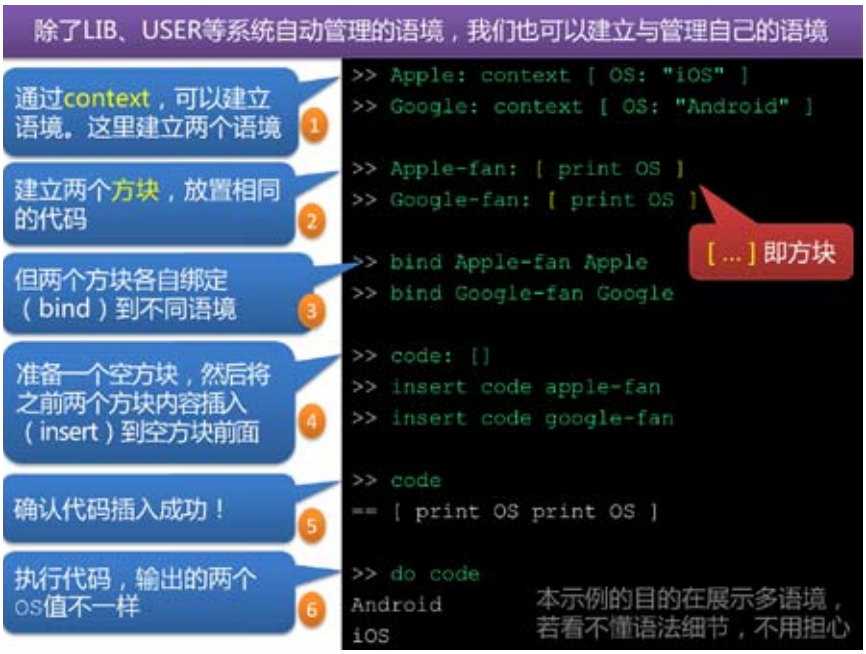
第7章

多语境的操作



根据前一章的说法，代码运行时会从 USER 语境中查询单字的定义，其实不完全是这样的。代码中的单字是可以从不同语境中查询定义的。为了证明这一点，这里再做一个实验。

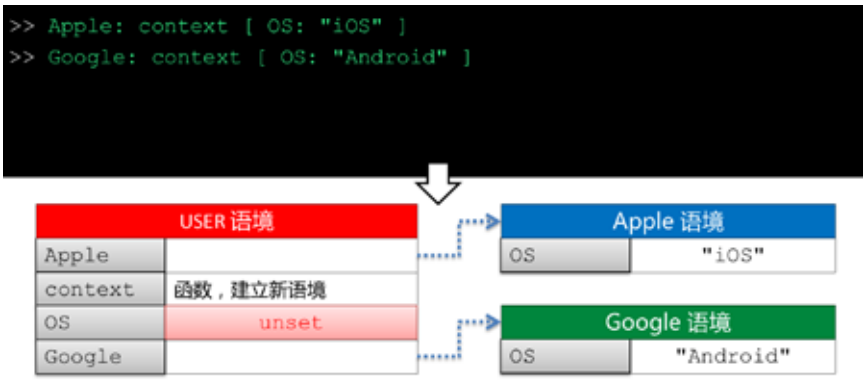
步骤一，我先通过 `context` 函数，分别为苹果（Apple）与谷歌（Google）建立它们各自专用的语境。苹果的语境叫做 `Apple`，里面只有一个单字 `OS`，定义为 `"iOS"`。谷歌的语境叫做 `Google`，里面只有一个单字 `OS`，定义为 `"Android"`（安卓）。意思是这两家公司谈到 OS（操作系统）的时候，指的是不同的东西，Apple 说的 OS 是 iOS，Google 则是指 Android。

步骤二，我建立两个代码方块，分别叫做 `Apple-fan`（果粉）与 `Google-fan`（谷粉），内容都是 `[print OS]`。

步骤三是重点：通过 `bind` 函数，让 `Apple-fan` 采用 `Apple` 语境，让 `Google-fan` 采用 `Google` 语境。

步骤四，我通过 `insert` 函数，把两段代码串在一起（把 `Apple-fan` 与 `Google-fan` 的内容都放到 `code` 中）。我在步骤五证实合并正确。步骤六，执行这段代码，发现显示出来的结果居然不一致，说明一段代码可以采用不同的语境。

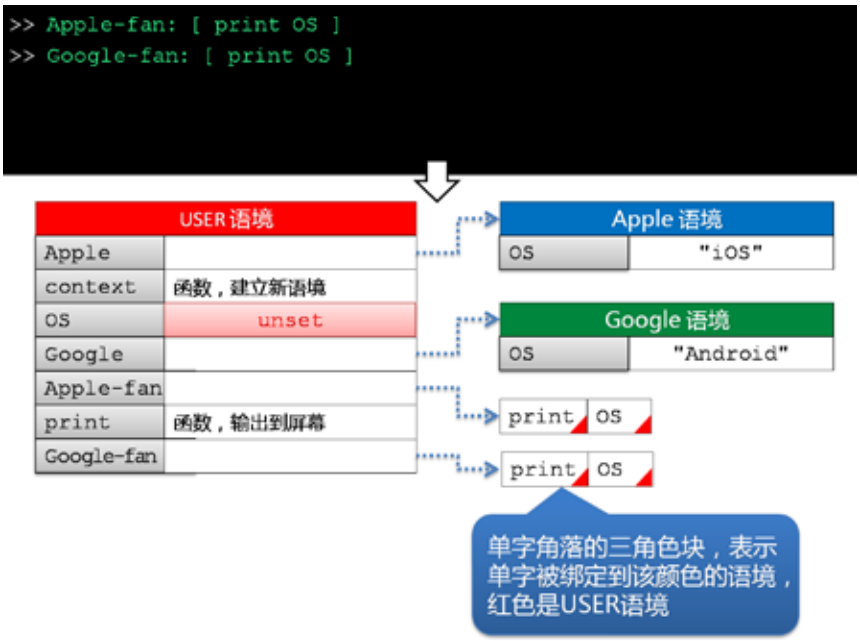
为了加深印象，请看下面的慢动作分解。




```
Apple: context [ OS: "iOS" ]
Google: context [ OS: "Android" ]
```

为了方便解说，我假设上述两个命令行是一起执行的（就像是一个命令行）。

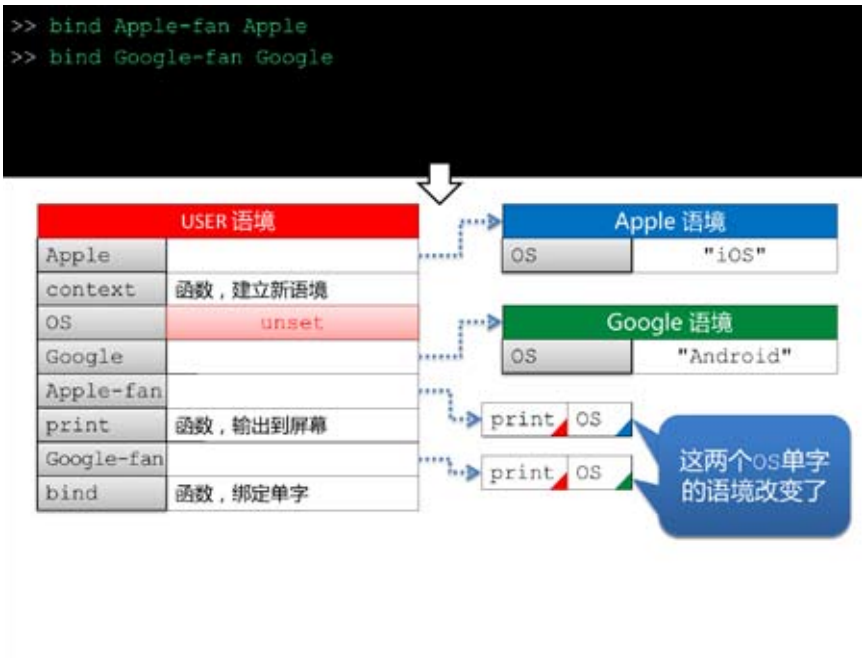
执行上述代码前，REBOL 解释器会在 USER 语境内准备好 Apple、context、OS、Google 这四个单字，并从 LIB 语境中将 context 定义复制过来（LIB 语境中没有 Apple、OS、Google，所以无法复制这三者的定义）。执行此代码后，会产生两个新的语境，分别成为 USER 语境中 Apple 单字与 Google 单字的定义。最后 USER 语境中剩下 OS 的定义是特殊值 unset（表示尚未设定）。




```
Apple-fan: [ print OS ]
Google-fan: [ print OS ]
```

为了方便解说，我假设上述两个命令行是一起执行的（就像是一个命令行）。

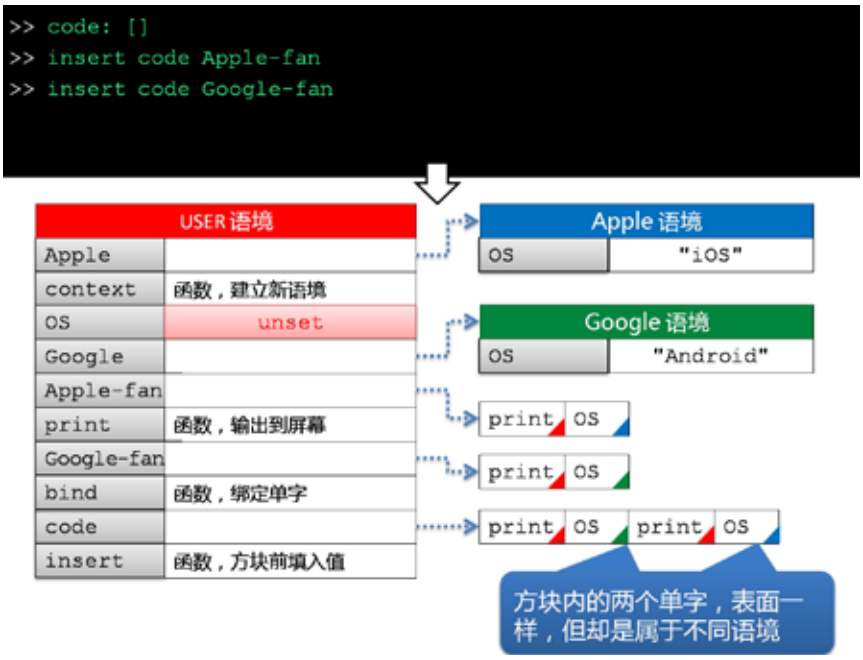
执行上述代码前，REBOL 解释器会在 USER 语境内新增 Apple-fan、print、Google-fan 这三个单字，并从 LIB 语境中将 print 定义复制过来（LIB 语境中没有 Apple-fan 与 Google-fan 的定义）。执行后，Apple-fan 与 Google-fan 的定义都是代码方块，内容都是 [print OS]。print 与 OS 被标上红色三角，表示它们都是采用 USER 语境（红色的表）的单字定义。




```
bind Apple-fan Apple
bind Google-fan Google
```

为了方便解说，我假设上述两个命令行是一起执行的（就像是一个命令行）。

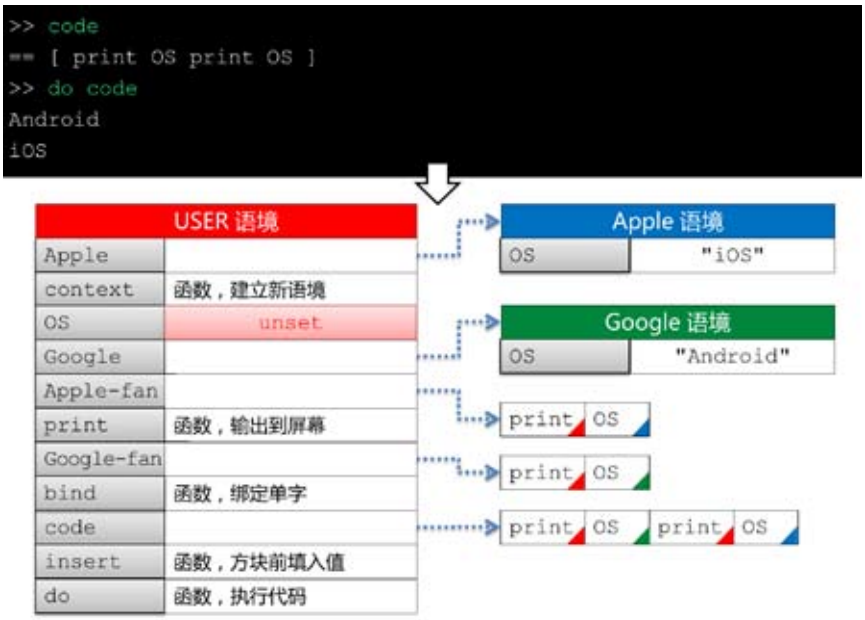
执行上述代码前，REBOL 解释器会在 USER 语境内新增 bind，并从 LIB 语境中将 bind 定义复制过来。执行此代码后，Apple-fan 内的代码要被绑定到 Apple 语境，但 Apple 语境中的单字只有 OS，没有 print，所以只有 OS 被绑定到 Apple 语境，print 维持原来的绑定（USER 语境）。同理，Google-fan 内的代码只有 OS 被绑定到 Google 语境，print 维持原来的绑定（USER 语境）。




```
code: []
insert code Apple-fan
insert code Google-fan
```

为了方便解说，我假设上述三个命令行是一起执行的（就像是一个命令行）。

执行这段代码前，REBOL 解释器会在 USER 语境内新增 code 与 insert，并从 LIB 语境中将 insert 定义复制过来。执行此代码后，code 内容指向一个新的代码方块，方块内有四个单字，复制自 Apple-fan 与 Google-fan。注意区分这里的四个单字分别采用哪些语境。由于 Google-fan 的代码较晚插入方块头部（insert 是插入头部的意思），所以第一个 OS 其实是来自 Google-fan 的。




```
code
do code
```

为了方便解说，我假设上述两个命令行是一起执行的（就像是一个命令行）。

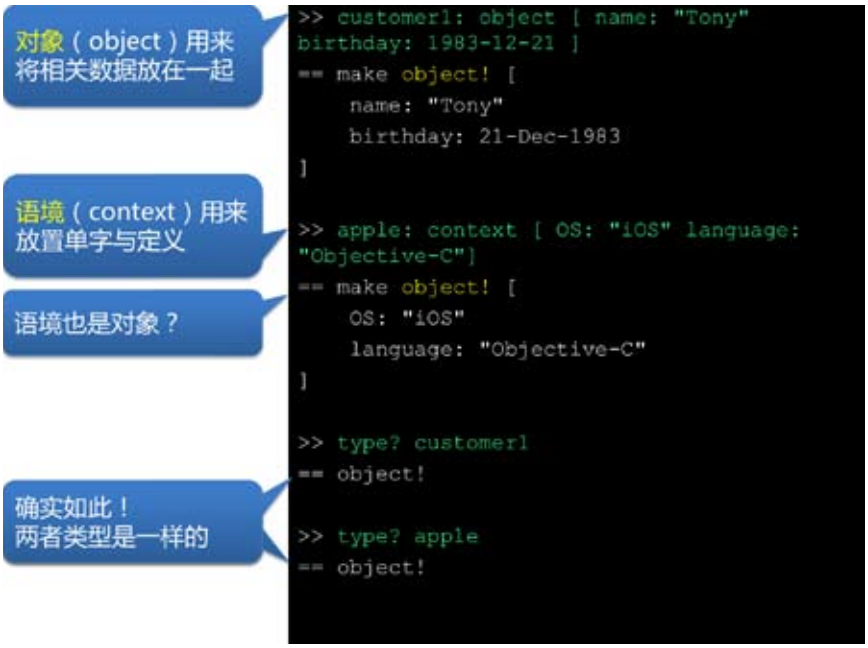
执行这段代码前，REBOL 解释器会在 USER 语境内新增 `do`，并从 LIB 语境中将 `do` 的定义复制过来。执行此代码后，屏幕上会依次打印出 `Google` 语境的 `OS` 值（`"Android"`）与 `Apple` 语境的 `OS` 值（`"iOS"`）。

接续前面的例子，Google 语境与 Apple 语境都有 OS。这时如果我们直接在交互环境中输入 OS，想取得 OS 的值，却得到报错，告诉我们 OS 没有值。USER 语境中确实有 OS，只是其定义是特殊值 unset（未设），相当于没有值。接着我们把 OS 设置为 "Windows"，以后我们取得的 OS 就会是 "Windows"。

要如何获取 Apple 或 Google 内的 OS？我们可以通过加入路径（path）来实现：在 OS 前面冠上它的来源语境，并用斜杠隔开，写成 Apple/OS 与 Google/OS。

路径不限定两层，可以有多层次。例如，system/options/home 就是三层的路径，可以取得 REBOL 主目录。

再来看另一个例子，PI（圆周率）的值原本是 3.14159265358979，我们把它改为简单一点的值 3.14。不用担心这个比较精确的值就因此消失了，你改变的只是 USER 语境中的 PI 复制版本，原本的 PI 还在 LIB 语境中，通过 LIB/PI 路径的写法就可以得到原本的值了。



写代码时，我们常需要让一些值聚在一起，例如，记录同一个人的信息就放在一起 `["Tony" 1983-12-21]`。为了清楚表达这些数据的意义，我们通常会在前面加上字段名称，变成 `[name: "Tony" birthday: 1983-12-21]`。我们常会把像这样的数据包装成对象（object）。对象的写法如下所示：

```
customer1: object  [name: "Tony" birthday: 1983-12-21]
customer2: object  [name: "John" birthday: 1978-1-15]
```

这里有两个对象，它们的数据分别被隔开，不用担心 `"John"` 会覆盖 `"Tony"`，因为它们的 `name` 是在不同的对象内。

这是不是感觉很像语境？没错！ REBOL 其实就是用处理对象的方式来处理语境。语境和对象在 REBOL 语言中是不区分的，作用完全一样。一般来说，语境内放的单字比较多；对象内放置的单字（字段）比较少。语境内的值不固定，变化很多；对象内的值则相对固定，就像是数据库的字段一样。语境内函数占大多数，而对象内数据居多。

程序执行的过程会需要经常查询语境，从中找出单字对应的定义，才知道要如何执行。

我们证明过，单字的定义可以来自不同的语境。甚至即使同样的值，真正执行时也可以有不同的意义（受到该值前面的函数控制），每个值都可能会影响后续值的意义。

REBOL 是 Relative Expression Based Object Language 的缩写，意思就是此语言让你能够根据上下文（语境）做出不同的表达。同样的一个字，在不同的上下文中可以有不同的定义与效果。与上下文有关，更容易理解，代码更容易阅读，更简短。

这是一门神奇的语言，威力强大，既神奇又先进。