# Mais sobre Funções

Funções são o bloco básico de construção de qualquer aplicação, seja funções locais, importadas de outro módulo, ou métodos de uma classe. Elas também são valores e, assim como outros valores, o TypeScript possui várias maneiras de descrever como as funções podem ser chamadas. Vamos aprender como escrever tipos que descrevem funções.

## Expressões de Tipo de Função

A maneira mais simples de descrever uma função é com uma *expressão de tipo de função*. Esses tipos são sintaticamente semelhantes às funções de seta:

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

A sintaxe `(a: string) => void` significa “uma função com um parâmetro, chamado `a`, do tipo `string`, que não tem um valor de retorno”. Assim como nas declarações de função, se um tipo de parâmetro não for especificado, ele é implicitamente `any`.

> Note que o nome do parâmetro é obrigatório. O tipo de função `(string) => void` significa “uma função com um parâmetro chamado `string` do tipo `any`”!

Claro, podemos usar um alias de tipo para nomear um tipo de função:

```
type GreetFunction = (a: string) => void;
function greeter(fn: GreetFunction) {
  // ...
}
```

## Assinaturas de Chamada

No JavaScript, as funções podem ter propriedades além de serem chamáveis. No entanto, a sintaxe de expressão de tipo de função não permite declarar propriedades. Se quisermos descrever algo chamável com propriedades, podemos escrever uma *assinatura de chamada* em um tipo de objeto:

```
type DescribableFunction = {
  description: string;
  (someArg: number): boolean;
};
function doSomething(fn: DescribableFunction) {
  console.log(fn.description + " returned " + fn(6));
}

function myFunc(someArg: number) {
  return someArg > 3;
}
myFunc.description = "default description";
```

```
doSomething(myFunc);
```

Note que a sintaxe é ligeiramente diferente em comparação com uma expressão de tipo de função - use `:` entre a lista de parâmetros e o tipo de retorno em vez de `=>`.

## Assinaturas de Construção

As funções do JavaScript também podem ser invocadas com o operador `new`. O TypeScript se refere a elas como *construtores* porque geralmente criam um novo objeto. Você pode escrever uma *assinatura de construção* adicionando a palavra-chave `new` na frente de uma assinatura de chamada:

```
type SomeConstructor = {
  new (s: string): SomeObject;
};
function fn(ctor: SomeConstructor) {
  return new ctor("hello");
}
```

Alguns objetos, como o objeto `Date` do JavaScript, podem ser chamados com ou sem `new`. Você pode combinar assinaturas de chamada e construção no mesmo tipo de forma arbitrária:

```
interface CallOrConstruct {
  (n?: number): string;
  new (s: string): Date;
}
```

## Funções Genéricas

É comum escrever uma função onde os tipos da entrada estão relacionados ao tipo da saída, ou onde os tipos de duas entradas estão relacionados de alguma forma. Vamos considerar por um momento uma função que retorna o primeiro elemento de um array:

```
function firstElement(arr: any[]) {
  return arr[0];
}
```

Essa função faz seu trabalho, mas, infelizmente, tem o tipo de retorno `any`. Seria melhor se a função retornasse o tipo do elemento do array.

No TypeScript, *genéricos* são usados quando queremos descrever uma correspondência entre dois valores. Fazemos isso declarando um parâmetro de tipo na assinatura da função:

```
function firstElement<Type>(arr: Type[]): Type | undefined {
  return arr[0];
}
```

Ao adicionar um *parâmetro de tipo* `Type` a essa função e usá-lo em dois lugares, criamos um vínculo entre a entrada da função (o array) e a saída (o valor de retorno). Agora, quando a chamamos, um tipo mais específico é retornado:

```
// s é do tipo 'string'
const s = firstElement(["a", "b", "c"]);
// n é do tipo 'number'
const n = firstElement([1, 2, 3]);
```

```
// u é do tipo undefined
const u = firstElement([]);
```

## Inferência

Note que não precisávamos especificar `Type` neste exemplo. O tipo foi *inferido* - escolhido automaticamente - pelo TypeScript.

Podemos usar múltiplos parâmetros de tipo também. Por exemplo, uma versão independente de `map` seria assim:

```
function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {
  return arr.map(func);
}

// O parâmetro 'n' é do tipo 'string'
// 'parsed' é do tipo 'number[]'
const parsed = map(["1", "2", "3"], (n) => parseInt(n));
```

Note que neste exemplo, o TypeScript pôde inferir tanto o tipo do parâmetro de tipo `Input` (a partir do array de `string` fornecido), quanto o parâmetro de tipo `Output` baseado no valor de retorno da expressão da função `(number)`.

## Restrições

Escrevemos algumas funções genéricas que podem funcionar com *qualquer* tipo de valor. Às vezes, queremos relacionar dois valores, mas só podemos operar em um certo subconjunto de valores. Nesse caso, podemos usar uma *restrição* para limitar os tipos que um parâmetro de tipo pode aceitar.

Vamos escrever uma função que retorna o mais longo de dois valores. Para isso, precisamos de uma propriedade `length` que seja um número. Nós *construímos* a restrição do parâmetro de tipo para esse tipo, escrevendo uma cláusula `extends`:

```
function longest<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a;
  } else {
    return b;
  }
}

// longerArray é do tipo 'number[]'
const longerArray = longest([1, 2], [1, 2, 3]);
// longerString é do tipo 'alice' | 'bob'
const longerString = longest("alice", "bob");
// Erro! Números não têm uma propriedade 'length'
const notOK = longest(10, 100);
Argumento do tipo 'number' não é atribuível ao parâmetro do tipo '{ length: number;
}'.
```

Há algumas coisas interessantes a notar neste exemplo. Permitimos que o TypeScript *inferisse* o tipo de retorno de `longest`. A inferência de tipo de retorno também funciona em funções genéricas.

Como restringimos `Type` a `{ length: number }`, pudemos acessar a propriedade `.length` dos parâmetros `a` e `b`. Sem a restrição de tipo, não poderíamos acessar essas propriedades porque os valores poderiam ser de outro tipo sem uma propriedade length.

Os tipos de `longerArray` e `longerString` foram inferidos com base nos argumentos. Lembre-se, genéricos são todos sobre relacionar dois ou mais valores com o *mesmo* tipo!

Finalmente, como desejávamos, a chamada para `longest(10, 100)` é rejeitada porque o tipo `number` não tem uma propriedade `.length`.

## Trabalhando com Valores Restringidos

Aqui está um erro comum ao trabalhar com restrições genéricas:

```
function minimumLength<Type extends { length: number }>(
  obj: Type,
  minimum: number
): Type {
  if (obj.length >= minimum) {
    return obj;
  } else {
    return { length: minimum };
Type '{ length: number; }' não é atribuível ao tipo 'Type'.
  '{ length: number; }' é atribuível à restrição do tipo 'Type', mas 'Type' poderia
ser instanciado com um subtipo diferente da restrição '{ length: number; }'.
  }
}
```

Pode parecer que essa função está OK - `Type` é restringido a `{ length: number }`, e a função retorna `Type` ou um valor correspondente a essa restrição. O problema é que a função promete retornar o *mesmo* tipo de objeto que foi passado, não apenas *algum* objeto que corresponda à restrição. Se esse código fosse legal, você poderia escrever código que definitivamente não funcionaria:

```
// 'arr' recebe o valor { length: 6 }
const arr = minimumLength([1, 2, 3], 6);
// e trava aqui porque arrays têm
// um método 'slice', mas não o objeto retornado!
console.log(arr.slice(0));
```

## Especificando Argumentos de Tipo

O TypeScript geralmente pode inferir os argumentos de tipo pretendidos em uma chamada genérica, mas nem sempre. Por exemplo, digamos que você escreveu uma função para combinar dois arrays:

```
function combine<Type>(arr1: Type[], arr2: Type[]): Type[] {
  return arr1.concat(arr2);
}
```

Normalmente, seria um erro chamar essa função com arrays incompatíveis:

```
const arr = combine([1, 2, 3], ["hello"]);
Tipo 'string' não é atribuível ao tipo 'number'.
```

Se você pretendia fazer isso, no entanto, poderia especificar manualmente `Type`:

```
const arr = combine<string | number>([1, 2, 3], ["hello"]);
```

## Diretrizes para Escrever Boas Funções Genéricas

Escrever funções genéricas é divertido, e pode ser fácil exagerar com parâmetros de tipo. Ter muitos parâmetros de tipo ou usar restrições onde não são necessárias pode tornar a inferência menos bem-sucedida, frustrando os chamadores da sua função.

## Empurre Parâmetros de Tipo para Baixo

Aqui estão duas maneiras de escrever uma função que parecem similares:

```
function firstElement1<Type>(arr: Type[]) {
  return arr[0];
}

function firstElement2<Type extends any[]>(arr: Type) {
  return arr[0];
}

// a: number (bom)
const a = firstElement1([1, 2, 3]);
// b: any (ruim)
const b = firstElement2([1, 2, 3]);
```

Essas podem parecer idênticas à primeira vista, mas `firstElement1` é uma maneira muito melhor de escrever essa função. Seu tipo de retorno inferido é `Type`, mas o tipo de retorno inferido de `firstElement2` é `any` porque o TypeScript tem que resolver a expressão `arr[0]` usando o tipo de restrição, em vez de "esperar" para resolver o elemento durante uma chamada.

> Regra: Quando possível, use o parâmetro de tipo em vez de restringi-lo.

## Use Menos Parâmetros de Tipo

Aqui está outro par de funções similares:

```
function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {
  return arr.filter(func);
}

function filter2<Type, Func extends (arg: Type) => boolean>(
  arr: Type[],
  func: Func
): Type[] {
  return arr.filter(func);
}
```

Criamos um parâmetro de tipo `Func` que *não* relaciona dois valores. Isso é sempre um sinal de alerta, porque significa que os chamadores que desejam especificar argumentos de tipo precisam manualmente especificar um argumento de tipo extra sem necessidade. `Func` não faz nada além de tornar a função mais difícil de ler e entender!

> Regra: Sempre use o menor número possível de parâmetros de tipo.

## Parâmetros de Tipo Devem Aparecer Duas Vezes

Às vezes esquecemos que uma função pode não precisar ser genérica:

```
function greet<Str extends string>(s: Str) {
  console.log("Hello, " + s);
}

greet("world");
```

Poderíamos facilmente ter escrito uma versão mais simples:

```
function greet(s: string) {
  console.log("Hello, " + s);
}
```

Lembre-se, parâmetros de tipo são para *relacionar os tipos de múltiplos valores*. Se um parâmetro de tipo é usado apenas uma vez na assinatura da função, ele não está relacionando nada. Isso inclui o tipo de retorno inferido; por exemplo, se `Str` fosse parte do tipo de retorno inferido de `greet`, estaria relacionando os tipos de argumento e retorno, portanto, seria usado *duas vezes* apesar de aparecer apenas uma vez no código escrito.

> Regra: Se um parâmetro de tipo aparece apenas em uma localização, reconsidere fortemente se você realmente precisa dele.

## Parâmetros Opcionais

Funções em JavaScript muitas vezes aceitam um número variável de argumentos. Por exemplo, o método toFixed de `number` aceita uma contagem de dígitos *opcional*:

```
function f(n: number) {
  console.log(n.toFixed()); // 0 argumentos
  console.log(n.toFixed(3)); // 1 argumento
}
```

Podemos modelar isso em TypeScript marcando o parâmetro como opcional com `?`:

```
function f(x?: number) {
  // ...
}
f(); // OK
f(10); // OK
```

Embora o parâmetro seja especificado como tipo `number`, o parâmetro `x` terá, na verdade, o tipo `number | undefined`, pois parâmetros não especificados em JavaScript recebem o valor `undefined`.

Você também pode fornecer um parâmetro *padrão*:

```
function f(x = 10) {
  // ...
}
```

Agora, no corpo de `f`, `x` terá o tipo `number`, pois qualquer argumento `undefined` será substituído por `10`. Note que quando um parâmetro é opcional, os chamadores podem sempre passar `undefined`, já que isso simula simplesmente um argumento “ausente”:

```
// Todos OK
f();
f(10);
f(undefined);
```

## Parâmetros Opcionais em Callbacks

Uma vez que você aprendeu sobre parâmetros opcionais e expressões de tipo de função, é muito fácil cometer os seguintes erros ao escrever funções que invocam callbacks:

```
function myForEach(arr: any[], callback: (arg: any, index?: number) => void) {
  for (let i = 0; i < arr.length; i++) {
    callback(arr[i], i);
  }
}
```

O que as pessoas geralmente pretendem ao escrever `index?` como um parâmetro opcional é que elas querem que ambas as chamadas a seguir sejam legais:

```
myForEach([1, 2, 3], (a) => console.log(a));
myForEach([1, 2, 3], (a, i) => console.log(a, i));
```

O que isso *realmente* significa é que *callback pode ser invocado com um argumento*. Em outras palavras, a definição da função diz que a implementação pode ser assim:

```
function myForEach(arr: any[], callback: (arg: any, index?: number) => void) {
  for (let i = 0; i < arr.length; i++) {
    // Não estou a fim de fornecer o índice hoje
    callback(arr[i]);
  }
}
```

Por sua vez, o TypeScript imporá esse significado e emitirá erros que não são realmente possíveis:

```
myForEach([1, 2, 3], (a, i) => {
  console.log(i.toFixed());
'i' é possivelmente 'undefined'.
});
```

Em JavaScript, se você chamar uma função com mais argumentos do que há parâmetros, os argumentos extras simplesmente são ignorados. O TypeScript se comporta da mesma forma. Funções com menos parâmetros (dos mesmos tipos) podem sempre substituir funções com mais parâmetros.

> Regra: Ao escrever um tipo de função para um callback, *nunca* escreva um parâmetro opcional, a menos que você tenha a intenção de *chamar* a função sem passar esse argumento.

## Sobrecargas de Função

Algumas funções JavaScript podem ser chamadas com diferentes contagens e tipos de argumentos. Por exemplo, você pode escrever uma função para produzir uma `Date` que aceita um timestamp (um argumento) ou uma especificação de mês/dia/ano (três argumentos).

Em TypeScript, podemos especificar uma função que pode ser chamada de maneiras diferentes escrevendo *assinaturas de sobrecarga*. Para fazer isso, escreva um número de assinaturas de função (geralmente duas ou mais), seguidas pelo corpo da função:

```
function makeDate(timestamp: number): Date;
function makeDate(m: number, d: number, y: number): Date;
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
  if (d !== undefined && y !== undefined) {
    return new Date(y, mOrTimestamp, d);
  } else {
```

```
    return new Date(mOrTimestamp);
  }
}
const d1 = makeDate(12345678);
const d2 = makeDate(5, 5, 5);
const d3 = makeDate(1, 3);
Nenhuma sobrecarga espera 2 argumentos, mas sobrecargas existem que esperam 1 ou 3
argumentos.
```

Neste exemplo, escrevemos duas sobrecargas: uma aceitando um argumento e outra aceitando três argumentos. Essas duas primeiras assinaturas são chamadas de *assinaturas de sobrecarga*.

Em seguida, escrevemos uma implementação de função com uma assinatura compatível. Funções têm uma *assinatura de implementação*, mas essa assinatura não pode ser chamada diretamente. Mesmo que tenhamos escrito uma função com dois parâmetros opcionais após o obrigatório, ela não pode ser chamada com dois parâmetros!

## Assinaturas de Sobrecarga e a Assinatura de Implementação

Esta é uma fonte comum de confusão. Frequentemente, as pessoas escrevem código como este e não entendem por que há um erro:

```
function fn(x: string): void;
function fn() {
  // ...
}
// Espera-se poder chamar com zero argumentos
fn();
Esperado 1 argumento, mas recebeu 0.
```

Novamente, a assinatura usada para escrever o corpo da função não pode ser “vista” do exterior.

> A assinatura da *implementação* não é visível do exterior. Ao escrever uma função sobrecarregada, você deve sempre ter *duas* ou mais assinaturas acima da implementação da função.

A assinatura de implementação também deve ser *compatível* com as assinaturas de sobrecarga. Por exemplo, essas funções apresentam erros porque a assinatura de implementação não corresponde às sobrecargas de forma correta:

```
function fn(x: boolean): void;
// O tipo do argumento não está correto
function fn(x: string): void;
Esta assinatura de sobrecarga não é compatível com sua assinatura de implementação.
function fn(x: boolean) {}

function fn(x: string): string;
// O tipo de retorno não está correto
function fn(x: number): boolean;
Esta assinatura de sobrecarga não é compatível com sua assinatura de implementação.
function fn(x: string | number) {
  return "oops";
}
```

## Escrevendo Boas Sobrecargas

Assim como em generics, existem algumas diretrizes que você deve seguir ao usar sobrecargas de função. Seguir esses princípios tornará sua função mais fácil de chamar, entender e implementar.

Vamos considerar uma função que retorna o comprimento de uma string *ou* um array:

```
function len(s: string): number;
function len(arr: any[]): number;
function len(x: any) {
  return x.length;
}
```

Essa função está boa; podemos invocá-la com strings ou arrays. No entanto, não podemos invocá-la com um valor que pode ser uma string ou um array, porque o TypeScript só consegue resolver uma chamada de função para uma única sobrecarga:

```
len(""); // OK
len([0]); // OK
len(Math.random() > 0.5 ? "hello" : [0]);
Nenhuma sobrecarga corresponde a esta chamada.
  A sobrecarga 1 de 2, '(s: string): number', deu o seguinte erro.
    Argumento do tipo 'number[] | "hello"' não é atribuível ao parâmetro do tipo
'string'.
      Tipo 'number[]' não é atribuível ao tipo 'string'.
  A sobrecarga 2 de 2, '(arr: any[]): number', deu o seguinte erro.
    Argumento do tipo 'number[] | "hello"' não é atribuível ao parâmetro do tipo
'any[]'.
      Tipo 'string' não é atribuível ao tipo 'any[]'.
```

Como ambas as sobrecargas têm a mesma contagem de argumentos e o mesmo tipo de retorno, podemos em vez disso escrever uma versão não sobrecarregada da função:

```
function len(x: any[] | string) {
  return x.length;
}
```

Isso é muito melhor! Os chamadores podem invocar isso com qualquer um dos tipos de valor, e como um bônus, não precisamos descobrir uma assinatura de implementação correta.

> Sempre prefira parâmetros com tipos de união em vez de sobrecargas, quando possível.

## Declarando `this` em uma Função

O TypeScript inferirá qual deve ser o `this` em uma função por meio da análise de fluxo de código, por exemplo, no seguinte:

```
const user = {
  id: 123,

  admin: false,
  becomeAdmin: function () {
    this.admin = true;
  },
};
```

O TypeScript entende que a função `user.becomeAdmin` tem um `this` correspondente que é o objeto externo `user`. O `this`, *heh*, pode ser suficiente para muitos casos, mas há muitos casos em que você precisa de mais controle sobre qual objeto `this` representa. A especificação do JavaScript afirma que você não pode ter um parâmetro chamado `this`, então o TypeScript usa esse espaço de sintaxe para permitir que você declare o tipo para `this` no corpo da função.

```
interface DB {
  filterUsers(filter: (this: User) => boolean): User[];
}

const db = getDB();
const admins = db.filterUsers(function (this: User) {
  return this.admin;
});
```

Esse padrão é comum em APIs de estilo callback, onde outro objeto normalmente controla quando sua função é chamada. Note que você precisa usar `function` e não funções de seta para obter esse comportamento:

```
interface DB {
  filterUsers(filter: (this: User) => boolean): User[];
}

const db = getDB();
const admins = db.filterUsers(() => this.admin);
A função de seta captura o valor global de 'this'.
O elemento implicitamente tem um tipo 'any' porque o tipo 'typeof globalThis' não
possui uma assinatura de índice.
```

## Outros Tipos a Conhecer

Existem alguns tipos adicionais que você vai querer reconhecer e que aparecem frequentemente ao trabalhar com tipos de função. Como todos os tipos, você pode usá-los em qualquer lugar, mas eles são especialmente relevantes no contexto de funções.

### void

`void` representa o valor de retorno de funções que não retornam um valor. É o tipo inferido sempre que uma função não tem nenhuma instrução `return`, ou não retorna nenhum valor explícito dessas instruções de retorno:

```
// O tipo de retorno inferido é void
function noop() {
  return;
}
```

Em JavaScript, uma função que não retorna nenhum valor irá implicitamente retornar o valor `undefined`. No entanto, `void` e `undefined` não são a mesma coisa no TypeScript. Existem mais detalhes no final deste capítulo.

> `void` não é o mesmo que `undefined`.

### `object`

O tipo especial `object` refere-se a qualquer valor que não seja um primitivo (`string`, `number`, `bigint`, `boolean`, `symbol`, `null` ou `undefined`). Isso é diferente do *tipo de objeto vazio* `{ }`, e também diferente do tipo global `Object`. É muito provável que você nunca use `Object`.

> `object` não é `Object`. **Sempre** use `object`!

Note que em JavaScript, valores de função são objetos: eles têm propriedades, têm `Object.prototype` em sua cadeia de protótipos, são `instanceof Object`, você pode chamar `Object.keys` neles, e assim por diante. Por essa razão, os tipos de função são considerados `objects` no TypeScript.

### `unknown`

O tipo `unknown` representa *qualquer* valor. Isso é semelhante ao tipo `any`, mas é mais seguro porque não é legal fazer nada com um valor `unknown`:

```
function f1(a: any) {
  a.b(); // OK
}
function f2(a: unknown) {
  a.b();
'a' é do tipo 'unknown'.
}
```

Isso é útil ao descrever tipos de função, pois você pode descrever funções que aceitam qualquer valor sem ter valores `any` no corpo da sua função.

Por outro lado, você pode descrever uma função que retorna um valor de tipo desconhecido:

```
function safeParse(s: string): unknown {
  return JSON.parse(s);
}

// Precisa ter cuidado com 'obj'!
const obj = safeParse(someRandomString);
```

### `never`

Algumas funções *nunca* retornam um valor:

```
function fail(msg: string): never {
  throw new Error(msg);
}
```

O tipo `never` representa valores que *nunca* são observados. Em um tipo de retorno, isso significa que a função lança uma exceção ou termina a execução do programa.

`never` também aparece quando o TypeScript determina que não há nada mais em uma união.

```
function fn(x: string | number) {
  if (typeof x === "string") {
    // faz algo
  } else if (typeof x === "number") {
    // faz algo diferente
  } else {
    x; // tem o tipo 'never'!
```

```
  }
}
```

`Function`

O tipo global `Function` descreve propriedades como `bind`, `call`, `apply`, e outras presentes em todos os valores de função no JavaScript. Ele também tem a propriedade especial de que valores do tipo `Function` podem sempre ser chamados; essas chamadas retornam `any`:

```
function doSomething(f: Function) {
  return f(1, 2, 3);
}
```

Isso é uma *chamada de função não tipada* e geralmente é melhor evitá-la devido ao retorno inseguro do tipo `any`.

Se você precisa aceitar uma função arbitrária, mas não pretende chamá-la, o tipo `() => void` é geralmente mais seguro.

# Parâmetros Rest e Argumentos

## Parâmetros Rest

Além de usar parâmetros opcionais ou sobrecargas para criar funções que podem aceitar uma variedade de contagens fixas de argumentos, também podemos definir funções que aceitam um número *ilimitado* de argumentos usando *parâmetros rest*.

Um parâmetro rest aparece após todos os outros parâmetros e usa a sintaxe `...`:

```
function multiply(n: number, ...m: number[]) {
  return m.map((x) => n * x);
}
// 'a' recebe o valor [10, 20, 30, 40]
const a = multiply(10, 1, 2, 3, 4);
```

No TypeScript, a anotação de tipo nesses parâmetros é implicitamente `any[]` em vez de `any`, e qualquer anotação de tipo dada deve ser da forma `Array<T>` ou `T[]`, ou um tipo de tupla (que aprenderemos mais tarde).

## Argumentos Rest

Por outro lado, podemos *fornecer* um número variável de argumentos a partir de um objeto iterável (por exemplo, um array) usando a sintaxe de spread. Por exemplo, o método `push` de arrays aceita qualquer número de argumentos:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
arr1.push(...arr2);
```

Note que, em geral, o TypeScript não assume que arrays são imutáveis. Isso pode levar a alguns comportamentos surpreendentes:

```
// O tipo inferido é number[] -- "um array com zero ou mais números",
// não especificamente dois números
const args = [8, 5];
const angle = Math.atan2(...args);
```

```
Um argumento de spread deve ter um tipo de tupla ou ser passado para um parâmetro rest.
```

A melhor solução para essa situação depende um pouco do seu código, mas, em geral, um contexto `const` é a solução mais direta:

```
// Inferido como tupla de 2 elementos
const args = [8, 5] as const;
// OK
const angle = Math.atan2(...args);
```

Usar argumentos rest pode exigir ativar downlevelIteration ao direcionar para ambientes mais antigos.

## Desestruturação de Parâmetros

Você pode usar desestruturação de parâmetros para desempacotar convenientemente objetos fornecidos como argumento em uma ou mais variáveis locais no corpo da função. Em JavaScript, fica assim:

```
function sum({ a, b, c }) {
  console.log(a + b + c);
}
sum({ a: 10, b: 3, c: 9 });
```

A anotação de tipo para o objeto vai após a sintaxe de desestruturação:

```
function sum({ a, b, c }: { a: number; b: number; c: number }) {
  console.log(a + b + c);
}
```

Isso pode parecer um pouco verboso, mas você pode usar um tipo nomeado aqui também:

```
// Mesmo que o exemplo anterior
type ABC = { a: number; b: number; c: number };
function sum({ a, b, c }: ABC) {
  console.log(a + b + c);
}
```

## Atribuição de Funções

### Tipo de Retorno

O tipo de retorno `void` para funções pode produzir um comportamento incomum, mas esperado.

A tipagem contextual com um tipo de retorno `void` não força funções a não retornarem algo. Outra forma de dizer isso é que um tipo de função contextual com um retorno `void` (tipo `voidFunc = () => void`) pode retornar *qualquer* outro valor, mas será ignorado.

Assim, as seguintes implementações do tipo `() => void` são válidas:

```
type voidFunc = () => void;

const f1: voidFunc = () => {
  return true;
};

const f2: voidFunc = () => true;
```

```
const f3: voidFunc = function () {
  return true;
};
```

E quando o valor de retorno de uma dessas funções é atribuído a outra variável, ele manterá o tipo `void`:

```
const v1 = f1();

const v2 = f2();

const v3 = f3();
```

Esse comportamento existe para que o seguinte código seja válido, mesmo que `Array.prototype.push` retorne um número e o método `Array.prototype.forEach` espere uma função com um tipo de retorno `void`.

```
const src = [1, 2, 3];
const dst = [0];

src.forEach((el) => dst.push(el));
```

Há um outro caso especial a ser observado: quando uma definição de função literal tem um tipo de retorno `void`, essa função não deve retornar nada.

```
function f2(): void {
  // @ts-expect-error
  return true;
}

const f3 = function (): void {
  // @ts-expect-error
  return true;
};
```

Para mais informações sobre `void`, consulte essas outras entradas de documentação:

* manual v2
* FAQ - “Por que funções que retornam não-void são atribuíveis a funções que retornam void?”