

🔗 1 Branch


🏷 0 Tags

🔍 Go to file

Go **About**

Code

⋮

 **kourge** Bump 0.2.0 e0c6c07 · last year 🕒

| | | |
|----------------|---------------------|-------------|
| 📁 src | feat: added an ... | last year |
| 📁 test | Update test/ind... | last year |
| 📄 .gitignore | Set up ignore fi... | 8 years ago |
| 📄 .npmign... | Set up ignore fi... | 8 years ago |
| 📄 .prettierrc | Update .prettierrc | 7 years ago |
| 📄 CHANG... | Bump 0.2.0 | last year |
| 📄 LICENSE | Initial commit | 8 years ago |
| 📄 READM... | feat: added an ... | last year |
| 📄 package... | Bump all dev d... | last year |
| 📄 package... | Bump 0.2.0 | last year |
| 📄 tsconfig.... | Set up skeleton | 8 years ago |

Reusable type branding in TypeScript

- 📖 Readme
- 📄 MIT license
- 📈 Activity
- ★ 407 stars
- 👁 2 watching
- 🔗 10 forks

Report repository

Releases

No releases published

Packages

No packages published

Used by 3.6k



📖 README


📄 MIT license

⋮

ts-brand

With ts-brand , you can achieve [nominal typing](#) by leveraging a technique that is called "type branding" in the TypeScript community. Type branding works by intersecting a base type with a object type with a non-existent property. It is closely related in principal and usage to Flow's [opaque type aliases](#).

👤 Contributors 5



📄 Languages

TypeScript 100.0%

Installation

```
npm install --save ts-brand
```



Motivation and Example

Let's say we have the following API:

```
declare function getPost(postId: number): Promise<Post>;
declare function getUser(userId: number): Promise<User>;

interface User {
  id: number;
  name: string;
}

interface Post {
  id: number;
  authorId: number;
  title: string;
  body: string;
}
```



We want to leverage this API to write a function that, given a post's ID, can retrieve the user who wrote said post:

```
function authorOfPost(postId: number): Promise<User> {
  return getPost(postId).then(post => getUser(post.id));
}
```



Do you spot the bug? We're passing `post.id` to `getUser`, but we should have passed `post.authorId`:

```
function authorOfPost(postId: number): Promise<User> {
  return getPost(postId).then(post => getUser(post.authorId));
}
```



Nominal typing gives us a way to avoid conflating a user ID with a post ID, even though they are both numbers:

```
import {Brand} from 'ts-brand';

declare function getPost(postId: Post['id']): Promise<Post>;
declare function getUser(userId: User['id']): Promise<User>;
```



```
interface User {
  id: Brand<number, 'user'>;
  name: string;
}

interface Post {
  id: Brand<number, 'post'>;
  authorId: User['id'];
  title: string;
  body: string;
}
```

We have:

- Defined the ID types in terms of branded types with different branding types
- Substituted ad-hoc `number` types with a [lookup type](#), thus designating the interface as the centerpiece
- Retained the same runtime semantics as the original code
- Made our original buggy example fail to compile

There is one more risk left. If someone else were to define a different kind of `Post`, and also wrote `Brand<number, 'post'>`, it would still be possible to conflate the two accidentally. To solve this, we can take advantage of the fact that TypeScript interfaces can be recursive:

```
interface User {
  id: Brand<number, User>;
  name: string;
}

interface Post {
  id: Brand<number, Post>;
  authorId: User['id'];
  title: string;
  body: string;
}
```



By defining the ID type in terms of the interface surrounding it, we have made it such that the only way an ID can be treated like a `Post['id']` is if its branding type matches the structure of `Post` exactly.

API

This module exports four type members and two function members.

type Brand<Base, Branding, [ReservedName]>

A `Brand` is a type that takes at minimum two type parameters. Given a base type `Base` and some unique and arbitrary branding type `Branding`, it produces a type based on but distinct from `Base`. The resulting branded type is not directly assignable from the base type, and not mutually assignable with another branded type derived from the same base type.

Take care that the branding type is unique. Two branded types that share the same base type and branding type are considered the same type! There are two ways to avoid this.

The first way is to supply a third type parameter, `ReservedName`, with a string literal type that is not `__type__`, which is the default.

The second way is to define a branded type in terms of its surrounding interface, thereby forming a recursive type. This is possible because there are no constraints on what the branding type must be. It does not have to be a string literal type, even though it often is.

The third way is to define a branded type using an empty enum. In TypeScript, enums are nominally typed: two structurally identical enums are not considered the same type. This prevents two enum-branded types from ever being conflated for one another.

Examples:

```
type Path = Brand<string, 'path'>;
type UserId = Brand<number, 'user'>;
type DifferentUserId = Brand<number, 'user', '__kind__'>;
interface Post {
  id: Brand<number, Post>;
}
enum TimeTag {}
type Time = Brand<number, TimeTag>;
```



type AnyBrand

An `AnyBrand` is a branded type based on any base type branded with any branding type. By itself it is not useful, but it can act as type constraint when manipulating branded types in general.

type BaseOf<B extends AnyBrand>

`BaseOf` is a type that takes any branded type `B` and yields its base type.

type Brander<B extends AnyBrand>

A `Brander` is a function that takes a value of some base type and casts that value to a branded type derived from said base type. It can be thought of as the type of a "constructor", in the functional programming sense of the word.

Example:

```
type UserId = Brand<number, 'user'>;  
// A Brander<UserId> would take a number and return a UserId
```



function identity<B extends AnyBrand> (underlying: BaseOf): B

A generic function that, when given some branded type, can take a value with the base type of the branded type, and cast that value to the branded type. It fulfills the contract of a `Brander`.

At runtime, this function simply returns the value as-is.

Example:

```
type UserId = Brand<number, 'user'>;  
const UserId: Brander<UserId> = identity;
```



function make<B extends AnyBrand>(): Brander

Produces a `Brander`, given a brand type `B`. By default this returns `identity` but relies on type inference to give the return type the correct type.

Example:

```
type UserId = Brand<number, 'user'>;  
const UserId = make<UserId>();  
const myUserId = UserId(42);
```



Optionally, you may provide a validation function to assert that the value is the expected data shape. This may be done by passing a function to `make`:

```
type UserId = Brand<number, 'user'>;  
const UserId = make<UserId>((value) => {  
  if (value <= 0) {  
    throw new Error(`Non-positive value: ${value}`);  
  }  
});
```



```
UserId(42); // OK  
UserId(-1); // Error: Non-positive value: -1
```

Complete Example

We can form a cohesive, intuitive API definition by leveraging several features at once, such as namespace merging, type inference, and lookup types:

```
import {Brand, make} from 'ts-brand';
```



```
export interface User {  
  id: Brand<number, User>;  
  name: string;  
}
```

```
export namespace User {  
  export type Id = User['id'];  
  export const Id = make<Id>();  
}
```

```
export interface Post {  
  id: Brand<number, Post>;  
  authorId: User.Id;  
  title: string;  
  body: string;  
}
```

```
export namespace Post {  
  export type Id = Post['id'];  
  export const Id = make<Id>();  
}
```