

# 多态

胡船长

初航我带你，远航靠自己

# 多态基础

1. 基础概念：多态的基本分类
2. 运行时多态：虚函数与纯虚函数
3. 探究虚函数的对象模型
4. 多态的应用：访问者模式

# 多态基础

1. **基础概念：多态的分类**
2. 运行时多态：虚函数与纯虚函数
3. 探究虚函数的对象模型
4. 多态的应用：访问者模式

# 基础概念：多态

在编程语言和类型论中，

**多态**（英语：polymorphism）指  
为不同数据类型的实体提供统一的接口。

# 多态：函数重载

在编程语言和类型论中，

**多态**（英语：polymorphism）指  
为不同数据类型的实体提供统一的接口。

```
int f(int);
```

```
int f(double);
```

```
int f(string);
```

# 多态：运算符重载

在编程语言和类型论中，

**多态**（英语：polymorphism）指  
为不同数据类型的实体提供统一的接口。

```
operator+(int, int)
```

```
operator+(string, int)
```

```
operator+(double, string)
```

# 基础概念：多态的分类

**编译时多态**

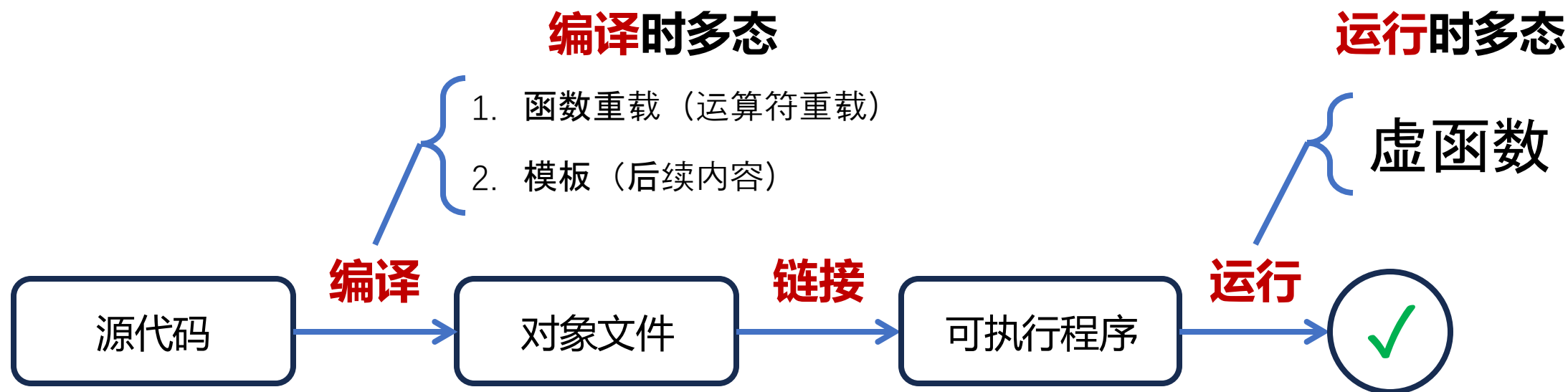
**运行时多态**

# 基础概念：多态的分类





# 基础概念：多态的分类



# 多态基础

1. 基础概念：多态的基本分类
- 2. 运行时多态：虚函数与纯虚函数**
3. 探究虚函数的对象模型
4. 多态的应用：访问者模式

# 如此继承

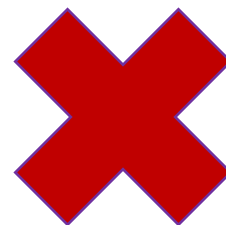
```
class Animal {  
public :  
    Animal(const string &name) : __name(name) {}  
    void run();  
protected :  
    string __name;  
};  
  
class Cat : public Animal {  
public :  
    Cat() : Animal("cat") {}  
    void run() {  
        cout << "I can run with four legs" << endl;  
    }  
};
```

```
Cat a;  
Animal &b = a;  
Animal *c = &a;  
a.run();  
b.run();  
c->run();
```

# 如此继承

```
class Animal {  
public :  
    Animal(const string &name) : __name(name) {}  
    void run();  
protected :  
    string __name;  
};  
  
class Cat : public Animal {  
public :  
    Cat() : Animal("cat") {}  
    void run() {  
        cout << "I can run with four legs" << endl;  
    }  
};
```

有一只猫  
这只猫是个动物  
让这只猫『跑』  
让这个动物『跑』



# 虚函数

```
class Animal {  
public :  
    Animal(const string &name) : __name(name) {}  
    virtual void run() {  
        cout << "I don't know how can run" << endl;  
    };  
protected :  
    string __name;  
};  
  
class Cat : public Animal {  
public :  
    Cat() : Animal("cat") {}  
    void run() override {  
        cout << "I can run with four legs" << endl;  
    }  
};
```

有一只猫  
这只猫是个动物  
让这只猫『跑』  
让这个动物『跑』



# 纯虚函数

```
class Animal {  
public :  
    Animal(const string &name) : __name(name) {}  
    virtual void run() = 0;  
protected :  
    string __name;  
};  
  
class Cat : public Animal {  
public :  
    Cat() : Animal("cat") {}  
    void run() override {  
        cout << "I can run with four legs" << endl;  
    }  
};
```

有一只猫  
这只猫是个动物  
让这只猫『跑』  
让这个动物『跑』



# 抽象类

Animal 即为抽象类，抽象类不生成对象

```
class Animal {  
public :  
    Animal(const string &name) : __name(name) {}  
    virtual void run() = 0;  
protected :  
    string __name;  
};  
  
class Cat : public Animal {  
public :  
    Cat() : Animal("cat") {}  
    void run() override {  
        cout << "I can run with four legs" << endl;  
    }  
};
```

# virtual 关键字

**作用：**将父类的某个方法定义成接口，允许子类自行实现

**成员方法调用时：**

`virtual` 关键字的方法跟着【对象】  
非 `virtual` 关键字的方法跟着【类】

**限制：**

不能用来修饰【类方法-`static`】



# 纯虚函数

**作用：**通过父类，定义那些子类必须实现的方法接口

**应用场景：**定义接口

# 查漏补缺

1. 为什么会有 **override** 关键字?

2. **final** 关键字的作用

3. C++ 中的类型转换

# 查漏补缺

1. 为什么会有 override 关键字?

**2. final 关键字的作用**

3. C++中的类型转换

# 查漏补缺

1. 为什么会有 override 关键字?

2. final 关键字的作用

**3. C++ 中的类型转换**

# C++ 中的类型转换

1. 静态转换: `static_cast`
2. 动态转换: `dynamic_cast`
3. 常量转换: `const_cast`
4. 指针转换: `reinterpret_cast`

# C++ 中的类型转换

1. 静态转换: `static_cast`

1. 动态转换: `dynamic_cast`

2. 常量转换: `const_cast`

3. 指针转换: `reinterpret_cast`

# C++ 中的类型转换

## 1. 静态转换: `static_cast`

1. 动态转换: `dynamic_cast`

2. 常量转换: `const_cast`

3. 指针转换: `reinterpret_cast`

# C++ 中的类型转换

1. 静态转换: `static_cast`

**1. 动态转换: `dynamic_cast`**

2. 常量转换: `const_cast`

3. 指针转换: `reinterpret_cast`



# C++ 中的类型转换

1. 静态转换: `static_cast`

1. 动态转换: `dynamic_cast`

**2. 常量转换: `const_cast`**

3. 指针转换: `reinterpret_cast`

# C++ 中的类型转换

1. 静态转换: `static_cast`

1. 动态转换: `dynamic_cast`

2. 常量转换: `const_cast`

**3. 指针转换: `reinterpret_cast`**

# 多态基础

1. 基础概念：多态的基本分类
2. 运行时多态：虚函数与纯虚函数
- 3. 探究虚函数的对象模型**
4. 多态的应用：访问者模式

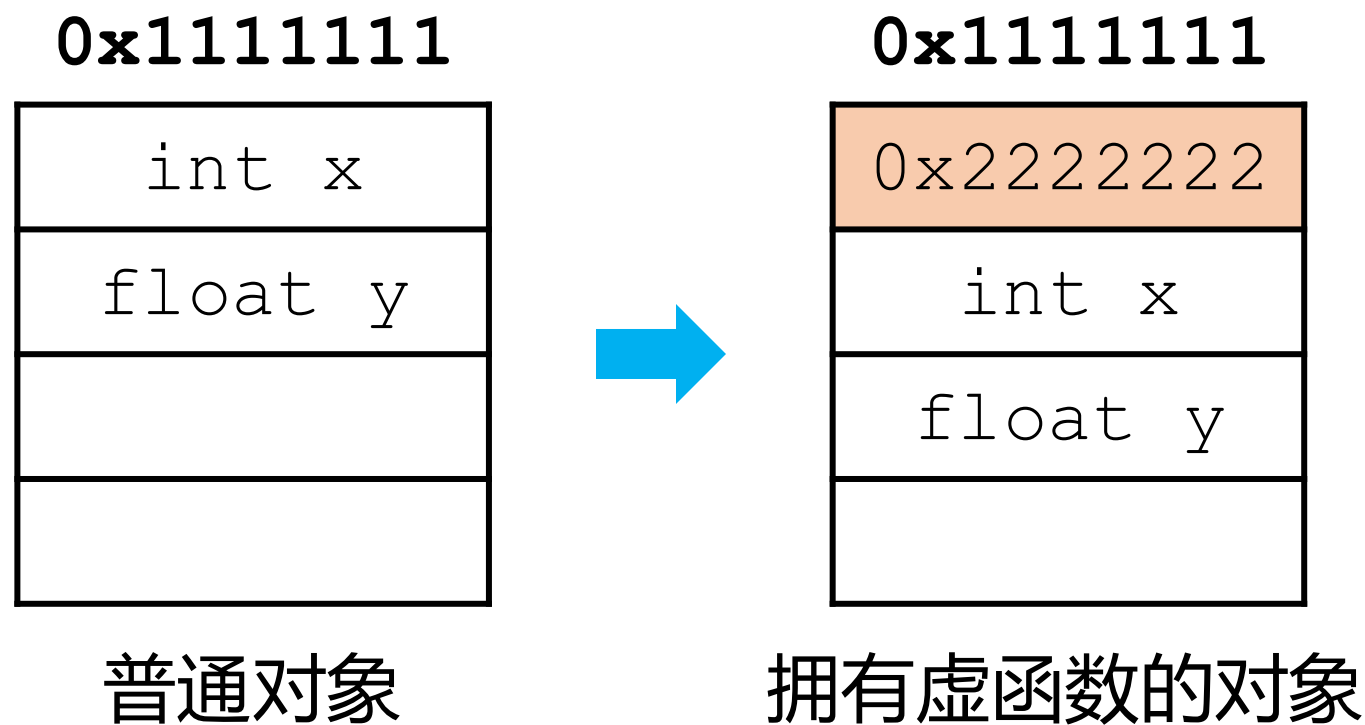
# 虚函数的对象模型

**0x11111111**

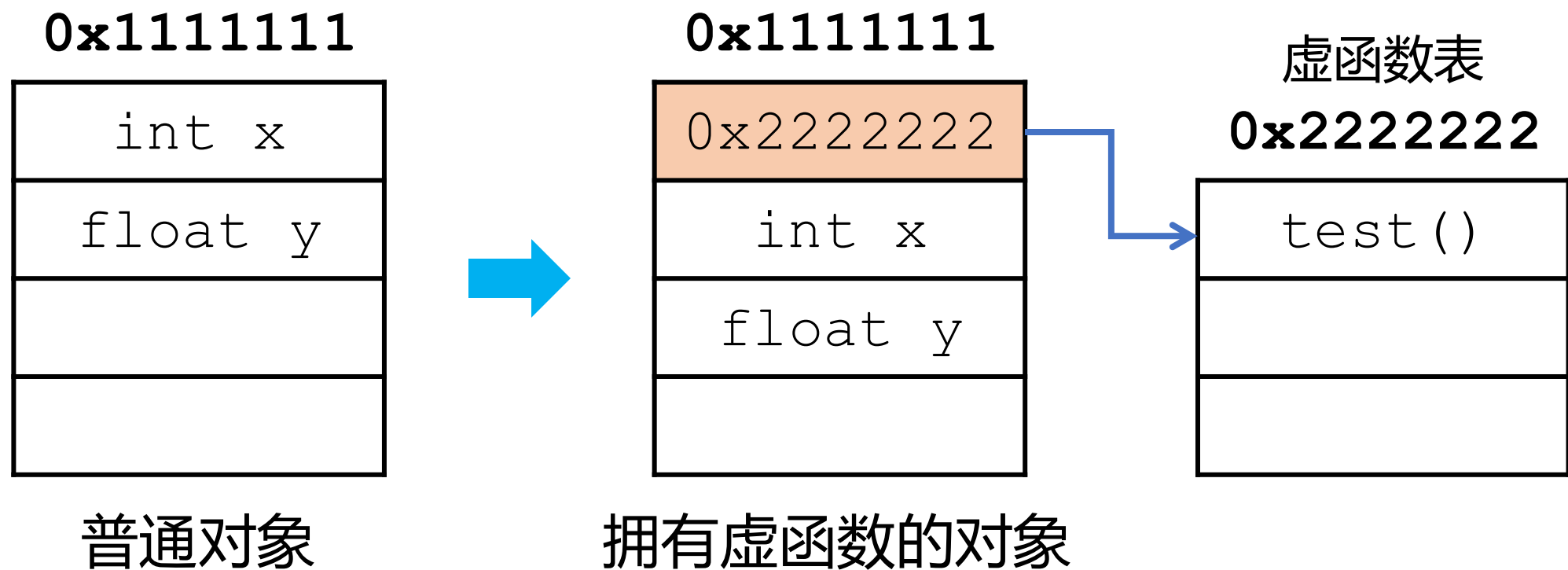
int x
float y

普通对象

# 虚函数的对象模型



# 虚函数的对象模型



# 虚函数的对象模型

Class A: 虚函数表



Class B: 虚函数表



Class C: 虚函数表



# 虚函数的对象模型

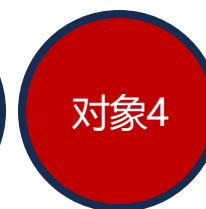
Class A: 虚函数表



Class B: 虚函数表



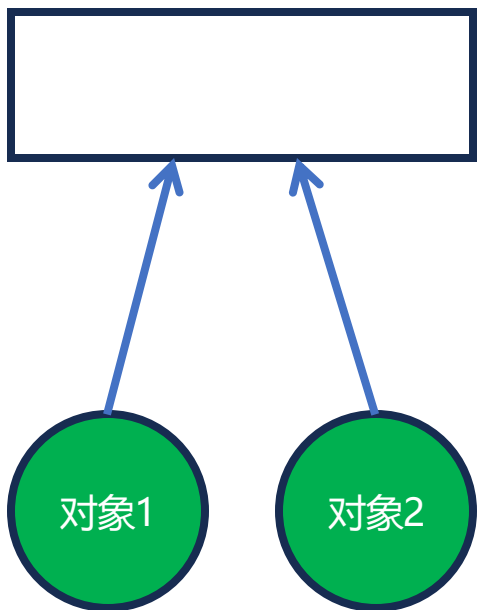
Class C: 虚函数表



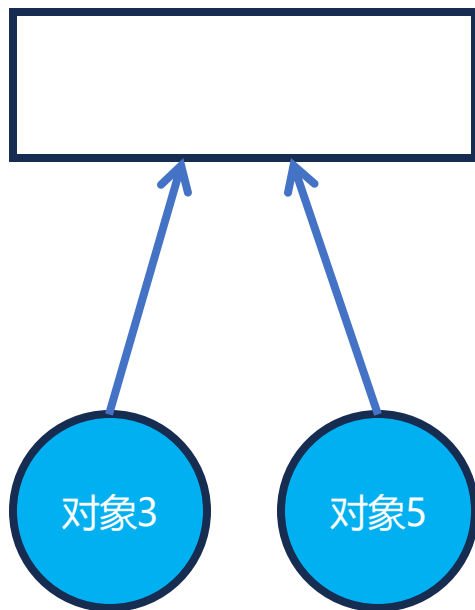


# 虚函数的对象模型

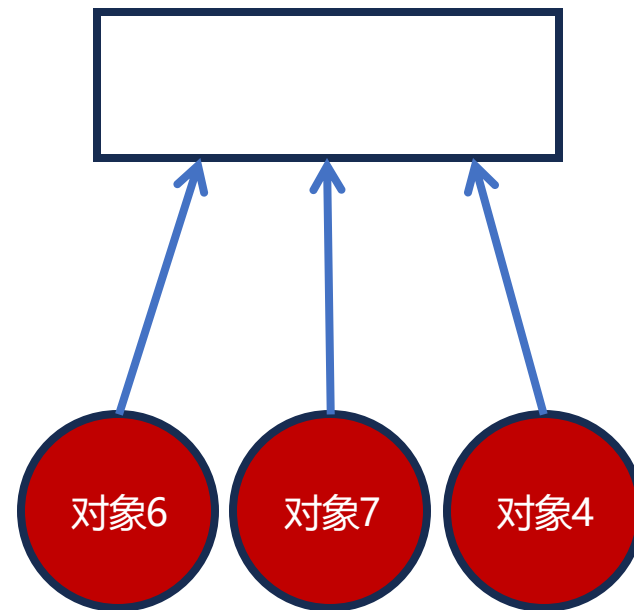
Class A: 虚函数表



Class B: 虚函数表



Class C: 虚函数表



# 多态基础

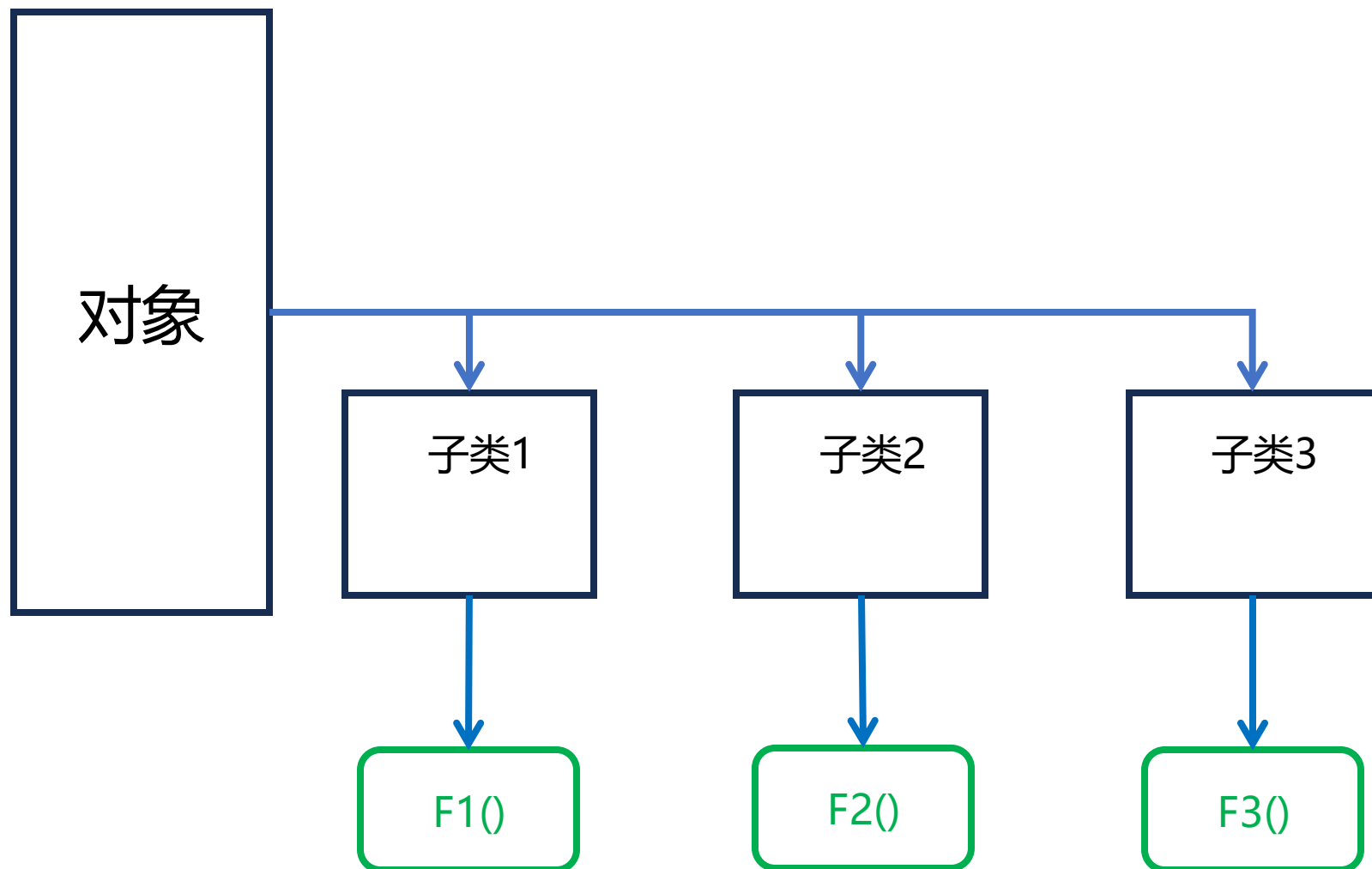
1. 基础概念：多态的基本分类
2. 运行时多态：虚函数与纯虚函数
3. 探究虚函数的对象模型
- 4. 多态的应用：访问者模式**

# 访问者模式

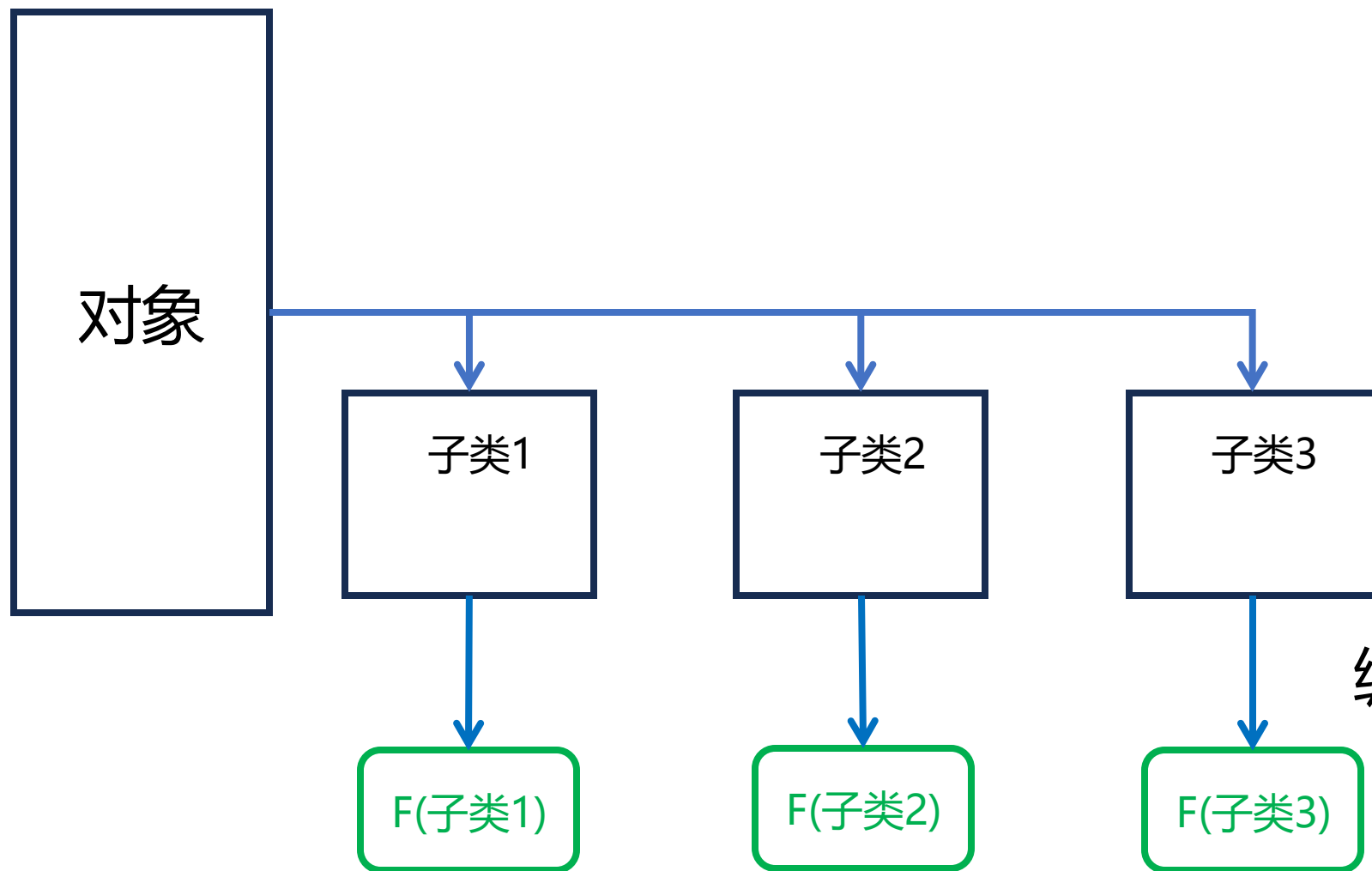
## 使用场景

1. 需要根据派生类的类型，执行不同操作
2. 不希望通过增加成员方法实现此功能
3. 代码维护方便，功能扩展方便

# 访问者模式

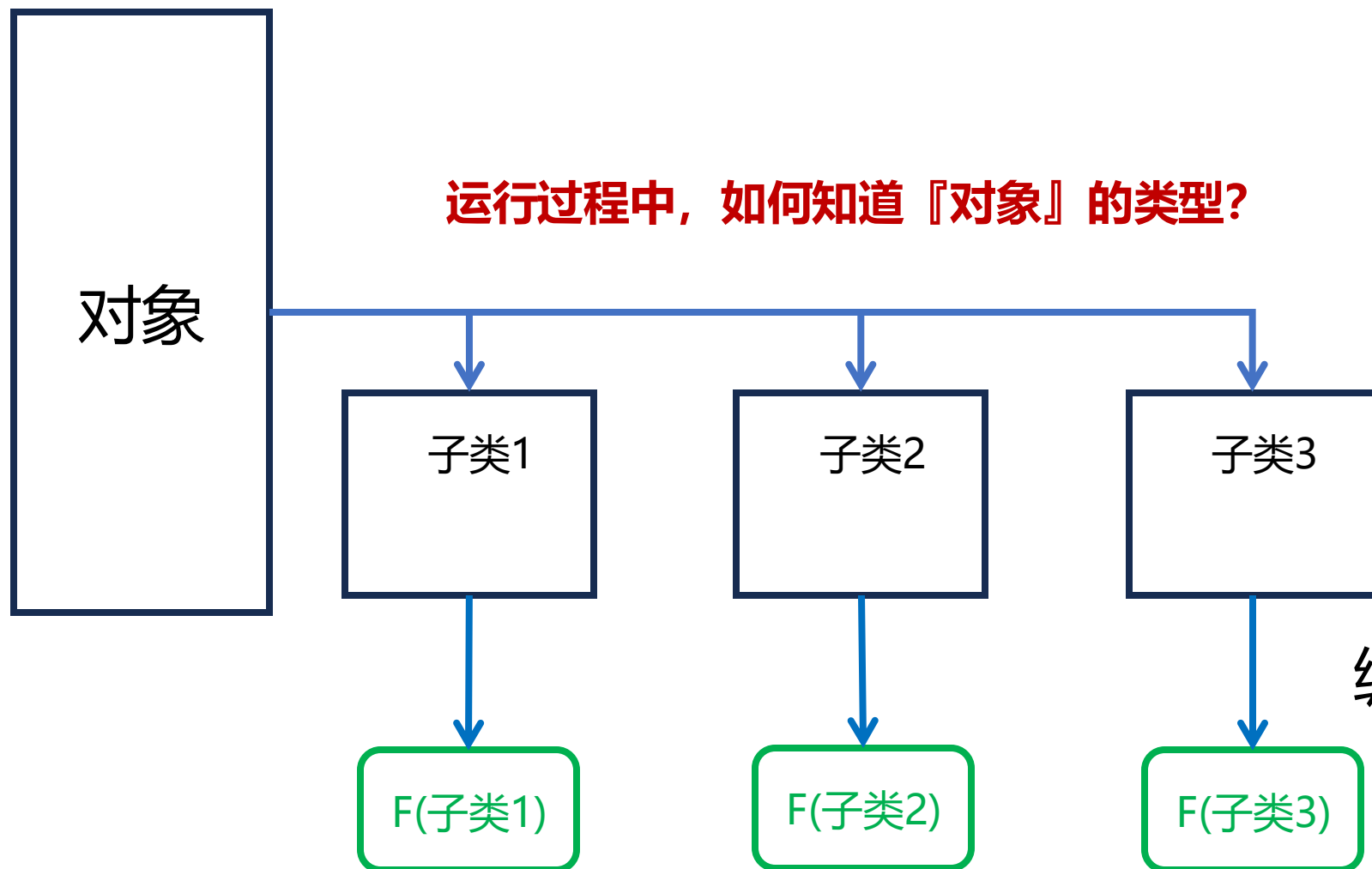


# 访问者模式



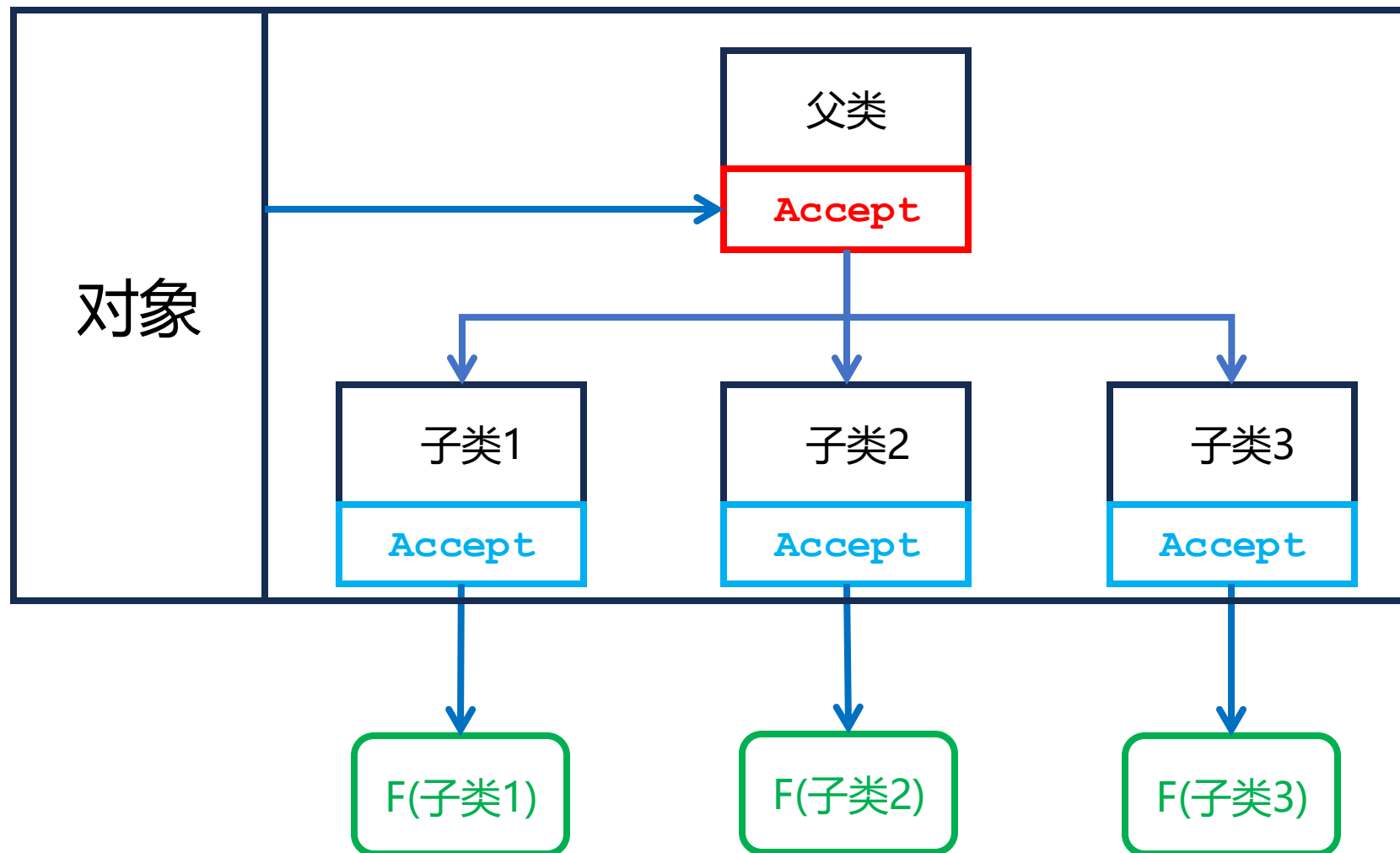
编译时多态：函数重载

# 访问者模式



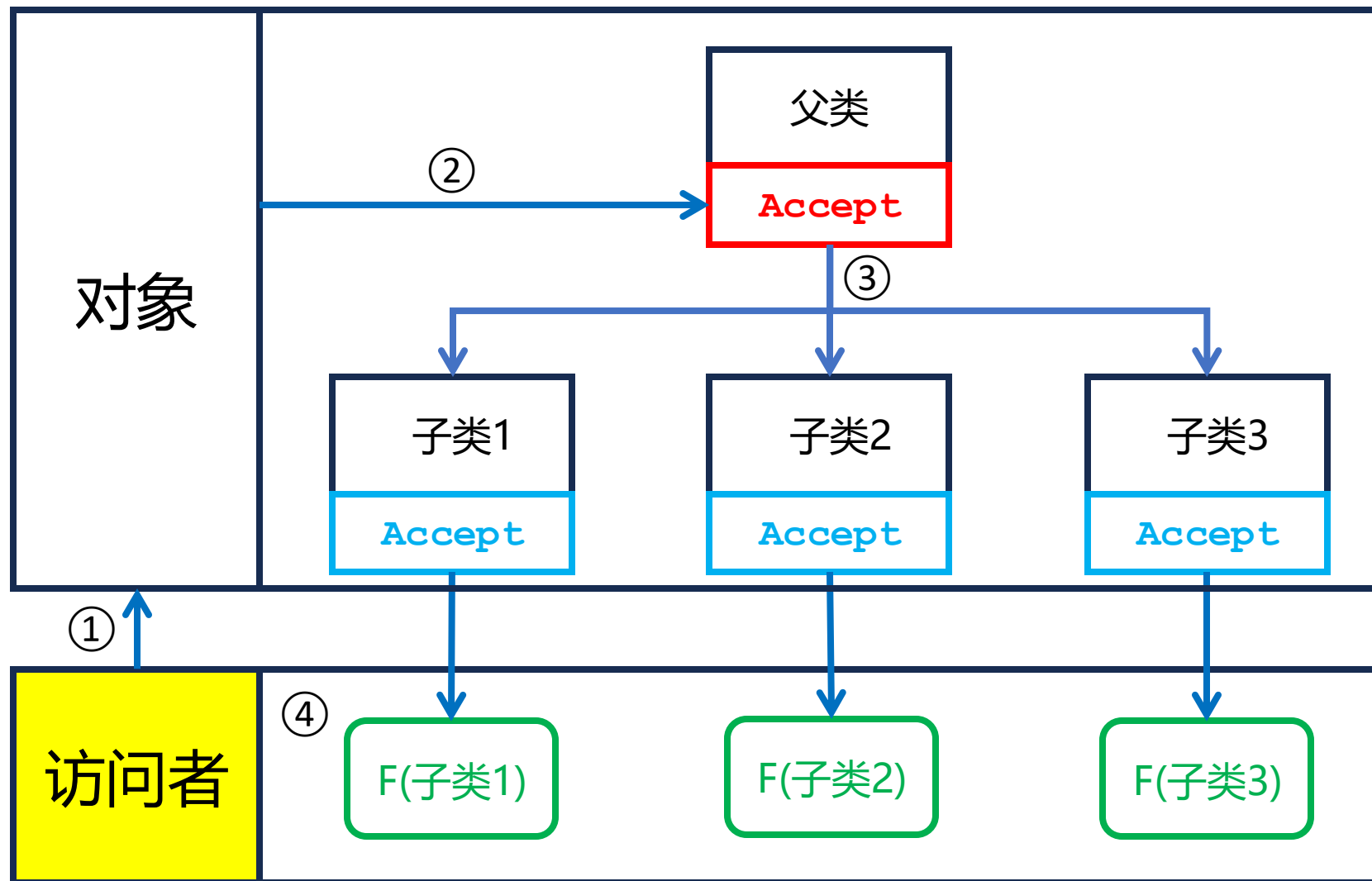
编译时多态：函数重载

# 访问者模式



运行时多态：虚函数

# 访问者模式





# 访问者模式-核心问题

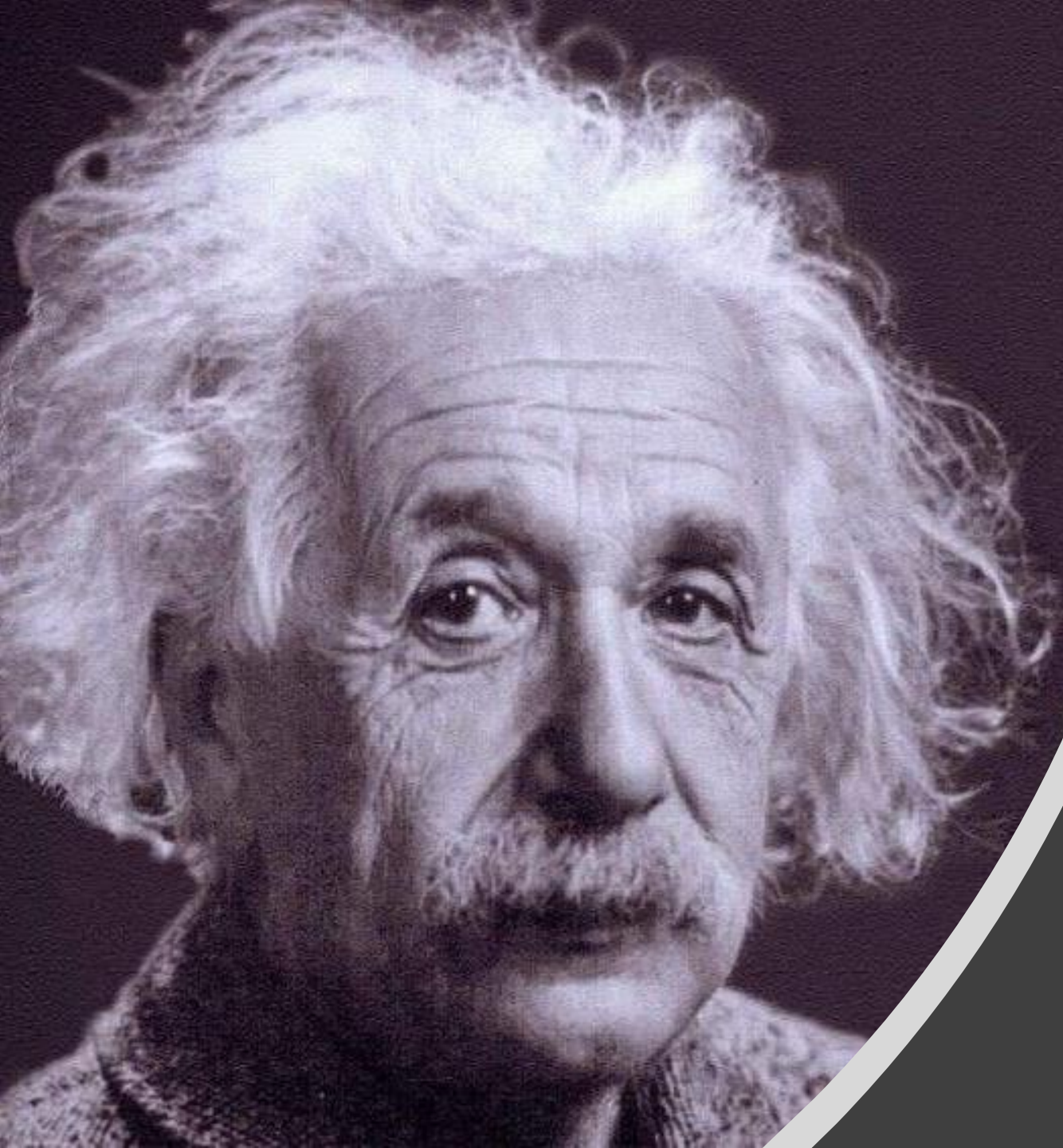
1. 访问者模式使用了哪几类『多态』？
2. 利用『运行时多态』解决什么问题？
3. 利用『编译时多态』解决什么问题？
4. 为什么要封装一个『访问者』类？

# 访问者模式-总结

利用『运行时多态』解析出当前对象的类型

通过 `this` 指针 和 『编译时多态』对应到要执行的功能函数

不同的功能，就对应不同的『访问者』



为什么  
会出一样的题目？