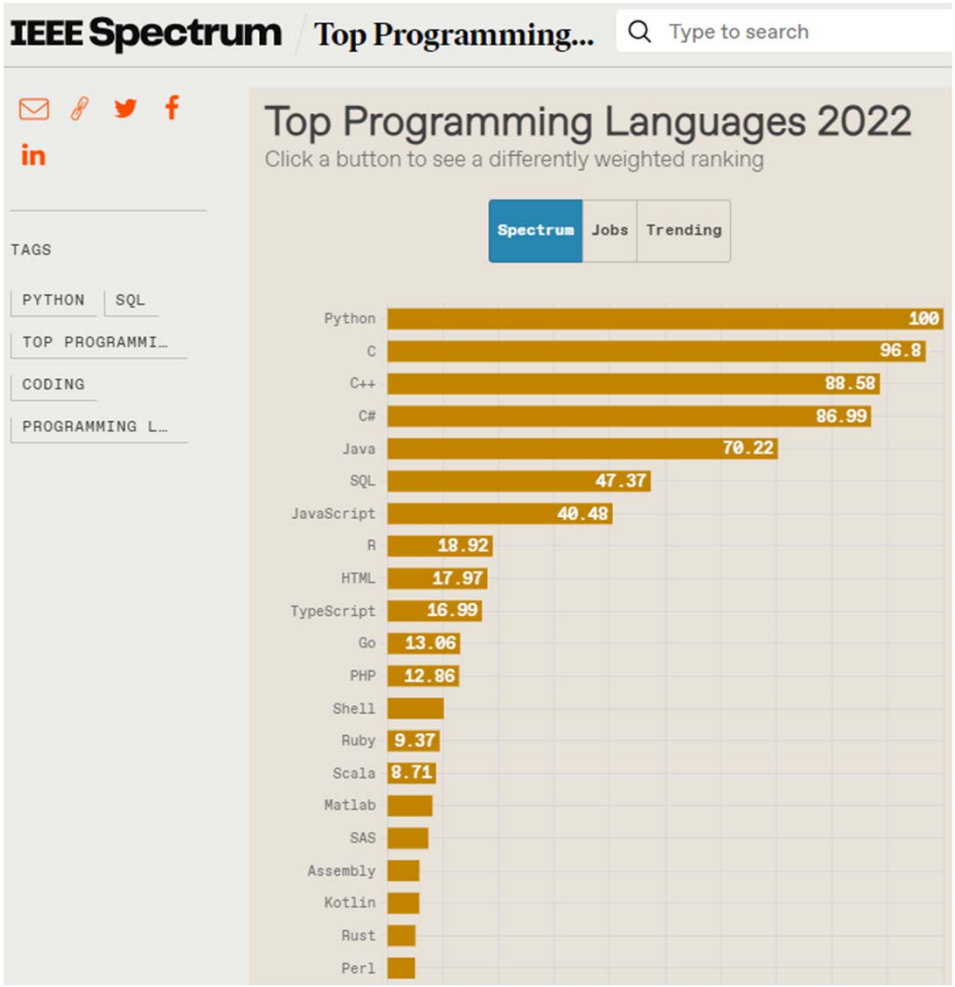
Основы анализа больших данных

Лекция 6. Профилирование и оптимизация кода в R.

Введение


IEEE Spectrum
Top Programming...
Type to search
TAGS
PYTHON
SQL
TOP PROGRAMMI…
CODING
PROGRAMMING L…
Top Programming Languages 2022
Click a button to see a differently weighted ranking
Spectrum
Jobs
Trending
Python 100
C 96.8
C++ 88.58
C# 86.99
Java 70.22
SQL 47.37
JavaScript 40.48
R 18.92
HTML 17.97
TypeScript 16.99
Go 13.06
PHP 12.86
Shell
Ruby 9.37
Scala 8.71
Matlab
SAS
Assembly
Kotlin
Rust
Perl

Профилирование производительности в R

Профилирование производительности - это процесс измерения производительности кода, который позволяет выявить участки, где программа тратит наибольшее количество времени. При профилировании кода осуществляется сбор характеристик программы во время ее выполнения (замер времени выполнения, количества вызовов отдельных функций и строк в коде программы).

Встроенные инструменты R для профилирования:

- system.time
- Rprof

Пакеты для профилирования:

- profvis
- microbenchmark

system.time()

```
system.time(expr, gcFirst = TRUE)
```

```
# Пример использования system.time
factorial <- function(n) {
  if (n == 0 || n == 1) {
    return(1)
  } else {
    return(n * factorial(n - 1))
  }
}

# Измеряем время выполнения функции для n = 10
timing <- system.time({
  factorial(10)
})

print(timing)
```

```
пользователь      система       прошло
           0            0            0
```

```
# еще пример:
> system.time({Sys.sleep(60)})
пользователь      система       прошло
        0.05         0.02        60.75

# если хотим замерить повторные expr
> system.time(replicate(1000, expr))
```

Rprof()

```
# Пример использования Rprof
# Запускаем профилировщик
Rprof(filename = "output.txt")

# Наша функция, которую будем профилировать:
fibonacci <- function(n) {
  if (n <= 1) {
    return(n)
  } else {
    return(fibonacci(n - 1) + fibonacci(n - 2))
  }
}

# Вызываем функцию и профилируем её
fibonacci(20)

# Останавливаем профилировщик
Rprof(NULL)

# Анализ результатов профилирования
summaryRprof("output.txt")
```

```
$by.self
           self.time self.pct total.time total.pct
"fibonacci"      0.66      100       0.66       100

$by.total
           total.time total.pct self.time self.pct
"fibonacci"       0.66       100      0.66      100

$sample.interval
[1] 0.02

$sampling.time
[1] 0.66
```

Пакет tictoc

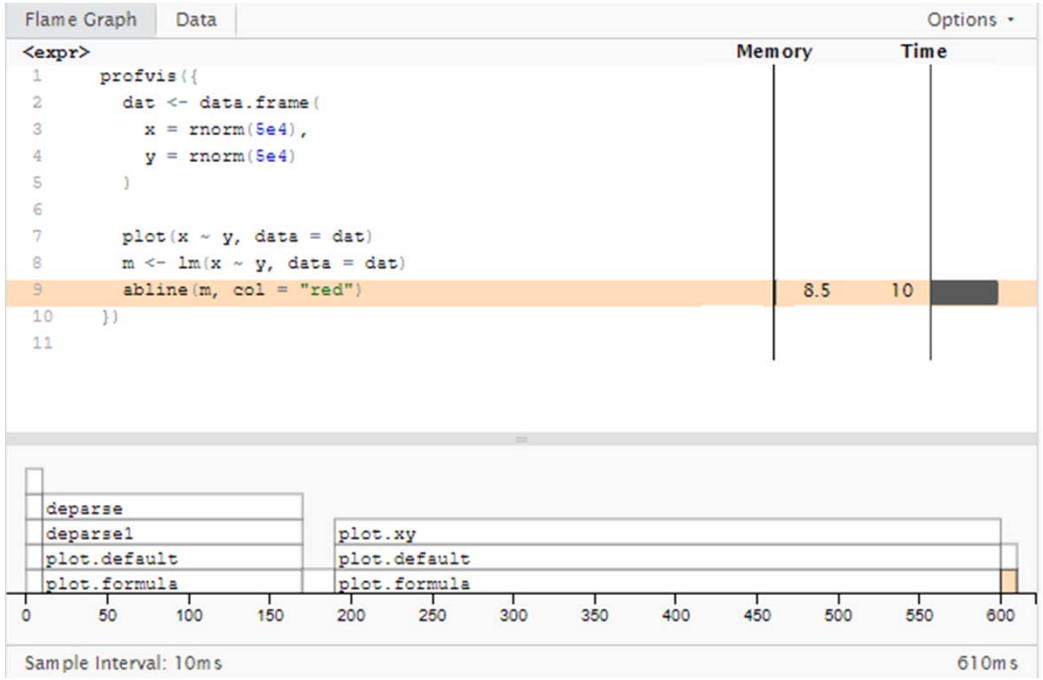
```
# Пример использования tic и toc
install.packages("tictoc")
library(tictoc)

> tic("fibonacci")
> fibonacci(30)
[1] 832040
> tic("factorial")
> rf <- replicate(100,factorial(100))
> toc()
factorial: 0.02 sec elapsed
> toc()
fibonacci: 1.09 sec elapsed
```

Пакет profvis

```
# Пример использования profvis
install.packages("profvis")
library(profvis)

profvis({
  dat <- data.frame(
    x = rnorm(5e4),
    y = rnorm(5e4)
  )
plot(x ~ y, data = dat)
  m <- lm(x ~ y, data = dat)
  abline(m, col = "red")
})
```



Пакет microbenchmark

```
microbenchmark(
  ...,
  list = NULL,
  times = 100L,
  unit = NULL,
  check = NULL,
  control = list(),
  setup = NULL
)
```

Особенности пакета `microbenchmark`:

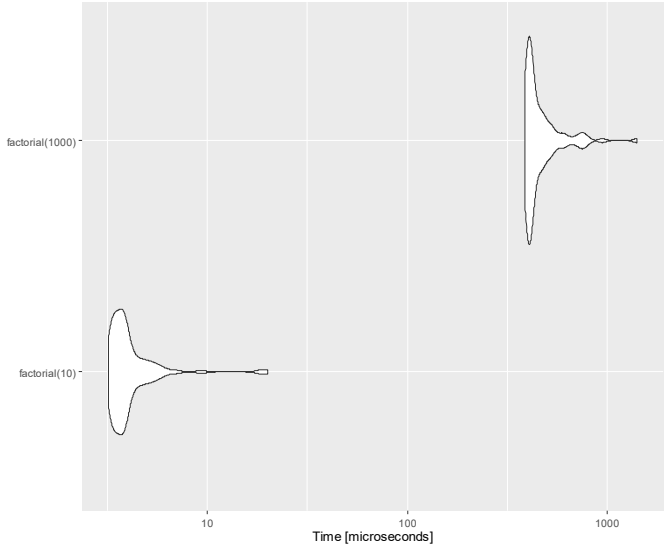
- Возможность измерения времени выполнения выражения вплоть до наносекунд;
- Возможность контролировать последовательность выполнения выражений: случайно или последовательно;
- Возможность проведения предварительных испытаний до начала процесса измерений.
- Также с помощью функции `microbenchmark()` можно получить исходную информацию о времени выполнения каждой попытки, что даёт достаточно широкие возможности по обработке и анализу полученных результатов;
- Интеграция с `ggplot2` для построения графиков результатов `microbenchmark()`.

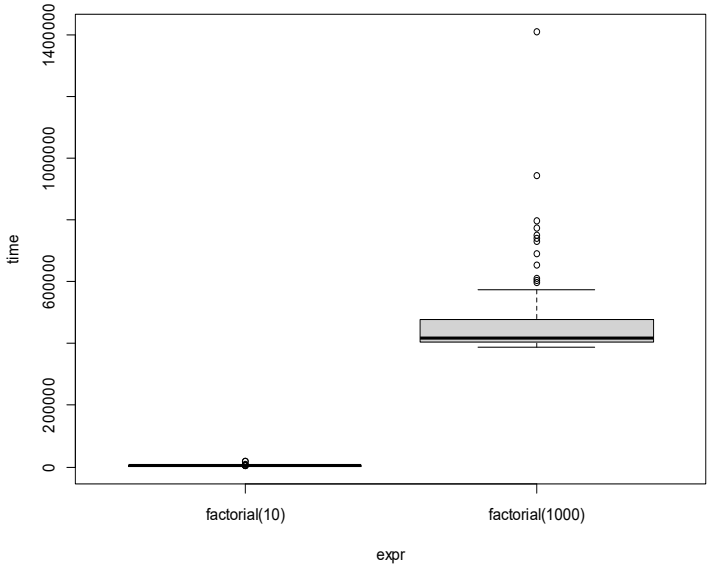
Пакет microbenchmark - пример

```
install.packages("microbenchmark")
library(microbenchmark)
library("ggplot2")

measure <- microbenchmark(factorial(10),factorial(1000))
print(measure)
autoplot(measure)
plot(measure)
```

```
Unit: microseconds
           expr   min     lq     mean median     uq    max neval
   factorial(10)   3.2   3.40   4.291   3.80   4.20   19.9   100
 factorial(1000) 388.6 403.85 472.699 416.65 476.35 1408.3   100
```





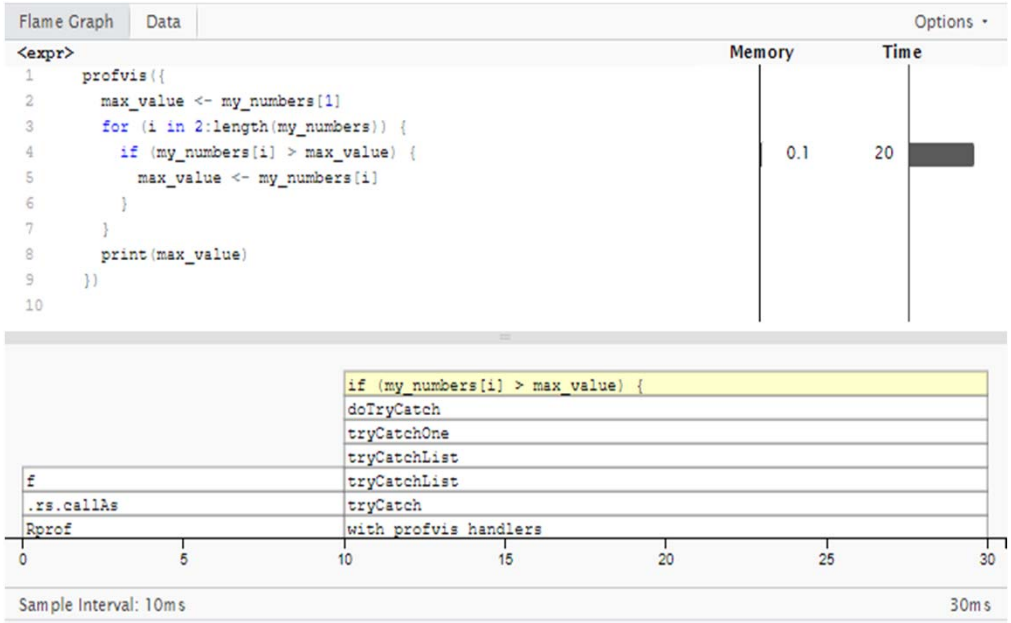
Анализ результатов профилирования

Пример 1: Идентификация узких мест в функции

```
find_max <- function(numbers) {
  max_value <- numbers[1]
  for (i in 2:length(numbers)) {
    if (numbers[i] > max_value) {
      max_value <- numbers[i]
    }
  }
  return(max_value)}
}
# Вектор чисел
my_numbers <- rnorm(1e5)

# Запускаем профилирование
profvis({
  max_value <- my_numbers[1]
  for (i in 2:length(my_numbers)) {
    if (my_numbers[i] > max_value) {
      max_value <- my_numbers[i]
    }
  }
  print(max_value)
})
```

```
[1] 4.65647
```




```
> microbenchmark(find_max(my_numbers), max(my_numbers))
Unit: milliseconds
                  expr     min       lq      mean   median       uq
max neval
 find_max(my_numbers) 29.1512 30.32500 30.948572 30.98210 31.4577
36.1159   100
      max(my_numbers)  1.0061  1.24565  1.343449  1.32485  1.3693
2.4083   100
```

Анализ результатов профилирования

Пример 2: Идентификация затратных операций

```
# Функция сортировки вставками
insertion_sort <- function(numbers) {
  n <- length(numbers)
  for (i in 2:n) {
    key <- numbers[i]
    j <- i - 1
    while (j >= 1 && numbers[j] > key) {
      numbers[j + 1] <- numbers[j]
      j <- j - 1
    }
    numbers[j + 1] <- key
  }
  return(numbers)
}
# Вектор чисел
my_numbers <- c(3, 8, 2, 5, 10, 4, 7, 6)

# Запускаем профилирование
Rprof(filename = "output.txt")
insertion_sort(my_numbers)
Rprof(NULL)
summaryRprof("output.txt")
```

Методы оптимизации R-скриптов

Использование векторизации и функций из базового пакета R, Оптимизация циклов

```
# Пример использования векторизации
my_vector <- c(1, 2, 3, 4, 5)

# Способ вычисления суммы элементов
вектора с использованием цикла:
sum_result <- 0
for (i in 1:length(my_vector)) {
  sum_result <- sum_result +
my_vector[i]
}

# Способ с использованием векторизации:
sum_result_vectorized <- sum(my_vector)

print(sum_result)
print(sum_result_vectorized)
```

```
# Пример оптимизации циклов
n <- 1000
numbers <- 1:n

# Вычисление суммы квадратов чисел через цикл:
sum_squared <- 0
for (i in numbers) {
  sum_squared <- sum_squared + i^2
}

# Использование функции sum() и векторизации:
sum_squared_vectorized <- sum(numbers^2)

print(sum_squared)
print(sum_squared_vectorized)
```

Методы оптимизации R-скриптов

Пакеты для оптимизации: compiler, Rcpp, data.table и другие

- **compiler**

```
# Пример использования пакета compiler
install.packages("compiler")
library(compiler)

# Некомпилированная функция среднего
mean_fun = function(x) {
  m = 0
  n = length(x)
  for (i in seq_len(n))
    m = m + x[i] / n
  m
}
# Компилируем функцию
compiled_mean_fun <- cmpfun(mean_fun)

# Проверяем производительность обеих функций
x <- rnorm(1000)
microbenchmark(times = 10, unit = "ms",
           mean_fun(x), compiled_mean_fun(x), mean(x))
```

```
Unit: milliseconds
                 expr    min     lq    mean  median     uq    max neval
         mean_fun(x) 0.0336 0.0339 0.32040 0.03405 0.0364 2.8923    10
 compiled_mean_fun(x) 0.0337 0.0339 0.03513 0.03395 0.0370 0.0384    10
             mean(x) 0.0051 0.0053 0.00664 0.00605 0.0064 0.0140    10
```

Методы оптимизации R-скриптов

- **Rcpp**

```
# Пример использования пакета Rcpp
install.packages("Rcpp")
library(Rcpp)

# Код на C++ для вычисления факториала
cppFunction('
int factorial_cpp(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return n * factorial_cpp(n - 1);
  }
}')
```

```
# Вызываем функцию на R с использованием C++ кода
x <- 10
print(factorial_cpp(x))
```

```
> microbenchmark(times=10,unit="ms",factorial(30),factorial_cpp(30))
Unit: milliseconds
             expr    min     lq    mean median     uq    max neval
     factorial(30) 0.0102 0.0102 0.01179 0.0104 0.0105 0.0246    10
 factorial_cpp(30) 0.0008 0.0009 0.00168 0.0009 0.0010 0.0086    10
```

Методы оптимизации R-скриптов

- **data.table**

```
# Пример использования пакета data.table
install.packages("data.table")
library(data.table)

# Создаем две таблицы
table1 <- data.table(id = 1:5, value = c(10, 20, 30, 40, 50))
table2 <- data.table(id = c(2, 4, 6), additional_value = c(100, 200, 300))

# Выполняем операцию слияния таблиц
merged_table <- merge(table1, table2, by = "id", all.x = TRUE)

> print(merged_table)
   id value additional_value
1:  1    10              NA
2:  2    20             100
3:  3    30              NA
4:  4    40             200
5:  5    50              NA
```

Методы оптимизации R-скриптов

Оптимизация работы с памятью и управление переменными

```
# Пример оптимизации работы с памятью и переменными
# Удаляем ненужные объекты после использования
x <- 1:10000
y <- x^2
z <- x + y
rm(y)

# Используем меньше переменных, чтобы избежать дополнительного
копирования
z <- x + x^2
```

Работа с большими данными в R

Оценка объема данных

```
# Пример оценки объема данных
n <- 10^10
big_vector <- 1:n

# Оцениваем размер в байтах
object_size <- object.size(big_vector)
print(paste("Размер вектора данных:", object_size, "байт"))

[1] "Размер вектора данных: 80000000048 байт"

# размер в мегабайтах
> print(object_size, units="Mb")
76293.9 Mb
```

Работа с большими данными в R

Стратегии для работы с большими объемами данных на R:

- **Разделение/дробление данных (chunking):** Это подход, при котором мы разделяем данные на небольшие части (чанки) и обрабатываем их по очереди. Например, мы можем обрабатывать блоки строк в таблице, а не всю таблицу целиком.

- **Использование баз данных**: Перемещение данных из оперативной памяти на диск может быть полезным для обработки больших объемов данных. R поддерживает различные пакеты для работы с базами данных, такие как `RSQLite`, `RMySQL`, `MonetDB`, которые позволяют нам выполнять запросы и агрегации непосредственно на данных в базах данных, минимизируя использование оперативной памяти.

- **Пакеты для обработки больших данных**: R имеет несколько пакетов, специально разработанных для работы с большими объемами данных, такие как `bigmemory`, `ff`, `data.table`, `dplyr backend`. Эти пакеты предоставляют оптимизированные структуры данных и алгоритмы для эффективной обработки больших данных в памяти или на диске.

Подытожим

Основные подходы к профилированию и оптимизации R-скриптов:

- Использовать векторизацию при работе с данными, чтобы избежать использования циклов там, где это возможно.
- Оптимизировать работу с памятью, освобождая ненужные объекты и минимизируя количество переменных.
- При необходимости использовать специальные пакеты для работы с большими объемами данных, такие как `bigmemory`, `ff`, `data.table`, которые предоставляют оптимизированные структуры данных и алгоритмы.
- Тестировать и сравнивать производительность различных решений для выбора наиболее эффективного.