```
  set id(userId) {
    wm.set(this, userId);
  }

  get id() {
    return wm.get(this);
  }
}
```

由于这个实现依赖 User 实例的对象标识，在这个实例被代理的情况下就会出问题：

```
const user = new User(123);
console.log(user.id); // 123

const userInstanceProxy = new Proxy(user, {});
console.log(userInstanceProxy.id); // undefined
```

这是因为 User 实例一开始使用目标对象作为 WeakMap 的键，代理对象却尝试从自身取得这个实例。要解决这个问题，就需要重新配置代理，把代理 User 实例改为代理 User 类本身。之后再创建代理的实例就会以代理实例作为 WeakMap 的键了：

```
const UserClassProxy = new Proxy(User, {});
const proxyUser = new UserClassProxy(456);
console.log(proxyUser.id);
```

2. 代理与内部槽位

代理与内置引用类型（比如 Array）的实例通常可以很好地协同，但有些 ECMAScript 内置类型可能会依赖代理无法控制的机制，结果导致在代理上调用某些方法会出错。

一个典型的例子就是 Date 类型。根据 ECMAScript 规范，Date 类型方法的执行依赖 this 值上的内部槽位[[NumberDate]]。代理对象上不存在这个内部槽位，而且这个内部槽位的值也不能通过普通的 get()和 set()操作访问到，于是代理拦截后本应转发给目标对象的方法会抛出 TypeError：

```
const target = new Date();
const proxy = new Proxy(target, {});

console.log(proxy instanceof Date);  // true

proxy.getDate();  // TypeError: 'this' is not a Date object
```


9.2 代理捕获器与反射方法

代理可以捕获 13 种不同的基本操作。这些操作有各自不同的反射 API 方法、参数、关联 ECMAScript 操作和不变式。

正如前面示例所展示的，有几种不同的 JavaScript 操作会调用同一个捕获器处理程序。不过，对于在代理对象上执行的任何一种操作，只会有一个捕获处理程序被调用。不会存在重复捕获的情况。

只要在代理上调用，所有捕获器都会拦截它们对应的反射 API 操作。

9.2.1 get()

get()捕获器会在获取属性值的操作中被调用。对应的反射 API 方法为 Reflect.get()。