入門編

第1章 Julia のインストールと開発

1-1. Julia の特徴

本章では、早速 Julia の世界にご案内したいと思います。まずは Julia の特徴を紹介し、Julia というプログラミング言語のイメージをつかんでもらいます。それから、Julia をインストールして実際に簡単なプログラムを実行できるところまでの導入のお手伝いをします。

まずは、いくつかのキーワードを元に、Julia の様々な特徴を紹介していきます。

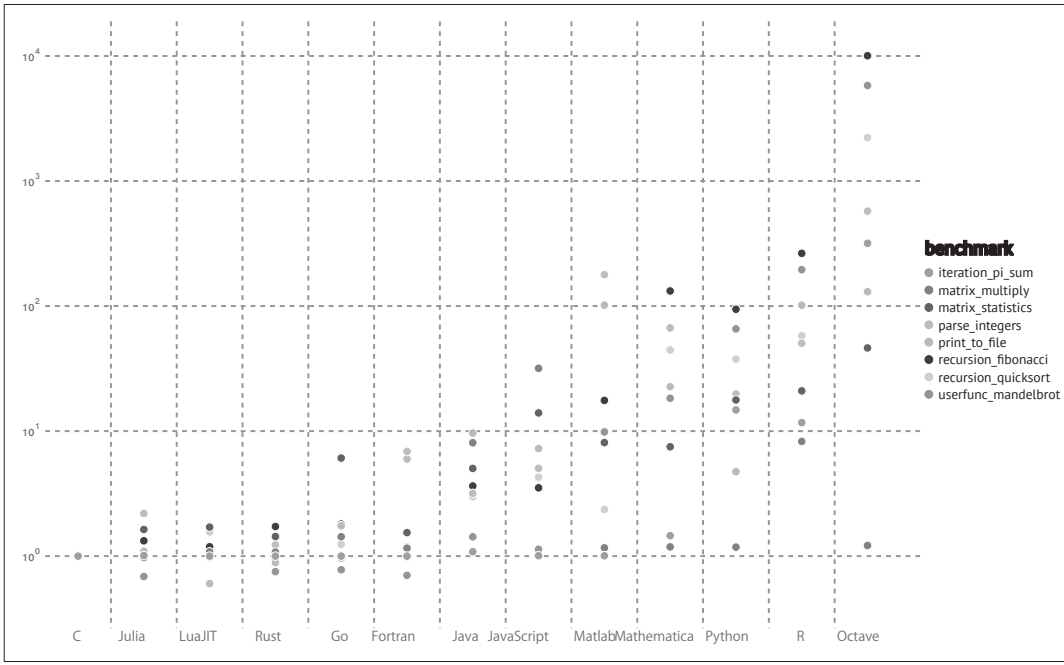
1-1-1. Julia は高速！

Julia を紹介する上で欠かせない第一の特徴は、「Julia は速い！」ということです。

図1-1. は、公式サイトで紹介されているベンチマーク結果のグラフです。関数呼び出し、文字列解析、ソート、数値ループ、乱数生成、再帰、配列操作などの一般的なコードパターンの範囲でコンパイラの性能をテストしたものとなっています。C言語の処理時間を 1 とした時間比であり、点の位置が低いほど高速であることを示しています。

Python・R・MATLAB 等の、主に数値計算や科学技術計算の分野でよく比較対象となる言語と比較すると、Julia の高速性が際立つ結果となっています。例えばフィボナッチ数の再帰計算では Python と比較して約50倍速となっています（片対数グラフであることにも注目してください！）。

▼図1-1. Julia Micro-Benchmarks (https://julialang.org/benchmarks/ より)



筆者もある特殊な状況下でのベンチマークを計測し結果を比較したことがあります。その時の結果が 表1-1. のようになります。ある数学計算をするコード（関数）を Julia/Python/Haskell の3言語で記述し、実行時間を比較した表となっています。

▼表1-1. ある数学計算の実行時間比較（単位：秒）

	Julia	Python	GHCi	GHC
ex1	1.78	91.67	189.05	1.44
ex2	2.63	20.66	174.26	1.40

表1-1. の Julia の実行時間は、以下の方法で REPL にて計測したものです（仮想コード1-1.）（REPL については本章の「1-3. Julia を REPL で使用する」で概説します）。`@time` は、実行時間を計測する **マクロ** です。結果は標準出力に出力されます（実行時間以外にメモリ使用量やコンパイル時間の割合が表示されることもあります）。

▼仮想コード1-1. Julia における `@time` マクロを利用した実行時間計測例

```
julia> @time ex1();
  1.783576 seconds

julia> @time ex2();
  2.629585 seconds (320.55 k allocations: 16.377 MiB, 66.23% compilation time)
```

なお、GHC/GHCi は Haskell の処理系であり、GHCi はインタプリタ（REPL）環境（`:set +s` を利用して時間を計測）、GHC はコンパイルしたバイナリ（`import Data.Time` を利用して時間を計測）のそれぞれの実行結果

です。また Python は `time` モジュールを利用して時間を計測しています。

これらを比較すると以下のようなことが分かります。

- Julia は（静的）コンパイル言語並みに速い！
- Julia は REPL で実行しても十分に速い！

もちろんこれらのベンチマークが全てではありません（あくまで『マイクロベンチマーク』です）が、Julia の大きな特徴としての 高速性 は概観していただけたと思います。特にここで紹介したように Julia には `@time` という処理時間を計測する（≒ Julia の高速性を体感できる）手軽で便利な手段が提供されています。本書でも随所に登場してきますし、読者の皆さんも積極的に使っていきましょう！

さて、Julia が速い理由はいくつかありますが、大きな要因は以下の3点だと言えます。

- JIT コンパイル
- 型推論
- 徹底的な最適化の実施

Julia は LLVM コンパイラフレームワークによる JIT コンパイルを行ってから実行する仕組みを採っています。**JIT（Just In Time）コンパイル**（日本語で **実行時コンパイル** とも呼ばれます）とは、文字通り「実行時に（今まさにその機能を利用しようとしたその時に）その場でコンパイルする」という機能のことです。

Julia の JIT コンパイルは主に関数単位で行われます。その関数が最初に実行されようとしたときに JIT コンパイルが実施され、結果のバイトコード（ネイティブコード）が実行されます。その結果はキャッシュされ、2回目以降に同じ関数を実行しようとしたときはそのコンパイル済のネイティブコードが直接実行されます。つまり基本的には高速に動作するネイティブコードを実行するので、これが Julia が（静的コンパイル言語並みに）速い一番の理由となっています。

それだけでなく、Julia は JIT コンパイル時（正確には JIT コンパイルの直前）に **型推論** と **コード最適化** を実施しています。

型推論について詳細は後述しますが、高速性のコンテキストで簡単に説明すると、JIT コンパイル前にコンパイル対象コード内の変数の型が確定するので、最適なコードにコンパイルされる大きな助けとなり高速に動作するバイトコードが生成される、という仕組みです。

その最適化については、Julia は多段階的に徹底して最適化を実施しています。JIT コンパイル時にも「CPUやLLVMに用意されている命令が利用できるときは利用する」「関数呼び出しをインライン化してオーバーヘッドを減らす」といったコンパイル言語によくある最適化はもちろん実施されます。しかし Julia ではその前段階でも可能な限り最適化が実施されます。ものすごく単純な例としては、コード中に `1 + 2` という計算式があったら、JIT コンパイル前に `3` という計算結果の値に変換されます（実行されるバイトコードでは足し算は実施されません）。型推論と絡めた例を挙げると、例えば `Int(n)` というコードがあったとき、変数 `n` の型が型推論の結果 `Int` と確定していたら、JIT コンパイル前の段階でただの `n` というコードになります（型変換コードは生成されません）。

このように最適化によって無駄な計算や関数呼び出し等を可能な限りなくしたコードが生成されます。それがコンパイルされ実行されるのです、高速に動作しないわけがありません！　これは Julia が高速性を **貪欲に** 追求した結果と言えます。

1-1-2. Julia は動的！

1-1-1. で何度か「Julia は静的コンパイル言語並みに速い」という言葉を使いました。

Julia は **動的言語** と呼ばれる部類に属します。これは端的に言い換えると、**静的コンパイル言語ではないプログラミング言語** です。

例えば Python や R、また Ruby などは動的言語です。これらはいずれも、ソースコードを直接読み込んでプログラムを実行します。実行時にソースコードをその場で解析・解釈しながら実行するタイプの動的言語は **インタプリタ言語** とも呼ばれます。ここに挙げた Python・R・Ruby はいずれもインタプリタ言語です。

これに対して **静的コンパイル言語** というのは、事前にソースコードを解析・変換（＝コンパイル）して実行形式バイナリ（または中間形式ファイル）を生成して実行する言語のことを指して言います。C 言語/C++ や、Java・C#、Rust・Go、また先ほど例に挙げた Haskell などは代表的な静的コンパイル言語です。

Julia は動的言語の中でも、静的コンパイル言語に対して **動的コンパイル言語** という言い方ができます。基本的には Python のようにソースコードを読み込んで実行しますが、それを解釈しながら実行するのではなく、1-1-1. で説明したとおり JIT コンパイルを行って得られたバイトコード（ネイティブコード）を実行します（ですので Julia はインタプリタ言語**ではありません**）。この時中間形式ファイル等を出力せず、バイトコードはメモリ上に配置されて直接実行されます（ですので Julia は静的コンパイル言語**でもありません**）。

Julia がインタプリタ方式を採用していないのは、1-1-1. で述べたとおり「高速に動作する！」を実現するためです。そして静的コンパイルを採用しなかったのは、「はじめに」で紹介した開発者ブログ記事の言葉、「C のように高速だけど、Ruby のようなダイナミズム（＝**動的性**）を併せ持っていてほしい」を実現するためです。

ではその動的性とは何でしょうか？　その主な特長は、以下のように説明できます。

- コードの追加や機能の拡張が行いやすい。
- インタラクティブ環境（REPL 等）との相性が良い。
- 実行環境に特化した最適化が実施可能。

このうち最初の「機能拡張のしやすさ（拡張性）」については後述します。

インタラクティブ性は、動的言語（特に **スクリプト言語** と呼ばれる類のもの）に元々求められている性質です。つまり「その場でコードを入力して、その場で実行されてすぐに結果が得られる」という即時性です。これは静的コンパイル言語でも不可能ではありません（例えば 1-1-1. で紹介したように Haskell にもインタプリタ環境（GHCi）があります）が、見た目にはインタラクティブに動作しても裏側では静的コンパイルが走って中間ファイルが生成されていたり、もしくはコンパイラの生成するコードをエミュレートするようなインタプリタ方式になっていたりして、必ずしも効率の良いものとは限らないのです（1-1-1. の GHCi のベンチマークを見てみてください）。

以上は Julia に限らず多くの動的言語（インタプリタ言語を含む）に言える特徴ですが、動的コンパイル言語ではさらに、その動的性を活かした最適化処理が実施できることも大きな特徴です。1-1-1. でも最適化処理の一例を挙げましたが、動的性のコンテキストで別の例を挙げると、動的コンパイル時（例えばコンパイル単位の関数等を実行しようとしたとき）に、現在の実行環境を知ることができ、CPU アーキテクチャや使用可能なメモリ容量などの情報を知ることができます。それに合わせて最適な CPU 命令や効率の良いメモリ割当処理コードを選択してネイティブコードを生成し実行することができるのです。これは静的コンパイル言語には困難な挙動です。Julia の JIT コンパイラは、この動的性をフルに活かした最適化が実施されるようになっています！

1-1-3. Julia は動的型付け！

Julia は前節で紹介した『動的言語』という意味での動的性（ダイナミズム）以外に、もう一つの動的性を持っています。それは **動的型付き言語** である、ということです。

動的型付け とは実行時の値によって型が決まる性質を指し、そのような性質を持つ言語のことを動的型付き言語と呼びます。Julia は変数宣言時に型の指定は不要で、ソースコード中でどのような型の値も変数に代入ができます。ただし実行時には全ての値にはある **具象型**[注1] が定まっています。typeof() 関数を用いると値に対する具象型を調べることができます。

▼コード1-1. typeof() 関数による具象型の調査

```
julia> typeof(1)  # 環境によって結果が異なることがあります。
Int64

julia> typeof(9.99)
Float64

julia> typeof("123ABCあいう漢字")
String
```

それに対して実行前（コンパイル時など）に型が決まる性質を **静的型付け**、そのような性質を持つ言語を **静的型付き言語** と呼び、コンパイル時の型検査により実行時に余分なエラーチェックをしなくて済むなどの利点があります。

Julia は動的型付き言語ですが、実は静的型付けの機能・特徴も取り入れています。主なものは以下の2点にまとめられます。

- 動的型付けだけど構造的に設計された型システムを持っている
- 動的型付けなのに型推論が実行される

Julia は構造的に設計された **型システム** を持っています。具象型の他に **抽象型** という概念もあり、型の親子関係（**基本型**-**派生型**）もあります（**サブタイピング** と言います）。これにより、例えば「数値」型の下に「整数」型と「浮動小数点数」型、「整数」型の下に「符号付き整数」と「符号なし整数」等と言ったような **型階層** を持つことができます。また他言語の総称型（ジェネリクス型）にあたる **パラメトリック型** という型パラメータを必要とする型もあり、それらを利用した **型制約** という仕様もあります。これらの仕様や仕組みにより構造的なプログラミング設計が可能となっています。

▼コード1-2. Julia の型システム概観

```
julia> Int64 <: Signed <: Integer <: Number  # 型階層
true

julia> isconcretetype(Int64)  # 具象型であることの確認
true
```

注1　本書では英語の術語 Concrete Type の訳語として 具象型 を用います。第4章でも紹介しています。

```
julia> isabstracttype(Number)  # 抽象型であることの確認
true

julia> typeof([1, 2, 3])  # パラメトリック型の例
Vector{Int64} (alias for Array{Int64, 1})

julia> typeof([1, 2, 3]) <: Vector{<:Integer}  # 型制約
true
```

静的型付き言語ですとコンパイル時の型検査のことを考えてこのようなしっかりとした型システムを持つことも多いのですが、Julia でも言語仕様としてこのようなしっかりした型システムが組み込まれているのです。

型推論 とは、変数や関数の引数に型が宣言されていなくても、渡ってくる値やその周辺のコードを見て各変数の型を推論する機能です。静的型付き言語でよく使われ、C++ でも auto と宣言された変数に対して型推論によってコンパイル時（コンパイル前）に型が特定されます。

動的型付き言語、かつ動的コンパイル言語である Julia では、JIT コンパイルの直前にこの型推論が実施されます。推論対象となるのは、コンパイル対象となった関数（等）の内部で使用されている各変数に代入される値と、その関数自体の戻り値です。型推論を実施する目的は、1-1-1. で述べた通り型が確定することでコード最適化の大きな助けになるためです。

Julia の型システムの詳細は 第4章 で解説します。

1-1-4. Julia は多重ディスパッチ！

Julia の最も大きな特徴（と言っても過言ではないもの）は、**多重ディスパッチ** という機構を備えていると言うことです。

多重ディスパッチとは、引数の組合せ（型シグニチャ）の違いによって多重定義された関数を、実行時の引数情報によって適切な実装実体（これを **メソッド** と呼びます）を選択して実行する仕組みのことです。つまりこれは、Julia の型システムと動的性を十二分に活かした機構です。

多重ディスパッチに対して、Python を初め多くの言語が備えているのは **単一ディスパッチ** です。これは引数のうちの1つだけが特別に扱われて、それによって呼び出す関数（メソッド）が決まる仕組みです。多くのオブジェクト指向言語ではそのメソッドを特定する対象をレシーバと呼ぶこともあり、メソッドとはそのレシーバオブジェクトに属するもの、という考え方が一般的です。

Julia（などの多重ディスパッチを備えた言語）では、メソッドは関数に属するという考え方になります。つまり特定の引数によってメソッドが決まるのではなく、その関数にどのような引数（の組合せ）を適用するかによってメソッドが決まります。

多重ディスパッチの特長としては、例えば以下のような点が挙げられます。

- 言語仕様（特に型定義周り）がシンプルになる。
- プログラム設計・API 設計がシンプルになる。
- 機能追加がしやすい。

1つめの「言語仕様がシンプルになる」というのは、型定義（≒データ構造）に着目したときにそれを振る舞い（関数）定義と分離できるので定義がスッキリし可読性が上がる、という点を指します。

2つめは今度は関数定義の方に着目した特長で、単一の関数に大まかな「振る舞い」を役割として与えて機能を集約し、「こういう目的ならこの関数を使うことを覚えておけば大抵OK」という状況を作れる、ということです。

3つめの機能追加は、1-1-2. で触れた「既存の型に新しい関数を適用する」「既存の関数に新しい型を適用させる」の両方を指します（後者は先ほど述べた「2つめの特長」とも関連します）。単一ディスパッチ（Python やその他のオブジェクト指向言語を想像してください）では、これらのうちどちらかは実現が難しいか、実現方法が単純ではありません（そのためだけに新しいクラスを設計・実装しなければならなくなるなど）。Julia では単純です。新しい関数を定義する、あるいは新しい型を定義する、それだけなのですから。

多重ディスパッチについては 第5章 で詳しく解説します。

1-1-5. 直感的な記述！

Julia の文法は、数多くの他言語から影響を受けておりそれらとの類似点が多いため、他のプログラミング言語を触ったことがあるなら関数定義や制御構文等の習得は容易です。

2つだけ例を挙げておきます（コード1-3.）。

▼コード1-3. 一般的で直感的な記述のコード例

```
julia> numbers = [1, 2, 3]

julia> function add(x, y)
           x + y
       end
```

前者は **配列**（1次元配列）の定義です。他言語でもよく見かけるおなじみの、値をカンマ区切りで並べて大括弧 `[〜]` で囲う書式です。分かりやすいですね。

後者は関数定義です。予約語 `function` で始まり、関数名・引数列が続き、その後関数本体を記述する行が続き（この例では1行だけですが）、最後に予約語 `end` で終わります。Julia では `return` 文 は不要で、最後に実行した式の値が戻り値になります。この例は「引数 `x` と `y` を受け取って、`x + y` という計算した結果を返す関数 `add`」を定義しています。簡単ですね。

さらに Julia は、色々な言語の「いいとこ取り」をしています。例えば関数や行列を数学の数式記述によく似た記法で定義できます（主に MATLAB 由来の記法です）。

$$f(x) = x^2 - 2x + 1$$

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

は以下のように記述できます（コード1-4.）。

▼コード1-4. 数式に似た直感的な記述のコード例

```
julia> f(x) = x^2 - 2x + 1

julia> A = [1 2
            3 4]
```

先ほどは関数を `function add(x, y) 〜 end` のように定義しましたが、この例のように `f(x) = 〜` というように、数式ほぼそのままの形で書くことも可能です。両者の定義方法の違いに機能面での差異はありません。それより注目すべきは、`x^2` が `x` の二乗、`2x` が `2 * x` という掛け算を表している[注2]、ということ。見た目そのままなので分かりやすいですね！

また後者の例は行列ですが、配列（1次元配列）と同じ大括弧 `[〜]` を使い、数値を空白区切りで並べているだけです。こちらも見た目そのままで記述できて、とても分かりやすいですね！

数式のような記述という意味ではもう一つの特徴として、Unicode 文字を識別子として利用できる（利用しやすい）という点も上げられます。例えば以下のような変数名をよく見かけます（仮想コード1-2.）。

▼仮想コード1-2. 変数（識別子）として有効な Unicode 文字の例

```
θ
Fₙ
σ²
μ
x̄
```

それぞれ「角度」「ある数列の一般項」「分散」「平均（値）」「平均（変数）」として数式でよく使われます。

Unicode 文字を識別子として使える言語は他にもありますが、Julia の Unicode 識別子サポートはさらに一歩二歩進んでいます。具体的には以下のような特徴があります。

- Tab 補完による入力サポートがある。
- 一部の Unicode 文字を演算子として利用できる（オリジナル定義もできる）。

Tab 補完は、Julia の REPL や各種エディタの Julia 対応プラグインにより、例えば先ほどの例は以下のように入力できます。

- `\theta` + Tab（⇒ θ）
- `F\_n` + Tab（⇒ F_n）
- `\sigma` + Tab + `\^2` + Tab（⇒ σ^2）
- `\mu` + Tab（⇒ μ）
- `x\bar` + Tab（⇒ x̄）

この Tab 補完ルール、基本的には TEX の記述ルールに従っています[注3]。そのルールを知っていれば大抵の文字は入力できますし、REPL のヘルプ機能を使えば入力方法の支援もしてくれます。例えば π について調べてみましょう（コード1-5.）。

注2　乗算で演算子 * を省略できるのは条件があります。詳細は第2章で紹介します。

注3　公式マニュアル（オンライン）の Unicode Input（https://docs.julialang.org/en/v1/manual/unicode-input）も参照してください。

▼コード1-5. REPLのhelpモード（πという定数の調査）の例

```
help?> π
"π" can be typed by \pi<tab>

search: π

  π
  pi

  The constant pi.

  Examples
  ==========

  julia> pi
  π = 3.1415926535897...
```

π は既に円周率を表す定数として定義されているので、その情報も出てきますが、最初の行に「"π" can be typed by \pi<tab>（\pi + Tab とタイプすれば π が入力できます）」という説明が出力されていますね！

今、既に定義されている定数（π）の例を挙げましたが、いくつかの Unicode 演算子も既に定義されています。例を挙げると、÷、∈、≤、≥、≡ など。分かりやすいのは ÷ でしょうか、これは整数除算演算子です。すなわち例えば `10 ÷ 2 == 5` です。

定義されているもの以外にも演算子として利用可能な Unicode 文字が数多くあります。例えば適切に定義すれば `A ⊗ B` という演算（テンソル積）も記述できます。

このように Julia では、言語レベルで積極的に「Unicode 識別子を使っていきましょう」という文化を形成してさえいます。

1-1-6. 真のマクロ！

Julia は強力な **メタプログラミング** 機構を備えています。メタプログラミングとは、簡潔に言うと「プログラムによってプログラム自体を記述すること」です。Java や Ruby に見られるメタクラスや動的メソッド、C++ や Haskell にあるテンプレートという機構はメタプログラミングの一種です。

Julia のメタプログラミング機能の中で特筆すべきものは **マクロ** です。マクロとは、プログラムの実行前（コンパイル言語の場合はコンパイル前）にコードの一部を書き換えて機能追加をしたり新しいコードを追加したりする機能のことです。先ほどの C++ や Haskell のテンプレートもマクロの一種と見ることもできます。

Julia のマクロの特徴は以下のような点が挙げられます。

- 通常のプログラミングと同じ書き方（同じ文法）で記述できる。
- テキストベースではなく「プログラム構造」そのものを変換できる。

まずはマクロの定義例を見てみましょう（コード1-6.）。

▼コード1-6. マクロの定義例

```
macro time_ns(ex)
    ex = esc(ex)
    quote
        t0 = time_ns()
        val = $ex
        t1 = time_ns()
        println("elapsed time: ", Int(t1-t0), " nanoseconds")
        val
    end
end
```

Julia のマクロは、予約語 macro で始める以外は（一般的な）関数定義と全く同じ書式になっています。マクロの中で他の関数を呼び出したりもできます。

次にこのマクロの使い方を見てみましょう（仮想コード1-3.）。

▼仮想コード1-3. マクロの実行例（ex1() という関数を定義しないと動作しません）

```
julia> ex1()
11

julia> @time_ns ex1()
elapsed time: 1782868700 nanoseconds
11

julia> @time ex1()
  1.783576 seconds (162.24 k allocations: 8.645 MiB, 61.34% compilation time)
11
```

定義したマクロは前に @ を付けて @time_ns 〜 のようにして実行します。この例は「実行時間をナノ秒単位で計測して表示するマクロ」というわけです。

はい。1-1-1. で紹介したマクロ **@time** を「車輪の再発明」してみた、というわけです。@time の使用例も併載しておきました。こちらの方がメモリ使用量等も表示できるのでより便利ですが、同じような機能があの数行のコードで実現できると考えれば、その有用性はお分かりいただけるのではないでしょうか？

ポイントは「マクロ呼び出し時のコード（ex1()）の前後にコードを追加して時刻計測と表示を実施するコードに変換している」ということ。このように通常のプログラムと同じ書き方でプログラムの構造そのものを変換できるマクロを **真のマクロ** と呼びます。

マクロを含むメタプログラミングについての詳細は、第8章で解説します。

1-1-7. 並行・並列プログラミング！

通常のプログラミングでは、プログラムは「上から順番に実行」されます。

少し詳しく言い換えると、プログラムというのは基本的には「ソースコードに書かれた文（式）が書かれた順序で逐次実行」されます。実際には、if などの条件分岐によって実行されたりされなかったりするコードブロックが存在したり、for や while によるループで何度も実行されるコードブロックが存在したり、関数によって全く別のコードブロックに処理が飛んだりすることもありますが、それらが記述された文法に従って「順番に」「逐次」

実行されるということには変わりありません。

これに対して、**並行計算・並列計算** という言葉があります。細かいニュアンスの違いはあります（というよりどちらの用語を使用すべきかのコンテキストにそもそも違いがあります）が、ここでは深く考えないでおきましょう。これらを誤解を恐れずに一言で説明すると以下のようになります。

- 複数の計算を同時に実行すること！

Julia にはこの並行・並列計算の仕組みも最初から組み込まれています。簡単な例を挙げましょう。以下のような関数を考えます（コード1-7.）。

▼コード1-7. 処理に時間のかかる関数の例（フィボナッチ数を計算する再帰関数）

```
function fib(n)
    if n ≤ 1
        n
    else
        fib(n - 2) + fib(n - 1)
    end
end
```

これは整数 n を引数に受け取り、n 番目のフィボナッチ数を計算して返す関数です。先ほど「1-1-5. 直感的な記述！」で説明したとおり、Julia では return 文がなくても関数内で最後に実行した式の値が関数の戻り値になります。さらに if 〜 end も式（if式 と言います）であり、内部で最後に実行（評価）した式の値が if式 の値（よって結果的にこの関数の戻り値）になります。

実装に何も工夫していないので、大きな n を与えると非常に時間がかかります（fib(100) などは生きているうちに答えが返ってこないかもしれません！）。取り敢えず fib(40) くらいで動かしてみましょう。@time マクロを使って計算時間と使用メモリ量も表示してみます（コード1-8.）。

▼コード1-8. 処理時間計測例（1）：そのまま実行

```
julia> @time println(fib(40))
102334155
  0.511058 seconds (11 allocations: 320 bytes)
```

約0.5秒で、正しい答えが表示されました。これを4回繰り返してみましょう。for 文を使って以下のように実行してみます（コード1-9.）。

▼コード1-9. 処理時間計測例（2）：for 文で4回実行

```
julia> @time for _=1:4
           println(fib(40))
       end
102334155
102334155
102334155
102334155
  2.050395 seconds (44 allocations: 1.250 KiB)
```

実際に実行してみると分かるのですが、答えである 102334155 が大体0.5秒おきにぽんぽんぽんぽんと表示されます。

実行時間（計算時間）の方は約4倍になりました。メモリ使用量もちょうど4倍になっている模様です。println(fib(40)) が4回実行（計算）された合計だと考えれば、当たり前だと思うかもしれません。

では、少し変えて以下のように実行してみましょう[注4]（コード1-10.）。

▼コード1-10. 処理時間計測例（3）：for 文で4回実行（マルチスレッド化）

```
julia> @time Threads.@threads for _=1:4
           println(fib(40))
       end
102334155
102334155
102334155
102334155
  0.690362 seconds (50.09 k allocations: 3.015 MiB, 12.10% compilation time)
```

実行時間が短くなりました！　答えの方も、一瞬の間があってその後ばばばばっと連続して表示された事と思います。

この例だと "12.10% compilation time" と表示されていますね。表示された時間から 12.1% 減らすと 約0.62秒。単独で実行したときが 約0.51秒 であることを考えるとその差は 0.11秒。1回だけ実行するのとあまり変わらない所要時間で4回同じ計算がされたことになります。これは文字通り、fib(40) という計算が4つ **同時に** 実行されたためです。

コード中に記述された Threads.@threads は、for 文を簡単に **マルチスレッド化** してくれる便利なマクロです。これにより、for ブロック内のコードを複数のスレッドで「同時に走らせる」処理（**非同期処理** と言います）、「それらが全部終わるまで待つ」処理（**同期処理** と言います）などが実行前（動的コンパイル前）に挿入されます。時間が短くなった一方でメモリ使用量は大幅に増えていますが、これはこれらの挿入された処理にメモリを余分に要しているためです。実行時間が1回実行時より少し多いのも同じ理由ですね。

このようにすこし記述を追加するだけで、逐次計算（コードを記述された順に一つずつ順に実行して計算すること）を並行・並列計算に変更することができます。これを本書では **並行・並列プログラミング** と呼ぶことにします。Julia には上に記述した方法の他にも様々な並行・並列プログラミングのための機能・仕様が用意されています。それは以下の3つのカテゴリーに分類することができます。

- タスク（コルーチン）による非同期処理
- スレッド並列
- マルチプロセシング・分散計算

先ほど挙げた例はこのうちの **スレッド並列** に該当します。スレッドという機能・仕様は他言語にもありますが、例えば Python のスレッドは完全な並列動作はしません。

Python はグローバルインタプリタロック（以下、GIL と略します）を採用しています。これはスレッドセーフでないコードを他のスレッドと共有してしまうことを防ぐ排他ロックのことで、1プロセスごとに1つのGILが存

注4　これを実行する前に、julia -t 4 のように Julia 起動時に -t オプションを指定する必要があります。

在します。これにより Python のスレッドで同じコードを実行しようとしても、他のスレッドが実行中は実行待ち状態となり、そのスレッドが実行完了したら他のスレッドが実行開始できる、という動作になります。つまり全然同時実行できてない状態になるのです。Python で完全並列動作を実現したい場合には、通常マルチプロセシングを採用することになります。

一方で Julia は GIL を採用していいません。このため、スレッドは許される限り個別のCPU（コア）に割り当てられて並列動作します（このことを指して **真のスレッド並列** と呼ぶことがあります）。もちろんマルチプロセシングの機構も用意されているので、プログラムの規模でこれらを使い分けることもできます。

ここでは触れられなかった内容も含め、並行・並列プログラミングについての詳細は第9章で解説します。

1-1-8. 組み込みパッケージマネージャ！

モダンなプログラミング言語の多くは、所謂 **外部パッケージ** を追加インストールすることで機能拡張・機能追加することができます。多くの言語では、このパッケージ（言語によってはモジュールという用語を使います）を、システムレベルで利用（その言語環境全体で共通して利用）するか、プロジェクトレベルで利用（現在行っているプログラミング環境でのみ利用）するかを選べるものもあります。後者に関連して、そのプログラミング環境を他とは独立した仮想環境として扱う機構、もしくはそれを文字通り「プロジェクト」という単位で管理できる機構を備えたものも存在します。

例えばPythonでは、PyPIというパッケージ管理リポジトリが存在し、`pip` というコマンドでパッケージのインストール・アップデートの等が行えます。ただしこれは基本的には Python のプログラミング環境全体に作用しますし、仮想環境やプロジェクトの管理をする機構は用意されていません。そのためそれを補完する他のツール（パッケージマネージャ等）が後から開発・発表されてきました（`virtualenv`、`pipenv`、`poetry` 等）が、乱立とまでは行かなくても状況がなかなか安定しなかったりします。またここに挙げたものの殆どは、Python をインストールしただけでは使えず何らかの方法で別途追加インストールする必要があります。

Julia には、その全てを統合的に扱える多機能なパッケージマネージャが組み込まれています。そう、Julia をインストールすれば他に何もしなくても使えるのです！

Julia のパッケージマネージャは、Julia の REPL を パッケージモード にすることで使用可能です。つまり対話形式でパッケージの追加・更新・削除、仮想環境/プロジェクト等の生成・アクティベート等の作業が行えます。

例えば、公式パッケージリポジトリに登録されているパッケージは、以下のように簡単に追加ができます（Julia v1.7 での動作例です）。ここでは3つのパッケージをインストールしています（コード1-11.）。

▼コード1-11. REPLのパッケージモードの例（1）：パッケージの追加・状態確認

```
(@v1.7) pkg> add BenchmarkTools Primes StaticArrays
    Updating registry at ~/.julia/registries/General
   Resolving package versions...
   Installed IntegerMathUtils ─ v0.1.0
   Installed Primes ──────────── v0.5.2
   Installed BenchmarkTools ──── v1.3.1
   Installed StaticArrays ────── v1.4.4
Updating ~/.julia/environments/v1.7/Project.toml
  [6e4b80f9] + BenchmarkTools v1.3.1
  [27ebfcd6] + Primes v0.5.2
  [90137ffa] + StaticArrays v1.4.4
Updating ~/.julia/environments/v1.7/Manifest.toml
```

```
  [6e4b80f9] + BenchmarkTools v1.3.1
    : 《一部省略》
  [8e850b90] + libblastrampoline_jll
Precompiling project...
  ✓ IntegerMathUtils
    : 《一部省略》
  ✓ StaticArrays
  7 dependencies successfully precompiled in 4 seconds (15 already precompiled)

(@v1.7) pkg> status
      Status ~/.julia/environments/v1.7/Project.toml
  [6e4b80f9] BenchmarkTools v1.3.1
  [27ebfcd6] Primes v0.5.2
  [90137ffa] StaticArrays v1.4.4
```

最後に status というコマンドを実行しています。これは現在インストールされているパッケージの情報等を表示するものです。

仮想環境の構築も簡単です。新しいディレクトリを作成して以下のように activate 〜 でそのディレクトリを指定するだけです（以下の実行例に表示されているパスは仮のものです）。仮想環境を生成（もしくはアクティベート）すると、パッケージモード のプロンプトも変わります（コード1-12.）。

▼コード1-12. REPLのパッケージモードの例（2）：仮想環境作成

```
(@v1.7) pkg> activate SampleEnv
  Activating new environment at /path/to/currentdir/SampleEnv/Project.toml

(SampleEnv) pkg> add BenchmarkTools
    Updating registry at ~/.julia/registries/General
   Resolving package versions...
Updating /path/to/currentdir/SampleEnv/Project.toml
  [6e4b80f9] + BenchmarkTools v1.3.1
Updating /path/to/currentdir/SampleEnv/Manifest.toml
  [6e4b80f9] + BenchmarkTools v1.3.1
    : 《以下省略》

(SampleEnv) pkg> status
Status /path/to/currentdir/SampleEnv/Project.toml
  [6e4b80f9] BenchmarkTools v1.3.1
```

ついでに add で BenchmarkTools というパッケージのみ追加してみました。statuts で確認すると、その情報しか表示されません。デフォルト環境にインストールした Primes や StaticArrays は見えていない（環境が分かれている）のが分かると思います。

実行例中に Project.toml や Manifest.toml などのファイル名が表示されていますが、特に重要なのは Project.toml の方です。これは「プロジェクト（仮想環境）管理ファイル」です。パッケージモード で add したパッケージの情報や、プロジェクト（仮想環境）の基本的な情報が記述されます。これを他のディレクトリにコピーして以下のように実行すれば、元のディレクトリで作成したプロジェクト（仮想環境）が再現できます。以下は otherdir/SampleEnv というディレクトリに先ほどの Project.toml がある状態で、そのディレクトリで Julia を実行した場合の実行例です（仮想コード1-4.）。