API Initialization

Log4j 2.x has a fairly complex API initialization process. Log4j 3.x intends to simplify and unify this into a proper SPI.

Overview of 2.23.0 API Initialization

In Log4j 2.23.0 and later the Log4j API subsystems are initialized roughly in this order:

1. `StatusLogger` and `AbstractLogger` are initialized with some dependent classes like `MarkerManager` . These classes of course can not call `StatusLogger.getLogger()` in their static initialization block nor use `PropertiesUtil` ,
2. The classloading subsystem is started: `LoaderUtil` , `ServiceLoaderUtil` , `OsgiServiceLocator` and `StackLocatorUtil` . These can use `StatusLogger` , but can not use `PropertiesUtil` in their static initialization block,
3. `PropertiesUtil` is initialized,
4. The other services are initialized and the system should pretty much be stable.

Overview of older 2.x API Initialization

Typical usage of Log4j begins with use of either `LogManager` or `ThreadContext.` The expectation is that Log4j will be initialized and ready to use as soon as user code calls `LogManager::getLogger` or similar. This introduces a hierarchy of dependent classes for initialization.

LogManager

Initializing `LogManager` begins with its static fields. Besides using `Strings.EMPTY` which itself ends up initializing `SystemPropertiesPropertySource,` a `Logger` field is initialized with the result of `StatusLogger::getLogger.` Entering `StatusLogger` initializes `AbstractLogger` which initializes `MarkerManager.` Both `StatusLogger` and `AbstractLogger` use `PropertiesUtil` which initialize instances for the `log4j2.StatusLogger.properties` and `log4j2.component.properties` files, the latter which corresponds to the general `PropertiesUtil::getProperties` instance. Loading any `PropertiesUtil` instance ultimately uses `ServiceLoaderUtil` to load `PropertySource` services to back the instance. `AbstractLogger` also uses `LoaderUtil` which itself avoids initializing `PropertiesUtil` until first usage in methods (and is required to use `LowLevelLogUtil` for logging similar to `PropertiesUtil` and its `PropertySource` implementations along with any other classes initialized at this point). The rest of the classes initialized due to `StatusLogger` are either interfaces or simple stateless classes and are not necessary to analyze.

Next, the static initialization block of `LogManager` is run to set its initial `LoggerContextFactory` static field. It uses `PropertiesUtil::getProperties` and `LoaderUtil` to load a class specified by the `log4j2.loggerContextFactory` property if provided. Otherwise, it checks `ProviderUtil::hasProviders` which leads into the next startup dependency chain. `ProviderUtil` contains a lock in a static field used for guarding the lazy initialization of installed providers. This allows the `Activator` support class to be used in OSGi environments to lock the startup until a Log4j provider is installed from another bundle (otherwise, eager initialization causes OSGi to only find the fallback `SimpleLoggerContextFactory` implementation as `log4j-core` is not installed and started until after `log4j-api` is installed). If no providers have been installed already, lazy initialization takes place which acquires the startup lock, loads all available `Provider` services via `ServiceLoaderUtil,` then loads any available `META-INF/log4j-provider.properties` specified Log4j providers (the original service provider definition format used by Log4j 2.x), then finally releases the startup lock.

`LogManager` then tries loading the `LoggerContextFactory` classes specified by the discovered Log4j providers. If only one provider was found, its `LoggerContextFactory` is used. If more than one provider is found, then the one with the highest priority value is used (along with a warning message with the list of discovered providers and priorities). Otherwise, `SimpleLoggerContextFactory` is used as a fallback. Once an initial factory is selected, `LogManagerStatus` sets its `initialized` flag to `true.`

Typical invocations of `LogManager` include `LogManager::getLogger` of some form. The zero-argument version of `getLogger` uses `StackLocatorUtil` to find the caller class whose initialization uses `StackLocator` which has its own static initialization invoking `LoaderUtil::loadClass` which has already been initialized from earlier. Any of the `getLogger` methods call out to one of the `getContext` methods to get the appropriate `LoggerContext` to use for getting `Logger` instances. These typically get the caller class and look up a `LoggerContext` using its `ClassLoader` which delegates to `LoggerContextFactory::getContext.` The details of this are specific to each Log4j provider. The default `log4j-core` configuration uses a `ClassLoaderContextSelector` class to keep track of one or more Log4j configurations isolated by `ClassLoader` such as in a Servlet environment.

ThreadContext

Initializing `ThreadContext` begins with the `ThreadContext::init` method invoked by its static initializer block. This initializes `ThreadContextMapFactory` which initializes `PropertiesUtil` (which follows the same dependency chain from `LogManager` with `ServiceLoaderUtil` and `PropertySource` service instances) to set initial static field values (which will be re-initialized each time `ThreadContextMapFactory::init` is invoked). Then it calls `ThreadContextMapFactory::init` which invokes the same static `init` method on `CopyOnWriteSortedArrayThreadContextMap,` `GarbageFreeSortedArrayThreadContextMap,` and `DefaultThreadContextMap` respectively before overwriting its initial static fields with whatever `PropertiesUtil` has now (typically the same value as not much time has passed). Each of those `init` methods sets static fields on their respective classes to values obtained from `PropertiesUtil.`

`ThreadContext::init` continues with a lookup to `PropertiesUtil` to find out if the thread context map, stack, or both have been disabled. Combined with the settings initialized by `ThreadContextMapFactory,` new instances of both `ThreadContextMap` and `ThreadContextStack` are assigned to static fields in `ThreadContext` which are used by the remaining static methods in `ThreadContext.`

Problem

There is a lot of static state in various classes in `log4j-api` which should not be doing so. Static initialization blocks in classes introduce hard to understand dependencies between classes during class loading which has already necessitated in the introduction of the `LowLevelLogUtil` class for classes initially loaded by `StatusLogger` to themselves have an ability to log errors. Eager static initialization of the logging system makes it extremely difficult to customize the behavior of Log4j startup and has led to various workarounds. This also makes it non-trivial to support concurrent execution of unit tests which leads to a long, serialized build configuration.

In Log4j 3.x, we should clean up this API initialization story to avoid static initialization (unless used for performance reasons such as cached lookup tables of pre-computed values), centralize the entry points between `LogManager` and `ThreadContext` along with supporting classes like `ProviderUtil` and `Activator` in OSGi into a slightly more sophisticated `Provider` SPI, and update our test infrastructure to reuse the same SPI.