

Цикл for-loop в стиле C

Краткое описание

```
для (( <EXPR1> ; <EXPR2> ; <EXPR3> )); сделать  
<СПИСОК>  
Выполнено
```

```
# как особый случай: без точки с запятой после ((...))  
для (( <EXPR1> ; <EXPR2> ; <EXPR3> )) выполните  
<СПИСОК>  
Выполнено
```

```
# альтернативный, исторический и недокументированный синтаксис  
для (( <EXPR1> ; <EXPR2> ; <EXPR3> )) {  
<СПИСОК>  
}
```

Описание

Цикл for в стиле C - это составная команда, производная от эквивалентной функции ksh88, которая, в свою очередь, является производной от ключевого слова C "for". Его цель - предоставить удобный способ вычисления арифметических выражений в цикле, а также инициализировать любые требуемые арифметические переменные. Это один из основных механизмов "цикла со счетчиком", доступных в языке.

((; ;)) Синтаксис в верхней части цикла - это не обычная арифметическая составная команда, а часть собственного синтаксиса цикла for в стиле C. Три раздела, разделенные точками с запятой, представляют собой контексты арифметических выражений. Каждый раз, когда требуется оценить один из разделов, раздел сначала обрабатывается для: фигурной скобки, параметра, команды, арифметики и замены / расширения процесса, как обычно для арифметических контекстов. Когда цикл вводится в первый раз, <EXPR1> оценивается, затем <EXPR2> оценивается и проверяется. Если <EXPR2> значение равно true, то выполняется тело цикла. После первой и всех последующих итераций <EXPR1> пропускается, <EXPR3> оценивается, затем <EXPR2> оценивается и проверяется снова. Этот процесс продолжается до <EXPR2> тех пор, пока не станет false .

- <EXPR1> предназначен для **инициализации переменных** перед первым запуском.

- `<EXPR2>` предназначен для **проверки** условия завершения. Это всегда последний раздел для оценки перед выходом из цикла.
- `<EXPR3>` заключается в **изменении** условий после каждой итерации. Например, увеличение счетчика.

⚠ Если одно из этих арифметических выражений в цикле `for` является пустым, оно ведет себя так, как если бы оно было равно 1 (**TRUE** в арифметическом контексте).

⚠ Как и все циклы (оба типа `for -loop`, `while` и `until`), этот цикл может быть:

- Завершается (прерывается) встроенным `break`, необязательно `break N` для выхода из `N` уровней вложенных циклов.
- Принудительно немедленно переходит к следующей итерации с использованием встроенной функции `continue`, необязательно в качестве `continue N` аналога `break N`.

Эквивалентная конструкция с использованием цикла `while` и составной команды арифметического выражения будет структурирована следующим образом:

```
(( <EXPR1> ))
while (( <EXPR2> )); делать
<СПИСОК>
(( <EXPR3> ))
Выполнено
```

Эквивалентная `while` конструкция не совсем одинакова, потому что и цикл, `for` и `while` цикл ведут себя по-разному, если вы используете команду `continue`.

Альтернативный синтаксис

Bash, Ksh93, Mksh и Zsh также предоставляют альтернативный синтаксис для `for` цикла - заключая тело цикла в `{...}` вместо `do ... done`:

```
для ((x=1; x<=3; x ++))
{
  echo $ x
}
```

Этот синтаксис **не документирован** и не должен использоваться. Я нашел определения синтаксического анализатора для него в коде 1.x и в современном коде 4.x. Я предполагаю, что он существует по соображениям совместимости. В отличие от других вышеупомянутых оболочек, Bash не поддерживает аналогичный синтаксис для `case..esac`.

Возвращает статус

Возвращаемый статус - это статус последней команды, выполненной из `<LIST>`, или `FALSE` если какое-либо из арифметических выражений завершилось ошибкой.

Альтернативы и лучшие практики

ЗАДАЧА: показать некоторые альтернативные варианты использования, включающие функции и локальные переменные для инициализации.

Примеры

Простой счетчик

Простой счетчик, цикл повторяется 101 раз (от "0" до "100" - это 101 число → 101 запуск!), И каждый раз переменной `x` присваивается текущее значение.

- Он **инициализирует** `x = 0`
- Перед каждой итерацией **проверяется**, `x ≤ 100`
- После каждой итерации он **меняется** `x++`

```
для ((x = 0 ; x <= 100; x ++)); выполнить  
echo "Счетчик: $ x"
```

Пошаговый счетчик

Это тот же самый счетчик (сравните его с простым примером счетчика выше), но внесенное **изменение** является `x += 10`. Это означает, что он будет считаться от 0 до 100, но с **шагом 10**.

```
для ((x = 0 ; x <= 100; x += 10)); сделать  
echo "Счетчик: $ x"  
готово
```

Анализатор битов

В этом примере перебираются битовые значения байта, начиная со 128 и заканчивая 1. Если этот бит установлен в `testbyte`, он выводит "1", иначе "0" ⇒ он выводит двоичное представление `testbyte` значения (8 бит).

```
#!/usr/bin/env bash
# Пример, написанный для http://wiki.bash-hackers.org/syntax/ccmd/c_for#bits_analyzer
# Основан на оригинале TheBonsai.

функция toBin {
    набор m = 1 $ n = 2 x= 'x[(n* = 2)>m]'
    для ((x = x; n /= 2;)); выполните
    printf %d $(( m & n && 1))
    Выполнено
}

функция main {
    [[ $1 == +([0-9]) ]] || возвращает
    результат набора
    текста, если (( $(ksh -c 'printf %..2d $1' _ "$1") == ( результат =
    $(тоБин "$1") ) )); затем
    printf '%s - это %s в базе 2!\n' "$1" "$result"
    else
    повторяет: "Ой, что-то пошло не так с нашими расчетами". > & 2
    выход 1
    фи
}

основной "$ {1:-123}"

# vim: установить fenc=utf-8 ff= unix ft= sh :
```

Почему он начинается с 128 (наивысшее значение слева), а не с 1 (наименьшее значение справа)? Проще печатать слева направо...

Мы получаем 128 for `n` через рекурсивное арифметическое выражение, хранящееся в `x`, которое вычисляет следующую по величине степень 2 после `m`. Чтобы показать, что это работает, мы используем `ksh93` для двойной проверки ответа, потому что он имеет встроенную функцию для `printf` печати представления любого числа в произвольной базе (до 64). Очень немногие языки имеют встроенную такую способность, даже такие вещи, как Python.

Вверх, вниз, вверх, вниз...

Это отсчитывает вверх и вниз от 0 до `{1:-5}`, `{2:-4}` раз, демонстрируя более сложные арифметические выражения с несколькими переменными.

```
для (( incr = 1, n= 0, times = {2:-4}, step = {1:-5}; (n += incr) %
step || (incr *= -1, --times);)); выполните
printf '%s\n' "$ ((n +1))" "$ n"
готово
```

```
~ $ bash <(xclip -o)
1
2
3
4
5
4
3
2
1
0
1
2
3
4
5
4
3
2
1
```

Соображения о переносимости

- Циклы for в стиле C не являются POSIX. Они доступны в Bash, ksh93 и zsh. Все 3 имеют по существу одинаковый синтаксис и поведение.
- Циклы for в стиле C недоступны в mksh.

Ошибки

- ~~Исправлено в 4.3. Похоже, что в Bash 4.2p10 была ошибка, из-за которой списки команд нельзя отличить от разделителя арифметических аргументов цикла for (обе точки с запятой), поэтому замены команд в выражении цикла for в стиле C не могут содержать более одной команды.~~

Смотрите также

- Внутренний: арифметические выражения
- Внутренний: классический цикл for
- Внутренний: цикл while



Обсуждение

Мартин Кили, [2013/06/06 02:43 \(\)](#)

Следует отметить, что эквивалентность с циклом `while` является приблизительной; она отличается тем, что "продолжить" в цикле `for` приведет к вычислению 3-го ("инкрементного") выражения.