Dependency Management in Pants

status: draft
author: gmalmquist@squareup.com
date: 8 January 2016

This document is shared publicly outside of Square

Summary

A design for implementing the equivalent of Maven’s `<dependencyManagement>` stanza in Pants. See PANTS-207.

Review of first-pass implementation is here.

Background

In maven, there is a <dependencyManagement> stanza in the parent pom.xml that all other poms inherit from, which looks like:

```
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.foo.bar</groupId>
        <artifactId>some-artifact</artifactId>
        <version>1.2.3</version>
      </dependency>
      <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-collections4</artifactId>
        <version>4.1</version>
      </dependency>
  </dependencyManagement>
```

This actually accomplishes two use cases.

1. Standardizing Versions of Direct External Dependencies

The pom files for actual modules are then able to declare dependencies on artifacts without specifying a particular version; the version is inferred (and therefore standardized):

```
<dependencies>
  <dependency>
    <groupId>com.foo.bar</groupId>
    <artifactId>some-artifact</artifactId>
  </dependency>
</dependencies>
```

This behavior is currently available in pants through the use of the 3rdparty pattern. By convention, external libraries are defined in `3rdparty/BUILD`, which looks like:

```
jar_library(name='org.apache.commons.commons-collections4',
  jars=[
    jar(org='org.apache.commons', name='commons-collections4', rev='4.1',)
  ],
)
```

These are then referenced in individual BUILD targets like so:

```
java_library(name='my-library',
  sources=rglobs('*.java'),
  dependencies=[
    '3rdparty:org.apache.commons.commons-collections4',
    3rdparty:org.foo.bar.some-artifact',
  ],
)
```

2. Pinning versions for *transitive* external dependencies

Maven's `<dependencyManagement>` stanza also has desirable side-effect of pinning the transitive dependencies for the entire repository to particular versions.

Pants has ways of excluding particular jars, but it is clunky and harder to use than just declaring that you want to use a particular version of an artifact across the entire repository. In general Pants uses the Ivy strategy of picking the artifact with the highest version. This can be partially overridden by using the `force` argument on a jar, however that only takes effect if that jar is depended on directly; if there is no dependency on the "correct" jar, it won't even be loaded into the build graph, and thus pants will ignore it.

Another consideration is that we want to support explicit overrides of the globally set version. In maven this is done simply by actually defining the `<version>` for a dependency in the module pom.xmls. In pants we currently do this by manually specifying a direct dependency on the version we want and using the force field.

How other monorepos deal with this issue

Other well known monorepo deployments like Buck at Facebook and Blaze/Bazel at Google don't deal with this problem. They manually vet external artifacts. In Buck and Bazel you can see this in their targets. Some targets support checking artifacts into the repo (buck prebuilt_jar(), bazel java_import()) reading other projects from another place on disk (bazel external dependencies) or by downloading artifacts from external repos intransitively (buck remote_file(), bazel maven_jar()).

At Twitter's Pants installation, they actually do not want this consistently across their repo. They allow different projects to intentionally use many different versions of the same jar (for example, they have > 5 versions of the guava library for various projects.) However, they are looking for a way to begin to enforce this across some subset of projects. We intend to collaborate with the other users of Pants on this design.

At Foursquare's Pants installation, they have replaced Ivy with a custom solution that figures out transitive dependencies needed by all projects. They have an out of band process to perform the transitive analysis. An administrator takes an explicit action to find all transitive dependencies when a new artifact reference is added. Conflicts are resolved manually and the resulting resolution is checked into the repo.

Potential Solutions

A. Designate a particular BUILD file as defining "authoritative" artifacts

We could add a global option called `--dependency-management`, which we could define in pants.ini as something like:

```
[DEFAULT]
dependency_management: '3rdparty/managed'
```

Pants would then treat all jar_library targets in all targets defined in or referenced as dependencies by targets in `3rdparty/managed/BUILD*` as "authoritative", and would use their versions to pin all transitive jar dependencies. This would cause all the jar_libraries in the file to be explicitly loaded, which would solve the problem of pants not noticing jars that aren't directly depended on.

A reasonable convention would be to define all jar_libraries in `3rdparty/BUILD`, and have a single target() object in `3rdparty/managed` which references all of them as dependencies. These libraries would only be downloaded if there is an explicit reference or a transitive reference to the library.

Individual targets could override the global "correct" dependency version by adding a direct dependency on a jar.

A potential downside of this approach is that it is truly global across the entire repository; in maven, you can technically use different sets of <dependencyManagement> for different modules, by declaring a different <parent> pom.

Conflicts between directly defined artifact dependencies and those defined in the authoritative build file could be handled the same way as is described in Solution B (using a global flag for different strategies).

B. Create a managed_dependencies target

Define a new `managed_dependencies()` target, that looks like:

```
# 3rdparty/managed/BUILD

managed_dependencies(name='managed',
  artifacts=[
    jar(org='...', name='...', rev='...', ext='...'),
    jar(org='...', name='...', rev='...', ext='...'),
    jar(org='...', name='...', rev='...', ext='...'),
    jar(org='...', name='...', rev='...', ext='...'),
  ],
)
```

To avoid unnecessary duplication, in actual jar_library targets the version (rev) could be omitted if it was present in a managed_dependencies target.

```
# 3rdparty/BUILD

jar_library(name='commons-io.commons-io',
  jars=[
    jar(org='commons-io', name='commons-io'), # Rev from managed_dependencies
  ],
)
```

Alternatively, the managed_dependencies artifacts field could accept the build file address of a jar_library as input, and read from that. I.e.:

```
# 3rdparty/managed/BUILD

managed_dependencies(name='managed',
  artifacts=[
    jar(org='...', name='...', rev='...', ext='...'),
    jar(org='...', name='...', rev='...', ext='...'),
    jar(org='...', name='...', rev='...', ext='...'),
    jar(org='...', name='...', rev='...', ext='...'),
    '3rdparty:commons-io.commons-io',
    '3rdparty:com.google.guava.guava',
  ],
)
```

Any targets which depend on the dependency_management target would be forced to use the fixed versions of the artifacts it specifies when resolving transitive dependencies.

```
java_library(name='lib',
  sources=...,
  dependencies=[
    '3rdparty/managed',
  ],
)
```

This would allow different targets to use different sets of managed dependencies, if necessary.

To avoid the cumbersome need to specify the dependency management target as a dependency of almost every target in the repository, a default target could be specified in pants.ini:

```
[dependency-management]
default_target: '3rdparty/managed'
```

Which would apply to any target which did not transitively depend on any other managed_dependencies target.

Dependency management targets could feasibly be constructed modularly:

```
# 3rdparty/managed/BUILD

managed_dependencies(name='base',
  artifacts=[
    jar(org='commons-io', name='commons-io', rev='2.4'),
    jar(org='com.foo.bar', name='my-artifact', rev='1.2.3'),
  ],
)

managed_dependencies(name='with-guava',
  artifacts=[
    jar(org='com.google.guava', name='guava', rev='15.0'),
  ],
  dependencies=[
    ':base',
  ],
)

managed_dependencies(name='different-foobar',
  artifacts=[
    # Override dependency's version of com.foo.bar.my-artifact
    jar(org='com.foo.bar', name='my-artifact', rev='4.5.6'),
  ],
  dependencies=[
    ':base',
```

```
  ],
)
```

If a target directly depended on an artifact with a version at odds with the one defined in its corresponding managed_dependencies target, Pants' behavior should be configurable with another flag, choosing between:

- `FAIL`: Failing fast with a dependency conflict error message.
- `USE_DIRECT`: Logging a warning, and using the version of the direct dependency.
- `USE_MANAGED`: Logging a warning, and using the version in dependency management.
- `USE_NEWER`: Logging a warning, and using the newest version between the two.
- `SILENT_USE_DIRECT`: Silently using the version of the direct dependency.
- `SILENT_USE_MANAGED`: Silently using the version in dependency management.
- `SILENT_USE_NEWER`: Silently using the newer version between the two. (I think this would be terrible, and maybe shouldn't even be an option).

`USE_DIRECT` is probably the most sensible default, but others in the open-source community may disagree.

This might look like:

```
[dependency-management]
# A more concise variable name would be better
direct_dependency_conflict_strategy: USE_DIRECT
```

A single target should not be able to depend on multiple managed_dependencies targets that do not depend on each other, as that could potentially introduce strange conflicts with ambiguous and unstable resolutions.

For example, this is fine (because the artifacts in B unambiguously override those in A):

```
managed_dependencies(name='A',
  artifacts=[...],
)

managed_dependencies(name='B',
  artifacts=[...],
  dependencies=[':A'],
)

target(name='T',
  dependencies=[':B'],
)
```

But this should throw an error:

```
managed_dependencies(name='A',
  artifacts=[...],
)

managed_dependencies(name='B',
  artifacts=[...],
```

```
)

target(name='T',
  dependencies=[
    ':A',
    ':B',
  ],
)
```

Design Choices

Option B is probably best; it covers a superset of the use-cases of Option A, without adding significant complexity, and allows for an equally simple way to specify global dependency management in the basic use-case.

Neither option affects the current default behavior of Pants' jar resolution.

Option B could also be partially implemented (ie just for the global case), with the option to extend functionality to support having multiple managed_dependency targets later.

Design Details

In open-source pants:

- Create a DependencyManagement subsystem to store the global default dependency management target.
- Create a ManagedDependencies target to store a set of preferred artifacts.
- Change this logic in ivy_utils.py to match the logic described in Choice B:

  ```
  # As it turns out force is not transitive - it only works for dependencies pants knows about
  # directly (declared in BUILD files - present in generated ivy.xml). The user-level ivy docs
  # don't make this clear [1], but the source code docs do (see isForce docs) [2]. I was able to
  # edit the generated ivy.xml and use the override feature [3] though and that does work
  # transitively as you'd hope.
  #
  # [1] http://ant.apache.org/ivy/history/2.3.0/settings/conflict-managers.html
  # [2] https://svn.apache.org/repos/asf/ant/ivy/core/branches/2.3.0/
  #     src/java/org/apache/ivy/core/module/descriptor/DependencyDescriptor.java
  # [3] http://ant.apache.org/ivy/history/2.3.0/ivyfile/override.html
  overrides = [cls._generate_override_template(dep) for dep in dependencies if dep.force]
  ```

  This should only require changing the contents of the `overrides` list; the ivy template should be ok as-is.
- The dependency management targets and settings *must* be included in the fingerprint for the jar resolution.
- Give some kind of optional output (console/report) when there is a transitive dependency that is resolved through a managed_dependendies() definition but not listed in managed_dependencies()

In our java repo:

- Modify our build generation to create `3rdparty/managed/BUILD` with a single `managed_dependencies` target, created by parsing the `<dependencyManagement>` stanza of `parents/base/pom.xml`. This should be easy.

Resources

[Proposal for Isolating and Freezing External Artifact Resolution]