你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的f是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

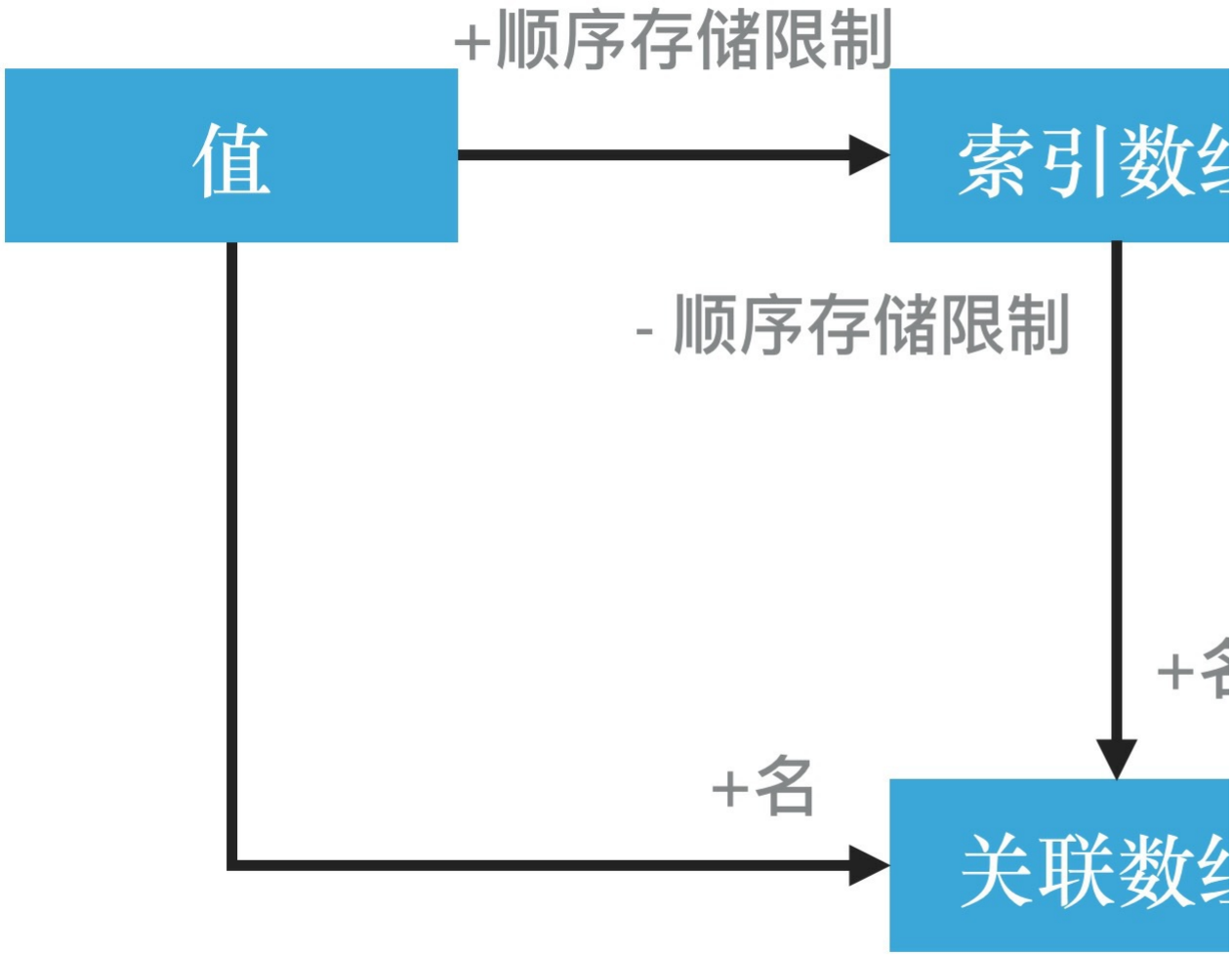
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的`x,y,z`这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}
```

```
// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}

// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标 _i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数 _f()_ 的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

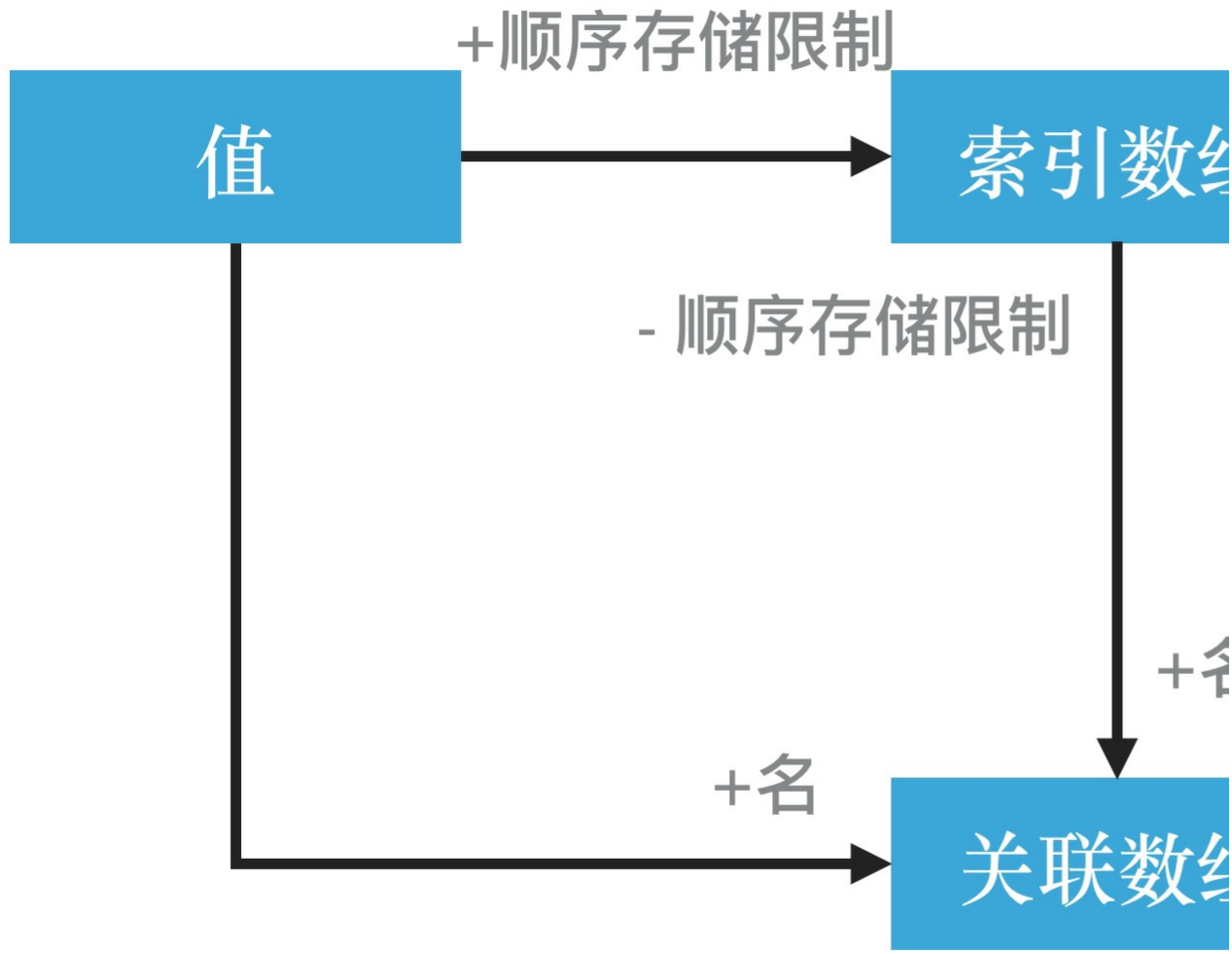
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
```

```
var [a, b] = {a, b}

// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

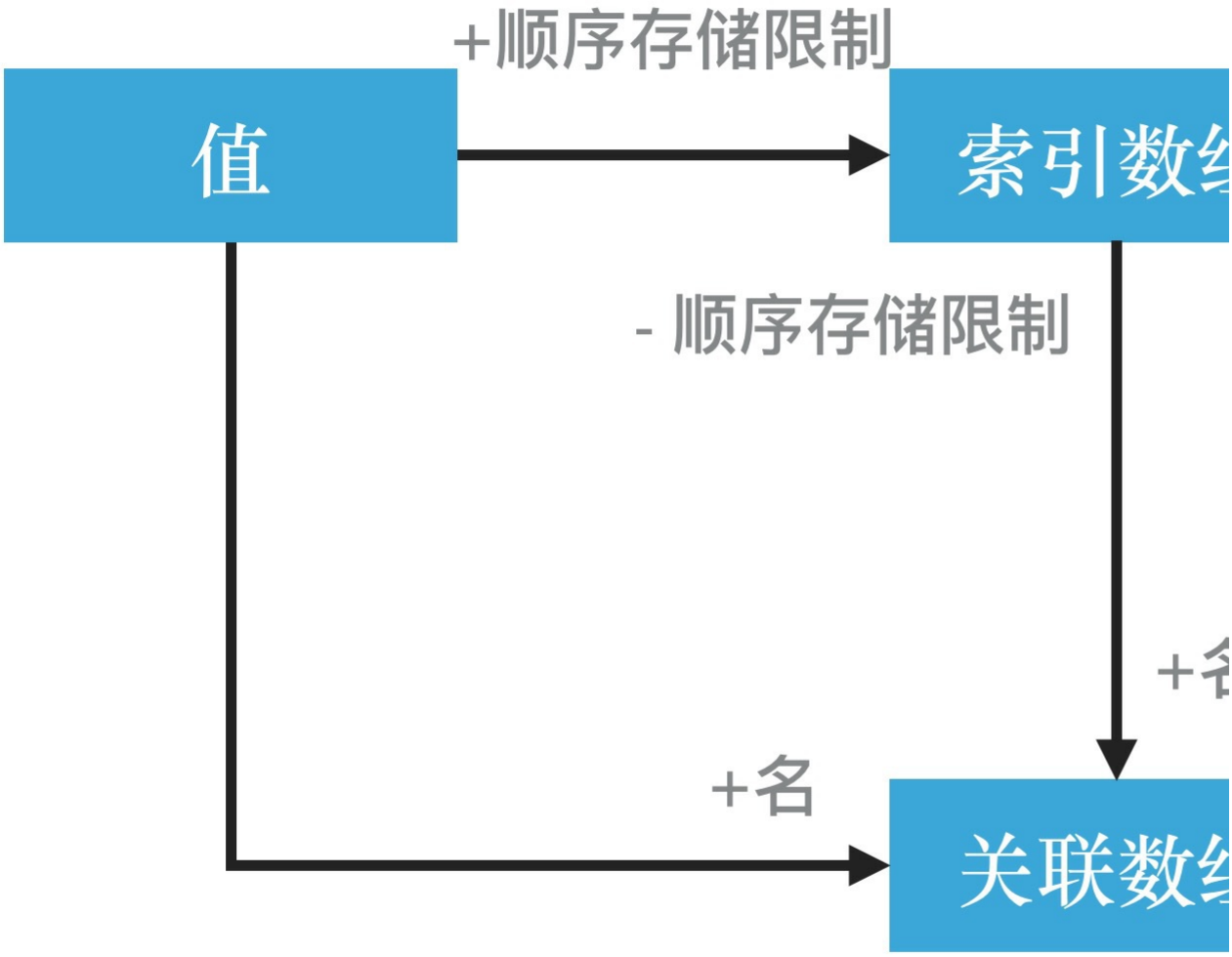
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

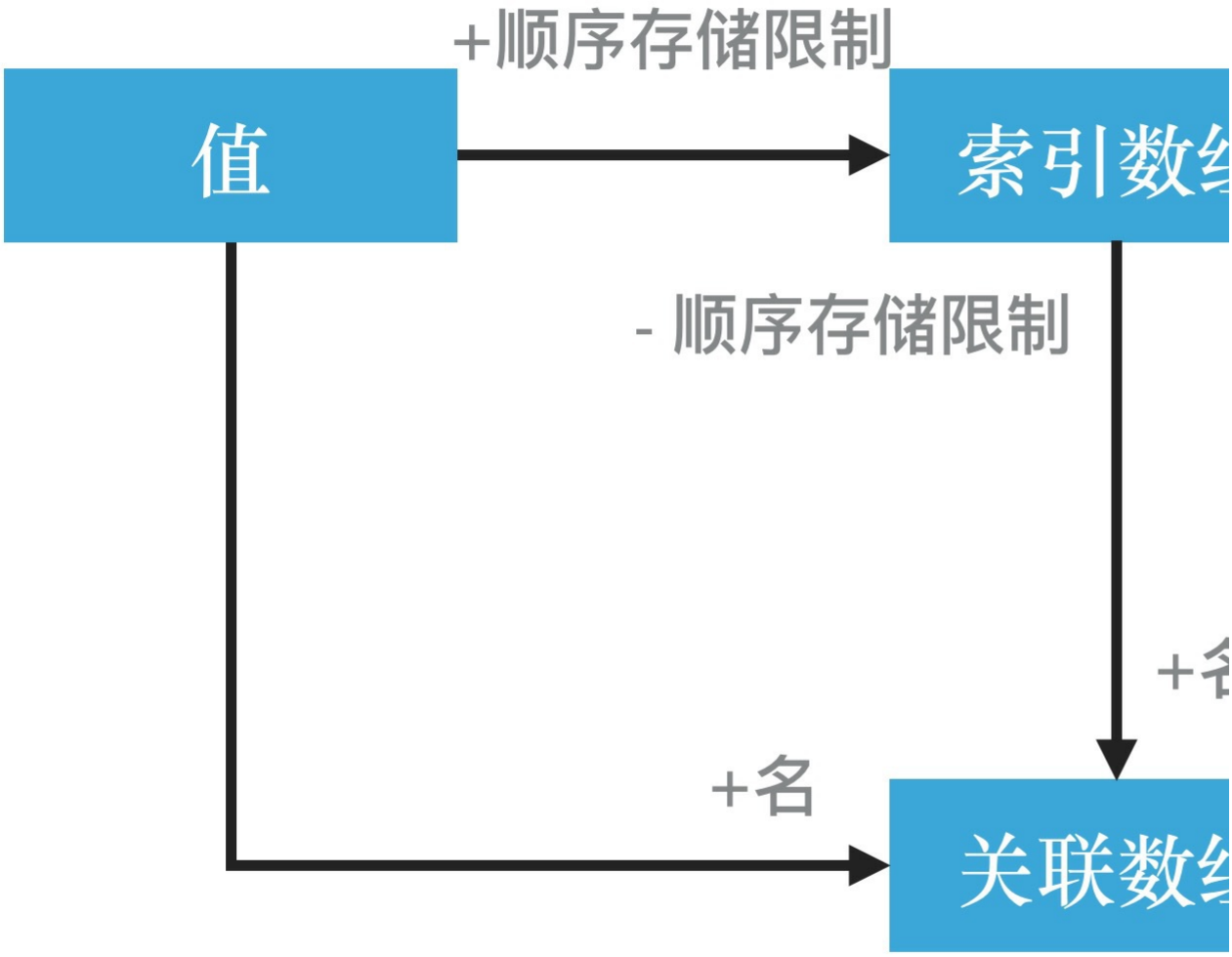
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

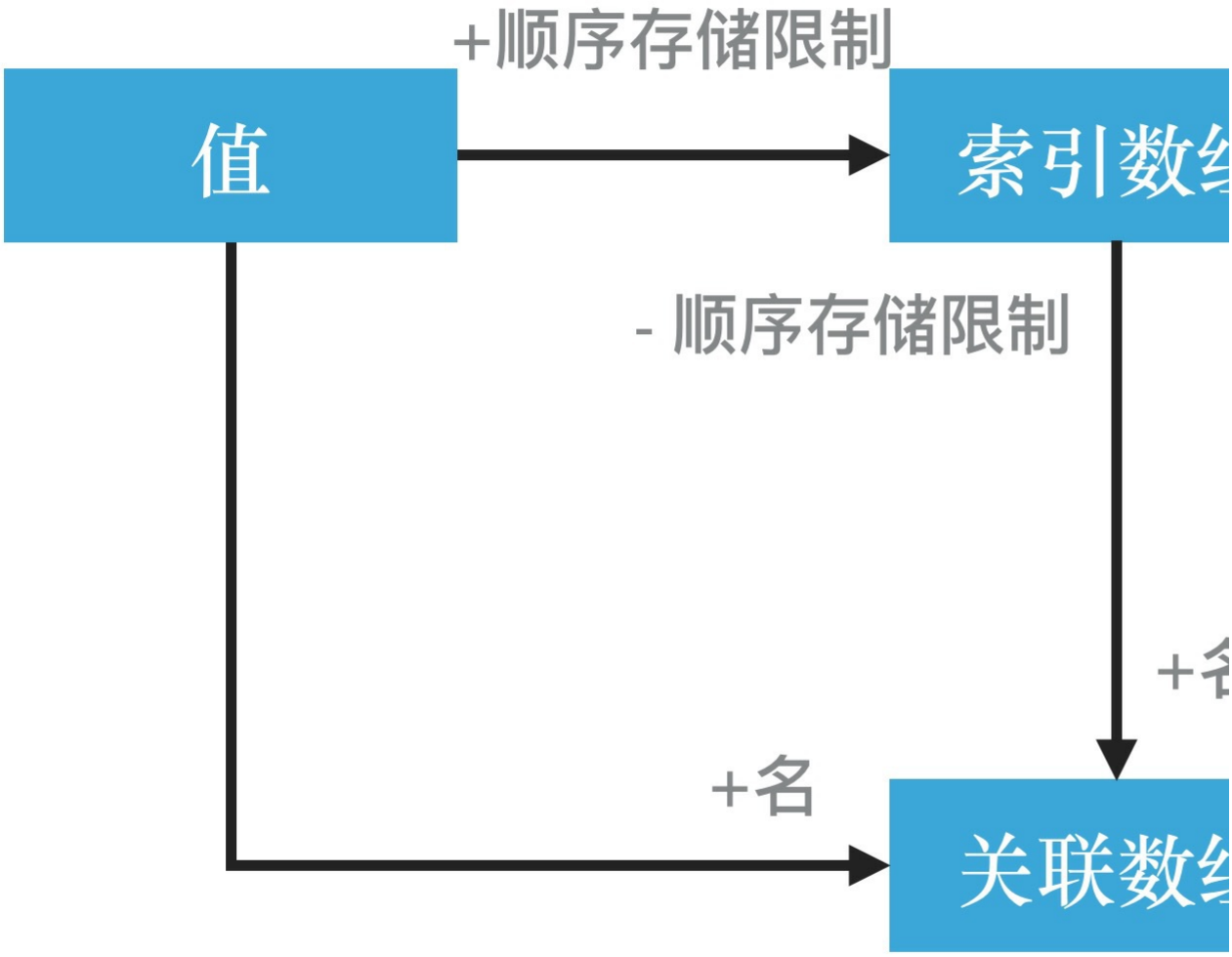
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的`x,y,z`这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

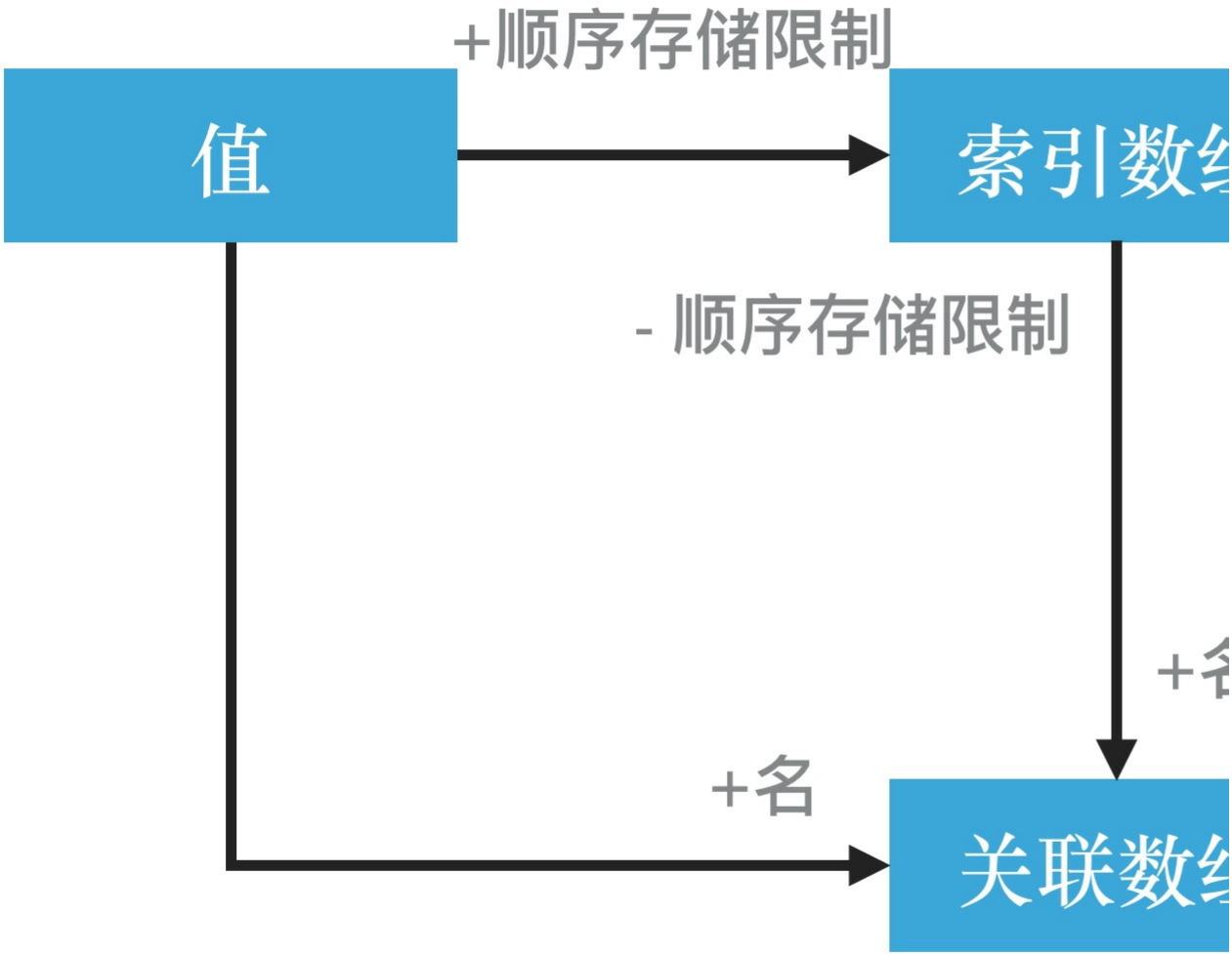
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的f是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

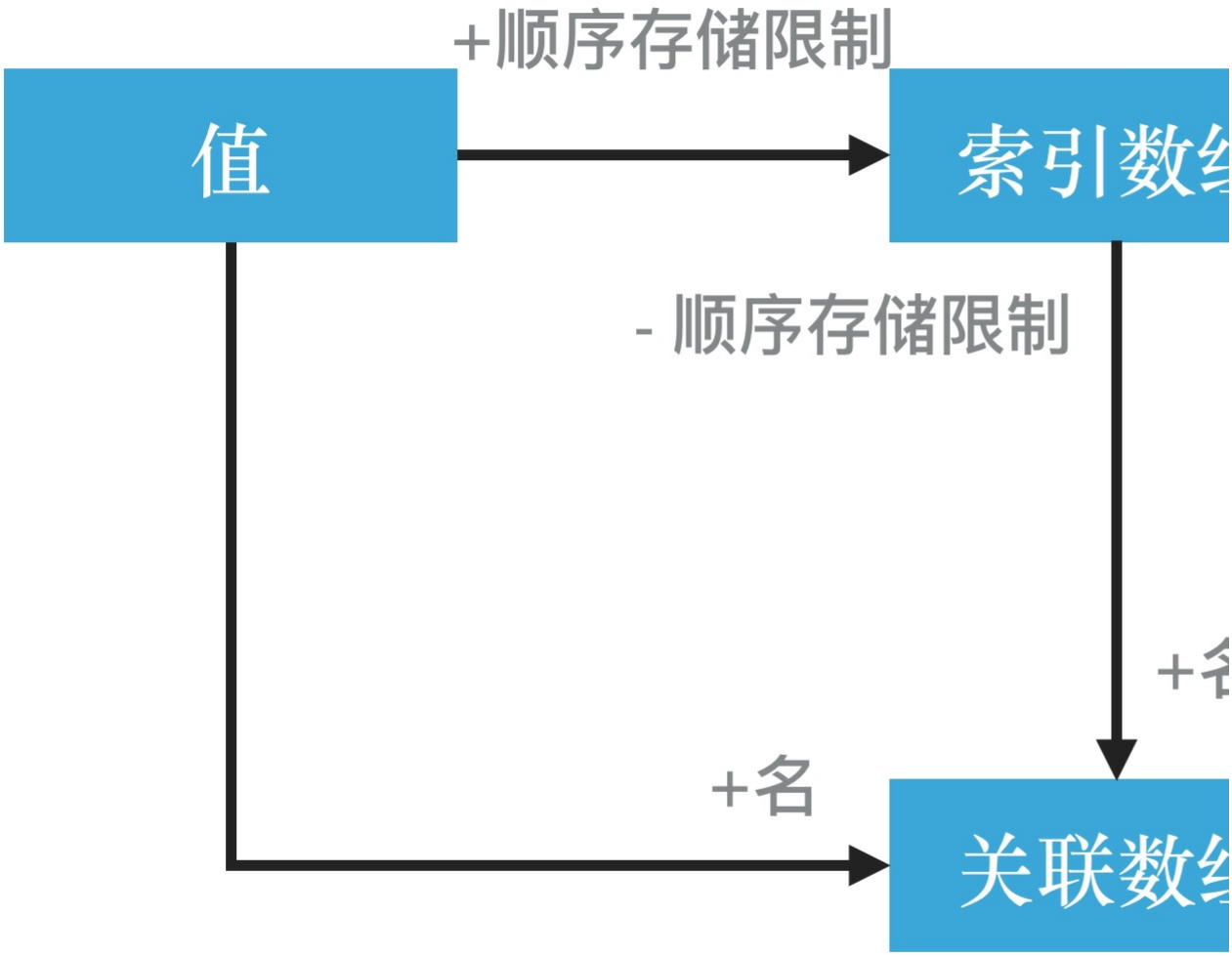
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

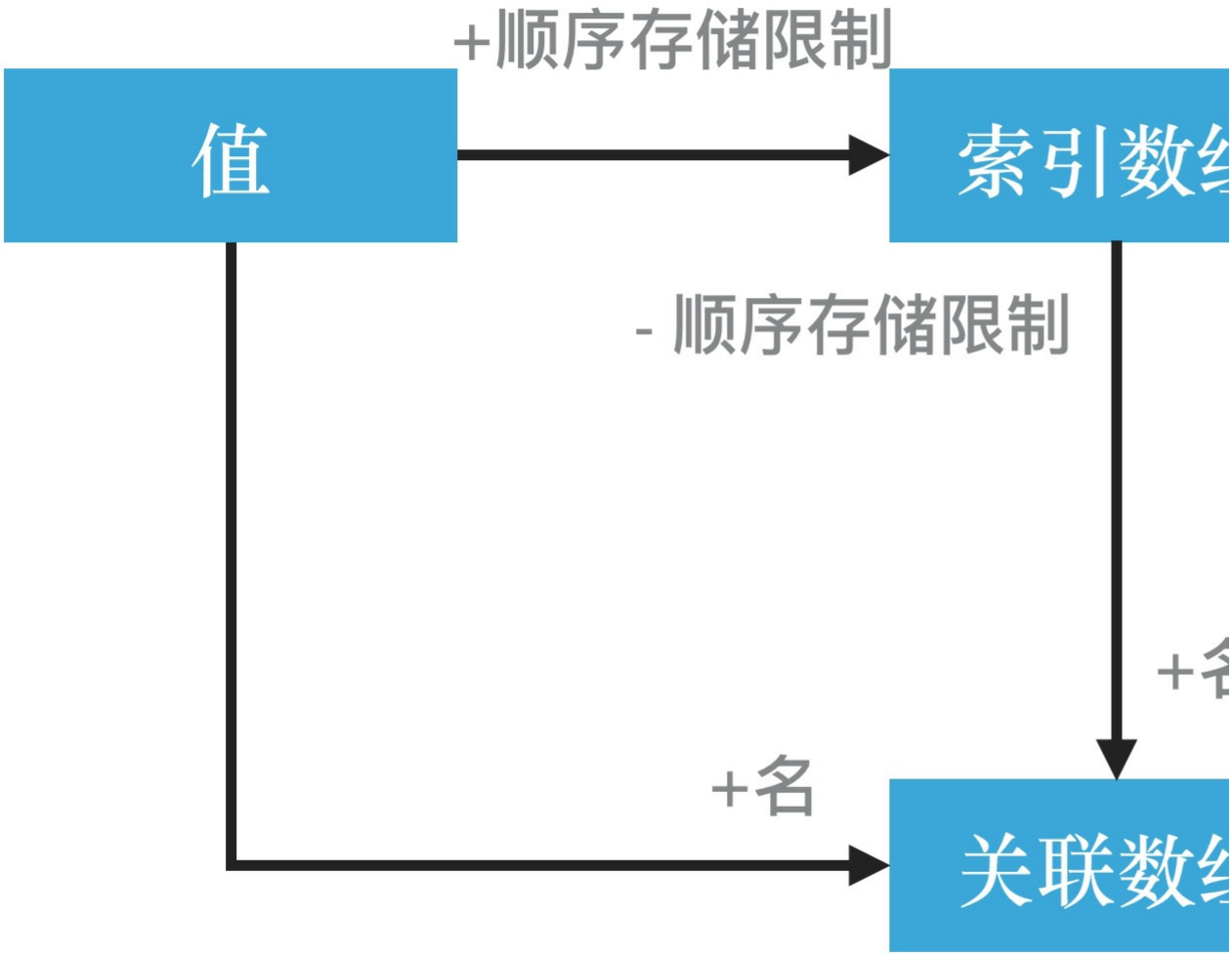
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的f是find的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

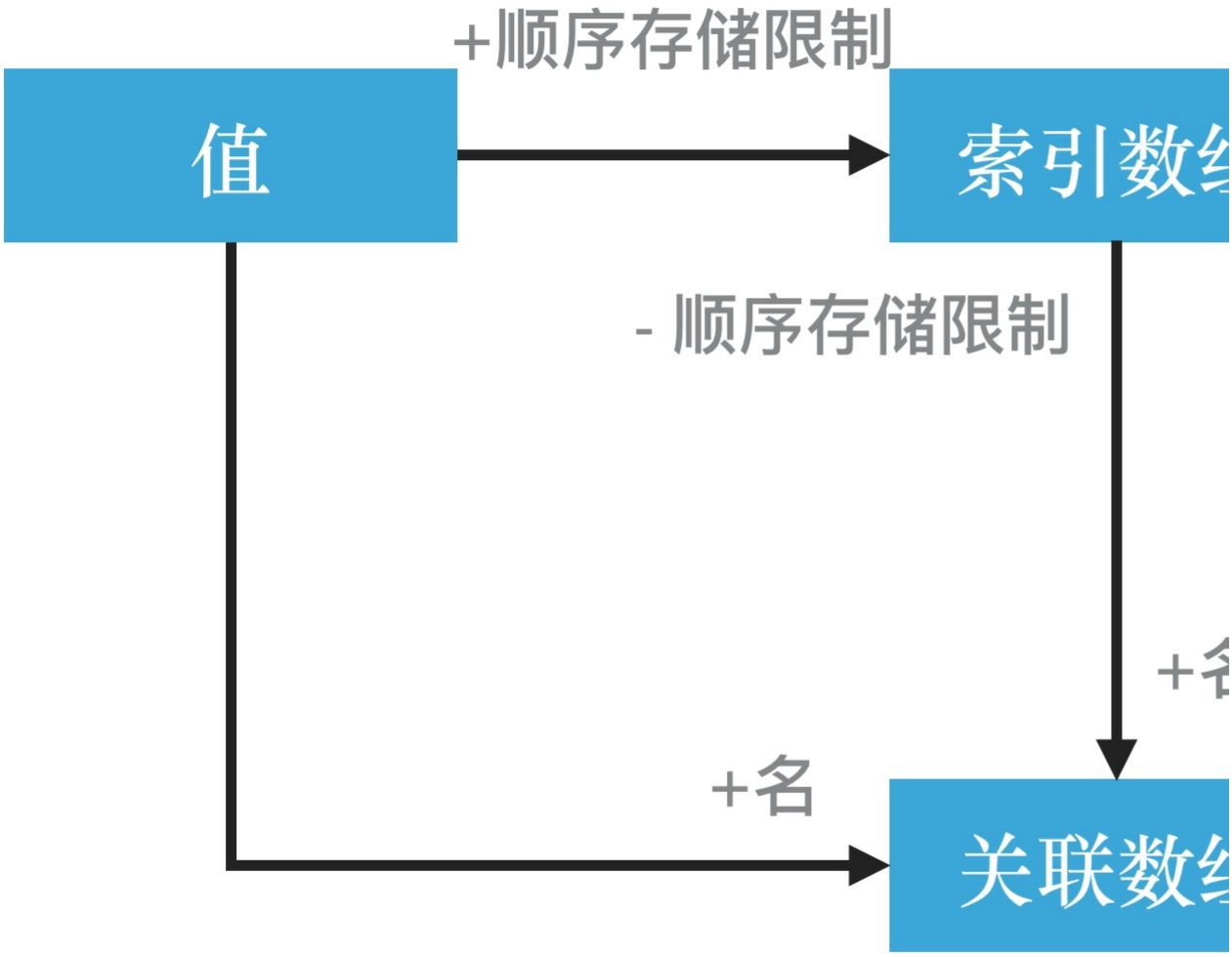
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

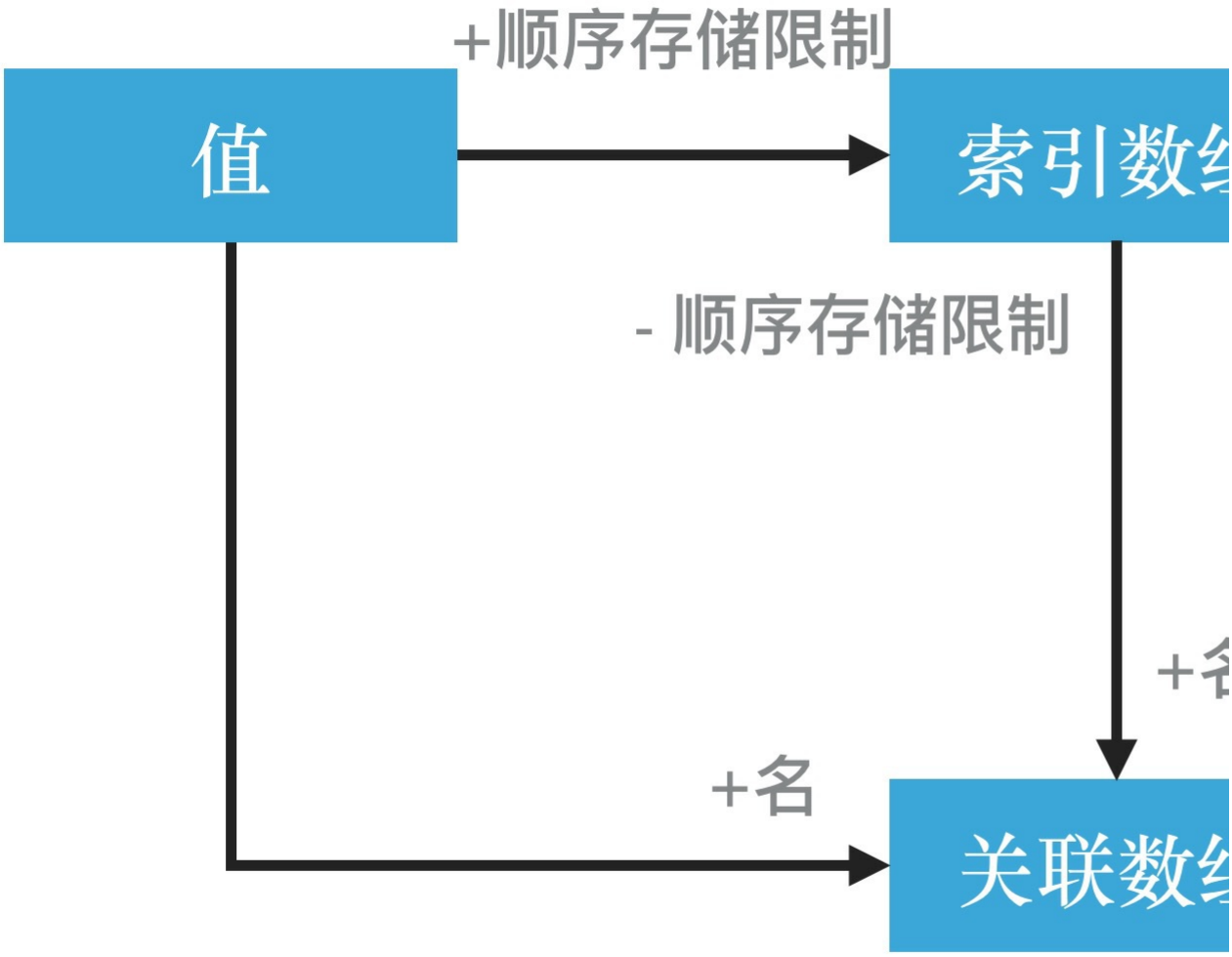
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

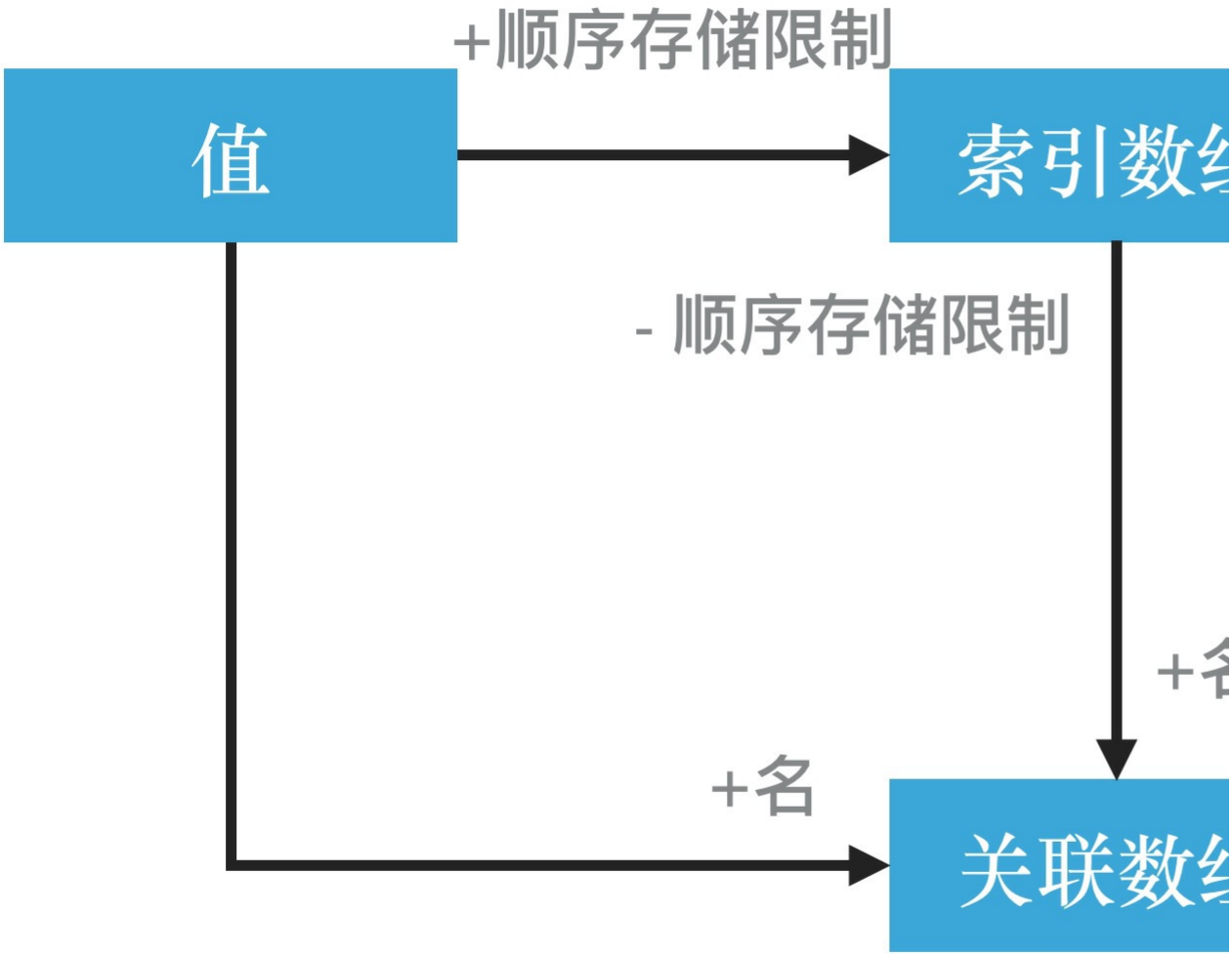
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

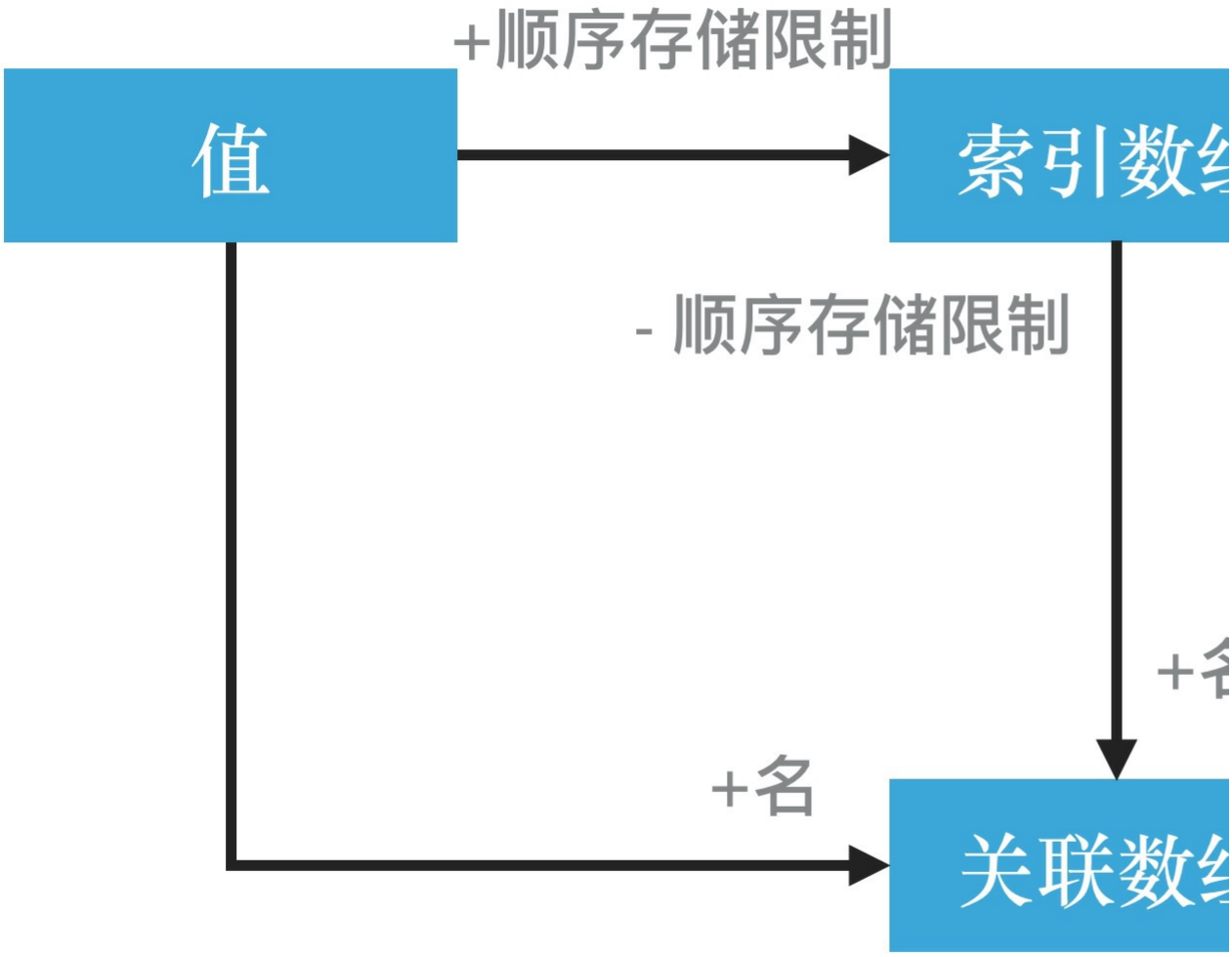
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

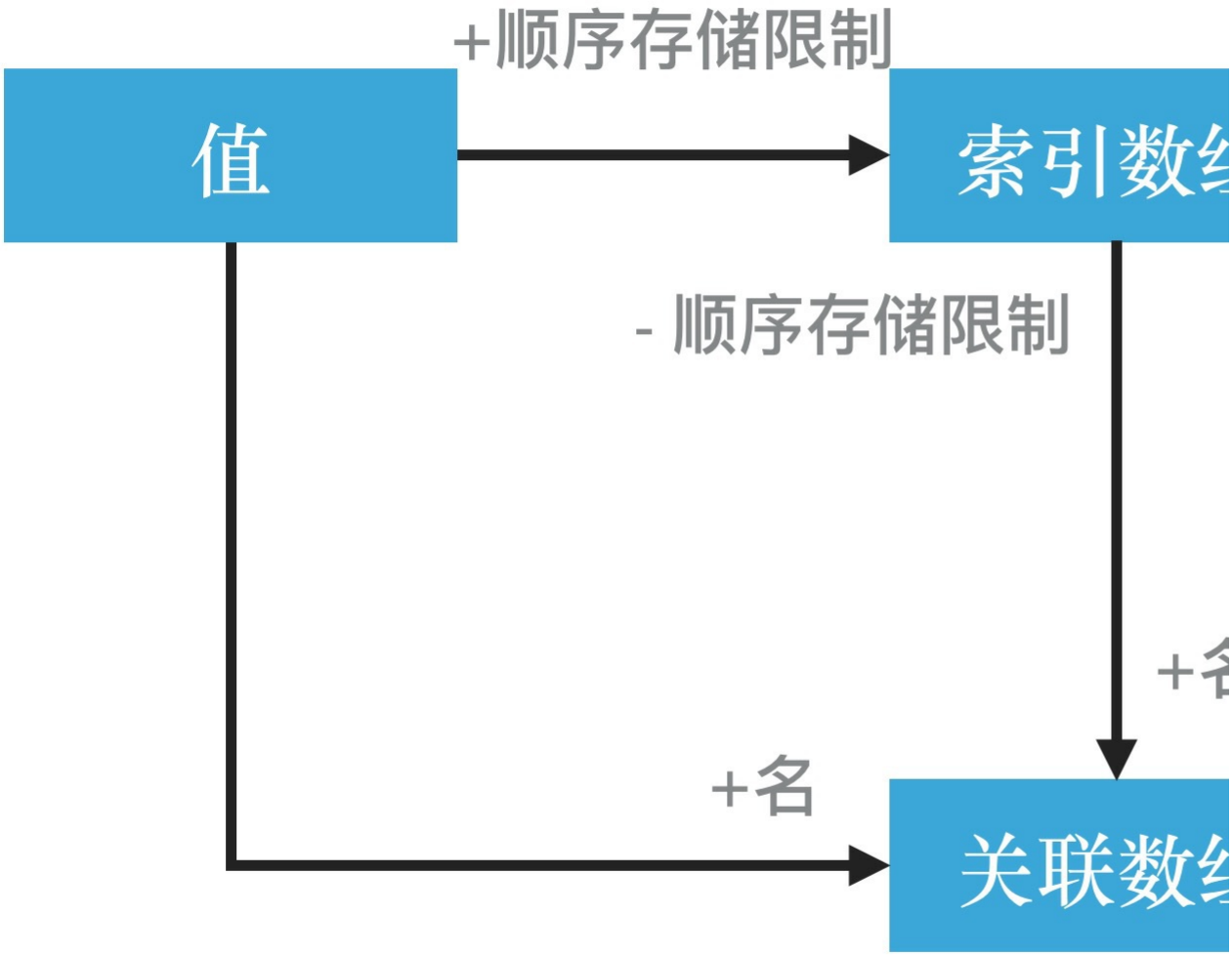
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的f是find的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

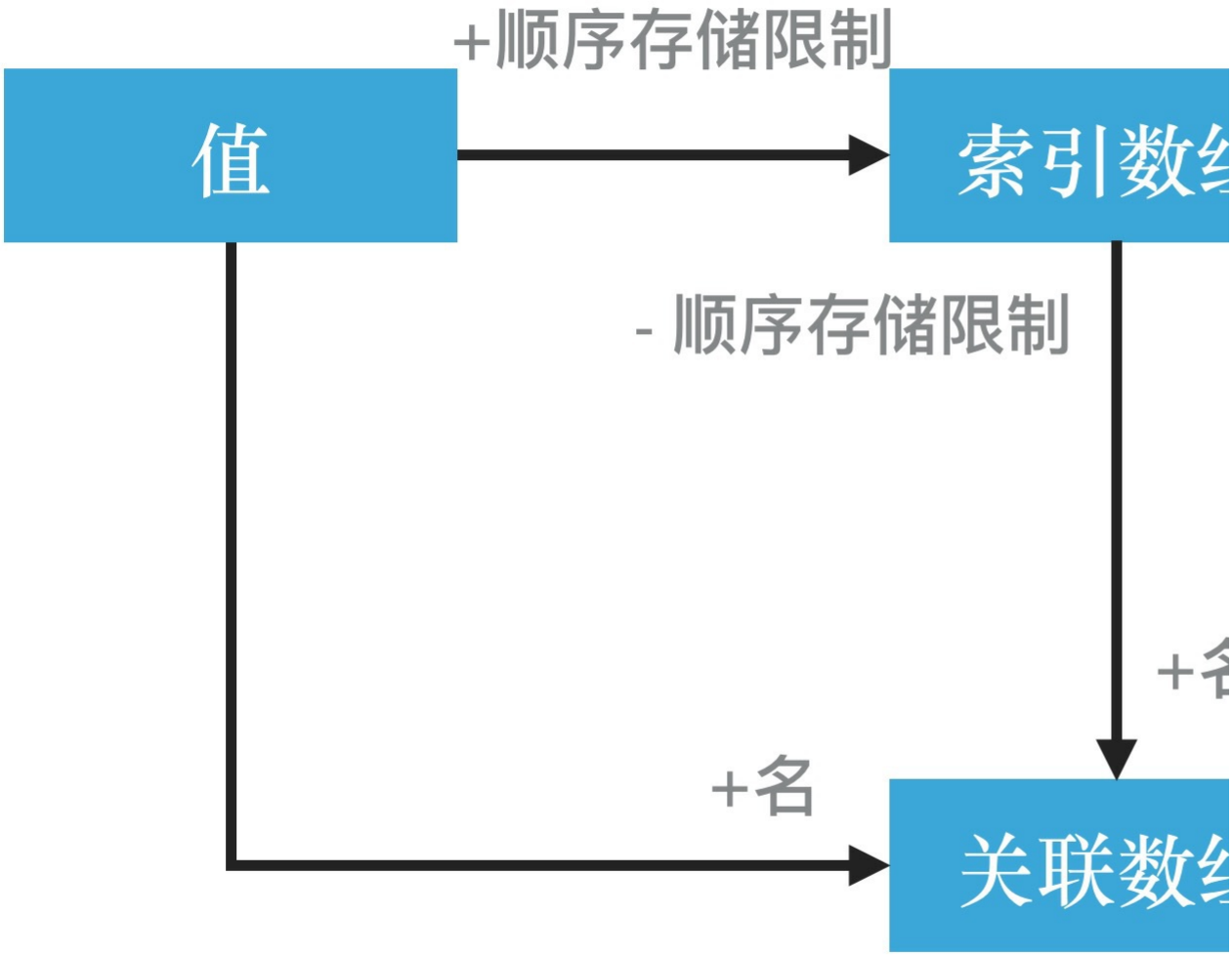
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

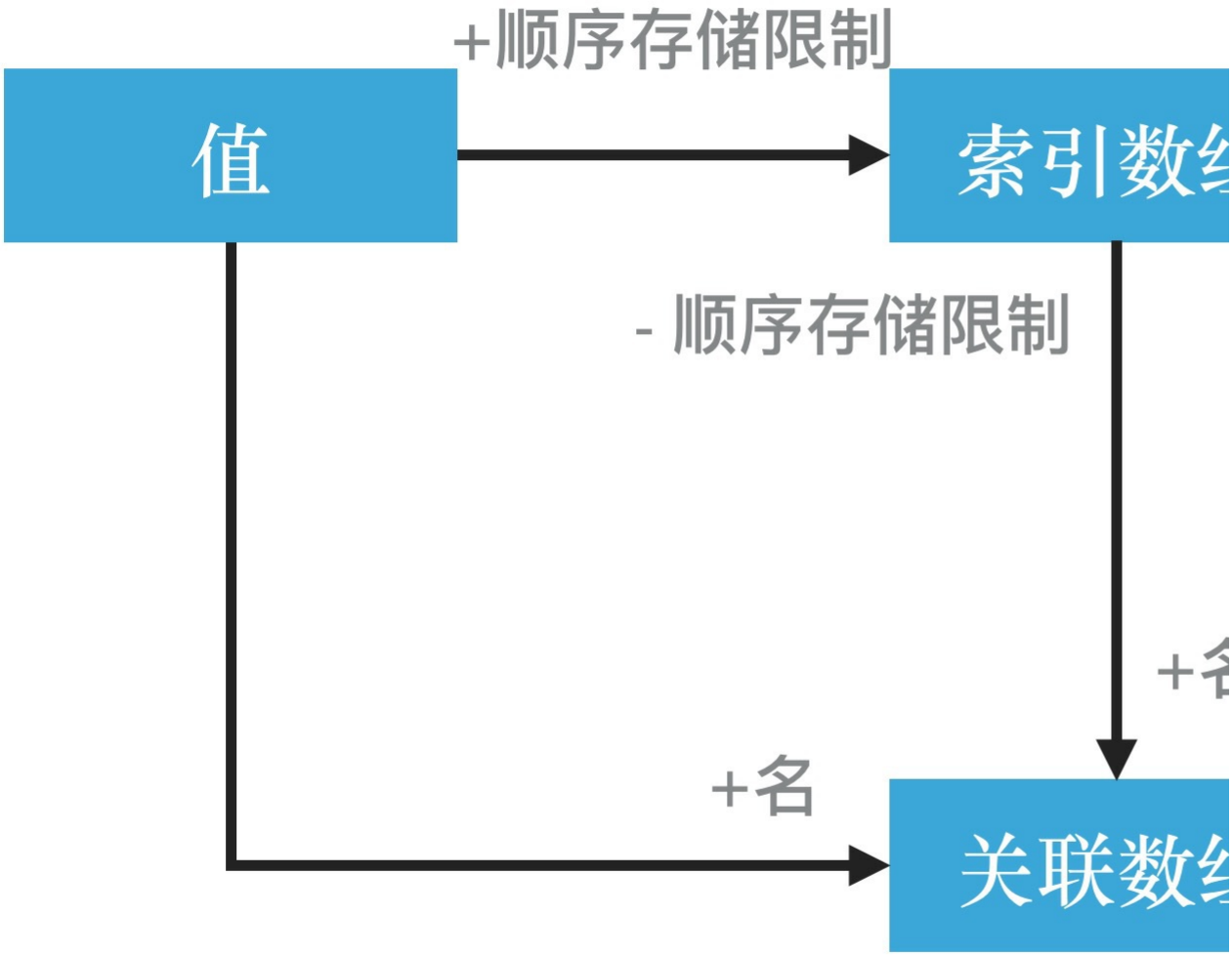
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。

你好，我是周爱民。欢迎回到我的专栏。

接下来的两讲，我要讲的仍然是JavaScript中的面向对象。有所不同的是，今天这一讲说的是JavaScript中的对象本质，而下一讲要说的，则是它最原始的形态（也通常称为原子对象）。

说回今天的话题，所谓的“对象本质”，就是从根本上来问，对象到底是什么？

对象的前生后世

要知道，面向对象技术并不是与生俱来、顺理成章就成为了占有率最高的编程技术的。

在早期，面向对象技术其实并不太受待见，因为它的抽象层级比较高，也就意味着它离具体的机器编程比较远，没有哪种硬件编程技术（在当时）是需要所谓的面向对象的。最核心的那部分编程逻辑通常就是写寄存器、响应中断，或者是发送指令。这些行为都是面向机器逻辑的，与什么面向对象之类的都无关。

最早，大概是1967年的时候，艾伦（Alan Kay）提出了这么一个称为“对象”的抽象概念和基于它的面向对象编程（object-oriented programming），这也成为他所发明的Smalltalk这个语言中的核心概念之一。

然而，回顾这段历史，这个所谓的“对象”的抽象概念中，只包含了**数据**和**行为**两个部分，分别称为**状态保存**和**消息发送**，再进一步地说，也就是我们今天讲的“**属性**”和“**方法**”。并且，在这个基础上，有了这些状态（或称为数据）的局部保存、保护和隐藏等概念，也就是我们现在说的**对象成员的可见性问题**。

你看，这里没有**继承**，也没有**多态**。历史中，最早出现的所谓**对象**，其实只是对数据的封装！

所以你会看到最近十余年来，无数的业界大师、众多的语言流派对所谓的“继承”，以及与此相关的“多态”特性发起非难。追根溯源，就在于这两个概念并非是“面向对象”思想的必然产物，因而它们的存在将有可能增加系统抽象的复杂性。

具体到你所了解的JavaScript，一些新的面向对象特性也总会在ECMAScript规范的草案阶段碰壁。

例如，近两年来最受非议的“Class Fields”提案，在添加了“**私有字段**”这个概念之后，却将“**保护属性**”这个皮球扔给了远未成熟的注解提案。究其原因呢，则是“**字段**”与“**继承性**”之间存在概念和实现模型的冲突。

这也不枉我常常说tc39中存在着大量的“OOP敌视者”，尽管是玩笑，但也确实反映了“面向对象编程思想”在这门语言中恶劣的生存状态。

然而并不仅仅如此。最近这些年的新语言，除了使用类似“**字段**”“**记录**”这样的抽象概念来驱逐面向对象之外，还对**函数式编程**洞开怀抱。在我看来，这既是流行的趋势，也确实是计算机编程语言进化的必然方向。但是，这也带来了更深层面的问题，使得**面向对象**的生存环境进一步恶化。

为什么呢？

你看，面向对象的**封装**、**继承**和**多态**三个核心概念中，多态有一部分是与继承性相关的，去掉继承性，多态就死了一半。而另一半，又被“接口（Interface）”这个概念给干掉了。于是，整个OOP的体系中就只剩下“封装”还算在概念上能独善其身。这也与上面说到的艾伦有关，毕竟他提出的“面向对象”的最初意图也就在于提高封装性。

然而，一旦引入“函数式编程”，情况就发生了变化。

函数式语言根本不考虑数据封装问题，逻辑之间的数据是由函数界面（也就是函数参数）来传递的，而函数自身又强调“无副作用”，也就意味着它不影响函数之外的数据——那函数外也就没有任何数据封装（例如隐蔽）的要求了。

所以，简单地说，函数式一出，面向对象的最后一根稻草——“封装”特性也就扑街了！

你看看，面向对象到底怎么了？混了半个世纪了，最终落下个谁谁都嫌弃、人人都喊打的局面，连个打根儿上起就存在的核心抽象概念，都被人家掘断了气儿。

讲到这，你是不是觉得我给你扯的太远了？其实不是的。

这一讲的标题是“x = y”这样一个赋值表达式，而赋值表达式右边的“y”，正是这样的一个“对象”。我与你说了半天的这些所谓“三个核心概念”，在这一行代码中，被瓦解掉了2/3，剩下的，正是最最原始的东西：

- 所谓对象，是对数据的封装；
- 所谓解构，就是从封装的对象中，抽取数据。

你看，聊了半天，我又圆回来了吧：对象，其实是一个数据结构；解构赋值，就是将这个结构解构了，拿去赋值。

要紧的地方在于：对象，是怎样的一个数据结构呢？

两种数据结构

其实所谓的“某某编程思想”，本质上就是在说两个东西：一个，是在编程中怎么管理数据，另一个则是怎么组织逻辑。

而结构化，又或者说具体到“数据结构”，无非是在说将系统中的数据用统一的、确切的、有限的数据样式给管理起来。这些样式，小到一个位（bit）、一个字节（byte），大到一个库（Database）、一个节点（Node），都是对数据加以规划的结果。编程的思想，在机器指令的编码与数据集群的管理里面，都是如出一辙的。在所有的这些思想的背后，都有一个核心的问题：

- 如何抽象“一堆”的数据，使得它们能被方便和有效地管理。

在我们的单机系统，或者说像JavaScript这类应用环境的编程语言中，这些数据是假设被放在“有限的存储空间里面”的。这个假设模拟了内存和指令带宽的基本性能。

那么，在这样有限的存储空间里面如何存储数据呢？又或者说，如何得到一个“最高的抽象层级的数据结构”，以便于通过编程语言来处理操作呢？

一个数据结构的抽象层次越是低级，那么对它的编程就越是复杂。例如说，如果你需要面向“位（bit）”来编程，那么差不多就需要写机器指令，或者手工去搬动逻辑电路的开关了。

所谓“最高的抽象层级”，在一个“有限的存储空间”里面，其实只能表达为一个“块”。简单地说，你只能称呼“一堆数据”为“一堆数据”，因为当你不了解它们的具体性质时，你只能这样称呼它。而“块”其实是对“有限空间”的边界分解，设定了“有限空间”，那么对应的，也就出来了“块”这个概念。

而由此带来的问题是：在一个有限空间中，如何找到一个“块”？

如果从这些“块”的相关位置出发，以位置关系来看，就只有两个解：

1. 为所有**连续的**块添加一个**连续的**“索引”；
2. 为所有**不连续**的块添加一个唯一的“名字”。

当然，关键点在于所谓的“连续”和“不连续”。“连续”“不连续”，在语义上就是二分的，所以也就只需要两个解。其中“索引”比较简单，它就对应于连续性本身，表达为可计算的特性是“a[*i*]”，也就是a的下标_i_。

而“名字”对应于“找到块”这一目的本身，表达为一个可计算的函数“*f()*”。你可以认为这里的`f`是`find`的简写。于是一旦系统认为一个函数“*f()*”可以用于找到它需要计算的数据，那么数据就可以理解为“b[*f()*]”，而其中的函数“*f()*”如何实现，则可以交给“另外的一个系统”去完成了。

那么，重要的是为什么不能将“i”也理解为“找到i”呢？

如果是这样，那么这个所谓的“索引”其实也可以作为名字啊？对的，如果这样来理解，那么也可以为上面的“a[*i*]”引入一个用于计算索引的函数f，只是该函数_f()_的唯一作用就是返回了“i”。也就是：

```
function f() {
  return i
}

a[i] === a[f()];
```

现在，我们看到了这两个数据结构——一种是“连续的块”，另一种是“不连续的块”，它们都存在一种统一的“找到块的模式”，也就是：通过一个函数来找到块。

进一步阐释的话，对于索引数组来说，这个函数是取数组成员的“索引”；对于关联数组来说，这个函数是取数组成员的“名字”。其中“关联数组”是用一对“名/值”来创建的数组，在实现中为了将无穷尽的“名字”收敛在一个有限范围内，通常是用值的HASH作为名字。

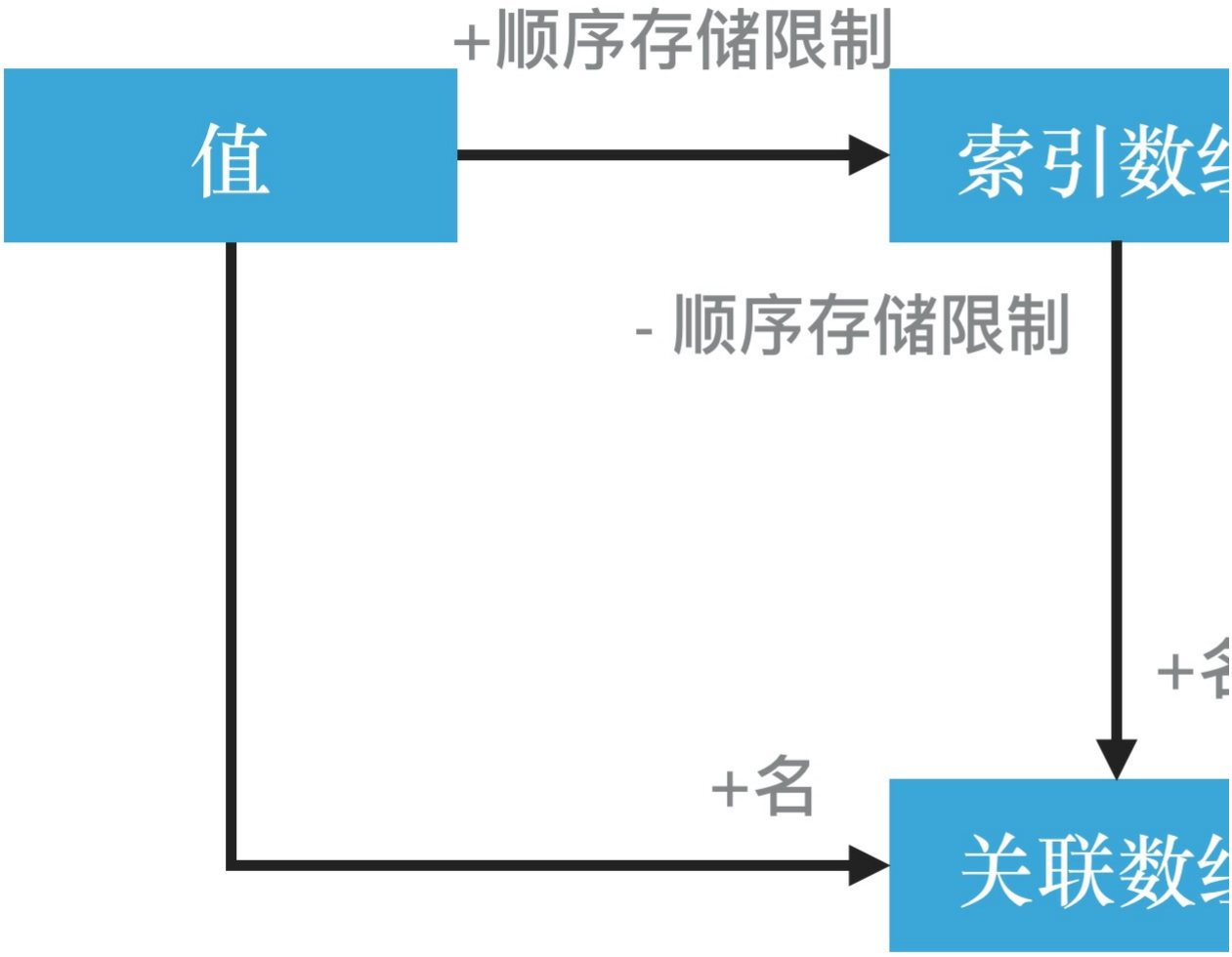
所以，在“怎么管理数据”这个问题上，你可以将所有数据看成只具有两种数据结构的构成，一种称为**索引数组**（对应于可索引的块），另一种称为**关联数组**（对应于不可索引的块）。而究其根本来说，索引数组其实是关联数组的一个特例——被存取的数据所关联的名字就是它的索引。

JavaScript中的“对象”，在本质上就是这样的一个关联数组。同时，所谓的“数组（Array）”——也就是索引数组（Index array），正是作为关联数组的一个特例来实现的。这样一来，JavaScript就实现了两种数据结构的大统一：

1. 数组（Array class）是一种对象（Object class）；
2. 对象本质上是关联数组（Associative array）。

解构

所以，对象不过是“稍微复杂一点的数据结构”，相比起来，它并不比稍早一点出现的“记录/结构体”更复杂。从抽象的演进过程来说，对象只是“没有顺序存储限制，以及添加了成员名字的”结构体而已。



图引自：《程序原本》“10.1 抽象本质上的一致性”

在前面的文章里我就讲过，计算的本质是求“值”，因此几乎所有的引用类型呢，最终都会将“**与它的相关的运算结果**”指向“**值**”。至于这一切背后的原因，其实也很简单，就是物理的计算系统最终也只能接收“字节、位”等等这样的值类型数据。但是在高级语言中，或者应用编程中呢，程序员又需要高层级的抽象来简化编程，所以才会有**结构体**，以及我们在这里讲到的**对象**。

还原这个过程，也就意味着“结构”是应用编程的必须，而“解构”是底层计算的必须。从一个“结构（这里是指数据结构，或者对象等复杂的结构）”中把那些值数据取出来，就称为解构。这一讲的代码标题，就是这样的一个“解构赋值”，它的目的呢，也正是“从一个结构中提取值”。你仔细看这行代码：

```
[a, b] = {a, b}
```

等号右侧是一个对象的字面量，它的语义是将a、b两个数据变成“对象”这个数据结构中的两个成员。其中，由于a、b都是既已约定的名字，所以在作为对象成员的时候，“名字+值”就都已经具备了，完全符合“关联数组（或名/值数据对）”的语义要求。

而再看它的左侧，是一个数组？不是的，这称为一个“（数组）赋值模板”。

所谓赋值模板，不过是“变量名字”和“它的值”之间的位置关系的一个“说明”，这个说明是描述型的、声明风格的。因此它事实上在JavaScript语法解析阶段就完成了处理，根本不会“产生”任何运行期的执行过程。

所以左侧的“赋值模板”只是说明了一堆被声明的变量，也就是说，它们跟代码`var x, y, z = 100`中的x,y,z这样的名字声明没有任何差异，在处理上也是一样的。但是，这些赋值模板中声明的变量，每一个都“绑定”了一段赋值过程。这样的“赋值过程”在之前讲**函数的非简单参数**时也讲过（参见第8讲），就是“初始器赋值”。在ECMAScript中，尽管它们调用的是相同的“赋值过程”，但这两者之间是有语义上的区别的。具体来说，就是：

- 当赋值模板用作声明（var/let/const）时，上面的“赋值过程”将作为值绑定的初始器；
- 当该模板用作赋值运算的右操作数时，右操作数将作为“赋值过程”的传入参数。

因此，对于标题中的代码来说，存在三种在语义上并不相同的逻辑：

```
// 1. lhsKind is assignment, call DestructuringAssignmentEvaluation
[a, b] = {a, b}

// 2. lhsKind is varBinding, call BindingInitialization,
// and env will be current function scope.
var [a, b] = {a, b}
```

```
// 3. lhsKind is lexicalBinding, call BindingInitialization and current env
let [a, b] = {a, b}
```

当然，其结果都是一样的，也就是左侧的a和b都将被赋以左侧对象{a, b}所解构出来的“值”。但是，如果你运行标题中的代码，你会发现它“可能”与你的预期并不一样。例如左侧的a和b与原来有的变量“a、b”并不一样（假设这些变量是有的话）。

在上面的三个例子中，示例三的let/const赋值将不成立，因为右侧的对象将不能被创建出来。例如：

```
> let [a, b] = {a, b}
ReferenceError: a is not defined
```

但前两个示例在代码逻辑上是可以成立的，只是“一般来说”运行会抛出异常。例如：

```
# “赋值未声明变量”
> a = 100, b = 200;

# 示例代码（与使用var声明相同）
> [a, b] = {a, b};
TypeError: {(intermediate value)(intermediate value)} is not iterable
```

现在你可以思考一个小小的问题：

- **有什么办法可以让这个代码可以执行呢？**

这就回到今天这一讲的标题的核心话题了。

两种数据结构的统一

既然我已经说过，对象和数组在本质上都是存放“一堆数据”的结构，而差异只是查找的过程不同。那么，模拟它们不同的查找过程，也就可以在这些结构之间完成统一的“赋值行为”。

“数组赋值模板”其实是引用了数组的下标索引过程，ECMAScript将索引次序用专门的增序来管理，并将右操作数视作为“迭代器”来取值。注意，你确实需要留意这两者之间的区别，重点在于：“迭代器”的取值是序列的，但并没有确定使用数组的下标（例如序号）。

所以，只要让右侧的对象成为一个“可迭代对象”，那么赋值表达式就可以知道如何将它赋给左侧的模板了。这并不难：

```
## 模拟成数组的迭代器
> Object.prototype[Symbol.iterator] = function() {
    return Array.prototype[Symbol.iterator].call(Object.values(this));
};

## 测试
> a = 100, b = 200;

> [a, b] = {a, b}
...
```

当然，你也可以不借用数组的迭代器。这是一个更简单的版本：

```
Object.prototype[Symbol.iterator] = function*() {
    yield* Object.values(this);
};

...
```

也就是说，只需要将“对象成员”的列举，变成“对象成员的值”的列举，那么关联数组就可以用作索引数组了。当然，在代码中你也通常不需要这样写。只要写成下面这样就足够了：

```
> [a, b] = Object.values({a, b})
...
```

既然将对象赋给数组（赋值模板）是可行的，那么将数组赋给“对象（赋值模板）”又是否可行呢？答案当然是“可以”。不过仍然和上面的问题一样，你得有办法在模板中“描述”索引与名字之间的关系才行。例如：

```
# 在对象赋值模板中声明变量名与索引的关系
> ({0: x, 1: y} = [a, b])

> console.log(x, y);
100 200
```

如果你直接使用像标题一样的代码（并且将它们反过来的话），例如：

```
{a, b} = [a, b]
```

那么由于没有这种关系描述，所以右侧的数组被“强制地”作为一个对象来使用，因此变成了取a、b这两个成员的值。当然，它的结果就是不可预知的了。这种不可预知，来自于“将右侧数组作为对象”的并尝试取得具体的成员这样的行为，并且还受到它的原型对象的影响。

当然，也有使类似行为不受到原型影响的办法，这就是“人人都爱”的所谓“展开语法（Spread syntax）”。

关于展开语法的特点，我之前在第9讲中也已经讲过了，你可以复习一下那一讲的内容。展开语法与这一讲略有关联的事情是：“对象展开（Object spread）”，以及与它相关的“剩余参数（Rest paraments）”这两种东西，都将只处理那些“可列举的、自有的”属性。因此，展开过程并不受对象原型的影响。例如：

```
# 测试变量
> var a = 100, b = 200;

# 将数组展开到一个对象（的成员）
> obj = {...[a,b]}
{0: 100, 1: 200}

# 或，将对象展开到一个数组
> iterator = function*() { yield* Object.values(this) };
> obj[Symbol.iterator] = iterator;
> arr = [...obj]
[ 100, 200
```

知识回顾

这一讲的话题，重点在于从抽象层面认识对象与数组这两种东西，以及它们更为学术的名词概念：关联数组和索引数组。

由于索引数组本质上是关联数组的特例，所以在JavaScript中，用关联数组（也就是对象）来实现索引数组（也就是一般概念上的数组对象）是合理的，并且也是有着很深层面的理论根基的一个设计。

由于两种数据结构既相关、又相同，因此在它们之间相互转换的行为，其实就是一个名字和索引变换的游戏，这也是本讲中会再次讨论“展开语法”的原因：展开语法是在两种数据类型之间的一个桥梁。

当然，这一讲的标题尽管并不能直接运行，但“如何让它能运行”这个问题所涉及的知识，与我们计算机领域中较深层面的运行原理，以及较高层次的抽象结构之间，都存在着密不可分的关系。无论是出于理解JavaScript代码的目的，还是出于理解语言中最本质的那些假设或前设，我都非常建议你尝试一下这篇文章中的示例代码。

思考题

最后，作为一个小小的思考与练习，我希望你能够在学习完这一讲之后回答一个问题：

- “有迭代器的对象”在哪些场合中可以替代“索引数组”？

谢谢你的收听，希望你喜欢我的分享，也欢迎你把文章分享给你的朋友。