一个值得注意的例外，以及此改变的解决方法是使用名称计算结果为 constructor 的计算属性。

```
class D {
  ["constructor"]() {
    console.log("我只是一个纯粹的方法，不是构造函数！")
  }
}
```

20.2 DOM 定义更新

lib.dom.d.ts 中移除或者修改了大量的定义。其中包括（但不仅限于）以下这些：

- 全局的 window 不再定义为 Window，它被更明确的定义 type Window & typeof globalThis 替代。在某些情况下，将它作为 typeof window 更好。
- GlobalFetch 已经被移除。使用 WindowOrWorkerGlobalScrope 替代。
- Navigator 上明确的非标准的属性已经被移除了。
- experimental-webgl 上下文已经被移除了。使用 webgl 或 webgl2 替代。

如果你认为其中的改变已经制造了错误，请提交一个 issue。

20.3 JSDoc 注释不再合并

在 JavaScript 文件中，TypeScript 只会在 JSDoc 注释之前立即查询以确定声明的类型。

```
/**
 * @param {string} arg
 */
/**
 * 你的其他注释信息
 */
function whoWritesFunctionsLikeThis(arg) {
  // 'arg' 是 'any' 类型
}
```

20.4 关键字不能包含转义字符

之前的版本允许关键字包含转义字符。TypeScript 3.6 不允许。

```
while (true) {
  \u0063ontinue;
//  ~~~~~~~~~~~~~
// 错误！关键字不能包含转义字符
}
```

参考：Announcing TypeScript 3.6

ts_breaking-changes

Catalog

在 TypeScript 3.4 中，现在可以在所有情况下正确探测使用 interface 声明的类型的变动。
这导致一个可见的重大变更，只要有类型参数的接口使用了 keyof（包括诸如 Record<K, T> 之类的地方，这是涉及 keyof K 的类型别名）。下例就是这样一个可能的变更。

```
interface HasX { x: any }
interface HasY { y: any }

declare const source: HasX | HasY;
declare const properties: KeyContainer<HasX>;

interface KeyContainer<T> {
  key: keyof T;
}

function readKey<T>(source: T, prop: KeyContainer<T>) {
  console.log(source[prop.key])
}

// 这个调用应该被拒绝，因为我们可能会这样做
// 错误地从 'HasY' 中读取 'x'。它现在恰当的提示错误。
readKey(source, properties);
```

此错误很可能表明原代码存在问题。
参考：原文

19 v3.5

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-3.5.html

19.1 lib.d.ts 包含了 Omit 辅助类型

TypeScript 3.5 包含一个 Omit 辅助类型。
因此，你项目中任何全局定义的 Omit 将产生以下错误信息：

```
Duplicate identifier 'Omit'.
```

两个变通的方法可以在这里使用：

1. 删除重复定义的并使用 lib.d.ts 提供的。
2. 从模块中导出定义避免全局冲突。现有的用法可以使用 import 直接引用项目的旧 Omit 类型。

20 v3.6

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-3.6.html

20.1 类成员的 constructor 现在被叫做 Constructors

根据 ECMAScript 规范，使用名为 constructor 的方法的类声明现在是构造函数，无论它们是使用标识符名称还是字符串名称声明。

```
class C {
  "constructor"() {
    console.log("现在我是构造函数了。");
  }
}
```

```
// 但是在之前的版本中类型为
//
//   {}[]
//
// 因此，插入一个 `string` 类型是错误的
x.value.push("hello");
```

x 上的显式类型注释可以清除这个错误。

18.2.1 上下文返回类型作为上下文参数类型传入

TypeScript 现在使用函数调用时传入的类型（如下例中的 then）作为函数上下文参数类型（如下例中的箭头函数）。

```
function isEven(prom: Promise<number>): Promise<{ success: boolean }>
{
  return prom.then<{success: boolean}>((x) => {
    return x % 2 === 0 ?
      { success: true } :
      Promise.resolve({ success: false });
    });
}
```

这通常是一种改进，但在上面的例子中，它导致 true 和 false 获取不合需要的字面量类型。

```
Argument of type '(x: number) => Promise<{ success: false; }> |
{ success: true; }' is not assignable to parameter of type '(value:
number) => { success: false; } | PromiseLike<{ success: false; }>'.
  Type 'Promise<{ success: false; }> | { success: true; }' is not
assignable to type '{ success: false; } | PromiseLike<{ success:
false; }>'.
    Type '{ success: true; }' is not assignable to type '{ success:
false; } | PromiseLike<{ success: false; }>'.
      Type '{ success: true; }' is not assignable to type '{ success:
false; }'.
        Types of property 'success' are incompatible.
```

合适的解决方法是将类型参数添加到适当的调用——本例中的 then 方法调用。

```
function isEven(prom: Promise<number>): Promise<{ success: boolean }>
{
  //             vvvvvvvvvvvvvvvvvv
  return prom.then<{success: boolean}>((x) => {
    return x % 2 === 0 ?
      { success: true } :
      Promise.resolve({ success: false });
  });
}
```

18.2.2 在 strictFunctionTypes 之外一致性推断优先

在 TypeScript 3.3 中，关闭 --strictFunctionTypes 选项时，假定使用 interface 声明的泛型类型在其类型参数方面始终是协变的。对于函数类型，通常无法观察到此行为。

但是，对于带有 keyof 状态的类型参数的泛型 interface 类型——逆变用法——这些类型表现不正确。

```
//     ~~~
// Cannot invoke an expression whose type lacks a call signature. Type
'(() => string) | (T & Function)' has no compatible call signatures.
    }
}
```

这是因为，不同于以前的 T 会被细化掉，如今 T 会被扩展成 T & Function。 然而，因为这个类型没有声明调用签名，类型系统无法找到通用的调用签名可以适用于 T & Function 和 () => string。

因此，考虑使用一个更确切的类型，而不是{}或 Object，并且考虑给 T 添加额外的约束条件。

17 v3.2

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-3.2.html

17.1 lib.d.ts 更新

17.1.1 wheelDelta 和它的小伙伴们被移除了。

wheelDeltaX、wheelDelta 和 wheelDeltaZ 全都被移除了，因为他们在 WheelEvents 上是废弃的属性。

解决办法：使用 deltaX、deltaY 和 deltaZ 代替。

17.1.2 更具体的类型

根据 DOM 规范的描述，某些参数现在接受更具体的类型，不再接受 null。

参考：原文

18 v3.4

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-3.4.html

18.1 顶级 this 现在有类型了

顶级 this 的类型现在被分配为 typeof globalThis 而不是 any。

因此，在 noImplicitAny 下访问 this 上的未知值，你可能收到错误提示。

```
// 在 `noImplicitAny` 下，以前可以，现在不行
this.whargarbl = 10;
```

请注意，在 noImplicitThis 下编译的代码不会在此处遇到任何更改。

18.2 泛型参数的传递

在某些情况下，TypeScript 3.4 的推断改进可能会产生泛型的函数，而不是那些接收并返回其约束的函数（通常是 {}）。

```
declare function compose<T, U, V>(f: (arg: T) => U, g: (arg: U) => V):
(arg: T) => V;

function list<T>(x: T) { return [x]; }
function box<T>(value: T) { return { value }; }

let f = compose(list, box);
let x = f(100)

// 在 TypeScript 3.4 中，'x.value' 的类型为
//
//   number[]
//
```

`MSAudioRecvSignal` * `MSAudioSendPayload` * `MSAudioSendSignal` * `MSConnectivity` * `MSCredentialFilter` * `MSCredentialParameters` * `MSCredentials` * `MSCredentialSpec` * `MSDCCEvent` * `MSDCCEventInit` * `MSDelay` * `MSDescription` * `MSDSHEvent` * `MSDSHEventInit` * `MSFIDOCredentialParameters` * `MSIceAddrType` * `MSIceType` * `MSIceWarningFlags` * `MSInboundPayload` * `MSIPAddressInfo` * `MSJitter` * `MSLocalClientEvent` * `MSLocalClientEventBase` * `MSNetwork` * `MSNetworkConnectivityInfo` * `MSNetworkInterfaceType` * `MSOutboundNetwork` * `MSOutboundPayload` * `MSPacketLoss` * `MSPayloadBase` * `MSPortRange` * `MSRelayAddress` * `MSSignatureParameters` * `MSStatsType` * `MSStreamReader` * `MSTransportDiagnosticsStats` * `MSUtilization` * `MSVideoPayload` * `MSVideoRecvPayload` * `MSVideoResolutionDistribution` * `MSVideoSendPayload` * `NotificationEventInit` * `PushEventInit` * `PushSubscriptionChangeInit` * `RTCIdentityAssertionResult` * `RTCIdentityProvider` * `RTCIdentityProviderDetails` * `RTCIdentityValidationResult` * `Screen.deviceXDPI` * `Screen.logicalXDPI` * `SVGElement.xmlbase` * `SVGGraphicsElement.farthestViewportElement` * `SVGGraphicsElement.getTransformToElement` * `SVGGraphicsElement.nearestViewportElement` * `SVGStylable` * `SVGTests.hasExtension` * `SVGTests.requiredFeatures` * `SyncEventInit` * `ValidateAssertionCallback` * `WebKitDirectoryEntry` * `WebKitDirectoryReader` * `WebKitEntriesCallback` * `WebKitEntry` * `WebKitErrorCallback` * `WebKitFileCallback` * `WebKitFileEntry` * `WebKitFileSystem` * `Window.clearImmediate` * `Window.msSetImmediate` * `Window.setImmediate`

推荐:

如果你的运行时能够保证这些名称是可用的（比如一个仅针对 IE 的应用），那么可以在本地添加那些声明，例如：

对于 Element.msMatchesSelector，在本地的 dom.ie.d.ts 文件里添加如下代码：

```
interface Element {
    msMatchesSelector(selectors: string): boolean;
}
```

相似地，若要添加 clearImmediate 和 setImmediate，你可以在本地的 dom.ie.d.ts 里添加 Window 声明：

```
interface Window {
    clearImmediate(handle: number): void;
    setImmediate(handler: (...args: any[]) => void): number;
    setImmediate(handler: any, ...args: any[]): number;
}
```

16.2 细化的函数现在会使用{}, Object 和未约束的泛型参数的交叉类型

下面的代码如今会提示 x 不能被调用：

```
function foo<T>(x: T | (() => string)) {
    if (typeof x === "function") {
        x();
```


1 v1.4

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-1.4.html

完整的破坏性改动列表请到这里查看:breaking change issues。

阅读 issue #868 以了解更多关于联合类型的破坏性改动。

1.1 多个最佳通用类型候选

当有多个最佳通用类型可用时，现在编译器会做出选择（依据编译器的具体实现）而不是直接使用第一个。

```
var a: { x: number; y?: number };
var b: { x: number; z?: number };

// 之前 { x: number; z?: number; }[]
// 现在 { x: number; y?: number; }[]
var bs = [b, a];
```

这会在多种情况下发生。具有一组共享的必需属性和一组其它互斥的（可选或其它）属性，空类型，兼容的签名类型（包括泛型和非泛型签名，当类型参数上应用了 any 时）。

推荐 使用类型注解指定你要使用的类型。

```
var bs: { x: number; y?: number; z?: number }[] = [b, a];
```

1.2 泛型接口

当在多个 T 类型的参数上使用了不同的类型时会得到一个错误，就算是添加约束也不行：

```
declare function foo<T>(x: T, y:T): T;
var r = foo(1, ""); // r used to be {}, now this is an error
```

添加约束：

```
interface Animal { x }
interface Giraffe extends Animal { y }
interface Elephant extends Animal { z }
function f<T extends Animal>(x: T, y: T): T { return undefined; }
var g: Giraffe;
var e: Elephant;
f(g, e);
```

在这里查看详细解释。

推荐 如果这种不匹配的行为是故意为之，那么明确指定类型参数：

```
var r = foo<{}>(1, ""); // Emulates 1.0 behavior
var r = foo<string|number>(1, ""); // Most useful
var r = foo<any>(1, ""); // Easiest
f<Animal>(g, e);
```

_或_重写函数定义指明就算不匹配也没问题：

```
declare function foo<T,U>(x: T, y:U): T|U;
function f<T extends Animal, U extends Animal>(x: T, y: U): T|U
{ return undefined; }
```

1.3 泛型剩余参数

不能再使用混杂的参数类型：

```
function makeArray<T>(...items: T[]): T[] { return items; }
var r = makeArray(1, ""); // used to return {}[], now an error
```

new Array(...)也一样

推荐 声明向后兼容的签名，如果 1.0 的行为是你想要的：

```
function makeArray<T>(...items: T[]): T[];
function makeArray(...items: {}[]): {}[];
function makeArray<T>(...items: T[]): T[] { return items; }
```

1.4 带类型参数接口的重载解析

```
var f10: <T>(x: T, b: () => (a: T) => void, y: T) => T;
var r9 = f10('', () => (a => a.foo), 1); // r9 was any, now this is an error
```

推荐 手动指定一个类型参数

```
var r9 = f10<any>('', () => (a => a.foo), 1);
```

1.5 类声明与类型表达式以严格模式解析

ECMAScript 2015 语言规范(ECMA-262 6th Edition)指明_ClassDeclaration_和_ClassExpression_使用严格模式。 因此，在解析类声明或类表达式时将使用额外的限制。例如:

```
class implements {}  // Invalid: implements is a reserved word in strict mode
class C {
    foo(arguments: any) {   // Invalid: "arguments" is not allow as a function argument
        var eval = 10;      // Invalid: "eval" is not allowed as the left-hand-side expression
        arguments = [];     // Invalid: arguments object is immutable
    }
}
```

关于严格模式限制的完整列表，请阅读 Annex C - The Strict Mode of ECMAScript of ECMA-262 6th Edition。

2 v1.5

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-1.5.html

完整的破坏性改动列表请到这里查看:breaking change issues。

2.1 不允许在箭头函数里引用 arguments

这是为了遵循 ES6 箭头函数的语义。之前箭头函数里的 arguments 会绑定到箭头函数的参数。参照 ES6 规范草稿 9.2.12，箭头函数不存在 arguments 对象。 从 TypeScript 1.5 开始，在箭头函数里使用 arguments 会被标记成错误以确保你的代码转成 ES6 时没语义上的错误。

例子:

```
function f() {
    return () => arguments; // Error: The 'arguments' object cannot be referenced in an arrow function.
}
```

推荐:

```
// 1. 使用带名字的剩余参数
function f() {
    return (...args) => { args; }
}

// 2. 使用函数表达式
```



```
* `CanvasRenderingContext2D.mozImageSmoothingEnabled` *
`CanvasRenderingContext2D.msFillRule` *
`CanvasRenderingContext2D.oImageSmoothingEnabled` *
`CanvasRenderingContext2D.webkitImageSmoothingEnabled` *
`Document.caretRangeFromPoint` * `Document.createExpression` *
`Document.createNSResolver` * `Document.execCommandShowHelp` *
`Document.exitFullscreen` * `Document.exitPointerLock` *
`Document.focus` * `Document.fullscreenElement` *
`Document.fullscreenEnabled` * `Document.getSelection` *
`Document.msCapsLockWarningOff` *
`Document.msCSSOMElementFloatMetrics` * `Document.msElementsFromRect`
* `Document.msElementsFromPoint` * `Document.onvisibilitychange` *
`Document.onwebkitfullscreenchange` *
`Document.onwebkitfullscreenerror` * `Document.pointerLockElement` *
`Document.queryCommandIndeterm` * `Document.URLUnencoded` *
`Document.webkitCurrentFullScreenElement` *
`Document.webkitFullscreenElement` *
`Document.webkitFullscreenEnabled` * `Document.webkitIsFullScreen` *
`Document.xmlEncoding` * `Document.xmlStandalone` *
`Document.xmlVersion` * `DocumentType.entities` *
`DocumentType.internalSubset` * `DocumentType.notations` *
`DOML2DeprecatedSizeProperty` * `Element.msContentZoomFactor` *
`Element.msGetUntransformedBounds` * `Element.msMatchesSelector` *
`Element.msRegionOverflow` * `Element.msReleasePointerCapture` *
`Element.msSetPointerCapture` * `Element.msZoomTo` *
`Element.onwebkitfullscreenchange` * `Element.onwebkitfullscreenerror`
* `Element.webkitRequestFullScreen` *
`Element.webkitRequestFullscreen` * `ElementCSSInlineStyle` *
`ExtendableEventInit` * `ExtendableMessageEventInit` *
`FetchEventInit` * `GenerateAssertionCallback` *
`HTMLAnchorElement.Methods` * `HTMLAnchorElement.mimeType` *
`HTMLAnchorElement.nameProp` * `HTMLAnchorElement.protocolLong` *
`HTMLAnchorElement.urn` * `HTMLAreasCollection` *
`HTMLHeadElement.profile` * `HTMLImageElement.msGetAsCastingSource` *
`HTMLImageElement.msGetAsCastingSource` *
`HTMLImageElement.msKeySystem` * `HTMLImageElement.msPlayToDisabled` *
`HTMLImageElement.msPlayToDisabled` *
`HTMLImageElement.msPlayToPreferredSourceUri` *
`HTMLImageElement.msPlayToPreferredSourceUri` *
`HTMLImageElement.msPlayToPrimary` *
`HTMLImageElement.msPlayToPrimary` * `HTMLImageElement.msPlayToSource`
* `HTMLImageElement.msPlayToSource` * `HTMLImageElement.x` *
`HTMLImageElement.y` * `HTMLInputElement.webkitdirectory` *
`HTMLLinkElement.import` * `HTMLMetaElement.charset` *
`HTMLMetaElement.url` * `HTMLSourceElement.msKeySystem` *
`HTMLStyleElement.disabled` * `HTMLSummaryElement` *
`MediaQueryListListener` * `MSAccountInfo` * `MSAudioLocalClientEvent`
* `MSAudioLocalClientEvent` * `MSAudioRecvPayload` *
```

- 除此之外，还可以使用 --keyofStringsOnly 编译器选项禁用新行为。

14.2 剩余参数后面不允许尾后逗号

以下代码是一个自 #22262 开始的编译器错误：

```
function f(
  a: number,
  ...b: number[], // 违规的尾随逗号
) {}
```

剩余参数上的尾随逗号不是有效的 JavaScript，并且，这个语法现在在 TypeScript 中也是一个错误。

14.3 在 strictNullChecks 中，无类型约束参数不再分配给 object

以下代码是自 24013 起在 strickNullChecks 下出现的编译器错误：

```
function f<T>(x: T) {
  const y: object | null | undefined = x;
}
```

它可以用任意类型（例如，string 或 number ）来实现，因此允许它是不正确的。 如果您遇到此问题，请将您的类型参数约束为 object 以仅允许对象类型。如果想允许任何类型，使用 {} 进行比较而不是 object。

参考：原文

15 v3.0

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-3.0.html

15.1 保留关键字 unknown

unknown 现在是一个保留类型名称，因为它现在是一个内置类型。为了支持新引入的 unknown 类型，取决于你对 unknown 的使用方式，你可能需要完全移除变量申明，或者将其重命名。

15.2 未开启 strictNullChecks 时，与 null/undefined 交叉的类型会简化到 null/undefined

关闭 strictNullChecks 时，下例中 A 的类型为 null，而 B 的类型为 undefined：

```
type A = { a: number } & null;      // null
type B = { a: number } & undefined; // undefined
```

这是因为 TypeScript 3.0 更适合分别简化交叉类型和联合类型中的子类型和超类型。但是，因为当 strictNullChecks 关闭时，null 和 undefined 都被认为是所有其他类型的子类型，与某种对象类型的交集将始终简化为 null 或 undefined。

建议

如果你在类型交叉的情况下依赖 null 和 undefined 作为单位元，你应该寻找一种方法来使用 unknown 而不是无论它们在哪里都是 null 或 undefined。

参考：原文

16 v3.1

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-3.1.html

16.1 一些浏览器厂商特定的类型从 lib.d.ts 中被移除

TypeScript 内置的.d.ts 库(lib.d.ts 等)现在会部分地从 DOM 规范的 Web IDL 文件中生成。 因此有一些浏览器厂商特定的类型被移除了。

点击这里查看被移除类型的完整列表：


```
function f() {
    return function(){ arguments; }
}
```

2.2 内联枚举引用的改动

对于正常的枚举，在 1.5 之前，编译器_仅会_内联常量成员，且成员仅在使用字面量初始化时才被当做是常量。这在判断检举值是使用字面量初始化还是表达式时会行为不一致。从 TypeScript 1.5 开始，所有非 const 枚举成员都不会被内联。

例子：

```
var x = E.a;  // previously inlined as "var x = 1; /*E.a*/"

enum E {
   a = 1
}
```

推荐： 在枚举声明里添加 const 修饰符来确保它总是被内联。 更多信息，查看#2183。

2.3 上下文的类型将作用于 super 和括号表达式

在 1.5 之前，上下文的类型不会作用于括号表达式内部。这就要求做显示的类型转换，尤其是在_必须_使用括号来进行表达式转换的场合。

在下面的例子里，m 具有上下文的类型，它在之前的版本里是没有的。

```
var x: SomeType = (n) => ((m) => q);
var y: SomeType = t ? (m => m.length) : undefined;

class C extends CBase<string> {
    constructor() {
        super({
            method(m) { return m.length; }
        });
    }
}
```

更多信息，查看#1425 和#920。

2.4 DOM 接口的改动

TypeScript 1.5 改进了 lib.d.ts 库里的 DOM 类型。这是自 TypeScript 1.0 以来第一次大的改动；为了拥抱标准 DOM 规范，很多特定于 IE 的定义被移除了，同时添加了新的类型如 Web Audio 和触摸事件。

变通方案：

你可以使用旧的 lib.d.ts 配合新版本的编译器。你需要在你的工程里引入之前版本的一个拷贝。这里是本次改动之前的 lib.d.ts 文件(TypeScript 1.5-alpha)。

变动列表：

- 属性 selection 从 Document 类型上移除
- 属性 clipboardData 从 Window 类型上移除
- 删除接口 MSEventAttachmentTarget
- 属性 onresize, disabled, uniqueID, removeNode, fireEvent, currentStyle, runtimeStyle 从 HTMLElement 类型上移除
- 属性 url 从 Event 类型上移除
- 属性 execScript, navigate, item 从 Window 类型上移除

- 属性 documentMode，parentWindow，createEventObject 从 Document 类型上移除
- 属性 parentWindow 从 HTMLDocument 类型上移除
- 属性 setCapture 被完全移除
- 属性 releaseCapture 被完全移除
- 属性 setAttribute，styleFloat，pixelLeft 从 CSSStyleDeclaration 类型上移除
- 属性 selectorText 从 CSSRule 类型上移除
- CSSStyleSheet.rules 现在是 CSSRuleList 类型，而非 MSCSSRuleList
- documentElement 现在是 Element 类型，而非 HTMLElement
- Event 具有一个新的必需属性 returnValue
- Node 具有一个新的必需属性 baseURI
- Element 具有一个新的必需属性 classList
- Location 具有一个新的必需属性 origin
- 属性 MSPOINTER_TYPE_MOUSE，MSPOINTER_TYPE_TOUCH 从 MSPointerEvent 类型上移除
- CSSStyleRule 具有一个新的必需属性 readonly
- 属性 execUnsafeLocalFunction 从 MSApp 类型上移除
- 全局方法 toStaticHTML 被移除
- HTMLCanvasElement.getContext 现在返回 CanvasRenderingContext2D | WebGLRenderingContex
- 移除扩展类型 Dataview，Weakmap，Map，Set
- XMLHttpRequest.send 具有两个重载 send(data?: Document): void;和 send(data?: String): void;
- window.orientation 现在是 string 类型，而非 number
- 特定于 IE 的 attachEvent 和 detachEvent 从 Window 上移除

以下是被新加的 DOM 类型所部分或全部取代的代码库的代表：

- DefinitelyTyped/auth0/auth0.d.ts
- DefinitelyTyped/gamepad/gamepad.d.ts
- DefinitelyTyped/interactjs/interact.d.ts
- DefinitelyTyped/webaudioapi/waa.d.ts
- DefinitelyTyped/webcrypto/WebCrypto.d.ts

更多信息，查看完整改动。

2.5 类代码体将以严格格式解析

按照 ES6 规范，类代码体现在以严格模式进行解析。行为将相当于在类作用域顶端定义了"use strict"；它包括限制了把 arguments 和 eval 做为变量名或参数名的使用，把未来保留字做为变量或参数使用，八进制数字字面量的使用等。

3 v1.6

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-1.6.html

完整的破坏性改动列表请到这里查看:breaking change issues。

13.2 从 lib.d.ts 中删除了一些 Microsoft 专用的类型

从 DOM 定义中删除一些 Microsoft 专用的类型以更好地与标准对齐。 删除的类型包括：

- MSApp
- MSAppAsyncOperation
- MSAppAsyncOperationEventMap
- MSBaseReader
- MSBaseReaderEventMap
- MSExecAtPriorityFunctionCallback
- MSHTMLWebViewElement
- MSManipulationEvent
- MSRangeCollection
- MSSiteModeEvent
- MSUnsafeFunctionCallback
- MSWebViewAsyncOperation
- MSWebViewAsyncOperationEventMap
- MSWebViewSettings

13.3 HTMLObjectElement 不再具有 alt 属性

根据 #21386，DOM 库已更新以反映 WHATWG 标准。
如果需要继续使用 alt 属性，请考虑通过全局范围中的接口合并重新打开 HTMLObjectElement:

```
// Must be in a global .ts file or a 'declare global' block.
interface HTMLObjectElement {
    alt: string;
}
```

14 v2.9

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.9.html

14.1 keyof 现在包括 string、number 和 symbol 键名

TypeScript 2.9 将索引类型泛化为包括 number 和 symbol 命名属性。以前，keyof 运算符和映射类型仅支持 string 命名属性。

```
function useKey<T, K extends keyof T>(o: T, k: K) {
  var name: string = k;  // 错误: keyof T 不能分配给 `string`
}
```

建议

- 如果你的函数只能处理名字符串属性的键，请在声明中使用 Extract<keyof T, string>:
- function useKey<T, K extends Extract<keyof T, string>>(o: T, k: K) {
- var name: string = k; // OK
- }
- 如果你的函数可以处理所有属性键，那么更改应该是顺畅的:
- function useKey<T, K extends keyof T>(o: T, k: K) {
- var name: string | number | symbol = k;
- }

```
function fails<K extends keyof O>(o: O, k: K) {
    var s: string = o[k]; // Previously allowed, now an error
                          // string | undefined is not assignable to a
string
}
```

12.4 in 表达式被视为类型保护

对于 n in x 表达式，其中 n 是字符串文字或字符串文字类型而 x 是联合类型，"true"分支缩小为具有可选或必需属性 n 的类型，并且 "false"分支缩小为具有可选或缺少属性 n 的类型。如果声明类型始终具有属性 n，则可能导致在 false 分支中将变量的类型缩小为 never 的情况。

```
var x: { foo: number };

if ("foo" in x) {
    x; // { foo: number }
}
else {
    x; // never
}
```

12.5 在条件运算符中不减少结构上相同的类

以前在结构上相同的类在条件或||运算符中被简化为最佳公共类型。现在这些类以联合类型维护，以便更准确地检查 instanceof 运算符。

```
class Animal {

}

class Dog {
    park() { }
}

var a = Math.random() ? new Animal() : new Dog();
// typeof a now Animal | Dog, previously Animal
```

12.6 CustomEvent 现在是一个泛型类型

CustomEvent 现在有一个 details 属性类型的类型参数。如果要从中扩展，则需要指定其他类型参数。

```
class MyCustomEvent extends CustomEvent {
}
```

应该成为

```
class MyCustomEvent extends CustomEvent<any> {
}
```

13 v2.8

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.8.html

13.1 在--noUnusedParameters 下检查未使用的类型参数

根据 #20568，未使用的类型参数之前在--noUnusedLocals 下报告，但现在报告在--noUnusedParameters 下。

3.1 严格的对象字面量赋值检查

当在给变量赋值或给非空类型的参数赋值时，如果对象字面量里指定的某属性不存在于目标类型中时会得到一个错误。

你可以通过使用--suppressExcessPropertyErrors 编译器选项来禁用这个新的严格检查。

例子：

```
var x: { foo: number };
x = { foo: 1, baz: 2 };  // Error, excess property `baz`

var y: { foo: number, bar?: number };
y = { foo: 1, baz: 2 };  // Error, excess or misspelled property `baz`
```

推荐：

为了避免此错误，不同情况下有不同的补救方法：

如果目标类型接收额外的属性，可以增加一个索引：

```
var x: { foo: number, [x: string]: any };
x = { foo: 1, baz: 2 };  // OK, `baz` matched by index signature
```

如果原始类型是一组相关联的类型，使用联合类型明确指定它们的类型而不是仅指定一个基本类型。

```
let animalList: (Dog | Cat | Turkey)[] = [    // use union type instead
of Animal
    {name: "Milo", meow: true },
    {name: "Pepper", bark: true},
    {name: "koko", gobble: true}
];
```

还有可以明确地转换到目标类型以避免此错误：

```
interface Foo {
    foo: number;
}
interface FooBar {
    foo: number;
    bar: number;
}
var y: Foo;
y = <FooBar>{ foo: 1, bar: 2 };
```

3.2 CommonJS 的模块解析不再假设路径为相对的

之前，对于 one.ts 和 two.ts 文件，如果它们在相同目录里，那么在 two.ts 里面导入"one"时是相对于 one.ts 的路径的。

TypeScript 1.6 在编译 CommonJS 时，"one"不再等同于"./one"。取而代之的是会相对于合适的 node_modules 文件夹进行查找，与 Node.js 在运行时解析模块相似。更多详情，阅读 the issue that describes the resolution algorithm。

例子：

./one.ts

```
export function f() {
    return 10;
}
```

./two.ts

```
import { f as g } from "one";
```

推荐：

修改所有计划之外的非相对的导入。

./one.ts

```
export function f() {
    return 10;
}
```

./two.ts

```
import { f as g } from "./one";
```

将--moduleResolution 编译器选项设置为 classic。

3.3 函数和类声明为默认导出时不再能够与在意义上有交叉的同名实体进行合并

在同一空间内默认导出声明的名字与空间内一实体名相同时会得到一个错误；比如，

```
export default function foo() {
}

namespace foo {
    var x = 100;
}
```

和

```
export default class Foo {
    a: number;
}

interface Foo {
    b: string;
}
```

两者都会报错。

然而，在下面的例子里合并是被允许的，因为命名空间并不具备做为值的意义：

```
export default class Foo {
}

namespace Foo {
}
```

推荐：

为默认导出声明本地变量并使用单独的 export default 语句：

```
class Foo {
    a: number;
}

interface foo {
    b: string;
}

export default Foo;
```

更多详情，请阅读 the originating issue。

12 v2.7

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.7.html

完整的破坏性改动列表请到这里查看:breaking change issues.

12.1 元组现在具有固定长度的属性

以下代码用于没有编译错误:

```
var pair: [number, number] = [1, 2];
var triple: [number, number, number] = [1, 2, 3];
pair = triple;
```

但是，这_是_一个错误:

```
triple = pair;
```

现在，相互赋值是一个错误。 这是因为元组现在有一个长度属性，其类型是它们的长度。 所以 pair.length: 2，但是 triple.length: 3。

请注意，之前允许某些非元组模式，但现在不再允许:

```
const struct: [string, number] = ['key'];
for (const n of numbers) {
  struct.push(n);
}
```

对此最好的解决方法是创建扩展 Array 的自己的类型:

```
interface Struct extends Array<string | number> {
  '0': string;
  '1'?: number;
}
const struct: Struct = ['key'];
for (const n of numbers) {
  struct.push(n);
}
```

12.2 在 allowSyntheticDefaultImports 下，对于 TS 和 JS 文件来说默认导入的类型合成不常见

在过去，我们在类型系统中合成一个默认导入，用于 TS 或 JS 文件，如下所示:

```
export const foo = 12;
```

意味着模块的类型为{foo: number, default: {foo: number}}。 这是错误的，因为文件将使用__esModule 标记发出，因此在加载文件时没有流行的模块加载器会为它创建合成默认值，并且类型系统推断的 default 成员永远不会在运行时存在。现在我们在 ESModuleInterop 标志下的发出中模拟了这个合成默认行为，我们收紧了类型检查器的行为，以匹配你期望在运行时所看到的内容。如果运行时没有其他工具的介入，此更改应仅指出错误的错误默认导入用法，应将其更改为命名空间导入。

12.3 更严格地检查索引访问泛型类型约束

以前，仅当类型具有索引签名时才计算索引访问类型的约束，否则它是 any。这样就可以取消选中无效赋值。在 TS 2.7.1 中，编译器在这里有点聪明，并且会将约束计算为此处所有可能属性的并集。

```
interface O {
    foo?: string;
}
```

10.5 从上下文类型中推荐类型参数

在 TypeScript 之前，下面例子中

```
let f: <T>(x: T) => T = y => y;
```

y 的类型将是 any。 这意味着，程序虽会进行类型检查，但是你可以在 y 上做任何事，比如：

```
let f: <T>(x: T) => T = y => y() + y.foo.bar;
```

推荐做法：

适当地重新审视你的泛型是否为正确的约束。实在不行，就为参数加上 any 注解。

11 v2.6

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.6.html

完整的破坏性改动列表请到这里查看:breaking change issues.

11.1 只写引用未使用

以下代码用于没有编译错误：

```
function f(n: number) {
    n = 0;
}

class C {
    private m: number;
    constructor() {
        this.m = 0;
    }
}
```

现在，当启用--noUnusedLocals 和--noUnusedParameters 编译器选项时，n 和 m 都将被标记为未使用，因为它们的值永远不会被_读_ 。以前 TypeScript 只会检查它们的值是否被_引用_。

此外，仅在其自己的实体中调用的递归函数被视为未使用。

```
function f() {
    f(); // Error: 'f' is declared but its value is never read
}
```

11.2 环境上下文中的导出赋值中禁止使用任意表达式

以前，像这样的结构

```
declare module "foo" {
    export default "some" + "string";
}
```

在环境上下文中未被标记为错误。声明文件和环境模块中通常禁止使用表达式，因为 typeof 之类的意图不明确，因此这与我们在这些上下文中的其他地方处理可执行代码不一致。现在，任何不是标识符或限定名称的内容都会被标记为错误。为具有上述值形状的模块制作 DTS 的正确方法如下：

```
declare module "foo" {
    const _default: string;
    export default _default;
}
```

编译器已经生成了这样的定义，因此这只应该是手工编写的定义的问题。

3.4 模块体以严格模式解析

按照 ES6 规范，模块体现在以严格模式进行解析。行为将相当于在模块作用域顶端定义了"use strict"；它包括限制了把 arguments 和 eval 做为变量名或参数名的使用，把未来保留字做为变量或参数使用，八进制数字字面量的使用等。

3.5 标准库里 DOM API 的改动

- **MessageEvent** 和 **ProgressEvent** 构造函数希望传入参数；查看 issue #4295。
- **ImageData** 构造函数希望传入参数；查看 issue #4220。
- **File** 构造函数希望传入参数；查看 issue #3999。

3.6 系统模块输出使用批量导出

编译器以系统模块的格式使用新的_export 函数批量导出的变体，它接收任何包含键值对的对象做为参数而不是 key，value。

模块加载器需要升级到 v0.17.1 或更高。

3.7 npm 包的.js 内容从'bin'移到了'lib'

TypeScript 的 npm 包入口位置从 bin 移动到了 lib，以防‘node_modules/typescript/bin/typescript.js’通过 IIS 访问的时候造成阻塞（bin 默认是隐藏段因此 IIS 会阻止访问这个文件夹）。

3.8 TypeScript 的 npm 包不会默认全局安装

TypeScript 1.6 从 package.json 里移除了 preferGlobal 标记。如果你依赖于这种行为，请使用 npm install -g typescript。

3.9 装饰器做为调用表达式进行检查

从 1.6 开始，装饰器类型检查更准确了；编译器会将装饰器表达式做为以被装饰的实体做为参数的调用表达式来进行检查。这可能会造成以前的代码报错。

4 v1.7

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-1.7.html

完整的破坏性改动列表请到这里查看:breaking change issues。

4.1 从 this 中推断类型发生了变化

在类里，this 值的类型将被推断成 this 类型。 这意味着随后使用原始类型赋值时可能会发生错误。

例子：

```
class Fighter {
    /** @returns the winner of the fight. */
    fight(opponent: Fighter) {
        let theVeryBest = this;
        if (Math.rand() < 0.5) {
            theVeryBest = opponent; // error
        }
        return theVeryBest
    }
}
```

推荐：

添加类型注解：

```
class Fighter {
    /** @returns the winner of the fight. */
```

```
    fight(opponent: Fighter) {
        let theVeryBest: Fighter = this;
        if (Math.rand() < 0.5) {
            theVeryBest = opponent; // no error
        }
        return theVeryBest
    }
}
```

4.2 类成员修饰符后面会自动插入分号

关键字 abstract，public，protected 和 private 是 ECMAScript 3 里的_保留关键字_并适用于自动插入分号机制。 之前，在这些关键字出现的行尾，TypeScript 是不会插入分号的。 现在，这已经被改正了，在上例中 abstract class D 不再能够正确地继承 C 了，而是声明了一个 m 方法和一个额外的属性 abstract。
注意，async 和 declare 已经能够正确自动插入分号了。
例子：

```
abstract class C {
    abstract m(): number;
}
abstract class D extends C {
    abstract
    m(): number;
}
```

推荐：
在定义类成员时删除关键字后面的换行。通常来讲，要避免依赖于自动插入分号机制。

5 v1.8

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-1.8.html
完整的破坏性改动列表请到这里查看:breaking change issues。

5.1 现在生成模块代码时会带有"use strict";头

在 ES6 模式下模块总是在严格模式下解析，对于生成目标为非 ES6 的却不是这样。从 TypeScript 1.8 开始，生成的模块将总为严格模式。这应该不会对现有的大部分代码产生影响，因为 TypeScript 把大多数因为严格模式而产生的错误当做编译时错误，但还是有一些在运行时才发生错误的 TypeScript 代码，比如赋值给 NaN，现在将会直接报错。你可以参考 MDN Article 学习关于严格模式与非严格模式的区别。
若想禁用这个行为，在命令行里传--noImplicitUseStrict 选项或在 tsconfig.json 文件里指定。

5.2 从模块里导出非局部名称

依据 ES6/ES2015 规范，从模块里导出非局部名称将会报错。
例子

```
export { Promise }; // Error
```

推荐
在导出之前，使用局部变量声明捕获那个全局名称。

```
const localPromise = Promise;
export { localPromise as Promise };
```


10.3.1 Promises

下面是改进后的 Promise 检查的例子：

```
let p = new Promise((c, e) => { c(12) });
let u: Promise<number> = p;
    ~
    类型 'Promise<{}>' 不能赋值给 'Promise<number>'
```

TypeScript 无法在调用 new Promise 时推断类型参数 T 的值。 因此，它仅推断为 Promise<{}>。 不幸的是，它会允许你这样写 c(12)和 c('foo')，就算 p 的声明明确指出它应该是 Promise<number>。
在新的规则下，Promise<{}>不能够赋值给 Promise<number>，因为它破坏了 Promise 的回调函数。 TypeScript 仍无法推断类型参数，所以你只能通过传递类型参数来解决这个问题：

```
let p: Promise<number> = new Promise<number>((c, e) => { c(12) });
//                                ^^^^^^^^ 明确的类型参数
```

它能够帮助从 promise 代码体里发现错误。 现在，如果你错误地调用 c('foo')，你就会得到一个错误提示：

```
let p: Promise<number> = new Promise<number>((c, e) => { c('foo') });
//                                                       ~~~~~
//  参数类型 '"foo"' 不能赋值给 'number'
```

10.3.2 （嵌套）回调

其它类型的回调也会被这个改进所影响，其中主要是嵌套的回调。 下面是一个接收回调函数的函数，回调函数又接收嵌套的回调。 嵌套的回调现在会以协变的方式检查。

```
declare function f(
  callback: (nested: (error: number, result: any) => void, index:
number) => void
): void;

f((nested: (error: number) => void) => { log(error) });
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'(error: number) => void' 不能赋值给 '(error: number, result: any) =>
void'
```

修复这个问题很容易。给嵌套的回调传入缺失的参数：

```
f((nested: (error: number, result: any) => void) => { });
```

10.4 更严格的泛型函数检查

TypeScript 在比较两个单一签名的类型时会尝试统一类型参数。 结果就是，当关系到两个泛型签名时检查变得更严格了，但同时也会捕获一些 bug。

```
type A = <T, U>(x: T, y: U) => [T, U];
type B = <S>(x: S, y: S) => [S, S];

function f(a: A, b: B) {
    a = b;  // Error
    b = a;  // Ok
}
```

推荐做法
或者修改定义或者使用--noStrictGenericChecks。

```
const x: X<> = new X<>();  // Error: Type parameter list cannot be empty.
```

10 v2.4

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.4.html

完整的破坏性改动列表请到这里查看:breaking change issues。

10.1 弱类型检测

TypeScript 2.4 引入了“弱类型（weak type）”的概念。 若一个类型只包含可选的属性，那么它就被认为是_弱（weak）_的。 例如，下面的 Options 类型就是一个弱类型：

```
interface Options {
    data?: string,
    timeout?: number,
    maxRetries?: number,
}
```

TypeScript 2.4，当给一个弱类型赋值，但是它们之前没有共同的属性，那么就会报错。 例如：

```
function sendMessage(options: Options) {
    // ...
}

const opts = {
    payload: "hello world!",
    retryOnFail: true,
}

// 错误!
sendMessage(opts);
// 'opts'与'Options'之间没有共同的属性
// 你是否想用'data'/'maxRetries'来替换'payload'/'retryOnFail'
```

推荐做法

1. 仅声明那些确定存在的属性。
2. 给弱类型添加索引签名（如：[propName: string]: {}）
3. 使用类型断言（如：opts as Options）

10.2 推断返回值的类型

TypeScript 现在可从上下文类型中推断出一个调用的返回值类型。 这意味着一些代码现在会适当地报错。 下面是一个例子：

```
let x: Promise<string> = new Promise(resolve => {
    resolve(10);
    //      ~~ 错误! 'number'类型不能赋值给'string'类型
});
```

10.3 更严格的回调函数参数变化

TypeScript 对回调函数参数的检测将与立即签名检测协变。 之前是双变的，这会导致有时候错误的类型也能通过检测。 根本上讲，这意味着回调函数参数和包含回调的类会被更细致地检查，因此 Typescript 会要求更严格的类型。 这在 Promises 和 Observables 上是十分明显的。

5.3 默认启用代码可达性（Reachability）检查

TypeScript 1.8 里，我们添加了一些可达性检查来阻止一些种类的错误。特别是：

1. 检查代码的可达性（默认启用，可以通过 allowUnreachableCode 编译器选项禁用）

```
2.    function test1() {
3.        return 1;
4.        return 2; // error here
5.    }
6.
7.    function test2(x) {
8.        if (x) {
9.            return 1;
10.       }
11.       else {
12.           throw new Error("NYI")
13.       }
14.       var y = 1; // error here
15.   }
```

16. 检查标签是否被使用（默认启用，可以通过 allowUnusedLabels 编译器选项禁用）

```
17. l: // error will be reported - label `l` is unused
18. while (true) {
19. }
20.
21. (x) => { x:x } // error will be reported - label `x` is unused
```

22. 检查是否函数里所有带有返回值类型注解的代码路径都返回了值（默认启用，可以通过 noImplicitReturns 编译器选项禁用）

```
23. // error will be reported since function does not return anything explicitly when `x` is falsy.
24. function test(x): number {
25.     if (x) return 10;
26. }
```

27. 检查控制流是否能进到 switch 语句的 case 里（默认禁用，可以通过 noFallthroughCasesInSwitch 编译器选项启用）。注意没有语句的 case 不会被检查。

```
28. switch(x) {
29.     // OK
30.     case 1:
31.     case 2:
32.         return 1;
33. }
34. switch(x) {
35.     case 1:
36.         if (y) return 1;
37.     case 2:
38.         return 2;
39. }
```

如果你看到了这些错误，但是你认为这时的代码是合理的话，你可以通过编译选项来阻止报错。

5.4 --module 不允许与--outFile 一起出现，除非 --module 被指定为 amd 或 system

之前使用模块指定这两个的时候，会生成空的 out 文件且不会报错。

5.5 标准库里的 DOM API 变动

- **ImageData.data** 现在的类型为 Uint8ClampedArray 而不是 number[]。查看 #949。
- **HTMLSelectElement .options** 现在的类型为 HTMLCollection 而不是 HTMLSelectElement。查看#1558。
- **HTMLTableElement.createCaption**，**HTMLTableElement.createTBody**，**HTMLTableElement.createTFoot**，**HTMLTableElement.createTHead**，**HTMLTableElement.insertRow**，**HTMLTableSectionElement.insertRow** 和 **HTMLTableElement.insertRow** 现在返回 HTMLTableRowElement 而不是 HTMLElement。查看#3583。
- **HTMLTableRowElement.insertCell** 现在返回 HTMLTableCellElement 而不是 HTMLElement 查看#3583。
- **IDBObjectStore.createIndex** 和 **IDBDatabase.createIndex** 第二个参数类型为 IDBObjectStoreParameters 而不是 any。查看#5932。
- **DataTransferItemList.Item** 返回值类型变为 DataTransferItem 而不是 File。查看#6106。
- **Window.open** 返回值类型变为 Window 而不是 any。查看#6418。
- **WeakMap.clear** 被移除。查看#6500。

5.6 在 super-call 之前不允许使用 this

ES6 不允许在构造函数声明里访问 this。

比如：

```
class B {
    constructor(that?: any) {}
}

class C extends B {
    constructor() {
        super(this);  // error;
    }
}

class D extends B {
    private _prop1: number;
    constructor() {
        this._prop1 = 10;  // error
        super();
    }
}
```

6 v2.0

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.0.html


```
func(() => { });
```

或者，你可以给 callback 的参数指定一个明确的类型：

```
func((c:number) => { });
```

7.6 逗号操作符使用在无副作用的表达式里时会被标记成错误

大多数情况下，这种在之前是有效的逗号表达式现在是错误。

示例

```
let x = Math.pow((3, 5)); // x = NaN, was meant to be `Math.pow(3, 5)`

// This code does not do what it appears to!
let arr = [];
switch(arr.length) {
  case 0, 1:
    return 'zero or one';
  default:
    return 'more than one';
}
```

推荐

--allowUnreachableCode 会禁用产生警告在整个编译过程中。或者，你可以使用 void 操作符来镇压这个逗号表达式错误：

```
let a = 0;
let y = (void a, 1); // no warning for `a`
```

7.7 标准库里的 DOM API 变动

- **Node.firstChild**，**Node.lastChild**，**Node.nextSibling**，**Node.previousSibling**，**Node.parentElement** 和 **Node.parentNode** 现在是 Node | null 而非 Node。
  查看#11113 了解详细信息。
  推荐明确检查 null 或使用!断言操作符（比如 node.lastChild!）。

8 v2.2

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.2.html

完整的破坏性改动列表请到这里查看:breaking change issues.

8.1 标准库里的 DOM API 变动

- 现在标准库里有 Window.fetch 的声明；仍依赖于@types\whatwg-fetch 会产生声明冲突错误，需要被移除。
- 现在标准库里有 ServiceWorker 的声明；仍依赖于@types\service_worker_api 会产生声明冲突错误，需要被移除。

9 v2.3

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.3.html

完整的破坏性改动列表请到这里查看:breaking change issues.

9.1 空的泛型列表会被标记为错误

示例

```
class X<> {}  // Error: Type parameter list cannot be empty.
function f<>() {}  // Error: Type parameter list cannot be empty.
```

但是，任何 FooError 的子类也必须要手动地设置原型。 对于那些不支持 Object.setPrototypeOf 的运行时环境，你可以使用__proto__。

不幸的是，[这些变通方法在 IE10 及其之前的版本](https://msdn.microsoft.com/en-us/library/s4esdbwz(v=vs.94).aspx) 你可以手动地将方法从原型上拷贝到实例上（比如从 FooError.prototype 到 this），但是原型链却是无法修复的。

7.3 const 变量和 readonly 属性会默认地推断成字面类型

默认情况下，const 声明和 readonly 属性不会被推断成字符串，数字，布尔和枚举字面量类型。这意味着你的变量/属性可能具有比之前更细的类型。这将体现在使用===和!==的时候。

示例

```
const DEBUG = true; // 现在为`true`类型，之前为`boolean`类型

if (DEBUG === false) { // 错误： 操作符'==='不能应用于'true'和'false'
    ...
}
```

推荐

针对故意要求更加宽泛类型的情况下，将类型转换成基础类型：

```
const DEBUG = <boolean>true; // `boolean`类型
```

7.4 不对函数和类表达式里捕获的变量进行类型细化

当泛型类型参数具有 string，number 或 boolean 约束时，会被推断为字符串，数字和布尔字面量类型。此外，如果字面量类型有相同的基础类型（如 string），当没有字面量类型做为推断的最佳超类型时这个规则会失效。

示例

```
declare function push<T extends string>(...args: T[]): T;

var x = push("A", "B", "C"); // 推断成 "A" | "B" | "C" 在 TS 2.1，在 TS 2.0 里为 string
```

推荐

在调用处明确指定参数类型：

```
var x = push<string>("A", "B", "C"); // x 是 string
```

7.5 没有注解的 callback 参数如果没有与之匹配的重载参数会触发 implicit-any 错误

在之前编译器默默地赋予 callback（下面的 c）的参数一个 any 类型。原因关乎到编译器如何解析重载的函数表达式。从 TypeScript 2.1 开始，在使用--noImplicitAny 时，这会触发一个错误。

示例

```
declare function func(callback: () => void): any;
declare function func(callback: (arg: number) => void): any;

func(c => { });
```

推荐

删除第一个重载，因为它实在没什么意义；上面的函数可以使用 1 个或 0 个必须参数调用，因为函数可以安全地忽略额外的参数。

```
declare function func(callback: (arg: number) => void): any;

func(c => { });
```


完整的破坏性改动列表请到这里查看:breaking change issues.

6.1 对函数或类表达式的捕获变量不进行类型细化(narrowing)

类型细化不会在函数，类和 lambda 表达式上进行。

例子

```
var x: number | string;

if (typeof x === "number") {
    function inner(): number {
        return x; // Error, type of x is not narrowed, c is number | string
    }
    var y: number = x; // OK, x is number
}
```

编译器不知道回调函数什么时候被执行。考虑下面的情况：

```
var x: number | string = "a";
if (typeof x === "string") {
    setTimeout(() => console.log(x.charAt(0)), 0);
}
x = 5;
```

当 x.charAt()被调用的时候把 x 的类型当作 string 是错误的，事实上它确实不是 string 类型。

推荐

使用常量代替：

```
const x: number | string = "a";
if (typeof x === "string") {
    setTimeout(() => console.log(x.charAt(0)), 0);
}
```

6.2 泛型参数会进行类型细化

例子

```
function g<T>(obj: T) {
    var t: T;
    if (obj instanceof RegExp) {
        t = obj; // RegExp is not assignable to T
    }
}
```

推荐 可以把局部变量声明为特定类型而不是泛型参数或者使用类型断言。

6.3 只有 get 而没有 set 的存取器会被自动推断为 readonly 属性

例子

```
class C {
  get x() { return 0; }
}

var c = new C();
c.x = 1; // Error Left-hand side is a readonly property
```

推荐

定义一个不对属性写值的 setter。

6.4 在严格模式下函数声明不允许出现在块(block)里

在严格模式下这已经是一个运行时错误。从 TypeScript 2.0 开始，它会被标记为编译时错误。

例子

```
if( true ) {
    function foo() {}
}

export = foo;
```

推荐

使用函数表达式代替：

```
if( true ) {
    const foo = function() {}
}
```

6.5 TemplateStringsArray 现是是不可变的

ES2015 模版字符串总是将它们的标签以不可变的类数组对象进行传递，这个对象带有一个 raw 属性（同样是不可变的）。 TypeScript 把这个对象命名为 TemplateStringsArray。便利的是，TemplateStringsArray 可以赋值给 Array<string>，因此你可以利用这个较短的类型来使用标签参数：

```
function myTemplateTag(strs: string[]) {
    // ...
}
```

然而，在 TypeScript 2.0，支持用 readonly 修饰符表示这些对象是不可变的。 这样的话，TemplateStringsArray 就变成了不可变的，并且不再可以赋值给 string[]。

推荐

直接使用 TemplateStringsArray（或者使用 ReadonlyArray<string>）。

7 v2.1

http://www.patrickzhong.com/TypeScript/zh/breaking-changes/typescript-2.1.html

完整的破坏性改动列表请到这里查看:breaking change issues.

7.1 生成的构造函数代码将 this 的值替换为 super(...)调用的返回值

在 ES2015 中，如果构造函数返回一个对象，那么对于任何 super(...)的调用者将隐式地替换掉 this 的值。 因此，有必要获取任何可能的 super(...)的返回值并用 this 进行替换。

示例

定义一个类 C：

```
class C extends B {
    public a: number;
    constructor() {
        super();
        this.a = 0;
    }
}
```

将生成如下代码：

```
var C = (function (_super) {
    __extends(C, _super);
    function C() {
        var _this = _super.call(this) || this;
        _this.a = 0;
        return _this;
    }
    return C;
}(B));
```

注意：

- _super.call(this)存入局部变量_this
- 构造函数体里所有使用 this 的地方都被替换为 super 调用的返回值（例如_this）
- 每个构造函数将明确地返回它的 this，以确保正确的继承

值得注意的是在 super(...)调用前就使用 this 从 TypeScript 1.8 开始将会引发错误。

7.2 继承内置类型如 Error，Array 和 Map 将是无效的

做为将 this 的值替换为 super(...)调用返回值的一部分，子类化 Error，Array 等的结果可以是非预料的。 这是因为 Error，Array 等的构造函数会使用 ECMAScript 6 的 new.target 来调整它们的原型链； 然而，在 ECMAScript 5 中调用构造函数时却没有有效的方法来确保 new.target 的值。 在默认情况下，其它低级别的编译器也普遍存在这个限制。

示例

针对如下的子类：

```
class FooError extends Error {
    constructor(m: string) {
        super(m);
    }
    sayHello() {
        return "hello " + this.message;
    }
}
```

你会发现：

- 由这个子类构造出来的对象上的方法可能为 undefined，因此调用 sayHello 会引发错误。
- instanceof 应用于子类与其实例之前会失效，因此(new FooError()) instanceof FooError 会返回 false。

推荐

做为一个推荐，你可以在任何 super(...)调用后立即手动地调整原型。

```
class FooError extends Error {
    constructor(m: string) {
        super(m);

        // Set the prototype explicitly.
        Object.setPrototypeOf(this, FooError.prototype);
    }

    sayHello() {
        return "hello " + this.message;
    }
}
```