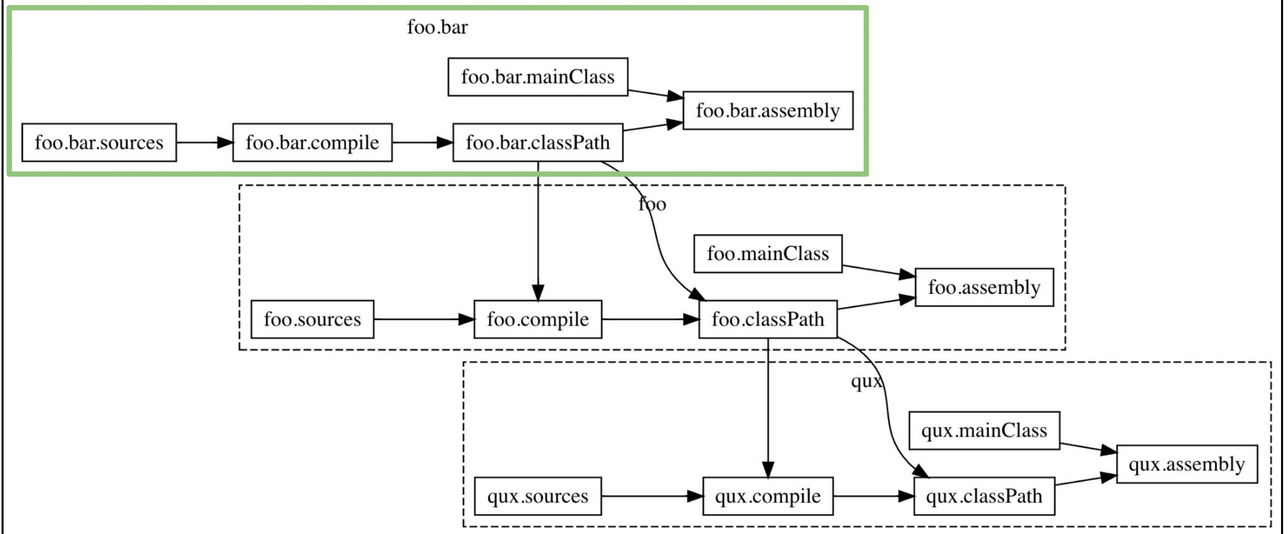
- Introspectable: you can run `./mill inspect` or `./mill show` to see their metadata and return value. Very useful for debugging purposes. You can also find this data and files on disk at each target's respective filesystem paths

3.3 DIY Java Modules

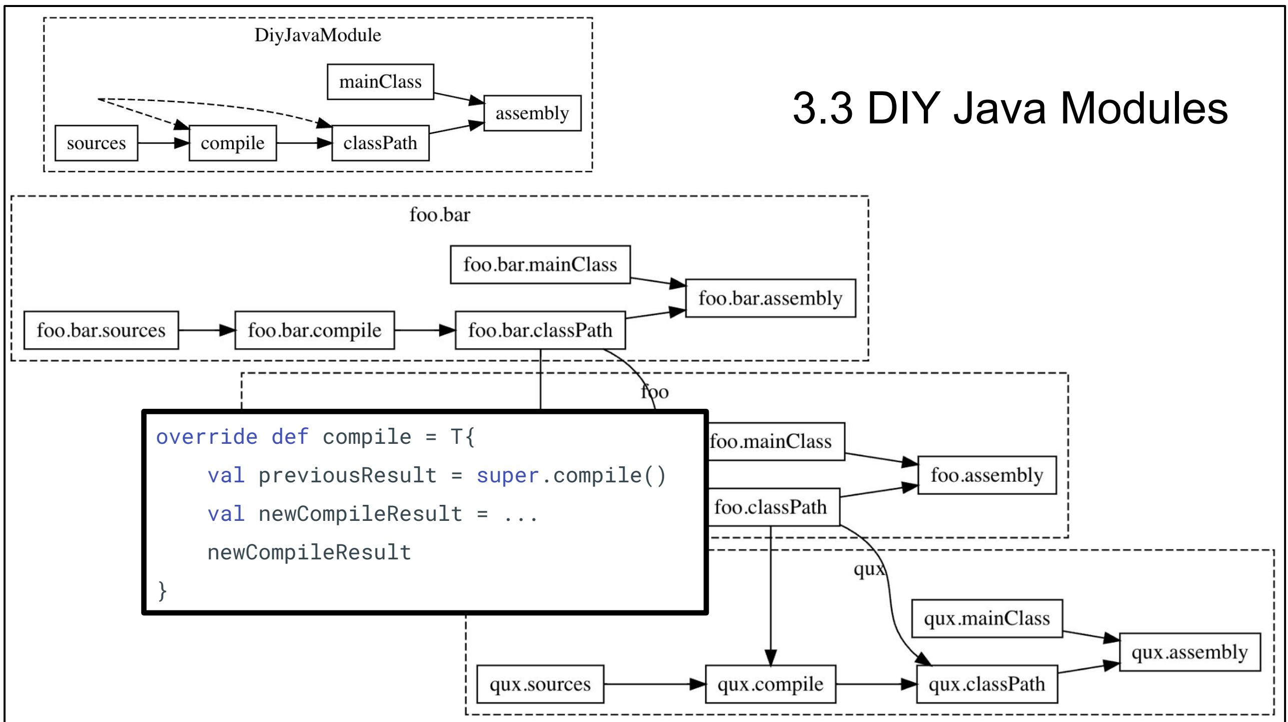


And you can easily generate visualizations of subsets of the call graph to help you understand what's going on, without being distracted by all the detail

3.3 DIY Java Modules



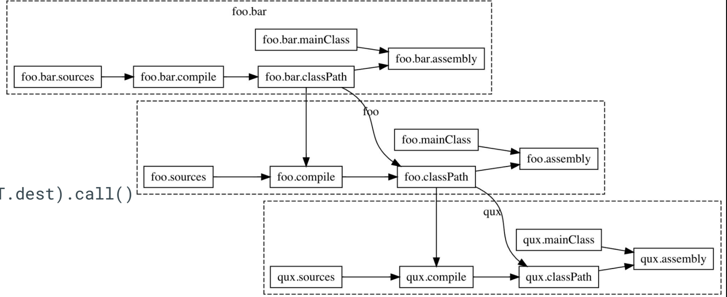
- Re-usable: you can instantiate as many `DiyJavaModules` as you want: top-level modules, sibling modules, sub-modules, etc.



- Extensible: you can `override` any target you want with a custom implementation, you can call `super` if necessary. Perhaps you want `foo.compile` to perform some bytecode re-writing, maybe you want `qux.assembly` to use ProGuard to remove unused classfiles: these can be easy done using `override` and `super`

```scala
import mill._
trait DiyJavaModule extends Module{
  def moduleDeps: Seq[DiyJavaModule] = Nil
  def mainClass: T[Option[String]] = None
  def upstream: T[Seq[PathRef]] = T{
    T.traverse(moduleDeps)(_.classPath)().flatten
  }
  def sources = T.source(millSourcePath / "src")
  def compile = T {
    val allSources = os.walk(sources().path)
    val cpFlag = upstream().map(_.path).mkString(":")
    os.proc("javac", "-cp", cpFlag, allSources, "-d", T.dest).call()
    PathRef(T.dest)
  }
  def classPath = T{ Seq(compile()) ++ upstream() }
  def assembly = T {
    for(cp <- classPath()) os.copy(cp.path, T.dest, mergeFolders = true)

    val mainFlags = mainClass().toSeq.flatMap(Seq("-e", _))
    os.proc("jar", "-c", mainFlags, "-f", T.dest / s"out.jar", ".")
      .call(cwd = T.dest)

    PathRef(T.dest / s"out.jar")
  }
}
```

3.3 DIY Java Modules



It’s worth calling out again that the `build.sc` file that the user writes does not contain any logic at all related to incremental computation, parallelization, introspectability, extensibility, or re-use. The user only needs to write the build logic unique to their specific use case: where do input files live, what subprocesses do I shell out to, with what flags. And Mill will take care of everything else that you would inevitably want out of a build tool.

3.3 DIY Java Modules

```scala
// build.sc
import mill._, scalalib._

trait MyModule extends ScalaModule {
  def scalaVersion = "2.13.11"
}

object foo extends MyModule {
  def moduleDeps = Seq(bar)
  def ivyDeps = Agg(ivy"com.lihaoyi::mainargs:0.4.0")
}

object bar extends MyModule {
  def ivyDeps = Agg(ivy"com.lihaoyi::scalatags:0.8.2")
}
```

```scala
object foo extends DiyJavaModule {
  def moduleDeps = Seq(bar)
  def mainClass = Some("foo.Foo")

  object bar extends DiyJavaModule
}

object qux extends DiyJavaModule {
  def moduleDeps = Seq(foo)
  def mainClass = Some("qux.Qux")
}
```

Earlier in this talk, we saw an example with two modules extending the Mill built-in `ScalaModule` `trait` that we wired up with `moduleDeps`. With `DiyJavaModule`, we have now implemented our own re-usable module trait from first principles that can be wired up the same way and perform some of the same tasks of compiling and packaging a multi-module Java codebase.

While Mill comes with built-in support for Java and Scala, you can use it to define build pipelines for other things as well: generating a static blog, creating PDFs, hashing and preparing static web assets for deployment on CDN. As a user of Mill, you can write whatever logic you want, in normal Scala code, using whatever normal JVM libraries or subprocess utilities you have available. Mill will whatever you throw at it and turn it into a parallel, incremental, extensible build pipeline!

A Deep Dive into the Mill Scala Build Tool

1. Why Build Tools Are Hard

2. Getting Started with Mill

3. Mill Fundamentals

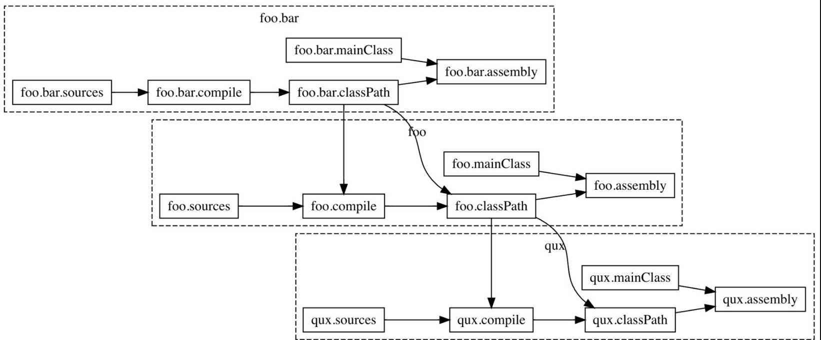
4. **Why Mill Works**

That's it for Mill's Fundamentals.

Now, we've talked at length about *how* Mill works. Now let's discuss *why* it works, as an intuitive and understandable build tool.

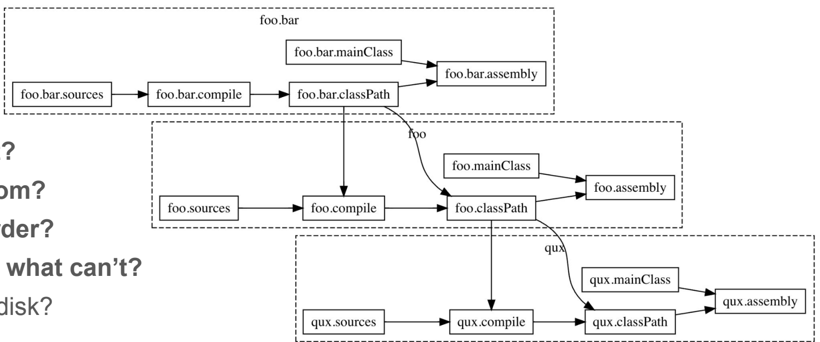
This comes down to Mill's two core concepts - the Task Graph and the Module Tree - and how they are embedded into the core syntax and semantics of the Scala language.

4.1 The Task Graph



The first core concept behind Mill is the Task Graph. Consider again the graph from our DIY Java Modules example earlier

This graph is captured ahead of time, and lets us do interesting things around caching, introspection, parallelization, and so on.



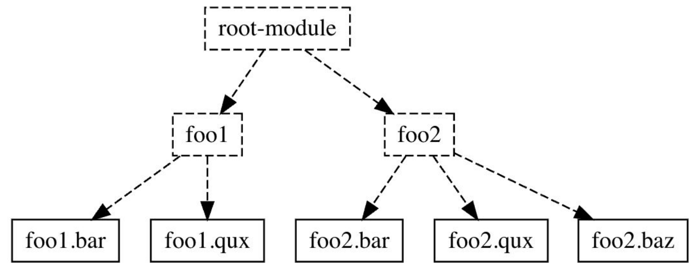
4.1 The Task Graph

1. **What tasks depends on what?**
2. **Where do input files come from?**
3. **What needs to run in what order?**
4. **What can be parallelized and what can't?**
5. Where can tasks read/write to disk?
6. How are tasks cached?
7. How are tasks run from the CLI?
8. How are cross-builds (across different configurations) handled?
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build?
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things?
15. How is in-memory caching handled?

Re-visiting the big list of questions *What's Hard about Build Tools* from early, we can see that the Task Graph answers the first few of them with a single data structure:

- **What tasks depends on what?**: the edges of a task graph
- **Where do input files come from?**: the "roots" of the task graph
- **What needs to run in what order?**: topological traversal of the task graph, serial or parallel
- **What can be parallelized and what can't?**: whether they transitively depend on each other in the task graph

Most developers are already familiar with graphs, and should find these answers relatively "obvious".



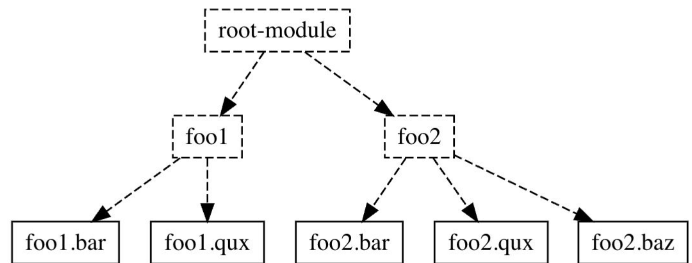
4.2 The Module Tree

1. ~~What tasks depends on what?~~
2. ~~Where do input files come from?~~
3. ~~What needs to run in what order?~~
4. ~~What can be parallelized and what can't?~~
5. Where can tasks read/write to disk?
6. How are tasks cached?
7. How are tasks run from the CLI?
8. How are cross-builds (across different configurations) handled?
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build?
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things?
15. How is in-memory caching handled?

The second core concept behind Mill is the Module Tree. Consider again this example tree from earlier.

Modules are just objects that extend mill.Module. These form a tree and can contain targets at the leaves.

4.2 The Module Tree

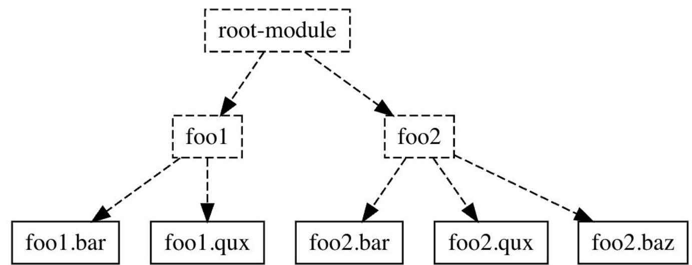


1. ~~What tasks depends on what?~~
2. ~~Where do input files come from?~~
3. ~~What needs to run in what order?~~
4. ~~What can be parallelized and what can't?~~
5. Where can tasks read/write to disk?
6. How are tasks cached?
7. How are tasks run from the CLI?
8. How are cross-builds (across different configurations) handled?
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build?
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things?
15. How is in-memory caching handled?

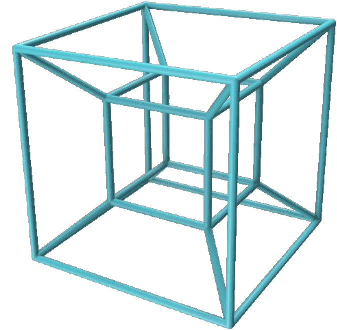
```
my-project/
    foo1/
        src/
            ...
        resources/
            ...
    foo2/
        src/
            ...
        resources/
            ...
```

The structure of the Module Tree intentionally mirrors the tree structure of your filesystem. Basically every software project is managed as files and folders on disk, and by following this Mill makes it very easy to make your Mill build match up with how you are *already* structuring your codebase.

4.2 The Module Tree

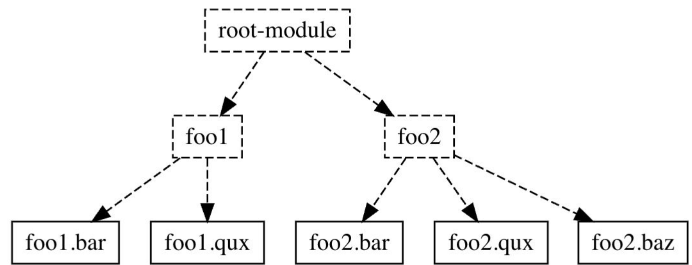


1. ~~What tasks depends on what?~~
2. ~~Where do input files come from?~~
3. ~~What needs to run in what order?~~
4. ~~What can be parallelized and what can't?~~
5. Where can tasks read/write to disk?
6. How are tasks cached?
7. How are tasks run from the CLI?
8. How are cross-builds (across different configurations) handled?
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build?
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things?
15. How is in-memory caching handled?



It's worth calling out that not every build tool makes the same decision here: SBT is famous for having a Four Dimensional Hyper-Matrix as its core data structure. I included a visualization I found online on the slide in case you don't know what exactly a four-dimensional hyper-matrix looks like.

But while most developers are familiar with trees, not nearly as many are familiar with four-dimensional hyper-matrices! Nobody stores their source code inside a filesystem tesseract. And so being structured as a tree really helps Mill in terms of understandability.



4.2 The Module Tree

1. ~~What tasks depends on what?~~
2. ~~Where do input files come from?~~
3. ~~What needs to run in what order?~~
4. ~~What can be parallelized and what can't?~~
5. **Where can tasks read/write to disk?**
6. **How are tasks cached?**
7. **How are tasks run from the CLI?**
8. **How are cross-builds (across different configurations) handled?**
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build?
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things?
15. How is in-memory caching handled?

In addition to following existing conventions, the Module Tree ends up answering the next few questions from our big list:

- **Where can tasks read/write to disk?:** In a `.dest` folder matching the task's position in the tree
- **How are tasks cached?:** In a `.json` file matching the task's position in the tree
- **How are tasks run from the CLI?:** Via their dot-delimited fully-qualified name or path in the tree
- **How are cross-builds handled?:** by having multiple modules in the tree, one for each cross version

Developers are already familiar with trees in general and the filesystem tree in particular. So Mill make full use of its tree structure to help answer these questions in a familiar and "obvious" manner

4.2 Embedding into Scala

1. ~~What tasks depends on what?~~
2. ~~Where do input files come from?~~
3. ~~What needs to run in what order?~~
4. ~~What can be parallelized and what can't?~~
5. ~~Where can tasks read/write to disk?~~
6. ~~How are tasks cached?~~
7. ~~How are tasks run from the CLI?~~
8. ~~How are cross-builds (across different configurations) handled?~~
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build?
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things?
15. How is in-memory caching handled?

```scala
import mill._, scalalib._
object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    Seq(PathRef(T.dest))
  }
}
```

The last and final reason Mill works so well is that it embeds itself into the Scala language smoothly enough that you may not even notice there is magic going on. Consider again the example build.sc from earlier:

4.2 Embedding into Scala

```scala
import mill._, scalalib._
object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    Seq(PathRef(T.dest))
  }
}
```

1. ~~What tasks depends on what?~~
2. ~~Where do input files come from?~~
3. ~~What needs to run in what order?~~
4. ~~What can be parallelized and what can't?~~
5. ~~Where can tasks read/write to disk?~~
6. ~~How are tasks cached?~~
7. ~~How are tasks run from the CLI?~~
8. ~~How are cross-builds (across different configurations) handled?~~
9. **How do I define my own custom tasks?**
10. **How do tasks pass data to each other?**
11. **How to manage the repetition in a build?**
12. **What is a "Module"? How do they relate to "Tasks"?**
13. **How do you customize a task or module to do something different?**
14. **What APIs do tasks use to actually do things?**
15. **How is in-memory caching handled?**

Mill uses its seamless embedding to answer the last few questions from our list:

- **How do I define my own custom tasks?**: write a Scala def and call other defs
- **How do tasks pass data to each other?**: via their Scala return value
- **How to manage the repetition inherent in a build?**: defining Scala traits and re-using them
- **What is a "Module"? How do they relate to "Tasks"?**: Modules are just Scala `object`s that contain `def`s which are Tasks
- **How do you customize a task or module to do something different?**: override and super, or even stackable traits
- **What APIs do tasks use to actually do things?**: The same libraries you might use in normal code! OS-Lib, requests-scala. Data is serialized using uPickle. Command line parameters are parsed using MainArgs
- **How is in-memory caching handled?**: via long-lived in-memory stateful `object`s, where Mill helps managing their instantiation and invalidation

Earlier in this talk we discussed how pure functional programs have a lot of similarities to the build graph, but are not enough. It turns out that the object oriented side of Scala has a lot to give as well: the way we use objects-as-namespaces, build a reference tree of objects, use overrides for customization, trait re-use and so on. These are all right out of the object-oriented playbook.

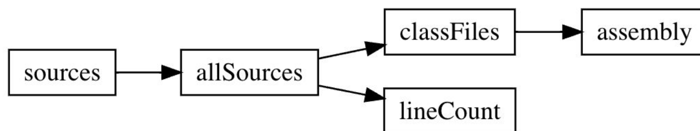
Functional Programming turns out to be just one facet of a build system: an important

one, but one that shares the stage with Object-Oriented Programming, and other parts of the Scala language in order to make the Mill build system *work*

With just three core concepts, we've answered our entire laundry-list of questions that any build tool needs answers for. But that's not all!

4.2 Embedding into Scala

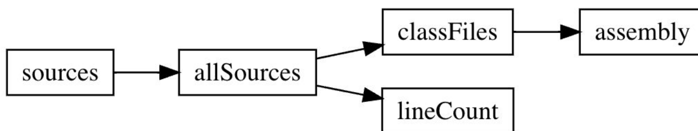
```
def sources = T.source { ... }

def allSources = T { ...sources()... }

def lineCount= T { ...allSources()... }

def classFiles = T {  ...allSources()...}

def assembly = T { ...classFiles()...}
```

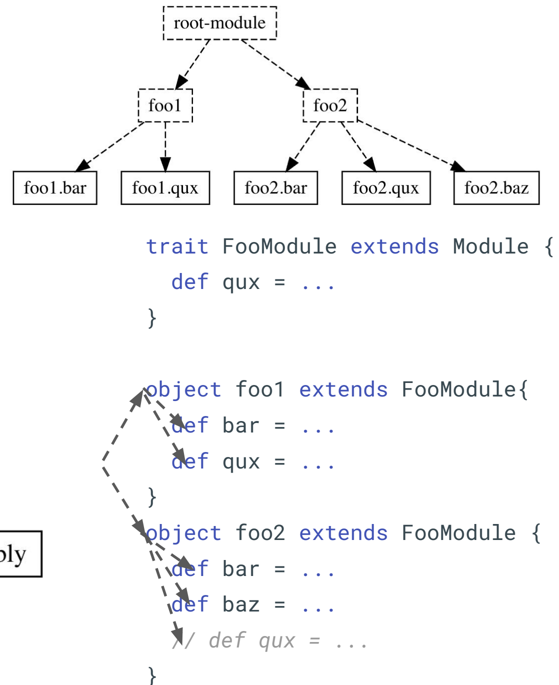


The seamless embedding into Scala means that the Task Graph matches the method Call Graph

4.2 Embedding into Scala

```
def sources = T.source { ... }

def allSources = T { ...sources()... }

def lineCount= T { ...allSources()... }

def classFiles = T {  ...allSources()...}

def assembly = T { ...classFiles()...}
```





And the Module Tree matches the object Reference Graph. Both of these are concepts that are already familiar to any programmer with prior experience in any almost programming language. Python, Java, Javascript, C++: they all have call graphs and reference graphs too! Programmers already *know* this stuff

So what's left as required-learning for someone to use Mill?

4.2 Embedding into Scala

1. What tasks depends on what?
2. Where do input files come from?
3. What needs to run in what order?
4. What can be parallelized and what can't?
5. Where can tasks read/write to disk?
6. How are tasks cached?
7. How are tasks run from the CLI?
8. How are cross-builds (across different configurations) handled?
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build?
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize a task or module to do something different?
14. What APIs do tasks use to actually do things?
15. How is in-memory caching handled?

```
object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }

  override def resources = T{
    os.write(T.dest / "line-count.txt", "" + lineCount())
    Seq(PathRef(T.dest))
  }
}
```

It turns out, the only pre-requisite for knowing how to use Mill is knowing how to write code. That's all you need!

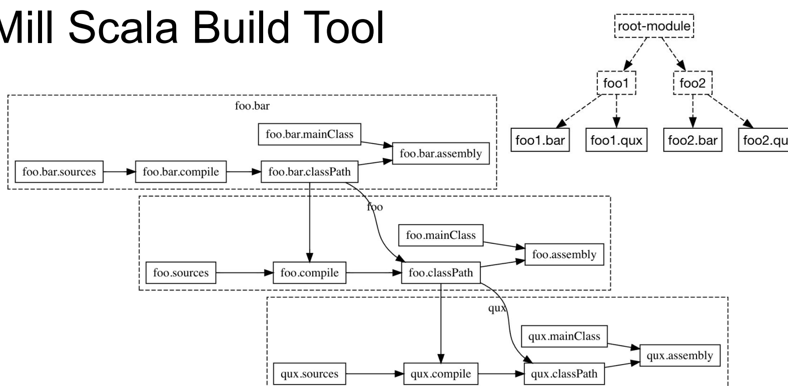
That means that an existing programmer can begin using Mill, learn approximately nothing, and immediately be familiar with how Mill answers this whole laundry list of questions. This lets them immediately focus on exactly the logic that is unique to their build, while in other build tool they would first need to read chapters and chapters of documentation before they can figure out how to even *begin* doing things.

People talk a lot about build tools, but the discussion is often focused on very superficial aspects: is Scala better than Groovy? Is YAML better than code? What about TOML? But once you crack the surface, it turns out that what really matters is how the build tool comes up with answers for every single entry in this rather long list of questions, and how it avoids a developer needing to read an entire book's worth of documentation before feeling comfortable working with it. Not every build tool succeeds in this, but Mill, by and large, does!

This heavy re-use of your prior programming knowledge is why people like using Mill, and why its design is uniquely intuitive and familiar even in the large solution space of build tooling and build automation!

A Deep Dive into the Mill Scala Build Tool

1. Why Build Tools Are Hard
2. Getting Started with Mill
3. Mill Fundamentals
4. Why Mill Works




```scala
import mill._, scalalib._
object foo extends RootModule with ScalaModule {
  def scalaVersion = "2.13.11"

  def lineCount = T{
    allSourceFiles().map(f => os.read.lines(f.path).size).sum
  }
```

https://mill-build.com/

In this talk, we did a deep dive into why Mill works as a build tool

We started off by looking at why build tools are hard, explored getting started with Mill for Scala, dug into the fundamental concepts that make up the Mill build tool, and lastly discussed why Mill can feel so familiar to a developer based even if they have never seen it before.

Among all the attempts at writing alternate Scala build tools to replace SBT, Mill is the one of the only ones that can confidently be used today. Major projects like Coursier and Scala-CLI are built using Mill, as is the whole com-lihaoyi ecosystem. Mill *works*!

But Mill is more than just a "better SBT". It is unique as a build tool that leverages everything you already know about programming, seamlessly enough to feel familiar from the moment you pick it up. A tool that lets you learn nothing, have a great time configuring your build system up front, and enjoy continuing to maintain it months and years into the future. That's why you should take away from this deep dive into the Mill Scala Build Tool!