重复注解

假设，现在有一个服务我们需要定时运行，就像Linux中的cron一样，假设我们需要它在每周三的12点运行一次，那我们可能会定义一个注解，有两个代表时间的属性。

```
public @interface Schedule {
    int dayOfWeek() default 1;   // 周几
    int hour() default 0;    // 几点
}
```

所以我们可以给对应的服务方法上使用该注解，代表运行的时间：

```
public class ScheduleService {
    // 每周3的12点运行
    @Schedule(dayOfWeek = 3, hour = 12)
    public void start() {
        // 执行服务
    }
}
```

那么如果我们需要这个服务在每周四的13点也需要运行一下，如果是JDK8之前，那么...尴尬了！你不能像下面的代码，会编译错误

```
public class ScheduleService {
    // jdk中两个相同的注解会编译报错
    @Schedule(dayOfWeek = 3, hour = 12)
    @Schedule(dayOfWeek = 4, hour = 13)
    public void start() {
        // 执行服务
    }
}
```

那么如果是JDK8，你可以改一下注解的代码，在自定义注解上加上@Repeatable元注解，并且指定重复注解的存储注解（其实就是需要需要数组来存储重复注解），这样就可以解决上面的编译报错问题。

```
@Repeatable(value = Schedule.Schedules.class)
public @interface Schedule {
    int dayOfWeek() default 1;
    int hour() default 0;

    @interface Schedules {
        Schedule[] value();
    }
}
```

同时，反射相关的API提供了新的函数getAnnotationsByType()来返回重复注解的类型。

添加main方法：

```
    public static void main(String[] args) {
        try {
            Method method = ScheduleService.class.getMethod("start");
            for (Annotation annotation : method.getAnnotations()) {
                System.out.println(annotation);
            }
            for (Schedule s : method.getAnnotationsByType(Schedule.class)) {
                System.out.println(s.dayOfWeek() + "|" + s.hour());
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
    }
```

输出：

```
@repeatannotation.Schedule$Schedules(value=[@repeatannotation.Schedule(hour=12,
dayOfWeek=3), @repeatannotation.Schedule(hour=13, dayOfWeek=4)])
3|12
4|13
```

扩展注解

> 注解就相当于一种标记，在程序中加了注解就等于为程序加了某种标记。

JDK8之前的注解只能加在:

```
public enum ElementType {
    /** Class, interface (including annotation type), or enum declaration */
    TYPE,        // 类、接口、枚举
    /** Field declaration (includes enum constants) */
    FIELD,       // 类型变量
    /** Method declaration */
    METHOD,      // 方法
    /** Parameter declaration */
    PARAMETER,   // 方法参数
    /** Constructor declaration */
    CONSTRUCTOR,  // 构造方法
    /** Local variable declaration */
    LOCAL_VARIABLE, // 局部变量
    /** Annotation type declaration */
    ANNOTATION_TYPE, // 注解类型
    /** Package declaration */
    PACKAGE           // 包
}
```

JDK8中新增了两种：

1. TYPE_PARAMETER，表示该注解能写在类型变量的声明语句中。
2. TYPE_USE，表示该注解能写在使用类型的任何语句中

checkerframework中的各种校验注解，比如：@Nullable， @NonNull等等。

```
public class GetStarted {
    void sample() {
        @NonNull Object ref = null;
    }
}
```

更好的类型推测机制

直接看代码：

```
public class Value<T> {
    public static<T> T defaultValue() {
        return null;
    }

    public T getOrDefault(T value, T defaultValue) {
        return value != null ? value : defaultValue;
    }

    public static void main(String[] args) {
        Value<String> value = new Value<>();
        System.out.println(value.getOrDefault("22", Value.defaultValue()));
    }
}
```

上面的代码重点关注**value.getOrDefault("22", Value.defaultValue())**，在JDK8中不会报错，那么在JDK7中呢？

答案是会报错：`Wrong 2nd argument type. Found: 'java.lang.Object', required: 'java.lang.String'`。所以Value.defaultValue()的参数类型在JDK8中可以被推测出，所以就不必明确给出。

参数名字保留在字节码中

先来想一个问题：**JDK8之前，怎么获取一个方法的参数名列表？**

在JDK7中一个Method对象有下列方法：

- `Method.getParameterAnnotations()` 获取方法参数上的注解
- `Method.getParameterTypes()` 获取方法的参数类型列表

但是没有能够获取到方法的参数名字列表！

在JDK8中增加了两个方法

- `Method.getParameters()` 获取参数名字列表
- `Method.getParameterCount()` 获取参数名字个数

用法：

StampedLock

是对读写锁ReentrantReadWriteLock的增强，该类提供了一些功能，优化了读锁、写锁的访问，同时使读写锁之间可以互相转换，更细粒度控制并发。

ReentrantLock

ReentrantLock类，实现了Lock接口，是一种可重入的独占锁，它具有与使用 synchronized 相同的一些基本行为和语义，但功能更强大。ReentrantLock内部通过内部类实现了AQS框架(AbstractQueuedSynchronizer)的API来实现独占锁的功能。

ReentrantReadWriteLock

ReentrantReadWriteLock和ReentrantLock不同，ReentrantReadWriteLock实现的是ReadWriteLock接口。

> 读写锁的概念：加读锁时其他线程可以进行读操作但不可进行写操作，加写锁时其他线程读写操作都不可进行。

但是，读写锁如果使用不当，很容易产生“饥饿”问题，在ReentrantReadWriteLock中，当读锁被使用时，如果有线程尝试获取写锁，该写线程会阻塞。，在读线程非常多，写线程很少的情况下，很容易导致写线程“饥饿”，虽然使用“公平”策略可以一定程度上缓解这个问题，但是“公平”策略是以牺牲系统吞吐量为代价的。

并行数组

Java 8增加了大量的新方法来对数组进行并行处理。可以说，最重要的是parallelSort()方法，因为它可以在多核机器上极大提高数组排序的速度。下面的例子展示了新方法（parallelXxx）的使用。

下面的代码演示了先并行随机生成20000个0-1000000的数字，然后打印前10个数字，然后使用并行排序，再次打印前10个数字。

```
        long[] arrayOfLong = new long [ 20000 ];

        Arrays.parallelSetAll( arrayOfLong,
                index -> ThreadLocalRandom.current().nextInt( 1000000 ) );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
                i -> System.out.print( i + " " ) );
        System.out.println();

        Arrays.parallelSort( arrayOfLong );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
                i -> System.out.print( i + " " ) );
        System.out.println();
```

获取方法参数的名字

在Java8之前，我们如果想获取方法参数的名字是非常困难的，需要使用ASM、javassist等技术来实现，现在，在Java8中则可以直接在Method对象中就可以获取了。

```
public class ParameterNames {
    public void test(String p1, String p2) {
    }

    public static void main(String[] args) throws Exception {
        Method method = ParameterNames.class.getMethod("test", String.class,
String.class);

        for (Parameter parameter : method.getParameters()) {
            System.out.println(parameter.getName());
        }
        System.out.println(method.getParameterCount());

    }
}
```

输出：

```
arg0
arg1
2
```

从结果可以看出输出的参数个数正确，但是名字不正确！需要在**编译**时增加**–parameters**参数后再运行。

在Maven中增加：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <compilerArgument>-parameters</compilerArgument>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin>
```

输出结果则变为：

```
p1
p2
2
```

CompletableFuture

当我们Javer说异步调用时，我们自然会想到Future，比如：

```java
public class FutureDemo {

    /**
     * 异步进行一个计算
     * @param args
     */
    public static void main(String[] args) {

        ExecutorService executor = Executors.newCachedThreadPool();
        Future<Integer> result = executor.submit(new Callable<Integer>() {
            public Integer call() throws Exception {
                int sum=0;
                System.out.println("正在计算...");
                for (int i=0; i<100; i++) {
                    sum = sum + i;
                }
                Thread.sleep(TimeUnit.SECONDS.toSeconds(3));
                System.out.println("算完了！ ");
                return sum;
            }
        });

        System.out.println("做其他事情...");

        try {
            System.out.println("result:" + result.get());
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("事情都做完了...");

        executor.shutdown();
    }
}
```

那么现在如果想实现异步计算完成之后，立马能拿到这个结果继续异步做其他事情呢？这个问题就是一个线程依赖另外一个线程，这个时候Future就不方便，我们来看一下CompletableFuture的实现：

```
public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(10);



        CompletableFuture result = CompletableFuture.supplyAsync(() -> {
            int sum=0;
            System.out.println("正在计算...");
            for (int i=0; i<100; i++) {
                sum = sum + i;
            }
            try {
                Thread.sleep(TimeUnit.SECONDS.toSeconds(3));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+"算完了！");
            return sum;
        }, executor).thenApplyAsync(sum -> {
            System.out.println(Thread.currentThread().getName()+"打印"+sum);
            return sum;
        }, executor);



        System.out.println("做其他事情...");

        try {
            System.out.println("result:" + result.get());
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("事情都做完了...");
```

```
        executor.shutdown();
    }
```

结果：

```
正在计算...
做其他事情...
pool-1-thread-1算完了!
pool-1-thread-2打印4950
result:4950
事情都做完了...
```

只需要简单的使用thenApplyAsync就可以实现了。

当然CompletableFuture还有很多其他的特性，我们下次单独开个专题来讲解。

> Java8的特性还有Stream和Optional，这两个也是用的特别多的，相信很多同学早有耳闻，并且已经对这两个的特性有所了解，所以本片博客就不进行讲解了，有机会再单独讲解，可讲的内容还是非常之多的
>
> 对于Java8，新增的特性还是非常之多的，就是目前Java11已经出了，但是Java8中的特性肯定会一直在后续的版本中保留的，至于这篇文章的这些新特性我们估计用的比较少，所以特已此篇来进行一个普及，希望都有所收货。

Java虚拟机（JVM）的新特性

PermGen空间被移除了，取而代之的是Metaspace（JEP 122）。JVM选项-XX:PermSize与-XX:MaxPermSize分别被-XX:MetaSpaceSize与-XX:MaxMetaspaceSize所代替。