

Iteradores

En un programa, se cuentan con tres operaciones principales:

1. Órdenes y comandos.
2. Estructuras condicionales.
3. Estructuras iterativas.

Durante el curso, ya hemos visto las dos primeras, y hemos descubierto cómo las estructuras condicionales pueden alterar completamente el modo de funcionamiento de un programa para que su ejecución varíe en el tiempo (o según unos datos de entrada).

Sin embargo, la verdadera potencia de los scripts y de la programación en general nace de **los bucles** y las **estructuras iterativas**.

Un bucle define en general una operación que se ha de realizar múltiples veces. Por ejemplo, **contar hasta 10** consiste en sumar ‘1’ al valor anterior hasta llegar al número “10”.

Una estructura iterativa define un conjunto de operaciones que se realizan sobre cada elemento de una estructura de datos. Por ejemplo, **cambiar los permisos** de todos los ficheros en el directorio actual.

En bash contamos con múltiples estructuras iterativas:

- **for**
- **while**
- **until**
- **select**

Cada una de las estructuras anteriores nos permiten realizar distintas operaciones y, en realidad, todas se podrían hacer o bien con bucles “**for**” o bien con bucles “**while**”.

A lo largo de este tema vamos a estudiar las características de cada uno de ellos y cuándo son más o menos útiles.

```
user@users-desktop:~$ ./weekdays.sh
Día de la semana: lunes
Día de la semana: martes
Día de la semana: miércoles
Día de la semana: jueves
Día de la semana: viernes
Día de la semana: sábado
Día de la semana: domingo
user@users-desktop:~$ ./double.sh
Dime un número (0 para salir): 5
El doble de 5 es: 10
Dime un número (0 para salir): 3
El doble de 3 es: 6
Dime un número (0 para salir): 0
user@users-desktop:~$ ./choose.sh
1) Saludo
2) Despedida
3) Salir
#? 1
¡Buenos días usuario!
#? 2
¡Hasta luego!
#? 3
user@users-desktop:~$
```

El bucle for

El bucle for

Un bucle “for” cuenta con una estructura básica muy definida:

`for VARIABLE in conjunto; do`

<acciones que se repiten por cada elemento que exista en “conjunto”>

`done`

“conjunto” es una estructura conformada por un conjunto de valores cualesquiera, separados por **espacios en blanco o saltos de línea**.

Como en bash todo son *strings*, “conjunto” puede contener cualquier valor siempre y cuando sea válido.

```
user@users-desktop:~$ cat weekday
#!/bin/env bash

for DIA in lunes martes miércoles jueves viernes sábado domingo; do
    echo "Día de la semana: $DIA"
done

user@users-desktop:~$ ./weekday
Día de la semana: lunes
Día de la semana: martes
Día de la semana: miércoles
Día de la semana: jueves
Día de la semana: viernes
Día de la semana: sábado
Día de la semana: domingo
user@users-desktop:~$ cat nums
#!/bin/env bash

for NUM in 1 2 3 4 5 6 7 8 9 10; do
    echo "NUM vale $NUM en esta iteración"
done

user@users-desktop:~$ ./nums
NUM vale 1 en esta iteración
NUM vale 2 en esta iteración
NUM vale 3 en esta iteración
NUM vale 4 en esta iteración
NUM vale 5 en esta iteración
NUM vale 6 en esta iteración
NUM vale 7 en esta iteración
NUM vale 8 en esta iteración
NUM vale 9 en esta iteración
NUM vale 10 en esta iteración
```

El bucle for

```
user@users-desktop:~$ cat lscripts
#!/bin/env bash
for SCRIPT in $(ls *.sh); do
    echo "Se ha encontrado un script con nombre \"\$NOMBRE\""
done

user@users-desktop:~$ cat NOMBRES
Ana
Miguel
Javier
José Antonio
María
Fernando
user@users-desktop:~$ cat iter-names
#!/bin/env bash

for C in $(cat NOMBRES); do
    echo "El nombre es \"\$C\" en esta iteración"
done

user@users-desktop:~$ ./iter-names
El nombre es "Ana" en esta iteración
El nombre es "Miguel" en esta iteración
El nombre es "Javier" en esta iteración
El nombre es "José" en esta iteración
El nombre es "Antonio" en esta iteración
El nombre es "María" en esta iteración
El nombre es "Fernando" en esta iteración
```

La verdadera potencia de esta estructura iterativa nace cuando **usamos la salida de otro comando como conjunto de datos**.

Los valores de entrada pueden ser cualquier cosa, y el bucle **for** se encargará de realizar las divisiones o bien por espacio en blanco o bien por salto de línea.

Esta última (el salto de línea) tiene precedencia sobre el espacio en blanco y será la opción prioritaria cuando se trabajen con bucles. Sin embargo, si se combinan se usarán ambos.

En el ejemplo de la izquierda, ¿ves algo raro con la salida producida?

El bucle for

Hasta ahora, los rangos numéricos los hemos escrito a mano como un conjunto de datos. Sin embargo, es bastante habitual que dicho rango numérico sea **bastante grande**, presente **saltos entre los elementos** y por tanto resulte difícil escribirlo a mano.

Es por eso por lo que existe la orden “seq”, del [paquete de utilidades GNU](#). Dicho comando presenta la siguiente sintaxis:

`seq last`

`seq first last`

`seq first step last`

De esta forma, una lista del ‘1’ al “10” sería:

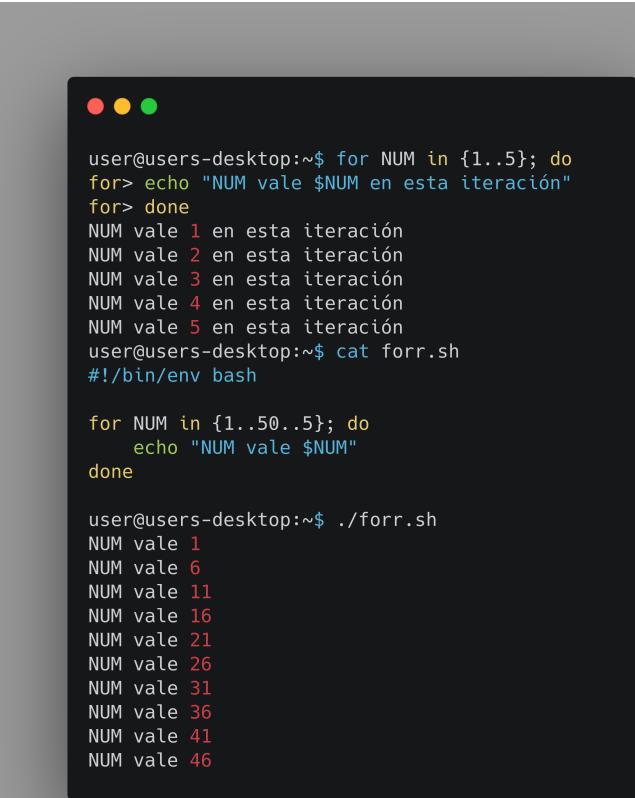
`seq 10`

```
user@users-desktop:~$ for NUM in $(seq 10); do
for> echo "NUM vale $NUM en este paso"
for> done
NUM vale 1 en este paso
NUM vale 2 en este paso
NUM vale 3 en este paso
NUM vale 4 en este paso
NUM vale 5 en este paso
NUM vale 6 en este paso
NUM vale 7 en este paso
NUM vale 8 en este paso
NUM vale 9 en este paso
NUM vale 10 en este paso
user@users-desktop:~$ cat suma100.sh
#!/bin/env bash

SUMA=0
for NUM in $(seq 1 100); do
    let SUMA+=NUM
done
echo "Los números del 1 al 100 suman: $SUMA"

user@users-desktop:~$ ./suma100.sh
Los números del 1 al 100 suman: 5050
```

El bucle for



```
user@users-desktop:~$ for NUM in {1..5}; do
for> echo "NUM vale $NUM en esta iteración"
for> done
NUM vale 1 en esta iteración
NUM vale 2 en esta iteración
NUM vale 3 en esta iteración
NUM vale 4 en esta iteración
NUM vale 5 en esta iteración
user@users-desktop:~$ cat forr.sh
#!/bin/env bash

for NUM in {1..50..5}; do
    echo "NUM vale $NUM"
done

user@users-desktop:~$ ./forr.sh
NUM vale 1
NUM vale 6
NUM vale 11
NUM vale 16
NUM vale 21
NUM vale 26
NUM vale 31
NUM vale 36
NUM vale 41
NUM vale 46
```

Desde la versión de bash 3.0 se introdujo una forma nativa de crear rangos, de forma que no es necesario utilizar el comando “seq”. Hoy en día, lo más habitual es que la versión de la consola de bash sea la 5.0 o superior, por lo que contaremos con estas características (bash --version).

Sin embargo, si se quiere añadir retrocompatibilidad con otras consolas será necesario usar “seq”.

La sintaxis de los rangos en bash es:

```
for VAR in {start..end};
```

Además, se puede especificar el salto:

```
for VAR in {start..end..step};
```

El bucle for

Una última opción **muy potente** es la de usar los bucles **for** del lenguaje C. Esta forma de definir bucles permite un **control total** de la ejecución, del “salto” y de cuánto termina de ejecutarse. Su sintaxis es:

```
for ((expresión-inicio;  
condición-continuar; incremento))
```

Por lo general, “expresión-inicio” será la asignación de una variable; “condición-continuar” comprobará el valor de dicha variable; e “incremento” será una operación matemática sobre el valor de la variable.

Por ejemplo, en la izquierda vemos cómo hacer un bucle que muestre los números pares entre 2 y 40, y un bucle infinito.

```
user@users-desktop:~$ for ((NUM = 2; NUM <= 40; NUM += 2)); do  
for> echo $NUM  
for> done  
0  
2  
4  
6  
8  
10  
12  
14  
16  
18  
20  
22  
24  
26  
28  
30  
32  
34  
36  
38  
40  
  
user@users-desktop:~$ for (( ; )); do  
for> echo "Esto es un bucle infinito. Pulsa \"Ctrl + C\" para salir"  
for> sleep 1  
for> done  
Esto es un bucle infinito. Pulsa "Ctrl + C" para salir  
Esto es un bucle infinito. Pulsa "Ctrl + C" para salir  
^C
```

El bucle for

```
user@users-desktop:~$ cat c-for.sh
#!/bin/env bash

echo "Demostración de los bucles for de 'C'"

# "fori" incremental
for ((i = 0; i < 10; i++)); do
    printf "%d " i
done
echo

# "fori" exponencial
for ((i = 0; i < 100; i *= 5)); do
    printf "%d " i
done
echo

# "forij" - varias variables
for ((i = 10, j = 5; i > 0 && j < 10; i--, j++)); do
    printf "(%d, %d) " i j
done
echo

# "fori" fraccional
for ((i = 100; i > 0; i /= 5)); do
    printf "%d " i
done
echo

# "forever" infinito
for (( ; )); do
    echo "Bucle infinito... (Ctrl + C para salir)"
done
```

Una de las opciones más interesantes de poder usar bucles “**for**” de C es que se “desbloquean” nuevas formas de trabajar con las variables y realizar modificaciones.

Aparecen estos nuevos operadores:

- **++** → VAR=VAR+1
- **--** → VAR=VAR-1
- **+=** → VAR=VAR+X
- **-=** → VAR=VAR-X
- ***=** → VAR=VAR*X
- **/=** → VAR=VAR/X
- **%=** → VAR=VAR%X

El bucle for

```
1#!/bin/env bash
2# Aplicación para copiar todos los scripts que existan en
3# el directorio establecido, indiferentemente de dónde estén, a un
4# USB (u otro directorio) que especificaremos. Si los ficheros
5# ya existen, se actualiza si procede. En otro caso, los copia.
6# Se buscan scripts con formato .sh
7
8#####
9#          DESCARGA: https://s.javinator9889.com/cpsh      #
10#####
11
12# Comprobaciones de seguridad sobre los parámetros
13[[ $# -ne 2 ]] && echo "Usage: $0 SOURCE DESTINATION" && exit 1
14
15SOURCE=$1
16DEST=$2
17
18[[ -z $SOURCE ]] && echo "Fatal! Source directory is required!" && exit 1
19[[ -z $DEST ]] && echo "Fatal! Destination directory is required!" && exit 1
20
21[[ ! -d $SOURCE ]] && echo "Fatal! Source directory does not exist!" && exit 1
22[[ ! -d $DEST ]] && { mkdir -p $DEST || { echo "Fatal! Cannot create $DEST" && exit 1; } }
23
24PROGNAME=$(basename ${0})
25echo "All copy errors will be available at ~/${PROGNAME}.err.log"
26echo "Querying all .sh files..."
27for SCRIPT in $(find $SOURCE -iname "*.sh" 2>/dev/null); do
28    echo "Creating a copy of $SCRIPT at $DEST/$(basename ${SCRIPT})"
29    rsync -auq $SCRIPT $DEST/ 2>>~/0.err.log
30done
31
```

<https://s.javinator9889.com/cpsh>

Bucles while y until

Bucles while y until

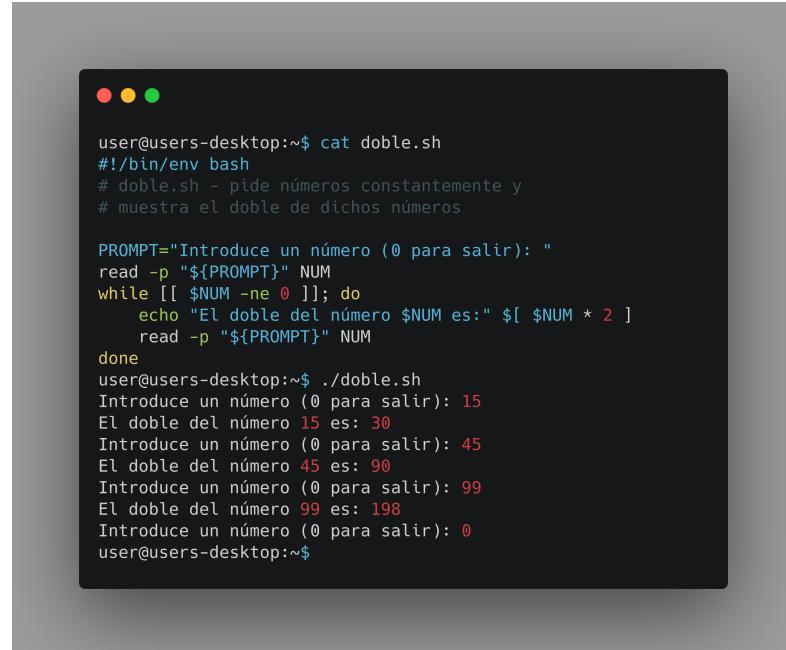
Hemos estudiado que los bucles “for” sirven para recorrer un conjunto de valores, bien sean pertenecientes a una lista o bien correspondientes a una secuencia iterativa.

Sin embargo, muchas veces las iteraciones que necesitemos hacer dependerán directamente de que una variable tome un valor u otro.

Aquí es donde entran en juego los bucles “while”, que simbolizan “haz esta secuencia de pasos **mientras** se cumpla la condición”.

La estructura es la siguiente:

```
while [[ expr ]]; do  
    <acciones a realizar mientras  
    "expr" sea "True">  
  
done
```



```
user@users-desktop:~$ cat doble.sh  
#!/bin/env bash  
# doble.sh - pide números constantemente y  
# muestra el doble de dichos números  
  
PROMPT="Introduce un número (0 para salir): "  
read -p "${PROMPT}" NUM  
while [[ $NUM -ne 0 ]]; do  
    echo "El doble del número $NUM es:" ${NUM}*2  
    read -p "${PROMPT}" NUM  
done  
user@users-desktop:~$ ./doble.sh  
Introduce un número (0 para salir): 15  
El doble del número 15 es: 30  
Introduce un número (0 para salir): 45  
El doble del número 45 es: 90  
Introduce un número (0 para salir): 99  
El doble del número 99 es: 198  
Introduce un número (0 para salir): 0  
user@users-desktop:~$
```

Bucles while y until

```
user@users-desktop:~$ cat doble.sh
#!/bin/env bash
# doble.sh - pide números constantemente y
# muestra el doble de dichos números

PROMPT="Introduce un número (0 para salir): "
read -p "${PROMPT}" NUM
until [[ $NUM -eq 0 ]]; do
    echo "El doble del número $NUM es:" $[ $NUM * 2 ]
    read -p "${PROMPT}" NUM
done
user@users-desktop:~$ ./doble.sh
Introduce un número (0 para salir): 15
El doble del número 15 es: 30
Introduce un número (0 para salir): 45
El doble del número 45 es: 90
Introduce un número (0 para salir): 99
El doble del número 99 es: 198
Introduce un número (0 para salir): 0
user@users-desktop:~$ cat sum.sh
#!/bin/env bash

NUM=1
until [[ $NUM -gt 7 ]]; do
    echo "Número vale: $NUM"
    let NUM=NUM+1
done
user@users-desktop:~$ ./sum.sh
Número vale: 1
Número vale: 2
Número vale: 3
Número vale: 4
Número vale: 5
Número vale: 6
Número vale: 7
```

Los bucles **until** funcionan exactamente igual que los bucles **while** con la diferencia de que la acción que realiza se ejecuta **hasta** que se cumple la condición (**no mientras** se cumple la condición).

La sintaxis es similar a la de **while**:

```
until [[ expr ]]; do
    <acción que se repite hasta
    que “expr” es True>
done
```

Bucles while y until

Una de las funcionalidades principales del bucle **while** es la de **leer ficheros** de texto o las salidas de los comandos línea a línea.

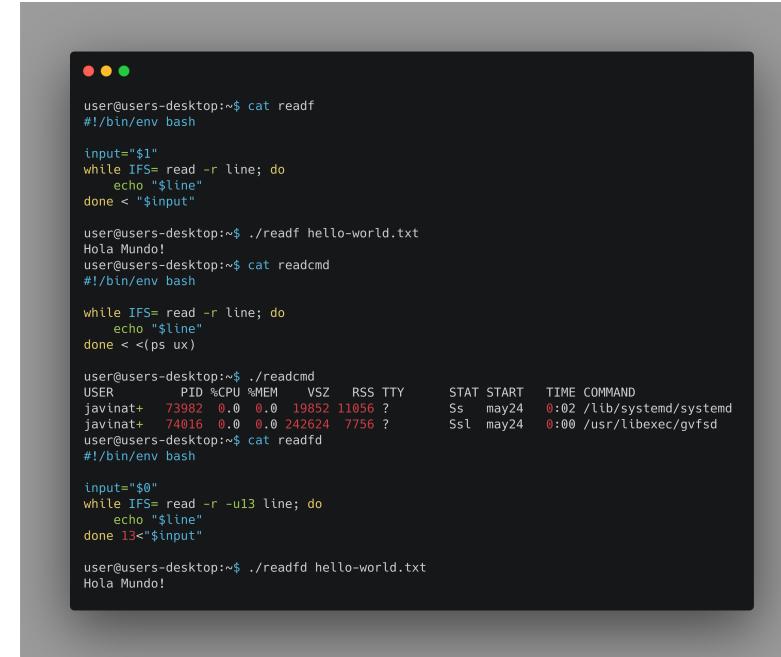
Esto se consigue o bien mediante **redirección estándar** o mediante otras técnicas, como puede ser el uso de **fds**.

La sintaxis es la siguiente:

```
while read -r line; do CMD; done < input.file
```

Se recomienda también añadir la opción **IFS** al inicio para evitar que se eliminan los espacios precedentes/subyacentes.

También se puede usar la sustitución de procesos en lugar de un fichero o los “*here strings*” o los descriptores de fichero para un archivo ya abierto.



```
user@users-desktop:~$ cat readf
#!/bin/env bash

input="$1"
while IFS= read -r line; do
    echo "$line"
done < "$input"

user@users-desktop:~$ ./readf hello-world.txt
Hola Mundo!
user@users-desktop:~$ cat readcmd
#!/bin/env bash

while IFS= read -r line; do
    echo "$line"
done < <(ps ux)

user@users-desktop:~$ ./readcmd
USER      PID %CPU %MEM   VSZ   RSS TTY      STAT START  TIME COMMAND
javinat+  73982  0.0  0.0  19852 11056 ?        Ss  may24  0:02 /lib/systemd/systemd
javinat+  74016  0.0  0.0  242624 7756 ?        Ssl may24  0:00 /usr/libexec/gvfsd
user@users-desktop:~$ cat readfd
#!/bin/env bash

input="$0"
while IFS= read -r -u13 line; do
    echo "$line"
done 13<"$input"

user@users-desktop:~$ ./readfd hello-world.txt
Hola Mundo!
```

Sentencia select

Sentencia select

select si bien es la última estructura iterativa, permite una gran flexibilidad y dinamismo a la hora de crear interacciones con el usuario.

El paso de parámetros que hemos visto ya sirve principalmente para desarrolladores u otras aplicaciones que quieran usar nuestro script, ya que se pueden enviar los parámetros de forma “programática”.

Sin embargo, en una aplicación final donde se quiera interactuar con el usuario, una manera muy simple y sencilla de crear menús es con “select”. Su sintaxis es:

```
select VAR in OPCIONES; do  
    <acciones a realizar>  
done
```

```
user@users-desktop:~$ cat select-example.sh  
#!/bin/env bash  
  
select RESP in Chiste Refrán Salir; do  
    case $RESP in  
        Chiste)  
            echo "¿Cómo llamas a un perro sin patas?"  
            echo "Da igual: no va a venir.";;  
        Refrán)  
            echo "A quién madruga Dios le ayuda";;  
        Salir)  
            break;;  
    esac  
done  
user@users-desktop:~$ ./select-example.sh  
1) Chiste  
2) Refrán  
3) Salir  
#? 1  
¿Cómo llamas a un perro sin patas?  
Da igual: no va a venir.  
1) Chiste  
2) Refrán  
3) Salir  
#? 2  
A quién madruga Dios le ayuda  
1) Chiste  
2) Refrán  
3) Salir  
#? 3  
user@users-desktop:~$
```

Sentencia select

```
user@users-desktop:~$ cat file-manager
#!/bin/env bash

echo "Seleccione uno de los siguientes ficheros:"
select FILE in $(ls); do
    echo "Se ha seleccionado \"$FILE\". ¿Qué operación quieresa aplicar?"
    select OP in Borrar Renombrar "Copia de seguridad" Salir; do
        case $OP in
            Borrar)
                rm -ri $FILE
                break;;
            Renombrar)
                read -p "Nuevo nombre/ruta: " NFILE
                mv -vi $FILE $NFILE
                break;;
            "Copia de seguridad")
                cp -v $FILE $FILE.$(date +%d%m%y)
                break;;
            Salir)
                echo "No se hará ninguna operación"
                break;;
        esac
    done
done

user@users-desktop:~$ ./file-manager
1) adb.1000.log
2) choose.sh
3) cpsh
4) doble.sh
5) file-manager
#? 3
Se ha seleccionado "cpsh". ¿Qué operación quieresa aplicar?
1) Borrar
2) Renombrar
3) Copia de seguridad
4) Salir
#? 2
Nuevo nombre/ruta: cp.sh
renamed 'cpsh' -> 'cp.sh'
#? ^C
```

Es importante notar que la sentencia “**select**” siempre se va a ejecutar por lo que es interesante añadir una opción de salida o escape para no tener que pulsar “Ctrl + C” cuando queramos terminar la ejecución.

Al igual que vimos en el bucle “**for**”, el conjunto de datos se puede generar a partir de una instrucción.

En el ejemplo de la izquierda, listamos los ficheros del directorio actual y se añade un nuevo “**select**” con las opciones a realizar con dicho fichero.

De esta forma, tendríamos un menú interactivo con el que poder gestionar los ficheros.

Sentencia select

```
1 #!/bin/env bash
2
3 # Limpiamos el terminal
4 clear
5
6 # Instalación de la aplicación si no existe
7 if [[ ! $(command -v mpg321) ]]; then
8     CORRECT=1
9     while [[ $CORRECT -ne 0 ]]; do
10         read -n1 -p "La aplicación \"mpg321\" no está instalada. ¿Instalar? [Y/n] " INSTALL
11         echo
12         case $INSTALL in
13             Y|y)
14                 sudo apt install -y mpg321
15                 CORRECT=$?;;
16             N|n)
17                 echo "No se instalará. Saliendo"
18                 exit 1;;
19             *)
20                 echo "Opción no reconocida '$INSTALL'"
21                 CORRECT=1;;
22         esac
23     done
24 fi
25 MP3_FILES=$(find . -iname "*.mp3" 2> /dev/null)
26 MP3_FILES=$MP3_FILES Salir"
27
28 select MP3 in $MP3_FILES; do
29     if [[ $MP3 = "Salir" ]]; then
30         break
31     fi
32
33     echo "Reproduciendo \"$MP3\"..."
34     mpg321 "$MP3" &> /dev/null
35 done
```

<https://s.javinator9889.com/mp3-player>