13 Отладка

Программисты часто проводят за отладкой программ больше времени, чем за их написанием. Хорошие навыки отладки чрезвычайно важны. В этой главе мы рассмотрим отладку в R.

13.1. Фундаментальные принципы отладки

> Остерегайтесь ошибок в приведенном выше коде; я только доказал его правильность, но не пробовал его запускать.
>
> *— Дональд Кнут, основоположник computer science*

Хотя отладка скорее относится к искусству, нежели к науке, у нее есть свои фундаментальные принципы. В этой главе будут рассмотрены некоторые приемы отладки.

13.1.1. Суть отладки: принцип подтверждения

Как мы с Питом Зальцманом (Pete Salzman) написали в книге, посвященной отладке («The Art of Debugging, with GDB, DDD, and Eclipse» (No Starch Press, 2008)), сутью отладки является принцип подтверждения.

«В сущности, исправление ошибок в программах — это процесс подтверждения того, что многие положения, в истинность которых вы верите в коде, действительно истинны. Когда вы обнаруживаете, что одно из ваших допущений не выполняется, вы тем самым находите ключ к местонахождению (если не точную природу) ошибки».

Это можно выразить иначе: «Сюрпризы — это хорошо!» Допустим, у вас имеется следующий код:

```
x <- y^2 + 3*g(z,2)
w <- 28
if (w+q > 0) u <- 1 else v <- 10
```

Вы думаете, что значение вашей переменной `x` равно 3 после присваивания? Подтвердите! Думаете, что будет выполнена секция `else`, а не `if` в третьей строке? Подтвердите!

Рано или поздно одно из утверждений, в истинности которых вы не сомневались, окажется неподтвержденным. Это будет означать, что вы обнаружили вероятное местонахождение ошибки, что позволит вам сосредоточиться на природе ошибки.

13.1.2. Запуск Small

Хотя бы в начале процесса отладки следует ограничиваться небольшими, простыми тестовыми сценариями. Работа с большими объектами данных усложнит анализ проблемы.

Конечно, со временем код придется тестировать для больших сложных случаев, но начинать следует с малого.

13.1.3. Модульная нисходящая отладка

Многие хорошие разработчики согласны с тем, что код следует писать по модульному принципу. Код первого уровня не должен быть длиннее, скажем, дюжины строк, большинство из которых должно состоять из вызовов функций. Эти функции должны быть не слишком длинными и вызывать другие функции в случае надобности. Это упрощает организацию кода на стадии написания и поможет в нем разобраться, когда придет время расширения кода.

Отладка должна осуществляться в нисходящем направлении. Допустим, вы установили статус отладки своей функции `f()` (то есть вызвали функцию `debug(f)`, о которой будет рассказано ниже), а `f()` содержит следующую строку:

```
y <- g(x,8)
```

К функции `g()` следует относиться по принципу «презумпции невиновности». Не торопитесь с вызовом `debug(g)` — сначала выполните строку и посмотрите,

вернет ли g() ожидаемое значение. Если значение будет получено, значит, вы только что избежали трудоемкого процесса пошаговой отладки g(). Если g() вернет другое значение, пришло время для вызова debug(g).

13.1.4. Защитное программирование

Также вы можете применить некоторые стратегии «защитного программирования». Допустим, имеется область кода, в которой переменная x должна быть положительной. В нее можно вставить следующую строку:

```
stopifnot(x > 0)
```

Если ранее в этом коде происходит ошибка, из-за которой переменная x равна, допустим, –12, вызов stopifnot() прервет выполнение программы с выдачей сообщения об ошибке:

```
Error: x > 0 is not TRUE
```

(Программисты C могут заметить сходство с командой assert языка C.)

После исправления ошибки и тестирования нового кода можно оставить эту команду на месте — она поможет обнаружить эту ошибку, если та вдруг случайно вернется в ваш код.

13.2. Для чего использовать отладочные средства?

В прежние времена программисты для получения отладочных подтверждений вставляли в свой код временные команды вывода, снова запускали программу и смотрели, что она выведет. Например, чтобы убедиться, что в приведенном выше коде x = 3 вы вставляли команду, которая выводила значение x, и делали нечто похожее для if-else:

```
x <- y^2 + 3*g(z,2)
cat("x =",x,"\n")
w <- 28
if (w+q > 0) {
   u <- 1
   print("the 'if' was done")
} else {
   v <- 10
   print("the 'else' was done")
}
```

Разработчик запускал программу заново и анализировал выведенный результат. После этого он удалял команды вывода и вставлял новые для обнаружения следующей ошибки.

Ручной процесс отладочного вывода хорошо работает один-два раза, но в ходе длинных сеансов отладки он становится утомительным. И что еще хуже, работа по отладке отвлекает разработчика и мешает ему сосредоточиться на поиске ошибки.

Итак, процесс отладки, основанный на вставке команд вывода в код, получается медленным и громоздким, и он отвлекает разработчика от основной работы. Если вы серьезно относитесь к программированию на любом конкретном языке, поищите хорошее отладочное средство для этого языка. Использование отладочных средств значительно упрощает получение значений переменных, проверку того, была ли выполнена секция `if` или `else`, и т. д. Более того, если ошибка приводит к сбою выполнения, отладочные средства смогут проанализировать ее за вас — и возможно, предоставить важную информацию об источнике ошибки. Все это существенно повысит производительность вашей работы.

13.3. Использование отладочных средств R

Базовый пакет R включает подборку отладочных средств; также доступны более функциональные отладочные пакеты. Далее будут рассмотрены как базовые средства, так и другие пакеты, а в расширенном примере будет представлен очень подробный сеанс отладки.

13.3.1. Пошаговое выполнение кода функциями debug() и browser()

Отладочные средства R работают на основе режима просмотра (browser). Разработчик может выполнять код в пошаговом режиме, строку за строкой, анализируя текущее состояние в процессе выполнения. Режим просмотра активизируется вызовом функции `debug()` или `browser()`.

13.3.2. Использование команд просмотра

В режиме просмотра приглашение `>` заменяется на `Browse[d]>` (где `d` — глубина цепочки вызовов). В этом приглашении вводятся любые из следующих команд:

- `n` (от «next») — приказывает R выполнить следующую строку, а затем сделать паузу. Нажатие клавиши Enter также выполняет это действие.
- `c` (от «continue») — аналог `n`, за исключением того, что до следующей паузы могут быть выполнены несколько строк кода. Если программа находится в цикле, эта команда выполнит оставшуюся часть цикла, а затем сделает паузу при выходе из цикла. Если программа выполняет функцию, но не находится в цикле, оставшаяся часть функции будет выполнена перед следующей паузой.
- любая команда R — в режиме просмотра вы все еще находитесь в интерактивном режиме R, а следовательно, можете запросить значение `x`, для чего достаточно ввести `x`. Конечно, если у вас имеется переменная, имя которой совпадает с командой режима просмотра, необходимо использовать явный вызов с `print()` — например, `print(n)`.
- `where` — выводит трассировку стека. Выводит последовательность вызовов функций, которая привела к текущей позиции.
- `Q` — завершает режим просмотра и возвращается к основному интерактивному режиму R.

13.3.3. Назначение точек прерывания

Вызов `debug(f)` помещает вызов `browser()` в начало `f()`. Тем не менее в некоторых случаях такой инструмент может быть слишком грубым. Если вы подозреваете, что ошибка скрыта в середине функции, вам придется долго пробираться через весь промежуточный код.

Проблема решается установкой точек прерывания в ключевых позициях вашего кода — в тех местах, в которых выполнение должно остановиться. Как сделать это в R? Вы можете вызвать режим просмотра напрямую или воспользоваться функцией `setBreakpoint()` (в R версии 2.10 и выше).

13.3.3.1. Прямой вызов browser()

Точку прерывания можно установить простым включением вызовов `browser()` в местах, представляющих для вас интерес в коде. Фактически это эквивалентно расстановке точек прерывания.

Вызов `browser()` можно сделать условным, чтобы он активизировался только в конкретных ситуациях. Используйте аргумент `expr` для определения таких ситуаций. Например, вы подозреваете, что ошибка возникает только тогда,

когда некая переменная s превышает 1. Для этого можно использовать следующий код:

```
browser(s > 1)
```

Режим просмотра будет включаться только в том случае, если s больше 1. Аналогичного эффекта можно добиться следующей командой:

```
if (s > 1) browser()
```

Прямой вызов browser() вместо входа в отладчик через debug() очень полезен в ситуациях с циклом с множеством итераций, если ошибка, скажем, всплывает только после 50-й итерации. Если переменная цикла равна i, можно включить следующую команду:

```
if (i > 49) browser()
```

И это позволит вам избежать хлопот с пошаговым выполнением первых 49 итераций!

13.3.3.2. Функция setBreakpoint()

Начиная с R 2.10, функция setBreakpoint() может использоваться в формате

```
setBreakpoint(имя_файла,номер_строки)
```

Это приведет к вызову browser() в строке *номер_строки* исходного файла *имя_файла*.

Данная возможность особенно полезна тогда, когда вы где-то в середине сеанса отладки выполняете код в пошаговом режиме. Допустим, в настоящее время вы находитесь в строке 12 исходного файла x.R и хотите установить точку прерывания в строке 28. Вместо того чтобы выходить из отладчика, добавлять вызов browser() в строке 28, а затем входить в функцию заново, достаточно ввести следующую команду:

```
> setBreakpoint("x.R",28)
```

После этого можно продолжить выполнение в отладчике — допустим, командой c.

Функция setBreakpoint() вызывает функцию trace(), рассмотренную в следующем разделе. Таким образом, для отмены точки прерывания следует отменить отслеживание. Например, если функция setBreakpoint() была вызвана в строке функции g(), ее можно отменить следующим вызовом:

```
> untrace(g)
```

Функцию `setBreakpoint()` можно вызывать независимо от того, находитесь вы в настоящий момент в отладчике или нет. Если вы не находитесь в отладчике и при выполнении соответствующей функции управление будет передано в точку прерывания, вы автоматически перейдете в режим просмотра. Происходит то же, что при вызове `browser()`, но в этом случае вам не придется изменять свой код в текстовом редакторе.

13.3.4. Функция trace()

Функция `trace()` отличается мощью и гибкостью, хотя на ее освоение у вас уйдет некоторое время. Мы рассмотрим некоторые простейшие формы, начиная со следующей:

```
> trace(f,t)
```

Этот вызов приказывает R вызывать функцию `t()` при каждом входе в функцию `f()`. Допустим, вы хотите установить точку прерывания в начале функции `gy()`. Для этого можно воспользоваться следующей командой:

```
> trace(gy,browser)
```

Произойдет то же, что происходит при включении команды `browser()` в исходный код `gy()`, но этот способ быстрее и удобнее: вам не нужно вставлять команду, сохранять файл и снова выполнять `source()` для загрузки новой версии файла. Вызов `trace()` не изменяет исходный файл, хотя и модифицирует временную версию файла, находящуюся под управлением R. Кроме того, он быстрее и проще отменяется — для этого достаточно выполнить `untrace()`:

```
> untrace(gy)
```

Отслеживание можно включать и отключать глобально вызовом `tracingState()`; с аргументом `TRUE` отслеживание включается, а с аргументом `FALSE` оно отключается.

13.3.5. Выполнение проверок после сбоя функциями traceback() и debugger()

Допустим, в вашем коде R происходит сбой, когда он не выполняется в отладчике. Даже после этого в вашем распоряжении имеется полезный инструмент отладки. Чтобы провести «вскрытие», вызовите `traceback()`. Вы узнаете, в какой функции возникла проблема и какая цепочка вызовов привела к этой функции.

Чтобы получать намного больше информации, настройте R для вывода дампа кадров в случае сбоя:

```
> options(error=dump.frames)
```

Если это было сделано, после сбоя выполните следующую команду:

```
> debugger()
```

Вам будет предложено выбрать уровни вызовов функций для просмотра. Для каждого выбранного уровня вы сможете просмотреть значения переменных на этом уровне. Чтобы после просмотра одного уровня вернуться в главное меню `debugger()`, нажмите клавишу N.

Чтобы автоматически входить в отладчик, напишите следующий код:

```
> options(error=recover)
```

Однако учтите, что если вы выберете автоматический вариант, он будет переводить вас в отладчик даже при простой синтаксической ошибке (не самый полезный момент для входа в отладчик).

Чтобы отключить все эти аспекты поведения, введите следующую команду:

```
> options(error=NULL)
```

Примеры использования этих возможностей продемонстрированы в следующем разделе.

13.3.6. Расширенный пример: два полных сеанса отладки

Итак, мы рассмотрели отладочные средства R. Теперь попробуем использовать их для поиска и исправления ошибок в коде. Начнем с простого примера, а потом перейдем к более сложному.

13.3.6.1. Поиск серий единиц

Для начала вспомните расширенный пример с поиском серий единиц из главы 2. Ниже приведена версия этого кода с ошибкой:

```
1 findruns <- function(x,k) {
2    n <- length(x)
3    runs <- NULL
4    for (i in 1:(n-k)) {
```

```
5       if (all(x[i:i+k-1]==1)) runs <- c(runs,i)
6    }
7    return(runs)
8 }
```

Проверим на небольшом тестовом сценарии:

```
> source("findruns.R")
> findruns(c(1,0,0,1,1,0,1,1,1),2)
[1] 3 4 6 7
```

Функция должна сообщить о сериях единиц с индексами 4, 7 и 8, но при запуске были обнаружены индексы, которых быть не должно, а некоторые серии были пропущены. Что-то пошло не так. Давайте войдем в отладчик и осмотримся.

```
> debug(findruns)
> findruns(c(1,0,0,1,1,0,1,1,1),2)
debugging in: findruns(c(1, 0, 0, 1, 1, 0, 1, 1, 1), 2)
debug at findruns.R#1: {
   n <- length(x)
   runs <- NULL
   for (i in 1:(n – k)) {
      if (all(x[i:i+k-1]==1))
         runs <- c(runs, i)
   }
   return(runs)
}
attr(,"srcfile")
findruns.R
```

В соответствии с принципом подтверждения мы сначала убедимся в том, что тестовый вектор был получен правильно:

```
Browse[2]> x
[1] 1 0 0 1 1 0 1 1 1
```

Пока все хорошо. Пройдем по этому коду более внимательно. Мы пару раз нажали `n`, чтобы код был выполнен в пошаговом режиме.

```
Browse[2]> n
debug at findruns.R#2: n <- length(x)
Browse[2]> n
debug at findruns.R#3: runs <- NULL
Browse[2]> print(n)
[1] 9
```

После каждого шага R сообщает, какая команда будет выполнена следующей. Другими словами, во время выполнения `print(n)` присваивание `NULL` переменной `runs` еще не было выполнено.

Также обратите внимание на то, что хотя обычно значение переменной можно вывести простым вводом имени, с переменной `n` такой способ не сработает, потому что `n` также является сокращением команды `next` отладчика. А значит, необходимо вызвать `print()`.

В любом случае выяснилось, что длина тестового вектора равна 9 — это подтверждает то, что нам уже известно. Продолжим выполнение в пошаговом режиме и войдем в цикл:

```
Browse[2]> n
debug at findruns.R#4: for (i in 1:(n-k+1)){
    if (all(x[i:i+k-1]==1))
        runs <- c(runs, i)
}
Browse[2]> n
debug at findruns.R#4: i
Browse[2]> n
debug at findruns.R#5: if (all(x[i:i+k-1]==1)) runs <- c(runs, i)
```

Так как переменная `k` равна 2 (то есть мы проверяем серии длины 2), команда `if()` должна проверить первые два элемента `x`, то есть (1,0). Убедимся в этом:

```
Browse[2]> x[i:i+k-1]
[1] 0
```

Подтверждение *не* прошло. Проверим правильность диапазона индексов — это должен быть диапазон `1:2`. Так?

```
Browse[2]> i:i+k-1
[1] 2
```

Снова неправильно. Тогда как насчет переменных `i` и `k`? Они должны быть равны 1 и 2 соответственно. Так ли это?

```
Browse[2]> i
[1] 1
Browse[2]> k
[1] 2
```

Что ж, это предположение подтвердилось. Следовательно, проблема должна быть с выражением `i:i+k-1`. После некоторых размышлений становится ясно, что здесь возникает проблема с приоритетом операторов. Исправим ее и приведем к правильной форме `i:(i+k-1)`.

Теперь нормально?

```
> source("findruns.R")
> findruns(c(1,0,0,1,1,0,1,1,1),2)
[1] 4 7
```

Нет. Как упоминалось ранее, должно быть (4, 7, 8).

Установим точку прерывания в цикле и присмотримся повнимательнее:

```
> setBreakpoint("findruns.R",5)
/home/nm/findruns.R#5:
 findruns step 4,4,2 in <environment: R_GlobalEnv>
> findruns(c(1,0,0,1,1,0,1,1,1),2)
findruns.R#5
Called from: eval(expr, envir, enclos)
Browse[1]> x[i:(i+k-1)]
[1] 1 0
```

Хорошо, мы имеем дело с первыми двумя элементами вектора, так что первое исправление ошибки работает. Перейдем ко второй итерации цикла:

```
Browse[1]> c
findruns.R#5
Called from: eval(expr, envir, enclos)
Browse[1]> i
[1] 2
Browse[1]> x[i:(i+k-1)]
[1] 0 0
```

Тоже правильно. Можно перейти к другой итерации, но вместо этого рассмотрим последнюю итерацию — одно из тех мест, в котором ошибки часто встречаются в циклах. Создадим условную точку прерывания:

```
findruns <- function(x,k) {
   n <- length(x)
   runs <- NULL
   for (i in 1:(n-k)) {
      if (all(x[i:(i+k-1)]==1)) runs <- c(runs,i)
      if (i == n-k) browser() # Прервать в последней итерации цикла
   }
   return(runs)
}
```

А теперь повторим попытку:

```
> source("findruns.R")
> findruns(c(1,0,0,1,1,0,1,1,1),2)
Called from: findruns(c(1, 0, 0, 1, 1, 0, 1, 1, 1), 2)
Browse[1]> i
[1] 7
```