Genéricos

Uma parte importante da engenharia de software é construir componentes que não apenas tenham APIs bem definidas e consistentes, mas que também sejam reutilizáveis. Componentes que são capazes de trabalhar com os dados de hoje, assim como com os dados de amanhã, proporcionarão as capacidades mais flexíveis para construir grandes sistemas de software.

Em linguagens como C# e Java, uma das principais ferramentas para criar componentes reutilizáveis é **genéricos**, ou seja, a capacidade de criar um componente que pode trabalhar sobre uma variedade de tipos, em vez de um único. Isso permite que os usuários consumam esses componentes e usem seus próprios tipos.

Olá Mundo dos Genéricos

Para começar, vamos fazer o "olá mundo" dos genéricos: a função identidade. A função identidade é uma função que retornará o que for passado. Você pode pensar nisso de maneira semelhante ao comando `echo`.

Sem genéricos, teríamos que dar à função identidade um tipo específico:

```
function identity(arg: number): number {
  return arg;
}
```

Ou poderíamos descrever a função identidade usando o tipo `any`:

```
function identity(arg: any): any {
  return arg;
}
```

Embora o uso de `any` seja certamente genérico, pois fará com que a função aceite qualquer tipo para o tipo de `arg`, na verdade estamos perdendo a informação sobre qual era esse tipo quando a função retorna. Se passássemos um número, a única informação que temos é que qualquer tipo poderia ser retornado.

Em vez disso, precisamos de uma maneira de capturar o tipo do argumento de forma que possamos usá-lo para denotar o que está sendo retornado. Aqui, usaremos uma **variável de tipo**, um tipo especial de variável que trabalha com tipos em vez de valores.

```
function identity<Type>(arg: Type): Type {
  return arg;
}
```

Agora adicionamos uma variável de tipo `Type` à função identidade. Esse `Type` nos permite capturar o tipo que o usuário fornece (por exemplo, `number`), para que possamos usar essa informação depois. Aqui, usamos `Type` novamente como o tipo de retorno. Ao inspecionar, agora podemos ver que o mesmo tipo é usado para o argumento e o tipo de retorno. Isso nos permite transportar essa informação de tipo de um lado da função e do outro.

Dizemos que esta versão da função identidade é genérica, pois funciona sobre uma gama de tipos. Ao contrário do uso de `any`, é também tão precisa (ou seja, não perde nenhuma informação) quanto a primeira função `identity` que usou números para o argumento e tipo de retorno.

Uma vez que escrevemos a função genérica `identity`, podemos chamá-la de duas maneiras. A primeira maneira é passar todos os argumentos, incluindo o argumento de tipo, para a função:

```
let output = identity<string>("myString");

let output: string
```

Aqui, definimos explicitamente `Type` como `string` como um dos argumentos da chamada da função, denotado usando `< >` em torno dos argumentos, em vez de `()`.

A segunda maneira é também talvez a mais comum. Aqui usamos **inferência de argumento de tipo** — ou seja, queremos que o compilador defina o valor de `Type` para nós automaticamente com base no tipo do argumento que passamos:

```
let output = identity("myString");

let output: string
```

Observe que não precisávamos passar explicitamente o tipo nos colchetes angulares (`<>`); o compilador apenas olhou para o valor `"myString"` e definiu `Type` para seu tipo. Embora a inferência de argumento de tipo possa ser uma ferramenta útil para manter o código mais curto e legível, você pode precisar passar explicitamente os argumentos de tipo como fizemos no exemplo anterior quando o compilador não consegue inferir o tipo, como pode acontecer em exemplos mais complexos.

Trabalhando com Variáveis de Tipo Genérico

Quando você começa a usar genéricos, notará que, ao criar funções genéricas como `identity`, o compilador fará valer que você use qualquer parâmetro de tipo genérico no corpo da função corretamente. Ou seja, que você realmente trate esses parâmetros como se pudessem ser qualquer tipo.

Vamos pegar nossa função `identity` de antes:

```
function identity<Type>(arg: Type): Type {
  return arg;
}
```

E se quisermos também registrar o comprimento do argumento `arg` no console a cada chamada? Poderíamos ser tentados a escrever isso:

```
function loggingIdentity<Type>(arg: Type): Type {
  console.log(arg.length);
Property 'length' does not exist on type 'Type'.
  return arg;
}
```

Quando fazemos isso, o compilador nos dará um erro de que estamos usando o membro `.length` de `arg`, mas em nenhum lugar dissemos que `arg` tem esse membro. Lembre-se, dissemos anteriormente que essas variáveis de tipo representam qualquer tipo, então alguém usando essa função poderia ter passado um `number`, que não tem um membro `.length`.

Digamos que realmente pretendemos que essa função trabalhe em arrays de Type em vez de Type diretamente. Como estamos trabalhando com arrays, o membro .length deve estar disponível. Podemos descrever isso da mesma forma que criaríamos arrays de outros tipos:

```
function loggingIdentity<Type>(arg: Type[]): Type[] {
  console.log(arg.length);
```

```
  return arg;
}
```

Você pode ler o tipo de `loggingIdentity` como “a função genérica `loggingIdentity` recebe um parâmetro de tipo `Type`, e um argumento `arg` que é um array de Types, e retorna um array de Types.” Se passássemos um array de números, teríamos um array de números de volta, pois `Type` se vincularia a `number`. Isso nos permite usar nossa variável de tipo genérico `Type` como parte dos tipos com os quais estamos trabalhando, em vez do tipo inteiro, nos dando maior flexibilidade.

Podemos alternativamente escrever o exemplo assim:

```
function loggingIdentity<Type>(arg: Array<Type>): Array<Type> {
  console.log(arg.length); // Array tem um .length, então sem mais erro
  return arg;
}
```

Você pode já estar familiarizado com esse estilo de tipo de outras linguagens. Na próxima seção, abordaremos como você pode criar seus próprios tipos genéricos como `Array<Type>`.

Tipos Genéricos

Nas seções anteriores, criamos funções de identidade genéricas que funcionavam sobre uma gama de tipos. Nesta seção, exploraremos o tipo das funções em si e como criar interfaces genéricas.

O tipo de funções genéricas é semelhante ao das funções não genéricas, com os parâmetros de tipo listados primeiro, de forma semelhante às declarações de função:

```
function identity<Type>(arg: Type): Type {
  return arg;
}

let myIdentity: <Type>(arg: Type) => Type = identity;
```

Poderíamos também ter usado um nome diferente para o parâmetro de tipo genérico no tipo, desde que o número de variáveis de tipo e como as variáveis de tipo são usadas estejam alinhados.

```
function identity<Type>(arg: Type): Type {
  return arg;
}

let myIdentity: <Input>(arg: Input) => Input = identity;
```

Podemos também escrever o tipo genérico como uma assinatura de chamada de um tipo literal de objeto:

```
function identity<Type>(arg: Type): Type {
  return arg;
}

let myIdentity: { <Type>(arg: Type): Type } = identity;
```

O que nos leva a escrever nossa primeira interface genérica. Vamos pegar o literal de objeto do exemplo anterior e movê-lo para uma interface:

```
interface GenericIdentityFn {
  <Type>(arg: Type): Type;
}
```

```
function identity<Type>(arg: Type): Type {
  return arg;
}

let myIdentity: GenericIdentityFn = identity;
```

Em um exemplo semelhante, podemos querer mover o parâmetro genérico para ser um parâmetro de toda a interface. Isso nos permite ver quais tipos estamos usando como genéricos (por exemplo, `Dictionary<string>` em vez de apenas `Dictionary`). Isso torna o parâmetro de tipo visível para todos os outros membros da interface.

```
interface GenericIdentityFn<Type> {
  (arg: Type): Type;
}

function identity<Type>(arg: Type): Type {
  return arg;
}

let myIdentity: GenericIdentityFn<number> = identity;
```

Observe que nosso exemplo mudou para ser algo ligeiramente diferente. Em vez de descrever uma função genérica, agora temos uma assinatura de função não genérica que faz parte de um tipo genérico. Quando usamos `GenericIdentityFn`, agora também precisaremos especificar o argumento de tipo correspondente (aqui: `number`), efetivamente bloqueando o que a assinatura de chamada subjacente usará. Entender quando colocar o parâmetro de tipo diretamente na assinatura de chamada e quando colocá-lo na própria interface será útil para descrever quais aspectos de um tipo são genéricos.

Além de interfaces genéricas, também podemos criar classes genéricas. Observe que não é possível criar enums e namespaces genéricos.

Classes Genéricas

Uma classe genérica tem uma forma semelhante a uma interface genérica. Classes genéricas têm uma lista de parâmetros de tipo genérico em colchetes angulares (`<>`) após o nome da classe.

```
class GenericNumber<NumType> {
  zeroValue: NumType;
  add: (x: NumType, y: NumType) => NumType;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
  return x + y;
};
```

Este é um uso bastante literal da classe `GenericNumber`, mas você pode ter notado que nada está restringindo seu uso apenas ao tipo `number`. Poderíamos, em vez disso, ter usado `string` ou até mesmo objetos mais complexos.

```
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function (x, y) {
```

```
  return x + y;
};

console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Assim como nas interfaces, colocar o parâmetro de tipo na própria classe nos permite garantir que todas as propriedades da classe estejam trabalhando com o mesmo tipo.

Como cobrimos em nossa seção sobre classes, uma classe tem dois lados em seu tipo: o lado estático e o lado de instância. Classes genéricas são apenas genéricas sobre seu lado de instância, em vez de seu lado estático, então, ao trabalhar com classes, membros estáticos não podem usar o parâmetro de tipo da classe.

Restrições Genéricas

Se você se lembra de um exemplo anterior, às vezes você pode querer escrever uma função genérica que funcione em um conjunto de tipos onde você tem **algum** conhecimento sobre quais capacidades esse conjunto de tipos terá. Em nosso exemplo `loggingIdentity`, queríamos ser capazes de acessar a propriedade `.length` de `arg`, mas o compilador não conseguiu provar que cada tipo tinha uma propriedade `.length`, então ele nos avisa que não podemos fazer essa suposição.

```
function loggingIdentity<Type>(arg: Type): Type {
  console.log(arg.length);
Property 'length' does not exist on type 'Type'.
  return arg;
}
```

Em vez de trabalhar com todos os tipos, gostaríamos de restringir essa função para trabalhar com todos os tipos que **também** possuem a propriedade `.length`. Desde que o tipo tenha esse membro, permitiremos, mas é necessário que tenha pelo menos esse membro. Para fazer isso, devemos listar nossa exigência como uma restrição sobre o que `Type` pode ser.

Para isso, criaremos uma interface que descreve nossa restrição. Aqui, criaremos uma interface que possui uma única propriedade `.length` e, em seguida, usaremos essa interface e a palavra-chave `extends` para denotar nossa restrição:

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length); // Agora sabemos que tem uma propriedade .length, então
sem mais erro
  return arg;
}
```

Como a função genérica agora está restrita, ela não funcionará mais sobre todos os tipos:

```
loggingIdentity(3);
Argument of type 'number' is not assignable to parameter of type 'Lengthwise'.
```

Em vez disso, precisamos passar valores cujo tipo tenha todas as propriedades exigidas:

```
loggingIdentity({ length: 10, value: 3 });
```

Usando Parâmetros de Tipo em Restrições Genéricas

Você pode declarar um parâmetro de tipo que é restringido por outro parâmetro de tipo. Por exemplo, aqui gostaríamos de obter uma propriedade de um objeto dado seu nome. Queremos garantir que não estamos acidentalmente pegando uma propriedade que não existe no `obj`, então colocaremos uma restrição entre os dois tipos:

```
function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) {
  return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a");
getProperty(x, "m");
Argument of type '"m"' is not assignable to parameter of type '"a" | "b" | "c" |
"d"'.
```

Usando Tipos de Classe em Genéricos

Ao criar fábricas em TypeScript usando genéricos, é necessário se referir aos tipos de classe por suas funções construtoras. Por exemplo,

```
function create<Type>(c: { new (): Type }): Type {
  return new c();
}
```

Um exemplo mais avançado usa a propriedade prototype para inferir e restringir relações entre a função construtora e o lado de instância dos tipos de classe.

```
class BeeKeeper {
  hasMask: boolean = true;
}

class ZooKeeper {
  nametag: string = "Mikle";
}

class Animal {
  numLegs: number = 4;
}

class Bee extends Animal {
  numLegs = 6;
  keeper: BeeKeeper = new BeeKeeper();
}

class Lion extends Animal {
  keeper: ZooKeeper = new ZooKeeper();
}

function createInstance<A extends Animal>(c: new () => A): A {
  return new c();
}
```

```
createInstance(Lion).keeper.nametag;
createInstance(Bee).keeper.hasMask;
```

Esse padrão é usado para implementar o padrão de design mixins.

Valores Padrão para Parâmetros Genéricos

Ao declarar um padrão para um parâmetro de tipo genérico, você torna opcional especificar o argumento de tipo correspondente. Por exemplo, uma função que cria um novo `HTMLElement`. Chamando a função sem argumentos, gera um `HTMLDivElement`; chamando a função com um elemento como primeiro argumento, gera um elemento do tipo do argumento. Você também pode opcionalmente passar uma lista de filhos. Anteriormente, você teria que definir a função da seguinte forma:

```
declare function create(): Container<HTMLDivElement, HTMLDivElement[]>;
declare function create<T extends HTMLElement>(element: T): Container<T, T[]>;
declare function create<T extends HTMLElement, U extends HTMLElement>(
  element: T,
  children: U[]
): Container<T, U[]>;
```

Com valores padrão para parâmetros genéricos, podemos reduzir isso para:

```
declare function create<T extends HTMLElement = HTMLDivElement, U extends
HTMLElement[] = T[]>(
  element?: T,
  children?: U
): Container<T, U>;

const div = create();

const div: Container<HTMLDivElement, HTMLDivElement[]>

const p = create(new HTMLParagraphElement());

const p: Container<HTMLParagraphElement, HTMLParagraphElement[]>
```

Um padrão de parâmetro genérico segue as seguintes regras:

- Um parâmetro de tipo é considerado opcional se tiver um padrão.
- Parâmetros de tipo obrigatórios não devem seguir parâmetros de tipo opcionais.
- Tipos padrão para um parâmetro de tipo devem satisfazer a restrição para o parâmetro de tipo, se existir.
- Ao especificar argumentos de tipo, você só precisa especificar argumentos de tipo para os parâmetros de tipo obrigatórios. Parâmetros de tipo não especificados serão resolvidos para seus tipos padrão.
- Se um tipo padrão for especificado e a inferência não puder escolher um candidato, o tipo padrão é inferido.
- Uma declaração de classe ou interface que mescla com uma declaração de classe ou interface existente pode introduzir um padrão para um parâmetro de tipo existente.

- Uma declaração de classe ou interface que mescla com uma declaração de classe ou interface existente pode introduzir um novo parâmetro de tipo desde que especifique um padrão.

Anotações de Variância

> Este é um recurso avançado para resolver um problema muito específico e deve ser usado apenas em situações em que você identificou uma razão para usá-lo.

Covariância e contravariância são termos da teoria dos tipos que descrevem qual é a relação entre dois tipos genéricos. Aqui está uma breve introdução ao conceito.

Por exemplo, se você tem uma interface que representa um objeto que pode `produzir` um determinado tipo:

```
interface Producer<T> {
  make(): T;
}
```

Podemos usar um `Producer<Cat>` onde um `Producer<Animal>` é esperado, porque um `Cat` é um `Animal`. Essa relação é chamada de **covariância**: a relação de `Producer<T>` para `Producer<U>` é a mesma que a relação de `T` para `U`.

Por outro lado, se você tem uma interface que pode `consumir` um determinado tipo:

```
interface Consumer<T> {
  consume: (arg: T) => void;
}
```

Então, podemos usar um `Consumer<Animal>` onde um `Consumer<Cat>` é esperado, porque qualquer função que é capaz de aceitar um Cat também deve ser capaz de aceitar um `Animal`. Essa relação é chamada de **contravariância**: a relação de `Consumer<T>` para `Consumer<U>` é a mesma que a relação de `U` para `T`. Note a reversão de direção em comparação com a covariância! É por isso que a contravariância "se cancela", mas a covariância não.

Em um sistema de tipos estrutural como o do TypeScript, a covariância e a contravariância são comportamentos que emergem naturalmente da definição de tipos. Mesmo na ausência de genéricos, veríamos relações covariantes (e contravariantes):

```
interface AnimalProducer {
  make(): Animal;
}
// Um CatProducer pode ser usado em qualquer lugar que um
// produtor de Animal é esperado
interface CatProducer {
  make(): Cat;
}
```

TypeScript tem um sistema de tipos estrutural, então ao comparar dois tipos, por exemplo, para ver se um `Producer<Cat>` pode ser usado onde um `Producer<Animal>` é esperado, o algoritmo usual seria expandir estruturalmente ambas as definições e comparar suas estruturas. No entanto, a variância permite uma otimização extremamente útil: se `Producer<T>` é covariante em relação a `T`, então podemos simplesmente verificar `Cat` e `Animal` em vez disso, pois sabemos que eles terão a mesma relação que `Producer<Cat>` e `Producer<Animal>`.

Note que essa lógica só pode ser usada quando estamos examinando duas instâncias do mesmo tipo. Se tivermos um `Producer<T>` e um `FastProducer<U>`, não há garantia de que `T` e `U` necessariamente se referem às mesmas posições nesses tipos, então essa verificação será sempre realizada estruturalmente.

Como a variância é uma propriedade emergente naturalmente de tipos estruturais, o TypeScript automaticamente **infere** a variância de cada tipo genérico. Em casos extremamente raros envolvendo certos tipos circulares, essa medição pode ser imprecisa. Se isso acontecer, você pode adicionar uma anotação de variância a um parâmetro de tipo para forçar uma variância particular:

```
// Anotação contravariante
interface Consumer<in T> {
  consume: (arg: T) => void;
}
// Anotação covariante
interface Producer<out T> {
  make(): T;
}
// Anotação invariável
interface ProducerConsumer<in out T> {
  consume: (arg: T) => void;
  make(): T;
}
```

Use isso apenas se você estiver escrevendo a mesma variância que **deve** ocorrer estruturalmente.

> Nunca escreva uma anotação de variância que não corresponda à variância estrutural!

É fundamental reforçar que as anotações de variância só estão em vigor durante uma comparação baseada em instanciação. Elas não têm efeito durante uma comparação estrutural. Por exemplo, você não pode usar anotações de variância para “forçar” um tipo a ser realmente invariável:

```
// NÃO FAÇA ISSO - anotação de variância
// não corresponde ao comportamento estrutural
interface Producer<in out T> {
  make(): T;
}
// Não é um erro de tipo -- esta é uma comparação estrutural,
// então as anotações de variância não estão em vigor
const p: Producer<string | number> = {
    make(): number {
        return 42;
    }
}
```

Aqui, a função `make` do literal de objeto retorna `number`, o que poderíamos esperar que causasse um erro, porque `number` não é `string | number`. No entanto, isso não é uma comparação baseada em instanciação, porque o literal de objeto é um tipo anônimo, não um `Producer<string | number>`.

> Anotações de variância não alteram o comportamento estrutural e são consultadas apenas em situações específicas.

É muito importante escrever anotações de variância apenas se você souber absolutamente por que está fazendo isso, quais são suas limitações e quando elas não estão em vigor. Se o TypeScript usa uma comparação baseada em instanciação ou uma comparação estrutural não é um comportamento especificado e pode mudar de versão para versão por razões de correção ou desempenho, portanto,

você deve escrever anotações de variância apenas quando elas corresponderem ao comportamento estrutural de um tipo. Não use anotações de variância para tentar “forçar” uma variância particular; isso causará comportamentos imprevisíveis em seu código.

> NÃO escreva anotações de variância a menos que elas correspondam ao comportamento estrutural de um tipo.

Lembre-se, o TypeScript pode inferir automaticamente a variância de seus tipos genéricos. Quase nunca é necessário escrever uma anotação de variância, e você deve fazê-lo apenas quando identificar uma necessidade específica. Anotações de variância **não** alteram o comportamento estrutural de um tipo, e dependendo da situação, você pode ver uma comparação estrutural feita quando esperava uma comparação baseada em instanciação. Anotações de variância não podem ser usadas para modificar como os tipos se comportam nesses contextos estruturais, e não devem ser escritas a menos que a anotação seja a mesma que a definição estrutural. Como isso é difícil de acertar, e o TypeScript pode inferir corretamente a variância na grande maioria dos casos, você não deve se ver escrevendo anotações de variância em código normal.

> Não tente usar anotações de variância para alterar o comportamento de verificação de tipos; não é para isso que elas servem.

Você **pode** achar anotações de variância temporárias úteis em uma situação de “depuração de tipos”, porque anotações de variância são verificadas. O TypeScript emitirá um erro se a variância anotada for identificavelmente errada:

```
// Erro, esta interface é definitivamente contravariante em T
interface Foo<out T> {
  consume: (arg: T) => void;
}
```

No entanto, anotações de variância podem ser mais restritivas (por exemplo, `in out` é válido se a variância real for covariante). Certifique-se de remover suas anotações de variância assim que terminar de depurar.

Por fim, se você está tentando maximizar o desempenho da verificação de tipos, **e** executou um profiler, **e** identificou um tipo específico que está lento, **e** confirmou que a inferência de variância especificamente está lenta, **e** validou cuidadosamente a anotação de variância que deseja escrever, você **pode** ver um pequeno benefício de desempenho em tipos extraordinariamente complexos ao adicionar anotações de variância.

> Não tente usar anotações de variância para alterar o comportamento de verificação de tipos; não é para isso que elas servem.