1. 脚本
2. 概述

脚本语言是一种通过在运行时解释程序文本，从而避免使用通常的编辑/编译/链接/运行循环的语言，脚本语言的优势是快速变更，可以修改运行着的程序，但是，脚本语言缺乏可以使编写复杂应用受益的特性，因此人们通常将脚本语言与传统语言的优势结合起来。脚本API使你可以在Java平台上实现这个目的，直接在Java程序中调用脚本。

1. 脚本引擎
2. 概述

脚本引擎是一个可以执行用某种特定语言编写的脚本的类库，当虚拟机启动时，它会发现可用的脚本引擎，为了枚举这些引擎，需要构造一个ScriptEngineManager，并调用getEngineFactories方法，可以向每个引擎工厂询问他们所支持的引擎名，MIME类型和文件扩展名，通过这三个参数来获取引擎。

ScriptEngine engine = manager.getEngineByName(“nashorn”);

1. 脚本计算与绑定

拥有了引擎，可以直接调用脚本：

Object result = engine.eval(scriptString);

engine eval(“n = 1”);

可以在一个引擎上调用多个脚本，如果一个脚本定义了变量，函数或者类，那么大多数引擎都会保留这些定义。

可以直接向引擎中添加新的变量绑定，绑定由名字及其关联的java对象构成：

engine.put(“k”, 1);

脚本代码从“引擎作用域”中的绑定里读取k的定义，还可以添加到全局作用域中，任何添加到ScriptEngineManager中的绑定对于所有引擎都是可视的。

1. 重定向输入和输出

可以通过调用脚本上下文的setReader和setWriter方法来重定向脚本的标准输入和输出。

var writer = new StringWriter();

engine.getContext().setWriter(new PrintWriter(writer, true));

那么脚本产生的输出都会被发送到writer上。

1. 调用脚本的函数

对于许多脚本引擎而言，都可以调用脚本语言的函数，而不用对实际的脚本代码进行计算，提供这种功能的脚本引擎实现了Invocable接口。要调用一个函数，需要使用函数名来调用invokeFunction方法，函数名后面是函数的参数。

engine.eval(“function greet(how, whom) {return how + “,” + whom }”);

result = ((Invocable) engine).invokeFunction(“greet”, “hello”, “world”);

进一步，可以让脚本引擎实现一个Java接口，然后就可以用Java方法调用的语法来调用脚本函数，需要为该Java接口中的每个方法都提供一个函数。如下面的接口：

public interface Greeter{

String welcome(String whom);

}

那么调用时，首先获取给定接口的实现类，该实现用脚本引擎中的函数实现了接口中的方法。

engine.eval(“function welcome(whom) {return “hello” + whom}”);

Greeter g = ((Invocable ) engine).getInterface(Greeter.class);

result = g.welcome(“world”);

1. 编译脚本

某些脚本引擎出于执行效率的考虑，将脚本代码编译为某种中间格式，这些引擎实现了Compilable接口，一旦脚本被编译，可以直接执行它，当然只有在需要重复执行某个脚本时，才会编译脚本。

1. 注解
2. 概述

注解是那些插入到源代码中使用其他工具可以对其进行处理的标签，这些工具可以在源码层次上操作，或者可以处理编译器在其中放置了注解的类文件。注解并不会改变程序的编译方式，Java编译器对于包含注解和不包含注解的代码会生成相同的虚拟机指令。

1. 注解的用途

附属文件的自动生成，例如部署描述符或者bean信息类；

测试，日志，事务语义等代码的自动生成；

1. 注解的使用

Java中，注解被当做一个修饰符俩使用的，它被置于注解项之前，中间没有分号，每个注解的名称之前都加上了@符号，如下所示：

public class MyClass{

@Test

public void checkRandom();

}

@Test注解自身不会做任何事，它需要工具的支持，如测试一个类时，Junit4测试工具可能会调用所有标识为@Test的方法。

注解也可以定义成包含元素的形式，如：

@Test(timeout=”10000”)

这些元素可以被读取这些注解的工具去处理，除了方法以外，还可以注解类，成员以及局部变量，这些注解可以存在于任何可以放置像public这样的修饰符的地方。每个注解需要通过一个注解接口来定义，这些接口中的方法与注解中的元素相对应，如下所示：

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Test

{

long timeout() default 0L;

}

@interface声明创建一个真正的Java接口，处理注解的工具将接受那些实现了注解接口的对象，这类工具可以调用timeout方法来获取某个特定Test注解的timeout元素。

@Target注解和@Retention是元注解，它们注解了Test注解，将Test注解标识为一个只能运用到方法上的注解，并且当类文件载入到虚拟机上时，它仍然可以保留下来，具体的在后面讨论。

定义了注解对象，这些注解只会存在于源文件中，编译器会将它们置于类文件中，并且虚拟机会载入这些文件。我们需要使用一个能够分析注解的工具来接受这些注解对象，这个工具类将会调用一个处理方法，该方法将会枚举出所有方法的注解对象，如果存在对应的注解对象，就对这个对象进行处理，注解也就真正起了作用。

1. 注解的语法

所有的注解接口都隐式地扩展自java.lang.annotation.Annotaion接口，无需为注解接口提供实现类。如果注解中某个元素的值并未指定，就是用声明的默认值，默认值并不是和注解存储在一起的，它们是动态计算得来的。

有两个特殊的注解可以简化注解，一个是标记注解，注解中没有任何元素或者所有元素都使用默认值，如下所示：

@BugReport

另一个是单值注解，如果一个元素具有特殊的名字value，且没有指定其他元素，则可以忽略这个value的元素名，直接在写值：

@ActionLiener(“yellow”)

一个项可以有多个注解，且如果注解的作者声明注解为可重复的，那么可以重复使用同一个注解。一个注解元素可以是另一个注解，因此可以将注解嵌套使用。

1. 注解的位置

注解可以出现在很多地方，这些地方分为两类，一类是声明注解，一类是类型用法声明注解。声明注解放置到其他修饰符的前面，类型用法注解放置到其他修饰符的后面。

1. 标准注解
2. 概述

Java SE定义了大量的注解接口，其中四个是元注解，用于描述注解接口的行为属性，其他三个是规则接口，用来注解代码中的项。

1. 用于编译的注解
2. @Deprecated

将项标记为过时的，该注解会一直持久化到运行时。

1. @SuppressWarnings

告知编译器阻止特定类型的警告信息。

1. @Override

应用到方法上，检查这种注解的方法是否真正覆盖了一个来自超类的方法。

1. @Generated

目的是供代码生成工具使用，任何生成的源代码都可以被注解，从而与开发者提供的代码区分开来。每个注解都必须包含一个表示代码生成器的唯一标识符。

1. 用于管理资源的注解
2. @PostConstruct和@PreDestroy

用于控制对象声明周期的环境中，例如Web容器，标记了这些注解的方法应该在对象被构建之后，或者对象被移除之前，紧接着调用。

1. @Resource

用于资源的注入，如配置文件的注入

1. 元注解
2. @Target

应用于注解，以限制该注解可以应用到哪些项上面，如方法，类…

@Target({ElementType.TYPE, ElementType.METHOD})

一条没有@Target限制的注解可以应用于任何项上。

1. @Retention

用于指定一条注解应该保留多长时间，默认是RetentionPolicy.CLSS：

SOURCE：不包含在类文件中的注解，既不会进入类文件

CLASS： 包含在类文件中的注解，但是虚拟机不需要载入它们

RUNTIME：包含在类文件中的注解，并且由虚拟机载入，通过反射API可以获得它们

1. @Documented

为像Javadoc这样的归档工具提供了提示，用于文件的归档。

1. @Inherited

只能用于对类的注解，如果一个类具有继承注解，那么它的所有子类都自动具有相同的注解，这个注解特别适用于Serializable这样的标记接口。

1. 源码级注解处理
2. 概述

注解的另一种用途是自动处理源代码以产生更多的源代码，配置文件，脚本或其他任何我们想要生成的东西。

1. 注解处理器

注解处理已经被集成到Java编译器中，编译器会定位源文件中的注解，每个注解处理器都会依次执行，并得到它表示感兴趣的注解，如果某个注解处理器创建了一个新的源文件，那么上述过程将会重复执行，如果某次处理循环没有再产生任何新的源文件，那么就编译所有的源文件。

注解处理器通常通过扩展AbstractProcessor类而实现Processor接口，处理器可以声明具体的注解类型，如下所示：

@SupportedAnnotationTypes(“com.horstmann.annotations.ToString”)

@SupportedSourceVersion(SourceVersion.RELEASE\_8)

public class ToStringAnnotationProcessor extends AbstractProcessor{}

1. 字节码工程
2. 概述

处理在运行期或者源码级别上处理注解，还可以在字节码级别上处理，除非将注解在源码级别上删除，否则它们会一直存在于类文件中。

1. 流程
2. 加载类文件中的字节码
3. 定位所有的方法
4. 对于每个方法，检查它是不是都有某个注解
5. 如果有，则在方法中添加某些指令的字节码