MergedAnnotations API Internals

This page attempts to document some of the design decisions taken for the internals of the `MergedAnnotations` API.

See also: Spring Annotation Programming Model

High-level Overview

The `MergedAnnotations` interface is designed to work with both Java reflection types and ASM bytecode reading. In the future, it's also possible that additional sources may be supported.

The boundary of ASM support extends only as far as the annotation *usage*. The actual `Annotation` class must be available at runtime and is always inspected using reflection. In other words, the annotation attributes can be read using ASM, but the actual annotation class is not.

Package private classes

As much as possible the user-facing API for `MergedAnnotations` is limited to just interfaces. The implementations are intentionally kept package-private and are not available to the user. Access is always via static methods on the interfaces.

Type caching

Calculating the way that annotation attributes are merged can be expensive, so as much as possible cache friendly structures are used. The two main caches used are in `AttributeMethods` and `AnnotationTypeMappings`.

AttributeMethods

The `AttributeMethods` class provides a consistent view of the attribute methods of an `Annotation` class. It primarily provides caching for the sorted `Method` instances.

AnnotationTypeMappings

The `AnnotationTypeMappings` class is responsible for crawling the meta-annotations on an `Annotation` and pre-computing as much information as possible. From `AnnotationTypeMappings` you can get an `AnnotationTypeMapping` and very quickly map attributes back to the root annotation.

It's important to understand that `AnnotationTypeMapping` represents a view of an annotation within the *context* of a root annotation.

For example, an instance of `AnnotationTypeMappings` for `@GetMapping` provides access to an `AnnotationTypeMapping` for `@RequestMapping` (i.e., the root type). However, this is different than the `AnnotationTypeMapping` for `@RequestMapping` accessed via `@PostMapping` or `@PutMapping`. You need to consider the entire chain of annotations back to the root type.

Alias Mappings

The internal `AnnotationTypeMapping.aliasMappings` array tracks how attributes are mapped via `@AliasFor` to the root annotation. Given an `AnnotationTypeMapping` it's possible to very quickly tell which of the root annotation attributes actually provides its value.

Remember that an `AnnotationTypeMapping` only knows information about the `Annotation` type. It doesn't directly know what attribute values will be used when it is actually declared. In other words, `AnnotationTypeMapping` tracks static metadata about an annotation type.

For example, suppose we have the following:

```
@interface Bar {

    String name() default "";

}

@Bar
@interface Foo {

    @AliasFor(annotation=Bar.class, attribute="name")
    String barName() default "";

}
```

The `AnnotationTypeMapping` for `Bar` (in the context of `Foo`) knows that the "name" attribute is aliased to "barName" on the root annotation. When we actually declare an annotation, for example `@Foo(barName="Spring")`, we can very quickly tell that calling `name()` on the `@Bar` meta-annotation (index `0`) just goes to `barName` on the root annotation (index `0`).

This mapping logic also supports multi-level meta-annotation hierarchies. For example, a `@ComposedFoo` annotation would also maintain mappings.

Convention-based Mappings

For backward compatibility, convention-based mappings are also tracked. These are implicit mappings used when the `@AliasFor` annotation is not present. They work for all attributes, except `value`.

For example, given the following, there is an implicit, convention based mapping from `Foo.name` to `Bar.name`:

```
@interface Bar {

    String name() default "";

}

@Bar
@Interface Foo {

    String name() default "";

}
```

The internal `AnnotationTypeMapping.conventionMappings` array works in exactly the same way as `AnnotationTypeMapping.aliasMappings` but for convention based mappings.

Annotation Value Mappings

Some meta-annotation values are actually resolvable from the type information. For example, the `@RequestMapping.method()` attribute on a `@PostMapping` will always return `RequestMethod.POST`.

For these elements the `AnnotationTypeMapping.annotationValueMappings` and `AnnotationTypeMapping.annotationValueSource` arrays are used.

In the example above, the `AnnotationTypeMapping` for `@RequestMapping` would point to the `@PostMapping` source and the `method` attribute. The value would be directly read from the declared meta-annotation:

```
@RequestMapping(method = RequestMethod.POST)
@interface PostMapping {

   // ...

}
```

Mirror Sets

The `@AliasFor` annotation can be used to declare that multiple attributes represent the same underlying value. In its simplest form, it looks like this:

```
@interface Foo {

    @AliasFor("value")
    String name() default "";

    @AliasFor("name")
    String value() default "";

}
```

The `AnnotationTypeMapping` class refers to these as "mirror" attributes. Mirror attributes can also be declared when two or more attributes declare an `@AliasFor` on the same meta-annotation attribute.

Although it's possible to detect mirror attributes from the type mapping, we can't actually tell which one to use until we have attribute values. We also can't fully verify that the user has not made an error.

For example, `@Foo("spring")` and `@Foo(name = "spring")` are valid, but `@Foo(value = "framework", name = "spring")` is not.

The `getMirrorSets().resolve(...)` method is used to work out which actual mirror attribute has been used on the declared annotation. It returns an array that maps to a real attribute, or it throws an exception if the user has misconfigured something.

Give the examples above, calling `resolve` for `@Foo("spring")` would return `[1,1]`. For `@Foo(name = "spring")` it would return `[0,0]`.

The index into the array is the attribute being requested; the value is the attribute to use.

For `@Foo("spring")` calling `Foo.name()` is a lookup at index `0`, returning `1` and `Foo.value()` is a lookup at index `1`, returning `1`. In other words, the `value` attribute is always the source of truth in this case.

As with the other elements, the design goal is to provide as much pre-computation as possible. The actual resolution process once we have declared attribute values should be quick.

Interface implementations

The final piece of the puzzle is to provide actual implementations of the `MergedAnnotations` and `MergedAnnotation` interfaces for the user to work with. The main classes here are `TypeMappedAnnotations` and `TypeMappedAnnotation`.

`TypeMappedAnnotations`

The `TypeMappedAnnotations` class is responsible for exposing declared annotations from an `AnnotatedElement` in a uniform way. The class has a `scan` method which is used when operating on a single annotation, and `stream` methods for working with all annotations.

Both annotations and meta-annotations need to be exposed. Declared annotations are discovered using the `AnnotationScanner`, then `AnnotationTypeMappings` are used to add meta-annotations.

There are some fairly complicated ordering rules that are contained in the inner `AggregatesSpliterator` class.

`AnnotationScanner`

The `AnnotationsScanner` class is used to find declared annotations from an `AnnotatedElement`. It supports various search strategies and deals with the complexities of searching for Inherited methods.

There are a few unusual aspects to the way the scanner has been implemented that can catch you out. One specifically worth mentioning is that the array passed to the `AnnotationsProcessor` callback can contain `null` elements. This was done as a performance optimization so that we don't need to copy the array simply to remove `null` elements.

Another interesting aspect of the `Scanner` is that it the processor can trigger an early exit. This is another performance tweak that prevents the need to scan a full class hierarchy if a result has already been found.

Since the `AnnotationsProcessor` is a package-private class, neither of these quirks are exposed to the end user.

`TypeMappedAnnotation`

The `TypeMappedAnnotation` class provides a `MergedAnnotation` implementation by combining an `AnnotationTypeMapping` with an actual annotation source. The source can be an actual declared Java annotation or a value read using ASM. The `valueExtractor` function provides the level of indirection needed to support both. For a real annotation, `ReflectionUtils::invokeMethod` can be used.

Most of the `TypeMappedAnnotation` implementation is fairly straightforward, and there's nothing too unusual with the class. One aspect that is worth noting is the set of `adapt...` methods. These are used when an extracted value needs to be adapted to a different type. This might be because a `String` value read from ASM now needs to be accessed as a class. Or it might be because a nested `Annotation` needs to be read (for example `@ComponentScan.includeFilters`)

`TypeMappedAnnotations.NONE`

It's quite common for `MergedAnnotations` to be called on a class that has no annotations. For these situations, we return a single shared instance of `TypeMappedAnnotations`. This saves us from needing to create a new empty instance each time and helps to reduce garbage collection.

`MissingMergedAnnotation`

The `MergedAnnotations` API has been designed not to return `null` whenever possible. Instead `MergedAnnotation.missing()` is returned when an annotation is not present.

The package-private `MissingMergedAnnotation` class provides the implementation for this "null object" pattern.