Deterministic Lifecycle Planning

Since the release of Maven 2.0, several persistent issues related to the lifecycle and plugins have surfaced that seem to cause subtle problems in the build. Modifications to the lifecycle processing logic in the trunk code (what will be 2.1 someday) of Maven seek to lay the groundwork for addressing these issues, and others.

Some Common, Recurrent Problems

For instance, in cases where multiple sources specify mojo bindings for a particular lifecycle phase (packaging, POM ancestry, profiles, and current POM are all possible contributors), predicting the ordering of mojo executions within this phase can be quite difficult. Controlling this ordering can sometimes prove impossible, and force users to assign non-intuitive lifecycle phases to their mojo bindings, to ensure proper ordering.

Another common problem is that a build may sometimes fail after many successful prior builds, simply because a plugin bound to a later lifecycle (which is not invoked often) doesn't exist, or is configured incorrectly. This can waste quite a lot of time on builds that have no chance of success, especially when combined with plugins that are not sensitive to the up-to-date status of their contribution to that build, and reprocess sources regardless of what the last build did. It can also cause problems with processes that can be very hard to recover from a partial state, such as releasing a project (eg. the release plugin which changes files, commits and tags them, and then tries to build for deployment).

Still other problems include:

- lack of feedback to the user about what the build will do, before it does it (even figuring out exactly what it did can be challenging, considering the sheer volume of output from some plugins)
- some plugins execute in every build, regardless of the packaging type of the project being built. When these are bound to a parent POM, they can cause problems in building that parent POM, since it doesn't have associated source files.
- some plugins may need to be omitted in rare cases, but ideally would be configured in the parent POM for the 95% of child modules that do need it. Here, it would be simpler to allow the child POM to direct Maven to omit that plugin execution for its own build.
- In some cases (admittedly theoretical, as of now), it could be advantageous to allow one plugin to inject several others into the build at different phases of the lifecycle, as a way of standardizing and versioning a particular multi-plugin configuration to be used in concert within many builds. Just-in-time lifecycle discovery and configuration (the old style in 2.0.x) makes almost no provision for this. There is certainly no way of retrieving the list of mojo executions to be processed, which makes it much harder to imagine how the build could be manipulated by such a plugin.

Maven's Lifecycle-Processing Design, Revisited

The revisions to Maven's lifecycle-processing logic over the past year center on deterministic build planning, and providing more options to the lifecycle specification to allow for mojo configuration. The goal was to allow users and developers to know the full build lifecycle before the build started, or to report on this lifecycle without executing a single mojo of the main build. Another key criteria was to make the resulting build plan data structure easily navigable, without requiring developers to implement a complex lifecycle-walking algorithm in order to understand what would execute, when.

Other features of the lifecycle redesign include a first-class model for mojos and lifecycles, which makes it straightforward to query any node in the build plan list (or the more complex lifecycle definition instance, from which the build plan is constructed) using a concrete, getter/setter-style API. Also included is configuration planning, to allow users and developers to see what mojo configuration options are provided for each step in the build plan.

Deterministic Build Planning

Starting with Maven 2.1, each build will be mapped out completely before it is executed. This means that all direct mojo invocations specified on the command line, all bindings brought in via the default or specified project packaging (the <packaging/> element contents in the POM), all mojo bindings that are configured in the project's own POM or one of its ancestors (and including those specified in an active profile within one of those POMs) will be explored and added to a single data structure. Then, once all of this information about the build is compiled, Maven will use it to construct a flattened, straight-through list of ordered mojo executions to be processed...which it will then process. The aim of this is to reduce the number of plugin errors that happen late in the build, potentially after some unrecoverable mojo error has left your build in limbo between success and failure.

Along the way, Maven must cope with some obvious roadblocks standing in the way of creating this simple list of mojos. First, where normal mojos execute once per project in the build, aggregated mojos execute only once. These typically make use of the ${reactorProjects} expression, and act on all projects in the build at one time. When Maven encounters an aggregated mojo, it dices up the list of tasks given on the command line (or in the setGoals(..) method of the MavenExecutionRequest) into contiguous segments of aggregator and non-aggregator invocations, then iterates through the list of segments, executing either in once-per-project (sub-iterating through the list of projects for each task) or aggregated (simply execute the task) for each segment. This means that certain aggregator mojos will not be present in the build plan for each project, but rather will exist only in the build plan of the "root" project for the build.

Add to this complication the fact that some mojos create a forked execution, directing Maven to execute some other mojo or lifecycle phase, then processing the result in some way before resuming the main lifecycle's execution. When this happens, it effectively causes Maven's lifecycle-processing logic to recurse, and handle the new fork as a new set of tasks to execute. During the forked execution, any mojo executions that led to the current fork (we keep a Stack of mojos that led to the current fork, pushing the forking mojo onto the stack prior to fork execution, and popping it back off immediately afterward) must be omitted from the new fork to prevent infinite recursion. Further complicating things, Maven allows forked executions to use a special "lifecycle overlay" which provides its own mojo bindings and even configurations for mojo bindings. These are applicable only during the fork execution, then the existing bindings and configuration are restored. When Maven calculates all of this during the build-planning step, it accounts for lifecycle overlays by marking each fork point with a separate, nested build plan instance, so the differences can be maintained and taken into account when the build plan is rendered (more on this below). Aside from cloning the build plan and adding lifecycle overlays as necessary for forked executions, and special handling logic during the rendering step to actually calculate the impact of the fork, forked executions are pretty simple additions to the build-planning process.

After the build plan has been constructed, a flattened List of mojo-execution instances (the actual class is MojoBinding) can be rendered according to the list of tasks used to construct it. Whenever a plugin fails to load during build-plan discovery, that MojoBinding is preceded by a special, internal late-binding mojo execution. The late-binding mojo simply acts as a last-ditch effort at resolving the plugin before the build fails. This is especially useful if the plugin to be used is built as part of the current Maven execution, where the plugin may not exist in binary form before Maven is called. In addition, whenever a forked execution is encountered during this rendering, Maven traverses the nested, forked-execution build plan and renders it into a list that is added to the main mojo list. However, this forked list is bracketed by some special handling in the form of three internal mojos.

1. Before the forked-execution mojos are added, a special StartForkedExecutionMojo runs. This mojo clones the current project, pushes the original instance onto a stack of forked projects, and replaces it with the clone so forked mojos don't affect the original project instance.
2. After this is added, the forked-mojo listing is added. These are the mojos effectively referenced by the @execution annotation in the forking (fork-initiating) mojo.
3. Next, an EndForkedExecutionMojo is added to pop the original project instance back out of the forked-project stack, and set the just-completed project instance as executedProject on the original instance. This is critical to allow the mojo which initiated the fork (usually, this mojo's source code makes use of the @execute annotation) to selectively transfer accumulated state from the forked project to the original one.
4. Finally, after the forking mojo is added, a third special mojo called ClearForkedExecutionMojo is added. This

mojo simply clears out the executionProject attribute from the current project instance, to hide the fork result as out-of-scope from subsequent mojo executions.

All of this complicated logic is used to first build, then render the complex web of forked-execution nestings and direct-mojo invocations - along with the straightforward mojo bindings that make up 80% or more of the mojo bindings typically in use - into a simple list of instructions for Maven to follow. While the complex data structure has value in terms of providing a single source of information about what Maven intends to do **before it does it**, the logic required to traverse this structure is too great to externalize. Therefore, the BuildPlan class incorporates logic for rendering itself into a List of MojoBinding instances, traversing direct mojo invocations and recursing into forked executions as needed. This allows developers to interact with the BuildPlan instance, then re-render it as needed to obtain updated MojoBinding lists.

The deterministic build plan enables users to use special plugins (or, even IDE tooling) to inspect the build plan before it is executed. Gone are the problems in determining what Maven is doing because all you see is a blur of console output. Gone is the guesswork trying to figure out where, relative to other mojos in a particular phase, your new mojo binding will be executed. Reduced - but not gone, as we see with the late-binding mojo - are the problems where the build fails in a late and often-unused phase because some plugin or other doesn't exist. Yes, build planning does incur some overhead during the bootstrapping phase of the build. However, the visibility into exactly **what** Maven is doing more than makes up for a small time penalty.

Extended Lifecycle Specification

In addition to deterministic build planning, Maven 2.1 offers the ability to specify lifecycle bindings for a given packaging type **that also contain mojo configurations**. Currently, the only implementation of this new type of lifecycle-binding loader (the class is LifecycleLoader) is one that reads a classloader path, set as part of that loader's component configuration. Whatever the source, Maven now provides a concrete syntax for specifying lifecycle bindings with an accompanying parser that enforces the use of correct lifecycle and lifecycle-phase names, and renders a true Java API for inspecting and manipulating the bindings (as opposed to the current, reflectively-populated, obscure class which is organized into a Map according to the name of the lifecycle it addresses). As an additional benefit, these new binding specifications can contain configurations for the mojos they list. This allows the same mojo to serve multiple related purposes, depending on the type of project being built and the configuration options supplied by the lifecycle mapping. It also allows the same mojo to execute multiple times with different configurations, all in the same build - something Maven supports even in 2.0.x when specified explicitly within a POM, but not from a lifecycle mapping.

Easily Navigable Lifecycle/Build-Plan Models

As mentioned above, the concrete LifecycleBindings/MojoBinding model classes - along with the BuildPlan derivative used to inject direct invocations and forked executions - provide an easy way to inspect and manipulate the list of mojos that will execute during a particular Maven build. However, this is worth highlighting separately: where 2.0.x provided an immutable and obscure LifecycleMapping class collected by lifecycle-name into a java.util.Map instance, contained as a private member of the DefaultLifecycleExecutor, Maven 2.1 will present a concrete, strict syntax and class model that can be built, rendered, inspected, and manipulated via documented APIs.

Using the New Lifecycle API

TODO: Programmatic Access

NOTE: This section will change as soon as the build-plan instance is made available in the MavenSession and MavenExecutionResult classes. Once this is finished, build-plan access should be as easy as querying either using

using the getBuildPlan(MavenProject) method. However, until task-segmentation issues are resolved (issues related to creating a template instance of the build plan, then cloning it for each task-segment executed) the BuildPlanner component is the only option.

The build plan for a given project can be accessed using the BuildPlanner component (injected either as a @plexus.requirement or @component annotation, depending on whether you're writing a Plexus component or an implementation of the Mojo interface), found in the maven-core artifact. The method:

```
org.apache.maven.lifecycle.plan.BuildPlanner
.constructBuildPlan(List, MavenProject,
MavenSession): BuildPlan
```

requires a List of tasks (these are just Strings), the current MavenProject instance, and the current MavenSession instance.

TODO: Change this section to talk about the up-coming MavenSession.getBuildPlan(MavenProject) and MavenExecutionResult.getBuildPlan(MavenProject) methods instead of direct use of the BuildPlanner component, which will not return the same *instance* of BuildPlan as that being used for the actual build execution.

Immediately Accessible Build Plan Listing:

Maven's debug mode (enabled using the -X flag) will always print the build plan before the build starts, preceded with the text "Our build plan is:", just to make sifting through all the console output marginally simpler.

However, far greater gains can be realized by using the still-experimental maven-lifecycle-plugin (http://svn.apache.org/repos/asf/maven/sandbox/trunk/plugins/maven-lifecycle-plugin). This plugin's lifecycle:build-plan goal takes one required parameter: -Dtasks=first,second,third and one optional parameter: -DextendedInfo=true. The tasks parameter gives the list of command-line tasks you would like to check. When executed, the build-plan mojo will print the list of mojos it would execute to complete the tasks listed, **in order**. With the extendedInfo flag enabled, each mojo execution will be accompanied by a small XML snippet showing the configuration options Maven would use for that invocation.

Along with the build-plan mojo, the lifecycle plugin also offers a mojo called lifecycle:show-lifecycle-phases. This mojo takes one optional parameter: -Dlifecycle=[build|site|clean] (which defaults to 'build', a new synonym for the 'default' lifecycle), and renders a listing of the specified lifecycle's phases, in order out to the console. This can be a great help to Maven users who don't have a lot of experience adding custom mojo bindings, to let them know what types of activities happen, in what order, during a Maven build.

Compatibility with 2.0.x

The new build-plan functionality should be completely backward compatible with Maven 2.0.x, even down to the private Map of LifecycleMapping instances present in the DefaultLifecycleExecutor (thanks to the Maven20xCompatibilityAspect, which populates the field out of the new build-plan discovery infrastructure). While the references and configuration for legacy-style lifecycle mappings have moved to a different component since 2.0.x (they're now in the component definition for DefaultLifecycleMappingParser, not DefaultLifecycleExecutor), these mappings are still supported. Maven now prefers to use LifecycleLoader implementations for a given packaging, but will default over to looking up the legacy LifecycleMapping component for the packaging type if no matching LifecycleLoader component is found.

Future Directions

This refactor of the Maven lifecycle discovery process sets the foundation for more work that will allow Maven users to tailor their build much more precisely. Once the BuildPlan is available via the MavenSession and MavenExecutionRequest classes, it only requires a small step farther to allow a custom Maven build extension to modify the build plan and turn off, reconfigure, or even replace certain mojo executions. It's even conceivable that a plugin or - better still - a modification to the POM's build section will allow fine-tuning of mojo ordering and execution on a project-by-project basis. This would provide the ultimate flexibility for large project sets, where plugins could be specified in a parent hierarchy for 95% of child modules to use, and simply turned off in the rest.

Aside from fine-tuning the lifecycle, deterministic build planning may open the door for modifications to the Maven plugin API that would allow a mojo to declare what modifications to build state it will make. IDE developers could then simply ask Maven for the build plan for a given project, then scan through and figure out which generated-source and -resource directories it should add to the workspace.

Conclusion

Maven has always had as one of its stated goals a high degree of determinism. The whole point of Maven is to allow users to provide a static description of a project, from which the tooling can draw its configuration and build the project. Discovery of the build process for a project in a just-in-time manner is counter-productive to this goal, since it effectively hides from the user the list and specific ordering of tasks that will be performed when the project is built. This hinders build-debugging efforts to a great degree, especially when combined with noisy plugins that create a flurry of console output as they execute. But it also turns Maven into a black-box system, where the user enters some project information, starts maven, **details ommitted here**, and out comes the project binary, ready for use. Resolving the details about the highlighted section requires a lot of effort on the part of the user, not to mention extraordinary measures like capturing console output and then sifting through it by hand. With the introduction of build planning, Maven 2.1 addresses the issues of transparency caused by JIT lifecycle discovery, and exposes the full plan of what it intends to execute, including all plugins and their associated configurations for each step, to the user and developer. In fact, the build plan is even available in Maven's debug output (using -X).

In addition to transparency, the new build-planning feature provides a solid foundation on which to build the capability for advanced build tuning and the extraction of build state information injected during the execution of certain mojos. Exposing the build plan to extension and plugin developers, along with consumers of Maven's embedder, could open the door to far more advanced use cases and tighter integration with third-party tooling than was ever possible using Maven 2.0.x.

Relevant Discussions and Issues

Wiki Articles

1. Lifecycle and Plugin Handling: This is the parent page for lifecycle-related design proposals (or is meant to be). Much of the work in described in this page is foundational to improvements proposed here.
2. Suppression, Ordering, and Replacement of Plugins and Mojos Bindings: Specific proposal designed to address many of the improvements mentioned in the above reference. Again, the work described on this page is foundational to the improvements outlined here.

JIRA Issues

1. MNG-2442 : Querying lifecycle composition is a direct effect of the work outlined here.
2. MNG-474 : Not entirely sure what this issue intends, but the build-planning subsystem does flatten forked execution into a single list of mojos to be executed. At the very least, this flattening should make analysis and elimination of needless, duplicate mojos easier.