原型模式**：**

将一个对象作为原型，通过对其进行复制这个原型来创建新的对象。

### 结构特点：

Prototype 抽象原型类：规定原型对象要实现的接口clone()

ConcretePrototype 具体原型类：实现抽象原型类的clone()方法，在clone方法中返回自己的一个克隆对象。

Client 客户类：让一个原型对象克隆自身从而创建一个新的对象，只需要直接实例化或通过工厂方法等方式创建一个原型对象，再通过调用该对象的克隆方法即可得到多个相同的对象。

Prototype

clone()

ConcretePrototype

clone()

Client

operation()

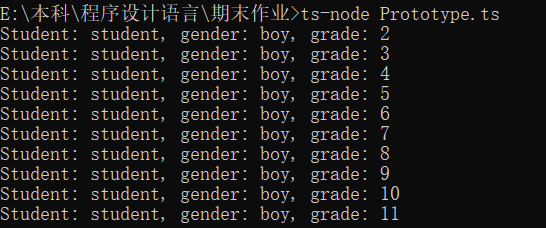
### 适用场景：

①需要快速创建大量相同对象的实例，而创建新对象成本较大。新实例可以通过复制已有实例获得。如果是相似的对象的实例，则可以对其成员变量稍作修改。②需要保存对象的状态，而对象的状态变化很小。③需要避免使用分层次的工厂类来创建分层次的对象，并且类的实例只有一个或很少的几个组合状态，通过复制原型对象得到新实例比使用构造函数创建一个新实例更加方便。

### 面向对象实现：

以Student类为例，在Student类内部实现clone():Student接口. StudentClient类封装了用户的一个自定义操作：实例化一个具有默认值的Student对象s，并通过clone()接口拷贝生成10个Student实例。每次拷贝让原型s的年级grade属性自增1. 执行结果如下图所示。可以看到Student的clone()方法实现了深拷贝。

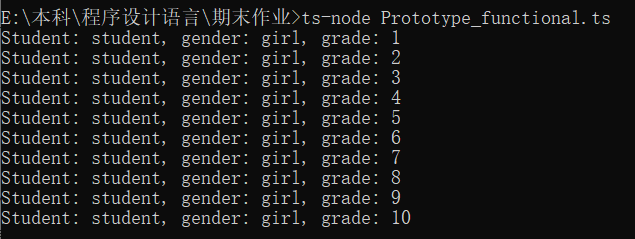
1. //Prototype: Specify the interface that needs to be implemented by the prototype object
2. interface Prototype{
3. clone():Prototype
4. }
6. //ConcretePrototype : let's take class student as an example
7. **class** Student implements Prototype{
8. **private** name: string;
9. **private** gender: string;
10. **private** grade: number;
11. constructor(){
12. **this**.name = "student";
13. **this**.gender = "boy";
14. **this**.grade = 1;
15. }
16. **public** clone(): Student {
17. let result = **new** Student;
18. result.name = **this**.name;
19. result.gender = **this**.gender;
20. result.grade = **this**.grade;
21. **return** result;
22. }
23. **public** toString(): string {
24. **return** "Student: " + **this**.name + ", gender: " + **this**.gender + ", grade: " + **this**.grade;
25. }
26. **public** gradePlus(): **void**{
27. **this**.grade++;
28. }
29. }
31. //Client : StudentClient has a method that create 10 students from class Student by clone.
32. **class** StudentClient {
33. **public** operation(): **void**{
34. let students = **new** Array;
35. let s = **new** Student();
36. **for**(let i = 0; i < 10; i++){
37. let s\_ = s.clone();
38. students.push(s\_);
39. s.gradePlus();
40. }
41. **for**(let i = 0; i < 10; i++)
42. console.log(students[i].toString());
43. }
44. }
46. //user operation
47. let sc = **new** StudentClient();
48. sc.operation();



### 函数式写法改造：

把原型Prototype 变为一个返回Student对象的函数类型。原来封装在Student类内部的clone()方法变为function cloneStudent(s : Stud):Stud，被克隆的Student示例为函数的第一个参数s. 用户自定义的创建10个学生示例的方法现在需要接受一个类型为Prototype的参数作为克隆时需要用到的方法。具体改写内容如下。执行结果如下图所示：

1. **class** Stud{
2. **private** name: string;
3. **private** gender: string;
4. **private** grade: number;
5. constructor(){
6. **this**.name = "student";
7. **this**.gender = "girl";
8. **this**.grade = 1;
9. }
10. **public** getName(): string{
11. **return** **this**.name;
12. }
13. **public** setName(n: string): **void**{
14. **this**.name = n;
15. }
16. **public** getGender(): string{
17. **return** **this**.gender;
18. }
19. **public** setGender(g: string): **void**{
20. **this**.gender = g;
21. }
22. **public** getGrade(): number{
23. **return** **this**.grade;
24. }
25. **public** setGrade(g: number): **void**{
26. **this**.grade = g;
27. }
28. **public** gradePlus(): **void**{
29. **this**.grade++;
30. }
31. **public** toString(): string {
32. **return** "Student: " + **this**.name + ", gender: " + **this**.gender + ", grade: " + **this**.grade;
33. }
34. }
35. //ProtoType is a function that returns a Student
36. type ProtoType = (stu: Stud) => Stud;
37. //cloneStudent is an implement of ProtoType
38. function cloneStudent(s : Stud):Stud{
39. let result = **new** Stud;
40. result.setName(s.getName());
41. result.setGender(s.getGender());
42. result.setGrade(s.getGrade());
43. **return** result;
44. }
45. //User defined operation: a method that create 10 Students
46. function create10Students(clonefunc: ProtoType){
47. let s0 = **new** Stud;
48. let students = **new** Array;
49. **for**(let i = 0; i < 10; i++){
50. let s = clonefunc(s0);
51. students.push(s);
52. s0.gradePlus();
53. }
54. **for**(let i = 0; i < 10; i++)
55. console.log(students[i].toString());
56. }
58. create10Students(cloneStudent);



### 两种实现方式的差异性：

类型体系有所区别；概念的封装性上有区别；在建模上的粒度不一样；抽象的聚焦点不同；

面向对象的实现需要一个接口Prototype和至少一个该接口的实现Student。函数式实现只需要声明原型的类型()=>Student和一个实现拷贝行为的函数cloneStudent(). 面向对象的拷贝行为封装在原型类内，而函数式实现的拷贝行为实现在类外。这就需要函数式实现中的类给出访问和修改原型类的所有成员变量的接口，或者利用TypeScript的类型系统来实现对类成员的深拷贝。当我们需要处理多个函数的时候，可以使用更加冗长的面向对象解决方案。如果原型类接口声明了多个克隆方法，就无法使用一个函数类型来替代接口。