前不久，“虚拟机”赛马俱乐部来了个年轻人，标榜自己是动态语言，是先进分子。

这一天，先进分子牵着一头鹿进来，说要参加赛马。咱部里的老学究 Java 就不同意了呀，鹿又不是马，哪能参加赛马。

当然了，这种墨守成规的调用方式，自然是先进分子所不齿的。现在年轻人里流行的是鸭子类型（duck typing）[1]，只要是跑起来像只马的，它就是一只马，也就能够参加赛马比赛。

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

class Deer {
  public void race() {
    System.out.println("Deer.race()");
  }
}

class Cobra {
  public void race() {
    System.out.println("How do you turn this on?");
  }
}
```

（如何用同一种方式调用他们的赛跑方法？）

说到了这里，如果我们将赛跑定义为对赛跑方法（对应上述代码中的 race()）的调用的话，那么这个故事的关键，就在于能不能在马场中调用非马类型的赛跑方法。

为了解答这个问题，我们先来回顾一下 Java 里的方法调用。在 Java 中，方法调用会被编译为 invokestatic，invokespecial，invokevirtual 以及 invokeinterface 四种指令。这些指令与包含目标方法类名、方法名以及方法描述符的符号引用捆绑。在实际运行之前，Java 虚拟机将根据这个符号引用链接到具体的目标方法。

可以看到，在这四种调用指令中，Java 虚拟机明确要求方法调用需要提供目标方法的类名。在这种体系下，我们有两个解决方案。一是调用其中一种类型

的赛跑方法，比如说马类的赛跑方法。对于非马的类型，则给它套一层马甲，当成马来赛跑。

另外一种解决方式，是通过反射机制，来查找并且调用各个类型中的赛跑方法，以此模拟真正的赛跑。

显然，比起直接调用，这两种方法都相当复杂，执行效率也可想而知。为了解决这个问题，Java 7 引入了一条新的指令 invokedynamic。该指令的调用机制抽象出调用点这一个概念，并允许应用程序将调用点链接至任意符合条件的方法上。

```
public static void startRace(java.lang.Object)
    0: aload_0              // 加载一个任意对象
    1: invokedynamic race   // 调用赛跑方法
```

作为 invokedynamic 的准备工作，Java 7 引入了更加底层、更加灵活的方法抽象 ：方法句柄（MethodHandle）。

1.方法句柄的概念