二

23 逃逸分析

我们知道，Java 中 `Iterable` 对象的 foreach 循环遍历是一个语法糖，Java 编译器会将该语法糖编译为调用 `Iterable` 对象的 `iterator` 方法，并用所返回的 `Iterator` 对象的 `hasNext` 以及 `next` 方法，来完成遍历。

```
public void forEach(ArrayList<Object> list, Consumer<Object> f) {
  for (Object obj : list) {
    f.accept(obj);
  }
}
```

举个例子，上面的 Java 代码将使用 foreach 循环来遍历一个 `ArrayList` 对象，其等价的代码如下所示：

```
public void forEach(ArrayList<Object> list, Consumer<Object> f) {
  Iterator<Object> iter = list.iterator();
  while (iter.hasNext()) {
    Object obj = iter.next();
    f.accept(obj);
  }
}
```

这里我也列举了所涉及的 `ArrayList` 代码。我们可以看到， `ArrayList.iterator` 方法将创建一个 `ArrayList$Itr` 实例。

```
public class ArrayList ... {
  public Iterator<E> iterator() {
    return new Itr();
  }
  private class Itr implements Iterator<E> {
    int cursor;       // index of next element to return
    int lastRet = -1; // index of last element returned; -1 if no such
    int expectedModCount = modCount;
    ...
    public boolean hasNext() {
      return cursor != size;
    }
    @SuppressWarnings("unchecked")
    public E next() {
```

```
      checkForComodification();
      int i = cursor;
      if (i >= size)
        throw new NoSuchElementException();
      Object[] elementData = ArrayList.this.elementData;
      if (i >= elementData.length)
        throw new ConcurrentModificationException();
      cursor = i + 1;
      return (E) elementData[lastRet = i];
    }
    ...
    final void checkForComodification() {
      if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
    }
  }
}
```

因此，有同学认为我们应当避免在热点代码中使用 foreach 循环，并且直接使用基于 `ArrayList.size` 以及 `ArrayList.get` 的循环方式（如下所示），以减少对 Java 堆的压力。

```
public void forEach(ArrayList<Object> list, Consumer<Object> f) {
  for (int i = 0; i < list.size(); i++) {
    f.accept(list.get(i));
  }
}
```

实际上，Java 虚拟机中的即时编译器可以将 `ArrayList.iterator` 方法中的实例创建操作给优化掉。不过，这需要方法内联以及逃逸分析的协作。

在前面几篇中我们已经深入学习了方法内联，今天我便来介绍一下逃逸分析。

逃逸分析

逃逸分析是“一种确定指针动态范围的静态分析，它可以分析在程序的哪些地方可以访问到指针”（出处参见 [1]）。

在 Java 虚拟机的即时编译语境下，逃逸分析将判断**新建**的对象是否**逃逸**。即时编译器判断对象是否逃逸的依据，一是对象是否被存入堆中（静态字段或者堆中对象的实例字段），二是对象是否被传入未知代码中。

前者很好理解：一旦对象被存入堆中，其他线程便能获得该对象的引用。即时编译器也因此无法追踪所有使用该对象的代码位置。

关于后者，由于 Java 虚拟机的即时编译器是以方法为单位的，对于方法中未被内联的方法

调用，即时编译器会将其当成未知代码，毕竟它无法确认该方法调用会不会将调用者或所传入的参数存储至堆中。因此，我们可以认为方法调用的调用者以及参数是逃逸的。

通常来说，即时编译器里的逃逸分析是放在方法内联之后的，以便消除这些“未知代码”入口。

回到文章开头的例子。理想情况下，即时编译器能够内联对 `ArrayList$Itr` 构造器的调用，对 `hasNext` 以及 `next` 方法的调用，以及当内联了 `Itr.next` 方法后，对 `checkForComodification` 方法的调用。

如果这些方法调用均能够被内联，那么结果将近似于下面这段伪代码：

```
public void forEach(ArrayList<Object> list, Consumer<Object> f) {
  Itr iter = new Itr; // 注意这里是 new 指令
  iter.cursor = 0;
  iter.lastRet = -1;
  iter.expectedModCount = list.modCount;
  while (iter.cursor < list.size) {
    if (list.modCount != iter.expectedModCount)
      throw new ConcurrentModificationException();
    int i = iter.cursor;
    if (i >= list.size)
      throw new NoSuchElementException();
    Object[] elementData = list.elementData;
    if (i >= elementData.length)
      throw new ConcurrentModificationException();
    iter.cursor = i + 1;
    iter.lastRet = i;
    Object obj = elementData[i];
    f.accept(obj);
  }
}
```

可以看到，这段代码所新建的 `ArrayList$Itr` 实例既没有被存入任何字段之中，也没有作为任何方法调用的调用者或者参数。因此，逃逸分析将断定该实例不逃逸。

基于逃逸分析的优化

即时编译器可以根据逃逸分析的结果进行诸如锁消除、栈上分配以及标量替换的优化。

我们先来看一下锁消除。如果即时编译器能够证明锁对象不逃逸，那么对该锁对象的加锁、解锁操作没有意义。这是因为其他线程并不能获得该锁对象，因此也不可能对其进行加锁。在这种情况下，即时编译器可以消除对该不逃逸锁对象的加锁、解锁操作。

实际上，传统编译器仅需证明锁对象不逃逸出线程，便可以进行锁消除。由于 Java 虚拟机

即时编译的限制，上述条件被强化为证明锁对象不逃逸出当前编译的方法。

在介绍 Java 内存模型时，我曾提过 `synchronized (new Object()) {}` 会被完全优化掉。这正是因为基于逃逸分析的锁消除。由于其他线程不能获得该锁对象，因此也无法基于该锁对象构造两个线程之间的 happens-before 规则。

`synchronized (escapedObject) {}` 则不然。由于其他线程可能会对逃逸了的对象 `escapedObject` 进行加锁操作，从而构造了两个线程之间的 happens-before 关系。因此即时编译器至少需要为这段代码生成一条刷新缓存的内存屏障指令。

不过，基于逃逸分析的锁消除实际上并不多见。一般来说，开发人员不会直接对方法中新构造的对象进行加锁。事实上，逃逸分析的结果更多被用于将新建对象操作转换成栈上分配或者标量替换。

我们知道，Java 虚拟机中对象都是在堆上分配的，而堆上的内容对任何线程都是可见的。与此同时，Java 虚拟机需要对所分配的堆内存进行管理，并且在对象不再被引用时回收其所占据的内存。

如果逃逸分析能够证明某些新建的对象不逃逸，那么 Java 虚拟机完全可以将其分配至栈上，并且在 new 语句所在的方法退出时，通过弹出当前方法的栈桢来自动回收所分配的内存空间。这样一来，我们便无须借助垃圾回收器来处理不再被引用的对象。

不过，由于实现起来需要更改大量假设了“对象只能堆分配”的代码，因此 HotSpot 虚拟机**并没有**采用栈上分配，而是使用了标量替换这么一项技术。

所谓的标量，就是仅能存储一个值的变量，比如 Java 代码中的局部变量。与之相反，聚合量则可能同时存储多个值，其中一个典型的例子便是 Java 对象。

标量替换这项优化技术，可以看成将原本对对象的字段的访问，替换为一个个局部变量的访问。举例来说，前面经过内联之后的 forEach 代码可以被转换为如下代码：

```
public void forEach(ArrayList<Object> list, Consumer<Object> f) {
  // Itr iter = new Itr; // 经过标量替换后该分配无意义，可以被优化掉
  int cursor = 0;     // 标量替换
  int lastRet = -1;   // 标量替换
  int expectedModCount = list.modCount; // 标量替换
  while (cursor < list.size) {
    if (list.modCount != expectedModCount)
      throw new ConcurrentModificationException();
    int i = cursor;
    if (i >= list.size)
      throw new NoSuchElementException();
    Object[] elementData = list.elementData;
    if (i >= elementData.length)
      throw new ConcurrentModificationException();
```

```
      cursor = i + 1;
      lastRet = i;
      Object obj = elementData[i];
      f.accept(obj);
    }
  }
```

可以看到，原本需要在内存中连续分布的对象，现已被拆散为一个个单独的字段 `cursor` ， `lastRet` ，以及 `expectedModCount` 。这些字段既可以存储在栈上，也可以直接存储在寄存器中。而该对象的对象头信息则直接消失了，不再被保存至内存之中。

由于该对象没有被实际分配，因此和栈上分配一样，它同样可以减轻垃圾回收的压力。与栈上分配相比，它对字段的内存连续性不做要求，而且，这些字段甚至可以直接在寄存器中维护，无须浪费任何内存空间。

部分逃逸分析

C2 的逃逸分析与控制流无关，相对来说比较简单。Graal 则引入了一个与控制流有关的逃逸分析，名为部分逃逸分析（partial escape analysis）[2]。它解决了所新建的实例仅在部分程序路径中逃逸的情况。

举个例子，在下面这段代码中，新建实例只会在进入 if-then 分支时逃逸。（对 `hashCode` 方法的调用是一个 HotSpot intrinsic，将被替换为一个无法内联的本地方法调用。）

```
public static void bar(boolean cond) {
  Object foo = new Object();
  if (cond) {
    foo.hashCode();
  }
}
// 可以手工优化为：
public static void bar(boolean cond) {
  if (cond) {
    Object foo = new Object();
    foo.hashCode();
  }
}
```

假设 if 语句的条件成立的可能性只有 1%，那么在 99% 的情况下，程序没有必要新建对象。其手工优化的版本正是部分逃逸分析想要自动达到的成果。

部分逃逸分析将根据控制流信息，判断出新建对象仅在部分分支中逃逸，并且将对象的新建操作推延至对象逃逸的分支中。这将使得原本因对象逃逸而无法避免的新建对象操作，不再出现在只执行 if-else 分支的程序路径之中。

综上，与 C2 所使用的逃逸分析相比，Graal 所使用的部分逃逸分析能够优化更多的情况，不过它编译时间也更长一些。

总结与实践

今天我介绍了 Java 虚拟机中即时编译器的逃逸分析，以及基于逃逸分析的优化。

在 Java 虚拟机的即时编译语境下，逃逸分析将判断新建的对象是否会逃逸。即时编译器判断对象逃逸的依据有两个：一是看对象是否被存入堆中，二是看对象是否作为方法调用的调用者或者参数。

即时编译器会根据逃逸分析的结果进行优化，如锁消除以及标量替换。后者指的是将原本连续分配的对象拆散为一个个单独的字段，分布在栈上或者寄存器中。

部分逃逸分析是一种附带了控制流信息的逃逸分析。它将判断新建对象真正逃逸的分支，并且支持将新建操作推延至逃逸分支。

今天的实践环节有两项内容。

第一项内容，我们来验证一下 `ArrayList.iterator` 中的新建对象能否被逃逸分析所优化。运行下述代码并观察 GC 的情况。你可以通过虚拟机参数 `-XX:-DoEscapeAnalysis` 来关闭默认开启的逃逸分析。

```
// Run with
// java -XX:+PrintGC -XX:+DoEscapeAnalysis EscapeTest
import java.util.ArrayList;
import java.util.function.Consumer;

public class EscapeTest {

  public static void forEach(ArrayList<Object> list, Consumer<Object> f) {
    for (Object obj : list) {
      f.accept(obj);
    }
  }

  public static void main(String[] args) {
    ArrayList<Object> list = new ArrayList<>();
    for (int i = 0; i < 100; i++) {
      list.add(i);
    }
    for (int i = 0; i < 400_000_000; i++) {
      forEach(list, obj -> {});
    }
  }
```

```
}
```

第二项内容，我们来看一看部分逃逸分析的效果。你需要使用附带 Graal 编译器的 Java 版本，如 Java 10，来运行下述代码，并且观察 GC 的情况。你可以通过虚拟机参数 `-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler` 来启用 Graal。

```
// Run with
// java -Xlog:gc Foo
// java -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler -Xlog:gc Foo
public class Foo {
  long placeHolder0;
  long placeHolder1;
  long placeHolder2;
  long placeHolder3;
  long placeHolder4;
  long placeHolder5;
  long placeHolder6;
  long placeHolder7;
  long placeHolder8;
  long placeHolder9;
  long placeHoldera;
  long placeHolderb;
  long placeHolderc;
  long placeHolderd;
  long placeHoldere;
  long placeHolderf;
  public static void bar(boolean condition) {
    Foo foo = new Foo();
    if (condition) {
      foo.hashCode();
    }
  }
  public static void main(String[] args) {
    for (int i = 0; i < Integer.MAX_VALUE; i++) {
      bar(i % 100 == 0);
    }
  }
}
```

[1] https://zh.wikipedia.org/wiki/ 逃逸分析 [2] http://www.ssw.uni-linz.ac.at/Research/Papers/Stadler14/Stadler2014-CGO-PEA.pdf

上一页 下一页