Q3`25-Slots-关于“基类构造调用子类成员函数修改子类成员变量”的问题分析-程序

原始案例:

1、原始代码

```
import { log } from "cc";

export class base {
    constructor() {
        console.warn("base class constructor called");
        this.initializeData();
    }

    initializeData() {
        console.warn("base class initializeData called");
    }
}
```

```
import { base } from "./base";

export class Derived extends base {
    chips: number[] = null;
    constructor() {
        console.warn("Derived class constructor called");
        super();
        console.warn("Derived class constructor called after base");

    }
    initializeData() {
        console.warn("Derived class initializeData called",this.chips);
        this.chips = [];
        this.chips.push(0);
    }
    addChip(chip: number) {
        this.chips.push(chip);
    }
    getChips() {
        return this.chips;
```

```
    }
}
```

```
import { _decorator, Component, Node } from 'cc';
import { base222 } from './base222';
const { ccclass, property } = _decorator;

@ccclass('Test')
export class Test extends Component {
    ctrl: derived = null;
    protected onLoad(): void {
        this.ctrl = new Derived();
    }

    protected onEnable(): void {
        this.ctrl.addChip(100); // 运行是报错: Cannot read properties of null
(reading 'push')
        console.log(this.ctrl.getChips());
    }
    update(deltaTime: number) {

    }
}
```

2、运行时报错:

```
▶ base222 class constructor called                                  base222.ts:6
▶ base class constructor called                                     base.ts:6
▶ base222 class initializeData called undefined                     base222.ts:12
▶ base222 class constructor called after base                       base222.ts:8
▶ ▶ ErrorEvent                                                      (索引) :283
Uncaught TypeError: Cannot read properties of null (reading 'push')  base222.ts:17
    at base222.addChip (base222.ts:17:20)
    at Test.onEnable (test.ts:13:19)
    at OneOffInvoker.invokeOnEnable [as _invoke] (component-scheduler.ts:332:18)
    at OneOffInvoker.invoke (component-scheduler.ts:162:14)
    at NodeActivator.activateNode (node-activator.ts:163:31)
    at Scene._activate (scene.ts:199:42)
    at Director.runSceneImmediate (director.ts:477:15)
    at main.js:1:4332
    at asset-manager.ts:827:31
    at utilities.ts:326:13
```

3、结论：

- “子类成员属性”定义，覆盖了“基类对象”构造时的子类成员属性定义；
- JS、TS 子类属性定义覆盖发生在：子类构造中 super() 返回时，立即生效，早于任何 super() 语句后的逻辑；

问题解构：

本质上是关于：

- **运行时多态：基类构造中调用子类成员函数问题；**
- **类成员初始化：继承关系中的成员变量、属性的初始化顺序、覆盖顺序问题；**

拓展分析：

⚙️ 一、运行时多态实现机制

语言	核心机制	特点与限制
C++	**虚函数表（vtable）+ 虚指针（vptr）**基类声明 `virtual` 函数，派生类通过重写实现多态。	动态绑定通过 vptr 指向 vtable 实现，运行时根据对象实际类型调用函数。**开销**：两次内存访问（取 vptr → 取函数地址）。**限制**：构造函数/析构函数中调用虚函数**无多态**（vptr 未初始化或已销毁）。
C#/Java	**虚方法表 + 接口表**使用 `virtual`/`abstract` 标记方法，派生类用 `override` 重写。	多态通过 CLR（C#）或 JVM（Java）在运行时解析。**安全性**：支持 `sealed`（C#）/`final`（Java）阻止进一步重写。**接口多态**：通过接口实现跨类多态，无需继承关系。

语言	核心机制	特点与限制
JS/TS	**原型链查找**通过原型链动态解析方法，子类覆盖父类方法实现多态。	**动态类型**：运行时根据对象原型链查找方法，无编译时类型检查（TS 提供类型约束但运行时行为同 JS）。**灵活性**：可通过 `call`/`apply` 手动绑定 `this`，实现类似多态。

关键差异：

- C++ 依赖手动管理的 vtable，C#/Java 由运行时环境托管，JS/TS 完全动态。
- C++ 多态**不适用于构造/析构阶段**，而 C#/Java 在构造函数中调用虚方法会**触发子类重写**（但可能因未完全初始化导致风险）。

📦 二、成员变量与属性初始化顺序

1. 初始化顺序规则

语言	初始化顺序	示例与说明
C++	1. 基类静态成员 2. 派生类静态成员 3. 基类成员变量（按声明顺序） 4. 基类构造函数 5. 派生类成员变量（按声明顺序） 6. 派生类构造函数。	成员变量初始化**与声明顺序一致**，与初始化列表顺序无关：`class A { int a, b; A(): b(0), a(b+2) {} }`→a 先初始化（值为未定义的 `b+2`），风险高！
C#	1. 子类静态字段 2. 父类静态字段 3. 子类实例字段 4. 父类实例字段 5. 父类构造函数 6. 子类构造函数。	字段初始化器优先于构造函数执行：`class B : A { private List<int> chips = new List<int>(); }`→chips 在基类构造前已初始化。
Java	1. 父类静态成员 2. 子类静态成员 3. 父类实例成员 4. 父类构造函数 5. 子类实例成员 6. 子类构造函数。	**与 C# 相反**：父类成员先于子类初始化。若父类构造函数调用虚方法，会调用子类重写方法，但子类成员尚未初始化。
JS/TS	无固定顺序，依赖代码执行流程。 TS 类成员按声明顺序初始化，但无静态/实例的严格阶段划分。	需手动控制初始化时机：`class Derived extends Base { data = []; }`→data 在 `super()` 后立即初始化，早于显示构造。

2. 静态成员初始化

- **C++/C#/Java**：静态成员在类加载时初始化（程序启动或首次使用类），且只执行一次。

- **JS/TS**：静态成员在类定义时初始化（代码执行到类声明处）。

⚠️ 三、基类调用子类成员的安全性分析

1. 构造函数中调用虚方法

语言	行为	风险
C++	基类构造中调用虚函数 → **调用基类实现**（无多态）。	安全但无多态，无法访问子类成员。
C#/Java	基类构造中调用虚函数 → **调用子类重写方法**。	**高风险**：子类成员可能未初始化（Java 子类成员未构造，C# 子类字段初始化器已执行但构造函数未运行）。
JS/TS	行为同 C#，调用子类方法，但子类成员可能未定义。	需确保子类成员在方法中被访问前已初始化。

以 C# 为例，由于**字段初始化器**优先于**构造函数**执行，以下代码虽然可以按预期执行，

```
class Base {
    public Base() { InitializeData(); } // 调用子类重写方法
    public virtual void InitializeData() {}
}
class Derived : Base {
    private List<int> data = new List<int>();
    public override void InitializeData() {
        data.Add(1); // data 已经由字段初始化器初始化，故可以正常执行
    }
}
```

但存在安全隐患，eg:

```
class Base {
    public Base() { InitializeData(); } // 调用子类重写方法
    public virtual void InitializeData() {}
}
class Derived : Base {
    private List<int> data;
    public override void InitializeData() {
        data.Add(1); // data 未初始化 → NullReferenceException
    }
}
```

2. 基类直接访问子类成员

- **所有语言**：**禁止**基类直接访问子类特有成员（违反封装原则）。
- **替代方案**：

> **抽象方法/属性**：基类定义抽象成员，强制子类实现（C++ 纯虚函数、C#/Java `abstract`）。
>
> **两段式初始化**：分离对象创建与初始化逻辑（如 `Initialize()` 方法）。

💎 四、总结

差异表：

维度	C++	C#	Java	JS/TS
运行时多态	虚函数表（手动管理）	虚方法表（CLR 托管）	虚方法表（JVM 托管）	原型链（动态查找）
初始化顺序	严格按静态 → 基类 → 派生类	子类静态 → 父类静态 → 子类实例 → 父类实例	父类静态 → 子类静态 → 父类实例 → 子类实例	无强制顺序
基类调子类成员	构造函数中无多态	构造函数中触发多态（高风险）	同 C#	同 C#

核心原则：

多态是提升代码灵活性的利器，但需谨慎处理对象生命周期与状态一致性，尤其在跨语言开发时更需注意机制差异。

实践守则：

1. **避免在构造函数中调用虚方法**：尤其在 C#/Java/JS/TS 中，易因未初始化导致运行时错误。
2. **遵守封装原则**：禁止基类直接访问子类特有成员。
3. **使用关键字限制重写**：对无需多态的方法添加限制，提升安全性与性能。
4. **语法检查**：利用 TS 类型检查或 C++ 的 `override` 关键字捕获重写错误。