DataFrames.md

> - 本文档的目标是翻译并总结 DataFrames.jl文件
> - version: DataFrames v0.21.4
> - Author: Andy.Yang
> - E-mail: yjd2008@hotmail.com

一、简介

DataFrames用来处理表格式数据（类似于Python中的Pandas），即每一列数据有相同的属性，不同列可以有不同的属性。

注：Excel，数据库(以下用SQL代替)也可以用来处理这样的数据。个人认为关系数据库中的每个表非常类似于DataFrames需要处理的数据排布。那么什么时候应该使用Excel，SQL？Excel的优势是明显，但是如果将提取出来的数据用作其它地方不方便，而且只能固定xls格式；SQL非常适用于大数据量的情况下，效率会比DataFrames高出很多，但是其体积较大。相比之下，DataFrames就比较适用于非固定格式、中小批量数据的分析处理、转化了。

二、安装

- 方法一：

```
julia> using Pkg
julia> Pkg.add("DataFrames")
```

- 方法二：

```
julia> ;
(@v1.5) pkg> add DataFrames
```

导入方法：

```
using DataFrames
```

以下均默认已正常安装，并且程序在REPL中测试，每行开头均已导入包

三、构造 DataFrame 类型

DataFrame 类型是由若干个向量构成的数据表，每一个向量对应于一列或变量。创建 DataFrame 类型最简单的方法是传入若干个关键字-向量对，如下所示：

```
julia> df = DataFrame(A = 1:4, B = ["M", "F", "F", "M"])
4×2 DataFrame
│ Row │ A     │ B      │
│     │ Int64 │ String │
├─────┼───────┼────────┤
│ 1   │ 1     │ M      │
│ 2   │ 2     │ F      │
│ 3   │ 3     │ F      │
│ 4   │ 4     │ M      │

# 构造空类型
julia》 df = DataFrame()

# 从具名元组(NamedTuples)构造
julia> v = [(a=1,b=2), (a=3,b=4)]
2-element Array{NamedTuple{(:a, :b),Tuple{Int64,Int64}},1}:
 (a = 1, b = 2)
 (a = 3, b = 4)

julia> df = DataFrame(v)
2×2 DataFrame
│ Row │ a     │ b     │
│     │ Int64 │ Int64 │
├─────┼───────┼───────┤
│ 1   │ 1     │ 2     │
│ 2   │ 3     │ 4     │
```

注：由上可看出，列名是一个变量名，每列的数据类型必须是一致的，首列代表着行号。

四、基本操作

4.1 列数据获取

注：列数据的索引方法随着版本的更新不断变化，《Julia数据科学应用》中的许多API已经无法弃用。

1. df.colName
2. df."colName"
3. df[: , :colName]
4. df[: , "colName"]
5. df[! , :colName]
6. df[! , "colName"]

```
julia> df.A
julia> df."A"
julia> df[ : , :A ]
julia> df[ : , "A" ]
julia> df[ ! , :A ]
julia> df[ ! , "A" ]
```

- 方法1,2,5,6并不copy数据，因此速度相对3,4较快，但是更改数据会影响最原始的数据
- 上述列名也可以直接用列的位置代替

4.2 增加列

```
julia> df.C=2:5
julia> df
4×3 DataFrame
| Row | A     | B      | C     |
|     | Int64 | String | Int64 |
|-----|-------|--------|-------|
| 1   | 1     | M      | 2     |
| 2   | 2     | F      | 3     |
| 3   | 3     | F      | 4     |
| 4   | 4     | M      | 5     |
```

4.3 在行末尾增加一行数据

注：这种方法性能较差，不太适用于大量的行数据插入

```
# 在行末尾增加一行数据
julia> push!(df,(1,"N",6))
5×3 DataFrame
| Row | A     | B      | C     |
|     | Int64 | String | Int64 |
|-----|-------|--------|-------|
| 1   | 1     | M      | 2     |
| 2   | 2     | F      | 3     |
| 3   | 3     | F      | 4     |
| 4   | 4     | M      | 5     |
| 5   | 1     | N      | 6     |

# 使用字典增加一行数据
julia> push!(df,Dict(:A=>5, :B=>"G", :C=>7))
6×3 DataFrame
| Row | A     | B      | C     |
|     | Int64 | String | Int64 |
|-----|-------|--------|-------|
| 1   | 1     | M      | 2     |
| 2   | 2     | F      | 3     |
| 3   | 3     | F      | 4     |
| 4   | 4     | M      | 5     |
| 5   | 1     | N      | 6     |
| 6   | 5     | G      | 7     |
```

4.4 打印所有列名

```
julia> names(df)
2-element Array{String,1}:
 "A"
 "B"

julia> propertynames(df)
2-element Array{Symbol,1}:
 :A
 :B
```

注:

- :colName 类型是 Symbol，"colName" 类型是 String。一般使用 Symbol比String更快。
- 对列重命名见 4.8.4 select语法

4.5 获得表的尺寸

```
# 返回表的行数
julia> size(df,1)
4
# 返回表的列数
julia> size(df,2)
3
# 返回表的尺寸
julia> size(df)
(4, 3)
```

4.6 数据的导入与导出

将 DataFrame 存储为 CSV，要先 add CSV

- 导入

```
# 从 CSV导入，input为文件路径
julia> DataFrame(CSV.File(input))
```

- 导出

```
julia> using CSV
julia> CSV.write("dataframe.csv", df)

# 将 DataFrame 存储为关系数据库中的表，要先 add SQLite
# 注意：首先要建立关系数据库
julia> SQLite.load!(df, db, "dataframe_table")
```

4.7 打印 DataFrame 中的数据

- 默认 df 根据屏幕大小打印若干行数据（并非所有）。如果需要打印所有数据，手动设置：

```
# 打印所有行
julia> show(df, allrows=true)

# 打印所有列
julia> show(df, allcols=true)
```

- 打印最开始或最后的若干行数据

```
# 打印起始的3行数据
julia> first(df, 3)

# 打印末尾的2行数据
julia> last(df, 2)
```

4.8 获取 DataFrame 数据的子集（筛选出一部分数据）

4.8.1 普通索引

```
# 获取1-3行，所有列的数据
julia> df[1:3, :]

# 获取第1,5,10行，所有列的数据
julia> df[[1, 5, 10], :]

# 获取所有行，A和B列的数据
julia> df[:, [:A, :B]]

# 获取1-3行， B和A列的数据，列的显示顺序按照索引的次序
julia> df[1:3, [:B, :A]]

# 获取第3, 1行，C列的数据
julia> df[[3, 1], [:C]]

# 使用view宏，并不返回一个copy
julia> @view df[1:3, :A]
```

注： df[!, [:A]] 和 df[:, [:A]] 返回的数据类型是DataFrame，而 df[!, :A] and df[:, :A] 返回的是一个向量

4.8.2 正则表达式、Not、All 索引

```
julia> df = DataFrame(x1=1, x2=2, y=3);
julia> df[!, r"x"]
1×2 DataFrame
| Row | x1    | x2    |
|     | Int64 | Int64 |
|-----|-------|-------|
| 1   | 1     | 2     |

julia> df[!, Not(:x1)]
1×2 DataFrame
| Row | x2    | y     |
|     | Int64 | Int64 |
|-----|-------|-------|
| 1   | 2     | 3     |

julia> df = DataFrame(r=1, x1=2, x2=3, y=4);
# 将所有列名包含字符x的移动到最前方
julia> df[:, All(r"x", :)]
1×4 DataFrame
| Row | x1    | x2    | r     | y     |
|     | Int64 | Int64 | Int64 | Int64 |
|-----|-------|-------|-------|-------|
| 1   | 2     | 3     | 1     | 4     |

# 将所有列名包含字符x的移动到最后方
julia> df[:, All(Not(r"x"), :)]
1×4 DataFrame
| Row | r     | y     | x1    | x2    |
|     | Int64 | Int64 | Int64 | Int64 |
|-----|-------|-------|-------|-------|
| 1   | 1     | 4     | 2     | 3     |
```

4.8.3 条件索引

```
# 索引出A列数据大于500的所有行和所有列数据
julia> df[df.A .> 500, :]

# 列A大于500 并且 列C在(300,400)之间的所有行和所有列数据
julia> df[(df.A .> 500) .& (300 .< df.C .< 400), :]

# 列A中数据等于1,5,601的所有行和所有列的数据
julia> df[in.(df.A, Ref([1, 5, 601])), :]
```

4.8.4 对每行数据进行处理

通过使用 select，select! 可以选择、重命名、变换列数据。 注：

- 使用select将会创建一个新的DataFrame变量。
- 变换列数据指的是使用一个函数对某列源数据进行处理

```
julia> df = DataFrame(x1=[1, 2], x2=[3, 4], y=[5, 6]);

# 丢弃 df 中的列 x1
julia> select(df, Not(:x1))
2×2 DataFrame
│ Row │ x2    │ y     │
│     │ Int64 │ Int64 │
├─────┼───────┼───────┤
│ 1   │ 3     │ 5     │
│ 2   │ 4     │ 6     │

# 选择 df 中所有包含字符 x 的列
julia> select(df, r"x")character
2×2 DataFrame
│ Row │ x1    │ x2    │
│     │ Int64 │ Int64 │
├─────┼───────┼───────┤
│ 1   │ 1     │ 3     │
│ 2   │ 2     │ 4     │

# 重命名列名
julia> select(df, :x1 => :a1, :x2 => :a2)
2×2 DataFrame
│ Row │ a1    │ a2    │
│     │ Int64 │ Int64 │
├─────┼───────┼───────┤
│ 1   │ 1     │ 3     │
│ 2   │ 2     │ 4     │

# 对列 x2 施加一个函数（减去本列数据中最小的数，局部函数变量x代表列向量），处理后的列名仍为 x2
julia> select(df, :x1, :x2 => (x -> x .- minimum(x)) => :x2)
2×2 DataFrame
│ Row │ x1    │ x2    │
│     │ Int64 │ Int64 │
├─────┼───────┼───────┤
│ 1   │ 1     │ 0     │
│ 2   │ 2     │ 1     │

# 对列 x2 中所有行数据求开方
julia> select(df, :x2, :x2 => ByRow(sqrt))
2×2 DataFrame
│ Row │ x2    │ x2_sqrt │
│     │ Int64 │ Float64 │
├─────┼───────┼─────────┤
│ 1   │ 3     │ 1.73205 │
│ 2   │ 4     │ 2.0     │
```

默认 select 会 copy 原始数据返回一个新的 DataFrame 变量，若要使用引用（想要修改源数据时），传递关键字 copycols=false 或使用 select!

```
julia> df2 = select(df, :x1, copycols=false);

julia> df2.x1 === df.x1
true

julia> df2 = select!(df, :x1);

julia> df2.x1 === df.x1
true
```

transform, transform! 和 select, select! 的功能类似，但是前两者会将源数据中的所有列显示在新的 DataFrame 变量中。

```
julia> df = DataFrame(x1=[1, 2], x2=[3, 4], y=[5, 6]);

# All() 对每行的所有数据执行函数： +
julia> transform(df, All() => +)
2×4 DataFrame
| Row | x1    | x2    | y     | x1_x2_y_+ |
|     | Int64 | Int64 | Int64 | Int64     |
|-----|-------|-------|-------|-----------|
| 1   | 1     | 3     | 5     | 9         |
| 2   | 2     | 4     | 6     | 12        |

# 使用 ByRow 可以返回每行中满足函数 argmax 的列名，并不是返回数据。
julia> transform(df, AsTable(:)=>ByRow(argmax)=>:prediction)
2×4 DataFrame
| Row | x1    | x2    | y     | prediction |
|     | Int64 | Int64 | Int64 | Symbol     |
|-----|-------|-------|-------|------------|
| 1   | 1     | 3     | 5     | y          |
| 2   | 2     | 4     | 6     | y          |

# 计算每行的和，个数，均值（都忽略missing）
julia> using Statistics

julia> df = DataFrame(x=[1, 2, missing], y=[1, missing, missing]);

julia> transform(df, AsTable(:) .=>
                     ByRow.([sum∘skipmissing,
                             x -> count(!ismissing, x),
                             mean∘skipmissing]) .=>
                     [:sum, :n, :mean])
3×5 DataFrame
| Row | x       | y       | sum   | n     | mean    |
|     | Int64?  | Int64?  | Int64 | Int64 | Float64 |
|-----|---------|---------|-------|-------|---------|
| 1   | 1       | 1       | 2     | 2     | 1.0     |
| 2   | 2       | missing | 2     | 1     | 2.0     |
| 3   | missing | missing | 0     | 0     | NaN     |
```

注：虽然可以使用上述语法进行简单的数据操作，但是从个人的SQL使用经验来看，更推荐安装使用包 Query、DataramesMeta 来进行 DataFrame 的数据处理，其语法更简洁方便。

4.8.5 对每列数据进行处理

- 直接使用 Statistics 包对某列数据处理

```
julia> using Statistics

julia> mean(df.A)
```

- 使用 combine 对每列数据进行处理

```
julia> df = DataFrame(A = 1:4, B = 4.0:-1.0:1.0);

julia> combine(df, names(df) .=> sum, names(df) .=> prod)
1×4 DataFrame
| Row | A_sum | B_sum   | A_prod | B_prod  |
|     | Int64 | Float64 | Int64  | Float64 |
|-----|-------|---------|--------|---------|
| 1   | 10    | 10.0    | 24     | 24.0    |
```

4.8.6 数据描述

使用 describe 函数可以返回一个 DataFrame 的部分统计学特征量。

```
julia> df = DataFrame(A = 1:4, B = ["M", "F", "F", "M"])

julia> describe(df)
2×8 DataFrame
| Row | variable | mean    | min | median | max | nunique | nmissing | eltype   |
|     | Symbol   | Union…  | Any | Union… | Any | Union…  | Nothing  | DataType |
|-----|----------|---------|-----|--------|-----|---------|----------|----------|
| 1   | A        | 2.5     | 1   | 2.5    | 4   |         |          | Int64    |
| 2   | B        |         | F   |        | M   | 2       |          | String   |

# 如果想要仅对某列数据进行处理，见如下语法
julia> describe(df[!, [:A]))
1×8 DataFrame
| Row | variable | mean    | min   | median  | max   | nunique | nmissing | eltype
|
|     | Symbol   | Float64 | Int64 | Float64 | Int64 | Nothing | Nothing  |
DataType |
|-----|----------|---------|-------|---------|-------|---------|----------|------
---|
```

```
| 1   | A        | 2.5     | 1     | 2.5     | 4     |         |          | Int64
|
```

4.8.9 替换数据

- 使用 replace! 替换修改源数据（仅能一列）

```
julia> df = DataFrame(a = ["a", "None", "b", "None"], b = 1:4, c = ["None", "j",
"k", "h"], d = ["x", "y", "None", "z"]);

# 将列 a 中的 None 替换为 c
julia> replace!(df.a, "None" => "c");df
4×4 DataFrame
| Row | a       | b     | c       | d       |
|     | String  | Int64 | String  | String  |
|-----|---------|-------|---------|---------|
| 1   | a       | 1     | None    | x       |
| 2   | c       | 2     | j       | y       |
| 3   | b       | 3     | k       | None    |
| 4   | c       | 4     | h       | z       |
```

- 使用 ifelse 可以替换多列数据

```
# 将列 c 和 d 中的 None 替换为 c
julia> df[:, [:c, :d]] .= ifelse.(df[!, [:c, :d]] .== "None", "c", df[!, [:c,
:d]]);df
4×4 DataFrame
| Row | a       | b     | c       | d       |
|     | String  | Int64 | String  | String  |
|-----|---------|-------|---------|---------|
| 1   | a       | 1     | c       | x       |
| 2   | c       | 2     | j       | y       |
| 3   | b       | 3     | k       | c       |
| 4   | c       | 4     | h       | z       |

# 将 df 中所有的 c 替换为 None
julia> df .= ifelse.(df .== "c", "None", df)
4×4 DataFrame
| Row | a       | b     | c       | d       |
|     | String  | Int64 | String  | String  |
|-----|---------|-------|---------|---------|
| 1   | a       | 1     | None    | x       |
| 2   | None    | 2     | j       | y       |
| 3   | b       | 3     | k       | None    |
| 4   | None    | 4     | h       | z       |
```

注：上面的 .= 会修改df, 如果用 = 则会生成一个新 DataFrame。 如果想用 missing 替换缺失值时并且原有的DataFrame列不允许用missing时，就必须使用 = 或者 allowmissing!

```
# 将 df 中的 None 用 missing 替换
julia> df2 = ifelse.(df .== "None", missing, df);

julia> allowmissing!(df);
julia> df .= ifelse.(df .== "None", missing, df);
```

五、多个DataFrame的连接组合

这里的操作类似于关系数据库中的 join 操作。

- innerjoin：包含键存在所有DataFrame中的值
- leftjoin：包含键存在于左侧的参数中，不管是否在第二（右侧）参数
- rightjoin:包含键存在于第二（右侧的参数），不管左侧
- outerjoin:包含任意一个键值
- semijoin：类似innerjoin，但是输出严格限制在左侧参数所在列
- antijoin：仅包含左侧，不包含右侧。输出仅左侧的键
- crossjoin：所有DataFrame的笛卡尔积

```
julia> people = DataFrame(ID = [20, 40], Name = ["John Doe", "Jane Doe"]);

julia> jobs = DataFrame(ID = [20, 60], Job = ["Lawyer", "Astronaut"]);

# innerjoin
julia> innerjoin(people, jobs, on = :ID)
1×3 DataFrame
| Row | ID    | Name     | Job    |
|     | Int64 | String   | String |
|-----|-------|----------|--------|
| 1   | 20    | John Doe | Lawyer |

# leftjoin
julia> leftjoin(people, jobs, on = :ID)
2×3 DataFrame
| Row | ID    | Name     | Job     |
|     | Int64 | String   | String? |
|-----|-------|----------|---------|
| 1   | 20    | John Doe | Lawyer  |
| 2   | 40    | Jane Doe | missing |

# rightjoin
julia> rightjoin(people, jobs, on = :ID)
2×3 DataFrame
| Row | ID    | Name     | Job       |
|     | Int64 | String?  | String    |
|-----|-------|----------|-----------|
| 1   | 20    | John Doe | Lawyer    |
| 2   | 60    | missing  | Astronaut |

# outerjoin
```

```
julia> outerjoin(people, jobs, on = :ID)
3×3 DataFrame
| Row | ID    | Name     | Job       |
|     | Int64 | String?  | String?   |
|-----|-------|----------|-----------|
| 1   | 20    | John Doe | Lawyer    |
| 2   | 40    | Jane Doe | missing   |
| 3   | 60    | missing  | Astronaut |

# semijoin
julia> semijoin(people, jobs, on = :ID)
1×2 DataFrame
| Row | ID    | Name     |
|     | Int64 | String   |
|-----|-------|----------|
| 1   | 20    | John Doe |

# antijoin
julia> antijoin(people, jobs, on = :ID)
1×2 DataFrame
| Row | ID    | Name     |
|     | Int64 | String   |
|-----|-------|----------|
| 1   | 40    | Jane Doe |

# crossjoin
julia> crossjoin(people, jobs, makeunique = true)
4×4 DataFrame
| Row | ID    | Name     | ID_1  | Job       |
|     | Int64 | String   | Int64 | String    |
|-----|-------|----------|-------|-----------|
| 1   | 20    | John Doe | 20    | Lawyer    |
| 2   | 20    | John Doe | 60    | Astronaut |
| 3   | 40    | Jane Doe | 20    | Lawyer    |
| 4   | 40    | Jane Doe | 60    | Astronaut |
```

如果要匹配的两列名不同，可以使用(left, right)或 left=>right 对应

```
julia> a = DataFrame(ID = [20, 40], Name = ["John Doe", "Jane Doe"]);

julia> b = DataFrame(IDNew = [20, 40], Job = ["Lawyer", "Doctor"]);

julia> innerjoin(a, b, on = :ID => :IDNew)
2×3 DataFrame
| Row | ID    | Name     | Job    |
|     | Int64 | String   | String |
|-----|-------|----------|--------|
| 1   | 20    | John Doe | Lawyer |
| 2   | 40    | Jane Doe | Doctor |

julia> a = DataFrame(City = ["Amsterdam", "London", "London", "New York", "New
York"],
```

```
                      Job = ["Lawyer", "Lawyer", "Lawyer", "Doctor", "Doctor"],
                      Category = [1, 2, 3, 4, 5]);

julia> b = DataFrame(Location = ["Amsterdam", "London", "London", "New York", "New
York"],
                      Work = ["Lawyer", "Lawyer", "Lawyer", "Doctor", "Doctor"],
                      Name = ["a", "b", "c", "d", "e"]);

# 列a的City与b的Location对应，Job与Work对应
julia> innerjoin(a, b, on = [(:City, :Location), (:Job, :Work)])
```

六、数据分割、组合

许多数据分析任务需要将数据分割成group，然后对每个group应用函数，并将结果组成起来，这种策略称之为：https://juliadata.github.io/DataFrames.jl/stable/man/split_apply_combine/。

julia 使用 groupby, combine, select/select!, transform/transform! 等函数完成这一策略。groupby(df, cols)将会返回一个 GroupedDataFrame 类型变量，针对每个group可以使用combine, select, transform 函数。

```
julia> using DataFrames, CSV, Statistics

julia> iris = DataFrame(CSV.File(joinpath(dirname(pathof(DataFrames)),
"../docs/src/assets/iris.csv")));

# gdf 是一个GroupedDataFrame类型变量，依照Species分类
julia> gdf = groupby(iris, :Species)

# 求每个group的均值
julia> combine(gdf, :PetalLength => mean)
3×2 DataFrame
| Row | Species         | PetalLength_mean |
|     | String          | Float64          |
|─────|─────────────────|──────────────────|
| 1   | Iris-setosa     | 1.464            |
| 2   | Iris-versicolor | 4.26             |
| 3   | Iris-virginica  | 5.552            |

# 求每个group的数量
julia> combine(gdf, nrow)
3×2 DataFrame
| Row | Species         | nrow  |
|     | String          | Int64 |
|─────|─────────────────|───────|
| 1   | Iris-setosa     | 50    |
| 2   | Iris-versicolor | 50    |
| 3   | Iris-virginica  | 50    |

# 求每个group的数量，PetalLength的均值，并将结果列重命名为mean
julia> combine(gdf, nrow, :PetalLength => mean => :mean)
3×3 DataFrame
| Row | Species         | nrow  | mean     |
```

```
|      | String           | Int64 | Float64 |
|------|------------------|-------|---------|
| 1    | Iris-setosa      | 50    | 1.464   |
| 2    | Iris-versicolor  | 50    | 4.26    |
| 3    | Iris-virginica   | 50    | 5.552   |

# 将多列作为函数参数传递
julia> combine([:PetalLength, :SepalLength] => (p, s) -> (a=mean(p)/mean(s),
b=sum(p)),
               gdf)
3×3 DataFrame
| Row | Species          | a        | b       |
|     | String           | Float64  | Float64 |
|-----|------------------|----------|---------|
| 1   | Iris-setosa      | 0.292449 | 73.2    |
| 2   | Iris-versicolor  | 0.717655 | 213.0   |
| 3   | Iris-virginica   | 0.842744 | 277.6   |

# AsTable将两列作为一个namedtuple变量x，属性是列名。
julia> combine(gdf,
               AsTable([:PetalLength, :SepalLength]) =>
               x -> std(x.PetalLength) / std(x.SepalLength)) # passing a
NamedTuple
3×2 DataFrame
| Row | Species          | PetalLength_SepalLength_function |
|     | String           | Float64                          |
|-----|------------------|----------------------------------|
| 1   | Iris-setosa      | 0.492245                         |
| 2   | Iris-versicolor  | 0.910378                         |
| 3   | Iris-virginica   | 0.867923                         |

julia> combine(x -> std(x.PetalLength) / std(x.SepalLength), gdf) # passing a
SubDataFrame
3×2 DataFrame
| Row | Species          | PetalLength_SepalLength_function |
|     | String           | Float64                          |
|-----|------------------|----------------------------------|
| 1   | Iris-setosa      | 0.492245                         |
| 2   | Iris-versicolor  | 0.910378                         |
| 3   | Iris-virginica   | 0.867923                         |

# 第一列求函数cor（应该是卷积）,第二列求数量
julia> combine(gdf, 1:2 => cor, nrow)
3×3 DataFrame
| Row | Species          | SepalLength_SepalWidth_cor | nrow  |
|     | String           | Float64                    | Int64 |
|-----|------------------|----------------------------|-------|
| 1   | Iris-setosa      | 0.74678                    | 50    |
| 2   | Iris-versicolor  | 0.525911                   | 50    |
| 3   | Iris-virginica   | 0.457228                   | 50    |
```

与combine不同，select和transform函数返回与源数据同样数量、次序的DataFrame对象。 注：个人理解 combine 是对列进行操作，而select和transform是对每行进行操作

```
# 求每行中列1-2的cor函数
julia> select(gdf, 1:2 => cor)
150×2 DataFrame
│ Row │ Species         │ SepalLength_SepalWidth_cor │
│     │ String          │ Float64                    │
├─────┼─────────────────┼────────────────────────────┤
│ 1   │ Iris-setosa     │ 0.74678                    │
│ 2   │ Iris-setosa     │ 0.74678                    │
│ 3   │ Iris-setosa     │ 0.74678                    │
│ 4   │ Iris-setosa     │ 0.74678                    │
│ 5   │ Iris-setosa     │ 0.74678                    │
│ 6   │ Iris-setosa     │ 0.74678                    │
│ 7   │ Iris-setosa     │ 0.74678                    │
⋮
│ 143 │ Iris-virginica  │ 0.457228                   │
│ 144 │ Iris-virginica  │ 0.457228                   │
│ 145 │ Iris-virginica  │ 0.457228                   │
│ 146 │ Iris-virginica  │ 0.457228                   │
│ 147 │ Iris-virginica  │ 0.457228                   │
│ 148 │ Iris-virginica  │ 0.457228                   │
│ 149 │ Iris-virginica  │ 0.457228                   │
│ 150 │ Iris-virginica  │ 0.457228                   │

# chop是将字符串的前head个字符和后tail个字符去掉
julia> transform(gdf, :Species => x -> chop.(x, head=5, tail=0))
150×6 DataFrame
│ Row │ Species         │ SepalLength │ SepalWidth │ PetalLength │ PetalWidth │
Species_function │
│     │ String          │ Float64     │ Float64    │ Float64     │ Float64    │
SubString…       │
├─────┼─────────────────┼─────────────┼────────────┼─────────────┼────────────┼──
──────────────┤
│ 1   │ Iris-setosa     │ 5.1         │ 3.5        │ 1.4         │ 0.2        │
setosa           │
│ 2   │ Iris-setosa     │ 4.9         │ 3.0        │ 1.4         │ 0.2        │
setosa           │
│ 3   │ Iris-setosa     │ 4.7         │ 3.2        │ 1.3         │ 0.2        │
setosa           │
│ 4   │ Iris-setosa     │ 4.6         │ 3.1        │ 1.5         │ 0.2        │
setosa           │
│ 5   │ Iris-setosa     │ 5.0         │ 3.6        │ 1.4         │ 0.2        │
setosa           │
│ 6   │ Iris-setosa     │ 5.4         │ 3.9        │ 1.7         │ 0.4        │
setosa           │
│ 7   │ Iris-setosa     │ 4.6         │ 3.4        │ 1.4         │ 0.3        │
setosa           │
⋮
│ 143 │ Iris-virginica  │ 5.8         │ 2.7        │ 5.1         │ 1.9        │
virginica        │
```

```
| 144 | Iris-virginica | 6.8         | 3.2        | 5.9         | 2.3         |
virginica        |
| 145 | Iris-virginica | 6.7         | 3.3        | 5.7         | 2.5         |
virginica        |
| 146 | Iris-virginica | 6.7         | 3.0        | 5.2         | 2.3         |
virginica        |
| 147 | Iris-virginica | 6.3         | 2.5        | 5.0         | 1.9         |
virginica        |
| 148 | Iris-virginica | 6.5         | 3.0        | 5.2         | 2.0         |
virginica        |
| 149 | Iris-virginica | 6.2         | 3.4        | 5.4         | 2.3         |
virginica        |
| 150 | Iris-virginica | 5.9         | 3.0        | 5.1         | 1.8         |
virginica        |
```

如果想要把数据分割为子集，使用groupby函数

```
julia> for subdf in groupby(iris, :Species)
           println(size(subdf, 1))
       end
50
50
50

# 使用pairs函数可得到每个group的列名
julia> for (key, subdf) in pairs(groupby(iris, :Species))
           println("Number of data points for $(key.Species): $(nrow(subdf))")
       end
Number of data points for Iris-setosa: 50
Number of data points for Iris-versicolor: 50
Number of data points for Iris-virginica: 50
```

索引GroupedDataFrame变量的方法：使用Tuple或NamedTuple

```
julia> df = DataFrame(g = repeat(1:3, inner=5), x = 1:15);

julia> gdf=groupby(df, :g);

julia> gdf[(g=1,)]
5×2 SubDataFrame
| Row | g     | x     |
|     | Int64 | Int64 |
|-----|-------|-------|
| 1   | 1     | 1     |
| 2   | 1     | 2     |
| 3   | 1     | 3     |
| 4   | 1     | 4     |
| 5   | 1     | 5     |
```

```
julia> gdf[[(1, ), (3,)]]
GroupedDataFrame with 2 groups based on key: g
First Group (5 rows): g = 1
| Row | g     | x     |
|     | Int64 | Int64 |
|-----|-------|-------|
| 1   | 1     | 1     |
| 2   | 1     | 2     |
| 3   | 1     | 3     |
| 4   | 1     | 4     |
| 5   | 1     | 5     |
⋮
Last Group (5 rows): g = 3
| Row | g     | x     |
|     | Int64 | Int64 |
|-----|-------|-------|
| 1   | 3     | 11    |
| 2   | 3     | 12    |
| 3   | 3     | 13    |
| 4   | 3     | 14    |
| 5   | 3     | 15    |
```

将一个函数应用到GroupedDataFrame的所有列上

```
julia> gd = groupby(iris, :Species);

# 所有列求均值
julia> combine(gd, valuecols(gd) .=> mean);

# 所有列求标准差，输出的列名仍是原来的列名
julia> combine(gd, valuecols(gd) .=> (x -> (x .- mean(x)) ./ std(x)) .=>
valuecols(gd))
```

七、重塑和透视数据 Reshaping and Pivoting Data

使用stack函数将数据从wide转换为long格式。 个人理解：原有DataFrame的每列均为数据，使用stack函数后，将指定的表名转换为新列variable，其数据存储在新列value中，只是将原来的数据的存储方向旋转90度。

文档中的例子太复杂了，这里使用 ?stack中的示例解释

```
julia> d1 = DataFrame(a = repeat([1:3;], inner = [4]),
                      b = repeat([1:4;], inner = [3]),
                      c = randn(12),
                      d = randn(12),
                      e = map(string, 'a':'l'));
12×5 DataFrame
| Row | a     | b     | c         | d         | e      |
|     | Int64 | Int64 | Float64   | Float64   | String |
|-----|-------|-------|-----------|-----------|--------|
```

```
| 1   | 1     | 1     | 0.490128  | -0.842285 | a     |
| 2   | 1     | 1     | -0.129096 | -1.81426  | b     |
| 3   | 1     | 1     | -1.26274  | 1.21582   | c     |
| 4   | 1     | 2     | -0.471777 | 0.209103  | d     |
| 5   | 2     | 2     | -1.14992  | 1.25682   | e     |
| 6   | 2     | 2     | 0.180661  | -1.01992  | f     |
| 7   | 2     | 3     | -0.297241 | -2.11296  | g     |
| 8   | 2     | 3     | 0.541566  | 1.74813   | h     |
| 9   | 3     | 3     | 1.06162   | -1.19485  | i     |
| 10  | 3     | 4     | -1.63669  | -0.677432 | j     |
| 11  | 3     | 4     | 0.487654  | 0.561469  | k     |
| 12  | 3     | 4     | 0.724918  | -1.20389  | l

# 使用列位置将列c和d stack
julia> d1s=stack(d1,3:4)
24×5 DataFrame
| Row | a     | b     | e      | variable | value     |
|     | Int64 | Int64 | String | Cat…     | Float64   |
|-----|-------|-------|--------|----------|-----------|
| 1   | 1     | 1     | a      | c        | 0.490128  |
| 2   | 1     | 1     | b      | c        | -0.129096 |
| 3   | 1     | 1     | c      | c        | -1.26274  |
| 4   | 1     | 2     | d      | c        | -0.471777 |
| 5   | 2     | 2     | e      | c        | -1.14992  |
| 6   | 2     | 2     | f      | c        | 0.180661  |
| 7   | 2     | 3     | g      | c        | -0.297241 |
| 8   | 2     | 3     | h      | c        | 0.541566  |
| 9   | 3     | 3     | i      | c        | 1.06162   |
| 10  | 3     | 4     | j      | c        | -1.63669  |
| 11  | 3     | 4     | k      | c        | 0.487654  |
| 12  | 3     | 4     | l      | c        | 0.724918  |
| 13  | 1     | 1     | a      | d        | -0.842285 |
| 14  | 1     | 1     | b      | d        | -1.81426  |
| 15  | 1     | 1     | c      | d        | 1.21582   |
| 16  | 1     | 2     | d      | d        | 0.209103  |
| 17  | 2     | 2     | e      | d        | 1.25682   |
| 18  | 2     | 2     | f      | d        | -1.01992  |
| 19  | 2     | 3     | g      | d        | -2.11296  |
| 20  | 2     | 3     | h      | d        | 1.74813   |
| 21  | 3     | 3     | i      | d        | -1.19485  |
| 22  | 3     | 4     | j      | d        | -0.677432 |
| 23  | 3     | 4     | k      | d        | 0.561469  |
| 24  | 3     | 4     | l      | d        | -1.20389  |


# 使用列名将列c和d stack，结果同上
julia> d1s=stack(d1,[:c,:d]);
```

上面两个参数的stack函数会将未stack的所有列给重复出来，如果仅想显示部分未stack的列，加上第3个参数即可。

```
julia> d1s2 = stack(d1, [:c, :d], [:a])
24×3 DataFrame
| Row | a      | variable | value     |
|     | Int64  | Cat…     | Float64   |
|-----|--------|----------|-----------|
| 1   | 1      | c        | 0.490128  |
| 2   | 1      | c        | -0.129096 |
| 3   | 1      | c        | -1.26274  |
| 4   | 1      | c        | -0.471777 |
| 5   | 2      | c        | -1.14992  |
| 6   | 2      | c        | 0.180661  |
| 7   | 2      | c        | -0.297241 |
| 8   | 2      | c        | 0.541566  |
| 9   | 3      | c        | 1.06162   |
| 10  | 3      | c        | -1.63669  |
| 11  | 3      | c        | 0.487654  |
| 12  | 3      | c        | 0.724918  |
| 13  | 1      | d        | -0.842285 |
| 14  | 1      | d        | -1.81426  |
| 15  | 1      | d        | 1.21582   |
| 16  | 1      | d        | 0.209103  |
| 17  | 2      | d        | 1.25682   |
| 18  | 2      | d        | -1.01992  |
| 19  | 2      | d        | -2.11296  |
| 20  | 2      | d        | 1.74813   |
| 21  | 3      | d        | -1.19485  |
| 22  | 3      | d        | -0.677432 |
| 23  | 3      | d        | 0.561469  |
| 24  | 3      | d        | -1.20389  |
```

使用Not关键字可将其余的列stack，如下所示：

```
julia> d1m = stack(d1, Not([:a, :b, :e]))
24×5 DataFrame
| Row | a      | b      | e      | variable | value     |
|     | Int64  | Int64  | String | Cat…     | Float64   |
|-----|--------|--------|--------|----------|-----------|
| 1   | 1      | 1      | a      | c        | 0.490128  |
| 2   | 1      | 1      | b      | c        | -0.129096 |
| 3   | 1      | 1      | c      | c        | -1.26274  |
| 4   | 1      | 2      | d      | c        | -0.471777 |
| 5   | 2      | 2      | e      | c        | -1.14992  |
| 6   | 2      | 2      | f      | c        | 0.180661  |
| 7   | 2      | 3      | g      | c        | -0.297241 |
| 8   | 2      | 3      | h      | c        | 0.541566  |
| 9   | 3      | 3      | i      | c        | 1.06162   |
| 10  | 3      | 4      | j      | c        | -1.63669  |
| 11  | 3      | 4      | k      | c        | 0.487654  |
| 12  | 3      | 4      | l      | c        | 0.724918  |
| 13  | 1      | 1      | a      | d        | -0.842285 |
| 14  | 1      | 1      | b      | d        | -1.81426  |
```

```
| 15  | 1     | 1     | c     | d     | 1.21582   |
| 16  | 1     | 2     | d     | d     | 0.209103  |
| 17  | 2     | 2     | e     | d     | 1.25682   |
| 18  | 2     | 2     | f     | d     | -1.01992  |
| 19  | 2     | 3     | g     | d     | -2.11296  |
| 20  | 2     | 3     | h     | d     | 1.74813   |
| 21  | 3     | 3     | i     | d     | -1.19485  |
| 22  | 3     | 4     | j     | d     | -0.677432 |
| 23  | 3     | 4     | k     | d     | 0.561469  |
| 24  | 3     | 4     | l     | d     | -1.20389  |
```

使用unstack函数可以将stacked的数据（long format）还原为原始数据，但是需要指定三列：id，variable, values。

```
# 将d1增加一列id
julia> d1.id=1:size(d1,1);

# 总共有4列stack，新生成的行数为：12*4=48
julia> longdf=stack(d1,Not([:id, :a]))
48×4 DataFrame
| Row | a     | id    | variable | value |
|     | Int64 | Int64 | Cat…     | Any   |
|-----|-------|-------|----------|-------|
| 1   | 1     | 1     | b        | 1     |
| 2   | 1     | 2     | b        | 1     |
| 3   | 1     | 3     | b        | 1     |
| 4   | 1     | 4     | b        | 2     |
| 5   | 2     | 5     | b        | 2     |
| 6   | 2     | 6     | b        | 2     |
| 7   | 2     | 7     | b        | 3     |
⋮
| 41  | 2     | 5     | e        | e     |
| 42  | 2     | 6     | e        | f     |
| 43  | 2     | 7     | e        | g     |
| 44  | 2     | 8     | e        | h     |
| 45  | 3     | 9     | e        | i     |
| 46  | 3     | 10    | e        | j     |
| 47  | 3     | 11    | e        | k     |
| 48  | 3     | 12    | e        | l     |

julia> widedf=unstack(longdf, :id, :variable, :value)
12×5 DataFrame
| Row | id    | b   | c         | d         | e   |
|     | Int64 | Any | Any       | Any       | Any |
|-----|-------|-----|-----------|-----------|-----|
| 1   | 1     | 1   | 0.490128  | -0.842285 | a   |
| 2   | 2     | 1   | -0.129096 | -1.81426  | b   |
| 3   | 3     | 1   | -1.26274  | 1.21582   | c   |
| 4   | 4     | 2   | -0.471777 | 0.209103  | d   |
| 5   | 5     | 2   | -1.14992  | 1.25682   | e   |
| 6   | 6     | 2   | 0.180661  | -1.01992  | f   |
```

```
| 7   | 7     | 3   | -0.297241 | -2.11296  | g   |
| 8   | 8     | 3   | 0.541566  | 1.74813   | h   |
| 9   | 9     | 3   | 1.06162   | -1.19485  | i   |
| 10  | 10    | 4   | -1.63669  | -0.677432 | j   |
| 11  | 11    | 4   | 0.487654  | 0.561469  | k   |
| 12  | 12    | 4   | 0.724918  | -1.20389  | l   |
```

八、排序

- 使用sort(df)将会从第一列开始排序，当左列数据相同时，从下一列开始排序。sort会创建一个新的DataFrame，使用sort!会修改源数据。
- 倒序：

```
julia> iris = DataFrame(CSV.File(joinpath(dirname(pathof(DataFrames)),
"../docs/src/assets/iris.csv")));

julia> sort!(iris, rev=true)
```

- 按某几列排序：

```
julia> sort!(iris, [:Species, :SepalWidth])
```

- Species列按长度排序，SepalLength列逆序排列。by意味着排序前先应用一个函数

```
julia> sort!(iris, (order(:Species, by=length), order(:SepalLength, rev=true)));
```

- 列名可以使用 :，或 All 或 Not 或 Between 或 Regex 进行筛选
- Species逆序，PetalLength正序

```
julia> sort!(iris, [:Species, :PetalLength], rev=(true, false));
```

九、分类数据

- v 的类型是 Array

```
julia> v = ["Group A", "Group A", "Group A", "Group B", "Group B", "Group B"]
6-element Array{String,1}:
 "Group A"
 "Group A"
 "Group A"
```

```
 "Group B"
 "Group B"
 "Group B"
```

- 使用 CategoricalArray生成cv, CategroicalArray也支持 missing 类型

```
julia> using CategoricalArrays;

julia> cv = CategoricalArray(v)
6-element CategoricalArray{String,1,UInt32}:
 "Group A"
 "Group A"
 "Group A"
 "Group B"
 "Group B"
 "Group B"

julia> cv = CategoricalArray(["Group A", missing, "Group A",
                              "Group B", "Group B", missing]);
```

- 使用levels可以查看所有的不同类

```
julia> levels(cv)
2-element Array{String,1}:
 "Group A"
 "Group B"
```

- 使用levels!可以改变类的排序

```
julia> levels!(cv, ["Group B", "Group A"]);

julia> levels(cv)
2-element Array{String,1}:
 "Group B"
 "Group A"
```

- 可以使用 compress 函数来节省存储空间

```
julia> cv = compress(cv);
```

- 直接使用categorical来创建CategoricalArray类型变量，并可使用关键字ordered、compress

```

```

```
julia> cv1 = categorical(["A", "B"], compress=true);

julia> cv2 = categorical(["A", "B"], ordered=true)

julia> cv2[1] < cv2[2]
true
```

- 使用isordered判断是否已排序，或使用ordered!来改变排序

```
julia> isordered(cv1)
false

julia> ordered!(cv1, true)
2-element CategoricalArray{String,1,UInt8}:
 "A"
 "B"

julia> isordered(cv1)
true
```

- 将DataFrame的某列（必须是AbstractString类型）类型变换为CategoricalArray

```
# 将df.A列类型变换
julia> categorical!(df, :A)

# 将 df 的所有列类型为AbstractString的变换
julia> categorical!(df, compress=true)
```

十、Missing 类型数据

- julia中缺失值的变量名是missing，typeof(missing)的结果是Missing。
- 可以通过 skipmissing(x) 来跳过 x 中的 missing 值进行数据处理

```
julia> x = [1, 2, missing]
3-element Array{Union{Missing, Int64},1}:
 1
 2
 missing

julia> sum(skipmissing(x))
3

julia> collect(skipmissing(x))
2-element Array{Int64,1}:
 1
 2
```

- 使用coalesce函数可以将missing值替换为其它值。注意"."表示替换x中的所有missing值

```
julia> coalesce.(x, 0)
3-element Array{Int64,1}:
 1
 2
 0
```

- dropmissing 和 dropmissing! 会移除所有包含missing值的行。前者返回一个新的DataFrame对象，后者修改源数据

```
julia> df = DataFrame(i = 1:5,
                      x = [missing, 4, missing, 2, 1],
                      y = [missing, missing, "c", "d", "e"]);

julia> dropmissing(df)
2×3 DataFrame
│ Row │ i     │ x       │ y       │
│     │ Int64 │ Int64?  │ String? │
├─────┼───────┼─────────┼─────────┤
│ 1   │ 4     │ 2       │ d       │
│ 2   │ 5     │ 1       │ e       │

# 仅移除列x中包含missing值的行
julia> dropmissing(df, :x)
3×3 DataFrame
│ Row │ i     │ x       │ y       │
│     │ Int64 │ Int64?  │ String? │
├─────┼───────┼─────────┼─────────┤
│ 1   │ 2     │ 4       │ missing │
│ 2   │ 4     │ 2       │ d       │
│ 3   │ 5     │ 1       │ e       │

# 把包含missing值的列属性从 Union{T, Missing} 修改为 T
julia> dropmissing(df, disallowmissing=true)
2×3 DataFrame
│ Row │ i     │ x     │ y       │
│     │ Int64 │ Int64 │ String  │  注意：此行的类型不再包含 ？
├─────┼───────┼───────┼─────────┤
│ 1   │ 4     │ 2     │ d       │
│ 2   │ 5     │ 1     │ e       │
```

- Missings包可以提供更多实用的函数，例如

```
julia> using Missings

# replace提供跟coalesce同样的功能
julia> collect(Missings.replace(x, 1))
3-element Array{Int64,1}:
 1
 2
 1

julia> collect(Missings.replace(x, 1)) == coalesce.(x, 1)
true

# missings函数可以生成指定尺寸的 missing Array
julia> missings(1, 3)
1×3 Array{Missing,2}:
 missing  missing  missing

julia> missings(Int, 1, 3)
1×3 Array{Union{Missing, Int64},2}:
 missing  missing  missing
```

十一、外部的数据处理语法框架（强烈推荐使用）

两种常用操作 DataFrame 的框架是： DataFramesMeta.jl 和 Query.jl。它们可以提供类似 dplyr 和 LINQ的语法。 这两个包的安装方法和详细操作见对应包的文档。

11.1 DataFramesMeta.jl

```
julia> using DataFrames, DataFramesMeta

julia> df = DataFrame(name=["John", "Sally", "Roger"],
                      age=[54., 34., 79.],
                      children=[0, 2, 4]);

# 筛选 age 大于40的行，输出新列名为 number_of_childeren（原children列），name
julia> @linq df |>
           where(:age .> 40) |>
           select(number_of_children=:children, :name)
2×2 DataFrame
| Row | number_of_children | name   |
|     | Int64              | String |
|-----|--------------------|--------|
| 1   | 0                  | John   |
| 2   | 4                  | Roger  |

julia> df = DataFrame(key=repeat(1:3, 4), value=1:12)
12×2 DataFrame
| Row | key   | value |
|     | Int64 | Int64 |
|-----|-------|-------|
```

```
| 1   | 1     | 1     |
| 2   | 2     | 2     |
| 3   | 3     | 3     |
| 4   | 1     | 4     |
| 5   | 2     | 5     |
| 6   | 3     | 6     |
| 7   | 1     | 7     |
| 8   | 2     | 8     |
| 9   | 3     | 9     |
| 10  | 1     | 10    |
| 11  | 2     | 11    |
| 12  | 3     | 12    |

julia> @linq df |>
           where(:value .> 3) |>
           by(:key, min=minimum(:value), max=maximum(:value)) |>
           select(:key, range=:max - :min)
3×2 DataFrame
| Row | key   | range |
|     | Int64 | Int64 |
|-----|-------|-------|
| 1   | 1     | 6     |
| 2   | 2     | 6     |
| 3   | 3     | 6     |

julia> @linq df |>
           groupby(:key) |>
           transform(value0 = :value .- minimum(:value))
12×3 DataFrame
| Row | key   | value | value0 |
|     | Int64 | Int64 | Int64  |
|-----|-------|-------|--------|
| 1   | 1     | 1     | 0      |
| 2   | 1     | 4     | 3      |
| 3   | 1     | 7     | 6      |
| 4   | 1     | 10    | 9      |
| 5   | 2     | 2     | 0      |
| 6   | 2     | 5     | 3      |
| 7   | 2     | 8     | 6      |
| 8   | 2     | 11    | 9      |
| 9   | 3     | 3     | 0      |
| 10  | 3     | 6     | 3      |
| 11  | 3     | 9     | 6      |
| 12  | 3     | 12    | 9      |
```

11.2 Query.jl

query操作起始于宏@from，第一个参数i代表着 df 中每一行来执行query命令，df代表数据源，@where执行过滤（filter）指令，@select指令将源数据的列映射为新的列，后方接的是 "{}"，这代表的是一个具名元组（namedtuple)；@collect决定了返回的数据结构类型,当没有这一项时，返回的是一个julia标准的迭代数据类型，collect后面不接类型时，返回一个Array

```
julia> using DataFrames, Query

julia> df = DataFrame(name=["John", "Sally", "Roger"],
                      age=[54., 34., 79.],
                      children=[0, 2, 4])
3×3 DataFrame
| Row | name   | age     | children |
|     | String | Float64 | Int64    |
|-----|--------|---------|----------|
| 1   | John   | 54.0    | 0        |
| 2   | Sally  | 34.0    | 2        |
| 3   | Roger  | 79.0    | 4        |

julia> q1 = @from i in df begin
            @where i.age > 40
            @select {number_of_children=i.children, i.name}
            @collect DataFrame
       end
2×2 DataFrame
| Row | number_of_children | name   |
|     | Int64              | String |
|-----|--------------------|--------|
| 1   | 0                  | John   |
| 2   | 4                  | Roger  |

# 无@collect项
julia> q2 = @from i in df begin
                  @where i.age > 40
                  @select {number_of_children=i.children, i.name}
              end;

julia> total_children = 0
0

julia> for i in q2
           global total_children += i.number_of_children
       end

julia> total_children
4

julia> y = [i.name for i in q2 if i.number_of_children > 0]
1-element Array{String,1}:
 "Roger"

# 返回一个 Array类型
julia> q3 = @from i in df begin
            @where i.age > 40 && i.children > 0
            @select i.name
            @collect
       end
1-element Array{String,1}:
 "Roger"
```