链滴

Lambda 表达式对递归的优化 (上) - 使用尾递归

作者：zhaozhizheng

原文链接：https://ld246.com/article/1619577773586

来源网站：链滴

原文链接

递归优化

很多算法都依赖于递归，典型的比如分治法(Divide-and-Conquer)。但是普通的递归算法在处理规较大的问题时，常常会出现StackOverflowError。处理这个问题，我们可以使用一种叫做尾调用(Tail-all Optimization)的技术来对递归进行优化。同时，还可以通过暂存子问题的结果来避免对子问题的复求解，这个优化方法叫做备忘录(Memoization)。

本文首先对尾递归进行介绍，下一票文章中会对备忘录模式进行介绍。

使用尾调用优化

当递归算法应用于大规模的问题时，容易出现StackOverflowError，这是因为需要求解的子问题过多递归嵌套层次过深。这时，可以采用尾调用优化来避免这一问题。该技术之所以被称为尾调用，是因在一个递归方法中，最后一个语句才是递归调用。这一点和常规的递归方法不同，常规的递归通常发在方法的中部，在递归结束返回了结果后，往往还会对该结果进行某种处理。

Java在编译器级别并不支持尾递归技术。但是我们可以借助Lambda表达式来实现它。下面我们会通在阶乘算法中应用这一技术来实现递归的优化。以下代码是没有优化过的阶乘递归算法：

```
public class Factorial {
public static int factorialRec(final int number) {
if(number == 1)
return number;
else
return number * factorialRec(number - 1);
}
}
```

以上的递归算法在处理小规模的输入时，还能够正常求解，但是输入大规模的输入后就很有可能抛出StackOverflowError：

```
try {
System.out.println(factorialRec(20000));
} catch(StackOverflowError ex) {
System.out.println(ex);
}

// java.lang.StackOverflowError
```

出现这个问题的原因不在于递归本身，而在于在等待递归调用结束的同时，还需要保存了一个number变量。因为递归方法的最后一个操作是乘法操作，当求解一个子问题时(factorialRec(number - 1))，要保存当前的number值。所以随着问题规模的增加，子问题的数量也随之增多，每个子问题对应着用栈的一层，当调用栈的规模大于JVM设置的阈值时，就发生了StackOverflowError。

转换成尾递归

转换成尾递归的关键，就是要保证对自身的递归调用是最后一个操作。不能像上面的递归方法那样：后一个操作是乘法操作。而为了避免这一点，我们可以先进行乘法操作，将结果作为一个参数传入到归方法中。但是仅仅这样仍然是不够的，因为每次发生递归调用时还是会在调用栈中创建一个栈帧(Stack Frame)。随着递归调用深度的增加，栈帧的数量也随之增加，最终导致StackOverflowError。可以过将递归调用延迟化来避免栈帧的创建，以下代码是一个原型实现：

```
public static TailCall<Integer> factorialTailRec(
final int factorial, final int number) {
if (number == 1)
return TailCalls.done(factorial);
else
return TailCalls.call(() -> factorialTailRec(factorial * number, number - 1));
}
```

需要接受的参数factorial是初始值，而number是需要计算阶乘的值。 我们可以发现，递归调用体现了call方法接受的Lambda表达式中。以上代码中的TailCall接口和TailCalls工具类目前还没有实现。

创建TailCall函数接口

TailCall的目标是为了替代传统递归中的栈帧，通过Lambda表达式来表示多个连续的递归调用。所以们需要通过当前的递归操作得到下一个递归操作，这一点有些类似UnaryOperator函数接口的apply法。同时，我们还需要方法来完成这几个任务：

判断递归是否结束了

得到最后的结果

触发递归

因此，我们可以这样设计TailCall函数接口：

```
@FunctionalInterface
public interface TailCall<T> {
TailCall<T> apply();
```

```
default boolean isComplete() { return false; }
default T result() { throw new Error("not implemented"); }
default T invoke() {
return Stream.iterate(this, TailCall::apply)
.filter(TailCall::isComplete)
.findFirst()
.get()
.result();
}
}
```

isComplete，result和invoke方法分别完成了上述提到的3个任务。只不过具体的isComplete和resul还需要根据递归操作的性质进行覆盖，比如对于递归的中间步骤，isComplete方法可以返回false，而对于递归的最后一个步骤则需要返回true。对于result方法，递归的中间步骤可以抛出异常，而递的最终步骤则需要给出结果。

invoke方法则是最重要的一个方法，它会将所有的递归操作通过apply方法串联起来，通过没有栈帧尾调用得到最后的结果。串联的方式利用了Stream类型提供的iterate方法，它本质上是一个无穷列，这也从某种程度上符合了递归调用的特点，因为递归调用发生的数量虽然是有限的，但是这个数量可以是未知的。而给这个无穷列表画上终止符的操作就是filter和findFirst方法。因为在所有的递归调中，只有最后一个递归调用会在isComplete中返回true，当它被调用时，也就意味着整个递归调用链结束。最后，通过findFirst来返回这个值。

如果不熟悉Stream的iterate方法，可以参考上一篇文章，在其中对该方法的使用进行了介绍。

创建TailCalls工具类

在原型设计中，会调用TailCalls工具类的call和done方法：

call方法用来得到当前递归的下一个递归

done方法用来结束一系列的递归操作，得到最终的结果

```
public class TailCalls {
public static <T> TailCall<T> call(final TailCall<T> nextCall) {
return nextCall;
}
public static <T> TailCall<T> done(final T value) {
return new TailCall<T>() {
@Override public boolean isComplete() { return true; }
@Override public T result() { return value; }
@Override public TailCall<T> apply() {
throw new Error("end of recursion");
}
};
}
```

}

在done方法中，我们返回了一个特殊的TailCall实例，用来代表最终的结果。注意到它的apply方法实现成被调用抛出异常，因为对于最终的递归结果，是没有后续的递归操作的。

以上的TailCall和TailCalls虽然是为了解决阶乘这一简单的递归算法而设计的，但是它们无疑在任何需尾递归的算法中都能够派上用场。

使用尾递归函数

使用它们来解决阶乘问题的代码很简单：

System.out.println(factorialTailRec(1, 5).invoke()); // 120

System.out.println(factorialTailRec(1, 20000).invoke()); // 0

第一个参数代表的是初始值，第二个参数代表的是需要计算阶乘的值。

但是在计算20000的阶乘时得到了错误的结果，这是因为整型数据无法容纳这么大的结果，发生了溢。对于这种情况，可以使用BigInteger来代替Integer类型。

实际上factorialTailRec的第一个参数是没有必要的，在一般情况下初始值都应该是1。所以我们可以出相应地简化：

public static int factorial(final int number) {

return factorialTailRec(1, number).invoke();

}

// 调用方式

System.out.println(factorial(5));

System.out.println(factorial(20000));

使用BigInteger代替Integer

主要就是需要定义decrement和multiple方法来帮助完成大整型数据的阶乘操作：

public class BigFactorial {

public static BigInteger decrement(final BigInteger number) {

return number.subtract(BigInteger.ONE);

}

```
public static BigInteger multiply(
    final BigInteger first, final BigInteger second) {
    return first.multiply(second);
}

final static BigInteger ONE = BigInteger.ONE;
final static BigInteger FIVE = new BigInteger("5");
final static BigInteger TWENTYK = new BigInteger("20000");
//...

private static TailCall<BigInteger> factorialTailRec(
    final BigInteger factorial, final BigInteger number) {
    if(number.equals(BigInteger.ONE))
        return done(factorial);
```

```
    else
        return call(() ->
            factorialTailRec(multiply(factorial, number), decrement(number)));
}

public static BigInteger factorial(final BigInteger number) {
    return factorialTailRec(BigInteger.ONE, number).invoke();
}

}
```