转载请注明链接: https://blog.csdn.net/feather_wch/article/details/82719313

> invokedynamic指令到底是什么？方法句柄又是什么？Java8是如何利用invokedynamic实现Lambda表达式的?
> 本文对invokedynamic的知识点进行归纳总结。

JVM invokedynamic调用指令

版本号:2018/09/16-1(0:00)

基础-方法句柄(32)

赛马问题

1、如何让非马的类能和马一样参加赛马比赛?

```
class Horse{
    public void race(){
        System.out.println("马在赛跑");
    }
}

class Duck{
    public void race(){
        System.out.println("鸭子在赛跑");
    }
}

class Deer{
    public void race(){
        System.out.println("马在赛跑");
    }
}
```

> 1. 第一种方法：将非马的类型包装成马类进行赛跑。
> 2. 第二种方法：通过反射机制，查找并调用各个类型中的赛跑方法。

2、invokedynamic指令的作用?

> 1. Java7引入的新指令
> 2. 该指令抽象出 调用点 这一个概念
> 3. 允许应用程序将调用点链接至任何符合条件的方法上
> 4. 解决了赛马问题，但是比包装和反射更高效。

3、invokedynamic的应用场景

> lambda表达式

4、invokedynamic底层机制的基石：方法句柄

方法句柄

5、方法句柄是什么?

> 1. MethodHandle-方法句柄
> 2. 是一种更加底层、更加灵活的方法抽象(Java7引入)
> 3. 方法句柄是一种强类型、并且能够被直接执行的引用。
> 4. 该引用可以指向常规的静态方法、实例方法、构造器、字段。
> 5. 当指向字段时，方法句柄实则指向包含字段访问字节码的 虚构方法 ，语义上等价于目标字段的 getter 或者 setter 方法。(但不会直接指向目标字段所在类中的 getter/setter)

6、方法句柄的类型是什么?

> 1. MethodType-方法句柄类型
> 2. 方法句柄类型由所指向方法的参数类型、返回类型组成的。
> 3. 是用来 确认方法句柄是否适配的唯一关键。
> 4. 当使用方法句柄时，并不需要关心方法句柄所指向方法的类名或者方法名

7、如果一个兔子的赛跑方法和睡觉方法的参数类型、返回类型一致，如何判断是哪一个方法?

> 1. 兔子传递的方法句柄，将无法判断是哪一个方法。

创建

8、方法句柄的创建?

> 1. 方法句柄的创建需要通过 MethodHandles.Lookup 类来完成。
> 2. Lookup提供了多个API，可以使用反射的Method来查找方法句柄。
> 3. 也可以使用类、方法名、方法句柄类型来查找。
>
> 1-Horse类:

```
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodHandles.Lookup;

public class Horse {
    public void race(){
        System.out.println("马在赛跑");
    }
    // 返回Lookup类
    public static Lookup lookup(){
        return MethodHandles.lookup();
    }
}
```

> 2-通过反射的Method来获取方法句柄(MethodHandle)

```
// 1、获取到Lookup
MethodHandles.Lookup lookup = Horse.lookup();
// 2、获取到反射的Method
Method method = Horse.class.getDeclaredMethod("race", Void.class);
// 3、获取到方法句柄
MethodHandle methodHandle = lookup.unreflect(method);
```

> 3-通过类、方法名、方法句柄类型，来获取MethodHandle

```
// 1、获取到Lookup
MethodHandles.Lookup lookup = Horse.lookup();
// 2、方法句柄类型，第一个是返回值类型，第二个是参数类型
MethodType methodType = MethodType.methodType(Void.class, Void.class);
// 3、创建方法句柄，依次使用类、方法名、方法句柄类型。
MethodHandle methodHandle = lookup.findVirtual(Horse.class, "race", methodType);
```

9、Lookup有哪些创建MethodHandle的API?

> 1. 对于invokestatic调用指令的静态方法：使用 `lookup.findStatic()`
> 2. 对于invokevirutal、invokeinterface调用指令的方法：使用 `lookup.findVirtual()`
> 3. 对于invokespecial调用指令的实例方法：使用 `lookup.findSpecial()`

10、方法句柄(MethodHandle)的权限问题?

> 1. 权限问题上和反射API不同
> 2. MethodHandle的权限检查是在句柄创建阶段完成
> 3. 实际调用过程中，JVM不会检查方法句柄的权限，如果被多次调用，性能比反射要高。
> 4. 方法句柄没有运行时检查权限，因此app需要负责方法句柄的管理。

11、方法句柄和反射的性能对比

> 1. 方法句柄的权限检查在创建阶段完成
> 2. 反射的权限检查是处于运行时。
> 3. JVM不会去检查方法句柄的权限。
> 4. 因此在多次调用的情况下，方法句柄会节省出权限检查的开销。

12、方法句柄的权限是如何判定的?

> 1. 方法句柄的访问权限不取决于MethodHandle的创建位置
> 2. 而是取决于Lookup对象的创建位置。
> 3. 如果Lookup对象在private字段所在类中获取，就拥有了对该private字段的访问权限。
> 4. 这样在所在类的外边，也可以通过该Lookup对象创建该private字段的getter或者setter。

操作

13、方法句柄的调用方法?

> 分为两种：
>
> 1. 需要严格匹配参数类型的invokeExact
> 2. 自动适配参数类型的invoke

14、invokeExact的使用

> 1. 对匹配参数类型非常严格
> 2. 如果MethodHandle接收一个Object类型的参数，传入String作为实参就会在运行时抛出异常
> 3. 必须要显式转换类型，让形参和实参一致。

```
methodHandle.invokeExact(str);
methodHandle.invokeExact((Object)str);
```

15、重载方法时的显式转化类型是为了什么?

> 1. 在显式转化后，参数的声明类型发生了改变，从而匹配到不同的方法描述符
> 2. 根据方法描述符的不同，选取到不同的目标方法。
> 3. 调用方法句柄也是一样的道理，并且涉及到 签名多态性-signature polymorphism 的概念

16、方法句柄API的注解 @PolymorphicSignature 是什么？这个方法描述符和invokeExact的关系?

> 1. 通过该注解，Java编译器在遇到这些方法调用时，会根据传入参数的声明类型来生成方法描述符
> 2. 而不是采用目标方法所声明的描述符。
> 3. invokeExact会严格要求方法句柄的类型和调用时的方法描述符是否匹配
> 4. 如下面的例子：就是根据 传入参数的声明类型 来生成描述符

```
methodHandle.invokeExact(str);
// 描述符：MethodHandle.invokeExact:(Ljava/lang/String;)V

methodHandle.invokeExact((Object)str);
// 描述符: MethodHandle.invokeExact:(Ljava/lang/Object;)V
```

17、什么是签名多态性?

> 1. 方法用 @PolymorphicSignature 注解进行标记
> 2. Java编译器在调用这些方法时，会根据传入参数的声明类型来生成方法描述符

18、invoke的使用和内部原理

```
methodHandle.invoke(str);
```

> 1. 该方法自动适配参数类型，也是一个签名多态性的方法
> 2. 调用方法句柄前，会调用 MethodHandle.asType() 生成一个适配器方法句柄，对传入的参数进行适配。

> 3. 方法句柄的返回值，也会进行适配然后返回给调用者。

19、方法句柄的增加、删除、修改参数的操作

> 本质都是通过生成另一个方法句柄来实现
>
> 1. 修改操作：MethodHandle.asType()
> 2. 删除操作: MethodHandles.dropArguments(), 将传入的部分参数直接抛弃，然后调用另一个方法句柄。
> 3. 增加操作：MethodHandle.bindTo(), 在传入的参数中增加额外的参数，再调用另一个方法句柄。

20、Java 8中捕获类型的Lambda表达式是如何实现的?

> 利用的方法句柄的增加操作来实现的。

21、MethodHandle的增加操作还可以实现方法的柯里化

> 1. 一个指向f(x,y)的句柄，将x绑定为4，生成一个方法句柄 `g(y)=f(4，y)` 。
> 2. 执行过程中，调用g(y)方法句柄，就会默认去调用f(4,y)的方法句柄

22、MethodHandle.bindTo()限制了只能使用引用类型，普遍是用来绑定this的，为什么能用来实现柯里化(将int x, int y 的第一个参数指定为常数4)?

> 1. bindTo的确限制了引用类型，但是方法句柄不区分调用者和参数。
> 2. 因此依然可以用做其他效果：可以用Integer，然后使用静态方法，或者在使用virtual方法时将bindTo返回的方法句柄再bindTo一个Integer.valueOf(4)。
> 3. BoundMethodHandle里有很多非public的bindArgumentXXX()方法，和这些有相关性。

方法句柄的实现

23、方法句柄的使用实例

```
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodHandles.Lookup;

public class Horse {
    public static void race(){
        System.out.println("马在赛跑");
    }
    // 返回Lookup类
    public static Lookup lookup(){
        return MethodHandles.lookup();
    }
}
```

```
// 1、获取到Lookup
MethodHandles.Lookup lookup = Horse.lookup();
// 2、方法句柄类型，第一个是返回值类型，第二个是参数类型
MethodType methodType = MethodType.methodType(void.class);
// 3、创建方法句柄，依次使用类、方法名、方法句柄类型。
MethodHandle methodHandle = lookup.findStatic(Horse.class, "race", methodType);

// 调用
methodHandle.invoke();
// 打印出
马在赛跑
```

24、查看方法句柄调用invoke、invokeExact的栈轨迹

```
public class Horse {
    public static void race(){
      // 打印栈轨迹
        new Exception().printStackTrace();
        System.out.println("马在赛跑");
    }
    // xxx
}
```

> 栈信息

```
java.lang.Exception
        at Horse.race(Horse.java:6)
        at java.base/java.lang.invoke.DirectMethodHandle$Holder.invokeStatic(DirectMethodHandle$Holder:1000
        at java.base/java.lang.invoke.LambdaForm$MH/411631404.invoke_MT(LambdaForm$MH:1000017)
        at Main.main(Main.java:17)
```

1. 启用 `-XX:+ShowHiddenFrames` 参数来打印被JVM隐藏的栈信息
2. 此外需要启用 `-XX:+UnlockDiagnosticVMOptions` 参数，不然会报错。
3. LambdaForm: 是一个特殊的适配器，会对invokeExact调用做特殊处理，调用至一个共享的、与方法句柄类型相关的LambdaForm适配器中。

25、通过虚拟机参数将LambdaForm导出成class文件

1. `-Djava.lang.invoke.MethodHandle.DUMP_CLASS_FILES=true`
2. 生成的class文件，位于 `DUMP_CLASS_FILES` 目录下。
3. 会依次调用：
   1. 调用checkGenericType()/checkExactType()检查参数类型
   1. 调用checkCustomized()，会在句柄执行次数超过阈值时进行优化。对应参数 `-Djava.lang.invoke.MethodHandle.CUSTOMIZE_THRESHOLD` ，默认值127。
   1. 最后调用方法句柄的invokeBasic()。该调用还是会做特殊处理，将调用至方法句柄本身所持有的适配器中。(同样是一个LambdaForm)

```
Code:
  stack=2, locals=3, args_size=2
     0: aload_0
     1: checkcast     #14                 // class java/lang/invoke/MethodHandle
     4: aload_1
     5: checkcast     #16                 // class java/lang/invoke/MethodType
     8: invokestatic  #22                 // Method java/lang/invoke/Invokers.checkGenericType:(Ljava/l
    11: astore_2
    12: aload_2
    13: invokestatic  #26                 // Method java/lang/invoke/Invokers.checkCustomized:(Ljava/la
    16: aload_2
    17: invokevirtual #30                 // Method java/lang/invoke/MethodHandle.invokeBasic:()V
    20: return
```

26、方法句柄执行次数的阈值如何修改?

> 1. 使用虚拟机参数 `-Djava.lang.invoke.MethodHandle.CUSTOMIZE_THRESHOLD`
> 2. 默认值是127

27、MethodHandle.invokeBasic内部的适配器LambdaForm做了哪些事情?

> 1. 获取到MethodHandle的MemberName类型的字段，并调用其linkToStatic方法
> 2. linkToStatic会根据MemberName参数所存储的方法地址或者方法表索引，直接跳转到目标方法。

28、checkCustomized()如何根据方法句柄的执行次数进行优化?

> 1. 方法句柄一开始持有的适配器是共享的
> 2. 多次调用之后，Invokers.checkCustomized()会为该MethodHandle生成一个特殊的适配器
> 3. 该适配器会将 `MethodHandle` 作为常量，直接获取其 `MemeberName` 字段。然后调用 `linkToStatic` 直接跳转到目标方法。

29、方法句柄的内联问题

> 1. 方法句柄的调用和反射调用一样，都是间接调用。
> 2. 和反射一样面临无法内联的问题。
> 3. 方法句柄的内联瓶颈在于即时编译器能否将该方法句柄识别为常量。
> 4. 反射调用的内联瓶颈不在这里。是什么?

方法句柄性能

30、方法句柄的性能测试

```
public class Main {
    // 测试的方法
    public static void say(String s){
    }
    public static void main(String args[]) throws Throwable {
        // 1、获取到Lookup
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        // 2、方法句柄类型，第一个是返回值类型，第二个是参数类型
        MethodType methodType = MethodType.methodType(void.class, String.class);
        // 3、创建方法句柄，依次使用类、方法名、方法句柄类型。
        MethodHandle methodHandle = lookup.findStatic(Main.class, "say", methodType);
        /**
         * 性能测试，使用invokeExact
         */
        long current = System.currentTimeMillis();
        for(int i = 1; i <= 2_000_000_000; i++){
            if(i % 100_000_000 == 0){
                long temp = System.currentTimeMillis();
                System.out.println(temp - current);
                current = temp;
            }
            //Main.say("String");
            methodHandle.invoke("String");
        }
    }
}
```

1. 每一亿次所消耗的时间。平均:443ms
2. 直接调用: 132ms
3. 性能开销是直接调用的3.35倍。

1	2	3	4	5	6	7	8	9	10
491	512	436	387	402	411	388	444	527	433

31、为什么方法句柄调用的性能开销这么大?

1. 即时编译器无法将该 方法句柄 识别为常量，从而无法内联。

32、将MethodHandle作为常量进行性能测试

1. 性能开销和直接调用一致
2. 即时编译器将方法句柄完全内联，成为空操作。

```
public class Main {
    /**====================================
     *  方法句柄作为常量
     *====================================*/
    static final MethodHandle methodHandle;
    static{
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodType methodType = MethodType.methodType(void.class, String.class);

        // 创建方法句柄
        try {
            methodHandle = lookup.findStatic(Main.class, "say", methodType);
        } catch (Throwable e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }

    }

    // 测试的方法
    public static void say(String s){
    }
    public static void main(String args[]) throws Throwable {
        /**
         * 性能测试，使用invokeExact
         */
        long current = System.currentTimeMillis();
        for(int i = 1; i <= 2_000_000_000; i++){
            if(i % 100_000_000 == 0){
                long temp = System.currentTimeMillis();
                System.out.println(temp - current);
                current = temp;
            }
//            Main.say("String");
            methodHandle.invoke("String");
        }
    }
}
```

invokedynamic调用指令(28)

1、invokedynamic指令机制概述?

> 1. Java7引入的新调用指令，用于支持动态语言的方法调用。
> 2. 将 调用点(CallSite) 抽象成Java类
> 3. 并且将方法调用和方法链接暴露给应用程序。(这些工作原本应该由JVM控制)
> 4. 运行过程中。每一条invokedynamic指令将捆绑一个调用点，并且会调用 调用点 所链接的方法句柄。

2、invokedynamic如何生成调用点?

1. 第一次执行invokedynamic指令时，JVM会调用该指令所对应的启动方法(BootStrapMethod)
2. 启动方法会生成调用点，并且绑定至该invokedynamic指令中
3. 后续运行时，JVM会直接调用绑定的调用点所链接的方法句柄

3、在字节码中，启动方法是用方法句柄来指定的。

1. 这个方法句柄指向一个返回类型为调用点的静态 方法
2. 该方法必须接收三个固定的参数
    1. 一个 Lookup 类实例
    2. 一个目标方法 名字的字符串
    3. 一个该调用点能够链接的方法句柄的类型。
3. BottStrapMethod大概代码参考如下：

```
public static CallSite bootstrap(MethodHandles.Lookup lookup, //Lookup实例
                                 String targetMethodName, // 目标方法名
                                 MethodType methodType){ // 该调用点链接的方法句柄的类型
    // 1、创建方法句柄
    MethodHandle methodHandle = lookup.findVirtual(Horse.class, targetMethodName, MethodTyp
    // 2、创建调用点。通过旧方法句柄生成方法句柄的适配器，依次创建调用点。
    ConstantCallSite constantCallSite = new ConstantCallSite(methodHandle.asType(methodType
    return constantCallSite;
}
```

调用点

4、调用点是什么?

1. 调用点通过启动方法-BootStrapMethod生成，并且和invokedynamic指令绑定。
2. 启动方法中，该调用点会和对应的方法句柄所链接。
3. 后续执行invokedynamic指令，会找到该指令绑定的调用点，再找到该调用点链接的方法句柄，然后直接执行该句柄。

5、ConstantCallSite是什么?

> 一种不可以更改链接对象的调用点

6、MutableCallSite 和 VolatileCallSite是什么?

1. Java 核心类库提供多种可以更改链接对象的调用点.
2. MutableCallSite 和 VolatileCallSite就是这种吊用点。
3. 这两个调用点的区别类似于 正常字段 和`volatile 字段之间的区别。
4. 此外，应用程序还可以自定义调用点类，来满足特定的重链接需求。

7、invokedynamic如何允许应用程序自己决定链接至哪一个方法中?

invokedynamic使用实例

8、invokedynamic调用Horse.race()方法：借助ASM字节码框架生成invokedynamic指令

> 1. Java不支持通过代码直接生成invokedynamic指令
> 2. 需要借助ASM工具
>
> 1- 马类、鹿类

```
public class Horse {
    public void race(){
        System.out.println("马在赛跑");
    }
}
public class Deer {
    public void race(){
        System.out.println("鹿在赛跑");
    }
}
```

> 2-比赛类: 包括startRace()和bootstrap启动方法。会通过ASM工具在startRace()中生成invokedynamic指令。最终会运行bootstrap方法，获取到调用点，并绑定到invokedynamic指令上。

```
/**
 * 赛马比赛
 */
public class Match {
    // 赛跑
    public static void startRace(Object object){
        // aload obj
        // invokedynamic race()
    }
    /**==================================================
     * BootStrap
     * @param lookup Lookup实例
     * @param targetMethodName 目标方法名
     * @param methodType 该调用点链接的方法句柄的类型
     * @return 调用点
     *==================================================*/
    public static CallSite bootstrap(MethodHandles.Lookup lookup, String targetMethodName, Meth
        // 1、创建方法句柄
        MethodHandle methodHandle = lookup.findVirtual(Horse.class, targetMethodName, MethodTyp
        // 2、创建调用点。通过旧方法句柄生成方法句柄的适配器，依次创建调用点。
        ConstantCallSite constantCallSite = new ConstantCallSite(methodHandle.asType(methodType
        return constantCallSite;
    }
}
```

> 3-借助ASMHelper在Match.startRace()方法中生成invokedynamic指令

```
import jdk.internal.org.objectweb.asm.*;

import java.io.IOException;
import java.lang.invoke.CallSite;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ASMHelper implements Opcodes {
    /**==================================
     * 1、自定义的类访问者：MyClassVisitor
     *  1. visitMethod()获取到方法的访问请求，根据判断可以替换成自定义的MethodVisitor。
     *==========================================*/
    static class MyClassVisitor extends ClassVisitor {
        public MyClassVisitor(int api, ClassVisitor cv) {
            super(api, cv);
        }
        @Override
        public MethodVisitor visitMethod(int access, String name, String descriptor, String sig
            MethodVisitor visitor = super.visitMethod(access, name, descriptor, signature, exce
            // 将main()方法替换为自定义的MethodVisitor
            if ("startRace".equals(name)) {
                return new MyMethodVisitor(ASM5, visitor);
            }
            return visitor;
        }
    }
    /**==================================
     * 2、自定义的方法访问者
     *================================*/
    static class MyMethodVisitor extends MethodVisitor {

        // BootStrapMethod-启动方法
        private static final String BOOTSTRAP_CLASS_NAME = Match.class.getName().replace('.', '
        private static final String BOOTSTRAP_METHOD_NAME = "bootstrap";
        private static final String BOOTSTRAP_METHOD_DESC = MethodType
                .methodType(CallSite.class, MethodHandles.Lookup.class, String.class, MethodTyp
                .toMethodDescriptorString();
        // 目标方法
        private static final String TARGET_METHOD_NAME = "race";
        private static final String TARGET_METHOD_DESC = "(Ljava/lang/Object;)V";

        private MethodVisitor mv;
        public MyMethodVisitor(int api, MethodVisitor mv) {
            super(api, null);
            this.mv = mv;
        }
        @Override
        public void visitCode() {
            mv.visitCode();
            // 1、在startRace()中生成字节码: aload obj
            mv.visitVarInsn(ALOAD, 0); //局部变量指令
```

```
            // 2、Match类的bootstrap方法
            Handle handle = new Handle(H_INVOKESTATIC,
                    BOOTSTRAP_CLASS_NAME,
                    BOOTSTRAP_METHOD_NAME,
                    BOOTSTRAP_METHOD_DESC,
                    false);
            /**============================================================
             * 3、生成invokedynamic指令
             *   1. 将Match类的bootstrap()生成的调用点，绑定到invokedynamic指令上。
             *   2. 还会将目标方法的方法句柄链接到调用点上。方便后续直接调用。
             *============================================================*/
            mv.visitInvokeDynamicInsn(TARGET_METHOD_NAME, TARGET_METHOD_DESC, handle);

            //4、return指令
            mv.visitInsn(RETURN);
            mv.visitMaxs(1, 1);
            mv.visitEnd();
        }
    }

    /**====================================================================
     *  运行能将Match的class文件中的字节码进行修改，增加invokedynamic指令
     *====================================================================*/
    public static void main(String[] args) throws IOException {
        // 1、Class的读取者。去加载Student的原始字节，并且翻译成访问请求。
        ClassReader cr = new ClassReader("Match");
        // 2、Class的写入者。
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        // 3、Reader和Writer的中间层，对访问操作进行拦截和处理。如果找到目标方法，就替换成自定义的M
        ClassVisitor cv = new ASMHelper.MyClassVisitor(ASM5, cw);
        cr.accept(cv, ClassReader.SKIP_FRAMES);
        // 4、将Write中的数据转为字节数组，写入到class文件中。
        Files.write(Paths.get("Match.class"), cw.toByteArray());

    }
}
```

1. 运行ASMHelper的main()方法，会加载Match文件，然后生成 `Match.class`
2. 或者采用命令行完成上述操作：
    1. 生成Match.class: `javac -encoding UTF-8 Match.java`
    2. 编译ASMHelper.java: `javac -cp asm-all-6.0_BETA.jar -encoding UTF-8 ASMHelper.java`
    3. 运行： `java ASMHelper`

4-ASMHelper会在startRace()中生成如下字节码：

```
public static void startRace(java.lang.Object);
   0: aload_0
   1: invokedynamic #80,  0 // race:(Ljava/lang/Object;)V
   6: return
```

> 5-测试运行

```
public class Main {
    public static void main(String args[]) throws Throwable {
        // 开始赛跑
        Match.startRace(new Horse());
    }
}
```

9、invokedynamic指令所有的方法描述符为何和实际调用的方法都不一致?

> 1. invokedynamic指令的方法描述符：race:(Ljava/lang/Object;)V
> 2. 和Horse.race()和Deer.race()方法的描述符都不一致
> 3. 因为该指令调用的是方法句柄，而方法句柄会将调用者作为第一个参数。
> 4. 所以race:(Ljava/lang/Object;)V会多了一个参数Object，这个就是调用者。

10、进一步改造：通过invokedynamic调用任意类的race()方法

> 1-实现一个简单的单态内联缓存，调用者的类型命中缓存中的类型，就直接调用缓存中的方法句柄。否则更新缓存。

```
// 需要更改ASMHelper.MyMethodVisitor中的BOOTSTRAP_CLASS_NAME
public class MonomorphicInlineCache {
    private final MethodHandles.Lookup lookup;   private final String name;

    public MonomorphicInlineCache(MethodHandles.Lookup lookup, String name) {
        this.lookup = lookup;
        this.name = name;
    }

    private Class<?> cachedClass = null;
    private MethodHandle methodHandle = null;

    public void invoke(Object receiver) throws Throwable {
        if (cachedClass != receiver.getClass()) {
            cachedClass = receiver.getClass();
            methodHandle = lookup.findVirtual(cachedClass, name, MethodType.methodType(void.cla
        }
        methodHandle.invoke(receiver);
    }

    // 作为bootstrap，将

    /**============================
     * bootstrap：启动方法
     *  1、ASMHelper中将Match的startRace()的invokedynamic的调用点绑定为该单态内联缓存中的调用点。
     *  2、该调用点是链接至invoke()方法，而不是之前的Horse的race()方法
     *  3、invoke()中会根据receiver的类型去调用具体的race()方法
     *==========================*/
    public static CallSite bootstrap(MethodHandles.Lookup l, String name, MethodType callSiteTy
        MonomorphicInlineCache ic = new MonomorphicInlineCache(l, name);
        MethodHandle mh = l.findVirtual(MonomorphicInlineCache.class, "invoke", MethodType.meth
        return new ConstantCallSite(mh.bindTo(ic));
    }
}
```

> 2-更改ASMHelper中的bootstrap的name

```
// 启动方法：  将单态内联缓存中的bootstrap()生成的调用点，绑定至invokedynamic。
private static final String BOOTSTRAP_CLASS_NAME = MonomorphicInlineCache.class.getName().repla
```

11、为什么能将内联缓存的invoke方法的句柄链接到调用点上？不是只能链接到需要调用的目标方法吗?

> 1. 调用点仅要求 方法句柄的类型 能够匹配。
> 2. 因此可以链接到符合该条件的invoke方法上。
> 3. invoke方法内部再根据参数类型，去获取对应类型的 `MethodHandle` ， 再调用方法句柄的invoke()方法---这样就完美完成了Horse或者Deer的race()方法的调用。

12、invokedynamic()支持调用任意类型的race()方法的实现和之前只能调用Horse类的race()方法的实现，有什么区别?

> 1. Horse版本: 采用Match中的bootstrap生成的调用点，并且将Horse的race方法句柄链接到调用点上。
> 2. 任意类型版本：采用单态内联缓存中的bootstrap生成的调用点，将内联缓存的invoke()方法和调用点链接起来。invoke()方法中根据参数的类型，去调用其方法句柄。

Java8 Lambda表达式

13、Java8是如何实现Lambda表达式的?

> 1. 借助 `invokedynamic` 指令实现
> 2. Java编译器利用invokedynamic指令生成 实现了函数式接口的适配器

14、函数式接口是什么?

> 1. 仅包括一个非default接口方法的接口，一般使用 `@FunctionalInterface` 注解
> 2. 但就算没有该注解，Java编译器也会将符合条件的接口辨认为 函数式接口

15、Lambda实例: 函数式接口

```
int x = 10;
IntStream.of(1, 2, 3)
        .map(i -> i * 2) //1次映射
        .map(i -> i * x) //1次映射
        .forEach((i)->System.out.println(""+i));
```

> 1. 上面对IntStream中的元素进行了两次映射
> 2. 映射方法map()所接受的参数是 `IntUnaryOperator`-一个函数式接口
> 3. 在运行过程中会将 `i -> i * 2` 和 `i -> i * x` 的lambda表达式转换为 `IntUnaryOperator` 实例。
> 4. 这个转换工作就是invokedynamic指令实现的。

16、编译器对lambda表达式的处理

> 1. Java 编译器会对Lambda表达式进行解语法糖（desugar），
> 2. 会生成一个方法来保存Lambda表达式的内容。
> 3. 该方法的参数列表不仅包含原本Lambda表达式的参数，还包含所捕获的变量。
> 4. 注: 方法引用，如 `Horse::race` ，则不会生成生成额外的方法。

17、Lambda实例生成的字节码

> 1-第一个Lambda表达式 `i -> i * 2` 没有捕获其他变量。

```
// i -> i * 2
private static int lambda$0(int);
  Code:
    0: iload_0
    1: iconst_2
    2: imul
    3: ireturn
```

> 2-第二个Lambda表达式 `i -> i * x` 则会捕获局部变量 `x` : 多了一个捕获的变量参数。

```
// i -> i * x
private static int lambda$1(int, int);
  Code:
    0: iload_1
    1: iload_0
    2: imul
    3: ireturn
```

18、Lambda表达式执行invokedynamic指令的流程

> 1. 执行invokedynamic时，所对应的启动方法(bootstrap)会通过ASM 来生成一个适配器类。
> 2. 这个适配器类实现了对应的函数式接口，如接口: IntUnaryOperator。
> 3. 启动方法(bootstrap)会返回调用点:ConstantCallSite，该调用点会链接到返回适配器类实例的方法句柄。
> 4. 根据Lambda表达式是否捕获其他变量，启动方法生成的适配器类和所链接的方法句柄都不同。
>   1. 没有捕获其他变量。启动方法将新建一个适配器类的实例，并且生成一个特殊的方法句柄，始终返回该实例。(只新建一次实例)
>   2. 捕获了其他变量。每次执行invokedynamic，都会新建一个适配器类的实例，防止变量被改变导致问题。(每次都会去调用其 `get$Lambda` 的方法来创建实例)

19、Lambda表达式的线程安全问题？如何保证?

> 1. 因为线程安全问题，不能够共享同一个适配器类的实例。
> 2. 每次执行invokedynamic指令时，所调用的方法句柄都需要新建一个适配器实例。
> 3. 在这种情况下，启动方法生成的适配器类将包含一个额外的静态方法，来构造适配器类的实例。
> 4. 该额外的静态方法将接收这些捕获的参数，并且将它们保存为适配器类实例的实例字段。

20、如何导出Lambda表达式对应的适配器类

> 1. 可以通过虚拟机参数导出:
>
>   `-Djdk.internal.lambda.dumpProxyClasses=FolderName`
> 2. 在项目根目录下，建立一个文件夹(和src同级)，将FolderName替换为该文件夹名。

21、不捕获其他变量的适配器类

```
  IntStream.of(1, 2, 3).map(i -> i * 2)
```

```
final class Main$$Lambda$1 implements java.util.function.IntUnaryOperator
{
  private Main$$Lambda$1();
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #10                 // Method java/lang/Object."<init>":()V
         4: return

  public int applyAsInt(int);
    Code:
      stack=1, locals=2, args_size=2
         0: iload_1
         1: invokestatic  #18                 // Method Main.lambda$main$0:(I)I
         4: ireturn
}
```

22、捕获其他变量的适配器类

```
  IntStream.of(1, 2, 3).map(i -> i * x)
```

```
final class Main$$Lambda$2 implements java.util.function.IntUnaryOperator
{
  private final int arg$1;

  private Main$$Lambda$2(int);
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: invokespecial #13                 // Method java/lang/Object."<init>":()V
         4: aload_0
         5: iload_1
         6: putfield      #15                 // Field arg$1:I
         9: return

  private static java.util.function.IntUnaryOperator get$Lambda(int);
    Code:
      stack=3, locals=1, args_size=1
         0: new           #2                  // class Main$$Lambda$2
         3: dup
         4: iload_0
         5: invokespecial #19                 // Method "<init>":(I)V
         8: areturn

  public int applyAsInt(int);
    xxx
}
```

> 1. 捕获了局部变量的 Lambda 表达式多出了一个 `get$Lambda` 的方法。

2. 启动方法会将 `get$Lambda` 的方法句柄和调用点相链接。
3. 每次执行invokedynamic指令时，都会调用至 `get$Lambda` 方法，并构造一个新的适配器类实例。

Lambda性能测试

23、Lambda表达式不捕获其他变量的性能测试

1. 性能和直接调用相比区别极小
2. 即使编译器能将Lambda所用的invokedynamic以及对IntConsumer.accept()方法的调用都进行内联。最终优化为空操作。

```
public class Main {
    public static void target(int i){
        // 空实现
        // new Exception("#" + i).printStackTrace();
    }
    public static void main(String[] args) {

        long current = System.currentTimeMillis();
        long averageTime = 0;
        // 1、循环20亿次
        for(int i = 1; i <= 2_000_000_000; i++){
            // 2、1亿次测试一次时间
            if(i % 100_000_000 == 0){
                long temp = System.currentTimeMillis();
                // 3、累加得到总时间
                averageTime += temp - current;
                current = temp;
            }

            //直接调用
            //Main.target(128);
            //Lambda
            ((IntConsumer)j->Main.target(j)).accept(128);
        }
        // 4、得到平均时间
        System.out.println("averageTime = " + averageTime / 20);
    }
}
```

24、为什么性能差距不大?

1. Lambda 表达式所使用的 invokedynamic 将绑定一个 ConstantCallSite， 其链接的目标方法无法改变。
    1. 因此，即时编译器会将该目标方法直接内联进来。
    2. 对于这类没有捕 获变量的 Lambda 表达式而言，目标方法只完成了一个动作，便是加载缓存的适配器类常量。
2. 另一方面，对 IntConsumer.accept 方法的调用实则是对适配器类的 accept 方法的调用。

> 1. accept方法内部仅包含一个方法调用，调用至Java编译器在解 Lambda 语法糖时生成的方法---就是Lambda 表达式的内容，上面实例中就是Main.target()方法。
> 2. 将这几个方 法调用内联进来之后，原本对 accept 方法的调用则会被优化为空操作。

25、Lambda表达式在捕获其他变量情况下的性能测试

> 1. 性能和直接调用也没有多大区别
> 2. 因为即使编译器的逃逸分析将新建实例的操作进行了优化

```
// 其他省略
 ((IntConsumer) j -> Test.target(x + j)).accept(128);
```

26、在没有逃逸分析的时候，Lambda表达式的性能怎么样?

> 1. `-XX:DoEscapeAnalysis` 能关闭逃逸分析
> 2. 性能开销为直接调用的 `2.5`倍

27、如何让逃逸分析能免除额外的新建实例的开销。

> 需要确保两点，才能让逃逸分析认为该适配器实例不会逃逸:
>
> 1. invokedynamic指令执行的方法句柄能够内联
> 2. accept方法的调用也能内联

28、Lambda表达式性能问题总结

> 1. 不会捕获其他变量的lambda表达式，性能和直接调用没有区别。
> 2. 捕获其他变量的lambda表达式，需要确保逃逸分析能判定适配器实例不会逃逸。这样才能具有和直接调用一致的性能。
> 3. 否则每次调用都会不断生成适配器类的实例。
> 4. 建议尽量使用非捕获的Lambda表达式。

知识扩展

1、字段访问字节码的虚构方法是什么?

2、适配器模式在方法句柄中的体现

> 1. 方法句柄的调用方法 `invoke` 会自动适配参数类型
> 2. 对于参数类型的适配，就是通过适配器对传入的参数进行适配，然后去调用方法句柄。

3、什么是柯里化?

> 1. 柯里化-Currying
> 2. 把多个参数的函数变换成一个参数的函数

4、反射调用的内联瓶颈

5、逃逸分析是什么意思?

> 1. 逃逸分析就是通过数据流进行分析，判断一个对象会不会被传递到当前编译的方法之外的地方。
> 2. 比如调用一个方法，将一个新建对象作为参数传递进去。如果该方法没有被内联，则这个新建对象会逃逸。

6、javac编译错误: 编码GBK的不可映射字符

> 1. 错误: 编码GBK的不可映射字符
> 2. javac -encoding UTF-8 xxx.java

7、final关键字能否促进方法的内联从而提高性能?

> 1. 错误观点: 极客时间-Java核心技术36讲-杨晓峰, 在final文章中，提出观点“当代JVM如HotSpot非常智能, 并不是通过final关键字进行方法内联判断，因此final能帮助JVM进行方法内联的想法是完全错误的”。
> 2. 在方法句柄的性能测试中，使用 `static final` 将方法句柄声明为常量，结果性能大幅度提升和直接调用的结果一致。
> 3. 结论：final能帮助JVM进行方法的内联，从而提高性能。

问题汇总

参考资料

1. JVM是怎么实现invokedynamic的?