Exercices Julia

TD Interfaces et Traits

16 juin 2021

Ce TD traitera de l'utilisation des interfaces dans Julia et touchera au sujet des Traits.

1 Rappels

Pour rappel, les *fonctions* dans Julia sont très généralement des objets qui attribuent à un tuple d'arguments une valeur de retour ou sortent une exception. Pour une fonction concrète plusieurs implémentations pour différentes arités et types du tuple d'arguments sont possibles : les *méthodes* de la fonction.
Donc quand on parle de *fonction* on va parler de la fonction en tant qu'objet avec un certain nom et un objectif général. Quand on parle de *méthode*, c'est une implémentation concrète d'une fonction pour un tuple d'arguments qui satisfait certaines conditions sur l'arité et les types.

2 Interfaces

Il arrive souvent de créer un nouveau type qui partage certaines caractéristiques avec des structures fréquemment utilisées. Prenons p-ex un type `MyMesh` qui décrit un maillage polygonal d'un objet (un mesh). Un autre package que vous utilisez propose un autre type `PlatoPolyhedron` qui permet de décrire les solides de Platon avec une représentation interne basique.

```julia
using StaticArrays

struct MyMesh{T}
    vertices::Vector{SVector{3,T}}
    faces::Vector{SVector{3,S}} where {S <: Integer}
end

struct PlatoPolyhedron
    n::Int
    side::Float
end
```

On voudrait écrire une fonction qui vérifie une certaine propriété pour tous les sommets du solide. Si les méthodes nécessaires existent pour ces deux types très différents, on pourrait écrire un code comme suit :

```julia
using LinearAlgebra

function coïncide(solide,point; rtol=1e-8)
    for i in solide
        if norm(i-point)<rtol
            return true
        end
    end
    return false
end
```

Notons que ici, `solide` peut être n'importe quelle représentation du solide. La fonction repose sur le fait que l'écriture `for i in solide` va attribuer à `i` les positions des sommets de `solide`. Bien que cette information soit facile à retrouver dans un objet de type `MyMesh`, c'est beaucoup plus difficile pour un objet du type `PlatoPolyhedron`. C'est l'interfaçage entre le type et l'itération qui cache cette complexité. Cette façade abstraite que l'on utilise lorsque l'on écrit `for i in solide` s'appelle une *interface*. Vous pouvez obtenir plus d'informations sur ce sujet dans le **Manuel**. Nous allons surtout parler de l'interface *iterateur* mais il en existe plusieurs autres qui permettent p-ex de gérer la manière dont le broadcasting avec le point `.` marche sur votre type.
Comme beaucoup d'autres choses dans Julia, les interfaces reposent sur le mécanisme de dispatch de Julia. Concrètement, les interfaces sont une collection de fonctions de base sur lesquelles sont construites beaucoup de méthodes génériques. Il suffit d'implémenter les méthodes de ces fonctions de base pour notre type afin de permettre de le traiter d'une manière type-agnostique.
Le but de ces exercices est d'implémenter plusieurs nouveaux types et des interfaces correspondantes.

2.1 Itérateur de Fibonacci

But de la section 1

L'interface *iterateur* permet de parcourir l'objet élément par élément (le sens du mot élément est à la discretion du programmeur écrivant le type et ses méthodes).

Un itérateur (objet qui implémente l'itération) permet de parcourir l'objet en se souvenant de l'état de l'itération en dehors de l'objet (sans le muter). L'état de l'itération est stocké dans le code qui effectue l'itération. Le passage d'un état au suivant s'effectue en appelant la fonction `élément, état_suivant = iterate(objet,état)` qui prend l'objet à parcourir et un état d'itération et retourne l'élément actuel de l'objet ainsi que l'état suivant. Des appels successifs de cette fonction permet de parcourir l'objet entier.

Fun fact 1

Selon le manuel de Julia, c'est exactement sur ce principe que marche la boucle `for`. Le code d'une boucle `for` est transformé comme suit :

```
for i in iter
    #body
end
```

devient :

```
next = iterate(iter)
while next !== nothing
    (i, state) = next
    # body
    next = iterate(iter, state)
end
```

Cette interface est utile dès lors qu'un objet présente une propriété de collection, d'ensemble ou de suite. C'est pour cela que l'exemple que nous allons implémenter est celui de la suite de Fibonacci.
Par souci de rigueur, rappelons la définition de la suite de Fibonacci que l'on va utiliser :

$$F_1 = 0$$
$$F_2 = 1$$
$$F_{n+2} = F_n + F_{n+1}$$

Rappelons que l'itérateur ne modifie pas en général l'état de l'objet sur lequel il itère (cf. cours précédent). On va pour l'instant considérer une sous-suite de Fibonacci finie : les N premiers éléments de la suite de Fibonacci infinie. Le type `Fibonacci` qui représente cet objet ne doit donc comporter qu'une seule information qui ne sera pas modifiée : le nombre d'éléments. Il y a deux moyens immédiats qui se présentent pour stocker cette information. Soit le type est paramétrisé par une valeur soit il contient un champ dans lequel on peut stocker cette valeur. Les deux possibilités sont présentées ci-dessous (notez que l'on paramétrise aussi `Fibonacci` avec un type `T` qui servira à indiquer le type des éléments de la suite comme p-ex `Int64` , `BigInt`, ...) :

```
1  struct Fibonacci_1{T,Val{N}} end
2
3  struct Fibonacci_2{T}
4      N
5  end
```

On va dans le reste du sujet supposer que la variante `Fibonacci_2` a été choisie et l'on va l'appeler `Fibonacci`. Un objet de ce type est donc non mutable et ne contient comme information que le nombre d'éléments dans la suite. C'est donc à la méthode `Base.iterate(iter::Fibonacci, state)` de parcourir la suite. Ici, le type ne joue donc que le rôle d'un marqueur pour le dispatch. C'est la fonction `iterate()` qui contient la logique de la suite.

Sujet de l'exercice 1

Implémentez les méthodes suivantes :

```
1  struct Fibonacci{T}
2      N
3  end
4
5  Base.iterate(F::Fibonacci) = #<ECRIRE>
6  Base.iterate(F::Fibonacci, state) = #<ECRIRE>
```

Les fonctions devraient retourner `nothing` lorsqu'il n'y a pas d'élément suivant. Notez que la variable `state` doit contenir toute l'information nécessaire pour pouvoir calculer l'état suivant et savoir quand s'arrêter.
Vérifier que l'itération fonctionne.

Afin de permettre quelques fonctionnalités supplémentaires (p-ex `collect(iter)`), il est nécessaire de définir quelques méthodes supplémentaires qui donneront les propriétés de notre type quant à son comportement d'itérateur.

Fun fact 2

La déclaration des propriétés d'un type peut être faite de manière native permettant d'intégrer la propriété directement au système de dispatch : c'est ce que l'on appelle *Traits*.
L'utilisation des *traits* pourra être revue plus tard, mais au fond ils correspondent très simplement à une fonction `propriété(::Type)` qui prend le type que l'on veut décrire et retourne une valeur dont le type décrit la valeur de la propriété. Pour chaque type, une méthode de la fonction **`propriété`** peut être définie avec différentes valeurs de retour décrivant différentes propriétés. Pour pouvoir dispatcher sur le type d'un type (puisqu'on envoie à la fonction un type, le dispatch se produit sur le type de ce type), un type spécial paramétrique est utilisé tel que le type par lequel il est paramétrisé en soit une instance. Ainsi, p-ex : `MyType isa Type{MyType}` va toujours être vrai.
Ainsi on peut donner des propriétés quelconques à des types :

```
struct IsInteger end
struct NotInteger end

is_an_integer(::Type{T}) where {T<:Integer}=IsInteger()
is_an_integer(::Type{T}) where {T<:AbstractFloat}=NotInteger()

mypow(x::T,n) where T = pow(is_an_integer(T),x,n)
mypow(::IsInteger, x, n) = begin
    r=x
    for i in 1:n r*=x end
    r
end
mypow(::NotInteger, x, n) = throw("Not integer power")
```

Ici, `is_an_integer(Real)` va retourner **`NotInteger()`** de type **`NotInteger`**. Cette propriété est incorporée au système de types ce qui permet d'effectuer un dispatch basé sur les propriétés du type au lieu de remplacer le dispatch avec des conditions. La fonction `mypow()` va dispatcher vers une des deux méthodes définies sans que l'on ait à écrire de conditionnels.

Sujet de l'exercice 2

Écrire les fonctions

```
Base.IteratorSize(::Type{Fibonacci}) = Base.HasLength() # Déclare que la
#suite est finie
Base.IteratorEltype(::Type{Fibonacci}) = Base.HasEltype() # Déclare que le
#type de la suite est connu

Base.length(iter::Fibonacci) = #<ECRIRE>
Base.eltype(iter::Fibonacci{T}) where {T} = #<ECRIRE>
```

Vérifier que il est possible d'appeller `collect(Fibonacci{Int64}(20))`

Vous pouvez désormais tester quelques fonctions telles que `sum()`, `mean()`, `std()`... (ces deux dernières provenant du package `Statistics`).
Finalement nous allons rajouter la possibilité d'avoir une suite infinie. Nous allons pour cela introduire un type `FibonacciInf`. Sa fonction `iterate()` ne va donc jamais retourner `nothing`.

Sujet de l'exercice 3

Implémentez les méthodes nécessaires pour faire fonctionner l'itération sur la suite de Fibonacci infinie.
L'itérateur étant infini, la fonction `Base.length()` ne nécessite pas de méthode pour ce type. La nature infinie de l'itérateur fait que la fonction `Base.in(x,iter)` qui vérifie la présence d'un élément ne saura s'arrêter. Il faut donc la redéfinir pour ce type. Vous pouvez utiliser la nature croissante de la suite.

```
struct FibonacciInf{T} end

Base.iterate(iter::FibonacciInf{T}) where {T} = #<ECRIRE>
Base.iterate(iter::FibonacciInf, state) = #<ECRIRE>

Base.IteratorEltype(::Type{Fibonacci}) = Base.HasEltype()
Base.IteratorSize(::Type{FibonacciInf}) = Base.IsInfinite()

Base.eltype(iter::FibonacciInf{T}) where {T} = #<ECRIRE>
Base.in(x,iter::FibonacciInf) = #<ECRIRE>
```

Vérifiez que `221123640630391454569941296974487399338795698862 3 ∉ FibonacciInf{BigInt}()`.

2.2 Accès par index

But de la section 2

Implémenter l'accès direct par indice à un objet d'un nouveau type.

Une autre interface utile est celle de l'accès par index. Elle est utile pour les objets dont le calcul du n-ème élément est rapide. Comme exemple on peut utiliser toute suite numérique dont les termes peuvent être calculés directement. Autre exemple, nous pouvons munir de cette interface type `MyMesh` vu au début de cette interface, ce qui permettrait d'accéder directement à la position du `i`-ème sommet avec `mesh[i]` (p-ex pour effectuer des calculs en utilisant les sommets de manière non séquentielle.
Pour permettre l'accès via les crochets, nous devons définir la méthode `getindex()`.

Sujet de l'exercice 4

Implémentez un type ainsi qu'un accès via des crochets. Pour une indexation a N indices vous pouvez utiliser `Base.getindex(objet::MyType,I::Vararg{Int, N}) where N = #fonction`. Si vous n'avez pas d'idées pour le type, vous pouvez implémenter un type qui stocke un mesh et un accès par indexe qui permet d'accéder aux facettes triangulaires.

3 Traits

On va finir par un exercice portant sur les Traits (voir cours préenregistré si publié ou sinon Fun Fact **2** pour un résumé semi-lisible.
Les Traits ne sont pas une construction spéciale, mais plutôt une conséquence du système de types de Julia et du système de dispatch.
Illustrons les Traits par un exemple qu'il faudra compléter. Considérons le code suivant qui mets en place quelques types de façon similaire à l'exercice sur les itérateurs :

```
1  abstract type Serie end
2  struct FibonacciInf{T}<:Serie end
3  struct ArithmeticSeries{T, Offset}<:Serie end
4  struct GeometricSeries{T, Coeff}<:Serie end
5  struct ColbachSeries{T, Initial}<:Serie end
```

On va tenter d'écrire la méthode `Base.in(item, serie::T) where {T<:Serie}`. On considère que tous ces types implémentent les itérateurs correspondants. On note que les séries représentées sont pour toutes des séries infinies donc la méthode de `Base.in()` doit être écrite de manière à savoir s'arrêter en temps fini.
L'argument qui mène à l'utilisation des Traits est la manière similaire de gérer ce problème d'arrêt pour différentes séries similaires. Ainsi pour une série croissante on pourrait vérifier que l'élément recherché ne dépasse pas la valeur itérée. On voudrait donc ne pas écrire le même code à plusieurs reprises pour chaque type concret, mais plutôt écrire une seule fois la méthode qui cherche un élément dans un itérateur croissant et indiquer au système de dispatch qu'il faudrait utiliser cette méthode pour certaines séries.
Un moyen naturel de le faire serait de créer un type abstrait `abstract type SérieCroissante<:Serie` et définir les séries croissantes comme sous-types de ce type. Puis la méthode de `in()` a pour signature plutôt `Base.in(item, serie::T) where {T<:SerieCroissante}`. Mais cette méthode est intrusive et peu flexible. On voudra probablement écrire d'autres fonctions sur le type `Serie` pour lesquelles la situation se reproduira et on va devoir rajouter de plus en plus de profondeur à l'arbre de sous-types de `Serie`.
C'est ici que les Traits font leur apparence. On va introduire un type singleton `struct Croissante end`. Ce type est une réponse possible à la question "Quelle est la monotonité de la série?"
On formalise la question par la définition d'une fonction `Monotonité()` qui prend un type (p-ex le type `FibonacciInf` et retourne la réponse. Pour permettre de définir une méthode sur un type, on utilise un type paramétrique spécial `TypeMonType` qui crée un type abstrait dont le *type* `MonType` est une instance. Ainsi pour associer aux types définis ci-dessus diverses propriétés de monotonité, on définit simplement des méthodes de la fonction `Monotonité()` comme suit :

```julia
struct Croissante end
struct DeCroissante end
struct NonMonotone end
Monotonité(::Type{FibonacciInf{T}}) where T = Croissante()
Monotonité(::Type{ArithmeticSeries{T, Offset}}) where {Offset,T} = begin
    if Offset<0
        DeCroissante()
    end
    return Croisssante()
end
Monotonité(::Type{GeometricSeries{T, Coeff}}) where {Coeff,T} = begin
    if Coeff < 0
        return NonMonotone()
    elseif Coeff < 1
        return DeCroissante()
    end
    return Croissante()
end
Monotonité(::Type{ColbachSeries{T,Initial}}) where {Initial,T} = NonMonotone()
```

Vous pouvez vérifier qu'en appelant `Monotonité(GeometricSeries{Float64,0.2})` vous obtenez bien le résultat prévu.
On a donc muni certains types d'une propriété. Implémentons désormais les méthodes nécessaires de la fonction `Base.in()` avec un système qui va diriger les suites croissantes vers une implémentation qui ne vas pas finir en boucle infinie. Pour cela on va bien évidemment utiliser le multiple dispatch de Julia :

```julia
Base.in(item, serie::T) where {T<:Serie} = my_in(Monotonité(T), item, serie)

my_in(::Croissante, item, serie) = # code de la fonction

my_in(::DeCroissante, item, serie) = # code de la fonction

my_in(::NonMonotone, item, serie) = throw("Non monotone, boucle infinie possible")
```

Vu que le résultat de la fonction `Monotonité()` sur le type de la série est une instance d'un des types-propriétés, le système de dispatch va utiliser la bonne méthode de `my_in()`. On vient de créer et utiliser un Trait.

Sujet de l'exercice 5

Créer un exemple fonctionnel de Traits avec un dispatch sur une propriété. Si vous n'avez pas d'idées d'exemples vous pouvez implémenter les méthodes `in` pour un ou deux types de série proposé, N'hésitez pas à modifier la propriété et utiliser quelque chose d'autre que la monotonité si cela vous tente.
Vous pouvez aussi utiliser des Traits paramétriques, c'est-à-dire la réponse à la question sur la propriété peut parfaitement bien contenir plus d'information sous la forme de paramètres du type de réponse.