外界修改了，自己都不知道，就可能造成内部逻辑的不一致，就像有外键关联的两个数据表，一个表修改了数据，而另外一个没有修改，这种情况是可怕的，不过因为外键约束的存在，数据库会进行这两个表的原子更新，但是内存中的对象没有这样的技术机制，而需要通过专门的设计来保证，因此不将聚合内部的对象直接暴露给外界是基本原则，外界如果需要一些数据，可以根据聚合内对象构造一个值对象使用。

例如订单上下文需要地址上下文中的地址 Address，那么只要地址上下文构造一个 Address 值对象给对方，其中包含地址唯一标识 ID，作为数据值还可以有一些其他关键信息，这样，通过构造不变的值对象，地址上下文和订单上下文之间的数据联系也只是一次性的，而不是一直耦合在一起。通过引用对象会造成耦合，而传递一个不变性的值对象给对方读取，能保证数据的真实性，也实现了数据共享。

因此，值对象一般可由聚合根随时负责构建，是聚合根中函数方法常用的返回类型。而 DTO 构建没有这些约束，非常自由，看起来方便，但带来问题是：几种 DTO 内部的字段差不多，但又不一样，而且构建的地方不一样，随时构建，非常混乱，造成大量临时对象满天飞，不但给理解代码带来障碍，也会在运行时造成垃圾回收（GC）机制不断进行垃圾回收，因为对象很多，偶尔还会触发全系统暂停，甚至内存溢出，这些都是系统稳定性的致命杀手。

值对象可以实现全局共享一个实例，这样能减少对象数目。想想，如果系统中所有的数据对象都是实体或都是 DTO，那么将非常耗费内存，而值对象因为可以共享，所以减少了内存消耗，非常经济，这也是值对象的存在价值之一。

下面再举例看看引入值对象的一些好处。平时返回的一些基本类型如 boolean 或 String，都可以使用值对象封装起来，并取一个业务相关的名称，这样可将更多业务引入抽象代码中。

假设发送电子邮件：

```
public class EmailSender {
    public boolean send(String title, String body) {
        return doSend(title, body);
    }
}
```

这个 send 方法有两个输入参数，两个参数类型还都是字符串，在调用时就有可能将输入参数颠倒，这样的错误虽然低级，但是也是繁忙中容易出现的。为了规范严谨起见，这里使用值对象封装这两个字符串：

```
public class EmailContent {
    private final String titile;
    private final String body;
    ……
}
```

通过使用正确的值对象名称，让代码记录自己的业务，只需要一瞥就可以看到对服务/组件/方法的输入有什么期望，以及在调用系统的这一部分之后会得到什么，这就是对函数