TypeScript 类型系统梳理习题参考答案

大部分题目都是开放性问题，**没有标准答案**，这里供大家参考

1. 举例说明如何用 interface 定义函数

```ts
interface FunCheckUser {
  (name: string, age: number): boolean;
}

let checkUser: FunCheckUser = (name, age) => {
  return age > name.length;
}
```

2. 为什么 TypeScript 要支持用 interface 定义函数的语法

因为 JavaScript 里 Function 也是 Object，可以具有数据结构
使用 interface 语法去承载这种情况比较自然

```ts
function message(data: string) {
  console.log(data, message.asdf);
}
message.asdf = "Hello World!";

interface FunctionMixObject {
  (data: string): void;
  asdf: string;
}

let func: FunctionMixObject = message;
func('test')                          // "test",  "Hello World!"
console.log('func.asdf', func.asdf) // "func.asdf",  "Hello World!"
```

3. 需要用 Object 表达 Key 是 string，Value 是 number，举例说明如何用 interface 定义它

```ts
var obj = {};
obj["hello"] = 5;
obj["world"] = 10086;
...
obj["lceda"] = 66666;
```

```ts
interface A {
  [key: string]: number
};
```

4. Interface 具有父子关系，那么函数类型是否也有父子关系？

```ts
type Func = (a:number, b:string) => string;

// 为什么函数签名不一样，但是编译器通过了
let f:Func = function (b:number) {
    return b.toString();
}
```

```ts
// 父类型的函数，编译器可以自动扩展形成子类型的函数，并且函数功能不会有差异
let f:Func = function (b:number, __compiler_auto_add_param__: string) {
    return b.toString();
}
```

5. 从 TypeScript 工具类型文档 中至少找 2 个类型，分析其实现方式

Exclude 类型能求类型减法，将泛型 T 中减去泛型 U
实现方式是采用类型的问号表达式，将类型 T 中属于 U 的部分，短路到 never 从而消除

```ts
type Exclude<T, U> = T extends U ? nenver : T;

type T0 = Exclude<"a" | "b" | "c", "a">;
//    ^ = type T0 = "b" | "c"
type T1 = Exclude<"a" | "b" | "c", "a" | "b">;
//    ^ = type T1 = "c"
type T2 = Exclude<string | number | (() => void), Function>;
//    ^ = type T2 = string | number
```

Extract 类型能求类型的交集，返回同时存在于泛型 T 和 U 的类型
实现方式是采用类型的问号表达式，将类型 T 中不属于 U 的部分，短路到 nerver 从而消除

```ts
type Extract<T, U> = T extends U ? T : never;

type T0 = Extract<"a" | "b" | "c", "a" | "f">;
//    ^ = type T0 = "a"
type T1 = Extract<string | number | (() => void), Function>;
//    ^ = type T1 = () => void
```

6. 谈谈 Union Type 是如何解决 “NULL”：计算机科学中的最严重错误，造成十亿美元损失

Null 的主要问题，在于在类似 Java 等编程语言中，除了基本类型，都等于默认 Union 了 null

```
string a = "hello"; // 这个 a 的类型其实是 string | null
a = null; // 合法
a.lowerCase(); // 直到运行时才会引起 NPE 异常
```

在开启编译器参数 strictNullChecks 后，TypeScript 中 null 和 undefined 不再和其它类型兼容

```
let a: string = "hello";
a = null; // 编译器报错
```

如果某些场景需要用 null 表达一些特殊的状态，仍然可以用 Union Type 让编译器严谨检查

```
function decode(s: string): string | null {
  if (s.length === 0) {
    return null;
  } else {
    return s.split(',').join('\r\n');
  }
}

let result = decode("abcd");

console.log(result.toLowerCase()); // 编译报错，明确指出 result 可能为 null

if (result !== null) {
  console.log(result.toLowerCase());
}
```

1. 查阅资料尝试说明 class 为什么 extends 只能单继承，而 implements 允许同时实现多个，如果采用类似 C++ 可以 extends 多个类，会带来哪些问题

C++ 的多继承，最大的问题是很多行为比较晦涩难懂

违背了程序语言应该简化开发者心智负担的初衷。比如：

a. 类的初始化顺序是隐晦的，和继承结构与代码里的顺序有关
b. 当多个同级父类函数签名相同时，重载规则复杂
c. 父类继承结构改变可能引起连锁反应，这些都不是普通开发者能很好掌握的
   更确切说这种不直观的反而容易造成误用的语法，违背了编程语言的初衷，是错误的语言设计
   就像之前提到的 null 的问题一样

implements 多个 interface 之所以没问题可以分几个角度理解

a. interface 只是规范，不包含实现，没有任何隐含的初始化顺序与调用目标的语义
b. Implements 多个 interface 其实就是实现这些 interface 的并集

只是类型的集合运算，同名、多个同级、上下级不会产生隐晦的选择问题

2. TypeScript 使用 Duck typing 类型体系，上文讲了 class 与 interface 的兼容性规则，那么 class 与 class 的兼容性规则呢?
   a. 尝试描述含有 private 的 class 的兼容规则
   b. 如果可以，寻找资料或者自己思考，进一步分析为什么要这么做

Class-Class 的兼容性规则如下

a. 先判断两个类里所有 成员/方法 的父子关系，等价于两个 interface 的父子关系判定
b. 特别的，Class 里的 private/protected 成员/方法
   编译器会隐含地将其改成全局唯一的内部名称，只有 extends 才能在另一个类里共享这个名称
   所以两个非继承关系的 class 里，同名 private/protected 是不同的名称从而类型不兼容

Class-Class 兼容性规则对 private/protected 特殊的规定，是为了完整实现 private/protected 语义

```
class A {
  private age: number = 5;

  cloneAge(other: A) {
    other.age = this.age;
  }
}

class B {
  private age: number = 5;

  cloneAge(other: B) {
    throw new Error("change age is not permitted")
  }
}

// 如果不做特例，A 和 B 的类型外观（type shape）是完全一致的
// 如果在 interface 里就是完全等价的类型

let a: A = new A();
let b: B = new B();

a.cloneAge(b); // 编译报错，A 与 B 私有成员 age 不兼容
// 如果它们类型兼容，那么 A 的 cloneAge 会将 B 的 age 修改了
// 然而这样违背了 class B 中 age 不应该更改的初衷
```

这个规则是来自 2016 年的一次 bug 报告

3. 因为 函数式编程范式 早期很多语法特性不够成熟，后来 面向对象编程范式 成了主流

但是纯粹的 面向对象范式 由于过于依赖单一的 class 机制，无法传递函数，没有闭包，导致很多设计相对来说较为繁琐，发展出了很多【设计模式】

这里 有一个案例，虽然使用 TypeScript，但用了纯粹的 面向对象范式 去解决问题，尝试运用所学简化它的设计

注意：在 TypeScript 可以同时使用两种编程范式，没必要舍此即彼，简化设计不是要求完全用函数式

```
interface Observer {
  // Receive update from subject.
  (subject: Subject): void;
}

const observer1: Observer = (subject: Subject) => {
  if (subject instanceof ConcreteSubject && subject.state < 3) {
    console.log('ConcreteObserverA: Reacted to the event.');
  }
}

const observer2: Observer = (subject: Subject) => {
  if (subject instanceof ConcreteSubject
    && (subject.state === 0 || subject.state >= 2)) {
    console.log('ConcreteObserverB: Reacted to the event.');
  }
}
```

这个案例场景太简单，体现不出函数传递，函数闭包的语法优势，未来大家在工作中慢慢发掘