【C++精讲系列】五分钟带你彻底搞懂虚函数，虚表和虚继承

已于 2026-01-05 16:33:08 修改
编辑

结论先行：

> 在 C++ 中，虚函数和虚继承并不是“高级特性”，而是两种**代价明确、目标不同**的机制。
> 虚函数解决的是**行为在运行期如何正确分发**的问题，是多态的核心工具；
> 而虚继承解决的则是**继承结构的冲突已经不可避免时，如何保证基类状态唯一**的问题。

在 C++ 中，“继承”和“多态”几乎是每个人都会接触到的概念。

语法层面，它们并不复杂：`virtual`、`override`、基类指针指向派生类对象，看起来一切都理所当然。

但我们真的思考过它们吗：

- 为什么**基类指针调用派生类函数**，能够在运行期正确分发？
- `virtual` 关键字究竟改变了什么？
- 多态带来的灵活性，**代价是什么**？
- 在高性能或底层代码中，虚函数到底该不该用？

这些问题的答案，都不在语法层面，而藏在 C++ 对象模型 之中。

本文不会停留在“什么是多态”这样的概念解释上，而是从**内存布局**和**运行时行为**出发，聚焦两个核心机制：**虚函数（virtual function）与虚函数表（vtable）**。
我们将一步步拆开 C++ 对象，看看一个看似普通的成员函数调用，在编译器和运行期到底经历了什么。

理解虚表，是为了在设计类层次、权衡性能、甚至阅读汇编和调试复杂 Bug 时，**心里有一张清晰的底图**。

从这里开始，聊一聊 C++ 多态背后真正发生的事情。

从一次普通的函数调用说起

在不涉及继承和多态的情况下，C++ 的函数调用其实非常“朴素”。

```cpp
struct A {
    void foo() {
        // ...
    }
};

int main() {
    A a;
    a.foo();
}
```

AI写代码cpp运行

这段代码在编译期就已经**完全确定**：`foo()` 是哪个函数、入口地址在哪里，编译器一清二楚。最终生成的指令，本质上就是一次**直接调用**。

也正因为如此，这类调用几乎没有运行时成本——没有额外的跳转，没有间接寻址，更谈不上什么“多态”。

当继承出现，发生了什么变化？

现在引入继承，但**不使用** `virtual`：

```cpp
struct Base {
    void foo() {
        // Base::foo
    }
};

struct Derived : public Base {
```

```cpp
    void foo() {
        // Derived::foo
    }
};

int main() {
    Base* p = new Derived;
    p->foo();
}
```

AI写代码cpp运行

很多初学者看到这里，直觉会认为调用的是 `Derived::foo()`，但事实恰恰相反——**这里调用的是 `Base::foo()`。**

原因并不神秘：`foo()` **不是虚函数**，编译器在编译期只看**指针的静态类型**，也就是 `Base*`，于是直接绑定到 `Base::foo()`。

换句话说，这依然是一次**静态绑定（static binding）**。

> **只要函数不是 `virtual`，继承关系的存在，对函数调用本身几乎没有任何影响。**

`virtual` 出现的真正意义

现在，只改一行代码：

```cpp
struct Base {
    virtual void foo() {
        // Base::foo
    }
};
```

AI写代码cpp运行

其它代码保持不变：

```cpp
Base* p = new Derived;
p->foo();
```

AI写代码cpp运行

这一次，调用结果变成了 `Derived::foo()`。

表面上看，只是多写了一个 `virtual`，但实际上，**整个调用机制已经彻底改变**：

- 编译期不再确定最终调用哪个函数
- 调用目标被推迟到**运行期**
- 对象内部必须携带额外的信息，用来完成**“动态分发”**

也正是在这里，C++ 引入了一个关键的数据结构——**虚函数表**（vtable）。

一个重要的分水岭

到这里，可以先明确一个结论：

> **是否使用 `virtual`，决定了一次函数调用是“编译期绑定”，还是“运行期分发”。**

而虚表，正是 C++ 为了实现这种运行期分发而采用的核心机制。

接下来，我们将暂时放下代码逻辑，从**对象的内存布局**入手，看看一个“带虚函数的对象”，在内存中究竟长什么样。

对象里多出来的那点东西：vptr

一旦一个类中**出现了至少一个虚函数**，这个类的对象在内存布局上，就和普通对象不一样了。

看这样一个最小示例：

```
struct Base {
    virtual void foo();
    int x;
};
```

AI写代码cpp运行

很多人第一次学虚表时，都会听到一句话：

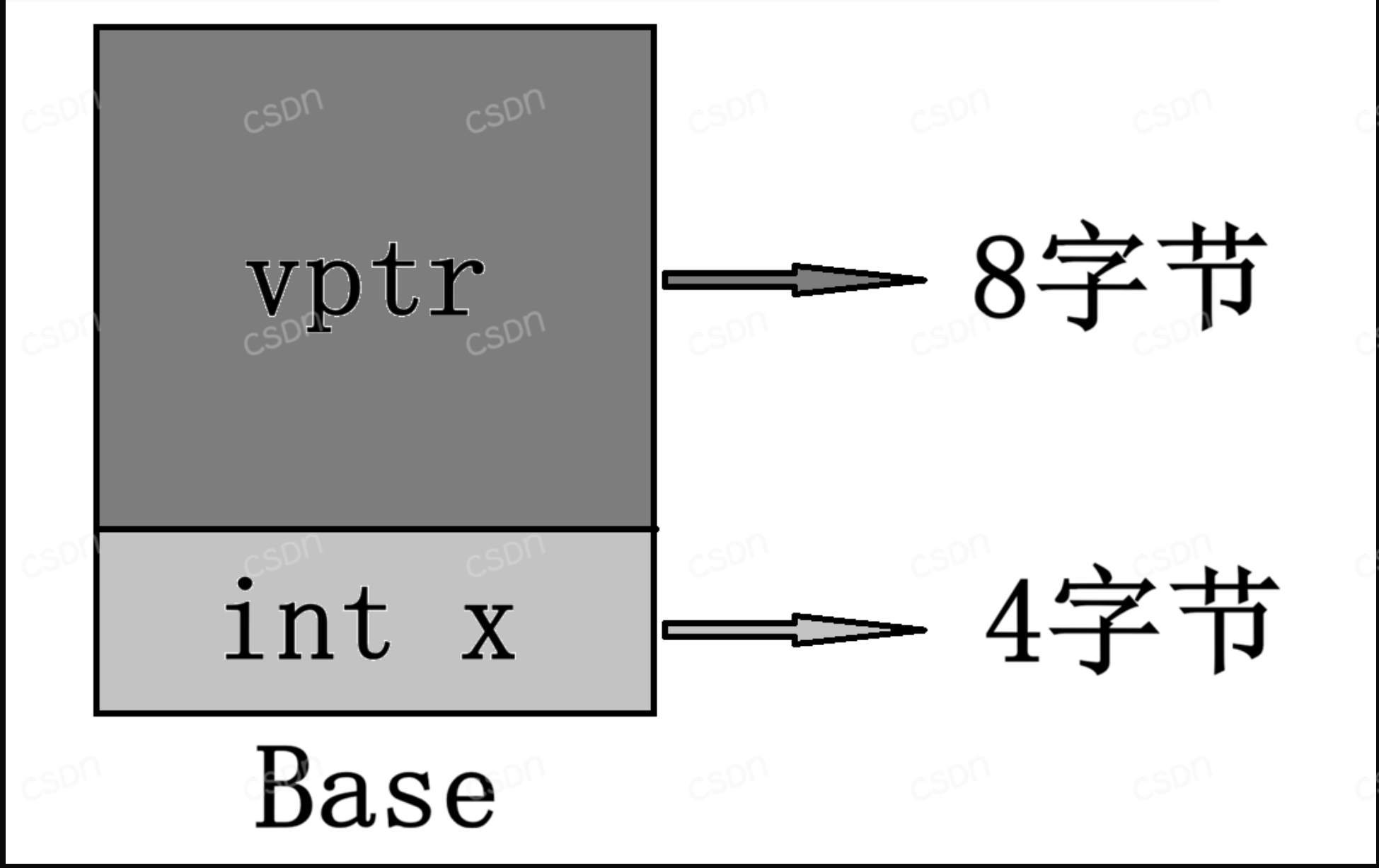
> “对象里有一张虚函数表。”

这句话**不准确**。更准确的说法是：

> **对象里保存的是一个指针，指向一张虚函数表。**

这个指针通常被称为 **vptr（virtual table pointer）**。

一个典型的内存布局



其中：

- vptr 是一个普通指针
- 它在**对象构造阶段被写入**

- 指向一个**只读、全局共享**的虚函数表

也就是说：

- **每个对象都有自己的 vptr**
- **同一类型的所有对象，共享同一张虚表**

虚函数表里到底有什么？

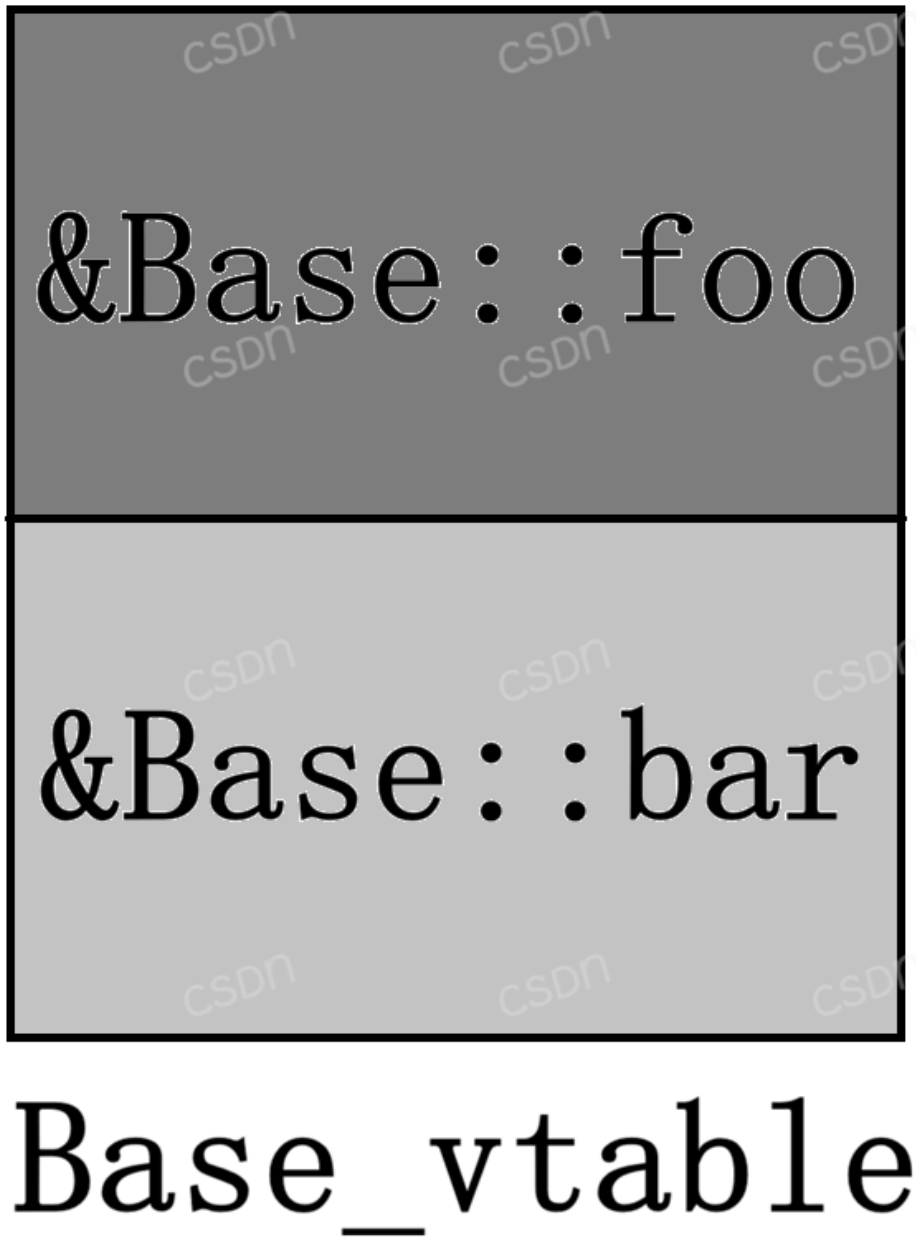
虚函数表本身，本质上就是一个**函数指针数组**。

还是上面的 Base：

```cpp
struct Base {
    virtual void foo();
    virtual void bar();
};
```

AI写代码cpp运行

它指向的虚表可以理解为：



顺序由编译器决定，但通常与虚函数在类中声明的顺序一致。

这里有一个非常关键的点：**虚表里存的是“最终可调用版本”的函数地址。**

这句话在讲继承和覆盖时，会变得非常重要。

派生类如何“改写”虚表？

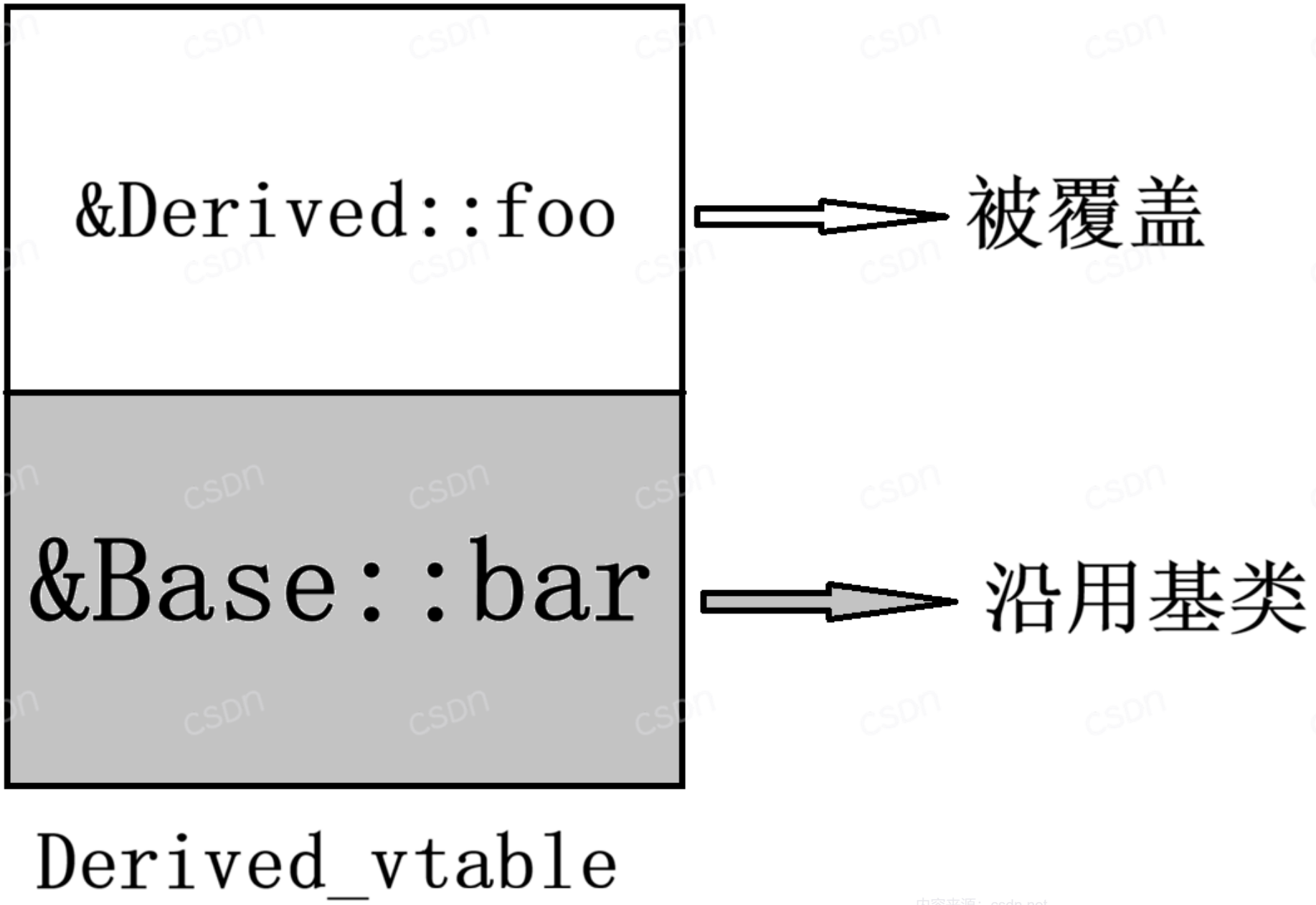
来看一个更完整的例子：

```cpp
struct Base {
    virtual void foo();
    virtual void bar();
};

struct Derived : public Base {
    void foo() override;
};
```

AI写代码cpp运行

Derived 覆盖了 foo()，但没有覆盖 bar()。那么它的虚表会长这样：



也就是说：

- **派生类会生成自己的一张虚表**
- 覆盖的虚函数，对应表项会被替换
- 没覆盖的虚函数，直接继承基类的实现

多态调用到底发生了什么？

现在回到那句经典代码：

```cpp
Base* p = new Derived;
p->foo();
```

AI写代码cpp运行

这行代码在运行时，实际发生的是：

1. 通过 `p` 找到对象首地址
2. 读取对象里的 `vptr`
3. 通过 `vptr` 找到 `Derived` 的虚表
4. 从虚表中取出 `foo()` 对应的函数指针
5. **间接调用**该函数

这就是为什么：

- 多态调用一定有一次**间接跳转**
- 它无法像普通函数调用那样被完全内联
- 在极端性能场景下，确实存在成本

一个容易被忽略的事实

virtual 并不是“函数的属性”，而是**参与对象布局和调用约定的设计选择**。

一旦类中出现虚函数：

- 对象体积发生变化
- 构造 / 析构阶段要维护 vptr
- 调用路径发生变化

这些变化，都会在后面产生连锁影响。

构造函数与析构函数中，虚函数为何“失效”？

很多人在第一次写出下面这段代码时，都会产生一个理所当然的期待：

```cpp
struct Base {
    Base() {
        foo();
    }
    virtual void foo() {
        std::cout << "Base::foo\n";
    }
};

struct Derived : public Base {
    Derived() {}
    void foo() override {
        std::cout << "Derived::foo\n";
    }
};
```

AI写代码cpp运行

```
Base* p = new Derived;
```

AI写代码cpp运行

直觉上会以为构造 Derived 时，foo() 应该调用 Derived::foo()。

但真实的输出是：

```
Base::foo
```

AI写代码cpp运行

不是“虚函数失效”，而是对象尚未成为派生类

关键不在于 virtual，而在于**对象此刻的“身份”**。

在 C++ 中，对象的**构造顺序**是：

1. **先构造基类子对象**
2. 再构造派生类自己的部分

在执行 Base 构造函数时：

- Derived 部分**尚未构造**
- 对象的动态类型，被视为 Base
- 对应的 vptr 指向的是 **Base 的虚表**

因此：

```cpp
Base::Base() {
    foo();  // 通过 Base 的 vptr 调用
}
```

AI写代码cpp运行

即使当前内存即将成为一个 Derived 对象，但在这一刻，它**还不是**。

析构函数中也是同样的道理

析构阶段的情况，恰好相反，但逻辑一致。

```cpp
struct Base {
    virtual ~Base() {
        foo();
    }
    virtual void foo() {
        std::cout << "Base::foo\n";
    }
};

struct Derived : public Base {
    ~Derived() {}
    void foo() override {
        std::cout << "Derived::foo\n";
    }
};
```

AI写代码cpp运行

销毁对象时，顺序是：

1. **先析构派生类部分**
2. 再析构基类部分

当执行 `Base::~Base()` 时：

- `Derived` 已经被销毁
- 对象动态类型退化为 `Base`
- `vptr` 已经被调整为指向 `Base` 的虚表

因此依然调用的是 `Base::foo()`。

这是一种刻意的设计

很多人第一次知道这一点时，会觉得：

> *“这不是很反直觉吗？”*

但如果反过来想：**如果构造 / 析构期间仍然发生真正的多态调用，反而会极其危险。**

原因很简单：

- 派生类成员可能尚未初始化 / 已被销毁
- 调用派生类虚函数，可能访问未定义状态
- 行为将变得不可控

所以 C++ 选择了一条更保守、也更安全的路线：

> **在构造与析构期间，虚函数调用被限制在当前构造层级。**

当继承不再是一条线：多继承带来的复杂性

在单继承模型下，我们前面讨论的内容都还算“干净”：

- 对象里一个 `vptr`
- 指向一张虚函数表
- 通过 `Base*` 调用虚函数，沿着这张表分发

但一旦进入**多继承**，这个模型就不再成立。


```
1 struct A {
```

```cpp
    virtual void fa();
};

struct B {
    virtual void fb();
    int b;
};

struct C : public A, public B {
    void fa() override;
    void fb() override;
    int c;
};
```

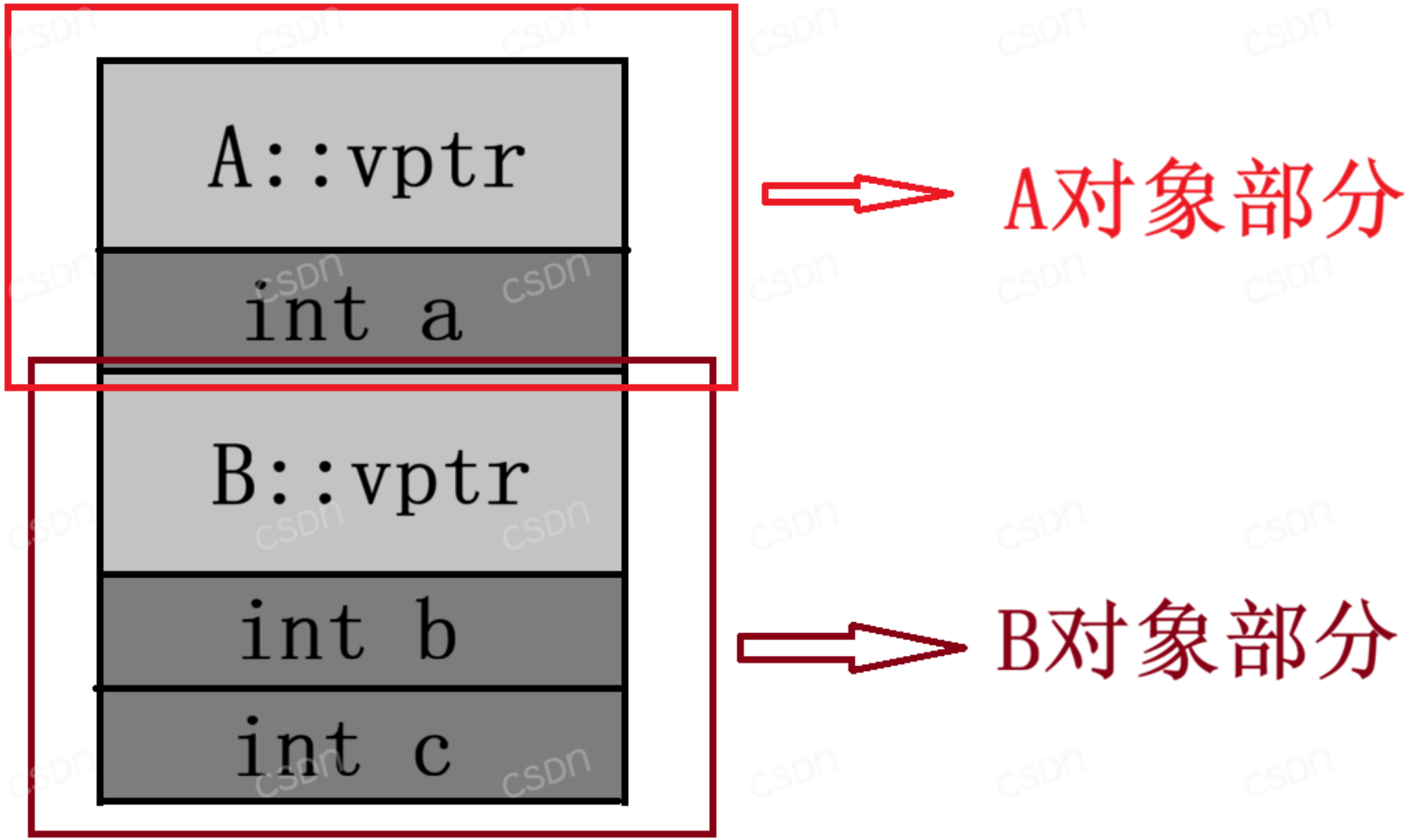
AI写代码cpp运行

直觉上，你可能会觉得 C 只是“把 A 和 B 拼起来”。

事实确实如此，但代价是：对象里不止一个虚表指针。

多继承对象的真实内存布局

在大多数主流编译器中，C 的内存布局大致是：



几个关键事实：

1. **每个带虚函数的基类子对象，都有自己的 vptr**
2. C 并不存在“唯一的一张虚表”

3. 多态是以“基类子对象”为单位发生的

这意味着：

```
C obj;
A* pa = &obj;
B* pb = &obj;
```

AI写代码cpp运行

- pa 指向的是 C 中的 **A 子对象**
- pb 指向的是 **B 子对象**
- 它们的地址甚至可能不同

---

this 指针，已经不再是“对象首地址”

这是多继承中最容易被忽略、但极其重要的一点。

在单继承中：

```
Base* p = &derived;
```

AI写代码cpp运行

p 基本等于对象首地址。

但在多继承中：

```
B* pb = static_cast<B*>(&obj);
```

AI写代码cpp运行

pb 实际上是：**obj的地址 + B所在位置的偏移量。**

也正因为如此：

- 多继承下的虚函数调用，往往伴随着 **this 指针调整**
- 虚表中的函数指针，可能并不是直接指向成员函数
- 而是指向一个 **编译器生成的 thunk（调整函数）**

这也是多继承性能和复杂度显著上升的根本原因。

菱形继承问题的本质

现在来看经典的“菱形继承”：

```cpp
struct Base {
    int x;
};

struct A : public Base {};
struct B : public Base {};
struct C : public A, public B {};
```

AI写代码cpp运行

此时，C 的内存里实际上有：

```
Base (通过 A 继承)
Base (通过 B 继承)
```

AI写代码cpp运行

也就是说：

- C 内部 **有两份** Base
- A::Base::x 和 B::Base::x 完全不是同一个

这会直接导致两个问题：

1. 语义歧义（访问哪个 Base？）
2. 状态不一致（修改了一份，另一份不变）

虚继承：解决“重复基类”的唯一手段

C++ 通过 **虚继承（virtual inheritance）** 来解决这个问题：

```cpp
struct Base {
    int x;
};

struct A : virtual public Base {};
struct B : virtual public Base {};
struct C : public A, public B {};
```

AI写代码cpp运行

虚继承下，对象模型再度升级

虚继承并不是“少复制一份 Base ”这么简单，它引入了新的机制：

- Base 子对象不再固定出现在某个位置
- A 和 B 内部需要保存 **指向虚基类的间接信息**
- 通常通过 **虚基类指针（vbptr）** 或偏移表实现

一个典型布局可能是：

C对象的内存分配

这意味着：

- 访问虚基类成员，需要 **额外一次间接寻址**
- 构造顺序由 **最派生类（C）负责**
- 构造函数参数传递规则更加严格

> **虚继承的唯一核心作用，是在菱形继承结构中，保证虚基类在最终对象中只存在一份。**

最终总结：虚函数与虚继承，各自站在不同的层级

在 C++ 中，**虚函数**和**虚继承**常常一起出现，但它们解决的是**完全不同层次的问题**。

虚函数：为“行为的多态”服务

虚函数的本质目标只有一个：

> **在运行期，根据对象的真实类型，正确地分发行为。**

为此，C++ 引入了：`vptr`，虚函数表（**vtable**），间接调用与运行期绑定

这些机制的代价是：一次额外的间接跳转，对象体积略微增大，构造 / 析构阶段的调用限制

但换来的是：清晰、稳定的多态语义，可扩展的接口设计，面向抽象编程的能力

虚继承：为“唯一状态”兜底的机制

虚继承的目标也只有一个：

> **在菱形继承结构中，保证虚基类只存在一份。**

它并不增强多态能力，而是：消除重复基类，明确共享状态的语义，解决继承结构不可避免的冲突

但其代价明显更高：更复杂的对象布局，额外的间接寻址，构造规则更严格