Implemente a descoberta automática em seu código Java com anotações

Se você precisar iniciar e parar várias partes do código em um sistema da Web baseado em Java, poderá implementar uma solução que use um único ServletContentListener para ouvir os eventos de início e parada do contêiner. Esse Listener pode usar o java.util.ServiceLoader para localizar classes registradas que desejam ouvir esses eventos.

Isso funcionaria bem o suficiente, mas e quanto a adicionar um processador de anotação em tempo de compilação? Quando você anota um método estático com @Lifecycle (LifecycleEvent.STARTUP) , ele será chamado na inicialização (e vice-versa para SHUTDOWN ). O processador gera classes e registra-as para o ServiceLoader. Você poderia aproveitar esse mesmo mecanismo para qualquer modelo de barramento de eventos: os ouvintes são registrados em tempo de compilação com anotações e depois chamados automaticamente pelo barramento quando o evento é acionado. Em essência, você pode usar anotações para descobrir automaticamente o código em tempo de execução com um ServiceLoader.

Em ação, o conceito é o seguinte:

1. Você anota métodos com @EventListener (que pode conter algumas meta-informações).
2. Um processador de anotação grava um EventDispatcher para cada método @EventListener , incluindo os filtros exigidos pela meta-informação na anotação.
3. O barramento de eventos usa java.util.ServiceLoader para localizar implementações EventDispatcher .
4. Quando EventBus.dispatch é chamado, todos os métodos interessados ​​que foram anotados com @EventListener são chamados.

artigo demonstra esse conceito percorrendo as etapas necessárias para criar um barramento de eventos que invoca métodos de ouvinte anotados sem qualquer registro manual. A discussão começa com o EventBus , passa para o processador de anotações e termina com um exemplo de uso.

### Organizando Seu Código

O download do código para o exemplo contém dois projetos IDE separados:

* EventBus - contém o barramento de eventos e o processador de anotações
* EventBusExample - contém um exemplo de uso do barramento de eventos

Ao lidar com processadores de anotação, você geralmente deve desativar a opção "Compilar ao salvar" (ou equivalente) no seu IDE. Essas opções têm o péssimo hábito de excluir as classes geradas pelo processador de anotações, deixando você coçando a cabeça.

As seções a seguir explicarão como o código nesses projetos funciona, com trechos fornecidos para ilustração.

### A anotação e o evento

A primeira coisa que você precisa é de uma anotação @EventListener para marcar métodos que irão escutar eventos. Aqui está um exemplo da anotação EventListener, que tem permissão para anotar apenas os métodos. Ele será descartado depois que o código for compilado, porque todo o processamento é feito contra o código-fonte.

@Target(ElementType.METHOD) @Retention(RetentionPolicy.SOURCE) public @interface EventListener {

String name() default ".\*";

Class<?> source() default Object.class;

}

Como esse exemplo é um modelo de barramento de eventos, é melhor que os métodos de ouvinte recebam apenas os eventos nos quais estão interessados. Para ajudar a aplicar essa regra, uma classe BusEventObject contém um nome que você pode filtrar (com base no nome no @ Anotação EventListener ). Para facilitar a filtragem de eventos, a classe EventObject normal possui um campo de nome adicionado. O BusEventObject também serve como um marcador para eventos que podem ser despachados através do EventBus.

public abstract class BusEventObject extends EventObject {

private final String name;

public BusEventObject( final Object source, final String name){

super(source);

if(name == null || name.isEmpty()) {

throw new IllegalArgumentException("empty or null name");

}

this.name = name;

}

public String getName(){

return name;

}

}

### O processador de anotação

Para começar a escrever processadores de anotações, primeiro você deve se familiarizar com os grupos de pacotes javax.annotation.processing e javax.lang.model . Geralmente, você pula a implementação da interface do processador diretamente e estende a classe abstrata de conveniência javax.annotation.processing.AbstractProcessor . O AbstractProcessorprecisa de algumas informações sobre sua implementação, fornecidas por você usando anotações. O EventListenerAnnotationProcessor na declaração de código de exemplo se parece com isto:

@SupportedSourceVersion(SourceVersion.RELEASE\_5) @SupportedAnnotationTypes(EventListenerAnnotationProcessor.ANNOTATION\_TYPE)

public class EventListenerAnnotationProcessor extends AbstractProcessor {

O @SupportedSourceVersion informa ao AbstractProcessor que você deseja apenas arquivos de origem escritos para o Java 5 ou superior; enquanto o @SupportedAnnotationTypes informa quais anotações você está interessado ( EventListener.class.getName () não funcionará como um valor de anotação, porque o compilador não pode avaliar tais expressões).

public static final String ANNOTATION\_TYPE = "eventbus.EventListener";

Para simplificar, o processador de anotação é dividido em duas classes principais ( EventListenerAnnotationProcessor e EventDispatcherGenerator ) e uma classe de utilitário genérica ( ServiceRegistration ). Para que a ferramenta de anotação do compilador execute o EventListenerAnnotationProcessor , você precisa registrá-la com um arquivo de serviços (o compilador também usa o ServiceLoader ).

eventbus.processor.EventListenerAnnotationProcessor

O arquivo de registro de Serviços ( META-INF / services / javax.annotation.processing.Processor ) é nomeado de acordo com a interface que oServiceLoader deve localizar implementações de.

A primeira ação do método EventListenerAnnotationProcessor.process () é encontrar todos os métodos @EventListener sendo compilados nesta rodada.

final Elements elements = processingEnv.getElementUtils();

final TypeElement annotation = elements.getTypeElement(ANNOTATION\_TYPE);

final Set<? extends Element> methods = roundEnv.getElementsAnnotatedWith(annotation);

Objetos de elemento são muito parecidos com objetos de reflexão para o compilador e processador de anotação. TypeElement é como classe , enquanto ExecutableElement é semelhante ao construtorou método . O RoundEnvironment (que representa esta rodada de processamento de anotações) terá retornado os elementos (neste caso, métodos) que são anotados por uma anotação @EventListener .

### O EventDispatcherGenerator

O EventDispatcherGenerator é um gerador de código muito simples. Você pode preferir usar um mecanismo de modelo (como FreeMarker ou Velocity) para gerar seu código-fonte, mas o código para este exemplo é gravado com um PrintWriter . Cada ExecutableElement representando um método anotado @EventListener é passado para o EventDispatcherGenerator.generate , que gravará o código-fonte para um EventDispatcher .

for(final Element m : methods) { // ensure that the element is a

method if(m.getKind() == ElementKind.METHOD) {

final ExecutableElement method = (ExecutableElement)m;

results.add(generator.generate(method));

}

}

O EventDispatcherGenerator precisará gerar um arquivo de origem Java para cada um desses métodos. Um processador de anotação usa o objeto Filer fornecido pelo ProcessingEnvironment para criar arquivos de origem nos quais ele pode gravar código.

final JavaFileObject file = processingEnvironment.getFiler().createSourceFile( className, // ie: com.mydomain.example.OnMessageDispatcher method); // ie: com.mydomain.example.Listener.onMessage(MessageEvent)

O ExecutableElement fornecido para o Filer neste exemplo representa o método anotado (o segundo argumento em createSourceFile ). Isso diz ao ambiente que você está gerando código-fonte relacionado a esse método, o que não é necessário, mas útil, no entanto. O código usa o JavaFileObject para abrir um gravador e começar a gerar o código-fonte.

final Writer writer = file.openWriter();

final PrintWriter pw = new PrintWriter(writer);

pw.append("package ").append(packageName).println(';');

Os valores especificados na anotação @EventListener para o método são usados ​​para gerar uma instrução if que filtrará BusEventObjects antes de chamar o método anotado. O EventDispatcherGenerator grava uma instrução if no código-fonte que decidirá se deve ou não despachar o objeto de evento para o método @EventListener .

public final class EventBus {

private static final EventDispatcher[] DISPATCHERS;

static { final ServiceLoader<EventDispatcher> loader = ServiceLoader.load(EventDispatcher.class);

final List<EventDispatcher> list = new ArrayList<EventDispatcher>();

for(final EventDispatcher dispatcher : loader) {

list.add(dispatcher);

}

DISPATCHERS = list.toArray(new EventDispatcher[list.size()]);

} private EventBus() { } public static void dispatch(final BusEventObject object) { if(object == null) { throw new IllegalArgumentException("null event object"); } for(final EventDispatcher dispatcher : DISPATCHERS) { dispatcher.dispatch(object); } } public static interface EventDispatcher { void dispatch(BusEventObject object); } }

### Registrando o EventDispatcher

A tarefa final para gerar os EventDispatchers é listar todos eles em um arquivo de serviços, para que o ServiceLoader possa encontrá-los quando o EventBus for inicializado. Existem alguns truques neste processo. O processador de anotações fornece uma lista apenas dos métodos incluídos na compilação atual. Como os desenvolvedores geralmente não compilam toda a sua base de código de uma só vez, o código do processador precisará controlar os dois métodos que já foram compilados e os que estão sendo compilados no momento. Esse é o trabalho da classe ServiceRegistration .

Primeiro, você indica ao ServiceRegistration para ler qualquer arquivo de serviço existente para EventDispatchers no caminho de origem ou no diretório de saída da classe. Em seguida, adicione as classes EventDispatcher recém-compiladas e, em seguida, grave o novo arquivo de serviço no diretório de saída da classe.

final AnnotationHelper annotation = new AnnotationHelper( method, EventListenerAnnotationProcessor.ANNOTATION\_TYPE, environment.getElementUtils()); final String nameFilter = (String)annotation.getValue("name"); final TypeElement sourceFilter = (TypeElement)environment.getTypeUtils(). asElement((TypeMirror)annotation.getValue("source")); pw.println("\tpublic void dispatch(eventbus.BusEventObject event) {"); pw.print("\t\tif(event instanceof "); pw.println(eventType.getQualifiedName()); pw.println("\t\t\t\t&& nameFilter.matcher(event.getName()).matches()"); pw.append("\t\t\t\t&& event.getSource() instanceof "). append(sourceFilter.getQualifiedName()).println(") {");

### Juntando Tudo

O projeto EventBus resulta em um único arquivo JAR com o tempo de compilação e o código de tempo de execução (embora você possa dividi-lo em dois JARs). Agora você precisa escrever uma subclasse de BusEventObject que pode ser despachada para os ouvintes através do EventBus . Você também precisará de um método @EventListener para receber instâncias de sua nova classe de eventos. Finalmente: você precisará de uma classe para despachar os eventos de (uma fonte).

Para verificar se o método @EventListener possui um EventDispatcher gerado, você precisará certificar-se de que o compilador saiba que deve executar o EventListenerAnnotationProcessor .Esse processo varia de um IDE para outro, mas a simples verificação de que o JAR está no caminho de classe ou nas bibliotecas do projeto é suficiente. Em alguns IDEs (como o Eclipse), você precisa registrar o processador de anotações manualmente. Para este exemplo, uma classe MessageEventserá despachada através do barramento de eventos:

public class MessageEvent extends BusEventObject { private final String message; // constructor, etc. public String getMessage() { return message; } }

Você precisará de um @EventListener para pegar os objetos MessageEvent e fazer algo com eles.Lembre-se de que você pode fazer isso em qualquer classe, desde que o processador de anotações tenha a chance de ver o código. Para este exemplo, esse código abre um JOptionPane com a mensagem.

@EventListener public static void onMessage(final MessageEvent event) { JOptionPane.showMessageDialog( null, event.getMessage(), "Message Event", JOptionPane.INFORMATION\_MESSAGE); }

Este ouvinte de exemplo MessageListener é um ouvinte coletor que receberá todos os objetosMessageEvent despachados pelo barramento de eventos.

A única coisa que resta a fazer é despachar um MessageEvent através do EventBus :

EventBus.dispatch(new MessageEvent( this, "message", "Hello World!"));

O construtor MessageEvent usa a origem do evento, o nome do evento e a mensagem. Este evento irá para qualquer número de métodos anotados por @EventListener e aceitando um MessageEvent como parâmetro.

### Onde está a saída?

Quando você tiver compilado o código, dê uma olhada no diretório de construção. Ao lado de cada arquivo de classe com um método @EventListener , deve haver um arquivo de origem Java \* EventDispatcher e o arquivo de classe resultante. Se esses arquivos não estiverem lá, verifique se você configurou o ambiente de criação para que o EventListenerAnnotationProcessor fique visível para as ferramentas de processamento de anotações e compiladores (consulte a seção "Organizando seu código").

Esperançosamente, você descobrirá que usar anotações com o ServiceLoader facilitará sua vida.