ジェネリクス

ジェネリクスは、使われるまで型が決まらないようないろいろな型の値を受け入れられる機能を作るときに使います。ジェネリクスは日本語で総称型と呼ばれることもあります。

ジェネリクスは、ライブラリを作る人のための機能です。画面を量産する時とかには基本的には出てこないでしょう。実装していて「これはどんな子要素の型が来ても利用できる汎用的な処理だ」といったことがあればそこで初めて登場します。

ジェネリクスの書き方

ジェネリクスは、関数、インタフェース、クラスなどと一緒に利用できます。

次の関数は指定された第一引数の値を、第二引数の数だけ含む配列を作って返すコードのサンプルです。

ジェネリクスの場合は名前の直後、関数の場合は引数リストの直前に、ジェネリクスの型パラメータを関数の引数のように（ただし、対になる不等号でくくる）記述します。下記のコードでは `T` がそれにあたります。 関数の宣言の場合は入出力の引数や関数本体の定義時に、Tがなんらかの型であるかのように利用できます。インタフェースやクラスの場合はメンバーのメソッドの宣言、クラスであればメンバーのフィールドやメソッドの実装の中で利用できます。

どの型が入ってくるかどうかは利用されるまではわかりませんが、 `T` は実際に使うときに、全て同じ型名がここに入ります。

ジェネリクスの関数宣言

```
function multiply<T>(value: T, n: number): Array<T> {
    const result: Array<T> = [];
    result.length = n;
    result.fill(value);
    return result;
}
```

`T` には `string` など、利用時に自由に型を入れることができます。宣言文と同じように `< >` で括られている中に型名を明示的に書くことで指定できます。 また、型推論も可能なので、引数の型から明示的に導き出せる場合には、型パラメータを省略することができます。

ジェネリクスの利用

```
// -1が10個入った配列を作る
const values = multiply<number>(-1, 10);

// ジェネリクスの型も推論ができるので、引数から明示的にわかる場合は省略可能
const values = multiply("すごい！", 10);
```

ジェネリクスの引数名

ジェネリクスでは、型のパラメータとしては `T` 、 `U` 、 `V` などの大文字の文字が一般的に使われます。あるいは `T1` 、 `T2` などでもいいでしょう。 一般に、インスタンス名（変数名）は小文字スタートの識別子が、クラスやインタフェースは大文字の識別子が使われてきたので、呼んだ時にも直感的に理解しやすいでしょう。 これはC++やJavaなどでも使われてきた慣習ですので、他の言語のユーザーも慣れた方法であります。

基本的な型付けの最後の方で触れたMapped Typeの場合は、オブジェクトのキーをパラメータのように扱っていました。これは `K` が使われることが多いようです。

ジェネリクスの型パラメータに制約をつける

ジェネリクスの型パラメータは列挙するだけの場合はどんな型の引数も受け入れるという意味になります。しかし、何かしらの特別な型のみを受け入れたいということがあるでしょう。 ジェネリクスは動的に型が決まるといっても、デフォルトでは `unknown` と同じように解釈されます。関数の本体の中で型パラメータのプロパティにアクセスするとエラーになります。

型パラメータもコンパイル時にチェックされる

```
function isTodayBirthday<T>(person: T): boolean {
    const today = new Date();
    // personの型は未知なのでgetBirthDay()メソッドがあるかどうか未定でエラーになる
    const birthDay = person.getBirthDay();
    return today.getMonth() === birthDay.getMonth() && today.getDate() === birthDay.getDate();
}
```

ここでは、 `getBirthDay()` メソッドを持っている型ならなんでも受け入れられるようにしたいですよね？そのようなときは、 `extends` を使ってTはこのインタフェースを満たす型でなければならないということを指定できます。

*extends*で型パラメータに制約を与える

```
 type Person = {
     getBirthDay(): Date;
 }

function isTodayBirthday<T extends Person>(person: T): boolean {
    const today = new Date();
    // personの型は少なくともPersonを満たす型なのでgetBirthDay()メソッドが利用可能
    const birthDay = person.getBirthDay();
    return today.getMonth() === birthDay.getMonth() && today.getDate() === birthDay.getDate();
}
```

このように書くことで、関数定義の実装時にエラーとなることはありません。また、利用時にも、この制約を満たさない場合にはエラーになります。

文字列などの合併型も `extends string` で設定できます。これで何かしらの文字列のみを型パラメータに指定できます。 `number` にすれば数値も扱えます。

```
// 何かしらの文字列とその合併型だけを受け付ける
function action<T extends string>(actionName: T) {
    :
}

action<keyof ActionList>("register");
```

`extends` に合併型を設定すればさらに特定の文字列だけに限定できます。

型パラメータの自動解決

TypeScriptの処理系は入力値の型などから型パラメータを推論しようとします。すべての型が解決可能であれば、型パラメータの指定を省略できます。また、型パラメータ同士で影響を与え合う（制約を与え合う）ような型パラメータの制約も書くことができます。その場合も、お互いの情報や引数の情報を元に、お互いに推論できるところから推論していって、自動解決できるものを解決していきます。

次の `setValue` は何やら不思議な型定義になっています。このうち、 `T` はオブジェクトの型、 `K` はオブジェクトのプロパティ名の合併型で、 `U` はオブジェクトのプロパティの方の型を表しています。やっていることは、オブジェクトの型にマッチした代入をするだけのなんの変哲も無い（役に立たない）コードです。

値の設定を大げさに書く

```
function setValue<T, K extends keyof T, U extends T[K]>(obj: T, key: K, value: U) {
    obj[key] = value;
}
```

Visual Studio CodeやTypeScriptのPlaygroundのページで次の `setValue` 呼び出しを書いてみてください。まず、最初の引数に `park` をタイプすると、型 `T` が決まります。 そうなると、ポップアップする引数 `key` の型は `"name" | "hasTako"` に、 `value` の型は `string | boolean` になります。 次に、二つ目の引数に `"name"` をタイプすると、 `value` の型は `string` となります。 このように連鎖的にパズルを解くようにTypeScriptの処理系は型の制約を解決していきます。

エディタの補完を試してみよう

```
const park: ParkForm = {
  name: "恵比寿東",
  hasTako: true
};

setValue(park, "name", "神明児童遊園");
```

ただし、型パラメータで設定することを期待しているのか、それとも引数だけからすべてを解決していけるように設計されているのかは一目見て理解するのは難しいので、どういった意図のコードになっているのかはドキュメントやサンプルコードで伝えるようにしたほうが良いでしょう。

ジェネリクスの文法でできること、できないこと

ジェネリクスでできることを一言で言えば、利用する側の手間を減らしつつ、型チェックをより厳しくすることです。 ジェネリクスを使うと、引数の型によって返り値の型が変わるとか、最初の引数の型によって、別の引数の型が変わるとか、そういったことが実現できます。 また完全に自由にするのではなく、特定の条件を満たす型パラメータのみを受け取ることも指定できましたよね。

一方でできないこともあります。C++のテンプレートのように、指定された型によってロジックを切り替えるといったことはできません。 例えば、要素の型とで、要素数が型パラメータで設定できる固定長配列などはジェネリクスやテンプレートで簡単に実現できます。 C++の場合は、例えば要素が32ビットの数値で要素数が4の場合だけSIMDを使って足し算を高速化するといった「特殊化」ができますが、ジェネリクスではそのようなことはできません。

また、即値の数値を型パラメータに入れることもC++ではできましたし、それの演算もできます。C++では特殊化と組み合わせて、次のような数学の漸化式のような型定義もできます。 これにより、4次元配列でも5次元配列でも簡単に作り出すことがC++では可能ですし、これを駆使したテンプレートメタプログラミングという技法も編み出されましたが、これもTypeScriptには不可能です。

- n次元配列はn-1次元配列の配列

- 1次元配列は普通の配列（特殊化）

TypeScriptの文法のうち、型宣言などのJavaScriptから追加されたものは、基本、そのまま切り落とせば単なるJavaScriptになる、というのが原則としてありました。 ジェネリクスについても同様ですので、型で実装を分岐というJavaScriptにないことはできません。

型変換のためのユーティリティ型

TypeScriptでは組み込みの型変換のためのジェネリクスのユーティリティ型を提供しています。詳細なリファレンスは本家のハンドブックの中のUtility Typesにあります。

オブジェクトに対するユーティリティ型

`T` に定義済みのオブジェクトを指定することで、特定の変更を加えた新しいオブジェクトの型が定義されます。

オブジェクトに対するユーティリティ型の使い方。

```
const userDiff: Partial<User> = {
  organization: "Future Corporation"
};
```

- `Partial<T>` : 要素が省略可能になった型
- `Readonly<T>` : 要素が読み込み専用になった型
- `Required<T>` : `Partial<T>` とは逆に、すべての省略可能な要素を必須に直した型

オブジェクトと属性名に対するユーティリティ型

次の3つの型は `T` 以外に、 `K` としてプロパティの文字列の合併型を持ち、新しいオブジェクトの型を作ります。

オブジェクトと属性名に対するユーティリティ型の使い方。

```
const viewItems: Pick<User, "name" | "gender"> = {
  name: "Yoshiki Shibukawa",
  gender: "male"
};
```

- `Record<K,T>` : `T` を子供の要素に持つ `Map` 型のようなデータ型（Kがキー）を作成。
- `Pick<T,K>` : `T` の中の特定のキー `K` だけを持つ型を作成
- `Omit<T,K>` : `T` の中の特定のキー `K` だけを持たない型を作成

型の集合演算のユーティリティ型

次の3つの型は、`T` と `U` （`NonNullable<T>` 以外）として、合併型をパラメータとして受け、新しい合併型を作り出します。

型の集合演算のユーティリティ型の使い方。

```
const year: NonNullable<string | number | undefined> = "昭和";
```

- `Exclude<T,U>`: `T` の合併型から、`U` の合併型の構成要素を除外した合併型を作る型
- `Extract<T,U>`: `T` の合併型と、`U` の合併型の両方に含まれる合併型を作る型
- `NonNullable<T>`: `T` の合併型から、`undefined` を抜いた合併型を作る型

関数のユーティリティ型

関数を渡すと、その返り値の型を返すユーティリティ型です。

- `ReturnType<T>`

クラスに対するユーティリティ型

クラスに対するユーティリティ型です。あまり使うことはないと思われます。

- `ThisType<T>`: JavaScript時代のコードは `this` が何を表すのかを外挿できましたのでそれを表現するユーティリティ型です。新しい型は作りません。`--noImplicitThis` がないと動かないとのこと。
- `InstanceType<T>`: `InstanceType<typeof C>` が `C` を返すとドキュメントに書かれていますが用途はよくわかりません。

`any` や `unknown` 、合併型との違い

未知の型というと、`any` や `unknown` が思いつくでしょう。また、複数の型を受け付けるというと、合併型もあります。これらとジェネリクスの違いについて説明します。

`any` や `unknown` の変数に値を設定してしまうと、型情報がリセットされます。取り出すときに、適切な型を宣言してあげないと、その後のエラーチェックが無効になったり、エディタの補完ができません。

次の関数は、初回だけ指定の関数を読んで値を取って来るが、2回目以降は保存した値をそのまま返す関数です。初回アクセスまで初期化を遅延させます。

*any*版の遅延初期化関数

```
function lazyInit(init: () => any): () => any {
  let cache: any;
  let isInit = false;
  return function(): any {
    if (!isInit) {
      cache = init();
      isInit = true;
    }
    return cache;
  }
}
```

`any` 版を使って見たのが次のコードです。

非ジェネリック版の使い方

```
const getter = lazyInit(() => "initialized");
const value = getter();
// valueはany型なので、上記のvalueの後ろで.をタイプしてもメソッド候補はでてこない
```

この場合、 `cache` ローカル変数に入っているのは文字列ですし、 `value` にも文字列が格納されます。 しかし、TypeScriptの処理系は `any` に入るだけで補完をあきらめてしまいます。

次のジェネリクス版を紹介します。ジェネリクス版は入力された引数の情報から返り値の型が正しく推論されるため、返り値の型を使うときに正しく補完できます。

*any*版の遅延初期化関数

```
function lazyInit<T>(init: () => T): () => T {
  let cache: T;
  let isInit = false;
  return function(): T {
    if (!isInit) {
      cache = init();
      isInit = true;
    }
    return cache;
  }
}
```

ジェネリック版の使い方

```
const getter = lazyInit(() => "initialized");
const value = getter();
// valueはstring型なので、上記のvalueの後ろで.をタイプするとメソッド候補が出てくる
```

合併型についても、型の補完時に余計な型情報がまざってしまうため、型ガードで必要な型である保証が必要です。また、ジェネリクスには2つの引数があって両方の型が同じ、という保証もしやすいメリットがあります。

まとめ

ジェネリクスについて紹介しました。

基本的に、画面を量産するという仕事ではなく、共通ライブラリを作り出すとか、そういったタスクで活躍する中級向けの機能です。 作り込めば作り込むほど、使う人にやさしく、間違った情報が入れにくい関数やクラス、インタフェースが作れます。

一方で、型情報の作り込みは読みにくいコードに直結します。書いているときには良いのですが数日後にいじるのが少し難しいコードになりがちです。凝った正規表現に近いものがあると思います