The Concise TypeScript Book

Simone Poggiali

O Concise TypeScript Book

O Concise TypeScript Book fornece uma visão geral abrangente e sucinta das capacidades do TypeScript. Ele oferece explicações claras cobrindo todos os aspectos encontrados na versão mais recente da linguagem, desde seu poderoso sistema de tipos até recursos avançados. Seja você um iniciante ou um desenvolvedor experiente, este livro é um recurso inestimável para aprimorar sua compreensão e proficiência em TypeScript.

Este livro é completamente Gratuito e de Código Aberto.

Acredito que educação técnica de alta qualidade deve ser acessível a todos, por isso mantenho este livro gratuito e aberto.

Se o livro ajudou você a corrigir um bug, entender um conceito complicado ou avançar em sua carreira, por favor considere apoiar meu trabalho pagando o quanto quiser (preço sugerido: 15 USD) ou patrocinando um café. Seu apoio me ajuda a manter o conteúdo atualizado e expandi-lo com novos exemplos e explicações mais profundas.

Traduções

Este livro foi traduzido para várias versões de idiomas, incluindo:

Chinês

Italiano

Português Brasileiro

Downloads e website

Você também pode baixar a versão Epub:

https://github.com/gibbok/typescript-book/tree/main/downloads

Uma versão online está disponível em:

https://gibbok.github.io/typescript-book

Índice

Introdução

Bem-vindo ao The Concise TypeScript Book! Este guia equipa você com conhecimento essencial e habilidades práticas para desenvolvimento TypeScript eficaz. Descubra conceitos-chave e técnicas para escrever código limpo e robusto. Seja você um iniciante ou um desenvolvedor experiente, este livro serve como um guia abrangente e uma referência prática para aproveitar o poder do TypeScript em seus projetos.

Este livro cobre o TypeScript 5.2.

Sobre o Autor

Simone Poggiali é um Staff Engineer experiente com paixão por escrever código de nível profissional desde os anos 90. Ao longo de sua carreira internacional, ele contribuiu para inúmeros projetos para uma ampla gama de clientes, de startups a grandes organizações. Empresas notáveis como HelloFresh, Siemens, O2, Leroy Merlin e Snowplow se beneficiaram de sua expertise e dedicação.

Você pode entrar em contato com Simone Poggiali nas seguintes plataformas:

- LinkedIn: https://www.linkedin.com/in/simone-poggiali
- GitHub: https://github.com/gibbok
- X.com: https://x.com/gibbok_coding
- Email: gibbok.coding gmail.com

Introdução ao TypeScript

O que é TypeScript?

TypeScript é uma linguagem de programação fortemente tipada que se baseia em JavaScript. Foi originalmente projetado por Anders Hejlsberg em 2012 e atualmente é desenvolvido e mantido pela Microsoft como um projeto de código aberto.

TypeScript compila para JavaScript e pode ser executado em qualquer runtime JavaScript (por exemplo, um navegador ou servidor Node.js).

TypeScript suporta múltiplos paradigmas de programação, como funcional, genérico, imperativo e orientado a objetos. TypeScript não é uma linguagem interpretada nem compilada.

Por que TypeScript?

TypeScript é uma linguagem fortemente tipada que ajuda a prevenir erros comuns de programação e evitar certos tipos de erros em tempo de execução antes que o programa seja executado.

Uma linguagem fortemente tipada permite ao desenvolvedor especificar várias restrições e comportamentos do programa nas definições de tipo de dados, facilitando a capacidade de verificar a correção do software e prevenir defeitos. Isso é especialmente valioso em aplicações de larga escala.

Alguns dos benefícios do TypeScript:

- Tipagem estática, opcionalmente fortemente tipada
- Inferência de Tipo
- Acesso aos recursos do ES6 e ES7
- Compatibilidade Cross-Platform e Cross-browser
- Suporte de ferramentas com IntelliSense

TypeScript e JavaScript

TypeScript é escrito em arquivos `.ts` ou `.tsx`, enquanto arquivos JavaScript são escritos em `.js` ou `.jsx`.

Arquivos com a extensão `.tsx` ou `.jsx` podem conter JavaScript Syntax Extension JSX, que é usado no React para desenvolvimento de UI.

TypeScript é um superconjunto tipado de JavaScript (ECMAScript 2015) em termos de sintaxe. Todo código JavaScript é código TypeScript válido, mas o inverso nem sempre é verdadeiro.

Por exemplo, considere uma função em um arquivo JavaScript com a extensão `.js`, como a seguinte:

```
const sum = (a, b) => a + b;
```

A função pode ser convertida e usada em TypeScript mudando a extensão do arquivo para `.ts`. No entanto, se a mesma função for anotada com tipos TypeScript, ela não poderá ser executada em nenhum runtime JavaScript sem compilação. O seguinte código TypeScript produzirá um erro de sintaxe se não for compilado:

```
const sum = (a: number, b: number): number => a + b;
```

TypeScript foi projetado para detectar possíveis exceções que podem ocorrer em tempo de execução durante o tempo de compilação, fazendo com que o desenvolvedor defina a intenção com anotações de tipo. Além disso, o TypeScript também pode detectar problemas se nenhuma anotação de tipo for fornecida. Por exemplo, o seguinte trecho de código não especifica nenhum tipo TypeScript:

```
const items = [{ x: 1 }, { x: 2 }];
const result = items.filter(item => item.y);
```

Neste caso, o TypeScript detecta um erro e relata:

```
Property 'y' does not exist on type '{ x: number; }'.
```

O sistema de tipos do TypeScript é amplamente influenciado pelo comportamento em tempo de execução do JavaScript. Por exemplo, o operador de adição (+), que em JavaScript pode realizar concatenação de string ou adição numérica, é modelado da mesma forma no TypeScript:

```
const result = '1' + 1; // O resultado é do tipo string
```

A equipe por trás do TypeScript tomou uma decisão deliberada de sinalizar uso incomum de JavaScript como erros. Por exemplo, considere o seguinte código JavaScript válido:

```
const result = 1 + true; // Em JavaScript, o resultado é igual a 2
```

No entanto, o TypeScript lança um erro:

```
Operator '+' cannot be applied to types 'number' and 'boolean'.
```

Este erro ocorre porque o TypeScript aplica rigorosamente a compatibilidade de tipos e, neste caso, identifica uma operação inválida entre um number e um boolean.

Geração de Código TypeScript

O compilador TypeScript tem duas responsabilidades principais: verificar erros de tipo e compilar para JavaScript. Esses dois processos são independentes um do outro. Tipos não afetam a execução do código em um runtime JavaScript, pois são completamente apagados durante a compilação. O TypeScript ainda pode gerar JavaScript mesmo na presença de erros de tipo. Aqui está um exemplo de código TypeScript com um erro de tipo:

```
const add = (a: number, b: number): number => a + b;
const result = add('x', 'y'); // Argument of type 'string' is not
assignable to parameter of type 'number'.
```

No entanto, ainda pode produzir saída JavaScript executável:

```
'use strict';
const add = (a, b) => a + b;
const result = add('x', 'y'); // xy
```

Não é possível verificar tipos TypeScript em tempo de execução. Por exemplo:

```
interface Animal {
    name: string;
}
interface Dog extends Animal {
    bark: () => void;
}
interface Cat extends Animal {
    meow: () => void;
}
const makeNoise = (animal: Animal) => {
```

```
    if (animal instanceof Dog) {
        // 'Dog' only refers to a type, but is being used as a
value here.
        // ...
    }
};
```

Como os tipos são apagados após a compilação, não há como executar este código em JavaScript. Para reconhecer tipos em tempo de execução, precisamos usar outro mecanismo. O TypeScript fornece várias opções, sendo uma comum a “união discriminada”. Por exemplo:

```
interface Dog {
    kind: 'dog'; // União discriminada
    bark: () => void;
}
interface Cat {
    kind: 'cat'; // União discriminada
    meow: () => void;
}
type Animal = Dog | Cat;

const makeNoise = (animal: Animal) => {
    if (animal.kind === 'dog') {
        animal.bark();
    } else {
        animal.meow();
    }
};

const dog: Dog = {
    kind: 'dog',
    bark: () => console.log('bark'),
};
makeNoise(dog);
```

A propriedade “kind” é um valor que pode ser usado em tempo de execução para distinguir entre objetos em JavaScript.

Também é possível que um valor em tempo de execução tenha um tipo diferente daquele declarado na declaração de tipo. Por exemplo,

se o desenvolvedor interpretou mal um tipo de API e o anotou incorretamente.

TypeScript é um superconjunto de JavaScript, então a palavra-chave "class" pode ser usada como um tipo e valor em tempo de execução.

```
class Animal {
    constructor(public name: string) {}
}
class Dog extends Animal {
    constructor(
        public name: string,
        public bark: () => void
    ) {
        super(name);
    }
}
class Cat extends Animal {
    constructor(
        public name: string,
        public meow: () => void
    ) {
        super(name);
    }
}
type Mammal = Dog | Cat;

const makeNoise = (mammal: Mammal) => {
    if (mammal instanceof Dog) {
        mammal.bark();
    } else {
        mammal.meow();
    }
};

const dog = new Dog('Fido', () => console.log('bark'));
makeNoise(dog);
```

Em JavaScript, uma "class" tem uma propriedade "prototype", e o operador "instanceof" pode ser usado para testar se a propriedade prototype de um construtor aparece em qualquer lugar na cadeia de protótipos de um objeto.

TypeScript não tem efeito no desempenho em tempo de execução, pois todos os tipos serão apagados. No entanto, o TypeScript introduz alguma sobrecarga de tempo de compilação.

JavaScript Moderno Agora (Downleveling)

TypeScript pode compilar código para qualquer versão lançada de JavaScript desde ECMAScript 3 (1999). Isso significa que o TypeScript pode transpilar código dos recursos JavaScript mais recentes para versões mais antigas, um processo conhecido como Downleveling. Isso permite o uso de JavaScript moderno enquanto mantém máxima compatibilidade com ambientes de runtime mais antigos.

É importante notar que durante a transpilação para uma versão mais antiga de JavaScript, o TypeScript pode gerar código que poderia incorrer em uma sobrecarga de desempenho em comparação com implementações nativas.

Aqui estão alguns dos recursos JavaScript modernos que podem ser usados no TypeScript:

- Módulos ECMAScript em vez de callbacks estilo AMD “define” ou instruções CommonJS “require”.
- Classes em vez de prototypes.
- Declaração de variáveis usando “let” ou “const” em vez de “var”.
- Loop “for-of” ou “.forEach” em vez do loop “for” tradicional.
- Arrow functions em vez de expressões de função.
- Atribuição de desestruturação.
- Nomes de propriedade/método abreviados e nomes de propriedade computados.
- Parâmetros de função padrão.

Ao aproveitar esses recursos JavaScript modernos, os desenvolvedores podem escrever código mais expressivo e conciso no TypeScript.

Começando com TypeScript

Instalação

O Visual Studio Code fornece excelente suporte para a linguagem TypeScript, mas não inclui o compilador TypeScript. Para instalar o compilador TypeScript, você pode usar um gerenciador de pacotes como npm ou yarn:

```
npm install typescript ––save–dev
```

ou

```
yarn add typescript ––dev
```

Certifique-se de fazer commit do arquivo lockfile gerado para garantir que todos os membros da equipe usem a mesma versão do TypeScript.

Para executar o compilador TypeScript, você pode usar os seguintes comandos

```
npx tsc
```

ou

```
yarn tsc
```

É recomendado instalar o TypeScript por projeto em vez de globalmente, pois isso fornece um processo de build mais previsível. No entanto, para ocasiões pontuais, você pode usar o seguinte comando:

```
npx tsc
```

ou instalando globalmente:

```
npm install -g typescript
```

Se você estiver usando o Microsoft Visual Studio, pode obter o TypeScript como um pacote no NuGet para seus projetos MSBuild. No Console do Gerenciador de Pacotes NuGet, execute o seguinte comando:

```
Install-Package Microsoft.TypeScript.MSBuild
```

Durante a instalação do TypeScript, dois executáveis são instalados: “tsc” como o compilador TypeScript e “tsserver” como o servidor standalone TypeScript. O servidor standalone contém o compilador e os serviços de linguagem que podem ser utilizados por editores e IDEs para fornecer conclusão de código inteligente.

Além disso, existem vários transpiladores compatíveis com TypeScript disponíveis, como Babel (via plugin) ou swc. Esses transpiladores podem ser usados para converter código TypeScript em outras linguagens ou versões de destino.

Configuração

O TypeScript pode ser configurado usando as opções CLI do tsc ou utilizando um arquivo de configuração dedicado chamado tsconfig.json colocado na raiz do projeto.

Para gerar um arquivo tsconfig.json pré-preenchido com configurações recomendadas, você pode usar o seguinte comando:

```
tsc --init
```

Ao executar o comando `tsc` localmente, o TypeScript compilará o código usando a configuração especificada no arquivo tsconfig.json

mais próximo.

Aqui estão alguns exemplos de comandos CLI que executam com as configurações padrão:

```
tsc main.ts // Compila um arquivo específico (main.ts) para
JavaScript
tsc src/*.ts // Compila quaisquer arquivos .ts na pasta 'src' para
JavaScript
tsc app.ts util.ts --outfile index.js // Compila dois arquivos
TypeScript (app.ts e util.ts) em um único arquivo JavaScript
(index.js)
```

Arquivo de Configuração TypeScript

Um arquivo tsconfig.json é usado para configurar o Compilador TypeScript (tsc). Geralmente, ele é adicionado à raiz do projeto, junto com o arquivo `package.json`.

Notas:

- tsconfig.json aceita comentários mesmo sendo no formato json.
- É aconselhável usar este arquivo de configuração em vez das opções de linha de comando.

No seguinte link você pode encontrar a documentação completa e seu schema:

https://www.typescriptlang.org/tsconfig

https://www.typescriptlang.org/tsconfig/

A seguir está uma lista das configurações comuns e úteis:

target

A propriedade “target” é usada para especificar qual versão do JavaScript ECMAScript seu TypeScript deve emitir/compilar. Para navegadores modernos ES6 é uma boa opção, para navegadores mais antigos, ES5 é recomendado.

lib

A propriedade “lib” é usada para especificar quais arquivos de biblioteca incluir no momento da compilação. O TypeScript inclui automaticamente APIs para recursos especificados na propriedade “target”, mas é possível omitir ou escolher bibliotecas específicas para necessidades particulares. Por exemplo, se você está trabalhando em um projeto de servidor, você poderia excluir a biblioteca “DOM”, que é útil apenas em um ambiente de navegador.

strict

A propriedade “strict” habilita garantias mais fortes e melhora a segurança de tipos. É aconselhável sempre incluir esta propriedade no arquivo tsconfig.json do seu projeto. Habilitar a propriedade “strict” permite ao TypeScript:

- Emitir código usando “use strict” para cada arquivo fonte.
- Considerar “null” e “undefined” no processo de verificação de tipo.
- Desabilitar o uso do tipo “any” quando nenhuma anotação de tipo está presente.
- Gerar um erro no uso da expressão “this”, que de outra forma implicaria o tipo “any”.

module

A propriedade “module” define o sistema de módulos suportado para o programa compilado. Durante o runtime, um carregador de módulos é usado para localizar e executar dependências com base no sistema de módulos especificado.

Os carregadores de módulos mais comuns usados em JavaScript são Node.js CommonJS para aplicações server-side e RequireJS para módulos AMD em aplicações web baseadas em navegador. O TypeScript pode emitir código para vários sistemas de módulos, incluindo UMD, System, ESNext, ES2015/ES6 e ES2020.

Nota: O sistema de módulos deve ser escolhido com base no ambiente de destino e no mecanismo de carregamento de módulos disponível nesse ambiente.

moduleResolution

A propriedade “moduleResolution” especifica a estratégia de resolução de módulos. Use “node” para código TypeScript moderno, a estratégia “classic” é usada apenas para versões antigas do TypeScript (antes da 1.6).

esModuleInterop

A propriedade “esModuleInterop” permite importação padrão de módulos CommonJS que não exportaram usando a propriedade “default”, esta propriedade fornece um shim para garantir compatibilidade no JavaScript emitido. Depois de habilitar esta opção, podemos usar `import MyLibrary from "my-library"` em vez de `import * as MyLibrary from "my-library"`.

jsx

A propriedade “jsx” se aplica apenas a arquivos .tsx usados no ReactJS e controla como as construções JSX são compiladas em JavaScript. Uma opção comum é “preserve” que compilará para um arquivo .jsx mantendo o JSX inalterado para que possa ser passado para diferentes ferramentas como Babel para transformações adicionais.

skipLibCheck

A propriedade “skipLibCheck” impedirá o TypeScript de verificar tipos de pacotes terceiros importados inteiros. Esta propriedade reduzirá o tempo de compilação de um projeto. O TypeScript ainda verificará seu código em relação às definições de tipo fornecidas por esses pacotes.

files

A propriedade “files” indica ao compilador uma lista de arquivos que devem sempre ser incluídos no programa.

include

A propriedade “include” indica ao compilador uma lista de arquivos que gostaríamos de incluir. Esta propriedade permite padrões semelhantes a glob, como “*” *para qualquer subdiretório,* “” para qualquer nome de arquivo e “?” para caracteres opcionais.

exclude

A propriedade “exclude” indica ao compilador uma lista de arquivos que não devem ser incluídos na compilação. Isso pode incluir arquivos como “node_modules” ou arquivos de teste. Nota: tsconfig.json permite comentários.

importHelpers

O TypeScript usa código auxiliar ao gerar código para certos recursos JavaScript avançados ou de nível inferior. Por padrão, esses auxiliares são duplicados em arquivos que os usam. A opção `importHelpers` importa esses auxiliares do módulo `tslib`, tornando a saída JavaScript mais eficiente.

Conselhos para Migração para TypeScript

Para projetos grandes, é recomendado adotar uma transição gradual onde código TypeScript e JavaScript coexistirão inicialmente. Apenas projetos pequenos podem ser migrados para TypeScript de uma só vez.

O primeiro passo desta transição é introduzir o TypeScript no processo da cadeia de build. Isso pode ser feito usando a opção do compilador “allowJs”, que permite que arquivos .ts e .tsx coexistam com arquivos JavaScript existentes. Como o TypeScript recorrerá a um tipo “any” para uma variável quando não puder inferir o tipo de arquivos JavaScript, é recomendado desabilitar “noImplicitAny” nas opções do compilador no início da migração.

O segundo passo é garantir que seus testes JavaScript funcionem junto com arquivos TypeScript para que você possa executar testes à medida que converte cada módulo. Se você está usando Jest, considere usar `ts-jest`, que permite testar projetos TypeScript com Jest.

O terceiro passo é incluir declarações de tipo para bibliotecas de terceiros em seu projeto. Essas declarações podem ser encontradas agrupadas ou no DefinitelyTyped. Você pode procurá-las usando https://www.typescriptlang.org/dt/search e instalá-las usando:

```
npm install --save-dev @types/package-name
```

ou

```
yarn add --dev @types/package-name.
```

O quarto passo é migrar módulo por módulo com uma abordagem bottom-up, seguindo seu Grafo de Dependências começando pelas folhas. A ideia é começar a converter Módulos que não dependem de outros Módulos. Para visualizar os grafos de dependência, você pode usar a ferramenta “madge”.

Bons módulos candidatos para essas conversões iniciais são funções utilitárias e código relacionado a APIs externas ou especificações. É possível gerar automaticamente definições de tipo TypeScript de contratos Swagger, GraphQL ou schemas JSON para serem incluídos em seu projeto.

Quando não há especificações ou schemas oficiais disponíveis, você pode gerar tipos de dados brutos, como JSON retornado por um servidor. No entanto, é recomendado gerar tipos de especificações em vez de dados para evitar perder casos extremos.

Durante a migração, abstenha-se de refatoração de código e concentre-se apenas em adicionar tipos aos seus módulos.

O quinto passo é habilitar “noImplicitAny”, que forçará que todos os tipos sejam conhecidos e definidos, fornecendo uma melhor experiência TypeScript para seu projeto.

Durante a migração, você pode usar a diretiva `@ts-check`, que habilita a verificação de tipo TypeScript em um arquivo JavaScript. Esta diretiva fornece uma versão flexível de verificação de tipo e pode ser usada inicialmente para identificar problemas em arquivos JavaScript. Quando `@ts-check` é incluído em um arquivo, o TypeScript tentará deduzir definições usando comentários no estilo JSDoc. No entanto, considere usar anotações JSDoc apenas em um estágio muito inicial da migração.

Considere manter o valor padrão de `noEmitOnError` em seu tsconfig.json como false. Isso permitirá que você gere código-fonte JavaScript mesmo se erros forem relatados.

Explorando o Sistema de Tipos

O Serviço de Linguagem TypeScript

O Serviço de Linguagem TypeScript, também conhecido como tsserver, oferece vários recursos, como relatório de erros, diagnósticos, compile-on-save, renomeação, ir para definição, listas de conclusão, ajuda de assinatura e muito mais. É usado principalmente por ambientes de desenvolvimento integrados (IDEs) para fornecer suporte IntelliSense. Ele se integra perfeitamente com o Visual Studio Code e é utilizado por ferramentas como Conquer of Completion (Coc).

Os desenvolvedores podem aproveitar uma API dedicada e criar seus próprios plugins de serviço de linguagem personalizados para aprimorar a experiência de edição TypeScript. Isso pode ser particularmente útil para implementar recursos especiais de linting ou habilitar conclusão automática para uma linguagem de template personalizada.

Um exemplo de plugin personalizado do mundo real é o “typescript-styled-plugin”, que fornece relatório de erros de sintaxe e suporte IntelliSense para propriedades CSS em styled components.

Para mais informações e guias de início rápido, você pode consultar o Wiki oficial do TypeScript no GitHub:
https://github.com/microsoft/TypeScript/wiki/

Tipagem Estrutural

O TypeScript é baseado em um sistema de tipos estrutural. Isso significa que a compatibilidade e equivalência de tipos são determinadas pela estrutura ou definição real do tipo, em vez de seu nome ou local de declaração, como em sistemas de tipos nominativos como C# ou C.

O sistema de tipos estrutural do TypeScript foi projetado com base em como o sistema de duck typing dinâmico do JavaScript funciona durante o runtime.

O seguinte exemplo é código TypeScript válido. Como você pode observar, “X” e “Y” têm o mesmo membro “a”, mesmo que tenham nomes de declaração diferentes. Os tipos são determinados por suas estruturas e, neste caso, como as estruturas são as mesmas, eles são compatíveis e válidos.

```
type X = {
    a: string;
};
type Y = {
    a: string;
};
const x: X = { a: 'a' };
const y: Y = x; // Válido
```

Regras Fundamentais de Comparação do TypeScript

O processo de comparação do TypeScript é recursivo e executado em tipos aninhados em qualquer nível.

Um tipo “X” é compatível com “Y” se “Y” tiver pelo menos os mesmos membros que “X”.

```
type X = {
    a: string;
};
const y = { a: 'A', b: 'B' }; // Válido, pois tem pelo menos os
mesmos membros que X
const r: X = y;
```

Parâmetros de função são comparados por tipos, não por seus nomes:

```
type X = (a: number) => void;
type Y = (a: number) => void;
let x: X = (j: number) => undefined;
let y: Y = (k: number) => undefined;
y = x; // Válido
x = y; // Válido
```

Os tipos de retorno de função devem ser os mesmos:

```
type X = (a: number) => undefined;
type Y = (a: number) => number;
let x: X = (a: number) => undefined;
let y: Y = (a: number) => 1;
y = x; // Inválido
x = y; // Inválido
```

O tipo de retorno de uma função fonte deve ser um subtipo do tipo de retorno de uma função alvo:

```
let x = () => ({ a: 'A' });
let y = () => ({ a: 'A', b: 'B' });
x = y; // Válido
y = x; // Inválido, membro b está faltando
```

Descartar parâmetros de função é permitido, pois é uma prática comum em JavaScript, por exemplo, usando "Array.prototype.map()":

```
[1, 2, 3].map((element, _index, _array) => element + 'x');
```

Portanto, as seguintes declarações de tipo são completamente válidas:

```
type X = (a: number) => undefined;
type Y = (a: number, b: number) => undefined;
let x: X = (a: number) => undefined;
let y: Y = (a: number) => undefined; // Parâmetro b faltando
y = x; // Válido
```

Quaisquer parâmetros opcionais adicionais do tipo fonte são válidos:

```
type X = (a: number, b?: number, c?: number) => undefined;
type Y = (a: number) => undefined;
let x: X = a => undefined;
let y: Y = a => undefined;
y = x; // Válido
x = y; // Válido
```

Quaisquer parâmetros opcionais do tipo alvo sem parâmetros correspondentes no tipo fonte são válidos e não são um erro:

```
type X = (a: number) => undefined;
type Y = (a: number, b?: number) => undefined;
let x: X = a => undefined;
let y: Y = a => undefined;
y = x; // Válido
x = y; // Válido
```

O parâmetro rest é tratado como uma série infinita de parâmetros opcionais:

```
type X = (a: number, ...rest: number[]) => undefined;
let x: X = a => undefined; // válido
```

Funções com overloads são válidas se a assinatura de overload for compatível com sua assinatura de implementação:

```
function x(a: string): void;
function x(a: string, b: number): void;
function x(a: string, b?: number): void {
    console.log(a, b);
}
x('a'); // Válido
x('a', 1); // Válido
```

```
function y(a: string): void; // Inválido, não compatível com a
assinatura de implementação
function y(a: string, b: number): void;
function y(a: string, b: number): void {
    console.log(a, b);
}
y('a');
y('a', 1);
```

A comparação de parâmetros de função é bem-sucedida se os parâmetros de origem e destino forem atribuíveis a supertipos ou subtipos (bivariância).

```
// Supertipo
class X {
    a: string;
    constructor(value: string) {
        this.a = value;
    }
}
// Subtipo
class Y extends X {}
// Subtipo
class Z extends X {}

type GetA = (x: X) => string;
const getA: GetA = x => x.a;

// Bivariância aceita supertipos
console.log(getA(new X('x'))); // Válido
console.log(getA(new Y('Y'))); // Válido
console.log(getA(new Z('z'))); // Válido
```

Enums são comparáveis e válidos com numbers e vice-versa, mas comparar valores Enum de diferentes tipos Enum é inválido.

```
enum X {
    A,
    B,
}
enum Y {
    A,
    B,
```

```
    C,
}
const xa: number = X.A; // Válido
const ya: Y = 0; // Válido
X.A === Y.A; // Inválido
```

Instâncias de uma classe estão sujeitas a uma verificação de compatibilidade para seus membros private e protected:

```
class X {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}

class Y {
    private a: string;
    constructor(value: string) {
        this.a = value;
    }
}

let x: X = new Y('y'); // Inválido
```

A verificação de comparação não leva em consideração a hierarquia de herança diferente, por exemplo:

```
class X {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}
class Y extends X {
    public a: string;
    constructor(value: string) {
        super(value);
        this.a = value;
    }
}
class Z {
    public a: string;
```

```
    constructor(value: string) {
        this.a = value;
    }
}
let x: X = new X('x');
let y: Y = new Y('y');
let z: Z = new Z('z');
x === y; // Válido
x === z; // Válido mesmo que z seja de uma hierarquia de herança
diferente
```

Generics são comparados usando suas estruturas com base no tipo resultante após aplicar o parâmetro genérico, apenas o resultado final é comparado como um tipo não genérico.

```
interface X<T> {
    a: T;
}
let x: X<number> = { a: 1 };
let y: X<string> = { a: 'a' };
x === y; // Inválido, pois o argumento de tipo é usado na estrutura
final

interface X<T> {}
const x: X<number> = 1;
const y: X<string> = 'a';
x === y; // Válido, pois o argumento de tipo não é usado na
estrutura final
```

Quando generics não têm seu argumento de tipo especificado, todos os argumentos não especificados são tratados como tipos com “any”:

```
type X = <T>(x: T) => T;
type Y = <K>(y: K) => K;
let x: X = x => x;
let y: Y = y => y;
x = y; // Válido
```

Lembre-se:

```
let a: number = 1;
let b: number = 2;
a = b; // Válido, tudo é atribuível a si mesmo
```

```
let c: any;
c = 1; // Válido, todos os tipos são atribuíveis a any

let d: unknown;
d = 1; // Válido, todos os tipos são atribuíveis a unknown

let e: unknown;
let e1: unknown = e; // Válido, unknown é atribuível apenas a si
mesmo e any
let e2: any = e; // Válido
let e3: number = e; // Inválido

let f: never;
f = 1; // Inválido, nada é atribuível a never

let g: void;
let g1: any;
g = 1; // Inválido, void não é atribuível a ou de nada exceto any
g = g1; // Válido
```

Observe que quando “strictNullChecks” está habilitado, “null” e “undefined” são tratados de forma semelhante a “void”; caso contrário, eles são semelhantes a “never”.

Tipos como Conjuntos

No TypeScript, um tipo é um conjunto de valores possíveis. Este conjunto também é referido como o domínio do tipo. Cada valor de um tipo pode ser visto como um elemento em um conjunto. Um tipo estabelece as restrições que cada elemento no conjunto deve satisfazer para ser considerado um membro desse conjunto. A tarefa primária do TypeScript é verificar e confirmar se um conjunto é um subconjunto de outro.

O TypeScript suporta vários tipos de conjuntos:

Termo do conjunto	TypeScript	Notas
Conjunto vazio	never	“never” não contém nada além de si mesmo
Conjunto de elemento único	undefined / null / tipo literal	
Conjunto finito	boolean / union	
Conjunto infinito	string / number / object	
Conjunto universal	any / unknown	Cada elemento é um membro de “any” e cada conjunto é um subconjunto dele / “unknown” é uma contraparte type-safe de “any”

Aqui estão alguns exemplos:

TypeScript	Termo do conjunto	Exemplo
never	∅ (conjunto vazio)	const x: never = ‘x’; // Erro: Type ‘string’ is not assignable to type ‘never’
Tipo literal	Conjunto de elemento único	type X = ‘X’; type Y = 7;

TypeScript	Termo do conjunto	Exemplo
Valor atribuível a T	Valor ∈ T (membro de)	type XY = ‘X’ | ‘Y’;
		const x: XY = ‘X’;
T1 atribuível a T2	T1 ⊆ T2 (subconjunto de)	type XY = ‘X’ | ‘Y’;
		const x: XY = ‘X’;
		const j: XY = ‘J’; // Type “‘J”’ is not assignable to type ‘XY’.
T1 extends T2	T1 ⊆ T2 (subconjunto de)	type X = ‘X’ extends string ? true : false;
T1 | T2	T1 ∪ T2 (união)	type XY = ‘X’ | ‘Y’;
		type JK = 1 | 2;
T1 & T2	T1 ∩ T2 (intersecção)	type X = { a: string }
		type Y = { b: string }
		type XY = X & Y
		const x: XY = { a: ‘a’, b: ‘b’ }
unknown	Conjunto universal	const x: unknown = 1

Uma união, (T1 | T2) cria um conjunto mais amplo (ambos):

```
type X = {
    a: string;
};
type Y = {
    b: string;
};
type XY = X | Y;
const r: XY = { a: 'a', b: 'x' }; // Válido
```

Uma intersecção, (T1 & T2) cria um conjunto mais restrito (apenas compartilhados):

```
type X = {
    a: string;
};
type Y = {
    a: string;
    b: string;
};
type XY = X & Y;
const r: XY = { a: 'a' }; // Inválido
const j: XY = { a: 'a', b: 'b' }; // Válido
```

A palavra-chave `extends` pode ser considerada como um "subconjunto de" neste contexto. Ela define uma restrição para um tipo. O extends usado com um generic, toma o generic como um conjunto infinito e o restringirá a um tipo mais específico. Observe que `extends` não tem nada a ver com hierarquia no sentido OOP (não há este conceito no TypeScript). O TypeScript trabalha com conjuntos e não tem uma hierarquia estrita. Na verdade, como no exemplo abaixo, dois tipos podem se sobrepor sem que nenhum seja um subtipo do outro tipo (o TypeScript considera a estrutura, forma dos objetos).

```
interface X {
    a: string;
}
interface Y extends X {
    b: string;
}
interface Z extends Y {
```

```
    c: string;
}
const z: Z = { a: 'a', b: 'b', c: 'c' };
interface X1 {
    a: string;
}
interface Y1 {
    a: string;
    b: string;
}
interface Z1 {
    a: string;
    b: string;
    c: string;
}
const z1: Z1 = { a: 'a', b: 'b', c: 'c' };

const r: Z1 = z; // Válido
```

Atribuir um tipo: Declarações de Tipo e Asserções de Tipo

Um tipo pode ser atribuído de diferentes maneiras no TypeScript:

Declaração de Tipo

No seguinte exemplo, usamos x: X (“: Type”) para declarar um tipo para a variável x.

```
type X = {
    a: string;
};

// Declaração de tipo
const x: X = {
    a: 'a',
};
```

Se a variável não estiver no formato especificado, o TypeScript reportará um erro. Por exemplo:

```
type X = {
    a: string;
};

const x: X = {
    a: 'a',
    b: 'b', // Erro: Object literal may only specify known
properties
};
```

Asserção de Tipo

É possível adicionar uma asserção usando a palavra-chave `as`. Isso diz ao compilador que o desenvolvedor tem mais informações sobre um tipo e silencia quaisquer erros que possam ocorrer.

Por exemplo:

```
type X = {
    a: string;
};
const x = {
    a: 'a',
    b: 'b',
} as X;
```

No exemplo acima, o objeto x é afirmado como tendo o tipo X usando a palavra-chave as. Isso informa ao compilador TypeScript que o objeto está em conformidade com o tipo especificado, mesmo que tenha uma propriedade adicional b não presente na definição do tipo.

Asserções de tipo são úteis em situações onde um tipo mais específico precisa ser especificado, especialmente ao trabalhar com o DOM. Por exemplo:

```
const myInput = document.getElementById('my_input') as
HTMLInputElement;
```

Aqui, a asserção de tipo as HTMLInputElement é usada para dizer ao TypeScript que o resultado de getElementById deve ser tratado como um HTMLInputElement. Asserções de tipo também podem ser usadas para remapear chaves, como mostrado no exemplo abaixo com template literals:

```
type J<Type> = {
    [Property in keyof Type as `prefix_${string &
        Property}`]: () => Type[Property];
};
type X = {
    a: string;
    b: number;
};
type Y = J<X>;
```

Neste exemplo, o tipo `J<Type>` usa um tipo mapeado com um template literal para remapear as chaves de Type. Ele cria novas propriedades com um “prefix_” adicionado a cada chave, e seus valores correspondentes são funções retornando os valores das propriedades originais.

Vale a pena notar que ao usar uma asserção de tipo, o TypeScript não executará verificação de propriedade excessiva. Portanto, geralmente é preferível usar uma Declaração de Tipo quando a estrutura do objeto é conhecida antecipadamente.

Declarações Ambientes

Declarações ambientes são arquivos que descrevem tipos para código JavaScript, eles têm um formato de nome de arquivo como `.d.ts.`. Eles são geralmente importados e usados para anotar bibliotecas JavaScript existentes ou para adicionar tipos a arquivos JS existentes em seu projeto.

Muitos tipos de bibliotecas comuns podem ser encontrados em: https://github.com/DefinitelyTyped/DefinitelyTyped/

e podem ser instalados usando:

```
npm install --save-dev @types/library-name
```

Para suas Declarações Ambientes definidas, você pode importar usando a referência "triple-slash":

```
/// <reference path="./library-types.d.ts" />
```

Você pode usar Declarações Ambientes mesmo dentro de arquivos JavaScript usando `// @ts-check`.

A palavra-chave `declare` habilita definições de tipo para código JavaScript existente sem importá-lo, servindo como um placeholder para tipos de outro arquivo ou globalmente.

Verificação de Propriedade e Verificação de Propriedade Excessiva

O TypeScript é baseado em um sistema de tipos estrutural, mas a verificação de propriedade excessiva é uma propriedade do TypeScript que permite verificar se um objeto tem as propriedades exatas especificadas no tipo.

A Verificação de Propriedade Excessiva é realizada ao atribuir literais de objeto a variáveis ou ao passá-los como argumentos para a propriedade excessiva da função, por exemplo.

```
type X = {
    a: string;
};
const y = { a: 'a', b: 'b' };
```

```
const x: X = y; // Válido por causa da tipagem estrutural
const w: X = { a: 'a', b: 'b' }; // Inválido por causa da
verificação de propriedade excessiva
```

Tipos Fracos

Um tipo é considerado fraco quando contém nada além de um conjunto de todas as propriedades opcionais:

```
type X = {
    a?: string;
    b?: string;
};
```

O TypeScript considera um erro atribuir qualquer coisa a um tipo fraco quando não há sobreposição, por exemplo, o seguinte lança um erro:

```
type Options = {
    a?: string;
    b?: string;
};

const fn = (options: Options) => undefined;

fn({ c: 'c' }); // Inválido
```

Embora não seja recomendado, se necessário, é possível contornar esta verificação usando asserção de tipo:

```
type Options = {
    a?: string;
    b?: string;
};
const fn = (options: Options) => undefined;
fn({ c: 'c' } as Options); // Válido
```

Ou adicionando `unknown` à assinatura de índice ao tipo fraco:

```ts
type Options = {
    [prop: string]: unknown;
    a?: string;
    b?: string;
};

const fn = (options: Options) => undefined;
fn({ c: 'c' }); // Válido
```

Verificação Estrita de Objeto Literal (Freshness)

A verificação estrita de objeto literal, às vezes referida como "freshness", é um recurso do TypeScript que ajuda a capturar propriedades excessivas ou mal digitadas que, de outra forma, passariam despercebidas em verificações de tipo estrutural normais.

Ao criar um objeto literal, o compilador TypeScript o considera "fresh". Se o objeto literal for atribuído a uma variável ou passado como um parâmetro, o TypeScript lançará um erro se o objeto literal especificar propriedades que não existem no tipo de destino.

No entanto, a "freshness" desaparece quando um objeto literal é ampliado ou uma asserção de tipo é usada.

Aqui estão alguns exemplos para ilustrar:

```ts
type X = { a: string };
type Y = { a: string; b: string };

let x: X;
x = { a: 'a', b: 'b' }; // Verificação de freshness: Atribuição inválida
var y: Y;
y = { a: 'a', bx: 'bx' }; // Verificação de freshness: Atribuição inválida

const fn = (x: X) => console.log(x.a);
```

```
fn(x);
fn(y); // Ampliação: Sem erros, estruturalmente compatível com o
tipo

fn({ a: 'a', bx: 'b' }); // Verificação de freshness: Argumento
inválido

let c: X = { a: 'a' };
let d: Y = { a: 'a', b: '' };
c = d; // Ampliação: Sem verificação de Freshness
```

Inferência de Tipo

O TypeScript pode inferir tipos quando nenhuma anotação é fornecida durante:

- Inicialização de variável.
- Inicialização de membro.
- Configuração de padrões para parâmetros.
- Tipo de retorno de função.

Por exemplo:

```
let x = 'x'; // O tipo inferido é string
```

O compilador TypeScript analisa o valor ou expressão e determina seu tipo com base nas informações disponíveis.

Inferências Mais Avançadas

Quando múltiplas expressões são usadas na inferência de tipo, o TypeScript procura os “melhores tipos comuns”. Por exemplo:

```
let x = [1, 'x', 1, null]; // O tipo inferido é: (string | number |
null)[]
```

Se o compilador não conseguir encontrar os melhores tipos comuns, ele retorna um tipo union. Por exemplo:

```
let x = [new RegExp('x'), new Date()]; // Tipo inferido é: (RegExp
| Date)[]
```

O TypeScript utiliza “tipagem contextual” com base na localização da variável para inferir tipos. No seguinte exemplo, o compilador sabe que `e` é do tipo `MouseEvent` por causa do tipo de evento `click` definido no arquivo lib.d.ts, que contém declarações ambientes para várias construções JavaScript comuns e o DOM:

```
window.addEventListener('click', function (e) {}); // O tipo
inferido de e é MouseEvent
```

Ampliação de Tipo

Ampliação de tipo é o processo no qual o TypeScript atribui um tipo a uma variável inicializada quando nenhuma anotação de tipo foi fornecida. Permite tipos mais restritos para tipos mais amplos, mas não vice-versa. No seguinte exemplo:

```
let x = 'x'; // TypeScript infere como string, um tipo amplo
let y: 'y' | 'x' = 'y'; // y tipos é uma união de tipos literais
y = x; // Inválido Type 'string' is not assignable to type '"x" |
"y"'.
```

O TypeScript atribui `string` a `x` com base no valor único fornecido durante a inicialização (`x`), este é um exemplo de ampliação.

O TypeScript fornece maneiras de ter controle sobre o processo de ampliação, por exemplo, usando “const”.

Const

Usar a palavra-chave `const` ao declarar uma variável resulta em uma inferência de tipo mais restrita no TypeScript.

Por exemplo:

```
const x = 'x'; // TypeScript infere o tipo de x como 'x', um tipo
mais restrito
let y: 'y' | 'x' = 'y';
y = x; // Válido: O tipo de x é inferido como 'x'
```

Ao usar `const` para declarar a variável x, seu tipo é restringido ao valor literal específico ‘x’. Como o tipo de x é restringido, ele pode ser atribuído à variável y sem qualquer erro. A razão pela qual o tipo pode ser inferido é porque variáveis `const` não podem ser reatribuídas, então seu tipo pode ser restringido a um tipo literal específico, neste caso, o tipo literal ‘x’.

Modificador Const em Parâmetros de Tipo

A partir da versão 5.0 do TypeScript, é possível especificar o atributo `const` em um parâmetro de tipo genérico. Isso permite inferir o tipo mais preciso possível. Vamos ver um exemplo sem usar `const`:

```
function identity<T>(value: T) {
    // Sem const aqui
    return value;
}
const values = identity({ a: 'a', b: 'b' }); // Tipo inferido é: {
a: string; b: string; }
```

Como você pode ver, as propriedades `a` e `b` são inferidas com um tipo de `string`.

Agora, vamos ver a diferença com a versão `const`:

```
function identity<const T>(value: T) {
    // Usando modificador const em parâmetros de tipo
    return value;
}
const values = identity({ a: 'a', b: 'b' }); // Tipo inferido é: {
a: "a"; b: "b"; }
```

Agora podemos ver que as propriedades `a` e `b` são inferidas como `const`, então `a` e `b` são tratados como string literals em vez de apenas tipos `string`.

Asserção const

Este recurso permite que você declare uma variável com um tipo literal mais preciso com base em seu valor de inicialização, indicando ao compilador que o valor deve ser tratado como um literal imutável. Aqui estão alguns exemplos:

Em uma única propriedade:

```
const v = {
    x: 3 as const,
};
v.x = 3;
```

Em um objeto inteiro:

```
const v = {
    x: 1,
    y: 2,
} as const;
```

Isso pode ser particularmente útil ao definir o tipo para uma tuple:

```
const x = [1, 2, 3]; // number[]
const y = [1, 2, 3] as const; // Tuple de readonly [1, 2, 3]
```

Anotação de Tipo Explícita

Podemos ser específicos e passar um tipo, no seguinte exemplo a propriedade `x` é do tipo `number`:

```
const v = {
    x: 1, // Tipo inferido: number (ampliação)
};
v.x = 3; // Válido
```

Podemos tornar a anotação de tipo mais específica usando uma união de tipos literais:

```
const v: { x: 1 | 2 | 3 } = {
    x: 1, // x agora é uma união de tipos literais: 1 | 2 | 3
};
v.x = 3; // Válido
v.x = 100; // Inválido
```

Estreitamento de Tipo

Estreitamento de Tipo é o processo no TypeScript onde um tipo geral é estreitado para um tipo mais específico. Isso ocorre quando o TypeScript analisa o código e determina que certas condições ou operações podem refinar as informações do tipo.

O estreitamento de tipos pode ocorrer de diferentes maneiras, incluindo:

Condições

Ao usar instruções condicionais, como `if` ou `switch`, o TypeScript pode estreitar o tipo com base no resultado da condição. Por exemplo:

```
let x: number | undefined = 10;

if (x !== undefined) {
    x += 100; // O tipo é number, que havia sido estreitado pela condição
}
```

Lançar ou retornar

Lançar um erro ou retornar cedo de um branch pode ser usado para ajudar o TypeScript a estreitar um tipo. Por exemplo:

```
let x: number | undefined = 10;

if (x === undefined) {
    throw 'error';
}
x += 100;
```

Outras maneiras de estreitar tipos no TypeScript incluem:

- Operador `instanceof`: Usado para verificar se um objeto é uma instância de uma classe específica.
- Operador `in`: Usado para verificar se uma propriedade existe em um objeto.
- Operador `typeof`: Usado para verificar o tipo de um valor em tempo de execução.
- Funções integradas como `Array.isArray()`: Usadas para verificar se um valor é um array.

União Discriminada

Usar uma "União Discriminada" é um padrão no TypeScript onde uma "tag" explícita é adicionada aos objetos para distinguir entre diferentes tipos dentro de uma união. Este padrão também é referido como uma "união marcada". No seguinte exemplo, a "tag" é representada pela propriedade "type":

```
type A = { type: 'type_a'; value: number };
type B = { type: 'type_b'; value: string };

const x = (input: A | B): string | number => {
    switch (input.type) {
        case 'type_a':
            return input.value + 100; // type é A
        case 'type_b':
            return input.value + 'extra'; // type é B
    }
};
```

Type Guards Definidos pelo Usuário

Em casos onde o TypeScript não consegue determinar um tipo, é possível escrever uma função auxiliar conhecida como "type guard definido pelo usuário". No seguinte exemplo, utilizaremos um Type Predicate para estreitar o tipo após aplicar certa filtragem:

```
const data = ['a', null, 'c', 'd', null, 'f'];

const r1 = data.filter(x => x != null); // O tipo é (string | null)
[], TypeScript não conseguiu inferir o tipo corretamente

const isValid = (item: string | null): item is string => item !==
null; // Type guard customizado

const r2 = data.filter(isValid); // O tipo está correto agora
string[], ao usar o type guard predicado conseguimos estreitar o
tipo
```

Tipos Primitivos

O TypeScript suporta 7 tipos primitivos. Um tipo de dado primitivo refere-se a um tipo que não é um objeto e não tem métodos associados a ele. No TypeScript, todos os tipos primitivos são imutáveis, o que significa que seus valores não podem ser alterados uma vez atribuídos.

string

O tipo primitivo `string` armazena dados textuais, e o valor é sempre entre aspas duplas ou simples.

```
const x: string = 'x';
const y: string = 'y';
```

Strings podem abranger várias linhas se cercadas pelo caractere de crase (`):

```
let sentence: string = `xxx,
   yyy`;
```

boolean

O tipo de dado `boolean` no TypeScript armazena um valor binário, seja `true` ou `false`.

```
const isReady: boolean = true;
```

number

Um tipo de dado `number` no TypeScript é representado com um valor de ponto flutuante de 64 bits. Um tipo `number` pode representar inteiros e frações. O TypeScript também suporta hexadecimal, binário e octal, por exemplo:

```
const decimal: number = 10;
const hexadecimal: number = 0xa00d; // Hexadecimal começa com 0x
const binary: number = 0b1010; // Binário começa com 0b
const octal: number = 0o633; // Octal começa com 0o
```

bigInt

Um `bigInt` representa valores numéricos que são muito grandes (2^53 - 1) e não podem ser representados com um `number`.

Um `bigInt` pode ser criado chamando a função integrada `BigInt()` ou adicionando `n` ao final de qualquer literal numérico inteiro:

```
const x: bigint = BigInt(9007199254740991);
const y: bigint = 9007199254740991n;
```

Notas:

- Valores `bigInt` não podem ser misturados com `number` e não podem ser usados com `Math` integrado, eles devem ser coagidos para o mesmo tipo.
- Valores `bigInt` estão disponíveis apenas se a configuração de destino for ES2020 ou superior.

Symbol

Symbols são identificadores únicos que podem ser usados como chaves de propriedade em objetos para evitar conflitos de nomeação.

```
type Obj = {
    [sym: symbol]: number;
};

const a = Symbol('a');
const b = Symbol('b');
let obj: Obj = {};
obj[a] = 123;
obj[b] = 456;

console.log(obj[a]); // 123
console.log(obj[b]); // 456
```

null e undefined

Os tipos `null` e `undefined` ambos representam nenhum valor ou a ausência de qualquer valor.

O tipo `undefined` significa que o valor não é atribuído ou inicializado ou indica uma ausência não intencional de valor.

O tipo `null` significa que sabemos que o campo não tem um valor, portanto o valor está indisponível, indica uma ausência intencional de valor.

Array

Um `array` é um tipo de dado que pode armazenar múltiplos valores do mesmo tipo ou não. Pode ser definido usando a seguinte sintaxe:

```ts
const x: string[] = ['a', 'b'];
const y: Array<string> = ['a', 'b'];
const j: Array<string | number> = ['a', 1, 'b', 2]; // Union
```

O TypeScript suporta arrays readonly usando a seguinte sintaxe:

```ts
const x: readonly string[] = ['a', 'b']; // Modificador Readonly
const y: ReadonlyArray<string> = ['a', 'b'];
const j: ReadonlyArray<string | number> = ['a', 1, 'b', 2];
j.push('x'); // Inválido
```

O TypeScript suporta tuple e readonly tuple:

```ts
const x: [string, number] = ['a', 1];
const y: readonly [string, number] = ['a', 1];
```

any

O tipo de dado `any` representa literalmente “qualquer” valor, é o valor padrão quando o TypeScript não pode inferir o tipo ou não é especificado.

Ao usar `any`, o compilador TypeScript pula a verificação de tipo, então não há segurança de tipo quando `any` está sendo usado. Geralmente não use `any` para silenciar o compilador quando um erro ocorre, em vez disso foque em corrigir o erro, pois com o uso de `any` é possível quebrar contratos e perdemos os benefícios do autocomplete do TypeScript.

O tipo `any` pode ser útil durante uma migração gradual de JavaScript para TypeScript, pois pode silenciar o compilador.

Para novos projetos use a configuração TypeScript `noImplicitAny` que habilita o TypeScript a emitir erros onde `any` é usado ou inferido.

O tipo `any` é geralmente uma fonte de erros que pode mascarar problemas reais com seus tipos. Evite usá-lo o máximo possível.

Anotações de Tipo

Em variáveis declaradas usando `var`, `let` e `const`, é possível adicionar opcionalmente um tipo:

```
const x: number = 1;
```

O TypeScript executa uma análise estática automática das expressões e é geralmente capaz de inferir o tipo sem que este seja anotado. No exemplo anterior, o tipo poderia ser omitido:

```
const x = 1; // TypeScript infere o tipo number
```

Propriedades Opcionais

Um tipo de objeto pode ter zero ou mais propriedades opcionais adicionando um `?` após o nome da propriedade:

```
type X = {
    a: string;
    b?: string; // Opcional
};
```

Propriedades Readonly

É possível marcar uma propriedade como readonly para o TypeScript, isso não altera nenhum comportamento em tempo de

execução, mas uma propriedade marcada como readonly não pode ser escrita durante a verificação de tipo.

```
type X = {
    readonly a: string;
};

const x: X = { a: 'a' };
x.a = 'b'; // Inválido
```

Também é possível usar um “Mapping Modifier” para remover atributos readonly.

Assinaturas de Índice

Às vezes você não conhece antecipadamente os nomes das propriedades de um tipo, mas conhece a forma dos valores. Nesses casos, você pode usar uma assinatura de índice para descrever o tipo de valores possíveis. Uma assinatura de índice deve ser `string`, `number`, `symbol`, ou um template string pattern:

```
type X = {
    [key: string]: number;
};

const x: X = { a: 1, b: 2 };
```

É possível tornar uma assinatura de índice readonly adicionando a palavra-chave readonly:

```
type X = {
    readonly [key: string]: number;
};

const x: X = { a: 1, b: 2 };
x.a = 3; // Inválido
```

Estendendo Tipos

É possível estender uma interface (copiar membros de outros tipos nomeados) usando a palavra-chave `extends`:

```
interface X {
    a: string;
}

interface Y extends X {
    b: string;
}
```

Também é possível estender de múltiplos tipos:

```
interface A {
    a: string;
}

interface B {
    b: string;
}

interface Y extends A, B {
    y: string;
}
```

A palavra-chave `extends` funciona apenas em interfaces e classes, para tipos use uma intersecção:

```
type A = {
    a: number;
};

type B = {
    b: number;
};

type C = A & B;
```

Também é possível estender de uma interface com um tipo (usando intersecções):

```
interface A {
    a: string;
}

type B = A & {
    b: number;
};
```

Tipos Literais

Um tipo literal é um tipo que representa exatamente um único valor.

Por exemplo, uma variável pode aceitar apenas um único valor específico:

```
const x: 'a' = 'a'; // x pode ser apenas a literal 'a'
```

Combinando literais em uniões permite expressar conceitos como, por exemplo, uma função que aceita apenas um conjunto conhecido de valores:

```
const move = (direction: 'up' | 'down') => {
    // ...
};
```

Usar literais não-primitivos como number ou string não é permitido, pois o compilador os avaliaria para true ou false:

```
type X = 2 | 3;
type Y = 'a' | 'b';
```

Inferência Literal

Tipos literais são inferidos de variáveis declaradas com `var` ou `let`, que podem ser alteradas, em oposição a `const` que não pode:

```
const x = 'x'; // tem tipo 'x' (o valor não pode mudar)
let y = 'y'; // tem tipo string
```

strictNullChecks

Por padrão `null` e `undefined` são atribuíveis a todos os tipos, eles são ignorados pelo checker. É possível usar `--strictNullChecks` para impor que `null` e `undefined` sejam considerados ao fazer a verificação de tipo.

Ao usar a opção `strictNullChecks`, podem surgir erros que poderiam ser silenciados (sem a opção ativa). Ao usar o modo `strictNullChecks`, `null` e `undefined` têm seus próprios tipos chamados `null` e `undefined` respectivamente.

Em casos onde um valor pode ser de um tipo ou nulo/indefinido, você pode usar a união opcional:

```
type X = string | null | undefined;
```

Enums

Enums permitem que um desenvolvedor defina um conjunto de constantes nomeadas. Usar enums pode tornar mais fácil documentar a intenção ou criar um conjunto de casos distintos.

Enums numéricos

Por padrão, os enums são baseados em números, começando de zero e a cada membro é atribuído um incremento de um.

```
enum Direction {
    Up, // 0
    Down, // 1
    Left, // 2
    Right, // 3
}
```

Você pode atribuir o valor de qualquer membro manualmente, apenas o valor inicial neste caso:

```
enum Direction {
    Up = 1,
    Down, // 2
    Left, // 3
    Right, // 4
}
```

Ou todos os membros:

```
enum Direction {
    Up = 1,
    Down = 2,
    Left = 4,
    Right = 8,
}
```

Para acessar um enum, apenas acesse o membro como uma propriedade fora do enum:

```
const up = Direction.Up;
```

Também é possível acessar pelo valor:

```
const upName = Direction[1]; // Up
```

Enums de string

Enums de string são similares mas cada valor do enum é inicializado com um valor string (em vez de numérico):

```
enum Direction {
    Up = 'UP',
    Down = 'DOWN',
    Left = 'LEFT',
    Right = 'RIGHT',
}
```

Enums constantes

Um enum constante é definido usando o modificador `const` e pode melhorar o desempenho como os valores do enum são “inlined” durante a compilação:

```
const enum Direction {
    Up = 'UP',
    Down = 'DOWN',
    Left = 'LEFT',
    Right = 'RIGHT',
}
```

Mapeamento reverso

Podemos acessar o valor de um membro e também a um nome de membro do valor em si (então mapeamento reverso). Dado o seguinte enum numérico:

```
enum Direction {
    Up,
    Down,
}
```

Nós podemos fazer:

```
const a = Direction.Up; // 0
const b = Direction[0]; // Up
```

O TypeScript compila isso em:

```
'use strict';
var Direction;
(function (Direction) {
    Direction[(Direction['Up'] = 0)] = 'Up';
    Direction[(Direction['Down'] = 1)] = 'Down';
})(Direction || (Direction = {}));
```

Enums ambiente

Um enum ambiente é usado para descrever a forma de enums que já existem. Eles são definidos usando a palavra-chave `declare`:

```
declare enum Direction {
    Up,
    Down,
    Left,
    Right,
}
```

Membros computados e constantes

Cada membro enum tem um valor, que pode ser constante ou computado. Um membro é considerado constante se:

- Não tem um inicializador e o membro anterior era uma constante numérica.
- O membro tem um inicializador constante, uma expressão constante que é um subconjunto de TypeScript que pode ser completamente avaliado em tempo de compilação. Uma expressão é uma expressão constante se é:
    - Literal string ou numérico

- Referência a um membro enum constante anteriormente definido (pode estar em um enum diferente)
- Uma expressão constante enum entre parênteses
- Um dos operadores unários +, -, ~ aplicados a uma expressão constante
- Operadores binários +, -, *, /, %, <<, >>, >>>, &, |, ^ com expressões constantes como operandos

Narrowing

Narrowing é o processo de refinar tipos para um tipo mais específico. Por exemplo, você pode ter um tipo union `string | number` e deseja especificar se algo é uma string ou um number.

Type guards typeof

Verificar se um determinado valor é do tipo primitivo usando o operador typeof é uma forma de type guards, narrowing e proteção. O TypeScript reconhece o uso do operador `typeof` e pode estreitar em certas branches.

```
const fn = (x: string | number) => {
    if (typeof x === 'number') {
        return x + 1; // x é number
    }
    return x + 'b'; // x é string
};
```

Narrowing de veracidade

Veracidade pode estreitar em qualquer valor que pode ser coagido em um boolean, por exemplo, `if` statements, `&&`, `||`, instruções condicionais, `!` e mais.

```
const toUpperCase = (name: string | null) => {
    if (name) {
        return name.toUpperCase();
    } else {
        return null;
    }
};
```

Narrowing de igualdade

O TypeScript pode estreitar tipos comparando diretamente valores usando ===, !==, ==, e != para estreitar tipos.

```
const checkStatus = (status: 'success' | 'error') => {
    if (status === 'success') {
        // status é 'success'
    }
};
```

Narrowing do operador In

O operador `in` no JavaScript é um método para determinar se um objeto tem uma propriedade com um nome específico, o TypeScript pode usar para estreitar os tipos possíveis.

```
type Dog = { bark: () => void };
type Cat = { meow: () => void };

const talk = (pet: Dog | Cat) => {
    if ('bark' in pet) {
        pet.bark(); // pet é Dog
    } else {
        pet.meow(); // pet é Cat
    }
};
```

Narrowing instanceof

O operador instanceof em JavaScript verifica se o protótipo de um construtor aparece em qualquer lugar na cadeia de protótipos de um objeto. O TypeScript pode usar para estreitar tipos:

```
class Square {
    constructor(public width: number) {}
}
class Rectangle {
    constructor(
        public width: number,
        public height: number
    ) {}
}

function area(shape: Square | Rectangle) {
    if (shape instanceof Square) {
        return shape.width * shape.width; // shape é Square
    } else {
        return shape.width * shape.height; // shape é Rectangle
    }
}
```

Atribuições

O TypeScript examina o lado direito de uma atribuição e estreita o lado esquerdo apropriadamente ao reconhecer possíveis valores:

```
let x: string | number = 1;
x = 'a';
x = 1;
```

Análise de Fluxo de Controle

Análise de fluxo de controle em TypeScript é o processo de determinação do tipo de uma variável em diferentes pontos de um

programa baseado no fluxo de controle do código. Possibilita ao TypeScript entender como o tipo de uma variável muda quando diferentes branches de código são executadas.

Em TypeScript, a análise de fluxo de controle é realizada pelos “type guards”, que são funções ou expressões que realizam uma verificação de tempo de execução em um tipo e garantem esse tipo em um escopo específico. Type guards podem ser usados para estreitar o tipo de uma variável dentro de uma branch condicional, e o TypeScript usará essa informação para fornecer verificação de tipo mais precisa.

```
const f = (x: string | number) => {
    if (typeof x === 'string') {
        x.length; // x é string
    } else {
        x + 1; // x é number
    }
};
```

Predicados de Tipo

Um predicado de tipo é uma função cujo tipo de retorno é um predicado, ela pode ser usada para realizar análise de fluxo de controle do tipo. Um predicado de tipo é definido retornando um tipo especial chamado “type predicate”, que toma a forma `parameterName is Type`, onde “parameterName” deve ser o nome de um parâmetro da assinatura da função atual. Quando o predicado é avaliado com alguma variável, o TypeScript estreitará essa variável para o tipo específico, se o tipo original for compatível.

```
const isString = (value: unknown): value is string => typeof value
=== 'string';

const foo = (bar: unknown) => {
    if (isString(bar)) {
        console.log(bar.toUpperCase());
```

```
    } else {
        console.log('not a string');
    }
};
```

Também é possível usar predicados de tipo em `filter`:

```
const arr = [1, 2, 'a', 'b'];
const isString = (value: unknown): value is string => typeof value
=== 'string';
const strings = arr.filter(isString); // ['a', 'b']
```

Uniões Discriminadas

Uniões discriminadas em TypeScript são um tipo de tipo union onde cada tipo tem uma propriedade comum, chamada “discriminant”, que o TypeScript pode usar para estreitar o tipo da união.

Exemplo:

```
type Dog = { type: 'dog'; bark: () => void };
type Cat = { type: 'cat'; meow: () => void };

const makeSound = (pet: Dog | Cat) => {
    if (pet.type === 'dog') {
        // TypeScript sabe que pet é Dog aqui
        pet.bark();
    } else {
        // TypeScript sabe que pet é Cat aqui
        pet.meow();
    }
};
```

O Tipo never

O tipo `never` no TypeScript representa valores que nunca ocorrem. É usado para denotar valores que nunca são observados pelo

TypeScript, como quando o estreitamento de união remove todas as possibilidades.

O tipo `never` é frequentemente usado como um tipo de retorno para funções que nunca retornam ou sempre lançam uma exceção:

```
const throwError = (message: string): never => {
    throw new Error(message);
};
```

Verificação de exaustividade

Verificação de exaustividade é uma técnica no TypeScript para garantir que todos os casos possíveis foram tratados em um bloco de código. Ela é usada frequentemente em conjunto com uniões discriminadas e instruções switch.

Exemplo:

```
type Shape = Circle | Square | Triangle;

const getArea = (shape: Shape) => {
    switch (shape.kind) {
        case 'circle':
            return Math.PI * shape.radius ** 2;
        case 'square':
            return shape.sideLength ** 2;
        case 'triangle':
            return (shape.base * shape.height) / 2;
        default:
            // Se todos os casos forem tratados, shape terá tipo
never aqui
            const _exhaustiveCheck: never = shape;
            throw new Error('Unhandled shape');
    }
};
```

Tipos de Objeto

Tipos de objeto no TypeScript descrevem a forma de objetos. Eles especificam os nomes e tipos das propriedades que um objeto pode ter.

```
let car: { brand: string; model: string; year: number };

car = { brand: 'Ford', model: 'Focus', year: 2017 };
```

Tipo Tuple (Anônimo)

Tipos Tuple permitem criar arrays onde os tipos de um número fixo de elementos são conhecidos. Tuples são estruturas de dados que têm um comprimento fixo e podem conter elementos de tipos diferentes.

```
let tuple: [string, number, boolean];
tuple = ['hello', 42, true];
```

Tipo Tuple Nomeado (Rotulado)

Tuples nomeadas permitem que você atribua nomes aos elementos de uma tuple, tornando seu código auto-documentado:

```
let tuple: [name: string, age: number, active: boolean];
tuple = ['Alice', 30, true];
```

Tuple de Comprimento Fixo

Um tuple de comprimento fixo em TypeScript é um tipo de array que tem um comprimento exatamente definido.

```
let tuple: [number, number];
tuple = [1, 2];
```

Tipo Union

Tipos union no TypeScript permitem expressar um valor que pode ser de vários tipos. Um tipo union usa o símbolo de barra vertical (|) para separar cada tipo.

```
let value: string | number;

value = 'hello';
value = 42;
```

Tipos de Intersecção

Tipos de intersecção permitem combinar múltiplos tipos em um único tipo. Um tipo de intersecção representa um valor que tem todas as propriedades de todos os tipos envolvidos.

```
type A = { a: string };
type B = { b: number };
type C = A & B;

const obj: C = { a: 'hello', b: 42 };
```

Indexação de Tipo

Indexação de tipo em TypeScript permite acessar e extrair tipos de propriedades de outro tipo usando uma sintaxe semelhante a índice.

```
type Person = {
    name: string;
    age: number;
};
```

```ts
type Name = Person['name']; // string
```

Tipo a partir de Valor

Em TypeScript, `typeof` pode ser usado para capturar o tipo de um valor:

```ts
const config = { url: 'https://example.com', port: 8080 };
type Config = typeof config; // { url: string; port: number; }
```

Tipo a partir de Retorno de Função

Em TypeScript, você pode usar o tipo utilitário `ReturnType` para extrair o tipo de retorno de uma função:

```ts
const getValue = () => ({ value: 42 });
type Value = ReturnType<typeof getValue>; // { value: number }
```

Tipo a partir de Módulo

Em TypeScript, `import type` permite importar um tipo de um módulo:

```ts
// person.ts
export type Person = {
    name: string;
    age: number;
};

// app.ts
import type { Person } from './person';
```

Tipos Mapeados

Tipos mapeados no TypeScript permitem criar novos tipos baseados em tipos existentes transformando as propriedades. Eles são particularmente úteis quando você deseja criar um novo tipo alterando ou estendendo as propriedades de um tipo existente.

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};

type Person = {
    name: string;
    age: number;
};

type ReadonlyPerson = Readonly<Person>;
// { readonly name: string; readonly age: number; }
```

Modificadores de Tipo Mapeado

Modificadores de tipo mapeado no TypeScript permitem controlar a mutabilidade e opcionalidade das propriedades ao criar novos tipos baseados em tipos existentes. Existem dois modificadores: `readonly` e `?` (opcional).

- `readonly`: Torna as propriedades imutáveis.
- `?`: Torna as propriedades opcionais.
- `-readonly`: Remove o modificador readonly.
- `-?`: Remove o modificador opcional.

```
type Mutable<T> = {
    -readonly [P in keyof T]: T[P];
};

type Optional<T> = {
```

```
    [P in keyof T]?: T[P];
};
```

Tipos Condicionais

Tipos condicionais no TypeScript permitem expressar transformações de tipo não-uniformes. Eles fornecem uma maneira de fazer seleções de tipo baseadas em condições expressas como relações de teste de tipo.

```
type Check<T> = T extends string ? 'string' : 'other';

type A = Check<string>; // 'string'
type B = Check<number>; // 'other'
```

Tipos Condicionais Distributivos

Tipos condicionais distributivos em TypeScript distribuem operações de tipo sobre uniões. Quando um tipo condicional é aplicado a um tipo union, ele se aplica a cada membro da união separadamente.

```
type ToArray<T> = T extends any ? T[] : never;

type A = ToArray<string | number>; // string[] | number[]
```

Inferência de Tipo infer em Tipos Condicionais

A palavra-chave `infer` em tipos condicionais fornece uma maneira de inferir e capturar tipos dentro da cláusula condicional.

```
type ElementType<T> = T extends (infer U)[] ? U : never;

type A = ElementType<number[]>; // number
```

Tipos Condicionais Predefinidos

TypeScript fornece vários tipos condicionais predefinidos que são úteis para transformações de tipo comuns:

- `Exclude<T, U>`: Exclui de T tipos que são atribuíveis a U.
- `Extract<T, U>`: Extrai de T tipos que são atribuíveis a U.
- `NonNullable<T>`: Exclui null e undefined de T.
- `ReturnType<T>`: Obtém o tipo de retorno de uma função.
- `Parameters<T>`: Obtém os tipos de parâmetro de uma função.
- E mais...

```
type A = Exclude<string | number | boolean, boolean>; // string | number
type B = Extract<string | number | boolean, boolean>; // boolean
type C = NonNullable<string | null | undefined>; // string
```

Tipos Union de Template

Tipos Union de Template em TypeScript permitem criar novas uniões de string concatenando, transformando ou combinando tipos string literais existentes.

```
type Color = 'red' | 'blue';
type Size = 'small' | 'large';
type Style = `${Color}-${Size}`; // 'red-small' | 'red-large' | 'blue-small' | 'blue-large'
```

Tipo Any

O tipo `any` em TypeScript é o tipo mais permissivo que representa qualquer tipo de valor. Usar `any` essencialmente desativa a verificação de tipo para essa variável.

```
let value: any;
value = 'hello';
value = 42;
value = true;
```

Tipo Unknown

O tipo `unknown` é uma alternativa type-safe ao `any`. Enquanto `any` permite que você faça qualquer coisa com uma variável, `unknown` requer que você primeiro verifique ou afirme o tipo antes de usá-lo.

```
let value: unknown;

value = 'hello';
value = 42;

// Precisa estreitar o tipo antes de usar
if (typeof value === 'string') {
    console.log(value.toUpperCase());
}
```

Tipo Void

O tipo `void` representa a ausência de um tipo. É comumente usado como tipo de retorno para funções que não retornam um valor.

```
const logMessage = (message: string): void => {
    console.log(message);
};
```

Tipo Never

O tipo `never` representa valores que nunca ocorrem. É comumente usado para funções que nunca retornam ou sempre lançam um erro.

```typescript
const throwError = (message: string): never => {
    throw new Error(message);
};

const infiniteLoop = (): never => {
    while (true) {}
};
```

Interface e Type

No TypeScript, tanto `interface` quanto `type` podem ser usados para definir a forma de objetos e contratos de função. Embora compartilhem similaridades, existem diferenças em seus recursos e casos de uso.

Sintaxe Comum

```typescript
// Interface
interface Person {
    name: string;
    age: number;
}

// Type
type Person = {
    name: string;
    age: number;
};
```

Tipos Básicos

Ambos podem descrever tipos básicos:

```typescript
interface Point {
    x: number;
    y: number;
}
```

```
type Point = {
    x: number;
    y: number;
};
```

Objetos e Interfaces

Para objetos, tanto interfaces quanto types podem ser usados de forma intercambiável na maioria dos casos.

```
interface User {
    name: string;
    age: number;
}

type User = {
    name: string;
    age: number;
};
```

Tipos Union e Intersection

Types suportam uniões e intersecções, enquanto interfaces não:

```
type Status = 'success' | 'error';
type Response = SuccessResponse | ErrorResponse;
type Combined = TypeA & TypeB;
```

Primitivos de Tipo Integrados

TypeScript fornece vários primitivos de tipo integrados:

- `string`: Valores de texto
- `number`: Valores numéricos
- `boolean`: Valores true/false

- `null`: Representa ausência intencional de valor
- `undefined`: Representa ausência não intencional de valor
- `symbol`: Identificadores únicos
- `bigint`: Inteiros grandes

Objetos JS Integrados Comuns

TypeScript fornece tipos para objetos JavaScript integrados comuns:

- `Date`: Representa datas
- `Error`: Objetos de erro
- `Array<T>`: Arrays
- `Map<K, V>`: Maps
- `Set<T>`: Sets
- `Promise<T>`: Promises
- `RegExp`: Expressões regulares

Overloads

Sobrecarga de função permite múltiplas assinaturas de função para a mesma função:

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
    return a + b;
}
```

Mesclagem e Extensão

Interfaces podem ser mescladas (declaração merging) e podem estender outras interfaces:

```typescript
interface Person {
    name: string;
}

interface Person {
    age: number;
}

// Mescladas para: { name: string; age: number; }

interface Employee extends Person {
    employeeId: number;
}
```

Diferenças entre Type e Interface

Diferenças principais:

1. **Declaração merging**: Interfaces suportam, types não
2. **Uniões**: Types suportam uniões, interfaces não
3. **Tipos computados**: Types podem usar tipos computados
4. **Extending**: Interfaces usam `extends`, types usam `&`

```typescript
// Interface - declaração merging
interface User {
    name: string;
}
interface User {
    age: number;
}

// Type - uniões
type Status = 'success' | 'error';

// Type - tipos computados
type Keys = 'a' | 'b';
type Obj = { [K in Keys]: string };
```

Class

Classes em TypeScript fornecem uma forma de criar objetos com propriedades e métodos.

Sintaxe Comum de Class

```
class Person {
    name: string;
    age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log(`Hello, I'm ${this.name}`);
    }
}

const person = new Person('John', 30);
person.greet(); // Hello, I'm John
```

Constructor

O construtor é um método especial chamado quando uma instância da classe é criada:

```
class Car {
    brand: string;

    constructor(brand: string) {
        this.brand = brand;
    }
}
```

Construtores Private e Protected

Construtores podem ser marcados como private ou protected para controlar como classes são instanciadas:

```
class Singleton {
    private static instance: Singleton;

    private constructor() {}

    static getInstance() {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }
}
```

Modificadores de Acesso

TypeScript suporta modificadores de acesso: `public`, `private`, e `protected`:

- `public`: Acessível de qualquer lugar (padrão)
- `private`: Acessível apenas dentro da classe
- `protected`: Acessível dentro da classe e subclasses

```
class Person {
    public name: string;
    private age: number;
    protected address: string;

    constructor(name: string, age: number, address: string) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
}
```

Get e Set

TypeScript suporta getters e setters para controlar acesso às propriedades:

```
class Person {
    private _age: number = 0;

    get age(): number {
        return this._age;
    }

    set age(value: number) {
        if (value < 0) {
            throw new Error('Age cannot be negative');
        }
        this._age = value;
    }
}
```

Auto-Accessors em Classes

TypeScript 4.9 introduziu auto-accessors, que simplificam a criação de getters e setters:

```
class Person {
    accessor name: string;

    constructor(name: string) {
        this.name = name;
    }
}
```

this

O `this` em classes refere-se à instância da classe:

```
class Counter {
    count = 0;

    increment() {
        this.count++;
    }
}
```

Propriedades de Parâmetro

TypeScript permite declarar e inicializar propriedades de classe diretamente nos parâmetros do construtor:

```
class Person {
    constructor(
        public name: string,
        private age: number
    ) {}
}

// Equivalente a:
class Person {
    public name: string;
    private age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
}
```

Classes Abstratas

Classes abstratas são classes base das quais outras classes podem derivar. Elas não podem ser instanciadas diretamente:

```
abstract class Animal {
    abstract makeSound(): void;
```

```
    move(): void {
        console.log('Moving...');
    }
}

class Dog extends Animal {
    makeSound(): void {
        console.log('Woof!');
    }
}
```

Com Generics

Classes podem ser genéricas:

```
class Box<T> {
    private value: T;

    constructor(value: T) {
        this.value = value;
    }

    getValue(): T {
        return this.value;
    }
}

const stringBox = new Box<string>('hello');
const numberBox = new Box<number>(42);
```

Decorators

Decorators fornecem uma maneira de adicionar anotações e metaprogramação à sintaxe de classes:

Class Decorators

```
function sealed(constructor: Function) {
    Object.seal(constructor);
```

```
    Object.seal(constructor.prototype);
}

@sealed
class Example {
    property = 'test';
}
```

Property Decorator

```
function readonly(target: any, key: string) {
    Object.defineProperty(target, key, {
        writable: false,
    });
}

class Example {
    @readonly
    property = 'test';
}
```

Method Decorator

```
function log(target: any, key: string, descriptor:
PropertyDescriptor) {
    const original = descriptor.value;
    descriptor.value = function (...args: any[]) {
        console.log(`Calling ${key}`);
        return original.apply(this, args);
    };
}

class Example {
    @log
    method() {
        console.log('Method called');
    }
}
```

Decorators de Getter e Setter

```
function configurable(value: boolean) {
    return function (target: any, key: string, descriptor:
PropertyDescriptor) {
        descriptor.configurable = value;
    };
}

class Example {
    private _value = 0;

    @configurable(false)
    get value() {
        return this._value;
    }
}
```

Metadados de Decorator

Usando a biblioteca `reflect-metadata`, você pode adicionar e ler metadados:

```
import 'reflect-metadata';

function meta(key: string, value: any) {
    return Reflect.metadata(key, value);
}

class Example {
    @meta('role', 'admin')
    method() {}
}
```

Herança

Classes podem herdar de outras classes usando `extends`:

```
class Animal {
    name: string;
```

```
    constructor(name: string) {
        this.name = name;
    }

    move(distance: number = 0) {
        console.log(`${this.name} moved ${distance}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof!');
    }
}

const dog = new Dog('Buddy');
dog.bark(); // Woof!
dog.move(10); // Buddy moved 10m.
```

Statics

Membros estáticos pertencem à própria classe, não às instâncias:

```
class MathUtils {
    static PI = 3.14159;

    static areaCircle(radius: number) {
        return this.PI * radius ** 2;
    }
}

console.log(MathUtils.PI); // 3.14159
console.log(MathUtils.areaCircle(5));
```

Inicialização de propriedade

TypeScript permite inicializar propriedades diretamente na classe:

```
class Person {
    name: string = 'Unknown';
    age: number = 0;
}
```

Sobrecarga de método

Métodos podem ter múltiplas assinaturas:

```
class Calculator {
    add(a: number, b: number): number;
    add(a: string, b: string): string;
    add(a: any, b: any): any {
        return a + b;
    }
}
```

Generics

Generics fornecem uma maneira de criar componentes reutilizáveis que funcionam com vários tipos em vez de um único tipo.

Tipo Generic

```
function identity<T>(value: T): T {
    return value;
}

const numberValue = identity<number>(42);
const stringValue = identity<string>('hello');
```

Classes Generic

```
class Box<T> {
    private content: T;
```

```
    constructor(content: T) {
        this.content = content;
    }

    getContent(): T {
        return this.content;
    }
}

const stringBox = new Box<string>('hello');
const numberBox = new Box<number>(42);
```

Restrições Generic

Você pode restringir tipos genéricos usando extends:

```
interface HasLength {
    length: number;
}

function logLength<T extends HasLength>(value: T): void {
    console.log(value.length);
}

logLength('hello'); // 5
logLength([1, 2, 3]); // 3
```

Narrowing contextual generic

TypeScript pode estreitar tipos genéricos com base no contexto:

```
function process<T>(value: T): void {
    if (typeof value === 'string') {
        console.log(value.toUpperCase()); // value é string aqui
    } else if (typeof value === 'number') {
        console.log(value.toFixed(2)); // value é number aqui
    }
}
```

Tipos Estruturais Apagados

No TypeScript, objetos não precisam corresponder a um tipo específico e exato. Por exemplo, se criarmos um objeto que satisfaz os requisitos de uma interface, podemos utilizar esse objeto em lugares onde essa interface é necessária, mesmo que não houvesse conexão explícita entre eles.

```
interface Point {
    x: number;
    y: number;
}

function printPoint(point: Point) {
    console.log(`${point.x}, ${point.y}`);
}

const point = { x: 1, y: 2, z: 3 };
printPoint(point); // Válido, mesmo tendo z extra
```

Namespacing

Namespaces em TypeScript são usados para organizar código em contêineres lógicos, prevenindo colisões de nome e fornecendo uma maneira de agrupar código relacionado junto.

```
namespace Validation {
    export interface StringValidator {
        isValid(s: string): boolean;
    }

    export class LettersValidator implements StringValidator {
        isValid(s: string): boolean {
            return /^[A-Za-z]+$/.test(s);
        }
    }
}
```

```
const validator = new Validation.LettersValidator();
```

Symbols

Symbols são um tipo de dado primitivo que representa valores imutáveis que são garantidos como globalmente únicos durante o tempo de vida do programa.

```
const sym1 = Symbol('key');
const sym2 = Symbol('key');

console.log(sym1 === sym2); // false

const obj = {
    [sym1]: 'value1',
    [sym2]: 'value2',
};
```

Diretivas Triple-Slash

Diretivas triple-slash são comentários especiais que fornecem instruções ao compilador sobre como processar um arquivo. Elas começam com três barras consecutivas (`///`) e são tipicamente colocadas no topo de um arquivo TypeScript.

```
/// <reference path="./types.d.ts" />
/// <reference types="node" />
/// <reference lib="es2015" />
```

Manipulação de Tipo

Criando Tipos a partir de Tipos

TypeScript permite criar novos tipos a partir de tipos existentes usando várias transformações.

Tipos de Intersecção

```
type A = { a: string };
type B = { b: number };
type C = A & B; // { a: string; b: number; }
```

Tipos Union

```
type Status = 'success' | 'error';
type Response = SuccessResponse | ErrorResponse;
```

Tipos Mapeados

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};
```

Tipos Condicionais

```
type TypeName<T> = T extends string
    ? 'string'
    : T extends number
      ? 'number'
      : 'other';
```

Tipos de Acesso Indexado

Tipos de acesso indexado permitem acessar os tipos de propriedades:

```typescript
type Person = {
    name: string;
    age: number;
};

type Name = Person['name']; // string
type Keys = Person['name' | 'age']; // string | number
```

Tipos Utilitários

TypeScript fornece vários tipos utilitários integrados:

Awaited<T>

Desempacota recursivamente tipos Promise:

```typescript
type A = Awaited<Promise<string>>; // string
type B = Awaited<Promise<Promise<number>>>; // number
```

Partial<T>

Torna todas as propriedades opcionais:

```typescript
type Person = {
    name: string;
    age: number;
};

type PartialPerson = Partial<Person>;
// { name?: string; age?: number; }
```

Required<T>

Torna todas as propriedades obrigatórias:

```typescript
type Person = {
    name?: string;
    age?: number;
};
```

```ts
type RequiredPerson = Required<Person>;
// { name: string; age: number; }
```

Readonly<T>

Torna todas as propriedades readonly:

```ts
type Person = {
    name: string;
    age: number;
};

type ReadonlyPerson = Readonly<Person>;
// { readonly name: string; readonly age: number; }
```

Record<K, T>

Constrói um tipo com um conjunto de propriedades K do tipo T:

```ts
type Roles = 'admin' | 'user' | 'guest';
type Permissions = Record<Roles, boolean>;
// { admin: boolean; user: boolean; guest: boolean; }
```

Pick<T, K>

Constrói um tipo escolhendo propriedades específicas K de T:

```ts
type Person = {
    name: string;
    age: number;
    address: string;
};

type PersonName = Pick<Person, 'name' | 'age'>;
// { name: string; age: number; }
```

Omit<T, K>

Constrói um tipo omitindo propriedades específicas K de T:

```
type Person = {
    name: string;
    age: number;
    address: string;
};

type PersonWithoutAddress = Omit<Person, 'address'>;
// { name: string; age: number; }
```

Exclude<T, U>

Exclui de T tipos que são atribuíveis a U:

```
type A = Exclude<'a' | 'b' | 'c', 'a'>; // 'b' | 'c'
```

Extract<T, U>

Extrai de T tipos que são atribuíveis a U:

```
type A = Extract<'a' | 'b' | 'c', 'a' | 'f'>; // 'a'
```

NonNullable<T>

Exclui null e undefined de T:

```
type A = NonNullable<string | null | undefined>; // string
```

Parameters<T>

Extrai os tipos de parâmetro de uma função:

```
function greet(name: string, age: number) {
    console.log(`Hello ${name}, you are ${age}`);
}
```

```ts
type GreetParams = Parameters<typeof greet>;
// [name: string, age: number]
```

ConstructorParameters<T>

Extrai os tipos de parâmetro de um construtor:

```ts
class Person {
    constructor(
        public name: string,
        public age: number
    ) {}
}

type PersonParams = ConstructorParameters<typeof Person>;
// [name: string, age: number]
```

ReturnType<T>

Extrai o tipo de retorno de uma função:

```ts
function getValue() {
    return { value: 42 };
}

type Value = ReturnType<typeof getValue>;
// { value: number }
```

InstanceType<T>

Extrai o tipo de instância de uma classe:

```ts
class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
}
```

```
type PersonInstance = InstanceType<typeof Person>;
// Person
```

ThisParameterType<T>

Extrai o tipo do parâmetro 'this' de uma função:

```
function toHex(this: Number) {
    return this.toString(16);
}

type ThisType = ThisParameterType<typeof toHex>; // Number
```

OmitThisParameter<T>

Remove o parâmetro 'this' de uma função:

```
function toHex(this: Number) {
    return this.toString(16);
}

type NoThisType = OmitThisParameter<typeof toHex>; // () => string
```

ThisType<T>

Serve como um marcador para um tipo 'this' contextual:

```
type ObjectDescriptor<D, M> = {
    data?: D;
    methods?: M & ThisType<D & M>;
};
```

Uppercase<T>

Converte string literal types para maiúsculas:

```
type Greeting = 'hello';
type LoudGreeting = Uppercase<Greeting>; // 'HELLO'
```

Lowercase<T>

Converte string literal types para minúsculas:

```
type Greeting = 'HELLO';
type QuietGreeting = Lowercase<Greeting>; // 'hello'
```

Capitalize<T>

Capitaliza a primeira letra de string literal types:

```
type Greeting = 'hello';
type CapitalizedGreeting = Capitalize<Greeting>; // 'Hello'
```

Uncapitalize<T>

Descapitaliza a primeira letra de string literal types:

```
type Greeting = 'Hello';
type UncapitalizedGreeting = Uncapitalize<Greeting>; // 'hello'
```

NoInfer<T>

Bloqueia a inferência de tipos dentro do escopo de uma função genérica:

```
function createArray<T>(items: T[], item: NoInfer<T>): T[] {
    return [...items, item];
}

const arr = createArray(['a', 'b'], 'c'); // OK
// const arr = createArray(['a', 'b'], 42); // Erro
```

Outros

Erros e Tratamento de Exceções

TypeScript permite capturar e tratar erros usando mecanismos padrão de tratamento de erro JavaScript:

```
try {
    throw new Error('Something went wrong');
} catch (error) {
    if (error instanceof Error) {
        console.log(error.message);
    }
} finally {
    console.log('Cleanup');
}
```

Tipos de erro personalizados:

```
class CustomError extends Error {
    constructor(message: string) {
        super(message);
        this.name = 'CustomError';
    }
}
```

Classes mixin

Classes mixin permitem combinar e compor comportamento de múltiplas classes em uma única classe:

```
type Constructor<T = {}> = new (...args: any[]) => T;

function Timestamped<TBase extends Constructor>(Base: TBase) {
    return class extends Base {
        timestamp = Date.now();
    };
}
```

```
function Activatable<TBase extends Constructor>(Base: TBase) {
    return class extends Base {
        isActivated = false;
        activate() {
            this.isActivated = true;
        }
    };
}

class User {
    name = '';
}

const TimestampedUser = Timestamped(User);
const TimestampedActivatableUser = Timestamped(Activatable(User));
```

Recursos de Linguagem Assíncrona

TypeScript tem recursos assíncronos integrados do JavaScript:

Promises

```
const fetchData = (): Promise<string> => {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve('Data fetched'), 1000);
    });
};
```

Async/Await

```
async function getData(): Promise<string> {
    const data = await fetchData();
    return data;
}
```

Iteradores e Geradores

Iteradores

```
class NumberIterator implements Iterator<number> {
    private current: number;

    constructor(
        private start: number,
        private end: number
    ) {
        this.current = start;
    }

    next(): IteratorResult<number> {
        if (this.current <= this.end) {
            return { value: this.current++, done: false };
        }
        return { value: undefined, done: true };
    }
}
```

Geradores

```
function* numberGenerator(start: number, end: number):
Generator<number> {
    for (let i = start; i <= end; i++) {
        yield i;
    }
}

for (const num of numberGenerator(1, 5)) {
    console.log(num);
}
```

Referência JSDoc do TsDocs

TypeScript suporta anotações JSDoc para fornecer informações de tipo em código JavaScript:

```
/**
 * Adiciona dois números
 * @param {number} a - O primeiro número
 * @param {number} b - O segundo número
 * @returns {number} A soma de a e b
 */
function add(a, b) {
    return a + b;
}
```

@types

Pacotes sob a organização @types são convenções especiais de nomenclatura de pacotes usadas para fornecer definições de tipo para bibliotecas JavaScript existentes:

```
npm install --save-dev @types/node
npm install --save-dev @types/react
```

JSX

JSX é uma extensão de sintaxe para JavaScript que permite escrever código semelhante a HTML em seus arquivos TypeScript:

```
const element = <h1>Hello, world!</h1>;

type Props = {
    name: string;
};

const Greeting = ({ name }: Props) => <h1>Hello, {name}!</h1>;
```

Módulos ES6

TypeScript suporta módulos ES6:

```
// export
export const PI = 3.14;
export function circle(radius: number) {
    return 2 * PI * radius;
}

// import
import { PI, circle } from './math';

// default export
export default class Calculator {}

// default import
import Calculator from './Calculator';
```

Operador de Exponenciação ES7

```
const result = 2 ** 3; // 8
```

A Instrução for-await-of

Permite iterar sobre objetos iteráveis assíncronos:

```
async function* asyncGenerator() {
    yield Promise.resolve(1);
    yield Promise.resolve(2);
    yield Promise.resolve(3);
}

(async () => {
    for await (const num of asyncGenerator()) {
        console.log(num);
    }
})();
```

Meta-propriedade new target

Permite determinar se uma função ou construtor foi invocado usando o operador new:

```
class Parent {
    constructor() {
        console.log(new.target);
    }
}

class Child extends Parent {
    constructor() {
        super();
    }
}

const parent = new Parent(); // [Function: Parent]
const child = new Child(); // [Function: Child]
```

Expressões de Import Dinâmico

Permite carregar módulos condicionalmente ou sob demanda:

```
async function loadModule() {
    if (condition) {
        const module = await import('./module');
        module.doSomething();
    }
}
```

“tsc –watch”

Inicia o compilador TypeScript em modo watch:

```
tsc --watch
```

Operador de Asserção Não-nulo

O operador `!` afirma que um valor não é null ou undefined:

```
function getValue(value: string | null) {
    const len = value!.length; // Afirma que value não é null
}
```

Declarações com valor padrão

Parâmetros de função podem ter valores padrão:

```
function greet(name: string = 'Guest') {
    console.log(`Hello, ${name}`);
}

greet(); // Hello, Guest
greet('John'); // Hello, John
```

Encadeamento Opcional

O operador `?.` permite acessar propriedades que podem ser null ou undefined:

```
type Person = {
    name: string;
    address?: {
        street: string;
        city: string;
    };
};

const person: Person = { name: 'John' };
console.log(person.address?.city); // undefined
```

Operador de coalescência nula

O operador `??` retorna o valor do lado direito se o lado esquerdo for null ou undefined:

```
const value1 = null ?? 'default'; // 'default'
const value2 = 0 ?? 'default'; // 0
const value3 = '' ?? 'default'; // ''
```

Tipos Literais de Template

Tipos literais de template permitem criar novos tipos string manipulando tipos string existentes:

```
type World = 'world';
type Greeting = `hello ${World}`; // 'hello world'

type EmailLocaleIDs = 'welcome_email' | 'email_heading';
type FooterLocaleIDs = 'footer_title' | 'footer_sendoff';
type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
// 'welcome_email_id' | 'email_heading_id' | 'footer_title_id' |
'footer_sendoff_id'
```

Sobrecarga de função

Permite definir múltiplas assinaturas para a mesma função:

```
function parse(value: string): string[];
function parse(value: number): number;
function parse(value: string | number): string[] | number {
    if (typeof value === 'string') {
        return value.split(',');
    }
    return value;
}
```

Tipos Recursivos

Tipos que se referem a si mesmos:

```
type JSONValue =
    | string
    | number
    | boolean
    | null
    | JSONValue[]
    | { [key: string]: JSONValue };

const data: JSONValue = {
    name: 'John',
    age: 30,
    hobbies: ['reading', 'coding'],
    address: {
        city: 'New York',
        country: 'USA',
    },
};
```

Tipos Condicionais Recursivos

Tipos condicionais que se referem a si mesmos:

```
type Flatten<T> = T extends Array<infer U> ? Flatten<U> : T;

type Nested = [[[string]]];
type Flat = Flatten<Nested>; // string
```

Suporte a Módulo ECMAScript no Node

TypeScript suporta módulos ECMAScript no Node.js usando a extensão `.mts` ou configurando `"type": "module"` no package.json:

```
// math.mts
export const add = (a: number, b: number) => a + b;
```

```
// app.mts
import { add } from './math.mjs';
```

Funções de Asserção

Funções de asserção permitem expressar verificações invariantes que lançam um erro se a condição não for satisfeita:

```
function assert(condition: any, msg?: string): asserts condition {
    if (!condition) {
        throw new AssertionError(msg);
    }
}

function processValue(value: string | null) {
    assert(value !== null, 'Value cannot be null');
    console.log(value.toUpperCase()); // value é string aqui
}
```

Tipos Tuple Variádicos

Tuples que podem ter um número variável de elementos:

```
type Tuple<T extends any[]> = [string, ...T, number];

type T1 = Tuple<[boolean]>; // [string, boolean, number]
type T2 = Tuple<[boolean, string]>; // [string, boolean, string,
number]
```

Tipos boxed

Tipos primitivos têm contrapartes de objeto correspondentes (boxed types):

```
// Primitivos
const str: string = 'hello';
```

```typescript
const num: number = 42;
const bool: boolean = true;

// Boxed (geralmente não recomendado)
const strObj: String = new String('hello');
const numObj: Number = new Number(42);
const boolObj: Boolean = new Boolean(true);
```

Covariância e Contravariância no TypeScript

Covariância e contravariância descrevem como os relacionamentos de tipos funcionam com herança:

```typescript
// Covariância (arrays)
class Animal {
    name: string = '';
}
class Dog extends Animal {
    bark() {}
}

let animals: Animal[] = [];
let dogs: Dog[] = [];
animals = dogs; // Válido: Dog[] é atribuível a Animal[]

// Contravariância (funções)
type Logger<T> = (value: T) => void;

let logAnimal: Logger<Animal> = animal => console.log(animal.name);
let logDog: Logger<Dog> = logAnimal; // Válido
```

Anotações de Variância Opcionais para Parâmetros de Tipo

TypeScript 4.7+ permite anotações explícitas de variância:

```typescript
type Producer<out T> = () => T; // Covariante
type Consumer<in T> = (value: T) => void; // Contravariante
```

```
type Mapper<in T, out U> = (value: T) => U; // Ambos
```

Assinaturas de Índice de Padrão de String de Template

Permite usar padrões de template string em assinaturas de índice:

```
type DataProps = {
    [key: `data-${string}`]: string;
};

const props: DataProps = {
    'data-id': '123',
    'data-name': 'John',
    // 'id': '456' // Erro
};
```

O Operador satisfies

O operador `satisfies` permite verificar se um tipo satisfaz uma interface enquanto preserva o tipo mais específico:

```
type Color = 'red' | 'green' | 'blue';
type RGB = [number, number, number];

const color = { red: [255, 0, 0], green: '#00ff00' } satisfies
Record<
    Color,
    RGB | string
>;

const redNormalized = color.red[0]; // OK: [255, 0, 0] é inferido
como tuple
// const greenNormalized = color.green[0]; // Erro: string não tem
índice
```

Importações e Exportações Somente de Tipo

Permite importar e exportar apenas os tipos, não os valores:

```
// types.ts
export type User = {
    name: string;
    age: number;
};

// app.ts
import type { User } from './types';

export type { User };
```

Declaração using e Gerenciamento Explícito de Recursos

A declaração `using` permite gerenciar recursos que precisam ser descartados:

```
{
    using resource = getResource();
    // Usa o resource
} // resource.dispose() é chamado automaticamente
```

Declaração await using

Para recursos assíncronos:

```
{
    await using connection = await getConnection();
    // Usa a connection
} // await connection.dispose() é chamado automaticamente
```

Atributos de Import

Permite passar metadados adicionais para imports:

```
import data from './data.json' with { type: 'json' };
```

Atributos de Import do TypeScript 5.3 (rótulos para imports) dizem ao runtime como lidar com módulos (JSON, etc.). Isso melhora a segurança garantindo imports claros e alinha com a Content Security Policy (CSP) para carregamento de recursos mais seguro. O TypeScript garante que eles são válidos, mas deixa o runtime lidar com sua interpretação para tratamento específico de módulos.

Exemplo:

```
import config from './config.json' with { type: 'json' };
```

com import dinâmico:

```
const config = import('./config.json', { with: { type: 'json' } });
```