Atypical Plugin Use Cases

[DESIGN DISCUSSION for 2.1] Atypical Plugin Use-Cases

The Parent Problem

When a child project is referenced by a parent POM, and that parent POM defines plugins to be run, those plugins may execute before or after the child modules build, depending on the child modules' usage of the <parent/> section of their POMs.

If the child POM references its parent using a <parent/> section, Maven 2.0.x views that as a dependency of the project, which means if that parent POM is present in the current reactor, it will be sorted to build **ahead** of the child module. **HOWEVER**, if the child POM fails to declare its <parent/>, it will be sorted to build ahead of the parent POM, other things being equal. This can lead to very subtle and unintuitive behaviors, such as:

- When children use <parent/> sections, assemblies created in the parent build that reference <moduleBinaries/> which are not yet present, resulting in the assembly plugin failing (MASSEMBLY-94)
- <module/> specifications without a corresponding <parent/> in the referenced POM will not fail, but may result in the child being built before the parent.

Proposed changes for 2.1

Force parent-first semantics; either a module reference OR a parent reference would be used to establish a parent-child dependency relationship, instead of just the parent reference now. This just leaves open the question of how to force some configuration in the parent build to run AFTER all children are built, as in the case of MASSEMBLY-94.

Backward-compat considerations

There really shouldn't be any problem documenting this break on compatibility. It's more of a technical break, since only those who are intimately familiar with Maven's parent-child dependency logic would be able to exploit this, and then only under certain circumstances. The most notable of these being the use of a orchestration-only top-level POM that lists a bunch of modules, among them a parent POM that is referenced in the <parent/> section of all the other module POMs. In this case, the orchestrating POM would be built last under 2.0.x, but first under 2.1.

Aggregation

Aggregation (using the @aggregator mojo annotation) was originally designed to give plugin developers the opportunity to summarize some part of the result of all builds in the current reactor, and produce its own output. As such, it was designed to run at most once per build, and have access to the list of projects in the current reactor via the ${reactorProjects} plugin-parameter expression. However, this design does not address all of the use cases for summary in the build; far from it. In fact, aggregator mojos tend to inject a lot of uncertainty and counter-intuitiveness into a build.

Aggregator mojos are implicitly designed to run from the command line (or, directly at the user's request), since to bind them to the lifecycle of a particular POM means they would either violate their own summary intention (the current state of things in 2.0.x), or they will execute (or not) in an unpredictable fashion when multiple of the same aggregator-mojo binding is encountered during a single build. Consider the case:

```
TOP
|
+- CHILD-A
|  |
|  +- binding for AGGREGATOR-1
|
+- CHILD-B
|  |
|  +- binding for AGGREGATOR-1
```

NOTE: Neither child project depends on the other.

If either CHILD-A or CHILD-B is built in isolation, its corresponding aggregator binding should run as part of that build. However, if the build is run from the project TOP, which binding of AGGREGATOR-1 should be executed? In Maven 2.0.x, both bindings are executed, thus violating the at-most-once execution semantics of the @aggregator annotation. To make matters worse, if the project TOP specified a binding for AGGREGATOR-1 and the build were run from the TOP level using Maven 2.0.x, AGGREGATOR-1 would be executed once for each project: TOP, CHILD-A, and CHILD-B...unless it specified the annotation '@inheritByDefault false' (which will simply prevent that plugin from being inherited to child POMs).

Since the only truly safe way to run an aggregator mojo is from the command line, proper use of this feature serves to destroy the controlled vocabulary that helps to make Maven a compellingly portable tool. It also raises serious concerns about how to configure the aggregator mojo in a portable way, with the only clear way being an entry in the pluginManagement section of the top-level POM. Since the definition of "top-level POM" can vary according to whether you're building a project, product containing the project, distribution containing both, etc, this still leaves open the possibility of conflicting aggregator configurations within a single build.

Proposed changes for 2.1

Deprecate @aggregator and further develop child-processing and child-summary support from parent POMs

Introduce a new, special phase: summarize

This might be implemented as a special lifecycle phase. Pre-child processing can take place in the normal parent phase bindings, since parents will be forced to the head of the line, ahead of children (see The Parent Problem, above).

Implications

One serious implication of this change is that it would require Maven to track parent-child relationships within the reactor. This is critical to enable Maven to wait until the last descendant project of a particular POM has been built, before calling the new special summary phase.

Criteria for the execution of this new phase would be the same as other phases. That is, it would require that all preceding steps (phases/mojos) succeed. Failure ahead of this phase - if not configured using something like --fail-never - would prevent the summary phase from executing.

${subProjects} expression

This expression would return all descendant projects of the current project being built, and would have to interact with the @requiresDependencyResolution annotation, as described below in '${reactorProjects} and Dependency Resolution'. It would provide an easy way for summary mojos to grab the list of projects they are supposed to summarize, complete with any dependencies they require.

Backward-compat considerations

Legacy @aggregator mojos should be supported, at least for 2.1. However, their use should prompt a deprecation warning to the user.

Build Introspection

This is the ability to query the build for the set of changes it will make to the build state. It's useful for things like IDE tooling, eclipse:eclipse, and clean:clean, since it would allow the user to discover which additional resources it should pay attention to. In the case of IDE tooling, it would allow for addition of new source folder definitions corresponding to generated-source target directories and so forth. In the case of clean:clean, it would allow the plugin to determine which directories (outside of target/, that is) it should clear when it executes.

Currently, modification of build state amounts to one of a host of API calls on such objects as the current MavenProject instance, or the MavenProjectHelper component instance. Unfortunately, only static bytecode analysis or actual execution of the plugin can determine when these changes occur; there is no way to ask the mojo to report on the changes it would make, if it were to execute. In the case of bytecode analysis, this is problematic because the actual call could be conditional based on some context information. Obviously, direct execution also carries heavy burdens, since it requires the tooling or plugins in need of this information to actually run that build...which in turn means that the build must succeed in order to extract the resulting build-state changes.

Proposed changes for 2.1

New Mojo super-classes to provide support for common types of mojos

- code generation
- resource generation
- artifact creation

Pros

- querying for build-state changes becomes a trivial matter of casting to the right mojo super-class, and calling the methods that provide the build-state changes like getCompileSourceRootAdditions().
- Instantiating, configuring, and querying the mojos in a build is relatively lightweight, compared to running a full build.
- Each super-class could implement a large part of the boilerplate code needed by these different types of plugins.

Cons

- provisioning for mojos that generate both code and resources becomes complicated
- we'll face the continual deficiency problem like we did with lifecycle phases, where we never have exactly the right combination that users ask for

- remember, this is baked into maven's core, since maven has to know how to query the mojos for information.

Backward-compat considerations

- old-style mojos will not be queryable

Forbid modification of build state without appropriate state-change annotations/metadata

This would be something like:

- @exportResource dir="target/generated-resources/plexus"
- @exportSourceRoot dir="target/generated-sources/modello"
- @exportAttachedArtifact classifier="src" type="tar.gz" file="target/foo-1.0-src.tar.gz"

Pros

- Makes state-change queries extremely lightweight; just a matter of retrieving the mojo component descriptors from the plugin artifact. This is something we already do to plan the build in 2.1.
- Could be implemented as a part of the shutdown/dispose component-lifecycle phase for Mojos, to query for these sorts of @exportXXX annotations.
- @exportXXX could be written to be extensible, for instance registering whether a mojo ran or skipped its execution because everything is up-to-date
- Boilerplate code for common types of mojos is still problematic, but could be solved using maven shared APIs, as we're starting to do now.

Cons

- Harder to capture conditional logic that may prevent the export of certain resources, source-roots, or attached artifacts, such as when the assembly plugin has configuration of attach=false.
- This means splitting all build-state into two interfaces, one MutableXXX and one XXX (immutable). MutableXXX would be passed throughout the system, but NOT to mojos

Backward-compat considerations

- we would need some sort of marker for mojos that are holdovers from 2.0.x's way of doing things, so we don't restrict their access to read-only build-state. This limits our ability to use the new features with a high degree of confidence, but preserves backward compatibility.

${reactorProjects} and Dependency Resolution

Non-aggregator and aggregator mojos alike have access to the list of projects in the current reactor, either using ${reactorProjects}, or using ${session}.getSortedProjects(). However, because project dependencies are resolved just-in-time, ahead of the mojo's execution, and according to the mojo's use of the @requiresDependencyResolution annotation, the full list of projects in the session (or the result of the ${reactorProjects} expression) WILL NOT have their dependencies resolved to the scope specified.

Proposed changes for 2.1

Add projects="sub" and projects="all" as qualifiers to the @requiresDependencyResolution annotation

This doesn't represent anything fundamentally different in terms of artifact resolution. What's new is the scope of Maven's resolution efforts. The unqualified scope specification will result in the dependencies of the current project being resolved to that scope. When qualified with 'projects="all"', the same thing happens for all projects in the

reactor. Finally, when qualified with 'projects="sub"', dependency resolution for that scope happens for the current project AND its descendant projects. This last qualifier is critical when taken with the changes to accommodate summary mojos (See the discussion on Aggregation, above). In many cases where summary mojos run, they will need information about ONLY those projects that are self-or-descendant for the current project. This helps the summary mojo escape the effects of running the build from an arbitrarily super-level of a large nested build structure.

Backward-compat considerations

The dependency-resolution features of Maven 2.0.x are fairly counter-intuitive when coupled with ${reactorProjects} or ${session}.getSortedProjects(), since it only resolves dependencies for the current project. Since this behavior is still intact by default, it should produce no compatibility problems.

The only uncertainty happens when considering whether QDox allows a value that's unattached to an attribute in mixed mode with other values that DO have attributes. For instance:

```
/**
 * @goal test
 * @requiresDependencyResolution test
projects="all"
 */
```

If this doesn't work, it's no big problem to support two forms of the requiresDependencyResolution annotation. Of course, the old specification would be supported:

```
/**
 * @goal test
 * @requiresDependencyResolution test
 */
```

In addition, a new format would be introduced:

```
/**
 * @goal test
 * @requiresDependencyResolution
scope="test" projects="all"
 */
```