Refinamento

Imagine que temos uma função chamada `padLeft`.

```
function padLeft(padding: number | string, input: string): string {
  throw new Error("Não implementado ainda!");
}
```

Se `padding` for um `number`, ele será tratado como o número de espaços que queremos adicionar antes de `input`. Se `padding` for um `string`, ele deve apenas adicionar `padding` antes de `input`. Vamos tentar implementar a lógica para quando `padLeft` receber um `number` para `padding`.

```
function padLeft(padding: number | string, input: string): string {
  return " ".repeat(padding) + input;
Argument of type 'string | number' is not assignable to parameter of type 'number'.
  Type 'string' is not assignable to type 'number'.
}
```

Ops, estamos recebendo um erro no `padding`. O TypeScript está nos avisando que estamos passando um valor com o tipo `number | string` para a função `repeat`, que aceita apenas um `number`, e ele está certo. Em outras palavras, não verificamos explicitamente se `padding` é um `number` primeiro, nem estamos lidando com o caso onde é um `string`, então vamos fazer exatamente isso.

```
function padLeft(padding: number | string, input: string): string {
  if (typeof padding === "number") {
    return " ".repeat(padding) + input;
  }
  return padding + input;
}
```

Se isso parecer principalmente código JavaScript não muito interessante, esse é um pouco o objetivo. Além das anotações que colocamos, esse código TypeScript parece com JavaScript. A ideia é que o sistema de tipos do TypeScript visa tornar o mais fácil possível escrever código JavaScript típico sem precisar se esforçar para obter segurança de tipos.

Embora possa não parecer muito, há na verdade muito acontecendo nos bastidores aqui. Assim como o TypeScript analisa valores em tempo de execução usando tipos estáticos, ele sobrepõe a análise de tipos sobre os construtos de controle de fluxo em JavaScript, como `if/else`, ternários condicionais, loops, checagens de veracidade, etc., que podem afetar esses tipos.

Dentro da nossa checagem `if`, o TypeScript vê `typeof padding === "number"` e entende isso como uma forma especial de código chamada *guarda de tipo*. O TypeScript segue os possíveis caminhos de execução que nossos programas podem tomar para analisar o tipo mais específico de um valor em uma determinada posição. Ele examina essas checagens especiais (chamadas de *guardas de tipo*) e atribuições, e o processo de refinar tipos para tipos mais específicos do que os declarados é chamado de *refinamento*. Em muitos editores podemos observar esses tipos conforme eles mudam, e faremos isso em nossos exemplos.

```
function padLeft(padding: number | string, input: string): string {
  if (typeof padding === "number") {
    return " ".repeat(padding) + input;

(parameter) padding: number
  }
```

```
  return padding + input;

(parameter) padding: string
}
```

Existem alguns construtos diferentes que o TypeScript entende para refinamento.

Guardas de tipo `typeof`

Como vimos, o JavaScript suporta um operador `typeof` que pode fornecer informações básicas sobre o tipo dos valores que temos em tempo de execução. O TypeScript espera que isso retorne um conjunto específico de strings:

* `"string"`
* `"number"`
* `"bigint"`
* `"boolean"`
* `"symbol"`
* `"undefined"`
* `"object"`
* `"function"`

Como vimos com `padLeft`, esse operador aparece com bastante frequência em várias bibliotecas JavaScript, e o TypeScript pode entendê-lo para refinar tipos em diferentes ramificações.

No TypeScript, verificar contra o valor retornado por `typeof` é uma guarda de tipo. Como o TypeScript codifica como o `typeof` opera em diferentes valores, ele conhece algumas de suas peculiaridades em JavaScript. Por exemplo, note que na lista acima, `typeof` não retorna a string `null`. Veja o exemplo a seguir:

```
function printAll(strs: string | string[] | null) {
  if (typeof strs === "object") {
    for (const s of strs) {
'strs' é possivelmente 'null'.
      console.log(s);
    }
  } else if (typeof strs === "string") {
    console.log(strs);
  } else {
    // não faz nada
  }
}
```

Na função `printAll`, tentamos verificar se `strs` é um objeto para ver se é um tipo de array (agora pode ser um bom momento para reforçar que arrays são tipos de objetos em JavaScript). Mas acontece que, em JavaScript, `typeof null` é na verdade `"object"`! Esse é um dos acidentes infelizes da história.

Usuários com experiência suficiente podem não se surpreender, mas nem todos encontraram isso em JavaScript; felizmente, o TypeScript nos avisa que `strs` foi apenas refinado para `string[] | null` em vez de apenas `string[]`.

Isso pode ser uma boa introdução ao que chamaremos de checagem de "veracidade".

Refinamento por Veracidade

A palavra "veracidade" pode não ser encontrada no dicionário, mas é algo que você ouvirá frequentemente em JavaScript.

Em JavaScript, podemos usar qualquer expressão em condicionais, `&&`, `||`, instruções `if`, negações booleanas (`!`), e mais. Por exemplo, as instruções `if` não esperam que sua condição tenha sempre o tipo `boolean`.

```
function getUsersOnlineMessage(numUsersOnline: number) {
  if (numUsersOnline) {
    return `Há ${numUsersOnline} usuários online agora!`;
  }
  return "Não há ninguém aqui. :(";
}
```

Em JavaScript, construções como `if` primeiro "coagem" suas condições para `booleanos` para compreendê-las e depois escolhem seus ramos dependendo se o resultado é `true` ou `false`. Valores como

* `0`
* `NaN`
* `""` (a string vazia)
* `0n` (a versão `bigint` de zero)
* `null`
* `undefined`

todos coagem para `false`, e outros valores são coacionados para `true`. Você sempre pode coagir valores para `booleanos` passando-os pela função `Boolean`, ou usando a negação dupla mais curta. (A última tem a vantagem de que o TypeScript infere um tipo literal booleano `true`, enquanto a primeira é inferida como tipo `boolean`.)

```
// ambos resultam em 'true'
Boolean("hello"); // tipo: booleano, valor: true
!!"world"; // tipo: true,    valor: true
Este tipo de expressão é sempre verdadeiro.
```

É bastante popular aproveitar esse comportamento, especialmente para proteger contra valores como `null` ou `undefined`. Como exemplo, vamos tentar usá-lo na nossa função `printAll`.

```
function printAll(strs: string | string[] | null) {
  if (strs && typeof strs === "object") {
    for (const s of strs) {
      console.log(s);
    }
  } else if (typeof strs === "string") {
    console.log(strs);
  }
}
```

Você notará que eliminamos o erro acima verificando se `strs` é verdadeiro. Isso ao menos nos previne de erros indesejados quando executamos nosso código, como:

```
TypeError: null não é iterável
```

No entanto, tenha em mente que a verificação de veracidade em primitivos pode frequentemente ser propensa a erros. Como exemplo, considere uma tentativa diferente de escrever `printAll`.

```
function printAll(strs: string | string[] | null) {
  // !!!!!!!!!!!!!!!!
  // NÃO FAÇA ISSO!
  // CONTINUE LENDO
  // !!!!!!!!!!!!!!!!
  if (strs) {
    if (typeof strs === "object") {
      for (const s of strs) {
        console.log(s);
      }
    } else if (typeof strs === "string") {
      console.log(strs);
    }
  }
}
```

Nós envolvemos todo o corpo da função em uma verificação de veracidade, mas isso tem uma desvantagem sutil: pode ser que não estejamos mais lidando corretamente com o caso da string vazia.

O TypeScript não nos prejudica aqui, mas esse comportamento é importante de notar se você está menos familiarizado com JavaScript. O TypeScript pode frequentemente ajudar a detectar bugs cedo, mas se você optar por *não fazer nada* com um valor, há apenas até certo ponto o que ele pode fazer sem ser excessivamente prescritivo. Se desejar, você pode garantir que lida com situações como essas com um linter.

Uma última palavra sobre refinamento por veracidade é que as negações booleanas com `!` filtram os ramos negados.

```
function multiplyAll(
  values: number[] | undefined,
  factor: number
): number[] | undefined {
  if (!values) {
    return values;
  } else {
    return values.map((x) => x * factor);
  }
}
```

Refinamento por Igualdade

O TypeScript também usa instruções `switch` e verificações de igualdade como `===`, `!==`, `==`, e `!=` para refinar tipos. Por exemplo:

```
function example(x: string | number, y: string | boolean) {
  if (x === y) {
    // Agora podemos chamar qualquer método de 'string' em 'x' ou 'y'.
    x.toUpperCase();
```

```
(método) String.toUpperCase(): string
    y.toLowerCase();

(método) String.toLowerCase(): string
  } else {
    console.log(x);

(parâmetro) x: string | number
    console.log(y);

(parâmetro) y: string | boolean
  }
}
```

Quando verificamos que `x` e `y` são iguais no exemplo acima, o TypeScript sabia que seus tipos também tinham que ser iguais. Como `string` é o único tipo comum que tanto `x` quanto `y` poderiam assumir, o TypeScript sabe que `x` e `y` devem ser `string` no primeiro ramo.

Verificar contra valores literais específicos (em vez de variáveis) também funciona. Em nossa seção sobre refinamento por veracidade, escrevemos uma função `printAll` que era propensa a erros porque acidentalmente não lidava corretamente com strings vazias. Em vez disso, poderíamos ter feito uma verificação específica para bloquear `null`s, e o TypeScript ainda removeria corretamente `null` do tipo de `strs`.

```
function printAll(strs: string | string[] | null) {
  if (strs !== null) {
    if (typeof strs === "object") {
      for (const s of strs) {

(parâmetro) strs: string[]
        console.log(s);
      }
    } else if (typeof strs === "string") {
      console.log(strs);

(parâmetro) strs: string
    }
  }
}
```

Os checagens de igualdade mais frouxas do JavaScript com `==` e `!=` também são refinadas corretamente. Se você não estiver familiarizado, verificar se algo é `== null` na verdade não apenas verifica se é especificamente o valor `null` - também verifica se é potencialmente `undefined`. O mesmo se aplica a `== undefined`: verifica se um valor é `null` ou `undefined`.

```
interface Container {
  value: number | null | undefined;
}

function multiplyValue(container: Container, factor: number) {
  // Remove tanto 'null' quanto 'undefined' do tipo.
  if (container.value != null) {
    console.log(container.value);
```

```
(propriedade) Container.value: number

    // Agora podemos multiplicar com segurança 'container.value'.
    container.value *= factor;
  }
}
```

Refinamento com o operador `in`

O JavaScript tem um operador para determinar se um objeto ou sua cadeia de protótipos possui uma propriedade com um determinado nome: o operador `in`. O TypeScript leva isso em consideração como uma forma de refinar os tipos potenciais.

Por exemplo, com o código: `"value" in x`, onde `"value"` é um literal de string e `x` é um tipo de união. O ramo "true" refina os tipos de `x` que possuem uma propriedade `value`, opcional ou obrigatória, e o ramo "false" refina para tipos que possuem uma propriedade `value` opcional ou ausente.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    return animal.swim();
  }

  return animal.fly();
}
```

Para reiterar, propriedades opcionais existirão em ambos os lados do refinamento. Por exemplo, um humano poderia nadar e voar (com o equipamento certo) e, portanto, deveria aparecer em ambos os lados da verificação `in`:

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };
type Human = { swim?: () => void; fly?: () => void };

function move(animal: Fish | Bird | Human) {
  if ("swim" in animal) {
    animal;

(parâmetro) animal: Fish | Human
  } else {
    animal;

(parâmetro) animal: Bird | Human
  }
}
```

Refinamento com `instanceof`

O JavaScript tem um operador para verificar se um valor é uma "instância" de outro valor. Mais especificamente, em JavaScript, `x instanceof Foo` verifica se a *cadeia de protótipos* de `x` contém `Foo.prototype`. Embora não vamos aprofundar aqui, e você verá mais sobre isso quando abordarmos classes, ainda pode ser útil para a maioria dos valores que podem ser construídos com `new`. Como você

deve ter adivinhado, `instanceof` também é uma guarda de tipo, e o TypeScript faz o refinamento em ramos protegidos por `instanceof`.

```
function logValue(x: Date | string) {
  if (x instanceof Date) {
    console.log(x.toUTCString());

(parâmetro) x: Date
  } else {
    console.log(x.toUpperCase());

(parâmetro) x: string
  }
}
```

Atribuições

Como mencionamos anteriormente, quando atribuímos a qualquer variável, o TypeScript olha para o lado direito da atribuição e refina o lado esquerdo de forma apropriada.

```
let x = Math.random() < 0.5 ? 10 : "hello world!";

let x: string | number
x = 1;

console.log(x);

let x: number
x = "goodbye!";

console.log(x);

let x: string
```

Observe que cada uma dessas atribuições é válida. Mesmo que o tipo observado de `x` tenha mudado para `number` após nossa primeira atribuição, ainda pudemos atribuir um `string` a `x`. Isso ocorre porque o *tipo declarado* de `x` - o tipo com o qual `x` começou - é `string | number`, e a atribuibilidade é sempre verificada em relação ao tipo declarado.

Se tivéssemos atribuído um `boolean` a `x`, teríamos visto um erro, já que isso não fazia parte do tipo declarado.

```
let x = Math.random() < 0.5 ? 10 : "hello world!";

let x: string | number
x = 1;

console.log(x);

let x: number
x = true;
Type 'boolean' is not assignable to type 'string | number'.

console.log(x);

let x: string | number
```

Análise de Fluxo de Controle

Até aqui, vimos alguns exemplos básicos de como o TypeScript faz o refinamento dentro de ramos específicos. Mas há mais acontecendo do que apenas percorrer cada variável e procurar guardas de tipo em `if`s, `while`s, condicionais, etc. Por exemplo:

```
function padLeft(padding: number | string, input: string) {
  if (typeof padding === "number") {
    return " ".repeat(padding) + input;
  }
  return padding + input;
}
```

O `padLeft` retorna de dentro do seu primeiro bloco `if`. O TypeScript foi capaz de analisar este código e perceber que o restante do corpo (`return padding + input;`) é *inacessível* no caso em que `padding` é um `number`. Como resultado, ele foi capaz de remover `number` do tipo de `padding` (refinando de `string | number` para `string`) para o restante da função.

Essa análise do código com base na acessibilidade é chamada de *análise de fluxo de controle*, e o TypeScript usa essa análise de fluxo para refinar tipos à medida que encontra guardas de tipo e atribuições. Quando uma variável é analisada, o fluxo de controle pode se dividir e se reverter repetidamente, e essa variável pode ser observada com um tipo diferente em cada ponto.

```
function example() {
  let x: string | number | boolean;

  x = Math.random() < 0.5;

  console.log(x);


  if (Math.random() < 0.5) {
    x = "hello";
    console.log(x);

  } else {
    x = 100;
    console.log(x);

  }

  return x;

}
```

Usando Predicados de Tipo

Trabalhamos com construtos existentes do JavaScript para lidar com o refinamento até agora, no entanto, às vezes você deseja um controle mais direto sobre como os tipos mudam ao longo do seu código.

Para definir um guarda de tipo definido pelo usuário, precisamos apenas definir uma função cujo tipo de retorno seja um *predicado de tipo*:

```
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
```

`pet is Fish` é nosso predicado de tipo neste exemplo. Um predicado tem a forma `parameterName is Type`, onde `parameterName` deve ser o nome de um parâmetro da assinatura atual da função.

Sempre que `isFish` for chamado com alguma variável, o TypeScript irá *refinar* essa variável para o tipo específico se o tipo original for compatível.

```
// Ambas as chamadas para 'swim' e 'fly' agora estão ok.
let pet = getSmallPet();

if (isFish(pet)) {
  pet.swim();
} else {
  pet.fly();
}
```

Observe que o TypeScript não só sabe que `pet` é um `Fish` no ramo `if`; ele também sabe que no ramo `else`, você *não* tem um `Fish`, então você deve ter um `Bird`.

Você pode usar o guarda de tipo `isFish` para filtrar um array de `Fish | Bird` e obter um array de `Fish`:

```
const zoo: (Fish | Bird)[] = [getSmallPet(), getSmallPet(), getSmallPet()];
const underWater1: Fish[] = zoo.filter(isFish);
// ou, de forma equivalente
const underWater2: Fish[] = zoo.filter(isFish) as Fish[];

// O predicado pode precisar ser repetido para exemplos mais complexos
const underWater3: Fish[] = zoo.filter((pet): pet is Fish => {
  if (pet.name === "sharkey") return false;
  return isFish(pet);
});
```

Além disso, classes podem usar `this` como Tipo para refinar seu tipo.

Funções de Aserção

Os tipos também podem ser refinados usando Funções de Aserção.

Uniões Discriminadas

A maioria dos exemplos que vimos até agora focaram no refinamento de variáveis únicas com tipos simples, como `string`, `boolean` e `number`. Embora isso seja comum, na maioria das vezes em JavaScript lidamos com estruturas um pouco mais complexas.

Para um exemplo mais concreto, vamos imaginar que estamos tentando codificar formas como círculos e quadrados. Círculos mantêm o controle dos seus raios e quadrados mantêm o controle dos comprimentos de seus lados. Usaremos um campo chamado `kind` para indicar com qual forma estamos lidando. Aqui está uma primeira tentativa de definir `Shape`.

```
interface Shape {
  kind: "circle" | "square";
  radius?: number;
```

```
  sideLength?: number;
}
```

Observe que estamos usando uma união de tipos literais de string: `"circle"` e `"square"` para indicar se devemos tratar a forma como um círculo ou um quadrado, respectivamente. Ao usar `"circle" | "square"` em vez de `string`, podemos evitar problemas de digitação.

```
function handleShape(shape: Shape) {
  // ops!
  if (shape.kind === "rect") {
    // ...
  }
}
```

Podemos escrever uma função `getArea` que aplica a lógica correta com base em se estamos lidando com um círculo ou um quadrado. Primeiro, tentamos lidar com círculos.

```
function getArea(shape: Shape) {
  return Math.PI * shape.radius ** 2;
}
```

Sob strictNullChecks, isso gera um erro — o que é apropriado, já que `radius` pode não estar definido. Mas e se fizermos as verificações apropriadas na propriedade `kind`?

```
function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius ** 2;
  }
}
```

Hmm, o TypeScript ainda não sabe o que fazer aqui. Chegamos a um ponto onde sabemos mais sobre nossos valores do que o verificador de tipos. Poderíamos tentar usar uma asserção de não-nulo (um `!` após `shape.radius`) para dizer que `radius` está definitivamente presente.

```
function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius! ** 2;
  }
}
```

Mas isso não parece ideal. Tivemos que "gritar" um pouco para o verificador de tipos com essas asserções de não-nulo (`!`) para convencê-lo de que `shape.radius` estava definido, mas essas asserções são propensas a erros se começarmos a mover o código. Além disso, fora de strictNullChecks, podemos acidentalmente acessar qualquer um desses campos de qualquer forma (já que as propriedades opcionais são assumidas como sempre presentes ao lê-las). Podemos definitivamente fazer melhor.

O problema com essa codificação de `Shape` é que o verificador de tipos não tem uma maneira de saber se `radius` ou `sideLength` estão presentes com base na propriedade `kind`. Precisamos comunicar o que *sabemos* ao verificador de tipos. Com isso em mente, vamos tentar definir `Shape` de uma forma diferente.

```
interface Circle {
  kind: "circle";
  radius: number;
}
```

```
interface Square {
  kind: "square";
  sideLength: number;
}

type Shape = Circle | Square;
```

Aqui, separamos corretamente `Shape` em dois tipos com valores diferentes para a propriedade `kind`, mas `radius` e `sideLength` são declarados como propriedades obrigatórias em seus respectivos tipos.

Vamos ver o que acontece quando tentamos acessar o `radius` de um `Shape`.

```
function getArea(shape: Shape) {
  return Math.PI * shape.radius ** 2;
}
```

Assim como com nossa primeira definição de `Shape`, isso ainda gera um erro. Quando `radius` era opcional, tivemos um erro (com strictNullChecks ativado) porque o TypeScript não conseguia determinar se a propriedade estava presente. Agora que `Shape` é uma união, o TypeScript nos diz que `shape` pode ser um `Square`, e `Square`s não têm `radius` definido! Ambas as interpretações estão corretas, mas apenas a codificação da união de `Shape` causará um erro independentemente de como strictNullChecks está configurado.

Mas e se tentarmos verificar a propriedade `kind` novamente?

```
function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius ** 2;
  }
}
```

Isso eliminou o erro! Quando cada tipo em uma união contém uma propriedade comum com tipos literais, o TypeScript considera isso uma *união discriminada* e pode refinar os membros da união.

Neste caso, `kind` foi a propriedade comum (que é considerada uma propriedade *discriminante* de `Shape`). Verificar se a propriedade `kind` era `"circle"` eliminou todos os tipos em `Shape` que não tinham uma propriedade `kind` com o tipo `"circle"`. Isso refinou `shape` para o tipo `Circle`.

A mesma verificação funciona com instruções `switch` também. Agora podemos tentar escrever nosso `getArea` completo sem aquelas chatas asserções de não-nulo `!`.

```
function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
  }
}
```

A coisa importante aqui foi a codificação de `Shape`. Comunicar a informação certa para o TypeScript — que `Circle` e `Square` eram realmente dois tipos separados com campos `kind` específicos — foi crucial. Fazer isso nos permite escrever código TypeScript seguro em tipos que parece exatamente como o JavaScript que teríamos escrito de qualquer forma. A partir daí, o sistema de tipos foi capaz de fazer a coisa "certa" e descobrir os tipos em cada ramo da nossa instrução `switch`.

> Como uma observação, tente brincar com o exemplo acima e remover alguns dos `return`. Você verá que a verificação de tipos pode ajudar a evitar erros ao acidentalmente cair em cláusulas diferentes em uma instrução `switch`.

Uniões discriminadas são úteis para mais do que apenas falar sobre círculos e quadrados. Elas são boas para representar qualquer tipo de esquema de mensagens em JavaScript, como ao enviar mensagens pela rede (comunicação cliente/servidor) ou codificar mutações em um framework de gerenciamento de estado.

O Tipo `never`

Ao refinar, você pode reduzir as opções de uma união a um ponto onde você remove todas as possibilidades e não tem mais nada. Nesses casos, o TypeScript usará um tipo `never` para representar um estado que não deveria existir.

Verificação de Exhaustividade

O tipo `never` é atribuível a todos os tipos; no entanto, nenhum tipo é atribuível a `never` (exceto `never` em si). Isso significa que você pode usar refinamentos e confiar que `never` apareça para realizar uma verificação exaustiva em uma instrução `switch`.

Por exemplo, adicionar um `default` à nossa função `getArea`, que tenta atribuir a forma a `never`, não levantará um erro quando todos os casos possíveis foram tratados.

```
type Shape = Circle | Square;

function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
    default:
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}
```

Adicionar um novo membro à união `Shape` causará um erro do TypeScript:

```
interface Triangle {
  kind: "triangle";
  sideLength: number;
}

type Shape = Circle | Square | Triangle;

function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
    default:
      const _exhaustiveCheck: never = shape;
```

```
      // Erro: 'Triangle' não é atribuível ao tipo 'never'.
      return _exhaustiveCheck;
  }
}
```