Лекция 3. Работа с массивами в языке Julia

1. Понятие однородного (плотного) и разнородного (не плотного) массива

Плотный массив — это массив, все элементы которого имеют один и тот же размер и тип и хранятся в последовательно, байт за байтом в **непрерывной области памяти**.

Массивы могут быть одномерными (такие массивы называются векторами), двумерными (матрицами) или многомерными (тензорами). Элементы n-мерного массива индексируются n индексами.

Например, если `a` - это вектор некоторого типа, то его `i`-ый элемент записывается как `a[i]`. Если `A` - это матрица, то элемент матрицы, находящийся на пересечении `i`-ой строки и `j`-го столбца, записывается, как `A[i,j]`. В случае `n` мерного массива `T` эго элементы определяются n индексами: `T[i1,i2,...,in]` (многоточие здесь - это просто условность).

Вне зависимости от размерности (числа измерений) массива в памяти эго элементы размещаются в линейном порядке (потому что память имеет линейное адресное пространство). Массивы Julia следуя традиции математических языков программирования таких как FORTRAN, Fortran, MATLAB, размещает свои массивы в памяти по столбцам (column-mjor порядок).

А именно, если речь идет о векторе, то в памяти эго элементы размещаются в естественном порядке.

Если речь идет о матрице, то в памяти сначала размещаются элементы первого столбца, следом - элементы второго и т.д.

Если речь идет о трехмерных массивах, то такие массивы рассматриваются как "многослойная матрица", и в памяти сначала размещается первая матрица (по указанному выше правилу), затем вторая и т.д..

В общем случае любой n-мерный массив, рассматривается как несколько "слоев" (n-1)-мерных массивов, причем номер слоя определяется последним индексом массива. И это определяет правило размещения многомерных массивов в памяти.

Благодаря этому правилу к элементам любого многомерного массива можно обращаться с помощью одного единственного индекса, как к элементам вектора.

Плотные массивы также позволяют эффективно использовать особенность современных процессоров, называемую **SIMD** (от англ. *Single Instruction, Multiple Data*, «одна инструкция — множество данных»). Это означает, что процессор может за один такт выполнить одну и ту же арифметическую операцию (например, сложение или умножение) сразу над несколькими значениями из массива.

Для этого крайне важно, чтобы данные:

- были размещены **в одной непрерывной области памяти**,
- имели **одинаковый тип и размер**,
- и находились **в предсказуемом порядке**.

Именно такие условия выполняются в случае **плотных массивов**, и именно поэтому плотные массивы — это основа эффективных численных вычислений.

И так, плотные массивы обеспечивают

- высокую эффективность при доступе к элементам (особенно в циклах);
- возможность использования векторных инструкций процессора (SIMD);
- предсказуемое поведение с точки зрения использования кеша процессора;
- отсутствие дополнительных расходов по сравнению со случаем, как было бы, если бы элементы массива были бы **разнородными**.

Бывают массивы, у которых типы элементов не обязаны быть одинаковыми. Это так называемые разнородные (не плотные) массивы. Не плотные массивы также могут быть одномерными, двумерными или многомерными, у них также есть индексы, и использование этих индексов внешне не отличается от случая однородных массивов.

Однако в случае разнородных массивов непосредственно в самом массиве (как плотном массиве) хранятся не сами элементы, а ссылки на них. Сами же элементы размещаются где-то в динамической памяти, и их размещение в адресном пространстве может быть достаточно хаотичным. Это обстоятельство делает неоднородные массивы малоэффективными в вычислительном отношении, работа с такими массивами требует от вычислительной системы значительных накладных расходов.

в этом случае было необходимо дополнительно где-то хранить в памяти формат и размер каждого элемента массива в отдельности, т.е. отдельные ссылки на них.

2. Виды памяти, статические и динамические массивы

Память компьютера, используемая программой, подразделяется на

- статическую
- динамическую (кучу)
- автоматическую (стековую)

Это если говорить о внутренней памяти компьютера, бывает еще внешняя память, реализуемая использованием внешних носителей, таких как диски. Но внешняя память - это очень медленная память, мы же говорим именно о внутренней памяти.

При программировании на языке Julia явное управление программистом статической памятью невозможно, и мы говорить о ней не будем. Мы будем говорить только о использовании динамической памяти (кучи) и автоматической памяти (стека).

Динамическая память используется для размещения в ней динамических массивов (и динамических объектов в более широком смысле). Динамические массивы - это массивы, которые могут изменять свои размеры. К динамическому массиву всегда можно добавить новые элементы или удалить из него некоторые уже имеющиеся. Это кажется удобным, но это приводит к значительным накладным расходам, и злоупотребление такими вещами может очень сильно снизить производительность вычислений. Так что без особой необходимости пользоваться этой возможностью недопустимо. Если создан динамический массив, но его размеры не изменяются, то в этом нет ничего страшного, снижения производительности от этого не происходит.

Кроме динамической памяти существует еще стековая или, по другому, автоматическая память. Эта память автоматически выделяется при вызове любой функции, она предназначена для размещения в ней параметров функции и её локальных переменных. Точнее говоря, при вызове функции система автоматически создает так называемый стековый кадр функции, размещая его на вершине специального стека, в котором и размещаются параметры и локальные переменные функции, а также адрес возврата, т.е. адрес адресного пространства, где размещен исполняемый код, с которого должно быть продолжено выполнение программы, после завершения вызванной подпрограммы.

При вызове очередной подпрограммы на вершину стека автоматически помещается соответствующий стековый кадр, а при ее завершении указатель стека перемещается на стековый кадр еще не завершенной программы (вызывавший ту, уже завершившуюся подпрограмму). И так по цепочке.

Автоматическая память - это наиболее эффективный механизм обмена данными между подпрограммами. Однако сами динамические массивы в сек не помещаются (потому что их размеры не определены), динамические массивы размещаются в куче, а в стек помещаются только ссылки на эти массивы.

Кроме того, надо всегда иметь в виду, что размеры стека ограничены и может возникнуть ситуация, когда стек окажется переполнен. В этом случае выполнение программы будет **прервано с ошибкой StackOverflowError**, которую нельзя перехватить стандартными средствами обработки исключений.

3. Конструкторы динамических массивов

Динамические массивы создаются с помощью конструкторов `Vector`, `Matrix` или `Array`, которые размещают массивы в динамической памяти и возвращают ссылку на размещенный в памяти массив.

На самом деле существует только один конструктор динамических массивов - это `Array{T, N}`, где `T` - это тип элементов массива, `N` - число измерений массива. Конструкторы `Vector` и `Matrix` - это всего лишь псевдонимы этого конструктора:

```
Vector{T} = Array{T, 1} where T

Matrix{T} = Array{T, 2} where T
```

Если `T` - это некоторый конкретный тип фиксированного размера, то создаваемый с помощью этих конструкторов массив будет плотным массивом. В противном случае создаваемый массив не будет плотным.

Рассмотрим примеры

```
a = Vector{Int64}(undef, 10) # - плотный массив типа Int64 из 10 элементов
A = Martix{Float64}(undef, 3, 4) # - плотнуй массив типа Float64 из 3*4=12

B = Vector{Vector{Float64}}(undef, 3)
```

В последнем случае массив `B` не является потным - это вектор длины 3, состоящий из векторов типа `Float64`, длина которых может быть произвольной (разные векторы могут иметь разную длину). Сам внешний вектор содержит лишь ссылки на векторы-элементы, и в этом смысле внешний массив тоже является однородным (т.е. плотным), но только сами данные, т.е. элементы типа `Float64`, в нем не содержатся.

Слово `undef` во всех случаях означает, что элементы создаваемых массивов не инициализируются, так что в результате в созданных массивах будет находиться какие-то случайные значения (информационный мусор это называется). Других значений кроме `undef` в этой этой позиции в данном случае не предусмотрено - это слово служит просто напоминанием программисту, что элементы массива не инициализированы.

Размеры многомерного массива могут указываться по отдельности, как в примере в случае создания матрицы `A` размером 3 на 4. Но допускается также объединять эти размеры в один кортеж, например, тот же массив можно было бы создать и так

```
A = Martix{Float64}(undef, (3, 4))
```

Это позволяет, например, создать массив нужного размера используя стандартную функцию `size`

```
C=Matrix{Float64}(undef, size(A))
```

где функция `size`, примененная к некоторому массиву, возвращает кортеж размеров этого массива.

Если нужно получить размер массива вдоль некоторого измерения, то у функции size предусмотрен для этого второй аргумент. Например получить число строк матрицы можно так `size(A, 1)`.

Стандартная функция `length` возвращает число элементов массива, фактически ее определение имеет вид

```
length(A) = prod(size(A))
```

где стандартная функция `prod` возвращает произведение всех элементов своего аргумента.

В случае плотной матицы `A` (см. выше), обращение к ее элементу с индексами `i,j` записывается как `A[i,j]`.

В случае же, матрицы `B`, представленной в виде вектора векторов обращение к ее элементу с индексами `i,j` будет выглядеть так: `B[i][j]`. Здесь `B[i]` - это `i`-ый элемент внешнего вектора (строка матрицы), а `B[i][j]==(B[i])[j]` - это уже `j`-ый элемент `i`-го вектора (строки матрицы). При этом разные строки матрицы могут иметь разную длину, т.е. это способ создать не прямоугольную матрицу. Однако использование такой структуры в некоторых случаях может привести к значительному снижению производительности программы.

Если при создании массива с помощью конструктора указанный тип элементов будет абстрактным типом, а не конкретным, то созданный массив с неизбежностью будет не плотным. Например,

```
A=Vector{Any}(undef, 10)
```

это не плотный массив длины 10, в который можно будет помещать элементы каких угодно типов. Конструкция

```
A=Any[]
```

создаст пустой массив типа `Any`. В общем случае пустой массив некоторого типа `T` можно создать так

```
A=T[]
```

и если T - конкретный тип фиксированного размера, то будет создан плотный пустой массив (потенциально плотным), в противном случае массив плотным не будет.

Добавляются в конец, или, соответственно, в начало массива, новые элементы с помощью стандартный функций push! (для добавления одного элемента) или append! (для добавления сразу нескольких элементов из некоторого другого вектора). Соответственно, функции `firstpush!` и `firstappend!` добавляют элементы в начало динамически расширяемого массива. Для извлечения элемента из конца массива имеется функция `pop!`, а для извлечения элемента из начала массива - функция `firstpop!`.

Кроме того, для вставки элемента в середину массива, и для удаления элемента из середины массива имеются функции `insert!` и `deleteat!`.

Все эти функции, могут приводить, функции `insert!` и `deleteat!` обязательно приведут, к перемещению массивов в памяти, и злоупотребление такими операциями может вызвать значительное снижение производительности кода. Правда операции связанные с добавлением и удалением элементов в конец массива реализованы по умному - там имеется довольно большой буфер (размеры которого можно изменять по своему усмотрению), предотвращающий массовые операции копирования, связанные с перемещением массива на новое место.

Для динамического изменения размера уже созданного массива имеется функция `resize`.

Получить тип элементов уже существующего массива можно с помощью функции `eltype`, получить тип самого массива - с помощью функции `typeof`.

4. Копирование массивов

Если имеется некоторый массив A, то присваивание

```
B = A
```

не приведет к созданию нового массива (копии массива A), потому что массивы представляются ссылками, а не значениями.

Для получения копии массива предусмотрена функция `copy`. В случае, если массив представляет собой, например, структуру типа вектор-векторов, то для его копирования понадобится функция `deepcopy`.

Для создания неинициализированной копии массива используется функция `similar`.

5. Вектор-строки и вектор-столбцы

Рассмотрим пример создания вектор-столбца

```
julia> a=[1,2,3,4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> typeof(a)
Vector{Int64} (alias for Array{Int64, 1})
```

Теперь рассмотрим пример создания вектор-строки

```
julia> a=[1 2 3 4]
1×4 Matrix{Int64}:
 1  2  3  4
```

Необходимо обратить внимание, что вектор-столбец всегда рассматривается именно как вектор, а вектор-строка - как матрица, состоящая только из одной строки.

Вектор может быть создан также с помощью конструкции вида

```
julia> [k^2 for k in 1:5 if k^2>3]
4-element Vector{Int64}:
  4
  9
```

```
 16
 25
```

(такую конструкцию часто называют "списковым включением").

Стоит напомнить, что в Julia аналогичная конструкция с заменой квадратных скобок на круглые создает объект, называемый генератором (генераторы - это специальный вид итераторов).

6. Конкатенация массивов

Несколько матриц подходящих размеров можно соединить в одну матрицу с помощью операции горизонтальной или вертикальной конкатенации. Например

```
julia> A= rand(1:9, 3,4)
3×4 Matrix{Int64}:
 4  1  9  8
 5  8  5  7
 4  5  8  6

julia> B= rand(1:9, 3,2)
3×2 Matrix{Int64}:
 3  8
 5  4
 7  9

julia> [a B]
3×3 Matrix{Int64}:
 1  3  8
 3  5  4
 4  7  9

julia> C = rand(1:9, 2, 4)
2×4 Matrix{Int64}:
 8  2  7  4
 7  3  1  7

julia> [A; C]
5×4 Matrix{Int64}:
 4  1  9  8
 5  8  5  7
 4  5  8  6
 8  2  7  4
 7  3  1  7
```

Те же самые операции могли бы быть записаны с использованием функций `hcat` и `vcat`

```
julia> hcat(A, B)
3×6 Matrix{Int64}:
 4  1  9  8  3  8
 5  8  5  7  5  4
 4  5  8  6  7  9

julia> vcat(A, C)
5×4 Matrix{Int64}:
 4  1  9  8
 5  8  5  7
 4  5  8  6
 8  2  7  4
 7  3  1  7
```

Вектор-столбцы тоже могут участвовать в операции конкатенации, в этом смысле они рассматриваются как матрицы, состоящие из одного столбца.

7. Некоторые функции для генерации массивов

Рассмотрим примеры использования таких функций

```
julia> fill(π, 2, 3) # Создает матрицу заданного размера, повторяя одно и тоже
заданное значени
2×3 Matrix{Irrational{:π}}:
 π  π  π
 π  π  π

julia> zeros(Float64, 3, 3) # создает матрицу заданного размера из нулей
заданного типа
3×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> ones(Int, 5, 1)    # создает матрицу заданного размера из единиц
заданного типа
5-element Vector{Int64}:
 1
 1
 1
 1
 1
```

```
julia> rand(Float64, 4, 5) # создает матрицу заданного размера из равномерно
распределенных на интервале [0, 1) значений заданного типа
4×5 Matrix{Float64}:
 0.40876   0.254414  0.338061  0.0792471  0.888733
 0.963217  0.894485  0.319233  0.503548   0.0371316
 0.845525  0.781952  0.64827   0.346225   0.630937
 0.645925  0.614105  0.251676  0.013469   0.493859

julia>     randn(Float64, 4, 5) # создает матрицу заданного размера из
равномерно распределенных по гауссовскому закону значений заданного типа
4×5 Matrix{Float64}:
 1.93633   -0.879862   0.279283  -0.37628   -1.51881
 0.792368   0.70649    0.340112  -0.361663   0.169132
 1.47073   -0.337297  -0.312106   1.25221   -0.0852966
 0.360937   0.596514  -1.38148   -2.16587   -0.297076
```

8. Срезы и представления срезов

Срезом массива называется массив, составленный из части элементов данного массива. Для получения срезов используются индексы-диапазоны, или векторные индексы.

Например, если имеется матрица A, то для получения ее среза можно использовать диапазоны индексов:

```
A[1:2:5, 1:5]
```

При этом в для записи диапазонов индексов можно использовать ключевые слова `begin` и `end`, в контексте индексного выражения обозначающие минимальное и максимальное значение индексов массива вдоль соответствующего измерения.

В частности, для получения из матрицы отдельного столбца, или отдельной строки, используются срезы вида

```
A[begin:end, j] # - j-ый столбец
A[:, j]         # - более краткая запись того же среза
A[i, :]         # - i-ая строка
```

Срезы представляют собой копии части массива, т.е. если исходный массив был плотным, то и его срезы будут тоже плотными массивами.

Для того, чтобы не создавать копий (частей массива) можно использовать так называемые представления срезов. Представления срезов играют роль ссылок ссылкам, но ссылаются они не на массив целиком, а только на его часть. Создаются ссылки с

помощью специальной функции `view`, или с помощью макроса `@view` (макросом пользоваться удобнее). Например

```
Aj = @view(A[:, j]) # - пердставление среза j-го столбца
Ai = @view(A[i, :]) # - представление среза i-ой строки
```

Срезы представляют собой отдельный специальный тип данных, являющийся подтипом `AbstractVector` или `AbstractArray`:

```
julia> A
3×4 Matrix{Int64}:
 4  1  9  8
 5  8  5  7
 4  5  8  6

julia> @view(A[2, :]) |> typeof
SubArray{Int64,1, Matrix{Int64}, Tuple{Int64, Base.Slice{Base.OneTo{Int64}}},
true}

julia> @view(A[:, 2]) |> typeof
SubArray{Int64,1, Matrix{Int64},Tuple{Base.Slice{Base.OneTo{Int64}}, Int64},
true}

julia> (@view(A[:, 2]) |> typeof) <: Vector{Int}
false

julia> (@view(A[:, 2]) |> typeof) <: AbstractVector{Int}
true
```

Следует иметь в виду, что представления срезов эффективны **только если срез представляет непрерывную область памяти**. Например, `A[:, j]` - срез столбца представляет непрерывную область памяти, а `A[i, :]` - срез строки не представляет непрерывной области памяти, и потому не эффективен. Он будет медленнее обрабатываться в цикле и при передаче в функции, поскольку каждый доступ к элементу требует вычисления смещения.

Чтобы получить потный массив из представления среза нужно воспользоваться функцией `collect`, например

```
Aj_dense = collect(Aj) # получен полотный массив (если массив A - плотный)
Ai_dense = collect(Ai) # получен полотный массив (если массив A - плотный)
```

В результате из представления среза получается сам срез, т.е. копия части массива.

9. Векторизованные и поэлементные операции с массивами, вычисления на месте

Рассмотрим пример

```
a = collect(0:0.1:1) # - диапазон преобразован в плотный массив
b = sin.(a)          # - получен полотный массив синусов элементов маасива a
# Замечание: вместо sin тут могла бы быть любая другая числовая функция
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не
скопированы)
for k in 1:10_000
    c = 2a + k*b
end
```

В последнем выражении каждый результат векторизованных операций `2a` и `k*b` будет скопирован (помещен) в отдельный временный буфер, затем к этим временно сохраненным промежуточным массивам будет применена векторизованная операция `+`, полученный результат будет помещен где-то в памяти, и ссылка на этот массив будет записана в переменную c. Причем это будет происходить на каждой итерации цикла заново. Отметим, что при этом не имело ни какого значения, что перед началом цикла мы создали массив `c` - в данном случае результат получаемый на каждой итерации цикла каждый раз будет записываться в новую область памяти.

Эти операции размещения результатов в промежуточные буферы и создание нового массива под результат на каждой итерации являются затратными, и для оптимизации кода их требуется избегать.

Попытаемся это исправить, но будем вносить исправления поэтапно, постепенно повышая эффективность нашего кода.
Первое, что можно сделать, это заменить в теле цикла оператор `=` на `.=`

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не
скопированы)
for k in 1:10_000
    c .= 2a + k*b # оператор = заменен
end
```

Это приведет к тому, что теперь результат на каждой итерации будет помещать всегда в одну и ту же заранее выделенную область памяти.

Следующее, что можно сделать - это заменить векторизованные арифметически операции на не векторизованные, поставив перед каждой из них точку

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не
скопированы)
for k in 1:10_000
    c .= 2 .* a .+ k .* b
end
```

Теперь никаких промежуточных буферов и лишних копирований в них (аллокаций) происходить не будет, все вычисления буду производиться на месте массива `c`.

Полученный код - это просто сокращенная запись следующего кода

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не
скопированы)
for k in 1:10_000
    for i in echindex(c)
        c[i] = 2 * a[i] + k * b[i]
    end
end
```

Зададимся вопросом, что бы получилось, если бы мы здесь перед какой-либо арифметической операцией не поставили точку, например, - перед первой операцией умножения:

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не
скопированы)
for k in 1:10_000
    c .= 2a .+ k .* b
end
```

В этом случае для результата операции `2a` потребовался бы промежуточный буфер, обозначим его условно именем `buf`. И только после этого стало бы возможным реализовать соответствующие поэлементные операции

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не
скопированы)
for k in 1:10_000
```

```
    c .= buf .+ k .* b
end
```

Для удобства, и для того чтобы не пропускать расстановку точек в критически важных местах имеется специальный макрос `@.`:

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не скопированы)
for k in 1:10_000
    @. c = 2a + k * b
end
```

Этот макрос автоматически заменит данный код на требуемый

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не скопированы)
for k in 1:10_000
    c .= 2a .+ k .* b
end
```

Для примера рассмотрим еще следующий код

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не скопированы)
for k in 1:10_000
    c .+= 2a .+ k .* b
end
```

Он будет эквивалентен такому коду

```
c = similar(a)
# - не инициализированная копия масссива a (сами значения массива не скопированы)
for k in 1:10_000
    for i in echindex(c)
        c[i] += 2 * a[i] + k * b[i]
    end
end
```

10. Массивы и циклы, правильный порядок итерирования индексов

Поскольку массивы Julia в памяти компьютера размещаются по столбцам, то порядок итерирования по индексам во вложенных массивах имеет значение для эффективного использования кэша процессора. С этой точки зрения правильно программировать так, чтобы индексы строки менялись бы во внутреннем, а не в наружном цикле

```
for j in axex(A, 2) # используется итератор индексов столбцов
    for i in axes(A, 1) # используется итератор индексов строк
        A[i,j] = ...
    end
end
```

То же самое в Julia можно записать в краткой форме

```
for j in axex(A, 2), i in axes(A, 1)
        A[i,j] = ...
end
```

По умолчанию в Julia осуществляется контроль за выходом индексов за пределы массива. Поскольку этот контроль связан с выполнением дополнительных проверок, то он снижает производительность. Для отключения этого контроля используется специальный макрос
`@inbounds`:

```
@inbounds for j in axex(A, 2), i in axes(A, 1)
        A[i,j] = ...
end
```

Но отключать эту проверку целесообразно только после тщательного тестирования кода.

11. Переформатирование уже созданного массива

Если имеется некоторый массив A размеров m1 на n1, то его можно переформатировать в массив размера m2 на n2, из тех же элементов, при условии, конечно, что `m1*n1 == m2*n2`, с помощью функции `reshape`. Например,

```
    julia> A = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
```

```
 4  5  6

julia> size(A) |> println #(2,3)
(2, 3)

julia>     B = reshape(A, 3, 2) # - это то же самое, что и B = reshape(A, (3,
2))
3×2 Matrix{Int64}:
 1  5
 4  3
 2  6

julia>     size(B) |> println #(2,3)
(3, 2)
```

Важно понимать, что при таком переформатировании физическое размещение массива в памяти не изменяется; при этом тип переформатированного массива совпадает с типом исходного. При этом возвращается ссылка на тот же самый массив.

Кроме этого, имеется также функция `vec`, которая произвольный массив преобразует в вектор, возвращая прежнюю ссылку.

12. Транспонирование массива

Рассмотрим следующие примеры

```
julia> A=[1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> B=A' # - получена эрмитово сопряженная матрица
3×2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4
 2  5
 3  6

julia> typeof(B) # - получена транспонированная матрица
LinearAlgebra.Adjoint{Int64, Matrix{Int64}}

julia> transpose(A)
3×2 transpose(::Matrix{Int64}) with eltype Int64:
 1  4
 2  5
 3  6
```

```
julia> typeof(B)
LinearAlgebra.Adjoint{Int64, Matrix{Int64}}
```

В случае вещественных матриц операции эрмитова сопряжения и транспонирования совпадают. В случае комплексных матриц эрмитово сопряжение означает, что кроме транспонирования осуществляется комплексное сопряжение каждого элемента матрицы.

Важно отметить, что операция транспонирования матрицы не приводит к физическому перемещению элементов матрицы в памяти. Вместо этого, как можно видеть, создается специальный тип `LinearAlgebra.Adjoint` или LinearAlgebra.Transpose. Этот тип представляет собой просто обертку исходной матрицы, в которой содержится информация о том, как после транспонирования правильно интерпретировать индексы элементов массива. Например:

```
julia> A=zeros(2,3)
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> typeof(A')
LinearAlgebra.Adjoint{Float64, Matrix{Float64}}

julia> typeof(transpose(A))
LinearAlgebra.Transpose{Float64, Matrix{Float64}}
```

При этом операции эрмитова сопряжения и транспонирования возвращают ссылку на исходный массив.