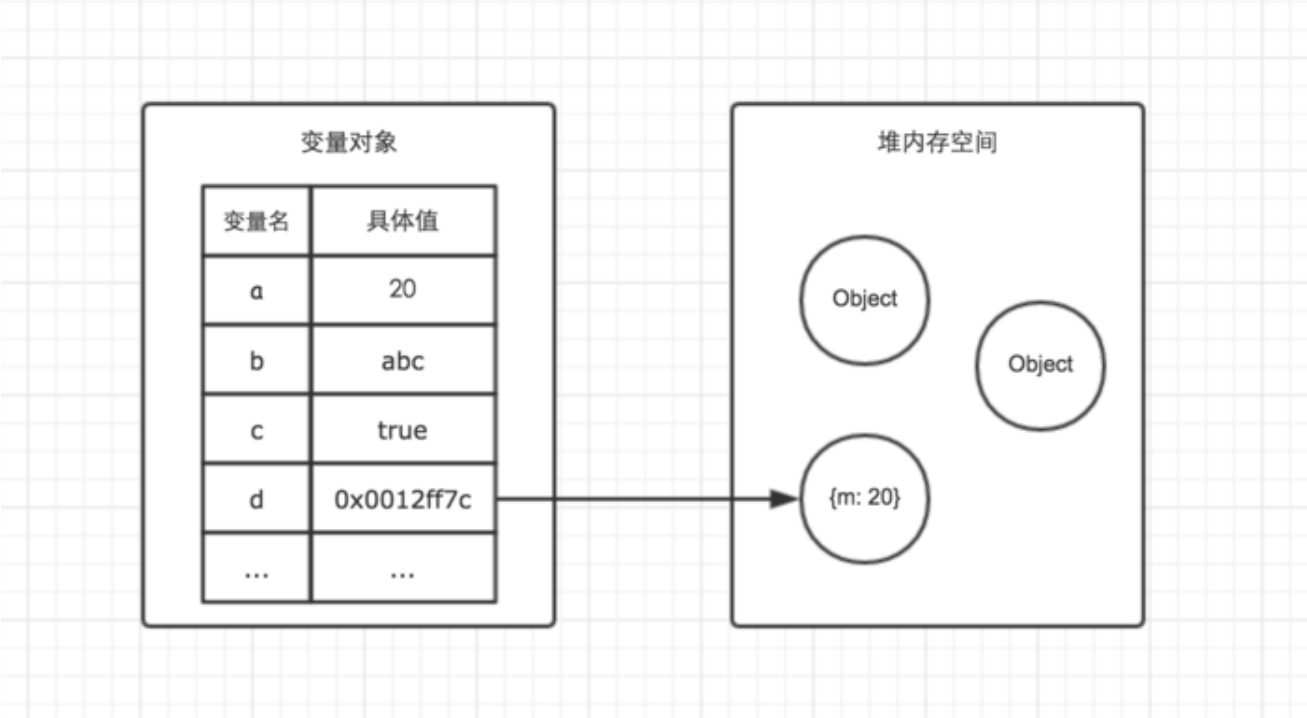
内存空间

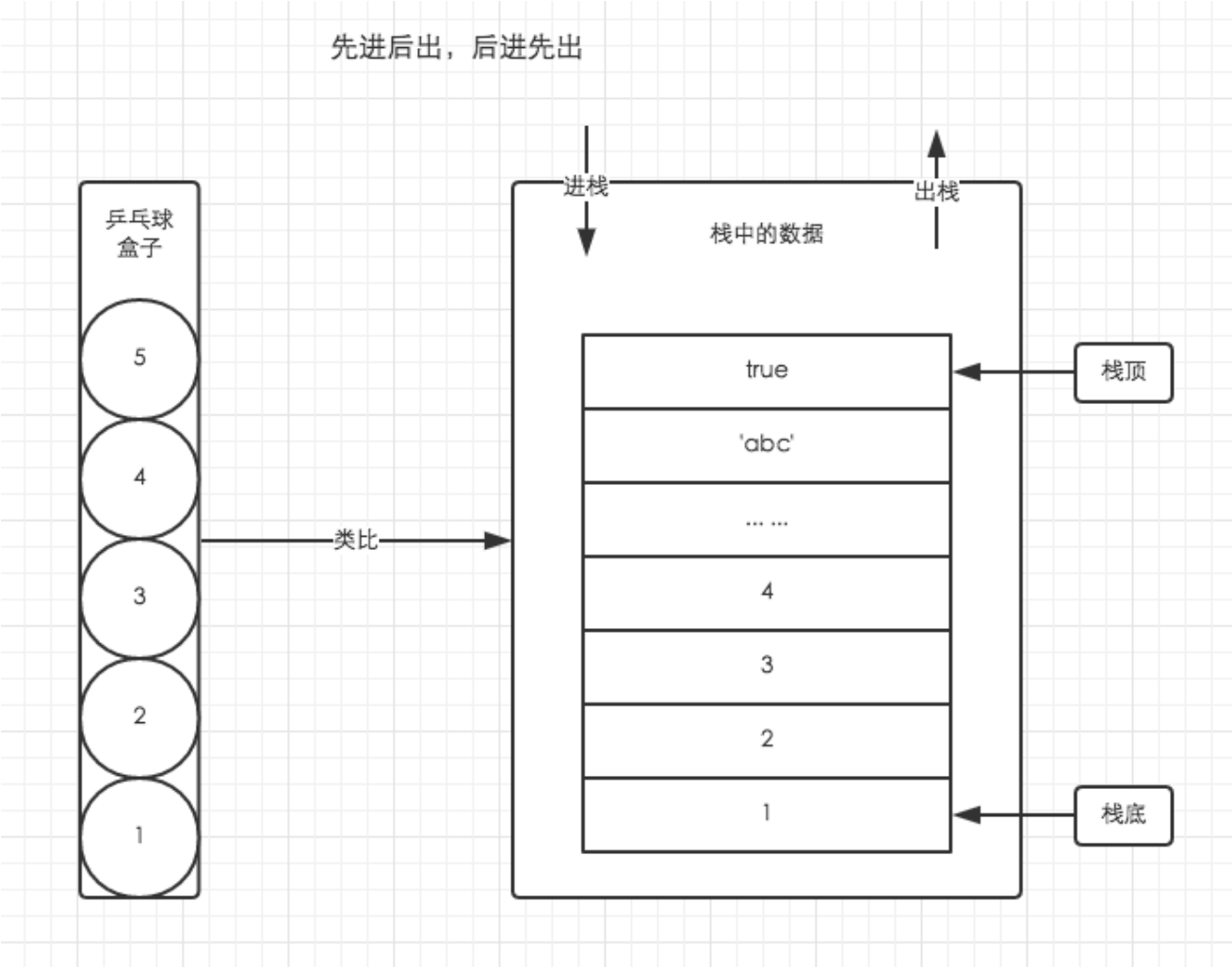


变量对象和堆内存图例如上。

```
var a = 20;
var b = 'abc';
var c = true;
var d = { m: 20 }
```

栈数据结构

`JavaScript`中并没有严格意义上区分栈内存与堆内存。因此我们可以简单粗暴的理解为`JavaScript`的所有数据都保存在堆内存中。但是在某些场景，我们仍然需要基于堆栈数据结构的思维来实现一些功能，比如`JavaScript`的执行上下文。执行上下文的执行顺序借用了栈数据结构的存取方式。因此理解栈数据结构的原理与特点十分重要。



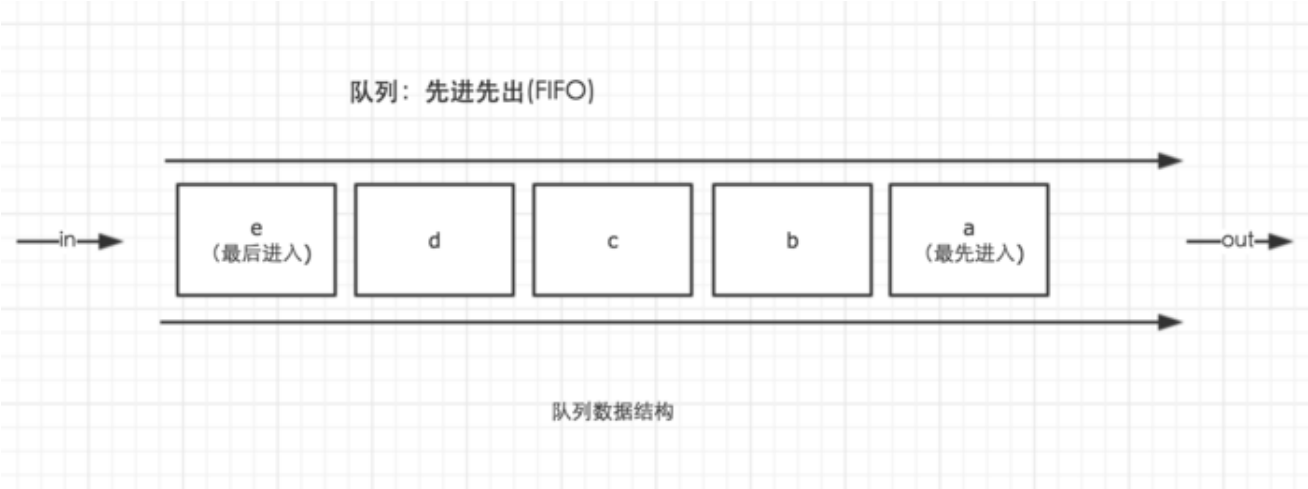
栈空间遵循**先进先出，后进后出**的规则。

堆数据结构

堆数据结构是一种树状结构。它的存取数据的方式，则与书架与书非常相似。书虽然也整齐的存放在书架上，但是我们只要知道书的名字，我们就可以很方便的取出我们想要的书。好比在`JSON`格式的数据中，我们存储的`key-value`是可以无序的，因为顺序的不同并不影响我们的使用，我们只需要关心书的名字。

队列

队列是一种先进先出（`FIFO`）的数据结构。正如排队过安检一样，排在队伍前面的人一定是最先过检的人。用以下的图示可以清楚的理解队列的原理。



队列数据结构

变量对象与基础数据类型

`JavaScript`的**执行上下文**生成之后，会创建一个叫做**变量对象**的特殊对象，`JavaScript`的基础数据类型往往都会保存在变量对象中。

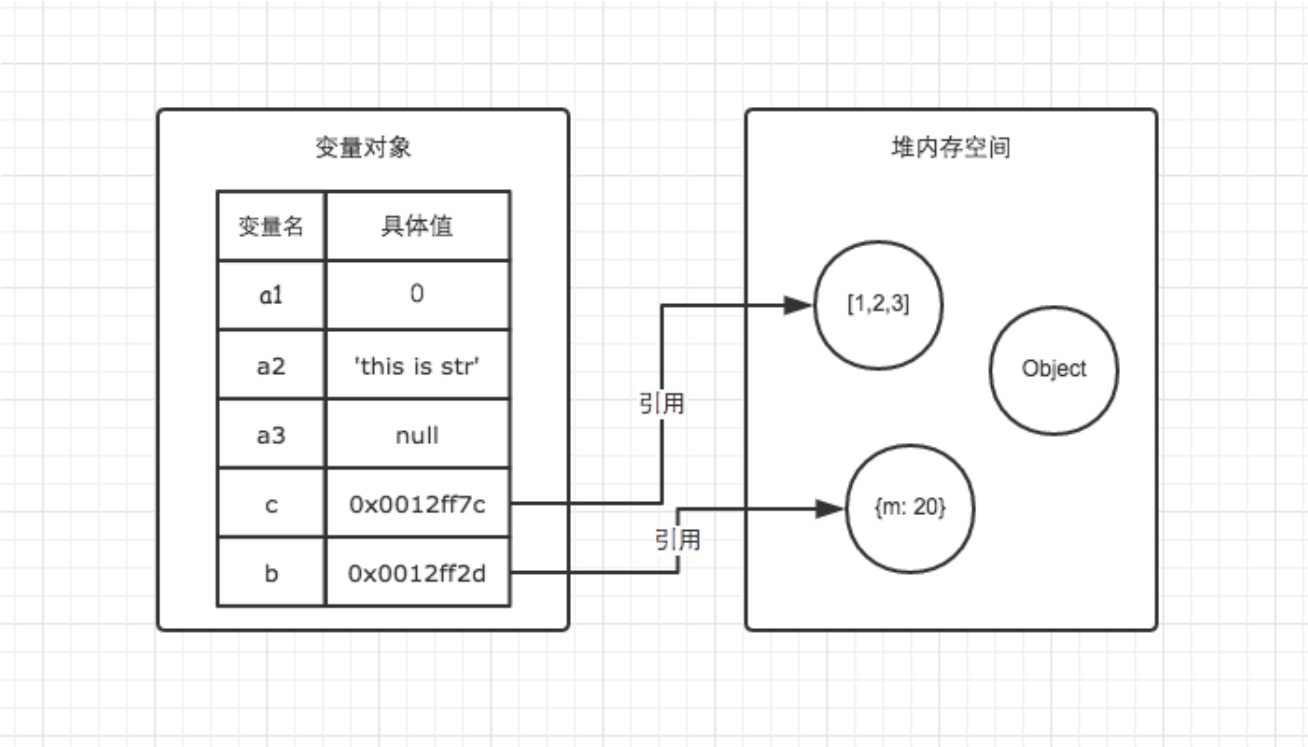
> 严格意义上来说，变量对象也是存放于堆内存中，但是由于变量对象的特殊职能，我们在理解时仍然需要将其于堆内存区分开来。

基础数据类型都是一些简单的数据段，`JavaScript`中有5中基础数据类型，分别是`Undefined、Null、Boolean、Number、String`。基础数据类型都是**按值访问**，因为我们可以**直接操作保存在变量中的实际的值。**（`ES6`中新加了一种基础数据类型`Symbol`，可以先不用考虑他 ）

引用数据类型与堆内存

引用数据类型的值是保存在堆内存中的对象。`JavaScript`不允许直接访问堆内存中的位置，因此我们不能直接操作对象的堆内存空间。在操作对象时，实际上是在操作对象的引用而不是实际的对象。因此，引用类型的值都是按引用访问的。这里的引用，我们可以理解为保存在变量对象中的一个地址，该地址与堆内存的实际值相关联。

```
var a1 = 0;   // 变量对象
var a2 = 'this is string'; // 变量对象
var a3 = null; // 变量对象
var b = { m: 20 }; // 变量b存在于变量对象中，{m: 20} 作为对象存在于堆内存中
var c = [1, 2, 3]; // 变量c存在于变量对象中，[1, 2, 3] 作为对象存在于堆内存中
```

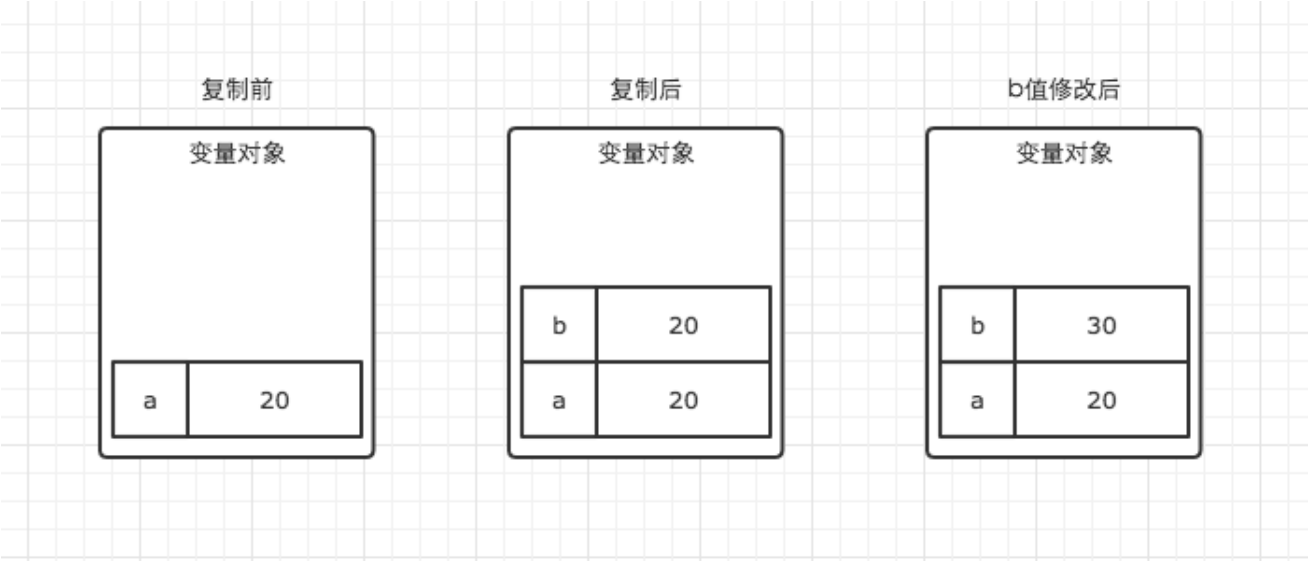


当我们要访问堆内存中的引用数据类型时，实际上我们首先是从变量对象中获取了该对象的地址引用（或者地址指针），然后再从堆内存中取得我们需要的数据。

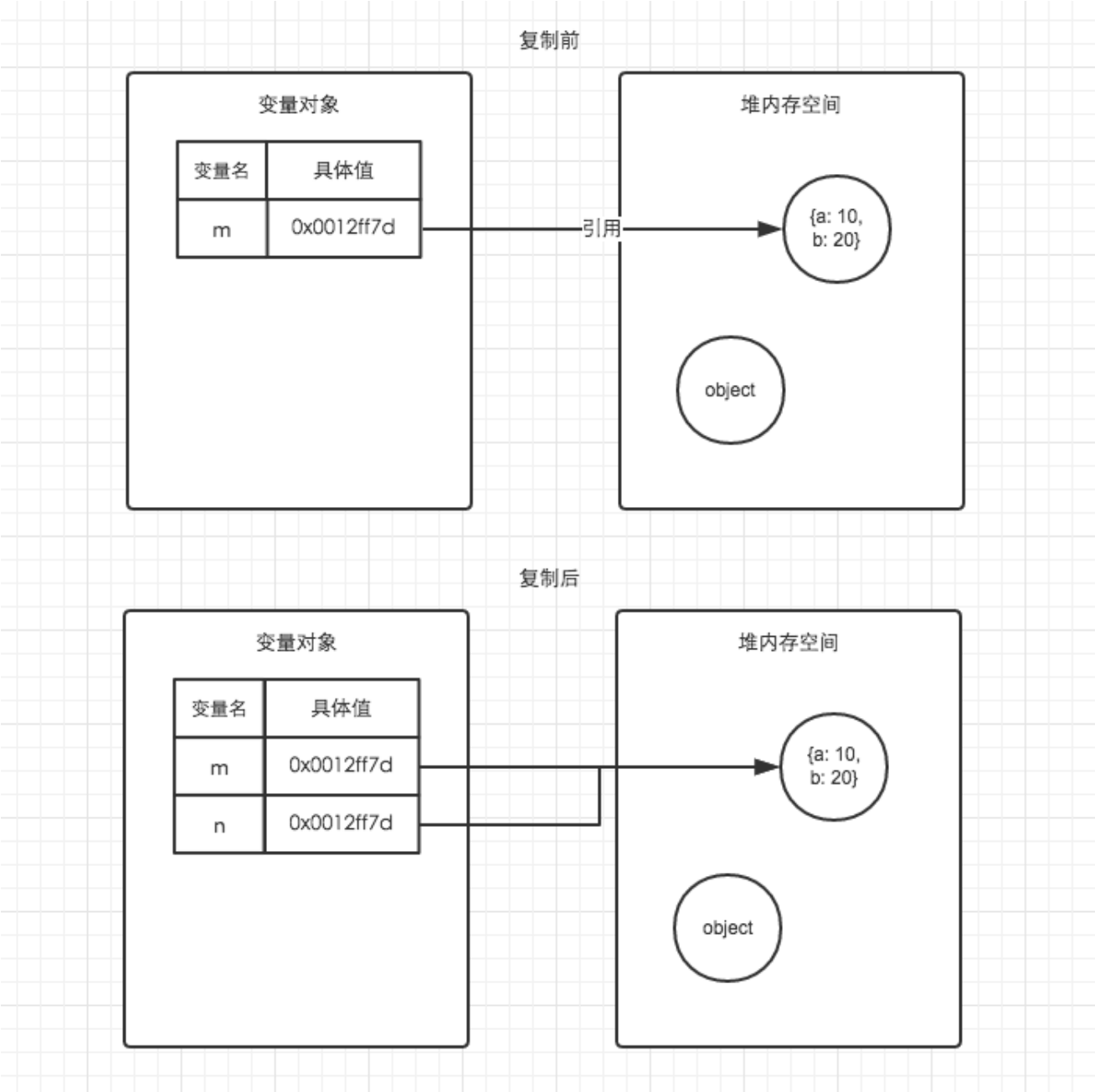
```
var a = 20;
var b = a;
b = 30;
//demo1 这时a的值是多少？
```

```
var m = { a: 10, b: 20 }
var n = m;
n.a = 15;
//demo2 这时m.a的值是多少
```

变量对象中的数据发生复制行为时，系统会自动为新的变量分配一个新值。`var b = a`执行之后，`a`与`b`虽然值都等于`20`，但是他们其实已经是相互独立互不影响的值了。 所以我们修改了`b`的值以后，`a`的值并不会发生变化。



在`demo2`中，我们通过`var n = m`执行一次复制引用类型的操作。引用类型的复制同样也会为新的变量自动分配一个新的值保存在变量对象中，但不同的是，这个新的值，仅仅只是引用类型的一个地址指针。当地址指针相同时，尽管他们相互独立，但是在变量对象中访问到的具体对象实际上是同一个。因此当我改变n时，m也发生了变化。这就是引用类型的特性。



JavaScript的内存生命周期

- 分配你所需要的内存
- 使用分配到的内存（读、写）
- 不需要时将其释放、归还

```
var a = 20;  // 在内存中给数值变量分配空间
console.log(a + 100);  // 使用内存
a = null; // 使用完毕之后，释放内存空间
```

在`JavaScript`中，最常用的是通过**标记清除**的算法来找到哪些对象是不再继续使用的，因此`a = null`其实仅仅只是做了一个释放引用的操作，让`a`原本对应的值失去引用，脱离执行环境，这个值会在下一次垃圾收集器执行操作时被找到并释放。而在适当的时候解除引用，是为页面获得更好性能的一个重要方式。

在局部作用域中，当函数执行完毕，局部变量也就没有存在的必要了，因此垃圾收集器很容易做出判断并回收。但是全局变量什么时候需要自动释放内存空间则很难判断，因此在我们的开发中，需要尽量避免使用全局变量。